Project 1
Solving the one dimensional Poisson equation

Kamilla Ida Julie Sulebakk, Andrea Jensen Marthinussen
& Hedvig Borgen Reiersrud

September 10, 2020

Abstract

We have investigated three different algorithms in order to solve a one dimensional Poisson equation numerically. The different methods each have their perks. Moreover, they have a wide spread in number of FLOPs from $4N$ for the specialized Gaussian elimination algorithm to $8N$ for the general Gaussian elimination algorithm and $\frac{2}{3}N^2$ for the LU-decomposition algorithm. In addition the CPU-times also differ. It was apparent that the specialized Gaussian elimination algorithm generated the lowest CPU-time. In addition we found that the spacing which generated the least numerical errors were $h = 10^{-6}$ for this algorithm. With all this in mind, we can thereby conclude that the special case algorithm with the spacing $h = 10^{-6}$ is best fitted to solve the differential equation for a fixed $f(x)$.

Introduction

In this project we are solving a one dimensional Poisson equation. The Poisson equation is an ordinary differential equation which is broadly used in theoretical physics as it describes the relation between i.e. mass or charge density and potential. Here we will explore the numerical errors as we compare our numerical solution of the differential equation with an analytical solution. Using Dirichlet boundary conditions we will rewrite the differential equation as a set of linear equations, and we will thereafter use different linear algebra methods to solve it.

Firstly, we will look at how we with a general matrix equation on the form $\mathbf{Av} = \mathbf{g}$, where $\mathbf{A}$ is a tridiagonal matrix, can use Gaussian elimination to solve for $\mathbf{v}$. We will thereby look at the case in which this type of matrix equation actually describes our differential equation, and use an specialized version of the Gaussian elimination method to give us a numerical approximation to the solution of our differential equation. We will then look at the difference in CPU-time for the general and the specialized Gaussian elimination methods and look at how specializing a code intended for a specific case improves the execution. Finally we will compare our results to another method of solving our differential equation with linear algebra, called LU decomposition.

In addition, we will investigate how the numerical errors behave for different spacing values. Decreasing numerical errors is a substantial element in numerical analysis. It goes without saying that in an approximation algorithm, e.g. the numerical integration of the Poisson equation, we want to decrease the numerical errors as much as is viable. Since we iterate calculations a great number of times, the errors can accumulate and provide unreasonable results. It is therefore of great interest to study the spacing providing the least errors in our calculations.

Theory

The one dimensional Poisson equation

The Poisson equation is an ordinary second order differential equation. In one dimension, x, it reads,

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

where $f(x)$ is an arbitrary function. The solution to the differential equation, $u(x)$, can be solved either numerically or analytically as equation 1 can be written as $u(x) = \iint f(x)dx^2$, which is analytically solvable.

We can rewrite equation 1 in a point x_0 with the second order centered difference approximation,

$$-\frac{d^2}{dx^2}u(x_0) = -\lim_{h\to 0}\frac{u(x_0+h) - 2u(x_0) + u(x_0-h)}{h^2} + O(h^2) = f(x_0). \tag{2}$$

Where $O(h^2)$ is the round-off error. By letting h be a non-zero, but small number, we can use equation 2 to approximate the solution (x). In order to solve the differential equation numerically for any function $f(x)$, we need to discretize x and approximate the solution $u(x)$ numerically by $\vec{v}(x) = (v(x_0), v(x_1), ...)$. We can rewrite the approximation as a set of linear equations as following $\mathbf{Av} = \mathbf{g}$ and using Gaussian elimination to find the solution $\mathbf{v}$. The Gaussian elimination is explained further in the method section.

LU-decomposition

We have now seen that we can express a differential equation as a matrix equation on the form $\mathbf{Av} = \mathbf{g}$, and solve it by Gaussian elimination. In the following section we will look at how we can use Lower-Upper (LU) decomposition to solve the same equation.

LU decomposition makes it rather simple to obtain $\mathbf{v}$. We can illustrate this by looking at a 4×4 matrix $\mathbf{A}$. The method is based on the fact that we can rewrite a non-singular matrix $\mathbf{A}$ as the product of two matrices $\mathbf{L}$ and $\mathbf{U}$ as follows;

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \tag{3}$$

In other words, we have that the matrix $\mathbf{A} \in R^{n \times n}$ has a LU factorization if the determinant is different from zero. If the LU factorization exists, then the LU factorization is unique. By construction the determinant of $\mathbf{L}$ is equal to 1 (since the diagonal elements are 1), and the determinant of $\mathbf{A}$ is given by;

$$\det(\mathbf{A}) = \det(\mathbf{LU}) = \det(\mathbf{L})\det(\mathbf{U}) = u_{11}u_{22}\dots u_{nm}$$

It is now easy to see that our linear equation can be written on the form

$$\mathbf{Av} = \mathbf{LUv} = \mathbf{g} \tag{4}$$

We can calculate the extended equation system in two steps; first by taking the matrix-vector product between the upper matrix $\mathbf{U}$ and the vector

v; and second by plugging in the new obtained vector, in which we name **y**, for **Uv** in equation 4. For our four dimensional example this will take the following form;

$$\mathbf{Uv} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} v_1u_{11} + v_2u_{12} + v_3u_{13} + v_4u_{14} \\ v_2u_{22} + v_3u_{23} + v_4u_{24} \\ v_3u_{33} + v_4u_{34} \\ v_4u_{44} \end{bmatrix} = \mathbf{y} \tag{5}$$

$$\mathbf{Ly} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} v_1u_{11} + v_2u_{12} + v_3u_{13} + v_4u_{14} \\ v_2u_{22} + v_3u_{23} + v_4u_{24} \\ v_3u_{33} + v_4u_{34} \\ v_4u_{44} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} = \mathbf{g} \tag{6}$$

Since the determinant of **L** is non-zero, we can use the inverse of **L** to obtain the unknown vector **y**;

$$\mathbf{Uv} = \mathbf{L}^{-1}\mathbf{g} = \mathbf{y}.$$

In both steps we are handling triangular matrices, which can easily be solved by forward and backward substitution.

Floating point operations

Floating point operations, also called FLOPs, is a measurement of efficiency as more FLOPs lead to a higher degree of workload. FLOPs are defined as the number of floating-point operations and can be found by counting the operations (multiplying, dividing, subtracting or adding) of a number, usually in a for-loop. Running a loop N times with c FLOPs per iteration, would mean the total number of FLOPs are $N \cdot c$.

In the LU decomposition algorithm in which we will use to calculate the solution of the Poisson equation, the total number of FLOPs is known. For N iterations the number of FLOPs is $\frac{2}{3}N^3$, in reference to [**LU**].

CPU-time and memory

CPU-time is defined as the elapsed time a central processing unit (CPU) uses to process a program in seconds. If the CPU-time is high, the user experience lagging when executing the program. Hence, we want to decrease the CPU-time as much as possible in addition to not decreasing the precision immensely.

The CPU processes fill in the computer memory. If the processes are substantial, we may experience a crashed system as it uses all the computer memory, which we need to avoid.

Numerical errors

From equation 2, we see that the smaller value of h, the step size, lead to a higher degree of accuracy of the approximation of the double derivative, since $O(h^2)$ decreases. This is called the round off error. However, subtraction of nearly equal operands may cause extreme loss of accuracy. This is scarily called catastrophic cancellation. In reference to equation 2 $u(x_0)$ gets very similar to $u(x_0 + h)$. This implies that smaller values of h lead to a larger degree of catastrophic cancellation. It is of interest to find the spacing that provides the best balance between the error sources, and produces the least numerical errors.

We can test the numerical errors by comparing the exact solution to the solution provided by the special case algorithm. The relative error is,

$$\epsilon_i = \log_{10}(|\ \frac{v_i - u_i}{u_i}\ |). \tag{7}$$

Method

We are now developing an algorithm to solve the one dimensional Poisson equation. We let $x \in (0, 1)$ and use the Dirichlet boundary conditions where $u(0) = u(1) = 0$. We discretize our function with n points such that $x = ih$ where $i \in (0, n)$ and h is the spacing $= \frac{1}{n-1}$ and $u(x) \approx \vec{v} = (v_1, v_2, ..., v_n)$ where n is the number of points. Using the Dirichlet boundary conditions let us rewrite the differential equation as a set of linear equations which we can solve numerically.

We set the right hand side of the Poisson equation to be $f(x) = 100e^{-10x}$. By plugging in the boundary conditions, the exact solution will be $u(x) = \int\int_0^1 f(x)dx^2 = 1 - (1 - e^{-10})x - e^{10x}$, which is the analytical solution to the Poisson equation which we will use in comparison with our numerically calculated results.

Our numerical approximations to the analytical solution $u(x)$ are found by the linear algebra methods we introduced in the theory section, and in our code we find their associated CPU times, as well as look at their accuracy in relation to the exact solution $u(x)$ by plotting our results next to the analytical solution.

Deriving the matrix equation from the second order centered difference approximation

By multiplying both sides of equation 2 with h^2 we get,

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i = g_i. \tag{8}$$

In addition, the boundary conditions tell us that $v_1 = 0$ and $v_n = 0$. Since the first element of the vector $\vec{v}$ is v_1 we get that $g_1 = 2v_1 - v_2$ by using equation 8. For the next iterations $g_2 = -v_1 + 2v_2 - v_3$, $g_3 = v_2 + 2v_3 - v_4$, ..., $g_{n-1} = -v_{n-2} + 2v_{n-1} - v_n$. For the last iteration $g_n = -v_{n-1} - 2v_n$ since there are no elements of v_i after n. Using these sets of equations, we can rewrite equation 2 to,

$$\begin{bmatrix} 2 & -1 & 0 & ... & & & 0 \\ -1 & 2 & -1 & 0 & ... & & \\ 0 & -1 & 2 & -1 & 0 & ... & \\ . & 0 & . & . & . & . & . \\ . & . & . & . & . & . & 0 \\ . & . & . & . & . & . & -1 \\ 0 & ... & & & 0 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ . \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ . \\ . \\ . \\ . \\ g_n \end{bmatrix}. \tag{9}$$

Gaussian elimination

In the general case in which we would like to solve an equation on the following form $\mathbf{Av} = \mathbf{g}$, we can write out the extended matrix for the equation and use Gaussian elimination to solve it. With $\mathbf{v}, \mathbf{g}$ being general vectors with n elements, and A as an $n \times n$ tridiagonal matrix with entries $b_0, b_1, ...b_{n-1}$

as the diagonal elements, $a_0, a_1, ..., a_{n-2}$ as the elements below the diagonal and $c_0, c_1, ..., c_{n-2}$ as the elements above the diagonal, the extended matrix looks as follows,

$$\begin{pmatrix} b_0 & c_0 & 0 & ... & & & 0 & g_1 \\ a_0 & b_1 & c_1 & 0 & ... & & & g_2 \\ 0 & a_1 & b_2 & c_2 & 0 & ... & & . \\ . & 0 & . & . & . & . & . & . \\ . & . & . & . & . & . & 0 & \\ . & . & . & . & . & . & c_{n-2} & \\ 0 & ... & & & 0 & a_{n-2} & b_{n-1} & g_{n-1} \end{pmatrix}. \tag{10}$$

To solve for $\mathbf{v}$ we will start by using forward substitution to zero out the elements $a_0, a_1, ..., a_{n-2}$. Naming the matrix rows $i = 0, 1, ..., n-1$, each row i is subtracted by the line $i-1$ times $a_{i-1}/\tilde{b}_{i-1}$. Here, $\tilde{b}_{i-1}$ is the updated b_{i-1} after the line $i-1$ has been subtracted by the line $i-2$, times $a_{i-2}/\tilde{b}_{i-2}$. We have $\tilde{b}_0 = b_0$, as the first line isn't altered using forward substitution. By updating each line in this way, we are left with the following expressions for the diagonal elements of $\mathbf{A}$ and the far right column of the new matrix,

$$\begin{aligned} \tilde{b}_i &= b_i - \frac{c_{i-1}a_{i-1}}{\tilde{b}_{i-1}}, \\ \tilde{g}_i &= g_i - \frac{\tilde{g}_{i-1}a_{i-1}}{\tilde{b}_{i-1}}. \end{aligned}$$

Moving on, after updating each line using forward substitution and getting rid of all non-zero elements below the diagonal elements $\tilde{b}_i$, we apply the backward substitution method to zero out all elements above the diagonal. We start by dividing the last line $n-1$ by $\tilde{b}_{n-1}$, such that

$$v_{n-1} = \frac{\tilde{g}_{n-1}}{\tilde{b}_{n-1}},$$

where v_i is the updated element in the far right column on line i after applying backward substitution. From each line i we subtract the line $i+1$ times c_i, followed by dividing the line i by $\tilde{b}_i$ such that the diagonal elements are all updated to be 1. From this we can write the expression for the right hand side-elements v_i as

$$v_i = \frac{\tilde{g}_i - v_{i+1}c_i}{\tilde{b}_i},$$

which gives us the solution $\mathbf{v}$ to our matrix equation.

In the general case algorithm we arrive at 3 FLOPs per iteration for $\tilde{b_i}$, $\tilde{g_i}$ and v_i, so that the total number of FLOPs for N iterations ends up with being $3N + 3N + 3N = 9N$.

Gaussian elimination for the special matrix equation

For the specific situation in which we want to solve equation 1, we know from our expression 2 that the matrix elements $a_0, a_1, ..., a_{n-2} = -1$, $b_0, b_1, ..., b_{n-1} = 2$ and $c_0, c_1, ..., c_{n-2} = -1$. This allows us to simplify and specialize our algorithm to maximize the performance of our code. Now, our right hand side-elements $\mathbf{v} = v_0, v_1, ...v_{n-1}$ which we solve for is our discretized approximation to $u(x)$, and $\mathbf{g} = g_0, g_1, ..., g_{n-1}$ is our known function $f(x)$ evaluated at the steps $x_0, x_1, ..., x_{n-1}$ times the step size h to the second power such that each element $g_i = h^2 f(x_i)$. Our extended matrix is now

$$\begin{pmatrix} 2 & -1 & 0 & ... & & & 0 & g_1 \\ -1 & 2 & -1 & 0 & ... & & & g_2 \\ 0 & -1 & 2 & -1 & 0 & ... & & . \\ . & 0 & . & . & . & . & . & . \\ . & . & . & . & . & . & 0 & \\ . & . & . & . & . & . & -1 & \\ 0 & ... & & & 0 & -1 & 2 & g_{n-1} \end{pmatrix}. \tag{11}$$

Like in the general case, we start out by applying the forward substitution method. We quickly see that if we add the first line times $1/2$ to the second line, a_0 is zeroed out. Our updated diagonal element on the second line is now $\tilde{b}_1 = 2 - 1/2 = 3/2$, with the updated right hand side-element $\tilde{g}_1 = g_1 + \tilde{g}_0/2$. As in the general case, since the first line isn't altered by forward substitution, $\tilde{b}_0 = b_0$, $\tilde{g}_0 = g_0$. We can further see that in order for a_1 to be zeroed out, we must add the second line times $2/3$ to the third line, thus giving us $\tilde{b}_2 = 2 - 2/3 = 4/3$, $\tilde{g}_2 = g_2 + 2g_1/3$. Continuing this we discover the pattern

$$\tilde{b}_i = \frac{i+2}{i+1},$$
$$\tilde{g}_i = g_i + \tilde{g}_{i-1}\frac{i}{i+1}.$$

After forward substituting all lines in the extended matrix, we divide the last line by $\tilde{b}_{n-1}$ as in the general case, so that $v_{n-1} = \tilde{g}_{n-1}/\tilde{b}_{n-1}$. Following our general backward substitution method, inserting $c_i = 2$ for all i's, we find an expression for each right hand side-element v_i,

$$v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}. \tag{12}$$

Per iteration of i the number of FLOPs is 2. Using our expression for $\tilde{b}_i$ from above, we arrive at the final expression for our solution v_i,

$$v_i = (\tilde{g}_i + v_{i+1})\frac{i+1}{i+2}. \tag{13}$$

In the algorithm in equation 13 the number of FLOPs per iteration of i is 2. This is because multiplying by $\frac{i+1}{i+2}$ only counts as one floating point operator, as it can be calculated beforehand. The total number of FLOPs in the special case algorithm is therefore $2N + 2N = 4N$.

Thus, from equation 13, we have an approximation $\mathbf{v}$ to the solution of the Poisson equation, which we then plot together with the exact solution $u(x)$.

LU decomposition

In our code we have implemented the two algorithms of Gaussian elimination, the general one as well as the specialized one, and we further want to compare our resulting approximation to the solution of the differential equation from the specialized Gaussian elimination method with another linear algebra method. We thereby implement a method called LU decomposition as described in the theory section.

Our algorithm for this method is based on LU decomposing the two-dimensional matrix $\mathbf{A}$, of dimension n, through a call to the function «ludcmp(double A, int n, int indx, double d)». This function takes $\mathbf{A}$ as input and replaces it by the LU decomposition. The vector index records the

number of interchanges of rows, and d keep in track whether the number of row interchanges is even or odd. Even row interchanges means that the determinant of the LU decomposed matrix equals +1. Likewise, odd row interchanges, indicates that the determinant equals -1.

The LU decomposed matrix is returned in the same place as $\mathbf{A}$, and by calling the function «ludcmp(double A, int n, int indx, double w)» we can easily obtain $\mathbf{v}$. This function solves the linear equation on the form $\mathbf{Av} = \mathbf{g}$. In addition taking matrix $\mathbf{A}$ from the LU decomposition, the function takes the right hand side, $\mathbf{g}$, of our linear equation as input. The solution $\mathbf{v}$ is then returned in $\mathbf{g}$.

Numerical errors

From equation 7 we can find the highest number of value errors by running through each value of ϵ for a given h and finding its highest value. We choose to look at ϵ for the Gauss elimination in the special case. The algorithm is as following.

```
epsilon_max = 0
for i=0 to N:
    epsilon = log10(absolute_value((y[i]-u[i])/u[i]))
    if epsilon > epsilon_max:
        epsilon_max = epsilon
```

We run the algorithm for $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$ and finally 10^7. Assuming the degree of error due to catastrophic cancellation dominate the total error ϵ_i when n increases (and h decreases) we expect ϵ_i to drop when n increases to a certain point h_1 and decreasing after h_1.

Since ϵ, for values where the catastrophic cancellation do not dominate, should be linear to $O(h^2)$, we can plot the max value of ϵ with $\log_{10}(h)$. We expect the plot to appear linear for higher h values. However for values lower than h_1, we do now expect a continuum of the linearity.

Results

In our code, we ran though the two versions of Gaussian elimination for the numbers of steps $n = 10, 10^2, 10^3, 10^4$ and 10^5, and made plots of our approximated solutions to the Poisson equation versus the exact solution $U(x)$. The plots for the general Gaussian elimination algorithm can be seen in figure 1,

and the plots for the specialized version of the method can be seen in figure 2.

We further plotted the approximated solution to the differential equation found by LU decomposition for the numbers of steps $n = 10, 10^2$ and 10^3, which can be seen in figure 3. The CPU-times for the different algorithms is shown in table 1.

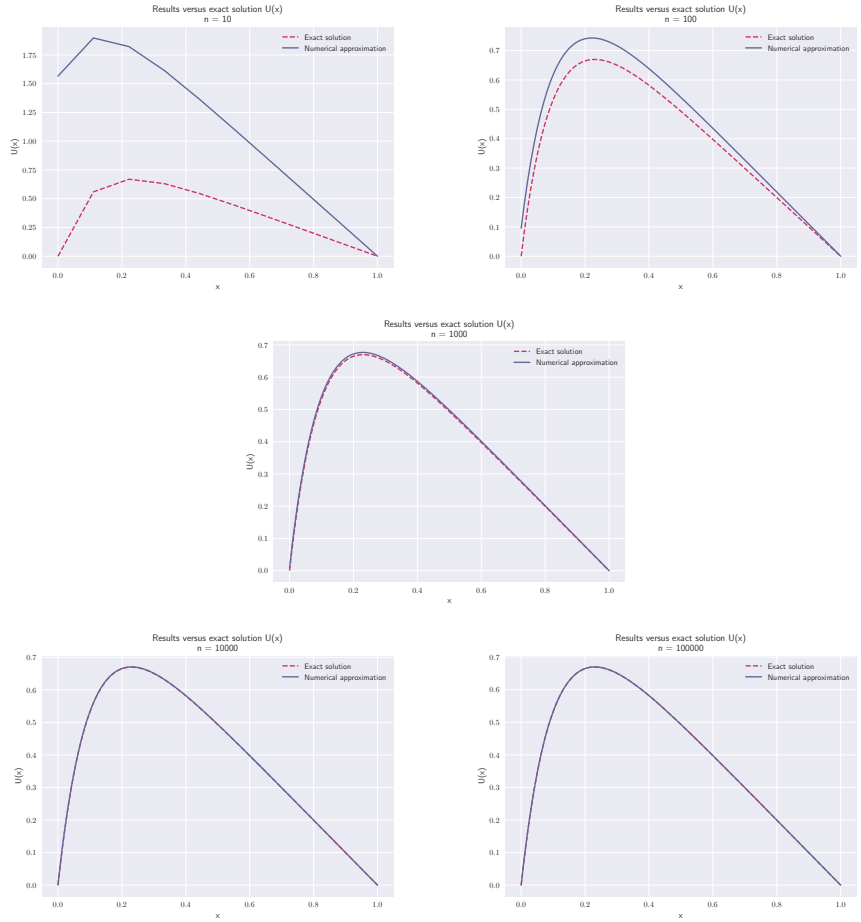

Figure 1: The numerical approximations found by the general Gaussian elimination algorithm plotted against the exact solution $U(x)$ for different numbers of steps n.

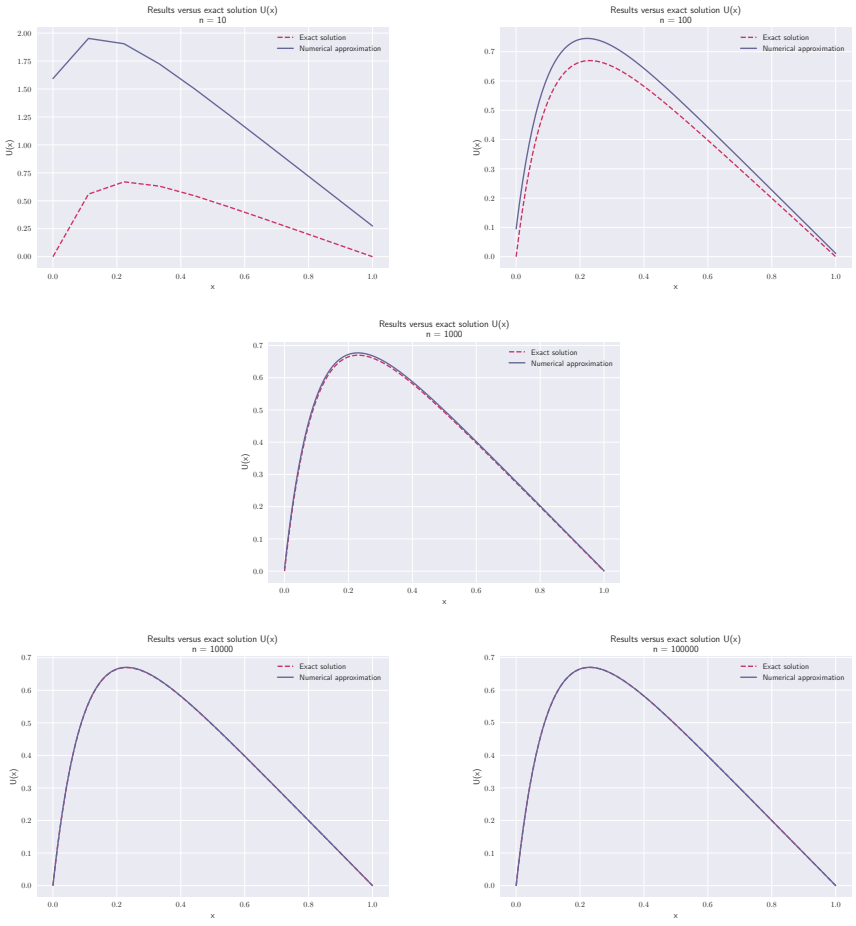

Figure 2: The numerical approximations found by the specialized version of the Gaussian elimination algorithm plotted against the exact solution $U(x)$ for different numbers of steps n.

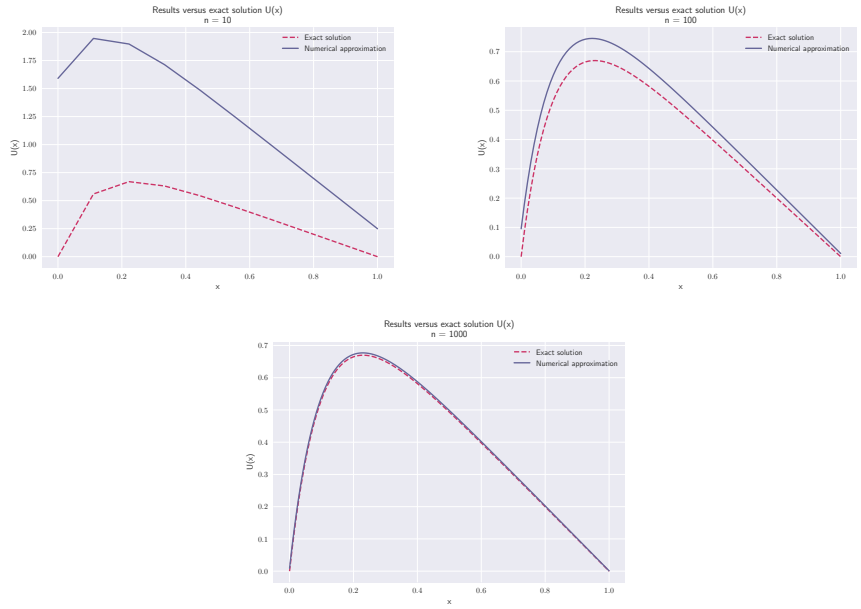

Figure 3: The numerical approximations found by LU decomposition plotted against the exact solution $U(x)$ for different numbers of steps n.

We see that as we increase the number of steps n, the approximations generated by each of the three algorithms become more alike the exact solution to the Poisson equation $u(x)$, although, for the Gaussian elimination algorithms the difference is nearly unnoticable graphically from $n = 10^4$ to $n = 10^5$. Moreover, we note that the CPU-times measured for all values of n with the LU decomposition algorithm is greater than for both of the Gaussian elimination algorithms, and furthermore that the specialized algorithm in which we apply Gaussian elimination is typically faster than the general algorithm.

Table 1: Table of the CPU-times in seconds measured for different algorithms given the numbers of steps n.

n	Gaussian elimination, general algorithm [s]	Gaussian elimination, specialized algorithm [s]	LU decomposition [s]
10	$3.0{\cdot}10^{-6}$	$3.0{\cdot}10^{-6}$	$1.6{\cdot}10^{-5}$
10^2	$1.0{\cdot}10^{-5}$	$5.0{\cdot}10^{-6}$	$2.0{\cdot}10^{-3}$
10^3	$4.7{\cdot}10^{-5}$	$1.9{\cdot}10^{-5}$	1.0
10^4	$5.4{\cdot}10^{-4}$	$2.1{\cdot}10^{-4}$	-
10^5	$3.5{\cdot}10^{-3}$	$2.7{\cdot}10^{-3}$	-

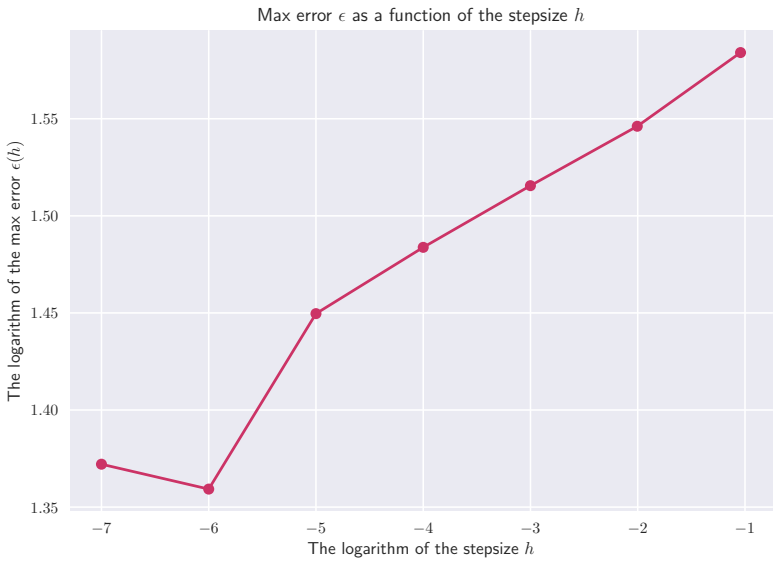

Figure 4: The figure shows the base then logarithm of the maximum value of ϵ_i for different values of the base ten logarithm of h for the special case algorithm.

In figure 4 it is apparent that the max value of ϵ for the special case algorithm for $h \approx 10^{-7}$ is larger than for $h \approx 10^{-6}$. In addition we see that the max value of ϵ_i seems almost linear from $\log_{10}(h) \approx -6$ to -1.

From the results we see that the solution obtained from the LU decomposition largely matches the exact solution for $N = 10^3$. However, we can

not run the standard LU decomposition for lager values of N.

Discussion

The Gaussian elimination algorithms and their results

As we could see from figures 1 and 2, the level of accuracy of the approximations generated by the Gaussian elimination algorithms is elevated as we increased the number of steps, which of course is coinciding with our expectations.

We did also note from table 1 that the specialized algorithm had significantly lower CPU-times than the general algorithm for all numbers of steps except for $n = 10$. This was to be anticipated as this specialized version of the Gaussian elimination method involves less FLOPs and consequently runs faster, as the specialized algorithm involves approximately $4n$ floating point operations, while the general algorithm involves approximately $9n$, where n is the number of steps.

From this we can draw the conclusion that for the specific task, in which we want to solve a specified equation, a specialized algorithm both compute the results faster and moreover provides results of a higher level of precision, both of which is due to the lower number of FLOPs involved in the calculations. The latter claim is that a lower number of FLOPs reduce the degree of catastrophic cancellations in the algorithm, thus granting results of higher accuracy.

The LU decomposition algorithm and its results

As for the the Gaussian elimination algorithms, the level of precision is higher for the LU algorithm as the number of grid points n increases, as presented in figure 3. This result aligned with our expectations.

LU decomposition is highly suitable, in cases, where we want to solve a linear equation multiple times for different right hand sides $\mathbf{g}$. Rather than using Gaussian elimination each time, we only need to do an LU decomposition of the matrix once, before solving the triangular system for different right hand sides. However, since we only have been working with one specific $\mathbf{g}$, we did not benefit from this quality. Instead it is safe to say, that the LU

decomposition by construction, increased the algorithm's CPU time.

The LU decomposition algorithm has a significantly larger CPU time than the Gaussian elimination algorithms (as presented in table 1), which is due to the increased number of floating point operations. For a quadratic matrices of size n, the method has approximately $\frac{2}{3}n^3$ FLOPs, where as the specialized Gaussian algorithm only has $4n$. This clearly affected the usability of the method, in such a way, that the algorithm could not take larger grid points than $n = 3$. Since all elements in our matrix $\mathbf{A}$ was declared as doubles, each element used eight bytes of the memory. In order to handle a 5×5 matrix, the computer would have to allocate $8 \cdot 10^5 \cdot 10^5 = 80$GB, which is larger than the memory of most computer laptops.

Nevertheless, it is remarkable that the LU algorithm could not handle a quadratic matrix of dimension 4. This might have been caused by memory leakage, which could have been detected by using a memory error detector, such as Valgrind Memcheck. Despite the fact that the LU decomposition apparently had a great level of precision for $n = 3$, it would not be wrong to say that the method proved itself to be a disadvantage. When implementing the method, we had wished for results to take in comparison with the results from the Gaussian elimination, which unfortunately was not the case.

Numerical errors

As expected, we see in figure 4 that the numerical errors do not necessarily decrease with h, due to the domination of errors as a consequence of subtraction of nearly equal operands. The maximum error at $n = 10^{-7}$ is not significantly increased from $n = 10^{-6}$, however the heightening is apparent.

The linearity of $\epsilon(\log_{10}(h))$ from $\log_{10}(h) \approx -5$ to -1 in figure 4 is as expected. However, the sudden deviation of $\epsilon(\log_{10}(h))$ at $h \approx 10^{-6}$ is unforeseen, since it is lower than the continuing line interpolating the points for the higher h values. This is most likely explained by an unknown index error in our algorithms. However, it could also be due to an unpredicted decreasing error that happens to appear at the specific h value, needless to say this seems highly unlikely.

Conclusion

In this project we have solved a one dimensional Poisson equation by utilizing the linear algebraic methods of Gaussian elimination and LU decomposition. In our code we have developed both a general algorithm employing Gaussian elimination for solving a set of linear equations as well as written a specialized algorithm, also utilizing Gaussian elimination, suited to solve the Poisson equation numerically. We have looked at the variations in CPU-time for the different algorithms, as well as the graphical deviations from the exact solution of the Poisson equation. We conclude with the fact that a specialized code maximize the execution of a specific task, in this case providing more accurate results as well as computing them faster.

On that note, more specifically we have discussed how the number of floating point operations (FLOPs) affect an algorithm's CPU-time, and amongst other we have seen the effects by comparing the LU decomposition algorithm's CPU-time by those of the Gaussian elimination algorithms. The larger CPU-time of the LU decomposition algorithm in comparison to the other two algorithms is as we have discussed due to the higher number of FLOPs involved.

We have further seen that the numerical errors do not strictly decrease by reducing the step size h. As we discovered a deviation in the graphical presentation of the logarithm of the maximum error of the solution, provided by the specialized Gaussian elimination algorithm, we suspect that this algorithm might be suffering from an indexing error.

All in all, after discussing the advantages and disadvantages of the three numerical algorithms investigated, it is apparent that the specialized Gaussian elimination algorithm produced the pre-eminent results.

References

[**LU**]: *LU decomposition*, Wikipedia, 2020,
cite: **https://en.wikipedia.org/wiki/LU_decomposition**