Universitetet i Oslo

FYS4150

Project 1

Jessie Warraich
Christina Knudsen

September 10, 2019

Contents

1 Abstract

We have in this article applied different methods for solving linear equations. First we created a general algorithm for solving an equation with a tridiagonal matrix, then we specialized it to apply to Poisson's equation, and then we used a LU-decomposition to solve the equation. We found that the specialized algorithm spent 78 percent of the time the general method spent at $n = 10^4$, and was about ten thousand times faster than the algorithm for the LU-decomposition. The accuracy for different n-values increased from $n = 10^1$ to $n = 10^5$ but then it decreased, due to loss of numerical precision.

2 Introduction

In this project we will solve the Poisson equation using two different algorithms. Poisson's equation is a partial differential equation of elliptic type with broad utility in mechanical engineering and theoretical physics. In the latter, it is for instance used to describe an electrostatic potential generated by a localized charge distribution or a mass distribution. By assuming spherical symmetry, the equation is reduced to one dimention.
We are going to use two different algorithms to solve the Poisson equation. The first is the tridioganal matrix algorithm and the second is the LU decomposition method. Where the tridiagonal matrix algorithm can be specialized to the specific problem at hand, the LU-decomposition algorithm is a general algorithm which can be used on any matrix. We will test the accuracy of the methods with different step sizes and measure the time it takes to run the different algorithms.

3 Theory

3.1 The Poisson Equation

The Poisson Equation is a classical equation from electromagnetism. The electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimentions it reads

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

With a spherically symmetric Φ and $\rho(\mathbf{r})$ the equations simplifies to a one-dimensional equation in r, namely

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

which can be rewritten via a substitution $\Phi(r) = \phi(r)/r$ as

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

The inhomogeneous term f or source term is given by the charge distribution ρ multiplied by r and the constant -4π. We will rewrite this equation by letting $\phi \to u$ and $r \to x$. The general one-dimensional Poisson equation then reads

$$-u''(x) = f(x).$$

In this project the source term is given as $f(x) = 100e^{-10x}$, and we will compare it to the analytical solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. We will solve the equation with Dirichlet boundary conditions in the interval $x \in [0, 1]$.

3.2 Approximation of the second derivative

We will solve the one-dimentional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. The discretized approximation is defined as u and v_i, with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length is given as $h = 1/(n+1)$. This gives the boundary conditions $v_0 = v_{n+1} = 0$. The second order derivative is approximated with the three point formula, and reads

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

After defining $\mathbf{f} = h^2 f$, one can rewrite the equation as $-v_{i+1} + v_{i-1} - 2v_i = h^2 f_i$. This can now be represented as a matrix equation, where $\mathbf{A}$ is a $n \times n$ triagonal matrix.

$$A\mathbf{v} = \mathbf{f} \tag{2}$$

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix} \tag{3}$$

3.3 The tridiagonal matrix

The matrix A can be rewritten for a general case in terms of one-dimensional vectors a, b and c. Then our linear equation reads

$$\begin{bmatrix} \mathrm{b}_1 & \mathrm{c}_1 & 0 & 0 & \dots & 0 & 0 & 0 \\ \mathrm{a}_1 & \mathrm{b}_2 & \mathrm{c}_2 & 0 & \dots & 0 & 0 & 0 \\ 0 & \mathrm{a}_2 & \mathrm{b}_3 & \mathrm{c}_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \mathrm{a}_{n-2} & \mathrm{b}_{n-1} & \mathrm{c}_{n-1} \\ 0 & 0 & 0 & 0 & \dots & 0 & \mathrm{a}_{n-1} & \mathrm{b}_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n \end{bmatrix} \tag{4}$$

This is a tridiagonal matrix, which is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_{i-1}v_{i-1} + b_i v_i + c_i v_{i+1} = f_i \tag{5}$$

for i = 1,2,...,n. To solve this equation, we use a forward and a backward substitution, common to the algorithms based on Gaussian elimination but with fewer floating point operations.

3.4 Algorithm for the tridiagonal matrix

The forward substitution gives an update to the diagonal elements b_i given by the elements $\tilde{b_i}$

$$\tilde{b_i} = b_i + \frac{a_{i-1}c_{i-1}}{\tilde{b_i}} \tag{6}$$

and $\tilde{f_i}$

$$\tilde{f_i} = f_i + \frac{a_{i-1}\tilde{f_{i-1}}}{\tilde{b_i}} \tag{7}$$

We loop over all elements i and have that $\tilde{b_1} = b_1$ and $\tilde{f_i}$=b_1. We get the final solution v by backward substitution

$$v_{i-1} = \frac{\tilde{f_{i-1}} - c_{i-1}v_i}{\tilde{b_{i-1}}} \tag{8}$$

with $v_n = \frac{\tilde{f_n}}{\tilde{b_n}}$ in the last point n.

We can make a specialized algorithm for the case where all the elements in each vector are the same, as it is in the approximation of the second derivative. Then we don't have to save the vectors a, b and c. This makes it easier to calculate $\tilde{b}$, $\tilde{f}$ and lastly v. This will reduce the number of floating point operations in each loop and save a lot of time.

3.5 LU-decomposition

One can also use the LU-decomposition method to find a solution to the Poisson equation. The LU-decomposition decomposes the initial matrix **A** to a lower triangular matrix **L** and an upper triagonal matrix **U**,

$$\mathbf{A} = \mathbf{LU}.$$

Now taking a look back at equation 2, and substitute **A** with LU

$$\mathbf{LUv} = \mathbf{f}.$$

where **L** and **U** is known and we want to solve for **v**. This can be calculated in two steps

$$\mathbf{Ly} = \mathbf{f} \tag{9}$$

$$\mathbf{Uv} = \mathbf{y} \tag{10}$$

Then we can use the inverse of **L** to obtain

$$Uv = L^{-1}f = y \tag{11}$$

and then, when we have y, we can find v by $\mathbf{Uv} = \mathbf{y}$. In our numerical solution we will use the library Armadillo to solve the LU-decomposition.

3.6 Number of FLOPS

In order to create an efficient code, it is important to look at the number of floating point operations (FLOPS) and see if one can make some adjustments. Everything that can be calculated before the loop should be left outside the loop to have as few as possible operations repeat itself. In our code we have two loops that run n-1 times. The first loop has 6 FLOPS and the second has 2, which gives FLOPS = $6(n-1) + 2(n-1) = 8(n-1) \approx 8n = O(8n)$. The LU-decomposition requires $O(\frac{2}{3}n^3)$ FLOPS.

4 Method

The programs can be found in our GIT repository, linked here.

4.1 General algorithm

This algorithm is made of two different loops, the decomposition and forward subtituion followed by the backward substitution.

```
f_tilde[0] = f[0];
b_tilde[0] = b[0];
for(int i=1; i<n+1; i++){
    b_tilde[i] = b[i] - a[i-1]*c[i-1]/b_tilde[i-1];
    f_tilde[i] = f[i] - a[i-1]*f_tilde[i-1]/b_tilde[i-1];
}
v[0] = v[n] = 0;
v[n-1] = f_tilde[n-1]/b_tilde[n-1];
for(int i=n-2; i>=1; i--){
    v[i] = (f_tilde[i]-c[i]*v[i+1])/b_tilde[i];
    error[i] = log10(abs((v[i]-u[i])/u[i]));
}
```

As seen from the psuedo code above, nothing is assumed of the tridiagonal matrix. Therefore it can be used as a general code to solve tridiagonal matrix multiplications.

4.2 Special algorithm

The algoritm with our specific case do require less of our computer, as discussed in section 3.3. Since some of the values used in the loop can be pre-computed, it is easier to implement. As in section 4.1, we still have two loops over the decomposition, forward and backward substitution, but these are now simpler.

```
f_tilde[0] = f[0];
b_tilde[0] = 2;
for(int i=1; i<n+1; i++){
    b_tilde[i] = 2 - 1/b_tilde[i-1];
    f_tilde[i] = f[i] + f_tilde[i-1]/b_tilde[i-1];
}
v[0] = v[n] = 0;
v[n-1] = f_tilde[n-1]/b_tilde[n-1];
for(int i=n-2; i>=1; i--){
    v[i] = (f_tilde[i]+v[i+1])/b_tilde[i]
    error[i] = log10(abs((v[i]-u[i])/u[i]));
}
```

4.3 LU-decomposition algorithm

The LU-decomposition algorithm is implemented by using the Armadillo Linear Algebra library for *C++*. After initializing the tridiagonal matrix $\mathbf{A}$, we can solve equation 9 and 10 by using Armadillo's *solve* function. When creating the matrix $\mathbf{A}$ we initialized some values in order to avoid *if*-statements in the for-loop and get a more efficient code.

```
A(0,0) = 2.;
A(n-1,n-1) = 2;
A(0,1) = -1;
```

```
A(1,0) = -1;
A(n-1,n-2) = -1;
A(n-2,n-1) = -1;
for (int i = 1; i <= n-2; i++){
    A(i,i) = 2;
    A(i, i+1) = -1;
    A(i, i-1) = -1;
}
```

4.4 Comparison to the analytical solution

The analytic solution was also implemented in the code. The algorithms were solved for different sizes of the matrices, and compared to the numerical solution by

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right). \tag{12}$$

Where u_i is the analytic solution and v_i is the numerical solution, both at a some point i. Equation 12 expresses the relative error between the two solutions. We will only look at the maximum error for different values of n.

5 Results

5.1 Numerical solution compared to the analytical

A plot of the analytical solution versus different values of n are shown in figure 1.

5.2 Relative error

Here the relative error is shown in the log-log plot in figure 2.

5.3 Computational speed

We also compared the computational speed of the different algorithms described in section 3. This is shown in table 1 below, for different n-values.

N-values	General algorithm [s]	Special algorithm [s]	LU-decomposition [s]
10^1	$2.5 \cdot 10^{-5}$	$4.0 \cdot 10^{-6}$	$7.2 \cdot 10^{-3}$
10^2	$2.6 \cdot 10^{-5}$	$2.7 \cdot 10^{-5}$	$5.3 \cdot 10^{-2}$
10^3	$2.12 \cdot 10^{-4}$	$2.14 \cdot 10^{-4}$	$2.2 \cdot 10^{-1}$
10^4	$2.11 \cdot 10^{-3}$	$1.65 \cdot 10^{-3}$	16.42
10^5	$2.26 \cdot 10^{-2}$	$2.31 \cdot 10^{-2}$	N/A
10^6	$1.22 \cdot 10^{-1}$	$1.17 \cdot 10^{-1}$	N/A
10^7	1.83	1.59	N/A

Table 1: comparison of the time used to solve Poisson's equation with different algorithms and different values of n.

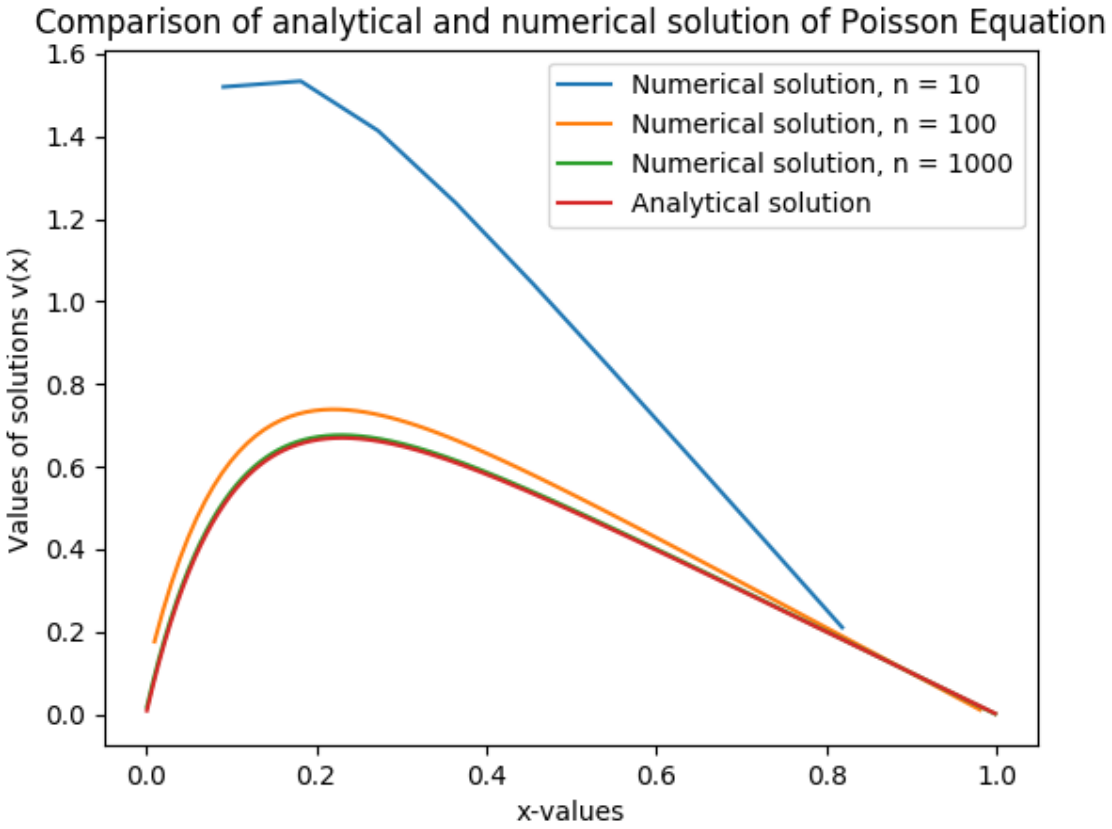

Figure 1: Plot of the analytical and numerical solution of Poisson's equation as described in section 3.1. Here the red line is the analytical solution, and the other lines are the numerical solutions with different step-sizes.

6 Discussion

We have in this article applied different methods for solving linear equations. As one can see from table 1, the time used for different n-values is smallest for the specialized algorithm and largest for the LU-decomposition. For small values of n one has to take into account that both the operating system and the time-precision in the computer can affect the measured time. For larger n, this becomes less relevant, as mentioned in section 3.6. For $n = 10^4$ the specialized algorithm uses about 78 percent of the time that the general algorithm does. The LU-decomposition method only works for n up to $n = 10^4$ as it requires more RAM than our computer has to create larger matrices. For this value of n, the LU-decomposition uses about ten thousand times more time than the specialized algorithm.

The accuracy of the numerical solution compared to the analytical solution for different values of n can be seen in figure 1 and the error in figure 2. For n=10, the error is large. The numerical solution becomes a lot more accurate by increasing n to a hundred, and at n=1000, the numerical solution is very close to the analytical. From the error-plot, one can see that at n larger than 10^5, due to loss of numerical precision, the error starts to increase. When taking into account both the speed and the accuracy, it seems that choosing $n = 10^5$ is ideal.

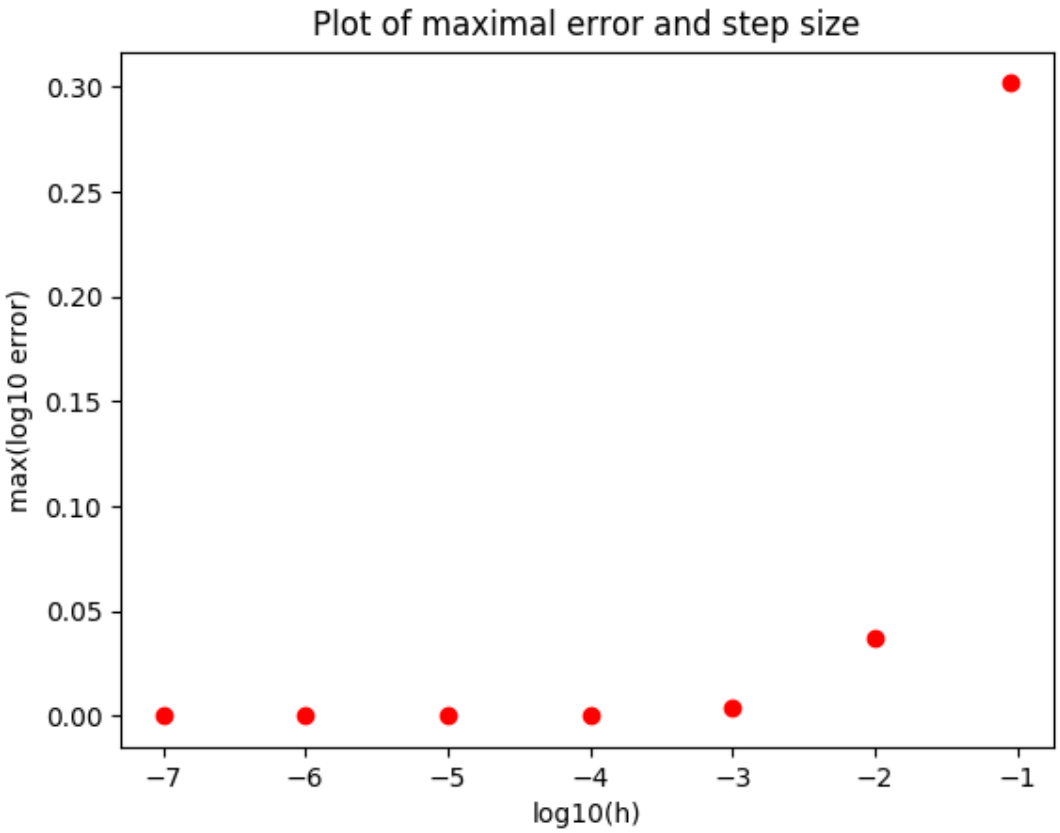

Figure 2: Plot of the relative error as described in section 4.4. Here the error is calculated by using equation 12.

7 Conclusion

It seems that in our case, the specialized method is preferable. It uses less flops, spends less time running and is easier to implement compared to the general algorithm and the LU-decomposition. The drawback with this method is that we cannot make any changes without having to alter the algorithm. The general method is just a bit slower and has the possibility to change the three vectors. The LU-decomposition is straight-forward to solve, but is much slower than the other two.
It is therefore important to analyze the problem at hand before you start coding to decide which method is easiest to implement, most accurate and fastest to run.