FYS3150-Project 1

Marcus Berget, Sebastian Amundsen, Andreas Wetzel

August 2020

Abstract

1   Introduction

In this project we will use numerical methods to solve the one dimensional Poisson equation. We will be rewriting the Dirichlet boundary conditions as a set of linear equations. Our numerical methods will have varying degrees of accuracy. We are going to compare our algorithms and the CPU time for the different numerical methods.

2   Method

The one dimensional Poisson equation with Dirichlet boundary conditions is given by:

$$-u''(x) = f(x) \qquad x \in (0,1) \qquad u(0) = u(1) = 0 \qquad (1)$$

Let's define the discretized approximation to $u(x)$ as v_i, and to $f(x)$ as f_i. We can then make an approximation of the second derivative of u given by:

$$\frac{-v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \text{ where } i = 1, 2, 3......, n, \qquad (2)$$

Here the step length of spacing is given by $h = 1/(n+1)$ and the grid points is defined by $x_i = ih$. We can rewrite equation 2 as a linear set of

equations on the form $\mathbf{Av} = \tilde{\mathbf{b}}$:

$$\mathbf{Av} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & & \\ 0 & \vdots & \vdots & -1 & 2 & -1 \\ 0 & \vdots & \vdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_n \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} 2v_0 - v_1 \\ -v_0 + 2v_1 - v_2 \\ -v_1 + 2v_2 - v_3 \\ \vdots \\ -v_{n-1} + 2v_n - v_{n+1} \\ -v_n + 2v_{n+1} \end{bmatrix}$$

$$= \begin{bmatrix} h^2 f_0 \\ h^2 f_1 \\ h^2 f_2 \\ \vdots \\ h^2 f_n \\ h^2 f_{n+1} \end{bmatrix} = \tilde{\mathbf{b}} \tag{3}$$

In our case we will use the source term $f(x) = 100e^{-10x}$ which gives us a closed form solution $u(x)$ given by:

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

We will use this solution as a point of reference to discuss the accuracy of our numerical methods.

2.1 General algorithm for tri-diagonal matrix

There exists an algorithm for solving generic sets of linear equations, but in the case for equation (3) where we have a tri-diagonal matrix we can use a different algorithm which decreases the amount of floating point operations needed. We denote the elements in the leading diagonal of A as $b_1, b_2, ..., b_n$, the elements above the leading diagonal as $a_2, a_3..., a_n$, and the elements below the leading diagonal as $c_1, c_2, ..., c_{n-1}$. The algorithm uses a forward substitution to replace the leading diagonal with elements denoted by $\tilde{b}_i$, and replaces the righthand side in with the elements $\tilde{f}_i$ as shown below:

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

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

where $\tilde{b}_1 = b_1$ and $\tilde{f}_i = f_i$. The algorithm then continues with a backward substitution which gives the solution:

$$u_{i-1} = \frac{\tilde{f}_{i-1} - c_{i-1}u_i}{\tilde{b}_{i-1}} \tag{5}$$

2.2   Algorithm for specific tri-diagonal matrix

In our special case we can implement a solver that is even simpler than what is described previously. We will exploit the fact that the matrix has identical matrix elements along the diagonal and identical values for the non diagonal elements $\vec{e}_i$. In this case we can precalculate the new values for the updated matrix elements d_i without taking into account the values for $\vec{e}_i$:

$$d_i = 2 - \frac{1}{\tilde{d}_{i-1}} = \frac{i+1}{i} \tag{6}$$

Here the initial value is $\tilde{d}_1 = 2$. The new righthand side solution $\tilde{f}_i$ is given by:

$$\tilde{f}_i = f_i + \frac{(i-1)\tilde{f}_{i-1}}{i} \tag{7}$$

Here the initial value is $\tilde{f}_1 = f_1$. The last step is to make a backward substitution which gives the final solution u_i:

$$u_{i-1} = \frac{i-1}{i}(\tilde{f}_{i-1} + \tilde{u}) \tag{8}$$

This method requires that we know the last value u_n in the u_i array. This value is given by $u_n = \tilde{f}_n / \tilde{b}_n$.

2.3   Relative error

Algorithms have a varying degree of uncertainty. We will test how precise our algorithm is for the specific tri-diagonal matrix case. The numerical solution will be compared to the analytical solution given the relative error ϵ_i:

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{9}$$

3

For the numerical v_i and analytical u_i function values. We wish to extract the max value of the relative error for varying numbers of grid points n. This will give us some information about how precise the numerical approximation is compared to the analytical solution.

2.4   LU decomposition

We are now going to LU decomposition the matrixes 10 x 10, 100 x 100, 1000 x 1000 and 10 000 x 10 000. What the LU decomposition does, is that it rewrite a matrix as the product of two other matrices L and U, like this:

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

There are several reasons why we use LU decomposition instead of standard Gaussian elimination. First of all, it is straight forward to solve the determinant of a matrix. Second, if we still need to solve a set of linear equation with the same matrix, but with a different vector, the number of FLOPS is of order n^3. Where FLOPS is floating point operations per second. Where a such operation is the inverse.
By using LU decomposition, it will be comfortably to solve the inverse, determinate and linear equation.

In order to calculate a set of linear equation by using the LU decomposition for matrix Ax=w. We first need to compose a matrix A to two matrices L and U. When we have done that we split our equation in to two equation, which you can see under here.

$$\mathbf{Ax = LUx = b}$$
$$\mathbf{Ax = LUx = b}$$
$$\mathbf{Ly = b \wedge Ux = y}$$

Where **b** is given as the one-dimensional Poisson equation times the square of the step length h.

$$\mathbf{b} = -\frac{d^2u}{dx^2}h^2 = f(x)h^2$$

4

Since we now have obtained two separated equation, we now solve for y in equation

$$\mathbf{Ly} = \mathbf{b} \rightarrow \mathbf{Ly} = \mathbf{b}$$

. Which we afterwards plug in

$$\mathbf{Ux} = \mathbf{y} \rightarrow \mathbf{x} = \mathbf{U^{-1}y}$$

, which then gives us vector **x**.

There are several reasons why we use LU decomposition instead of standard Gaussian elimination. First of all, it is straight forward to solve the determinant of a matrix. Second, if we still need to solve a set of linear equation with the same matrix, but with a different vector, the number of FLOPS is of order n^3. Where FLOPS is floating point operations per second. Where a such operation is the inverse.

3 Implementation

All programs used is available at:
`https://github.com/Sebamun/FYS3150_Projekter`
We implement the algorithms numerically with varying values of grid points n. The approximations are plotted for the general algorithm and we find the relative error for the specific case. The relative error is really high at the endpoints for the specific algorithm. This is due to the ratio between numerator and denominator in equation 9. This is solved by excluding the endpoints for the approximation.
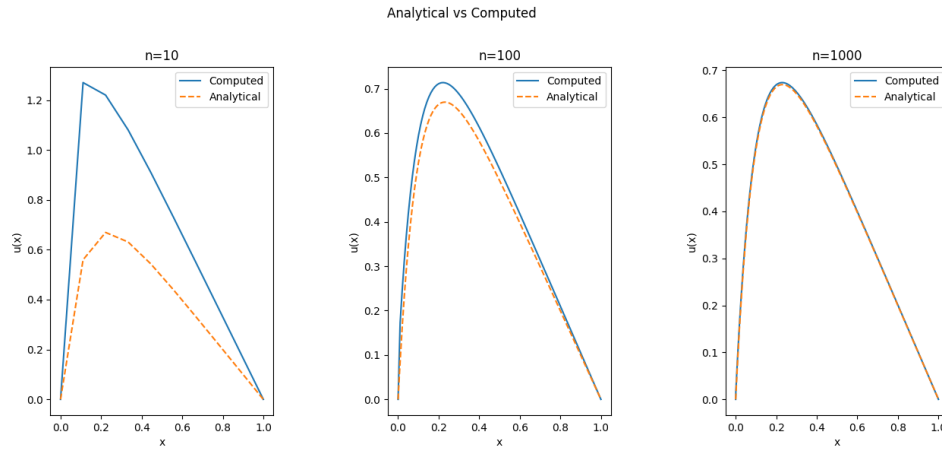
4 Results

4.1 General algorithm

Figure 1: Plot comparing analytical vs computed solutions of the differential equation. We can see that the computed solution approaches the analytical solution when the number of gridpoints increases.

4.2 Algorithm for specific tri-diagonal matrix

4.3 Relative error

4.4 LU decomposition

Country List			
Size	Generel algorithm	Special algorithm	LU decomposition
n =10	0.135 seconds		0.585 seconds
n = 100	0.137 seconds		0.591 seconds
n = 1000	0.146 seconds		0.736 seconds
n = 10 000	0.203 seconds		56.580 seconds
n = 100 000	0.730 seconds		...
n = 1 000 000	6.177 secondss		...

5 Discussion

5.1 LU decomposition

Table 1: Relative error for specific tri-diagonal matrix algorithm.

Steps (n)	Relative error (ϵ)
10	$6.20 * 10^{-1}$
10^2	$1.21 * 10^{-1}$
10^3	$2.36 * 10^{-2}$
10^4	$3.61 * 10^{-3}$
10^5	$4.89 * 10^{-4}$
10^6	$6.17 * 10^{-5}$
10^7	$7.45 * 10^{-6}$

5.2 LU decomposition

6 Discussion

We can see from table 1 that the relative error decreases as n increases. This is due to the fact that more grid points lead to better precision. However this is only true as long as we have enough memory to store our information. If we would increase the number of grid points further than $n = 10^7$ our relative error could potentially increase dramatically. Since we are using python, more than $n = 10^7$ grid points make the algorithm take too long to run. We could have used a larger n If we had used a faster programming language (like C++), making it possible to observe an increasing relative error at a larger n.

6.1 LU decomposition

If we compare our solution from the general algorithm and the CPU-time with the LU decomposition, we see that it takes a lot of more time to calculate the LU decomposition. This comes that it takes n^3 steps to find the LU decomposition, which is the same for gauss- elimination. But when we calculate LU decomposition, we calculate a set of two equation instead of

one. This shows us that there are more steps for calculating just one vector. If we are going to calculate other vectors as well for the same matrix will show us that the LU decomposition will be much more effective. This comes from that we have already calculated the LU decomposition, which lead us to the calculation of the vector now is just n^2.

The reason why we can't use LU decomposition on a $10^5 \times 10^5$ matrix comes from the random accesses memory problem. The data that should be stored from the operation takes to much memory that the computer cant handle the operation we want to.

The number of floating point operations that the LU decomposition use to solve the set of linear equation equals $\frac{2}{3}n^3$. This means that the FLOPS depend on the size of the matrix. When the matrix becomes large, we can ignore the orders n and n^2.

The number of floating point operations that the LU decomposition use to solve the set of linear equation equals $\frac{2}{3}n^3$. This means that the FLOPS depend on the size of the matrix. When the matrix becomes large, we can ignore the orders n and n^2.

7   Concluding remarks

References