FYS4150 Project 1: 1-dimensional Poisson equation

Marie Foss, Maria Hammerstrøm

Abstract

We explore the efficiency of two methods for solving a set of linear equations when the matrix involved is a tridiagonal matrix. The first method is a simplified algorithm that takes into account the simplicity of the tridiagonal matrix. The second method is an algorithm using a full LU decomposition, which will work for any matrix. We find that the simplified algorithm is superior when dealing with a large tridiagonal matrix, both with respect to accuracy of the result, as well as the computing time.

1 Introduction

In this project we solve the one-dimensional Poisson equation with Dirichlet boundary conditions. This is done by rewriting it as a set of linear equations, and solving these numerically in a program written in C++ [1]. The C++ program is accompanied by a MATLAB script which plots the results.

The equation to be solved is

$$-u''(x) = f(x), \tag{1}$$

with $x \in (0, 1)$ and $u(0) = u(1) = 0$.

We define the discretized approximation to u as v_i with grid points $x_i = ih$, $i = 0, 1, \ldots, n+1$. The step length, or spacing, is defined as $h = 1/(n+1)$, and the boundary conditions are $v_0 = v_{n+1} = 0$. The second derivative of u can be approximated to be

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

where $f_i = f(x_i)$. This set of equations can then be written as

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}, \tag{3}$$

where $\tilde{b}_i = h^2 f_i$ and $\mathbf{A}$ is an $n \times n$ matrix of the form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix}. \tag{4}$$

This transition can be shown by writing it out. We then get

$$\begin{aligned} 2v_1 - v_2 &= \tilde{b}_1 \\ -v_1 + 2v_2 - v_3 &= \tilde{b}_2 \\ -v_2 + 2v_3 - v_4 &= \tilde{b}_3 \\ &\dots \\ -v_{i-1} + 2v_i - v_{i+1} &= \tilde{b}_i, \end{aligned}$$

which is the same as Eq. (2).

We assume that the source term is $f(x) = 100\exp -10x$. Using the interval and boundary conditions stated above, equation (1) has a closed-form solution given by

$$u(x) = 1 - (1 - \exp -10)x - \exp -10x. \tag{5}$$

We will compare our numerical solution with this result.

2 Methods

2.1 Simple algorithm

In our case we are dealing with a simple tridiagonal matrix. We can therefore rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1{:}n$. The linear equation then reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}. \tag{6}$$

This can be written as

[1] All files associated with the project are located at: https://github.com/mariahammerstrom/Project1

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i \quad (7)$$

for $i = 1, 2, \ldots, n$. We want to find a simple algorithm to solve this set of equations. To do this, we can rewrite eq. (7) by a method of elimination, which leads to a system where there is only one unknown.

The first thing to do is to realize that eq. (7) gives a system of three equations,

$$b_1 v_1 + c_1 v_2 = \tilde{b}_1, \qquad i = 1 \quad (8)$$

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i, \qquad i = 2, \ldots, n-1 \quad (9)$$

$$a_n v_{n-1} + b_n v_n = \tilde{b}_n, \qquad i = n, \quad (10)$$

where the boundary conditions have been applied to simplify the equations. The idea now is to subtract one row with a scalar multiple of another to eliminate variables. First we eliminate v_1, by dividing the first equation by b_i, which gives

$$v_1 = \frac{1}{b_1}\tilde{b}_1 - \frac{c_1}{b_1} v_2. \quad (11)$$

By dividing each equation by b_i, we find similar expressions for v_i,

$$v_i = \tilde{b}_i / b_i - \frac{a_i}{b_i} v_{i-1} - \frac{c_i}{b_i} v_{i+1}. \quad (12)$$

Trying to insert v_{i-1} into the equations for v_i can prove to be difficult for large i. After some algebra one finds that

$$v_i = \left(\tilde{b}_i - a_i v_{i-1} - c_i v_{i+1}\right) \Big/ \left(b_i - \frac{a_i c_{i-1}}{d_{i-1}}\right), \quad (13)$$

where $d_{i-1} = b_{i-1} - a_{i-1} c_{i-2} / d_{i-2}$, and $d_2 = b_2 - a_2 c_1 / b_1$. This process is called forward substitution.

We then have an equation for each v_n, and can start going backwards. This is done with a backward substitution. As we see from eq. (13), we have to add a term $-c_i v_{i+1}/d_i$ to each v_i. The solution to $\mathbf{v}$ is then easily found.

2.2 LU decomposition

In addition to solving the linear second-order differential equation in eq. (1) using the simple algorithm described above, we want to solve the equation using LU decomposition. We then compare the results. Doing an LU decomposition means rewriting matrix $\mathbf{A}$ as a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, so that

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

For the LU decomposition we use the provided `lib.cpp` library.

3 Results

3.1 Comparing the analytical and numerical solution

We ran our code for different values of n and plotted the analytical and numerical solution to see how well they correspond. For different values of n, see Fig. 1.

3.2 Relative errors

The relative error in the data set is computed by calculating

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right), \quad (14)$$

$i = 1, \ldots, n$, where u_i represents the closed-form analytical solution, whereas v_i is the numerical solution given by our simple algorithm (see section 2.1). The relative error is a function of $log_{10}(h)$. For each step length h we want to extract the maximum value of the relative error. Doing this while increasing n from $n = 10$ to $n = 10^5$ gives the following results:

n	Maximum error ($\log_{10}$)
10	-0.312833
10^2	-0.305022
10^3	-0.30146
10^4	-0.301073

Table 1: Maximum error as a function of step length.

3.3 Number of floating point operations and time usage

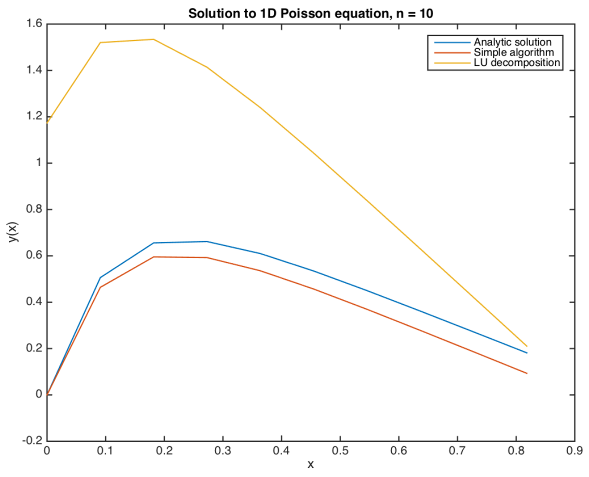
Some of the methods described above may be more computationally heavy for a computer to do than others. One way to evaluate this is to calculate the precise number of floating point operations ("FLOPS") needed to solve the equations.

Gaussian elimination requires $\frac{2}{3}n^3 + O(n^2)$ FLOPS and LU decomposition requires $O(n^3)$. By counting the floating point operations in our code, we find that for our tridiagonal system, $8n + 3$ FLOPS are required, where the number 3 represents the floating point operations outside of the `for`-loops, which can be ignored as $8n$ will be much larger.

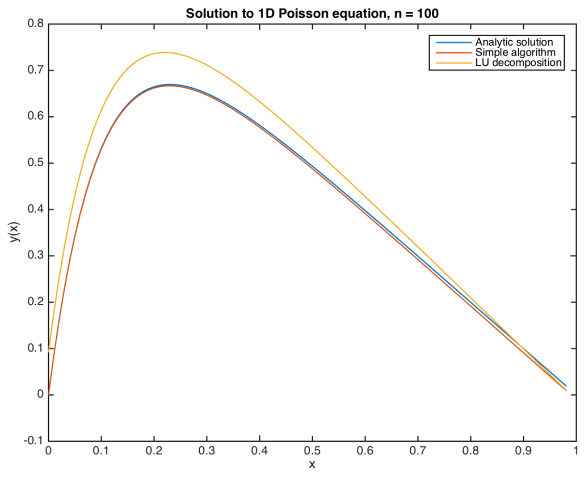
The number of floating point operations and time usage in the different cases, are shown in the table below. A brute force Gaussian elimination is not used in the code in this project, and therefore has no values for time usage.

Method	***n***	**FLOPS**	**Time *(s)***
Simple algorithm	10	8e+1	0
	10^2	8e+2	0
	10^3	8e+3	0
	10^4	8e+4	0
Gaussian elimination	10	1.6e+3	-
	10^2	1.51e+06	-
	10^3	1.501e+09	-
	10^4	1.5001e+12	-
LU decomposition	10	1e+3	0
	10^2	1e+06	0
	10^3	1e+09	3
	10^4	1e+12	6726

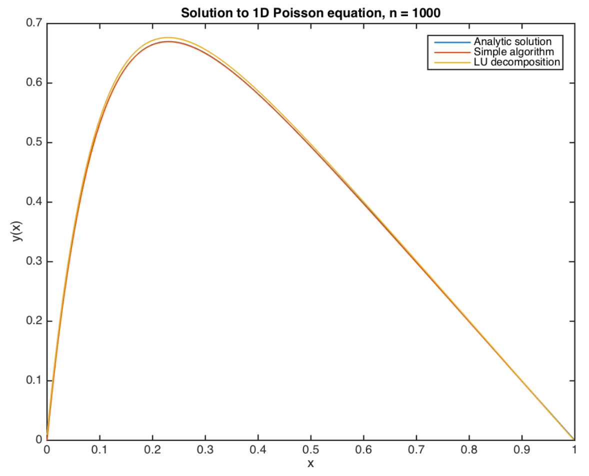
Table 2: The number of floating point operations and time usage by method.



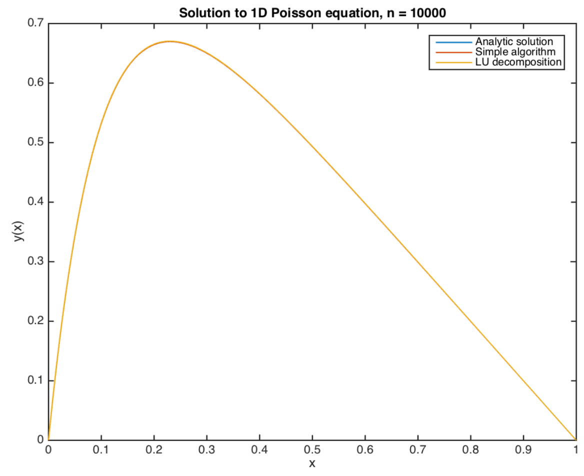
(a) $n = 10$



(b) $n = 10^2$



(c) $n = 10^3$



(d) $n = 10^4$

Figure 1: Comparison of the three different solutions for different values of n. In the three different cases, $y(x) = u(x)$ (analytical solution, blue line), $y(x) = v(x)$ (simple algorithm, red line) and $y(x) = b(x)$ (LU decomposition, yellow line).

4 Conclusions

We ran our code with n ranging from 10 to 10^4. Due to the amount of time it took to compute the LU decomposition for $n = 10^4$, see Table 2, calculating $n = 10^5$ was impossible.

In Fig. 1 the results for the different n-values are shown. For $n = 10$ the result is way off for the LU decomposition, while the other two solutions are relatively close to each other. By only increasing n by a factor of 10 the results improve significantly. The analytical solution and the solution of the simple algorithm are close to the same, while the result from the LU decomposition is lagging behind. Finally at $n = 10^3$, all the three different methods align quite nicely, with the LU decomposition only slightly off, while at $n = 10^4$ does the results from all methods coincide.

If we look at this development with regards to computing time as well, shown in Table 2, we see that the computing time for the simple algorithm for our special case of a tridiagonal matrix does not care about the increasing value of n, while the LU decomposition goes rapidly slower as n is increased. Adding this to the fact that the result from the LU decomposition is not very impressive, the simple algorithm is a highly preferred method to use in this case.

If we review the results of the error analysis shown in Table 1, we see that the result is as expected: The error decreases as n increases. The change in the relative error is largest when going from $n = 10$ to $n = 10^2$, and it hardly changes from $n = 10^2$ to $n = 10^3$ and higher. This corresponds well with what we see in Fig. 1 for the blue and red lines. This means that the effect of increasing n to a very large value when using our simple algorithm doesn't give us much new information. But because the code runs so quickly even if n is large, there is no reason to limit oneself to a small value for n.

From the calculations of the number of floating point operations in Table 2, we see that the simple algorithm continues to have a rather small number of FLOPS as n increases, while Gaussian elimination and LU decomposition have a number of FLOPS that is quickly increasing, which is the reason for the long time usage of the LU decomposition method.

5 List of codes

The codes developed for this project are:

`main.cpp` – main program, C++
`plotting.m` – plotting program, MATLAB