FYS4150 Project 1:
1-dimensional Poisson equation

Marie Foss, Maria Hammerstrøm

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1 Introduction

In this project we will solve the one-dimensional Poissson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations and solving these numerically in a program written in C++.The equation to be solved is:

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

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We then have the boundary conditions $v_0 = v_{n+1} = 0$. We can approximate the second derivative of u with:

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

where $f_i = f(x_i)$. Eq. (2) can be written as a linear set of equations of the form:

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

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 can be shown by writing it out. We then get

$$2v_1 - v_2 = \tilde{b}_1 \tag{5}$$
$$-v_1 + 2v_2 - v_3 = \tilde{b}_2 \tag{6}$$
$$-v_2 + 2v_3 - v_4 = \tilde{b}_3 \tag{7}$$
$$... \tag{8}$$
$$-v_{i-1} + 2v_n - v_{i+1} = \tilde{b}_i, \tag{9}$$

which is the same as Eq. (2).

We will assume that the source term is $f(x) = 100e^{-10x}$. Using the interval and boundary conditions as stated above, the above differential equation has a closed-form solution given by $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. 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:

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}. \tag{10}$$

which can be written as:

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i \tag{11}$$

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. 11 by a method of elimination, which will lead to a system where there is only one unknown.

The first thing to do is to realize that Eq. 11 gives a system of three equations:

$$b_1 v_1 + c_1 v_2 = \tilde{b}_1, \qquad i = 1 \tag{12}$$
$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i, \qquad i = 2, \ldots, n-1 \tag{13}$$
$$a_n v_{n-1} + b_n v_n = \tilde{b}_n, \qquad i = n \tag{14}$$

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 want to eliminate v_1:

(something)

Then we can follow the same approac to eliminate v_2:

(something)

The algorithm for solving this equation can be stated as follows:

(algorithm)

The algorithm used in our code is given on p. 186 in the lecture notes.

2.2 LU decomposition

In addition to solving the linear second-order differential equation Eq. 1 using the simple algorithm described above, we want to solve the equation using LU decomposition, then compare the results. Doing an LU decomposition, means rewriting matrix **A** as a product of a lower triangular matrix **L** and an upper triangular matrix **U**, so that:

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

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

3 Results

3.1 Comparing the analytical and numerical solution

We can run our code for different values of n and plot 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:

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right), \tag{15}$$

for $i = 1, \ldots, n$ for the function values u_i and v_i, where u_i represents the closed-form analytical solution, while v_i is the numerical solution given by the simple algorithm described above. The relative error will be a function of $log_{10}(h)$. For each step length h we want to extract the max value of the relative error. Doing this while increasing n to $n = 10000$ and $n = 10^5$, gives the following results:

n	Maximum error
10	1.18895
10^2	1.03632
10^3	1.00786
10^4	1.00076

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, $8n + 1$ FLOPS are required.

The number of floating point operations and time usage in the different cases, are shown in the table below.

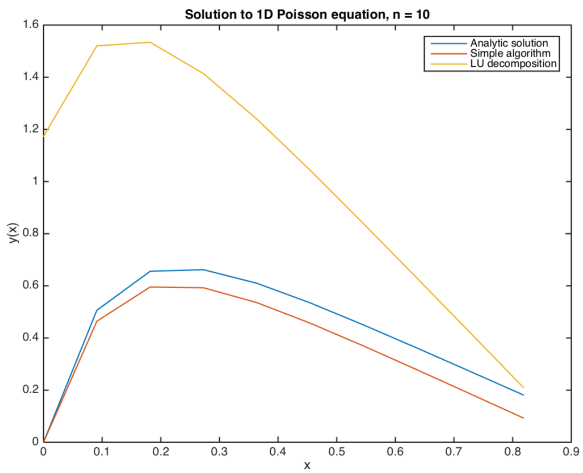
Method	n	**FLOPS**	**Time** *(s)*
Simple algortihm	10	XX	0
	10^2	XX	0
	10^3	XX	0
	10^4	XX	0
Gaussian elimination	10	XX	-
	10^2	XX	-
	10^3	XX	-
	10^4	XX	-
LU decomposition	10	XX	0
	10^2	XX	0
	10^3	XX	3
	10^4	XX	XX

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

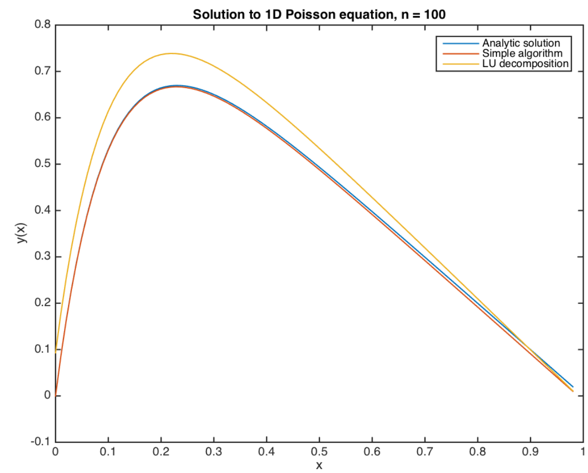
4 Conclusions

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.

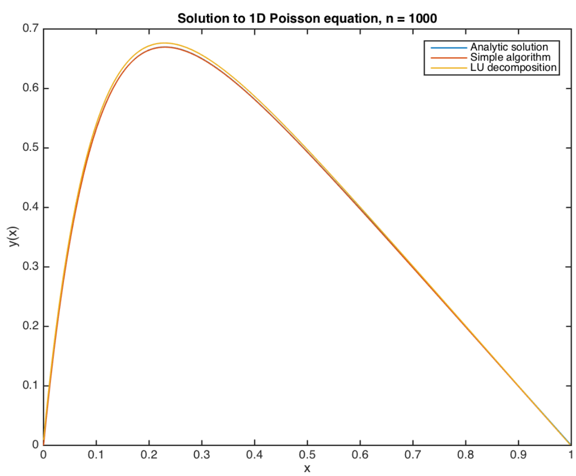
In Fig. 1 the results for the different n-values are shown. For $n = 10$ the result is way off for the LU decomposition, while the other two solutions are relatively close to each other.



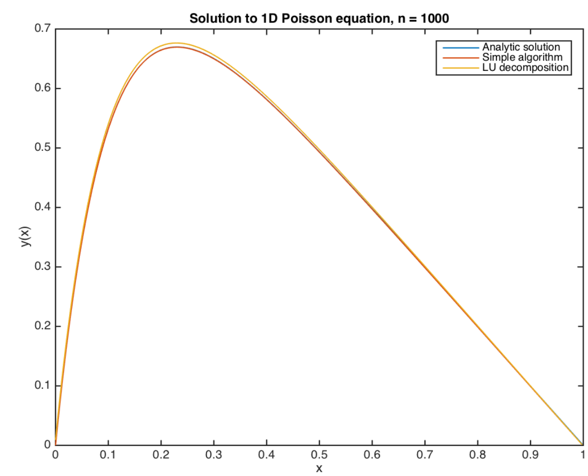
(a) $n = 10$



(b) $n = 100$



(c) $n = 1\,000$



(d) $n = 10\,000$ (still waiting on program)

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).