Project 1, FYS-3150

Ina K. B. Kullmann

September 14, 2015

Contents

All source codes can be found at: `https://github.com/inakbk/Project_1.git` in the folders `exercise_b` and `exercise_d`.

Abstract

The goal for this project is to solve the general one-dimentional Poisson equation with two different numerical methods and compare with the exact analythical solution. For each numerical method the relative error, execution time and number of floating point operations is calculated.

The two numerical methods used to solve the equations is forward/backward substitution and LU-decomposition. Both methods are using linear algebra to turn the problem into a set of many linear equations which can be represented by matrixes.

We will see that the execution time and relative error varies for the two methods and that increasing n gives smaller error up to a point.

1 Introduction: Motivation and Purpose

Many important differential equations in the Sciences can be written as linear second-order differential equations [1]

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x),$$

where f is normally called the inhomogeneous term and k^2 is a real function. It is therefore of special interest to be able to solve these kinds of equations.

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

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

This can be simplified with a spherically symmetric Φ and $\rho(\mathbf{r})$ to:

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

which is a simple one-dimensional equation. Simplifying further via a substitution $\Phi(r) = \phi(r)/r$ the equation reads:

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

We rewrite this equation again by letting $\phi \to u$ and $r \to x$. Then general one-dimensional **Poisson equation** reads:

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

where the inhomogeneous term f (or source term) is given by the charge distribution ρ multiplied by r and the constant -4π.

In this project the general equation 1 is the equation of interest that will be solved numerically. The purpose of this is to shed light on how to best solve simple linear second-order differential equations in Physics numerically by using forward/backward substitution and LU-decomposition.

2 Theory: Rewriting the equation into a linear algebra problem

To solve equation 1 we will use Dirichlet boundary conditions and rewrite the equation as a set of linear equations.

In specific we will solve:

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

where we assume that the the source term is given by $f(x) = 100e^{-10x}$. Then the above differential equation has a closed-form analytical solution given by:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{2}$$

[1] Very much of the text in this section and some of the other sections is copy or moderated text from the exercise text. I wrote this report to assume that the reader did not have the exercise text –> it then felt naturally to reuse some of the text when the .tex file was given. OK, not OK?

For the numerical methods 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 $h = 1/(n+1)$. The boundary conditions gives $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \dots, n, \tag{3}$$

where $f_i = f(x_i)$.

We can rewrite this equation as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}} \tag{4}$$

by rewriting equation 3 as

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

so that $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we write as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix} \tag{5}$$

and the left hand side is given by $\tilde{b}_i = h^2 f_i$. The total set of matrixes must then be given by:

$$\mathbf{A}\mathbf{v} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-1} \\ v_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 \\ h^2 f_2 \\ h^2 f_3 \\ \dots \\ h^2 f_{n-1} \\ h^2 f_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \dots \\ \tilde{b}_{n-1} \\ \tilde{b}_n \end{pmatrix} = \tilde{\mathbf{b}} \tag{6}$$

2.1 Forward and Backward substitution

We can rewrite our matrix $\mathbf{A}$ further in terms of one-dimensional vectors a, b, c of length $1 : n$:

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}. \tag{7}$$

A tridiagonal matrix 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 (Equation 4) can be written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i, \quad \text{for } i = 1, 2, \dots, n \tag{8}$$

The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and a backward substitution.

2.2 LU Decomposition

The LU decomposition method means that we can decompose the matrix $\mathbf{A}$ as the product of two matrices $\mathbf{L}$ and $\mathbf{U}$:

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

The matrix $\mathbf{L}$ has elements only below the diagonal (and thereby the naming lower) and a matrix $\mathbf{U}$ contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). The matrix $\mathbf{A}$ has an LU factorization if the determinant is different from zero. The LU factorization is unique if A is non-singular.

This factorisation can be used to solve equation 4 now written as

$$\mathbf{Av} = \mathbf{LUv} = \tilde{\mathbf{b}}$$

exploiting the fact that

$$\mathbf{Uv} = \mathbf{L}^{-1}\tilde{\mathbf{b}} = \mathbf{y} \Rightarrow \mathbf{Ly} = \tilde{\mathbf{b}}$$

to find $\mathbf{y}$. Then further use the now known $\mathbf{y}$ to find $\mathbf{v}$ by solving the much simpler equation:

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

3 Algorithm

All source codes can be found at: `https://github.com/inakbk/Project_1.git` in the folders `exercise_b` and `exercise_d`.

The algorithms are structured into two `c++` programs and one `python` program. Each of the `c++` programs solves the equations for an nxn matrix, one with the algorithm for forward/backward substitution and the other with the algorithm for LU decomposition. Both algorithms are described below. After solving the equations the result vector v is written to a `.txt` file along with the x value and execution time. The value of n is given on the command line so that the program is executed one time for each n.

The python program compiles and runs the `c++` program[2] so that the `.txt` files are given, loads and plots the data along with the analytical solution and computes and plots the relative error.

3.1 Forward and Backward substitution

The purpose of the forward and backward substitution is to make row operations on both left/right hand side of the equation so that at the left hand side eventually is left with the identity matrix times the unknown vector, $\mathbf{A} \rightarrow \mathbf{A}^* \rightarrow \mathbf{I}$ and $\tilde{\mathbf{b}} \rightarrow \tilde{\mathbf{b}}^*$ so the solution is then the altered left hand side:

$$\mathbf{Iv} = \mathbf{v} = \tilde{\mathbf{b}}^*$$

where * is meaning that the matrix in the algorithm now obtains a new value.

In our case we have a very special tridiagonal matrix $\mathbf{A}$, so we only need two operations to get the identity matrix.

[2] The aim was to only have one `c++` progam, but when I tried running the `c++` code with the LU decomposition from the python script i got a compilation error. The same compilation error came when running from the terminal, but not from QT creator. So with little time to finish the project i fixed the problem by running the LU decomposition from QT Creator before running the python script.

We start by the forward substitution which removes the elements below the diagonal from the second comumn and down to the n-th row. The algorithm is as follows:

$$\begin{aligned} b_i^* &= b_i - \frac{a_i c_{i-1}}{b_{i-1}} \\ \tilde{b}_i^* &= \tilde{b}_i - \frac{a_i}{b_{i-1}} \tilde{b}_{i-1}, \qquad \text{for } i = 2, \dots, n \end{aligned}$$

We can also rewrite $a_i = a = -1$ and $c_i = c = -1$ for all i to reduce the number of floating point operations:

$$\begin{aligned} b_i^* &= b_i - \frac{1}{b_{i-1}} \\ \tilde{b}_i^* &= \tilde{b}_i + \frac{\tilde{b}_{i-1}}{b_{i-1}} \end{aligned}$$

The backward substitution then removes all the elements above the diagonal from the next bottom row (n-1) of the matrix and up to row number one.

$$\tilde{b}_{i-1}^* = \tilde{b}_{i-1} - \frac{\tilde{b}_i}{b_i}, \qquad \text{for } i = n, \dots, 2$$

where $c = -1$ as above. This iteration does not affect the b_i since elements under the diagonal, a_i^*, is zero after the forward substitution.

Finally we normalize the left hand side to one to obtain the identity matrix, meaning that the right hand side is:

$$\tilde{b}_i^{**} = \frac{\tilde{b}_i^*}{b_i}$$

so that the solution of the set of equations is $v_i = \tilde{b}_i^{**}$.

Then the precice number of floating point operations needed to solve this set of equations is seven times the number of rows the algorithm iterates over which is n-1.

3.2 LU Decomposition

To solve the equations by LU decomposition we use the Armadillo library functions `lu(L,U,A)` and `solve(A,B)` with the simple code:

```
lu(L,U,A);
vec y = solve(L,b_thilde);
vec v = solve(U,y);
```

3.3 Relative error

The relative error in the data set in relation to the analytical solution can be computed by

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

where ϵ_i is the relative error for each corresponding value of x_i where $i = 1, \dots, n$.

It can be shown that the relative error is constant over the whole data set (for one n). We will therefor choose the relative error for each n, ϵ to be the mean of the relative error for the whole dataset, ϵ_i

$$\epsilon = \frac{\epsilon_1 + \epsilon_2 + \cdots + \epsilon_n}{n}$$

We will let n be an element of $N = [10^1, 10^2, \ldots, 10^5]$ plot the relative error for all the different n-values as a function of $log_{10}(h)$

3.4 Calculating execution time

To compute the execution time in `c++` the following statements where used:

Time in C++

```
using namespace std;
...
#include "time.h"   //  you have to include the time.h header
int main()
{
    // declarations of variables
    ...
    clock_t start, finish;  //  declare start and final time
    start = clock();
    // your code is here, do something and then get final time
    finish = clock();
    ( (finish - start)/CLOCKS_PER_SEC );
...
```

The timing only enclosed the code which did the calculations, not the code which generated the arrays/matrixes neded to do the calculations or the code which wrote the data to file.

4 Results

In figures 1 and 2 we see the two numerical methods plotted together with the analytical solution for n in $N = [10^1, 10^2, 10^3, 10^4]$. The backward/forward substitution method (blue) is clearly closest to the analytical solution (red) for low n. For bigger n the LU decomposition method lookes just as good as the substitution method with 'by eye' estimates. In figure 3 we see that the relative error for the two methods is reduced for bigger n (smaller h), but the error is clearly smallest for the substitution method for all n. For $n = 10^4$ the error is about a factor $\approx 10^3$ smaller for the substitution method than the LU decomposition method. We also see that the execution time is smaller for the substitution method, as much as by a factor $\approx 10^6$ for $n = 10^4$!

The LU decomposition method did not work for $n > 10^4$, then the program crashed, but the substitution method did not crash. For fun the last plot of this is in figure 4 we see the errors and execution time for $N = [10^1, \ldots, 10^8]$ for the substitution method and $N = [10^1, 10^2, 10^3, 10^4]$ for the LU decomposition method. We see that the relative error gets smaller and smaller untill $n = 10^6$ where it rises to the roof and stays there for higher n. But we can also see that the execution time does not rise rapidly at $n = 10^6$, but increases evenly. The execution time for $n = 10^8$ for the substitution method is still less than the execution time for $n = 10^4$ for the LU decomposition method.

Figure 1: Plots of the numerical and analytical solutions for $n = 10, 100$.

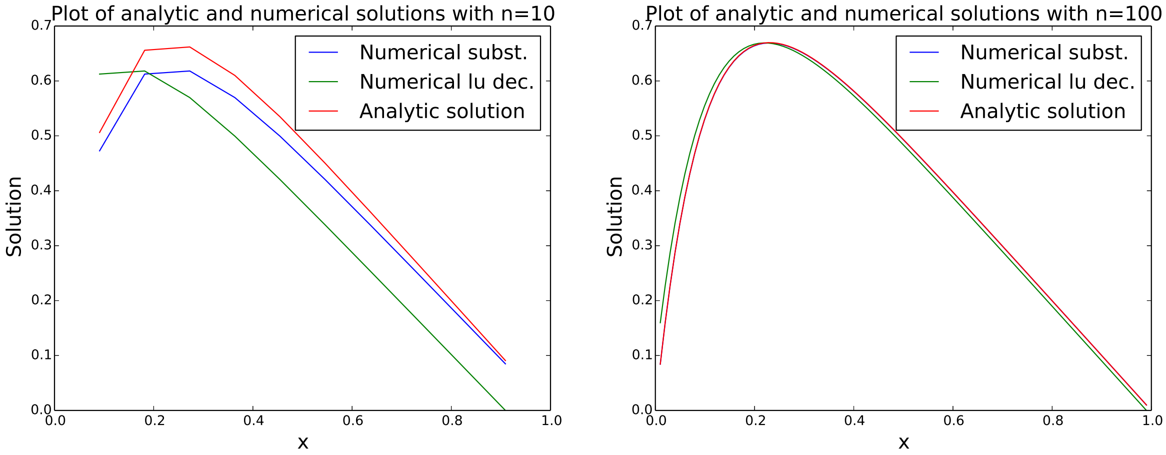
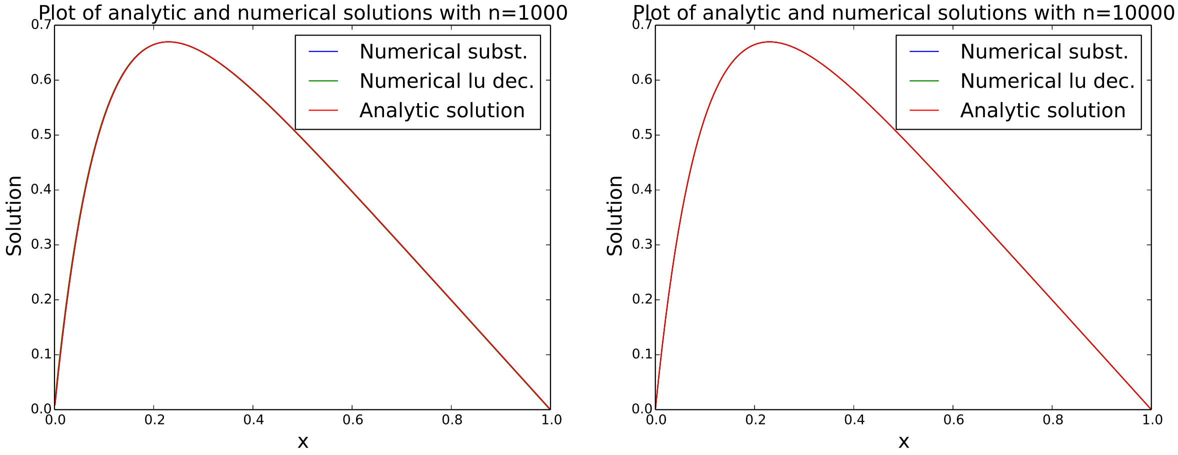


Figure 2: Plots of the numerical and analytical solutions for $n = 10^3, 10^4$.

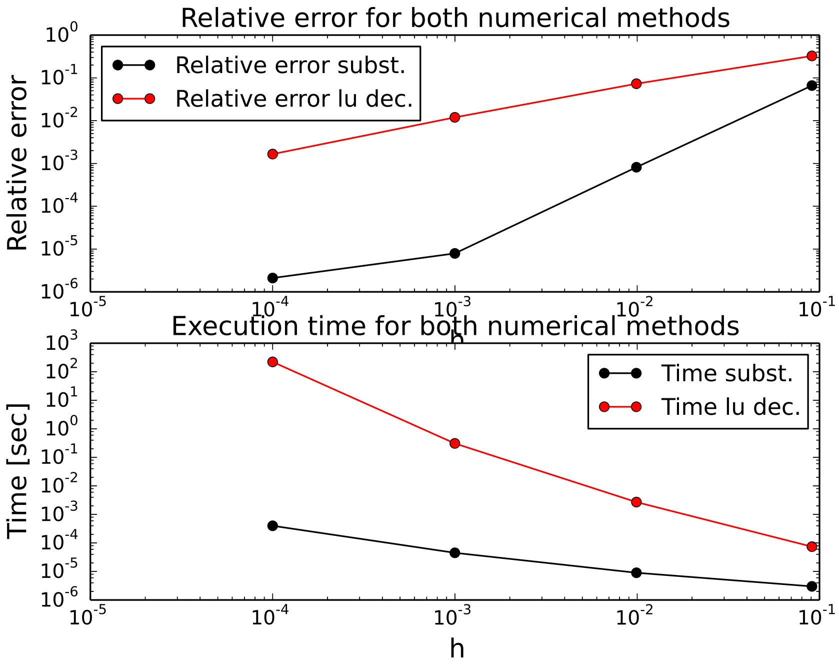


5 Discussion and experiences

We have seen that the forward/backward substitution method gives both less relative error and smaller execution time than the LU decomposition method from the Armadillo library. This is as expected because the substitution method is exploiting the very special tridiagonal form of the matrix $\mathbf{A}$. The LU decomposition algorithm must be much more general to solve all kinds of sets of equations. This generality is nice because one would not have to do so much analytical work befor solving the equations, but is lead to more round-off/owerflow errors which gives loss of precition and also a longer execution time. The factor of $\approx 10^6$ longer execution time for the LU decomposition method for $n = 10^4$ is a lot more than I expected!

It was interesting to see that the LU decomposition method did not work for $n > 10^4$, I

Figure 3: Plots of the relative error and execution time.

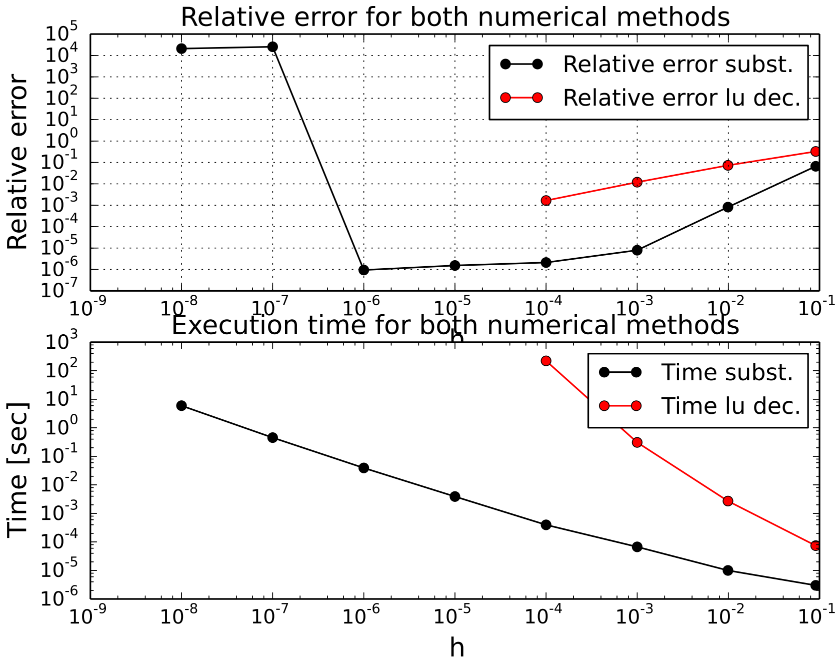


did not expect that. But I did expect the relative error to increase after a point for very large n. It was impressive to see that the execution time for large n did not increase drasticaly as the relative error did. As mentioned the LU decomposition method cannot be uset for large matrixes, but I would aviod using it anyway for matrixes where other possible smart methods can be used, unless loss of precition and execution time is not important. Another point is that iIt is always smart to do some analytical work to check if the results are making sense, as for the substitution method in this project.

Unfortunately I did not have time to try to implement the `new` and `delete` commands for dynamically memory allocation, I am using c++ for the first time. But I did construct the program such that the vectors and matrixes scaled up/down for the different n.

But it was nice to repeat some linear algebra and imlement the methods to solve huge systems of differential exuations in Physics. And of course, really smar and fun to start programing `c++`!

Figure 4: Plots of the relative error and execution time for ridiculous large n.


Relative error for both numerical methods
Relative error
Relative error subst.
Relative error lu dec.
10^{5}
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-9}
10^{-8}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
h
Execution time for both numerical methods
Time [sec]
Time subst.
Time lu dec.
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-9}
10^{-8}
10^{-7}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
h