Project 1, FYS-3150

Ina K. B. Kullmann

September 13, 2015

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.

1 Introduction: Motivation and Purpose

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

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

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 $\phi \rightarrow u$ and $r \rightarrow x$. 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:

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

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 **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

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

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 **A** as the product of two matrices **L** and **U**:

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

The matrix **L** has elements only below the diagonal (and thereby the naming lower) and a matrix **U** contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). The matrix **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 **y**. Then further use the now known **y** to find **v** by solving the much simpler equation:

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

3 Algorithm

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[1] 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, $\mathbf{A} \to \mathbf{A}^* \to \mathbf{I}$ and $\tilde{\mathbf{b}} \to \tilde{\mathbf{b}}^*$ 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.

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.

[1] 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.

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);
```

Find also the precise number of floating point operations needed to solve the above equations. ?!?

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 $i = 1, \ldots, n$.

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 elapsed time in c++ you can use the following statements

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 );
...
```

Exercise:
Your first task is to set up the algorithm for solving this set of linear equations. Find also the precise number of floating point operations needed to solve the above equations. Compare this with standard Gaussian elimination and LU decomposition.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you choose $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the closed-form solution for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

sdkjcfnwif

Compute the relative error in the data set $i = 1, \dots, n$,by setting up

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

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length extract the max value of the relative error. Try to increase n to $n = 10000$ and $n = 10^5$. Make a table of the results and comment your results.

Compare your results with those from the LU decomposition codes for the matrix of sizes 10×10, 100×100 and 1000×1000. Here you should use the library functions provided on the webpage of the course. Alternatively, if you use armadillo as a library, you can use the similar function for LU decomposition. The armadillo function for the LU decomposition is called *LU* while the function for solving linear sets of equations is called *solve*. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Alternatively, you can use the functions in C++, Fortran or Python that measure the time used.

Make a table of the results and comment the differences in execution time How many floating point operations does the LU decomposition use to solve the set of linear equations? Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

To compute the elapsed time in c++ you can use the following statements

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 );
...
```

README before delivery:

Introduction to numerical projects

Here follows a brief recipe and recommendation on how to write a report for each project.

- Give a short description of the nature of the problem and the eventual numerical methods you have used.
- Describe the algorithm you have used and/or developed. Here you may find it convenient to use pseudocoding. In many cases you can describe the algorithm in the program itself.
- Include the source code of your program. Comment your program properly.
- If possible, try to find analytic solutions, or known limits in order to test your program when developing the code.
- Include your results either in figure form or in a table. Remember to label your results. All tables and figures should have relevant captions and labels on the axes.
- Try to evaluate the reliabilty and numerical stability/precision of your results. If possible, include a qualitative and/or quantitative discussion of the numerical stability, eventual loss of precision etc.
- Try to give an interpretation of you results in your answers to the problems.
- Critique: if possible include your comments and reflections about the exercise, whether you felt you learnt something, ideas for improvements and other thoughts you've made when solving the exercise. We wish to keep this course at the interactive level and your comments can help us improve it.
- Try to establish a practice where you log your work at the computerlab. You may find such a logbook very handy at later stages in your work, especially when you don't properly remember what a previous test version of your program did. Here you could also record the time spent on solving the exercise, various algorithms you may have tested or other topics which you feel worthy of mentioning.

Format for electronic delivery of report and programs

The preferred format for the report is a PDF file. You can also use DOC or postscript formats or as an ipython notebook file. As programming language we prefer that you choose between C/C++, Fortran2008 or Python. The following prescription should be followed when preparing the report:

- Use Devilry to hand in your projects, log in at `http://devilry.ifi.uio.no` with your normal UiO username and password and choose either 'fys3150' or 'fys4150'. There you can load up the files within the deadline.
- Upload **only** the report file! For the source code file(s) you have developed please provide us with your link to your github domain. The report file should include all of

your discussions and a list of the codes you have developed. Do not include library files which are available at the course homepage, unless you have made specific changes to them.

- In your git repository, please include a folder which contains selected results. These can be in the form of output from your code for a selected set of runs and input parameters.

- Comments from us on your projects, approval or not, corrections to be made etc can be found under your Devilry domain and are only visible to you and the teachers of the course.

Finally, we encourage you to work two and two together. Optimal working groups consist of 2-3 students. You can then hand in a common report.