FYS4150 - Project 1

Simen Nyhus Bastnes
simennb

19. September 2016

Abstract

In project 1, we looked closer at how we can numerically solve the Poisson equation by expressing it as a linear set of equations on the form $\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}$. We develop an algorithm that makes use of the fact that our problem has a tridiagonal matrix and the identical nature of many of the values, and compare it to more general algorithms. We find that our special case algorithm follows $\mathcal{O}(4n)$ flops, the general tridiag. algorithm $\mathcal{O}(9n)$, and LU-decomposition $\mathcal{O}(\frac{2}{3}n^3)$. This results in execution time differences from fractions of seconds to many minutes when n gets larger.

1 Introduction

In this project, we want to study how to solve the one-dimensional Poisson equation from electromagnetism with numerical methods. Doing this allows us to get a deeper understanding for how various vector and matrix operations, and dynamic memory allocation works in C++, as well as using some of the programs in the library package of the course.

After simplifying the equation slightly, we take a look at various methods for solving linear equation sets, and make an algorithm for a tridiagonal matrix. This we are able to compare to the more rigorous LU (*lower-upper*) decomposition method.

In our very specific problem, there exists an analytic solution, which allows us to see how the error changes depending on the step size we use for our numerical solution. Finally, we look at the execution time of the different algorithms to see how large the benefits are by using algorithms specifically tailored for the problem.

In this project I collaborated with Eirik Ramsli Hauge on the programming, but for reasons, he got the dead line extended, so we are delivering separate reports. Since the repository is hosted on his github domain, I created a fork of it on my account, so that a copy of the project in its current/finished (on my part) state exists.

2 Theory

In this project, we will be looking at the Poisson equation, a classical equation from electromagnetism, which in a spherically symmetric electrostatic potential reads like

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

which we can rewrite by substituting $\Phi(r) = \phi(r)/r$ as

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

letting $\phi \to u$ and $r \to x$, we can further rewrite the equation.

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

which is the equation we want to attempt to solve numerically. Specifically, we want to solve the one-dimensional Poisson equation (1) with Dirichlet boundary conditions.

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

In order to start solve this numerically, we discretize the equation by approximating u as v_i with grid points $x_i = ih$ in the interval $x_0 = 0$ to $x_{n+1} = 1$, with step length $h = 1/(n+1)$. The second derivative can then be approximated as

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

where $f_i = f(x_i)$. We can show that this equation can be rewritten as a linear set of equation of the form

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

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix representing the second derivative, on the form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix},$$

and $\tilde{b}_i = h^2 f_i$. We will look closer at multiple ways to solve this set of equations in the next subsections.

In the rest of this project, we will assume that the source term $f(x)$ is given by

$$f(x) = 100e^{-10x} \tag{2}$$

and that the analytical solution $u(x)$ is given by

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{3}$$

We will also be interested in the relative error between the analytical solution u_i and our numerical solution v_i, which we find by

$$\varepsilon_i = \log_{10} \left| \frac{v_i - u_i}{u_i} \right| \tag{4}$$

2.1 LU-decomposition

One way of solving a linear equation set is by first factorizing $\mathbf{A}$ into a lower diagonal matrix $\mathbf{L}$, and an upper diagonal matrix $\mathbf{U}$.

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

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n1} & \dots & l_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & 0 & u_{nn} \end{pmatrix}$$

For a quadratic matrix $\mathbf{A}$, this factorization exists if the determinant of $\mathbf{A}$ is different from zero.

Our equation set can then be written as

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

which we can split up into two equations on the form

$$\mathbf{Ly} = \tilde{\mathbf{b}} \quad , \quad \mathbf{Uv} = \mathbf{y}$$

which we can solved by using backwards substitution to find $\mathbf{y}$, and then to finally find $\mathbf{v}$. Typically, solving an equation set via LU-decomposition and backwards substitution goes as $\mathcal{O}(\frac{2}{3}n^3)$, so for large values of n, this is quite computationally heavy.

2.2 General tridiagonal solver

Since our matrix $\mathbf{A}$ is a tridiagonal matrix (and thus a lot of matrix elements are zero), we realise that it would be ideal to solve the equation set in a more efficient way, rather than having to compute the full LU decomposition.

First of all, we will be making a general algorithm to solve $\mathbf{Av} = \tilde{\mathbf{b}}$, where we do not assume that we have a matrix with the same elements along the diagonal and the non-diagonal elements. Our

equation set then looks like

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & 0 & \dots & \dots \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & & 0 & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}$$

To see more clearly what we are doing, we will look at the case $n = 4$

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

and renaming $\tilde{b}_i$ to p_i to avoid some confusion. We recall that if we could get $\mathbf{A}$ on the form of an upper- or lower diagonal matrix, we could solve the equation set by backwards substitution. Luckily, we can easily transform this to an upper diagonal matrix by forward substitution. To remove a_1 from the matrix, we do the following operation

$$\begin{aligned} p_2^* &= p_2 - p_1 \cdot \frac{a_1}{b_1} \\ &= a_1 v_1 + b_2 v_2 + c_2 v_3 - \frac{a_1}{b_1}(b_1 v_1 + c_1 v_2) \\ &= \cancel{a_1 v_1} + b_2 v_2 + c_2 v_3 - \cancel{a_1 v_1} - \frac{c_1 a_1}{b_1} v_2 \\ &= v_2(b_2 - \frac{c_1 a_1}{b_1}) + c_3 v_3 = b_2^* v_2 + c_3 v_3 \end{aligned}$$

where $b_2^* = b_2 - \frac{a_1 c_1}{b_1}$. Now our matrix looks like

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b_2^* & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{pmatrix}$$

Repeating this, we can remove all the a_i's from the matrix, giving us the algorithm for forward substitution

Algorithm 1 Forward substitution

1: $b_1^* = b_1$
2: $p_1^* = p_1$
3: **for** $i = 2$, n **do**
4: $\quad b_i^* = b_i - a_{i-1} \cdot c_{i-1}/b_{i-1}^*$
5: $\quad p_i^* = p_i - p_{i-1}^* \cdot a_i/b_{i-1}^*$
6: **end for**

Now, we can substitute backwards, starting from n to find v_i

Algorithm 2 Backward substitution

$v_n = p_n^*/b_n^*$
for $i = n-1,\ 1$ **do**
    $v_i = (p_i^* - c_i \cdot v_{i+1})/b_i^*$
end for

Looking at the algorithms, we can count the number of floating point operations. In the forward substitution, we have 6 floating point operations, and 3 in the backward substitution, giving us $9n$ FLOPS.

2.3 Special tridiagonal solver

In the previous subsection, we found a general algorithm for solving an equation set with any tridiagonal matrix. And while this algorithm is considerably better than full LU-decomposition, we can use the fact that our matrix has identical matrix elements along the diagonal, and identical (but different) values for the non-diagonal elements.

Since all $a_i = c_i = -1$ and $b_i = 2$, we can update the diagonal elements in the matrix to be

$$d_i = \frac{i+1}{i}, \quad \text{where } d_0 = d_n = 2$$

and can be calculated beforehand, so that the full specialized algorithm ends up as

Algorithm 3 Specialized algorithm

$p_1^* = p_1$
for $i = 2,\ n$ **do** ▷ Forward substitution
    $p_i^* = p_i + p_{i-1}^*/d_{i-1}$
end for
$v_n = p_n^*/d_n$
for $i = n-1,\ 1$ **do** ▷ Backwards substitution
    $v_i = (p_i^* + v_{i+1})/d_i$
end for

Counting the number of floating point operations here, we get $4n$ FLOPS, which is a decent improvement over the general algorithm ($9n$ FLOPS).

2.4 Implementation

The programs where the algorithms discussed earlier have been implemented, and used to generate the results in the next section lies in the GitHub repository (link also in reference [1])

https://github.com/simennb/FYS4150-1/tree/master/project1

The C++ programs lies within the `src` folder, and the algorithms used to solve the problem are implemented in `solver.cpp`

When running, the project is divided into three segments, and which one you intend to run needs to be specified by command line arguments.

When specifying $\log_{10}(n)$, the program loops through all the powers up to the one specified, writing the desired results to file.

3 Results and discussion

In this section we will be looking at the results from implementing the algorithms discussed earlier.

3.1 Numerical accuracy

To see how well our numerical approximation works, we plot the analytical solution u_i and our approximation v_i for different values of n.

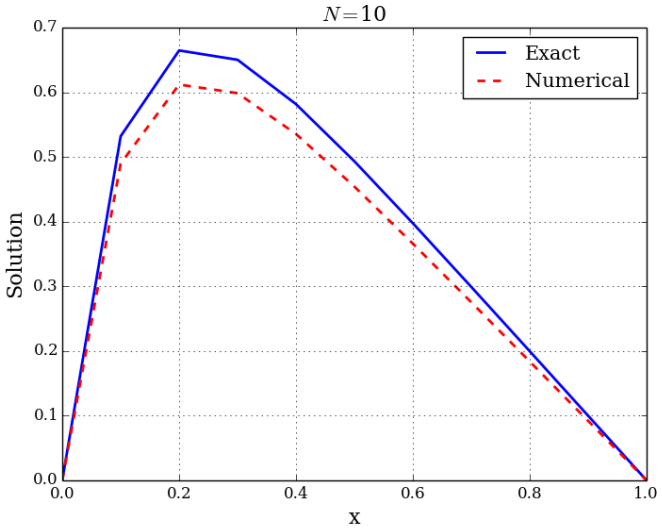
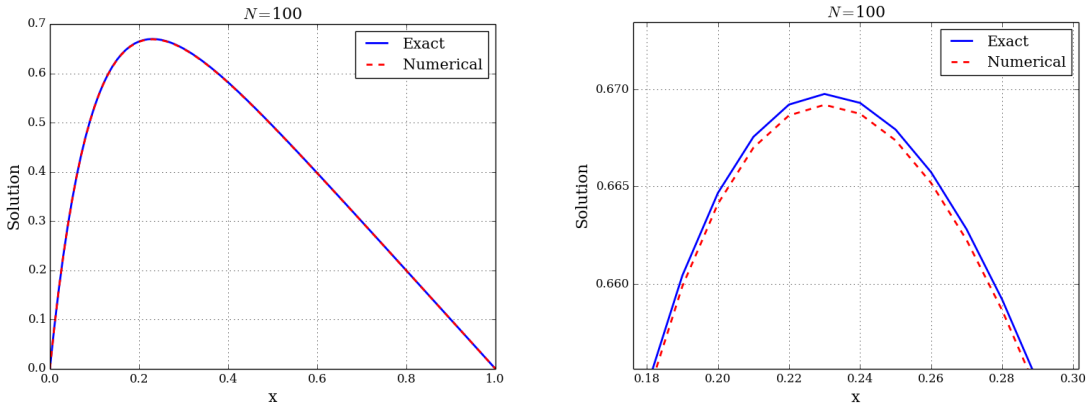


Figure 1: Numerical and analytical solution plotted for $N = 10$



(a) Numerical and analytical solution

(b) Numerical and analytical, zoomed in

Figure 2: Numerical and analytical solution plotted both with $N = 100$

As we see for figure 1, the approximation is quite crude at low values of n, which is not surprising. However, just going up to $n = 100$ (as shown in figure 2a), makes the error decrease significantly. Going to even higher values of n, we would assume the error to decrease even further, but at some point the round-off error should start affecting the solution. Rather than showing how the error behaves in plots, we make a table of the relative error (equation (4)).

Table 1: Table of step length $\log_{10}(h)$ and maximal error ε

$\log_{10}(h)$	Rel. error
-1	-1.1006
-2	-3.0794
-3	-5.0792
-4	-7.0793
-5	-9.0049
-6	-6.7714

In table 1, we see how the error changes as we decrease the step length h. We observe that at $\log_{10}(h) = -5$, the error reaches a minimum, and increases again as we decrease the step length. From this, we can assume that this is the ideal step length for our numerical solution, though more values of h could have been checked.

Setting $\log_{10}(h)$ to -7 however causes the error to go to $\varepsilon = -13.53$, and why this happens, or what it means is unclear to me.

3.2 Comparing the different algorithms

Another important factor in evaluating how good our algorithm works, is how long they take to execute. From earlier, we found that the number of floating point operations go as $\mathcal{O}(\frac{2}{3}n^3)$ for

the LU-decomposition, $\mathcal{O}(9n)$ for the general tridiagonal algorithm, and $\mathcal{O}(4n)$ for the specialized algorithm.

Table 2: Execution time for the different algorithms described earlier. For n larger than 10^4, there are no results for LU-decomposition.

$\mathbf{log_{10}(n)}$	**General [sec]**	**Special [sec]**	**LU-decomp. [sec]**
1	$2.00 \cdot 10^{-6}$	$1.00 \cdot 10^{-6}$	$1.30 \cdot 10^{-5}$
2	$4.00 \cdot 10^{-6}$	$3.00 \cdot 10^{-6}$	0.0023
3	$3.60 \cdot 10^{-5}$	$2.70 \cdot 10^{-5}$	1.5171
4	$3.69 \cdot 10^{-4}$	$2.88 \cdot 10^{-4}$	1980
5	0.0033	0.0025	-
6	0.0343	0.0283	-

From table 2 we observe that the execution time seems to roughly follow our FLOPS calculation. As expected, the full LU-decomposition takes considerably more time than any of the other methods. Furthermore, the LU-decomposition uses a lot more memory (as compared to the other), since it also stores a lot of zeros, and if we were to set $n = 10^5$, it would use a lot memory to store all the values. For a $10^5 \times 10^5$ matrix filled with doubles, this would be around 80 GB, which is a bit more than the standard of 8 GB most PCs have today. (the computer used while running this project has 4 GB).

What this shows, is that there is quite a lot to gain, both memory and execution time wise by using an algorithm specifically tailored for the problem you are solving.

References

[1] Project 1 GitHub repository
https://github.com/simennb/FYS4150-1/tree/master/project1