FYS4150 - Computational Physics - Project 1

Eimund Smestad
Email: eimundsm@fys.uio.no

8th September, 2014

Abstract

This report looks at different algorithms to solve the one-dimensional Poisson's equation with Dirichlet boundary condition. The algorithms are compared by speed and floating point error in the solution. The results show how different ways of writing code effect speed and floating point error, and discusses how the machines architecture and functionally are the reason for the performance difference.

1 Theory

We have the Poisson's equation from electromagnetism stated as

$$\nabla^2\Phi = -4\pi\rho\left(\vec{r}\right)$$

in three dimensional space with the electrostatic potential Φ generated by local charge distribution $\rho\left(\vec{r}\right)$. With a spherically symmetric Φ and $\rho\left(\vec{r}\right)$ the equations simplifies to a one-dimensional equation in r, namely

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

which can be rewritten via a substitution $\Phi\left(r\right) = \frac{\phi}{r}$ as

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2} = -4\pi\rho\left(r\right) \ ,$$

which again can be rewritten to

$$-u''\left(x\right) = f\left(x\right) \ .$$

In this project I will solve the following case of the one-dimensional Poisson's equation with Dirichlet boundary condition

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

2 Numerical approximation

Equation (1) can be approximated by discretizing u as v_i with grid points $x_i = ih$ with the interval $x_0 = 0$ and $x_{n+1} = 1$;

$$-v_{i+1} + 2v_i - v_{i-1} = b_i \qquad \text{for } v_0 = 0, v_{n+1} = 0 \text{ and } i \in \mathbb{N}_1^n, \tag{2}$$

where $b_i = h^2 f(x_i)$ and the step length is given by $h = \frac{1}{n+1}$. This difference equation can be rewritten to matrix form

$$\vec{\vec{A}}\vec{v} = \vec{b}$$

which we can show to satisfy the difference equation (2)

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^n a_{1,j}v_j \\ \sum_{j=1}^n a_{2,j}v_j \\ \vdots \\ \sum_{j=1}^n a_{n,j}v_j \end{bmatrix} = \begin{bmatrix} a_{1,1}v_1 + a_{1,2}v_2 \\ a_{2,1}v_1 + a_{2,2}v_2 + a_{2,3}v_3 \\ \vdots \\ a_{n,n-1}v_{n-1} + a_{n,n}v_n \end{bmatrix} = \begin{bmatrix} 2v_1 - v_2 \\ -v_1 + 2v_2 - v_3 \\ \vdots \\ -v_{n-1} + 2v_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

where the matrix elements are given by $a_{i,i} = 2$, $a_{i,i\pm 1} = -1$ and $a_{i,i\pm j} = 0$ for $j \geq 2$, and this is called a tridiagonal matrix. Please note the boundary condition in (1) gives $-v_0 + 2v_1 - v_2 = 2v_1 - v_2$ and $-v_{n-1} + 2v_n - v_{n+1} = -v_{n-1} + 2v_n$, which actually enables us to write the difference equation in (2) on matrix form without getting in trouble in the corners of the matrix.

2.1 LU decomposition

Gaussian elimination is used to solve linear equations, but if we use the same matrix A to solve for different source terms b it is smarter to decompose the matrix A to a upper and a lower triangle matrix with LU decomposition. The LU decomposition is independent of the source term b and needs only forward and backward substitution to solve the linear equation. Both Gaussian elimination and LU decomposition use $\sim \frac{2}{3}n^3$ FLOPS to be solved; the difference is that the next solution with the same matrix A the Gaussian elimination still needs $\frac{2}{3}n^3$ FLOPS, whereas we already have the LU decomposition and only need forward and backward substitution to find the solution, and forward and backward substitution need $O\left(n^2\right)$ FLOPS to be solved (due to double looping).

For the LU decomposition I have used armadillo and made my own forward and backward substitution solver. Armadillo uses the Doolittle algorithm which gives 1's on the diagonal $l_{i,i}$ on the lower triangle matrix;

$$\vec{\vec{A}} = \vec{\vec{L}}\vec{\vec{U}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{bmatrix}.$$

The forward substitution is done by

$$\vec{\vec{U}}\vec{v} = \vec{\vec{L}}^{-1}\vec{b} \sim \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & b_1 \\ l_{2,1} & 1 & \cdots & 0 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 & b_n \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \check{b}_1 \\ l_{2,1} & 1 & \cdots & 0 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 & b_n \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \check{b}_1 \\ 0 & 1 & \cdots & 0 & b_2 - l_{2,1}\check{b}_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 & b_n \end{array}\right]$$

$$= \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \check{b}_1 \\ 0 & 1 & \cdots & 0 & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 & b_n \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \check{b}_1 \\ 0 & 1 & \cdots & 0 & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & b_n - \sum_{i=1}^{n-1} l_{n,i}\check{b}_i \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \check{b}_1 \\ 0 & 1 & \cdots & 0 & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \check{b}_n \end{array}\right],$$

which gives the following algorithm for the forward substitution form Doolittle

$$\check{b}_i = b_i - \sum_{j=1}^{i-1} l_{i,j}\check{b}_j \qquad \text{with } \check{b}_1 = b_1. \tag{3}$$

The last step is to do the backward substitution

$$\begin{aligned}
\vec{v} = \vec{\vec{U}}^{-1}\vec{\vec{L}}^{-1}\vec{b} &\sim \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \check{b}_1 \\ 0 & u_{2,2} & \cdots & u_{2,n} & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & u_{n,n} & \check{b}_n \end{array}\right] \sim \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \check{b}_1 \\ 0 & u_{2,2} & \cdots & u_{2,n} & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \frac{\check{b}_n}{u_{n,n}} \end{array}\right] = \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \check{b}_1 \\ 0 & u_{2,2} & \cdots & u_{2,n} & \check{b}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & v_n \end{array}\right] \\
&\sim \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \check{b}_1 \\ 0 & 1 & \cdots & 0 & \frac{\check{b}_2 - \sum_{i=3}^{n} u_{2,i}v_i}{u_{2,2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & v_n \end{array}\right] = \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \check{b}_1 \\ 0 & 1 & \cdots & 0 & v_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & v_n \end{array}\right] \\
&\sim \left[\begin{array}{cccc|c} u_{1,1} & u_{1,2} & \cdots & u_{1,n} & \frac{\check{b}_1 - \sum_{i=2}^{n} u_{1,i}v_i}{u_{1,1}} \\ 0 & 1 & \cdots & 0 & v_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & v_n \end{array}\right] = \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & v_1 \\ 0 & 1 & \cdots & 0 & v_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & v_n \end{array}\right],
\end{aligned}$$

which gives the following algorithm for the backward substitution

$$v_i = \frac{\check{b}_i - \sum_{j=i+1}^{n} u_{i,j}v_j}{u_{i,i}} \qquad \text{with } v_n = \check{b}_n. \tag{4}$$

2.2 Tridiagonal matrix solver

A tridiagonal matrix can be solved faster than LU decomposition method, even when only doing the forward and backward substitution. We apply the Gaussian elimination to first reduce the tridiagonal matrix to the upper triangle matrix;

$$
\left[\begin{array}{ccccccc|c}
a_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & b_1 \\
a_{2,1} & a_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & b_2 \\
0 & a_{3,2} & a_{3,3} & \cdots & 0 & 0 & 0 & b_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & a_{n-2,n-2} & a_{n-2,n-1} & 0 & b_{n-2} \\
0 & 0 & 0 & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} & b_{n-1} \\
0 & 0 & 0 & \cdots & 0 & a_{n,n-1} & a_{n,n} & b_n
\end{array}\right]
$$

$$
\sim \left[\begin{array}{ccccccc|c}
a_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & b_1 \\
0 & a_{2,2} - \frac{a_{1,2}a_{2,1}}{a_{1,1}} & a_{2,3} & \cdots & 0 & 0 & 0 & b_2 - \frac{a_{1,2}b_1}{a_{1,1}} \\
0 & a_{3,2} & a_{3,3} & \cdots & 0 & 0 & 0 & b_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & a_{n-2,n-2} & a_{n-2,n-1} & 0 & b_{n-2} \\
0 & 0 & 0 & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} & b_{n-1} \\
0 & 0 & 0 & \cdots & 0 & a_{n,n-1} & a_{n,n} & b_n
\end{array}\right]
$$

$$
= \left[\begin{array}{ccccccc|c}
\check{a}_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & \check{b}_1 \\
0 & \check{a}_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & \check{b}_2 \\
0 & a_{3,2} & a_{3,3} & \cdots & 0 & 0 & 0 & b_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & a_{n-2,n-2} & a_{n-2,n-1} & 0 & b_{n-2} \\
0 & 0 & 0 & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} & b_{n-1} \\
0 & 0 & 0 & \cdots & 0 & a_{n,n-1} & a_{n,n} & b_n
\end{array}\right]
$$

$$
\sim \left[\begin{array}{ccccccc|c}
\check{a}_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & \check{b}_1 \\
0 & \check{a}_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & \check{b}_2 \\
0 & 0 & \check{a}_{3,3} & \cdots & 0 & 0 & 0 & \check{b}_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \check{a}_{n-2,n-2} & a_{n-2,n-1} & 0 & \check{b}_{n-2} \\
0 & 0 & 0 & \cdots & 0 & \check{a}_{n-1,n-1} & a_{n-1,n} & \check{b}_{n-1} \\
0 & 0 & 0 & \cdots & 0 & 0 & \check{a}_{n,n} & \check{b}_n
\end{array}\right].
$$

So the elements in the upper triangle matrix of reduced tridiagonal are given by the following two difference equations

$$\check{b}_i = b_i - \frac{a_{i,i-1}}{\check{a}_{i-1,i-1}}\check{b}_{i-1} \qquad \text{for } \check{b}_1 = b_1 \text{ and } i \in \mathbb{N}_2^n, \tag{5}$$

$$\check{a}_{i,i} = a_{i,i} - \frac{a_{i,i-1}}{\check{a}_{i-1,i-1}}a_{i,i-1} \qquad \text{for } \check{a}_{1,1} = a_{1,1} \text{ and } i \in \mathbb{N}_2^n, \tag{6}$$

$$a_{i,i+j} = 0 \qquad \text{for } j \in \mathbb{Z}_{-n}^{n}/\mathbb{Z}_{-1}^{0}. \tag{7}$$

If we now continue with Gaussian elimination to reduce tridiagonal further to a diagonal matrix

$$
\left[\begin{array}{ccccccc|c}
\breve{a}_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & \breve{b}_1 \\
0 & \breve{a}_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & \breve{b}_2 \\
0 & 0 & \breve{a}_{3,3} & \cdots & 0 & 0 & 0 & \breve{b}_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \breve{a}_{n-2,n-2} & a_{n-2,n-1} & 0 & \breve{b}_{n-2} \\
0 & 0 & 0 & \cdots & 0 & \breve{a}_{n-1,n-1} & a_{n-1,n} & \breve{b}_{n-1} \\
0 & 0 & 0 & \cdots & 0 & 0 & \breve{a}_{n,n} & \breve{b}_n
\end{array}\right]
$$

$$
\sim \left[\begin{array}{ccccccc|c}
\breve{a}_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & \breve{b}_1 \\
0 & \breve{a}_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & \breve{b}_2 \\
0 & 0 & \breve{a}_{3,3} & \cdots & 0 & 0 & 0 & \breve{b}_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \breve{a}_{n-2,n-2} & a_{n-2,n-1} & 0 & \breve{b}_{n-2} \\
0 & 0 & 0 & \cdots & 0 & \breve{a}_{n-1,n-1} & 0 & \breve{b}_{n-1} - \frac{a_{n-1,n}}{\breve{a}_{n,n}}\breve{b}_n \\
0 & 0 & 0 & \cdots & 0 & 0 & \breve{a}_{n,n} & \breve{b}_n
\end{array}\right]
$$

$$
= \left[\begin{array}{ccccccc|c}
\breve{a}_{1,1} & a_{1,2} & 0 & \cdots & 0 & 0 & 0 & \breve{b}_1 \\
0 & \breve{a}_{2,2} & a_{2,3} & \cdots & 0 & 0 & 0 & \breve{b}_2 \\
0 & 0 & \breve{a}_{3,3} & \cdots & 0 & 0 & 0 & \breve{b}_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \breve{a}_{n-2,n-2} & a_{n-2,n-1} & 0 & \breve{b}_{n-2} \\
0 & 0 & 0 & \cdots & 0 & \breve{a}_{n-1,n-1} & 0 & \hat{b}_{n-1} \\
0 & 0 & 0 & \cdots & 0 & 0 & \breve{a}_{n,n} & \hat{b}_n
\end{array}\right]
$$

$$
\sim \left[\begin{array}{ccccccc|c}
\breve{a}_{1,1} & 0 & 0 & \cdots & 0 & 0 & 0 & \hat{b}_1 \\
0 & \breve{a}_{2,2} & 0 & \cdots & 0 & 0 & 0 & \hat{b}_2 \\
0 & 0 & \breve{a}_{3,3} & \cdots & 0 & 0 & 0 & \hat{b}_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \breve{a}_{n-2,n-2} & 0 & 0 & \hat{b}_{n-2} \\
0 & 0 & 0 & \cdots & 0 & \breve{a}_{n-1,n-1} & 0 & \hat{b}_{n-1} \\
0 & 0 & 0 & \cdots & 0 & 0 & \breve{a}_{n,n} & \hat{b}_n
\end{array}\right]
$$

which gives the difference equation

$$
\hat{b}_i = \breve{b}_i - \frac{a_{i,i+1}}{\breve{a}_{i+1,i+1}}\hat{b}_{i+1} = \breve{b}_i - a_{i,i+1}v_{i+1} \qquad \text{for } \hat{b}_n = \breve{b}_n \text{ and } i \in \mathbb{N}_1^{n-1}. \tag{8}
$$

where v_i is the solution of the linear equation set represented by a tridiagonal matrix

$$
v_i = \frac{\hat{b}_i}{\breve{a}_{i,i}} \qquad \text{for } i \in \mathbb{N}_1^n. \tag{9}
$$

So the solution for a linear equation set represented by a tridiagonal matrix in (9) needs iteratively calculate first (6) and then (5) when i goes $1 \rightarrow n$, and then iteratively calculate (8) when i goes $n \rightarrow 1$. The difference equation (6) requires 3 FLOPS to be calculated for each iteration, but (5) requires only 2 FLOPS to be calculated for each iteration, because $\frac{a_{i-1,i}}{\breve{a}_{i-1,i-1}}$ is used in both (5) and (6) and needs only to be calculated once. Then (9) is calculated and requires a 1 FLOP, which is followed directly by (8) which uses (9) and therefore requires only 2 more FLOPS to be calculated. In total a general linear equation set of n equations that are represented by a tridiagonal matrix requires approximately $8n$ FLOPS to be solved. Which is much faster than Gaussian elimination and LU decomposition.

If we use the tridiagonal matrix generated by (2) we can simplify (5), (6), (8) and (9) accordingly

$$\check{b}_i = b_i + \frac{\check{b}_{i-1}}{\check{a}_{i-1,i-1}} = b_i + \frac{i-1}{i}\check{b}_{i-1} \qquad \text{for } \check{b}_1 = b_1 \text{ and } i \in \mathbb{N}_2^n, \tag{10}$$

$$\check{a}_{i,i} = 2 - \frac{1}{\check{a}_{i-1,i-1}} = \frac{i+1}{i} \qquad \text{for } \check{a}_{1,1} = 2 \text{ and } i \in \mathbb{N}_2^n, \tag{11}$$

$$\hat{b}_i = \check{b}_i + v_{i+1} \qquad \text{for } \hat{b}_n = \check{b}_n \text{ and } i \in \mathbb{N}_1^{n-1}, \tag{12}$$

$$v_i = \frac{\hat{b}_i}{\check{a}_{i,i}} = \frac{i}{i+1}\hat{b}_i \qquad \text{for } i \in \mathbb{N}_1^n. \tag{13}$$

We can show (11) by induction. Clearly (11) satisfy the initial condition $a_{1,1} = \lim_{i\to 1}\frac{i+1}{i} = 2$. The induction step can be shown as

$$\check{a}_{i+1,i+1} = 2 - \frac{1}{\check{a}_{i,i}} = 2 - \frac{i}{i+1} = \frac{2(i+1)-i}{i+1} = \frac{i+2}{i+1}.$$

Since (11) is actually a constant that doesn't change with the source terms b_i, it can be pre calculated and reused for solving the tridiagonal matrix. In that case (10) only needs 2 FLOPS to be calculated, (11) is pre calculated, (12) and (13) needs 1 FLOP each, and the total is $4n$ FLOPS to solve the linear equation in (2). If we don't want to use memory to save the $\frac{i+1}{i}$ coefficient, we need to add 2 FLOPS; 1 FLOP for calculate $\frac{i-1}{i}$ when i goes $1 \to n$, and another FLOP to calculate $\frac{i}{i+1}$ when i goes $n \to 1$. And this gives a $6n$ FLOPS to solve.

3 Implementation

In this section different algorithms are presented for solving the tridiagonal matrix generated of (2). Only the actual solver will be presented here, and not the pre calculation needed. These algorithms are compared to each other in section Result 4. The algorithms can be found in Matrix.h.

3.1 $8n$

This method solves a general tridiagonal matrix. It requires one matrix of size n and two matrix's of size $n-1$ to represent the tridiagonal matrix, but no extra memory of significance to solve the tridiagonal matrix.

Listing 1: `Matrix<MatrixType::TridiagonalGeneral, T>::Solve`

```
inline static bool Solve(T* a, T* b, T* c, T* f, unsigned int n) {
        T temp;
        T* start = f;
        T* end = &f[n-1];
        while(f != end) {
                temp = (*a++)/(*b++);
                *b -= temp*(*c++);          // Eq (6)
                *f -= temp*(*f++);           // Eq (5)
        }
        while(f != start) {
                *f /= (*b--);                // Eq (9)
                *f -= (*--c)*(*f--);         // Eq (8)
        }
        *f /= *b;
        return true;
}
```

3.2 $6n$

This method uses actually $8n$ FLOPS, but it's based on the $6n$ algorithm derived in section 2.2. The reason for number of FLOPS is speed consideration, which you can see in Table 4.1. The reason for this is if we use integers for i in $\frac{i+1}{i}$, we don't count the increment of i as a FLOP. However in this algorithm i is a floating point and therefore we need to count the increment of i as a FLOP. This method needs no memory to represent the tridiagonal matrix, and need no extra memory of significance to solve the tridiagonal matrix.

Listing 2: `Matrix<MatrixType::Tridiagonal_minus1_2_minus1_6n, T>::Solve`

```
static bool Solve(T* f, unsigned int n) {
        T* start = f;
        T* end = &f[n-1];
        T i = 1;
        while(f != end)
                *f += i++/i*(*f++);       // Eq (10)
        i++;
        while(f != start) {
                *f *= 1/(i--)*i;// Eq (13) (faster than *f /= i--/i)
                *f += *f--;     // Eq (12)
        }
        *f /= 2;
        return true;
}
```

3.3 $6n$ cutoff

If we look at the factor $\frac{i+1}{i} \to 1$ when $i \to \infty$, and it's tempting to approximate $\frac{i+1}{i} = 1$ for $i \geq$ cutoff. However this approximation shows to give fatal error as shown in Table 4.2. Otherwise this method is the same as $6n$ algorithm in 3.2.

Listing 3: `Matrix<MatrixType::Tridiagonal_minus1_2_minus1_6n, T>::Solve`

```
static bool Solve(T* f, unsigned int n, unsigned int cutoff) {
        T* start = f;
        T* mid = &f[n < cutoff ? n-1 : cutoff-1];
        T* end = &f[n-1];
        T i = 1;
        while(f != mid)
```

```
                *f += i++/i*(*f++);        // Eq (10)
        while(f != end)
                *f += *f++;                // Eq (10) with (i+1)/i = 1
        i++;
        while(f != mid)
                *f += *f--;                // Eq (12) with (i+1)/i = 1
        while(f != start) {
                *f *= 1/(i--)*i;           // Eq (13)
                *f += *f--;                // Eq (12)
        }
        *f /= 2;
        return true;
}
```

3.4 $6n$ Int

This method uses `unsigned int` as intrinsic data type for i in the coefficient $\frac{i+1}{i}$ instead of a floating point, as discussed in section 3.2. Otherwise this method is the same as $6n$ algorithm in 3.2, and we see in Table 4.1 how this small change in code double the computing time.

Listing 4: `Matrix<MatrixType::Tridiagonal_minus1_2_minus1_6n, T>::SolveInt`

```
static bool SolveInt(T* f, unsigned int n) {
        T* start = f;
        T* end = &f[n-1];
        unsigned int i = 1;
        while(f != end)
                *f += (T)(i++)/i*(*f++);   // Eq (10)
        i++;
        while(f != start) {
                *f *= (T)1/(i--)*i;          // Eq (13)
                *f += *f--;                  // Eq (12)
        }
        *f /= 2;
        return true;
}
```

3.5 $6n$ True

This method doesn't use induction proof in section 2.2 to get the factor $\frac{i+1}{i}$, but uses the difference equation in (10), (11), (12) and (13) directly, which gives $6n$ FLOP algorithm. This method does not need memory to represent the tridiagonal matrix, however it needs an array of size n to store the values from (11). And we see in Table 4.1 how this extra memory accessing doubles the computing time. Another disadvantage is that this method is not as numerical stable for larger n.

Listing 5: `Matrix<MatrixType::Tridiagonal_minus1_2_minus1_6n, T>::SolveTrue`

```
public: bool SolveTrue(T* f, unsigned int n) {
        if(_n < n)
                this->n = n;
        b[0] = 2;
        T temp;
        T* start = f;
        T* end = &f[n-1];
        while(f != end) {
                temp = (T)1/(*b++);
                *b = 2 - temp;              // Eq (11)
                *f += temp*(*f++);          // Eq (10)
        }
        while(f != start) {
                *f /= (*b--);               // Eq (13)
                *f += (*f--);               // Eq (12)
        }
        *f /= *b;
        return true;
}
```

3.5.1 $4n$ and $5n$

$5n$ is the same method as $4n$, only that $5n$ calculates the coefficient $\frac{i+1}{i}$ when solving the tridiagonal matrix, whereas $4n$ method pre calculates this values. These coefficients $\frac{i+1}{i}$ are stored in an array of size n, other wise this method is equal to $6n$ method in section 3.2, no memory need to represent the tridiagonal matrix. Even though $4n$ method has half the number of FLOPS needed to calculate as $6n$ (which is actually $8n$ FLOPS), we see in Table 4.1 that the speed is more or less the same. The reason is that $6n$ method has less memory access than the $4n$, and can do more FLOPS in CPU instead of waiting for memory.

Listing 6: `Matrix<MatrixType::Tridiagonal_minus1_2_minus1_4n, T>::Solve`

```
bool Solve(T* f, unsigned int n) {
        if(_n < n)
                this->n = n;
        T* start = f;
        T* end = &f[n-1];
        T* factor = this->factor;
        while(f != end)
                *f += (*f++)*(*factor++);   // Eq (10)
        while(f != start) {
                *f *= *factor--;            // Eq (13)
                *f += *f--;                 // Eq (12)
        }
        *f /= 2;
        return true;
}
```

3.6 Armadillo LU

This method uses the LU decomposition method discussed in section 2.1. Armadillo is used to find the lower and upper triangle matrix's L and U by the function `lu(L, U, matrix)` prior to finding the solution, and are therefore not included in the timing in Table 4.1. As we see from this table with $n = 1000$ the computation speed is approximately 1000 slower than the other methods, which is in accordance with that the forward and backward substitution uses $O\left(n^2\right)$ FLOPS. This method is very memory demanding, it needs three $n \times n$ arrays, on to represent the tridiagonal matrix and one each for both lower and upper triangle matrix. And for $n = 100000$ we need 80 GB of memory to represent a matrix with intrinsic data type `double`, this is impossible since the matrix indexing is represented by the intrinsic data type `int`, which is 32 bit and can only represent a array size of $2^{31} \approx 2.1$ GB.

Listing 7: `Matrix<MatrixType::LU_decomposition, T>::Solve`

```
bool Solve(T* f, unsigned int n) {
        if(n == _n) {
                T* f_;
                int i = 0, j;
                while(++i < n) {          // Forward solve Ly = f
                        f_ = &f[i];
                        for(j = 0; j < i; j++)
                                *f_ -= L(i,j)*f[j];
                }
                n--;
                while(i--) {              // Backward solve Ux = y
                        f_ = &f[i];
                        for(j = n; j > i; j--)
                                *f_ -= U(i,j)*f[j];
                        *f_ /= U(i,i);
                }
                return true;
        }
        return false;
}
```

4 Result

The algorithm described in section 3 is tested with the source term

$$f(x) = 100\mathrm{e}^{-10x} \tag{14}$$

which has the analytic solution

$$u(x) = 1 - \left(1 - \mathrm{e}^{-10}\right)x - \mathrm{e}^{-10x}, \tag{15}$$

which one can easily see by taking the derivative of this expression two times, and we get $eq_1 4$ and it also satisfy the boundary conditions $u(0) = u(1) = 0$. And all the algorithms are tested against this analytical solution in the following results. The relative error of the algorithms to the analytic solution (15) is calculated by

$$\epsilon_i = log_{10}\left|\frac{v_i - u_i}{u_i}\right|. \tag{16}$$

Number of steps	8n	6n	6n cutoff (10000)	6n int	6n true	5n	4n	Armadillo LU
10	0	0	0	0	0	0	1e-06	2e-06
100	2e-06	1e-06	1e-06	2e-06	2e-06	2e-06	1e-06	0.000102
1000	2.1e-05	9e-06	9e-06	1.9e-05	1.8e-05	1.3e-05	9e-06	0.00952
10000	0.000223	9.7e-05	8.8e-05	0.000189	0.000194	0.000141	8.7e-05	-
100000000	2.13964	0.889835	0.478423	1.89765	1.92346	1.29643	0.890628	-

Table 4.1: *Time used in seconds to solve the tridiagonal matrix with different solvers .*

Number of steps	8n	6n	6n cutoff (10000)	6n int	6n true	5n	4n	Armadillo LU
10	-1.1797	-1.1797	-1.1797	-1.1797	-1.1797	-1.1797	-1.1797	-1.1797
100	-3.08804	-3.08804	-3.08804	-3.08804	-3.08804	-3.08804	-3.08804	-3.08804
1000	-5.08005	-5.08005	-5.08005	-5.08005	-5.08005	-5.08005	-5.08005	-5.08005
10000	-7.07929	-7.07927	-7.07927	-7.07927	-7.07929	-7.07927	-7.07927	-
100000000	-2.88229	-10.2427	3.99967	-10.2427	-2.88229	-10.2414	-10.2414	-

Table 4.2: *Order of maximum relative error for the different solvers.*

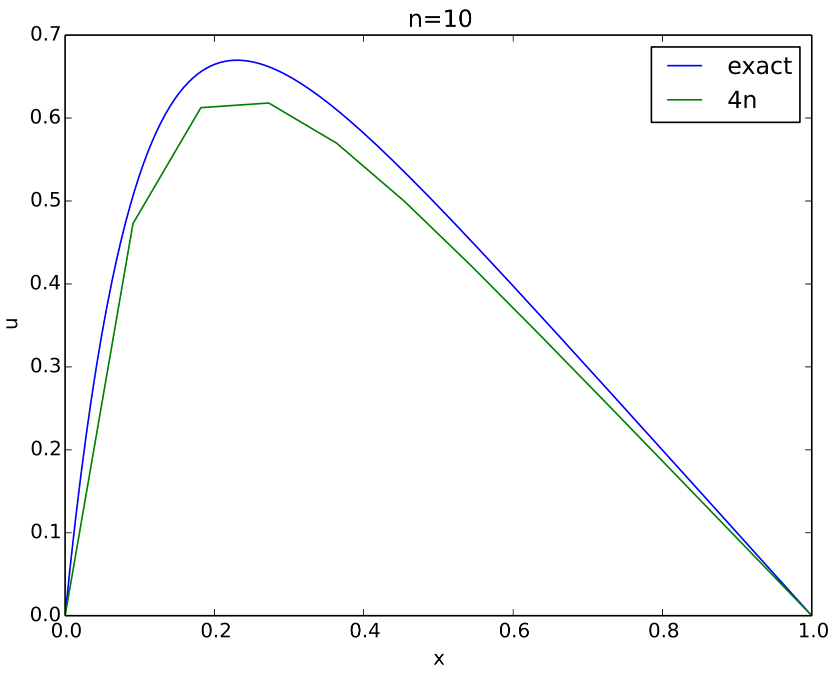

Figure 4.1: *Result from different tridiagonal matrix solvers for* $n = 10$ *steps compared to the closed form solution.*

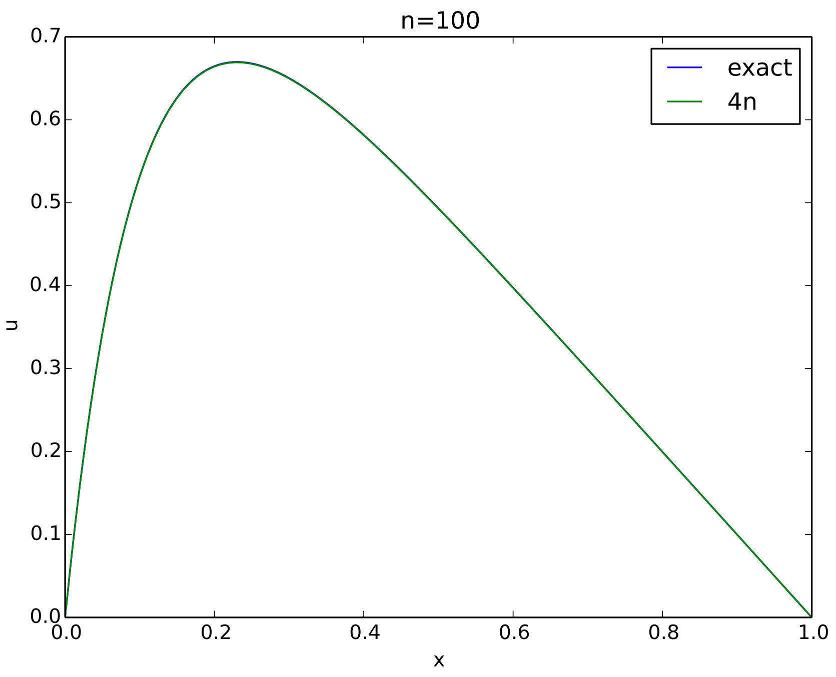

Figure 4.2: *Result from different tridiagonal matrix solvers for $n = 100$ steps compared to the closed form solution.*

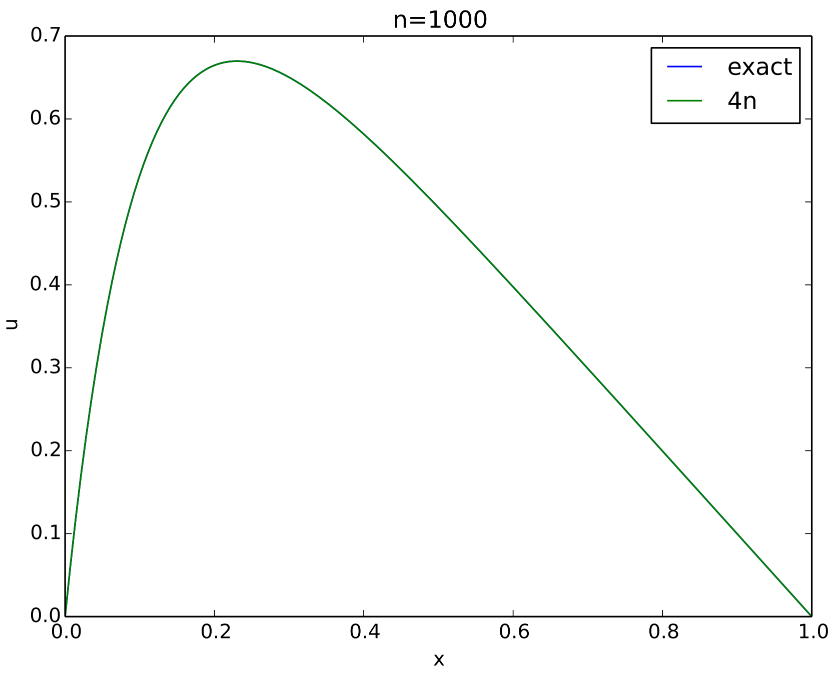

Figure 4.3: *Result from different tridiagonal matrix solvers for $n = 1000$ steps compared to the closed form solution.*

5 Conclusion

This reports shows that there are many aspects to consider when optimising an algorithm for a specific problem. A generic algorithms for solving problems are often slow because lot of FLOPS are need to be calculated to solve a problem to be able to handle every possible case. However if a problem have simplified properties to the general case, very often a fast algorithm can be devised to solve the problem, because fewer considerations are need and hence less computation time is needed. An indeed a simplified algorithm may be able to handle larger datasets than the general algorithm, due to difference in memory usage.

Number of FLOPS is often a good indication of computation time needed when we look at the order of FLOPS needed from n data points. However when we have few FLOPS difference between algorithm for each data point, then it's not as obvious which algorithm is the fastest. Memory accessing, overhead code and data type conversion are additional considerations when comparing algorithms speed with few FLOPS difference. Avoiding memory accessing may be beneficial because the RAM runs on a lower speed than the CPU. Table 4.1 shows very well how this different aspects come into affect.

But speed is not always the only consideration to take when evaluating algorithms, another important consideration is the error that the algorithms produce. Where certain algorithms are more numerical stable than others. Table 4.2 shows these aspects.

By looking at Table 4.1 and Table 4.2 we find two algorithms that preforms better than the other algorithms when it comes to solving (1), namely $6n$ and $4n$. Both are approximately equal in speed and precision. From these results I will consider $6n$ superior to $4n$, because the $6n$ algorithm does not need extra memory to solve the problem, whereas the $4n$ requires an array of size n extra. However computers with different architectures may favour these two algorithms differently, because the $6n$ algorithms uses more CPU power, whereas the $4n$ algorithms uses more memory power.

6 Attachments

The files produced in working with this project can be found at
https://github.com/Eimund/UiO/tree/master/FYS4150/Project%201/A

The source files developed are

1. Matrix.h
2. project1.cpp
3. plot.py

7 Resources

1. QT Creator 5.3.1 with C11
2. Armadillo
3. Eclipse Standard/SDK - Version: Luna Release (4.4.0) with PyDev for Python
4. Ubuntu 14.04.1 LTS

5. ThinkPad W540 P/N: 20BG0042MN with 32 GB RAM

References

[1] Morten Hjorth-Jensen, *FYS4130 - Project 1 - Description*, University of Oslo, 2014

[2] Morten Hjorth-Jensen, *Computational Physics - Lecture Notes Fall 2014*, University of Oslo, 2014

[3] `http://en.wikipedia.org/wiki/Gaussian_elimination`

[4] `http://en.wikipedia.org/wiki/LU_decomposition`

[5] `http://en.wikipedia.org/wiki/Triangular_matrix#Forward_and_back_substitution`