Project 1

Erik Fagerås Skaar and Thomas Aarflot Storaas
Department of Chemistry
University of Oslo

19. september 2017

1 Abstract

In this article the numerical approximation to a one dimensional Poisson equation is studied. Three approaches are taken. One with a straight forward Gaussian elimination in a general form. One in a specified gaussian elimination form. The third approach are done with LU decomposition using the libarary "Armadillo". The numerical accuracy is then analysed and the three methods are then compared with respect to FLOPS and time used. All the code can be found at github.

2 Underlying theory

The Poisson's equation is a classical equation from electromagnetism, which states that $\nabla^2\Phi = -4\pi\rho((\mathbf{r}))$. The electrostatic potential Φ is a sentro- symmetrical potential, which makes it possible to rewrite the potential as a one dimentional equation with respect to the distance from origo, $\mathbf{r}$.

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

Introducing the substitutions $\Phi(r) = \phi(r)/r$ gives:

$$\frac{\partial^2\phi}{\partial r^2} = -4\pi r\rho(r) \tag{1}$$

Further manipulation of the expression with letting $x \to u$ and $r \to x$ gives:

$$-u''(x) = f(x)$$

For this project the Poisson equation will be solved with Dirichlet boundary conditions and source function $f(x) = 100\mathrm{e}^{-10x}$, in the interval $x \in [0, 1]$, which gives.

$$u(0) = u(1) = 0 \tag{2}$$

For this specific source function a numerical solution can be found by doing a double integration, and using the Dirichlet boundary conditions.

$$\begin{aligned} f(x) &= 100\mathrm{e}^{-10x} \\ \int f(x)\,\mathrm{d}^2x &= c_1x + c_2 + e^{-10x} \\ &= 1 - \left(1 - \mathrm{e}^{-10}\right)x - \mathrm{e}^{-10x} \end{aligned}$$

The discretized approximation to u is v_i, where the step size is defined as $h = 1/(n+1)$ with grid points(sample step) $x_i = ih$. The discretized approximation from 2 gives $v_0 = v_{n+1} = 0$. The second derivative approximation of u are

$$f_i = -\frac{v_{i+1} + 2v_i - v_{i-1}}{h^2} \qquad\qquad i = 1, 2...n$$

For each step x_i a equation can be found:

$$\begin{aligned} f_2 &= (-v_1 + 2v_2 - v_3)/h^2 \\ f_3 &= (-v_2 + 2v_3 - v_4)/h^2 \\ f_4 &= (-v_3 + 2v_4 - v_5)/h^2 \end{aligned}$$

Note here that the end points are not included. This system of equations can be organized in a matrix, by defining $\tilde{b}_i = h^2 f_i$.

$$\begin{matrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{matrix}$$

A row reduction of $A\mathbf{x} = f$ will give the solution to all the equations.

3 Method

General Gaussian elimination

The matrix equations we need to solve is the following:

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

Applying a general approach to the gaussian elimination of a 4×4 matrix gives an expression which can be generalized:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & c_1/b_1 & 0 & 0 \\ 0 & b_2 - (c_1/b_1)a_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} \tilde{b}_1/b_1 \\ \tilde{b}_2 - (\tilde{b}_1/b_1)a_1 \\ \tilde{b}_3 \\ \tilde{b}_4 \end{bmatrix}$$

Note that i have scaled the first row, this is done in a separate step in the code. One more row reduction gives the second row:

$$\begin{bmatrix} 1 & c_1/b_1 & 0 & 0 \\ 0 & 1 & c_2/(b_2-(c_1/b_1)a_2) & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} \tilde{b}_1/b_1 \\ (\tilde{b}_2-(\tilde{b}_1/b_1)a_1)/(b_2-(c_1/b_1)a_2) \\ \tilde{b}_3 \\ \tilde{b}_4 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} 1 & c_1/b_1 & 0 & 0 \\ 0 & 1 & c_2/(b_2-(c_1/b_1)a_2) & 0 \\ 0 & 0 & b_3-c_2\cdot a_2/(b_2-(c_1/b_1)a_2) & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} \tilde{b}_1/b_1 \\ a_2(\tilde{b}_2-(\tilde{b}_1/b_1)a_1)/(b_2-(c_1/b_1)a_2) \\ \tilde{b}_3-a_2(\tilde{b}_2-(\tilde{b}_1/b_1)a_1)/(b_2-(c_1/b_1)a_2) \\ \tilde{b}_4 \end{bmatrix}$$

From this a general result can be drawn. The next line is always dependent upon the previous lines result. Let d_i denote the previous lines result, and v_i denote the previous $\tilde{b}_i$ result. Then a general case can be written as:

$$d_i = b_i - \frac{c_i}{d_{i-1}} a_i \tag{3}$$

$$v_i = \frac{b_i - v_{i-1}a_i}{d_i} \tag{4}$$

3.1 Specific Gaussian elimination

When we have the general Gaussian elimination, we simple take out anything that is unnecessary to calculate. For example we don't need to the a-vector. Simply because it will become zero. And for the backward substitution we don't need to calculate the c-vektor. Same reason as for the a-vector.

3.2 LU decomposition

If we assume that the matrix is invertible, we can do an LU decomposition. This means that instead of $A\mathbf{v} = \tilde{b}$ we can write the equation as $L\mathbf{y} = \tilde{b}$ and $U\mathbf{v} = \mathbf{y}$. Where L is a matrix where all the elements above the diagonal is zero and U is a matrix where all the elements below the diagonal is zero. We simply used the library armadillo and therefore did not use any special method for solving these equations.

3.3 Error

The problem does have an analytic solution. We started with $f(x) = 100e^{-10x}$ and filled our vector with values between 0 and 1. And this gives the analytic solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. Because we have the analytic solution we can calculate the relative error. u_i is the analytic solution and v_i is the numerical answer.

$$\epsilon_i = log_2\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

4 Result

4.1 General Gaussian elimination

The general Gaussian elimination implementation is showed below.

```
//forward substitution
for (int i = 1; i<n;i++){
    double alpha = a[i]/b[i-1];
    a[i] = a[i] - alpha*b[i-1];
    b[i] = b[i] - alpha*c[i-1];
    f_tilde[i] = f_tilde[i] - alpha*f_tilde[i-1];
}

//backward substitution
for (int i = n-2; i>-1;i--){
    double c_ledd = c[i]/b[i+1];
    c[i] = c[i] - c_ledd*b[i+1];
    f_tilde[i] = f_tilde[i] - c_ledd*f_tilde[i+1];
}

//scaling
for (int i = 0; i<n;i++){
    f_tilde[i] = f_tilde[i]/b[i];
    b[i] = b[i]/b[i];
}
```

The forward substitution part has n-1 loops. For each loop it's 5 FLOPs.
The backward substitution has n-1 loops. For each loop it's 5 FLOPs.
The scaling is because the b row in the matrix was not 1. It had different values for every b_i. The scaling has n loops. For each loop it's 2 FLOPs.
Which gives us:

$$\textbf{Total-FLOPs} = 5(n-1) + 5(n-1) + 2n$$
$$\textbf{Total-FLOPs} = 12n - 10 \simeq 12n$$

4.2 Specific Gaussian elimination

The specific Gaussian implementation is showed below.

```
//forward substitution
for (int i = 1; i<n;i++){
    double alpha = 1/b[i-1];
    b[i] = b[i] - alpha;
    f_tilde[i] = f_tilde[i] - alpha*f_tilde[i-1];
}
//backward substitution
for (int i = n-2; i>-1;i--){
    f_tilde[i] = f_tilde[i] - (1./b[i+1])*f_tilde[i+1];
}
//scaling
for (int i = 0; i<n;i++){
    f_tilde[i] = f_tilde[i]/b[i];
}
```

The forward substitution part has n-1 loops. For each loop it's 4 FLOPs.
The backward substitution has n-1 loops. For each loop it's 3 FLOPs.
The scaling is because the b row in the matrix was not 1. It had different values for every b_i. The scaling has n loops. For each loop it's 1 FLOP.
Which gives us:

$$\textbf{Total-FLOPs} = 4(n-1) + 3(n-1) + n$$
$$\textbf{Total-FLOPs} = 8n - 7 \simeq 8n$$

4.3 Armadillo LU decomposition

The implementation of LU decomposition is really simple when you use the package Armadillo. It is showed below.

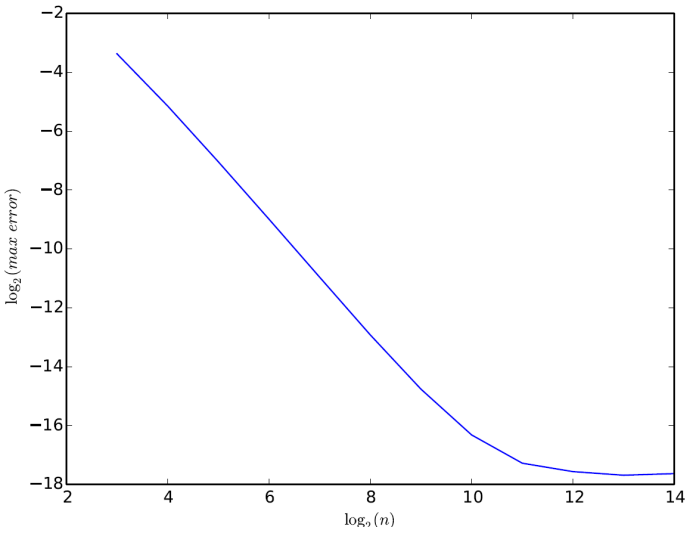
```
mat L,U;
lu(L,U,A);
mat Y = solve(L,f_tilde);
mat V = solve(U,Y);
```

Unfortunately it's quit hard to count the number of FLOPs in this code. From the lectures in FYS3150 at the University of Oslo we learn that this is correlated to n^3.

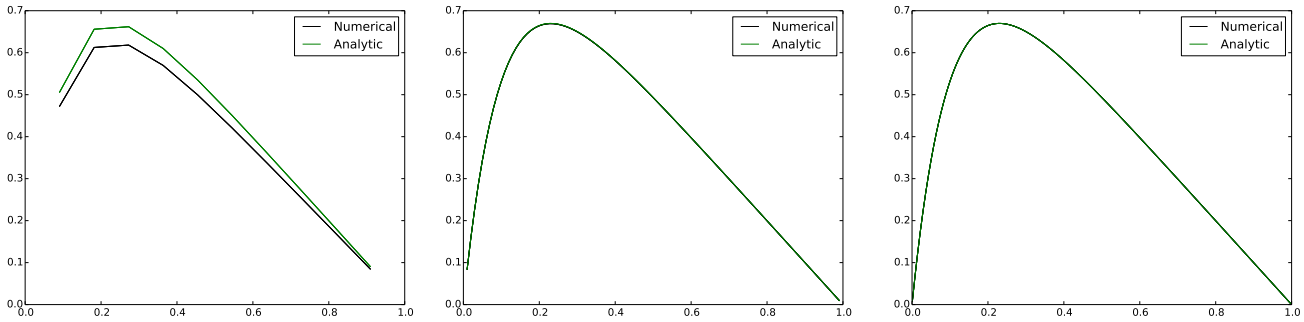
4.4 Time and Error

The time estimate for the different implementation is shown in the table below.

n	General	Specific	LU	fastest	slowest	$\frac{slowest}{fastest}$
10	6.5e-05	5e-06	4e-05	Specific	General	13.0
100	7.5e-05	8e-06	0.0023	Specific	LU	287.5
1000	0.00014	4e-05	0.26	Specific	LU	6500
10000	0.0007	0.0005	142.5	Specific	LU	285000



Figur 1: The plot is produced by the python script on our github. It runs the main.cpp program and checks the max error for different n-values.



Figur 2: The numerical answer against the analytic solution for n $n = 10$,$n = 100$ and $n = 1000$ (starting from the left).

5 Discussion

From 1 we can see how the error behaves. The slope is near 2 for the first part of the graph. When we get to 2^{-10} the slope goes to near zero and then up. This is because of the way the computer works. When you represent an infinite set of numbers with a finite number of bit, then you can only represents number accurate to a certain point. This is what is happening here. We are simply working with so small numbers that we cant represent them accurately anymore. This is the reason that the error suddenly starts to get bigger.

The results of the three methods give the same result. The specific will use less time, then the other two. We can see this from the FLOP calculation in the method part.

When you use the specific method you only need to work with two arrays in the memory. This in turn mean that you can work with larger arrays before running out of memory. And if you count the arrays in LU method. You will se that this method uses much more memory to do the calculation. This means that even tho the LU is simpler to implement, it has it's weaknesses.

6 Conclusion

The specific solution is the fastest, but if the matrix have a slight change it does not solve the problem we have. The general is almost as fast, but it can solve a wider range of tridiagonal matrices. The LU solution is the slowest. It is useful no matter what matrix you have, but it has a significant drawback in speed. By our calculations the error is the same for every method. The error seem to follow until a point where the error become bigger for smaller steps.