FYS 4150 - Computational Physics
Project 1: Solving Poisson's equation in one dimension

Maren Rasmussen Markus Leira Asprusten Metin San

4. September 2018

Abstract

This project involves solving the one-dimensional Possion equation with dirichlet boundary conditions using two different algorithms. The first method is the tridiagonal matrix algorithm while the second is the LU decomposition. The two methods are then compared in terms of efficiency. The conclusion of the project is that a specialized version of the tridiagonal algorithm is much faster.

1. Introduction

The main goal of this project is to familiarize us with vector and matrix operations with focus on memory allocation and the use of library packages such as the c++ library Armadillo. Many of the most important differential equations in physics can be written as linear second-order differential equations. We will therefore address our issue by studying such an equation, namely Possion's equation.

We will derive and experiment with two specific numerical algorithms which can be applied to solve the Possion equation. The first is the tridiagonal matrix algorithm, and the second is the LU-decomposition method. We will further tailor the more general tridiagonal matrix algorithm to specifically deal with the Possion equation and the given source term in order to increase the methods efficiency. Both algorithms will be tested at different precision levels using varying grid points. The speed and efficiency, along with the errors produced from the two methods will then be compared and discussed.

2. Theory

2.1. **The Poisson Equation.** Poisson's equation is a classical and well known differential equation from electromagnetism. The equation describes the potential field Φ generated from a charge distribution $\rho(\mathbf{r})$. For three dimensions, the equation is given by

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

where $\nabla =$ is the Laplace operator. If both Φ and $\rho(\mathbf{r})$ is spherically symmetric, the equation can be simplified to an one dimensional equation in r,

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

By substituting $\Phi = \phi(r)/r$, we can simplify the equation even more, giving

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

The final equation can be written in a general form by letting $\phi \to u$ and $r \to x$, and defining the right hand side of the equation as f,

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

In this study we will assume that the source term is $f(x) = 100e^{-10x}$ with $x \in [0, 1]$. We will also assume that we have Dirichlet boundary conditions, such that $u(0) = u(1) = 0$. With these assumptions the equation have an exact solution on the form $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

2.2. **Discretization of the problem.** To solve the Poisson equation numerically, we need to discretize the problem. The discritation can be done using $n + 1$ x-values, such that $x \in [x_0, x_1, x_2, ..., x_n]$ with $x_0 = 0$ and $x_n = 1$, and $u(x_i) = u_i$. Then we have that

$$x_i = x_0 + ih,$$

where $h = (x_n - x_0)/n$ is the step size.

A discretized version of $\frac{d^2u(x)}{dx^2}$ can be found by using Taylor expansion. We know that

$$u(x+h) = u(x) + hu' + \frac{h^2}{2!}u'' + O(h^2)$$
$$u(x-h) = u(x) - hu' + \frac{h^2}{2!}u'' + O(h^2)$$

The $O(h^2)$-term is the remaining terms from the Taylor expansion, or the error we get by excluding these terms. By adding $u(x + h)$ and $u(x - h)$, we can derive the desired expression:

$$u(x+h) + u(x-h) = 2u(x) + \frac{2}{2!}h^2u'' + O(h^2)$$
$$u''(x) = \frac{u(x+1) + u(x-h) - 2u(x)}{h^2} + O(h^4)$$
$$u'' = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + O(h^2)$$

So by excluding the rest of the terms in the Taylor expansion, and defining $f_i^* = f_i h^2 = f(x_i)h^2$, our equation is given by:

$$-u_{i+1} - u_{i-1} + 2u_i = f_i^*.$$

For each x-value we then get a set of linear equations,

$$\begin{aligned} -v_2 - v_0 + 2v_1 &= f_1^* \\ -v_3 - v_1 + 2v_2 &= f_2^* \\ &\vdots \\ -v_n - v_{n-2} + 2v_{n-1} &= f_{n-1}^*. \end{aligned}$$

It is easy to see that the right hand side of the equation set, can represent as a vector on the fllowing form

$$\hat{\mathbf{f}} = \begin{bmatrix} f_1^* \\ f_2^* \\ \vdots \\ f_{n-1}^* \end{bmatrix}$$

The left hand side of the equation set we can represent as a matrix product on the following form

$$\hat{\mathbf{A}}\hat{\mathbf{u}} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \end{bmatrix}.$$

This means that the set of equations can be written as a linear algebra problem on the form

$$\hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{f}}.$$

3. Algorithm

With the problem now formulated in terms of linear algebra, the next step is to solve it. We will tackle this problem through the implementation of two algorithms. The first is the Tridiagonal matrix algorithm, also known as the Thomas algorithm. The second is the LU-decomposition algorithm.

3.1. **Tridiagonal Matrix Algorithm.**

This algorithm is a simplified form of Gaussian elimination which can be used to solve tridiagonal systems of equations. In the general case, a tridiagonal system of n unknowns can be represented as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = b_i, \tag{1}$$

where $a_1 = c_1 = 0$. Or in matrix representation as $\mathbf{Au} = \mathbf{f}$. Written out in the 4×4 case, this becomes

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}. \tag{2}$$

The algorithm is quite simple and consists of mainly two steps, a forward substitution and a backwards substitution. The forward substitution reduces the tridiagonal matrix $\mathbf{A}$ to an upper tridiagonal matrix. This is achieved through Gaussian elimination. We want to get rid of the a_i terms located on the lower secondary diagonal. We perform the following row reduction on both sides of the equation

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{II}-\frac{a_2}{b_1}\text{I}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \xrightarrow{\text{II}-\frac{a_2}{b_1}\text{I}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ f_3 \\ f_4 \end{bmatrix}$$

where $\tilde{b_2} = b_2 - a_2 c_1 / b_1$, and $\tilde{f_2} = f_2 - f_1 a_2 / b_1$, and II and I denotes the row 1 and 2 in the $\mathbf{A}$. Similarly for the second row

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ \tilde{f_2} \\ f_3 \\ f_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ f_4 \end{bmatrix}$$

where $\tilde{b_3} = b_3 - a_3 c_2 / b_2$, and $\tilde{f_3} = f_3 - f_2 a_3 / b_2$. Finally we compute the last row reduction

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \xrightarrow{\text{IIII}-\frac{a_4}{b_3}\text{III}} \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & \tilde{b_4} \end{bmatrix}, \qquad \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ f_4 \end{bmatrix} \xrightarrow{\text{III}-\frac{a_3}{b_2}\text{II}} \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \tilde{f_4} \end{bmatrix}$$

where $\tilde{b_4} = b_4 - a_4 c_3 / b_3$, and $\tilde{f_4} = f_4 - f_3 a_4 / b_3$.

We are then left with the row reduced form of the set of equations $\tilde{\mathbf{A}}\mathbf{u} = \tilde{\mathbf{f}}$, or in matrix notation

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & \tilde{b_4} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ \tilde{f_4} \end{bmatrix}. \tag{3}$$

If one takes a closer look at the steps which we carried out, one notices the following pattern for $\tilde{b}$ and $\tilde{d}$. These can be generally expressed as

$$\tilde{b_i} = b_i - \frac{a_i c_{i-1}}{\tilde{b}_{i-1}}, \qquad \tilde{f_i} = f_i - \frac{a_i \tilde{f}_{i-1}}{\tilde{b}_{i-1}}, \qquad i \in [2, 4], \tag{4}$$

where $b_1 = \tilde{b_1}$ and $f_1 = \tilde{f}$. In general for a $(n \times n)$ matrix we would have $i \in [2, n]$. The forward substitution has been implemented in the following way in c++:

```
f_tilde[1] = f[1];
double ab;
// forward substitution
for (int i = 2; i < n; i++){
    ab = a[i]/b[i-1];
    b[i] = b[i] - ab*c[i-1];
    f_tilde[i] = f[i] - ab*f_tilde[i-1];
}
```

Note that instead of allocating memory for a seperate $\tilde{b}$ array, we have rather reused the b array. We also compute $a_i/\tilde{b}_{i-1}$ at the start of the loop which saves us a FLOP as it appears in both the expression for $\tilde{b}$ and $\tilde{f}$.

The last part of the tridiagonal algorithm is the backwards substitution. By setting up the set of equations in (?), we are able to solve each of these for their respective solution u_i. The first equation along with its solution is then

$$\tilde{b_1}u_1 + c_1 u_2 = \tilde{f_1} \qquad \rightarrow \qquad u_1 = \frac{\tilde{f_1} - c_1 u_2}{\tilde{b_1}},$$

where we have used that $b_1 = \tilde{b}_1$. Similarly for the second and the third rows

$$\tilde{b_2}u_2 + c_2 u_3 = \tilde{f_2} \qquad \rightarrow \qquad u_2 = \frac{\tilde{f_2} - c_2 u_3}{\tilde{b_2}},$$

$$\tilde{b_3}u_3 + c_3 u_4 = \tilde{f_3} \qquad \rightarrow \qquad u_3 = \frac{\tilde{f_3} - c_3 u_4}{\tilde{b_3}}.$$

For the final row, we simply get

$$\tilde{b_4}u_4 = \tilde{f_4} \qquad \rightarrow \qquad u_4 = \frac{\tilde{f_4}}{\tilde{b_4}}.$$

This is a result of the chosen dirichlet boundary conditions. Again we notice the solution u_i follows the following pattern

$$u_i = \frac{\tilde{f_i} - c_i u_{i+1}}{\tilde{b_i}}. \tag{5}$$

This is implemented in the code as

```
// backward substitution
u[n-1] = f_tilde[n-1]/b[n-1];          //setting the last term

for (int i = n-2; i > 0; i--){
   u[i] = (f_tilde[i] - c[i]*u[i+1])/b[i];
}
```

where we see that the last term has been computed separately as it differs from the general algorithm.

In general one of the most important aspects of any algorithm is its efficiency. The tridiagonal matrix algorithm is known to be a relatively fast algorithm as it only uses three diagonal vectors to represent the entire $(n \times n)$ matrix which severely reduces the number of floating points operations (FLOPS) required to solve the set of equations. We will assume that addition, subtraction, multiplication and division all counts as FLOPS. In reality, division operations are said to be "heavier" than the other three operations. The forward substitution method requires 6 FLOPS for each iteration, and it is computed $(n-2)$ times which results in a total of $6(n-2)$ FLOPS. The backward substitution requires $3(n-2)+1$ FLOPS where the $+1$ term comes from the definition of the last term, which has to be computed just once.

3.1.1 **Optimizing the Tridiagonal Matrix Algorithm.** The number of floating point operations in the algorithm can be severely reduced if we specialize it for our special case with the Poisson equation. Since we are only interested in the tridiagonal matrix which resulted from the discretization of the second derivative, we can use the precomputed matrix (?). This allows us to rewrite the expressions for the forward and backwards substitution. If one inserts for the constant $a_i = c_i = -1$ and $b_i = 2$ into equations (?) and (?), we find that we can in fact rewrite these into the form

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}} = \frac{i+1}{i}, \tag{6}$$

$$\tilde{f}_i = f_i + \frac{(i-1)\tilde{f}_{i-1}}{i}, \tag{7}$$

$$u_i = \frac{i}{i+1}(\tilde{f}_i + u_{i+1}). \tag{8}$$

Now, since the diagonal elements $\tilde{b}_i$ can be precomputed as they only depend on i, we can move this calculation outside of the main algorithm. Further, we spot that we can rewrite (?) in terms of $\tilde{b}_{i-1} = i/(i-1)$, to the form

$$\tilde{f}_i = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{9}$$

Which has now been reduced to 2 FLOPS down from 3. Similarly we rewrite the (?) in terms of $\tilde{b}_i$ to the from

$$u_i = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i}, \tag{10}$$

which has now also been reduced to 2 FLOPS down from 3.

The new specialized algorithm is implemented in the following way

```
// forward substitution
f_tilde[1] = f[1];
for (int i = 2; i < n; i++){
    f_tilde[i] = f[i] + (f_tilde[i-1]/d[i-1]);
}

// backward substitution
u[n-1] = f_tilde[n-1]/d[n-1];          // setting the last term

for (int i = n-2; i > 0; i--){
    u[i] = (f_tilde[i] + u[i+1])/d[i];
}
```

The total number of FLOPS is then reduced to $4(n-2)+1$ operation, which is half that of the general algorithm.