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 conclusion of the project is that a specialized version of the tridiagonal algorithm is much faster.

1. Introduction

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3. Algorithm

In order to solve Poisson's equation we need to be able to solve the discretized set of equations involving the tridiagonal matrix **A**. 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. 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 $\mathbf{Au} = \mathbf{b}$. 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 **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

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

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 **A**. Similarly for the second row

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

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

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

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

Note that instead of allocating memory for a seperate $\tilde{b}$ array, we have rather reused the b array.

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.