SOLVING THE POISSON-EQUATION IN ONE DIMENSION
FYS3150: COMPUTATIONAL PHYSICS

TRYGVE LEITHE SVALHEIM
SEBASTIAN G. WINTHER-LARSEN
GITHUB.COM/GREGWINTHER

ABSTRACT. Lorem ipsum dolor sit amet, consectur adipiscing lit. Nullam ut lacus eget lorem...

CONTENTS

1. Introduction

2. Theory

2.1. The Poisson Equation.

The Poisson equation is a classical equation from electromagnetism. The electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimensions the equation reads

$$\nabla^2 \Phi = -4\pi\rho(\mathbf{r}) \tag{1}$$

where ∇^2 is the Laplace operator. In three dimensions the Laplace operator can be expressed using spherical coordinates, but in this study I am assuming that Φ and ρ are spherically symmetric, thus reducing the equation to a one-dimensional problem. only dependent on radius r.

$$\nabla^2 = \frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) \tag{2}$$

By substituting $\Phi(r) = \phi(r)/r$ the Poisson equation is reduced to

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r) \tag{3}$$

and by letting $\phi \to u$ and $r \to x$ one is left with the very simple equation

$$-u''(x) = f(x) \tag{4}$$

The inhomogenous term f, or source term, is given by the charge distribution ρ multiplied by r and the constant -4π. In this study, however, the source term will be $f(x) = 100e^{-10x}$ and the results can be compared to the analytical solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

2.2. Approximation of the Second Derivative.

In this study the one-dimensional Poisson equation will be solved with Dirichlet boundary conditions by rewriting it as a set of linear equations. The discretized approximation of u is defined as v_i with grid points $x_i = ih$, step size of $h = \frac{1}{n+1}$, in the interval $x_0 = 0$ to $x_{n+1} = 1$ and with boundary conditions $v_0 = v_n + 1 = 0$. The interior solution $v_i \forall i \in 1, ..., n$ is to be found. The second order derivative is approximated with the three point formula such that equation 4 becomes

$$-\frac{v_{i+1} - 2v_i + vi - 1}{h} = f_i \tag{5}$$

By defining $\mathbf{f} = h^2 f_i$ one can rewrite equation 5 as $-v_{i+1} - 2v_i + v_{i-1} = h^2 f_i$. If we ignore the end points, $i = 0$ and $i = n + 1$, this equation can be represented as a matrix equation.

$$A\mathbf{v} = \mathbf{f} \tag{6}$$

$$
\begin{bmatrix}
2 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\
-1 & 2 & -1 & 0 & \cdots & 0 & 0 & 0 \\
0 & -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\
 & & \vdots & & \ddots & & \vdots & \\
0 & 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\
0 & 0 & 0 & 0 & \cdots & 0 & -1 & 2
\end{bmatrix}
\begin{bmatrix}
v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{n-1} \\ f_n
\end{bmatrix}
\tag{7}
$$

3. Algorithms

Two main methods are implemented and compared. The first method is gaussian elimination of the tridiagonal matrix A, also known as the *Thomas Algorithm* [1]. This is a simplified form of Gaussian elimination that can ve used to solve tridiagonal systems of equations. The method is improved upon in order to take into account the fact that the matrix we are dealing with has the same numbers along the diagonals. The second method is the LU-decomposistion method.

3.1. Tridiagonal Matrix Algorithm.

Our tridiagonal system can be represented by

$$(8) \qquad a_i x_{i-1} + b_i x_i + c_i x_{i+1} = \mathbf{f}$$

with $a_i = -1, b_i = 2$ and $c_i = -1$, except for $a_1 = 0$ and $c_n = 0$. Row reducing a matrix will reveal how the algorithm functions. Limiting the problem to four dimensions for easier reading and to save the rainforest[1].

$$(9) \quad \left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ a_2 & b_2 & c_2 & 0 & f_2 \\ 0 & a_3 & b_3 & c_3 & f_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right] \sim \left[\begin{array}{cccc|c} 1 & c_1/b_1 & 0 & 0 & b_1/b_1 \\ 0 & b_2 - \frac{c_1}{b_1}a_2 & c_2 & 0 & f_2 - \frac{f_1}{b_1}a_2 \\ 0 & a_3 & b_3 & c_3 & f_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right]$$

Now let $\beta_1 = b_1$ and $\beta_2 = b_2 - \frac{c_1}{b_1}a_2$. As new elements start to appear in vector $\mathbf{f}$, right to the vertical bar in the augmented matrix, they are also relabeled to $\tilde{f}_i$. For example $\tilde{f}_1 = f_1/\beta_1$. One more iteration will reveal the pattern of the algorithm.

$$(10)$$
$$\left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & (f_2 - \tilde{f}a_2)/\beta_2 \\ 0 & a_3 & b_3 & c_3 & f_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right] = \left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & a_3 & b_3 & c_3 & f_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right]$$

$$(11)$$
$$\sim \left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & 0 & b_3 - \frac{c_2}{\beta_2}a_3 & c_3 & f_3 - \tilde{f}_2 a_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right] \sim \left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & c_3/\beta_3 & (f_3 - \tilde{f}_2 a_3)/\beta_3 \\ 0 & 0 & a_4 & b_4 & f_4 \end{array} \right]$$

$$(12) \qquad \sim \cdots \sim \left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & c_3/\beta_3 & \tilde{f}_3 \\ 0 & 0 & 0 & 1 & \tilde{f}_4 \end{array} \right]$$

This pattern can be summarized nicely by the following difference equations. This computation will from now on be referred to as the "forward substitution".

$$(13) \qquad \beta_i = b_i - \frac{a_i c_{i-1}}{\beta_{i-1}}, \quad \tilde{f}_i = f_i - a_i \tilde{f}_{i-1}/\beta_{i-1}, \quad i \in [2, n],$$

In c++, this part of the algorithm is implemented just below. Notice that b_i is overwritten instead of initializing a seperate vector for β.

```
// GAUSSIAN ELIMINATION
// Forward substitution
f_tilde[1] = f[1];
for (int i=2; i<n+1; i++){
```

---

[1]Writing out a general case will also take up more paper space

```
        b_vec[i] = b_vec[i]−a_vec[i]*c_vec[i−1]/b_vec[i−1];
        f_tilde[i] = f[i] − a_vec[i]*f_tilde[i−1]/b_vec[i−1];
    }
```

The Gaussian elimination is not fully complete until a "backward substitution" has been performed as well. This ensures that all eliments in the row reduced matrix are pivot elements.

$$(14) \quad \begin{bmatrix} 1 & c_1/\beta_1 & 0 & 0 \\ 0 & 1 & c_2/\beta_2 & 0 \\ 0 & 0 & 1 & c_3/\beta_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \end{matrix} \sim \begin{bmatrix} 1 & c_1/\beta_1 & 0 & 0 \\ 0 & 1 & c_2/\beta_2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 - \frac{c_3}{\beta_3}\tilde{f}_4 \\ \tilde{f}_4 \end{matrix}$$

The notation is updated again to $v_3 = \tilde{f}_3 - \frac{c_3}{\beta_3}\tilde{f}_4$. By continuing this iterative process all the was to the beginning of the matrix by backward substitution leaves us with a second recurrence relation.

$$(15) \qquad v_i = (\tilde{f}_i - c_i v_{i+1})/\beta_i, \quad i \in \{n-1, n-2, \ldots, 1\}$$

The backward substitution is implemented in C++ like so:

```
// Backward substition:
v[n] = f_tilde[n]/b_vec[n];
for (int i=n−1; i>=1; i−−){
    v[i] = (f_tilde[i]−c_vec[i]*v[i+1])/b_vec[i];
}
```

An ever-important aspect of any algorithm is how efficient the algorithm is. It is therefor worthwhile to consider how many floating point operations the algorithm requires[2]. For the forward substitution there are 6 FLOPS in each iteration and $n-1$ iterations, resulting in $6(n-1)$ FLOPS for the entire loop. Additionally, $1(n-1)$ FLOPS are required for the backward substitution, adding up to

$$(16) \qquad N_{tridiag} = 8(n-1) = \mathcal{O}(8n).$$

3.1.1. *Optimizing the algorithm.* Our case is special, because the elements in each of the diagonal arrays in the tridiagonal matrix, A, are equal. $a_i = -1, b_i = 2$ and $c_i = -1$, except for $a_1 = 0$ and $c_n = 0$. This is an advantage, because the algorithm can be adjusted, and the number of FLOPS consequently reduced.

The algorithm for the forward substitution, in equation 13, can be simplified to

$$(17) \qquad \beta_i = b_i - 1/\beta_{i-1} \quad \tilde{f}_i = f_i - \tilde{f}_{i-1}/\beta_{i-1}$$

and the backward substitution, in equation 15, can be simplified to

$$(18) \qquad v_i = (\tilde{f}_i + v_{i+1})/\beta_i$$

This results in some new and sleeker C++ code

```
// GAUSSIAN ELIMINATION
// Forward substitution
f_tilde[1] = f[1];
for (int i=2; i<n+1; i++){
    b_vec[i] −= 1/b_vec[i−1];
    f_tilde[i] = f[i] + f_tilde[i−1]/b_vec[i−1];
}

// Backward substition:
v[n] = f_tilde[n]/b_vec[n];
for (int i=n−1; i>=1; i−−){
    v[i] = (f_tilde[i] + v[i+1])/b_vec[i];
```

---

[2]The number of $+, -, *$ and $/$ operations are counted

$$\left|\quad\quad\quad\}\right.$$

One can see that this new and improved algorithm only requires $6(n-1)$ FLOPS. In general, a Gaussian row reduction algorithm requires $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ FLOPS [2], a great improvement!

3.2. **LU Decomposition.** The LU decomposition method is a method where the matrix A in equation 6 ($A\mathbf{v} = \mathbf{f}$) can be rewritten as a product of two matrices L and U, where L is lower triangular and and U is upper triangular. This method only works for non-singular, invertible matrices, which the matrix A certainly is.

(19)

$$A = LU = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ l_{21} & 1 & 0 & \dots & 0 & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 & 0 \\ & \vdots & & \ddots & \vdots & \\ l_{n-11} & l_{n-12} & l_{n-13} & \dots & 1 & 0 \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn-1} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n-1} & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n-1} & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n-1} & u_{3n} \\ & \vdots & & \ddots & \vdots & \\ 0 & 0 & 0 & \dots & u_{n-1n-1} & u_{n-1n} \\ 0 & 0 & 0 & \dots & 0 & u_{nn} \end{bmatrix}$$

The previous equation can be solved in two steps

(20) $L\mathbf{w} = \mathbf{f}, \quad U\mathbf{v} = \mathbf{w}$

4. Results

5. Discussion

6. Conclusion

References

[1] Thomas, L.H. (1949), *Elliptic Problems in Linear Differential Equations over a Network.* Watson Sci. Comput. Lab Report, Columbia University, New York.

[2] Hjorth-Jensen, M. (2016). *Computational Physics - Lecture Notes 2016.* University of Oslo