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

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

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].

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

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.

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

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

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

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

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

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.

$$\left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & c_3/\beta_3 & \tilde{f}_3 \\ 0 & 0 & 0 & 1 & \tilde{f}_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/\beta_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & c_2/\beta_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & 0 & \tilde{f}_3 - \frac{c_3}{\beta_3}\tilde{f}_4 \\ 0 & 0 & 0 & 1 & \tilde{f}_4 \end{array}\right] \tag{14}$$

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.

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

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

```
    // Backward substitution:
    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

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

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

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

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

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

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 substitution:
    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

```
    }
```

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.

$$A = LU = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ l_{21} & 1 & 0 & \dots & 0 & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 & 0 \\ & \vdots & & \ddots & \vdots & \\ l_{n-11} & l_{n-12} & l_{n-13} & \dots & 1 & 0 \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn-1} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n-1} & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n-1} & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n-1} & u_{3n} \\ & \vdots & & \ddots & \vdots & \\ 0 & 0 & 0 & \dots & u_{n-1n-1} & u_{n-1n} \\ 0 & 0 & 0 & \dots & 0 & u_{nn} \end{bmatrix} \tag{19}$$

The equation $A\mathbf{v} = LU\mathbf{v} = \mathbf{f}$ can be solved in two steps

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

Written out, the full set of these linear equations will look like

$$\begin{aligned} w_1 &= f_1 && (21) \\ l_{21}w_1 + w_2 &= f_2 && (22) \\ l_{31}w_1 + l_{32}w_2 + w_3 &= f_3 && (23) \\ &\vdots && (24) \\ l_{n-11}w_1 + l_{n-12}w_2 + l_{n-13}w_3 + \cdots + w_{n-1} &= f_{n-1} && (25) \\ l_{n1}w_1 + l_{n2}w_2 + l_{n3}w_3 + \cdots + l_{nn-1}w_{n-1} + w_n &= f_n && (26) \\ u_{11}v_1 + u_{12}v_2 + u_{13}v_3 + \cdots + u_{1n-1}v_{n-1} + u_{1n}v_n &= w_1 && (27) \\ u_{22}v_2 + u_{23}v_3 + \cdots + u_{2n-1}v_{n-1} + u_{2n}v_n &= w_2 && (28) \\ u_{33}v_3 + \cdots + u_{3n-1}v_{n-1} + u_{3n}v_n &= w_3 && (29) \\ &\vdots && (30) \\ u_{n-1n-1}v_{n-1} &= w_{n-1} && (31) \\ u_{nn}v_n &= w_n && (32) \end{aligned}$$

First $\mathbf{w}$ can be determined by iterating forwards through equations 21 to 26. Then $\mathbf{v}$ can be found be iterating backwards from equation 32 to equation 27.

In this study the Armadillo library is employed instead of implementing the LU decomposition algorithm from scratch. The LU method is used as a benchmark against which the speed of the tridiagonal method is measured. The number of FLOPS required by the LU is $2n^3/3$, by any alogorithm used to implement it[3]. Additionally, n^2 FLOPS is required by the forward and backwards substitution.

$$N_{LU} = \frac{2}{3}n^3 - 2n^2 = \mathcal{O}(\frac{2}{3}n^3) \tag{33}$$

Using LU computation results in a quadratic slowdown compared to the tridiagonal matrix algorithm.

4. Results

The run time for the optimized tridiagonal matrix algorithm and the LU decomposition solver is in table 1. The reason run times are not given for $n > 10000$ for the LU method is because the method uses too much memory for the solver to even function. Regardless, the message of table 1 is quite clear, the specialized tridiagonal matrix method is vastly more efficient.

Table 1. Elapsed time for increasing n

n	TDMA [s]	LU [s]
10	0.0000035	0.00106083
100	0.0000116	0.0022319
1000	0.000077892	0.0677764
10000	0.000878769	21.9247
100000	0.00757418	n/a
1000000	0.08616075	n/a
10000000	0.76534	n/a

Tables 1 and 2 show the fit of the tridiagonal matrix solver for $n = 10$ and $n = 100$, respectively. One can easily see that already at $n = 100$ the fit is already very good.

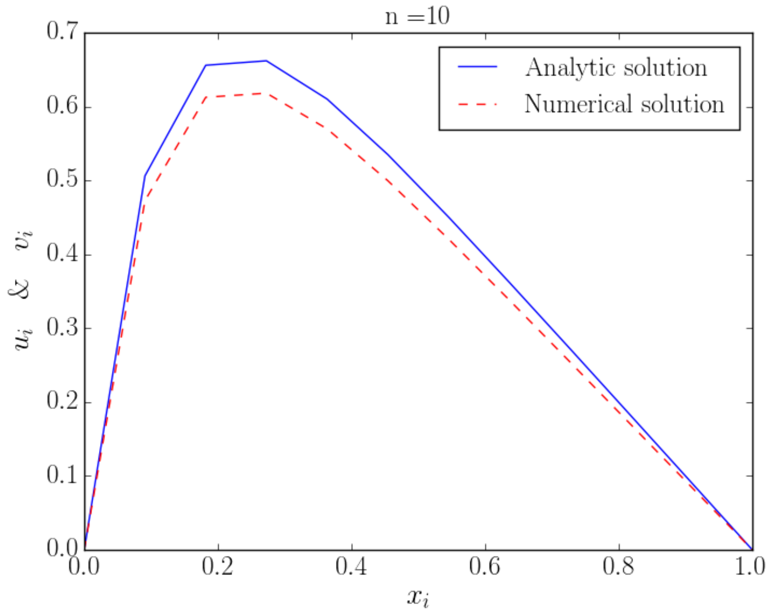

Figure 1. Numerical approximation of the tridiagonal matrix algorithm for $n = 10$

Figure 3 shows a plot of the maximum relative error of the tridiagonal solver as a function of step size. One can see that the relative error falls as the step size falls, but only to a certain point. By decreasing the step size further the error increases, most likely because of decreasing numerical precision because of the computers difficulty in representing very small numbers.

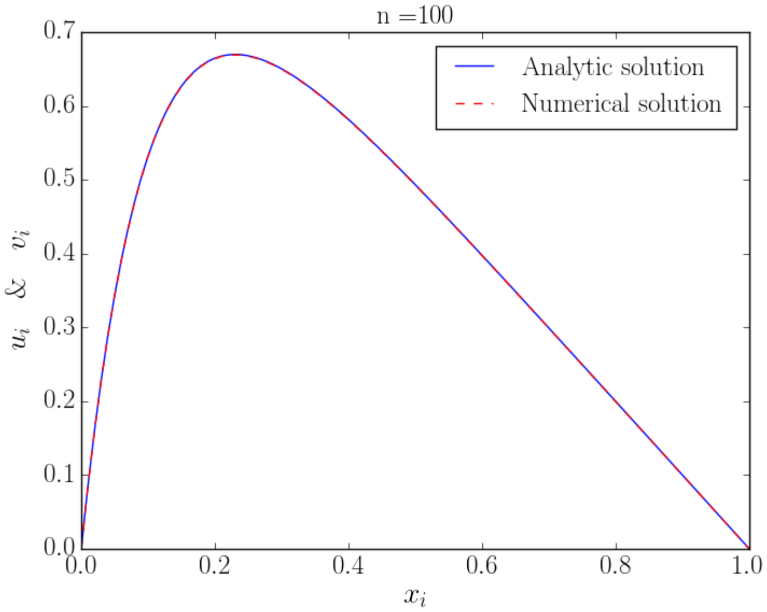

Figure 2. Numerical approximation of the tridiagonal matrix algorithm for $n = 100$

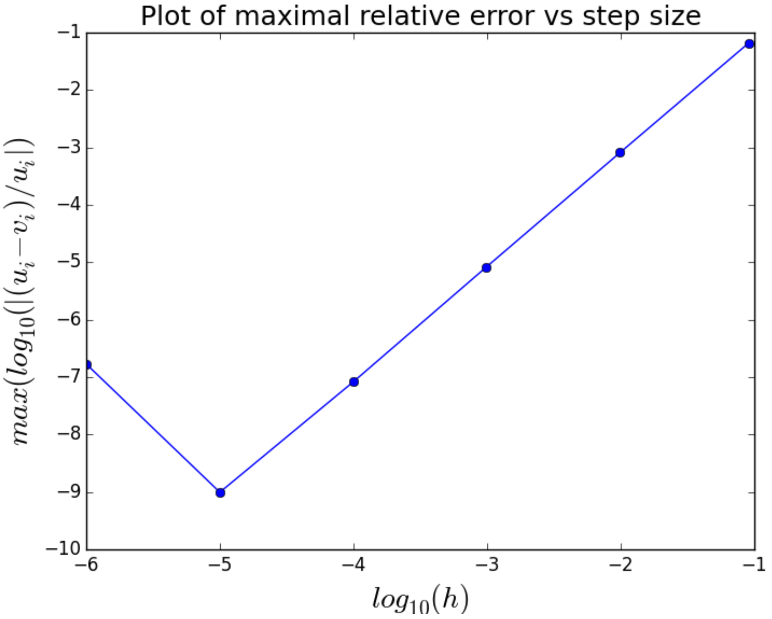

Figure 3. Plot of maximum relative error as a function of step size

5. Discussion

When choosing a suitible value for n, one must take into account both precision and speed. Our intuition tells us that a larger n should result in a higher precision for the numerical derivative. We can conclude from figure 3 that this is only true up to a certain point, because of loss in numerical representation accuracy within the computer. An ideal n will be in the range of $n \approx 10^5$ which is equivalent to $h = 10^-5$. Increasing this value will not increase run time dratically for the tridiagnoal matrix algorithm. Choosing $n = 10^5$ leaves us with the ability to calculate with a high certainty of 5 decimals points and with low memory cost.

By looking at the results one is led to believe that there are more reasons to prefer the tridiagonal matrix algorithm for this kind of problem. When using the LU decomposition method the short term memory is quickly filled with nothing but the matrix A. Choosing $n = 10^5$ is equivalent to 8×10^{10} bytes= 80GB of memory! A brand new computer ships with around 8 GB of memory today, which is a tenth of the space required. Obviously, this inhibits our ability to perform the calculation.

6. Conclusion

As we have seen, there is significant advantage to fine-tuning and picking the correct algorithm before solving a problem numerically. We have experienced the restrictions in memory when computing, and have been made aware of the problem this poses when doing numerical calculations. The specialised tridiagonal matrix algorithm was able to solve the one-dimensional Poisson equation more than 20000 times faster than the more general LU decomposition method. This can be explained by the fact that the LU decomposition method require $\mathcal{O}(\frac{2}{3}n^3)$ FLOPS while the specialized tridiagonal matrix algorithm will do with a meager $\mathcal{O}(8n)$ FLOPS.

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. (2015). *Computational Physics - Lecture Notes 2015.* University of Oslo

[3] Golub, G.H., van Loan, C.F. (1996). *Matrix Computations* (3rd ed.), Baltimore: John Hopkins.