

# Project 1

Erlend Lima, Frederik J. Mellbye, Aram Salihi, and Halvard Sutterud  
*University of Oslo, Oslo, Norway*  
 (Dated: September 13, 2017)

In this project three methods of solving the one-dimensional Poisson equation with Dirchelet boundary conditions are investigated. The equation was discretized, and written as a system of linear equations. This equates to a tridiagonal matrix equation, which was solved using the general Thomas algorithm, a specialized version of the Thomas algorithm and by LU-decomposing the matrix. The numerical error is then analyzed for both methods, and the algorithm speeds are investigated and discussed.

The specialization of the algorithm for our specific problem halved the amount of floating point operations. For large matrices ( $n \sim 10^5$ ) the LU-decomposition requires more memory than what was available, which was not a problem with the Thomas algorithm. Also, the Thomas algorithm produced solutions with smaller errors than the LU method. Therefore the Thomas is deemed most suited for solving the problem examined in this paper.

<b>CONTENTS</b>		Discussion	5
Introduction	1	Conclusion	5
Theory	1	References	5
Discretizing the Poisson equation	1	Appendix	6
Solving a general tridiagonal matrix problem	1	Second derivative	6
Optimizing the Thomas algorithm for a specific case	2	Deriving a general expression for the diagonal elements after forward substituting	6
Solving the general tridiagonal matrix problem by LU-decomposition	2		
Counting the number of FLOPS for each method	3		
The general Thomas algorithm	3		
The special case Thomas algorithm	3		
The LU decomposition method	3		
Analytic solution for a specific problem	3		
Method	4		
The general Thomas algorithm	4		
The optimized algorithm	4		
The LU-decomposition	4		
Error analysis	5		
Timing of the Algorithms	5		
Results	5		
Solution using the algorithms	5		
Error and error analysis	5		
Timing the algorithms	5		



## INTRODUCTION

In this paper numerical solutions to the one-dimensional Poisson equation are examined. Many important differential equations in science can be re-written

## THEORY

### Discretizing the Poisson equation

The one-dimensional Poisson equation with Dirchelet boundary conditions is

$$\frac{d^2u}{dx^2} = -f$$

We can approximate the second derivative of  $u(x)$  using a Taylor expansion and solving for  $\frac{d^2u}{dx^2}$ :

$$\frac{d^2u}{dx^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + O(h^2)$$

The equation is discretized with grid points  $x_1, x_2, \dots, x_n$ . Imposing Dirchelet boundary conditions forces  $x_1 = x_n = 0$ , using the handy notation  $u(x_i + h) = u_{i+1}$  and inserting the approximated second derivative into the Poisson equation yields

$$\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = -f_i$$

where  $h = \frac{1}{n+1}$  is the step size, i.e. the distance between grid points, and  $f_i = f(x_i)$ . This discretized version of the one dimensional Poisson equation is conveniently a linear system of equations, and can therefore be written as the matrix equation

$$\mathbf{A}\mathbf{u} = h^2\mathbf{f}$$

where

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & -1 & 0 & \\ \vdots & 0 & -1 & 2 & -1 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix}$$

is a  $n \times n$  tridiagonal matrix, the only non-zero elements are on, directly above or directly below the diagonal. Row reducing the matrix can therefore be done in an efficient way using the Thomas algorithm, as we shall see in the next paragraph.

### Solving a general tridiagonal matrix problem

A general tridiagonal matrix equation is on the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}$$

where  $v_i = h^2 f_i$  in the case examined in this project. To get the matrix in row reduced echelon form and solve the problem, the  $a_i$ 's are first eliminated by Gaussian elimination. First,  $\frac{a_2}{b_1}$  times the first row is subtracted from the second. This gives

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b'_2 & c_2 & 0 & & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \\ \vdots & 0 & a_4 & b_4 & c_4 & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v'_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}$$

where  $b'_2 = b_2 - \frac{a_2}{b_1}c_1$  and  $v'_2 = v_2 - \frac{a_2}{b_1}v_1$ . For the next row, exactly the same operation is used, but notice how the previous elements  $b'_2$  and  $v'_2$  now have been updated in the last iteration. Generalizing this argument for each row, a general expression for the diagonal elements is

$$b'_i = b_i - \frac{a_i}{b'_{i-1}}c_{i-1}$$

where  $b'_1 = b_1$  and  $i = 2, 3, \dots, n$ . Similarly, elements in the right-hand side vector  $\mathbf{v}'$  are modified to

$$v'_i = v_i - \frac{a_i}{b'_{i-1}}v'_{i-1}$$

where  $v'_1 = v_1$  and  $i = 2, \dots, n$ . After completing the forward substitution, the matrix equation reads

$$\begin{bmatrix} b'_1 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b'_2 & c_2 & 0 & & \vdots \\ 0 & 0 & b'_3 & c_3 & 0 & \\ \vdots & 0 & 0 & b'_4 & c_4 & 0 \\ & & & \ddots & \ddots & \ddots \\ 0 & \dots & \dots & 0 & 0 & b'_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} v'_1 \\ v'_2 \\ v'_3 \\ \vdots \\ v'_n \end{bmatrix}$$

By inspection, the solution for the final point  $u_n$  is explicitly given by the bottom row:

$$b'_n u_n = v'_n \rightarrow u_n = \frac{v'_n}{b'_n}$$

The rest of the solution  $u(x_i)$  is then computed by backward substitution, given  $u_i$  one can compute  $u_{i-1}$ . For instance, row  $n-1$  reads

$$b'_{n-1} u_{n-1} + c_{n-1} u_n = v'_{n-1}$$

$$u_{n-1} = \frac{v'_{n-1} - c_{n-1} u_n}{b'_{n-1}}$$

In general then, the back substitution is done by

$$u_i = \frac{v'_i - c_i u_{i+1}}{b'_i}$$

where  $i = n-1, n-2, \dots, 1$ .

### Optimizing the Thomas algorithm for a specific case

In the case presented in this project,  $a_i = c_i = -1$  and  $b_i = 2$ . The fact that there are just two different values in the matrix greatly simplifies the calculations, since all the calculations involving only  $a_i$ ,  $b_i$  and  $c_i$  can be done beforehand. Also, it is possible to derive an analytic expression for  $b'_i$ . This is given by

$$b'_i = \frac{i+1}{i}$$

and is found and proved by induction in appendix A?

### Solving the general tridiagonal matrix problem by LU-decomposition

Another way to solve matrix equations is LU-decomposing A, and then solving two matrix equations. This is a frequently used method for solving more general sets of linear equations. The LU-decomposition is a form of Gaussian elimination, where A is factored into matrices L and U. L is lower triangular with 1 in each element on the diagonal, and U is upper triangular. The matrix equation is

$$A\mathbf{u} = \mathbf{v}$$

$$LU\mathbf{u} = \mathbf{v}$$

Introducing  $\mathbf{y} = U\mathbf{u}$ , the solution  $\mathbf{u}$  can now be computed in two steps. First,  $\mathbf{y}$  is found by solving

$$L\mathbf{y} = \mathbf{v}$$

which is then inserted in

$$U\mathbf{u} = \mathbf{y}$$

and solved for  $\mathbf{u}$ .

### Counting the number of FLOPS for each method

A simple way of determining the computational speed of an algorithm is simply counting the number of floating point operations. The fewer calculations the computer has to make, the faster an algorithm will run. In this section approximate numbers for the amount of floating point operations (flops) are determined.

#### *The general Thomas algorithm*

The general algorithm with arbitrary tridiagonal elements consists of two steps, the forward substitution (two equations) and the backward substitution (one equation). In the forward substitution, some of the operations can be precomputed. Since  $a_i$ ,  $c_i$  is known for all  $i$  considered, the product  $a_i c_{i-1}$  can be precomputed and does not provide any flops. Therefore, computing a  $b'_i$  requires only two operations, and these are done  $(n-1)$  times. Computing a  $v'_i$  requires three flops, one product, one division and an addition. These operations are also repeated  $(n-1)$  times, so the total amount of flops in the forward substitution is:

$$2(n-1) + 3(n-1) = 5(n-1)$$

In the backward substitution, no quantities can be precomputed as they are produced by the algorithm itself. Counting the number of operations yields  $3(n-1)$  flops. In total, then, the number of flops required in the general Thomas algorithm is

$$\text{flops}_{\text{Thomas}} = 8(n-1)$$

#### *The special case Thomas algorithm*

For the specialized algorithm, even more quantities can be precomputed and the amount of flops can therefore be reduced further. First of all, an analytic expression for  $b'_i$ , the diagonal elements,

does not contribute with any flops as they can now be fully precomputed. The computation of  $v'_i$  now only requires two operations, since  $a_i = -1$  simply changes the sign. The backward substitution now consists of one subtraction and one division operation, so the total amount of flops for the tailored algorithm is

$$4(n-1) \quad (1)$$

This should therefore cut the running time of the algorithm approximately in half.

#### *The LU decomposition method*

Compared to the Thomas algorithm, the LU decomposition requires a much larger amount of flops. Where the Thomas neatly avoids dealing with all the  $\sim (n^2 - 3n)$  zeros, the LU algorithm considers every element in every row of the matrix. It is possible to show that the number of flops for this method is in the order  $\frac{2}{3}N^3$ .

### Analytic solution for a specific problem

In this paper the following special case of the Poisson equation is examined:

$$-\frac{d^2 u}{dx^2} = 100e^{-10x} \quad (2)$$

where  $0 < x < 1$  and  $u(0) = u(1) = 0$ , i.e. Dirchelet boundaries. An analytic solution is obtained by integrating with respect to  $x$  twice

$$u(x) = -e^{-10x} + A + Bx$$

Imposing the boundary condition  $x(0) = 0$  gives  $u(0) = 0 = -1 + A$ , so  $A = 1$ . Applying  $x(1) = 0$  gives  $0 = -e^{-10} + 1 + B$ , so  $B = e^{-10} - 1$ . Inserting the integration constants returns the analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \quad (3)$$

where  $0 < x < 1$  and  $u(0) = u(1) = 0$ .

## METHOD

The algorithms were implemented in C++ and Julia, with the purpose of learning multiple programming languages and verifying that the implementations were correct. The source code is available in the attached GIT-repository. The calculations were performed several times, and for each method the algorithm times were averaged to produce an average time. The implementations are explained and listed in the following paragraphs with pseudocode. The analysis of the solutions and errors was done in Python.

### The general Thomas algorithm

The general Thomas algorithm consists of the forward and backward substitutions examined in the theory section. In pseudocode the loop was implemented as

```
b_prime[0] = b[0];
v_prime[0] = v[0];

// Forward substitution
for(i = 1; i <= n-1; i++)
    b_prime[i] = b[i] - (a[i]/b_prime[i-1])*c[i-1];
    v_prime[i] = v[i] - (a[i]/b_prime[i-1])*v_prime[i-1];
}

// Backward substitution
for(i = n-2; i >= 1; i--){
    u[i] = (v_prime[i] - c[i]*u[i+1])/b_prime[i];
}
```

Listing 1: Thomas algorithm implementation for a general tridiagonal matrix

### The optimized algorithm

Because of the fact that the matrix examined in this paper is both tridiagonal and symmetric, further optimizations to the algorithm can be made. Since  $a_i = c_i = -1$  and  $b'_i = (i+1)/i$  can be precomputed, the algorithm simplifies to

```
// b_prime is now precomputed
for(i = 1; i <= n-1; i++)
    b_prime[i] = (i+1)/i;

v_prime[0] = v[0];

// Forward substitution
for(i = 1; i <= n-1; i++)
    v_prime[i] = v[i] + (v_prime[i-1]/b_prime[i-1]);
}

// Backward substitution
for(i = n-2; i >= 1; i--){
    u[i] = (v_prime[i] + u[i+1])/b_prime[i];
}
```

Listing 2: Optimized version of Thomas algorithm for a symmetric tridiagonal matrix

### The LU-decomposition

Solving the matrix equation using the LU-decomposition method was done with functions from the Armadillo C++ Linear Algebra Library. First, the tridiagonal matrix A is implemented, and then the Armadillo functions `lu()` and `solve()` are used to solve the two factored matrix equations. In pseudocode, the implementation is

```
#include <armadillo>

for (row = 0; row < size-1; row++){
    A(row, row) = 2;
    if (row > 0)
        A(row, row-1) = -1;
    if (row < size-1)
        A(row, row+1) = -1;

    lu(L,U,A);
    y = solve(L, btilde);
    u = solve(U, y);
}
```

Listing 3: Pseudocode of LU-decomposition method implementation

### Error analysis

The relative error of the numerical solution at  $x_i$  is given by

$$\epsilon_i = \left| \frac{v_i - u_i}{u_i} \right|$$

where  $v_i$  is the numerically calculated value and  $u_i$  is the analytic value. The implementation of the analytic solution is rather straightforward. For several  $n$  the errors  $\epsilon_i$  are calculated using the above formula, and the maximum error  $\epsilon_{n,\max}$  is extracted, saved and later plotted against  $n$ . For obvious reasons, for these calculations the optimized version of the Thomas algorithm was used.

### Timing of the Algorithms

In order to get an accurate timing of the algorithms, each computation was repeated  $10^6$  times where this was practical. For the general and special algorithms, this limit was at matrices of size  $10^4$ . For matrices of size  $10^{5-7}$ , as well as for all matrices using LU-decomposition, the computations were repeated  $10^3$  times. The final estimate of their time usage was taken as their averages.

## RESULTS

### Solution using the algorithms

#### Error and error analysis

#### Timing the algorithms

The time used by each algorithm is summarized in table I.

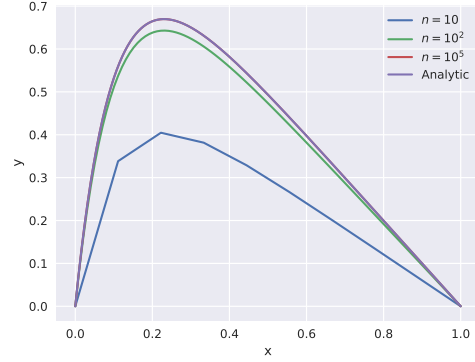


Figure 1: The plot

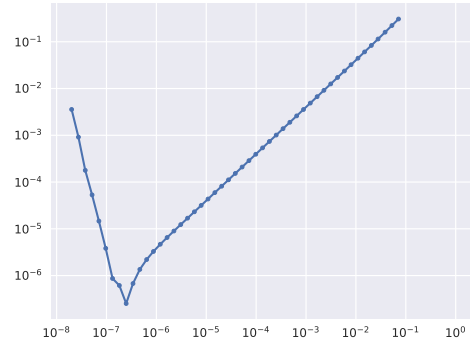


Figure 2: The error

## DISCUSSION

## CONCLUSION

## REFERENCES

Table I: Time usage of each algorithm. These are averages of the time used by each algorithm.

N	Time usage [s]	
	General	Special
$10^1$	$1.592 \times 10^{-7}$	
$10^2$	$1.683 \times 10^{-6}$	
$10^3$	$1.687 \times 10^{-5}$	
$10^4$	$1.682 \times 10^{-4}$	
$10^5$	$1.734 \times 10^{-5}$	

LU

$1.378 \times 10^{-5}$   
 $9.837 \times 10^{-4}$   
 $6.214 \times 10^{-1}$   
 -  
 -

### Deriving a general expression for the diagonal elements after forward substituting

The general expression for the diagonal elements in  $A$  after forward substituting was

$$b'_i = b_i - \frac{a_i}{b'_{i-1}c_i}$$

for  $i = 2, \dots, n$ . Since  $a_i = c_i$  for all  $i$  in the special case examined in this paper, the expression for  $b'_i$  simplifies to

$$b'_i = b_i - \frac{1}{b'_{i-1}}$$

In an attempt at further simplify the expression, a few iterations are calculated, and a pattern seems to emerge:

$$b'_1 = b_1 = 2$$

$$b'_2 = b_2 - \frac{1}{b'_1} = 2 - \frac{1}{2} = \frac{3}{2}$$

$$b'_3 = b_3 - \frac{1}{b'_2} = 2 - \frac{2}{3} = \frac{4}{3}$$

$$b'_4 = b_4 - \frac{1}{b'_3} = 2 - \frac{3}{4} = \frac{5}{4}$$

By inspection,  $b'_i$  seems to follow the pattern

$$b'_i = \frac{i+1}{i}$$

which is assumed valid for  $i = 1, 2, \dots, n$ . A proof by induction is used to show that the formula is indeed correct for all  $i \geq 1$ .

For any integer  $i \geq 1$ , let  $S(i)$  denote the statement

$$S(i) : b'_i = 2 - \frac{1}{b'_{i-1}} = \frac{i+1}{i}$$

where we have defined  $b'_1 = 2$ .

In the case  $i = 1$ ,  $S(1)$  is obviously true because  $b'_1$  is defined as  $b'_1 = b_1 = 2$ . Now, for some fixed  $j \geq 1$ , assume that  $S(j)$  holds. Then,  $b_{j+1}$  is

## Appendix

### Second derivative

In section 2 we introduced a approximated expression for the second derivative. To derive this expression, consider a general Taylor expansion

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!} (x-a)^n$$

Let  $u(x)$  be real valued function and  $x \in \mathbb{R}$ . We will now expand the function  $u(x+h)$  and  $u(x-h)$  around  $a = x$  to third order

$$u(x+h) \approx u(x) + \frac{du}{dx}h + \frac{d^2u}{dx^2} \frac{h^2}{2!} + \frac{d^3u}{dx^3} \frac{h^3}{3!} + \mathcal{O}(h^4)$$

$$u(x-h) \approx u(x) - \frac{du}{dx}h + \frac{d^2u}{dx^2} \frac{h^2}{2!} - \frac{d^3u}{dx^3} \frac{h^3}{3!} + \mathcal{O}(h^4)$$

Notice, the reason why  $u(x-h)$  and  $u(x+h)$  was expanded to third order is because the third derivatives cancel each other out, and therefore the error must lie around the fourth term expansion. Now add  $u(x-h)$  to  $u(x+h)$

$$u(x+h) + u(x-h) \approx 2u(x) + \frac{d^2u}{dx^2}h^2 + 2\mathcal{O}(h^4)$$

Taking the 2 inside  $\mathcal{O}(h^4)$  and solving it for  $\frac{d^2u}{dx^2}$

$$\frac{d^2u}{dx^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2)$$

note that  $\frac{\mathcal{O}(h^4)}{h^2} = \mathcal{O}(h^2)$ . If we let  $h \rightarrow 0$  we will get an exact expression for the second derivative.

$$\frac{d^2u}{dx^2} = \lim_{h \rightarrow 0} \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2)$$



given by

$$b'_{j+1} = 2 - \frac{1}{b'_j}$$

But since  $S(j)$  holds,  $b'_j = \frac{j+1}{j}$ , and

$$b'_{j+1} = 2 - \frac{j}{j+1} = \frac{j+2}{j+1} = \frac{(j+1)+1}{(j+1)}$$

which proves  $S(i)$  is true for all  $i \geq 1$ .