Computational Physics (FYS4150) Project 1

Solving the one-dimensional Poisson equation using Fortran2008

Bjørn Gilje Lillegraven, Lasse Kvasnes & Johan Raniseth

September 10, 2018

https://github.com/bjorngli/fys4150/tree/master/Project1

Abstract

Using two different algorithms, the tridiagonal matrix algorithm and the LU decomposition method, the one-dimensional Poisson equation with Dirichlet boundary conditions is solved. We find that an optimized algorithm is up to 10 000 000 times faster than the LU decomposition method due to the big difference in floating point operations needed for the calculations.

Contents

1 Introduction

This project study concerns solving the one-dimensional Poisson equation numerically using two different algorithms, in this case by utilizing the programming software Fortran2008. The Poisson equation is a partial differential equation broadly used in theoretical physics and mechanical engineering to, for instance, describe the potential field caused by a given charge or mass density distribution. It can be reduced to one dimension by assuming spherical symmetry.

The algorithms being used are the tridiagonal matrix algorithm and the LU decomposition method. We also examine how optimizing the former enhances the efficiency of the algorithm for a tridiagonal matrix with equal numbers along the diagonal and the non-diagonal elements, respectively. The algorithms' efficiency are compared and we find that the tridiagonal matrix algorithms are vastly more efficient than the LU decomposition method. This is shown by comparing the algorithms at different precision levels.

The report contains a review of how the one-dimensional Poisson equation is derived along with showing how the second order derivative is found using the three-point formula, followed by a presentation of the algorithms used. Then the results are presented and discussed before a short conclusion and references are listed.

2 Discussion of Methods

2.1 Theory

2.1.1 The Poisson Equation

A commonly used equation from electromagnetism is Poisson's equation. The electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimensions it reads

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

With a spherically symmetric Φ and $\rho(\mathbf{r})$, which is the case for our project study, the equations simplify to a one-dimensional equation in r, namely

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

which by substituting $\Phi(r) = \phi(r)/r$ reduces the equation to

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

This equation can be rewritten by letting $\phi \to u$ and $r \to x$, leaving us with this simple expression for the general one-dimensional Poisson equation

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

The inhomogeneous term f is given by the charge distribution ρ multiplied by r and the constant -4π. For this project though, the source term f is given as $f(x) = 100e^{-10x}$. This term will be used to calculate an approximation to u which we will call v, to later be compared to the analytic, closed form solution $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

2.1.2 Defining a Linear Set of Equations Based on the Approximation of the Second Derivative

In our project the general Poisson equation in one dimension with Dirichlet boundary conditions will be solved by rewriting it as a set of linear equations. This is the expression we will solve:

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0.$$

The discretized approximation is defined to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$ and the step length is defined as $h = 1/(n+1)$. The boundary conditions are then $v_0 = v_{n+1} = 0$. Our approximation of the second derivative of u using the three point formula[1] is

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \qquad \text{for } i = 1, \ldots, n.$$

We can then rewrite our approximation as

$$-v_{i+1} - v_{i-1} + 2v_i = f_i h^2 \qquad \text{for } i = 1, \ldots, n, \tag{1}$$

and by ignoring the end points $i = 0$ and $i = n + 1$, this equation can be represented as a linear set of equations:

$$\mathbf{Av} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix and $\tilde{\mathbf{b}} = [h^2\tilde{b_1}, h^2\tilde{b_2}, \ldots, h^2\tilde{b_n}]$. To show that we can write our approximation of $u''(x)$ on this form, we write out our matrix $\mathbf{A}$:

$$\begin{bmatrix} b_1 & c_1 & 0 & \ldots & \ldots & 0 \\ a_2 & b_2 & c_2 & 0 & \ldots & \ldots \\ 0 & a_3 & b_3 & c_3 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \ldots & \ldots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \ldots \\ \ldots \\ \ldots \\ \tilde{b}_n \end{bmatrix}$$

This gives us a set of linear equations:

$$\begin{aligned} b_1 v_1 + c_1 v_2 &= \tilde{b_1} \\ a_2 v_1 + b_2 v_2 + c_2 v_3 &= \tilde{b_2} \\ a_3 v_2 + b_3 v_3 + c_3 v_4 &= \tilde{b_3} \\ &\vdots \\ a_{n-1} v_{n-2} + b_{n-1} v_{n-1} + c_{n-1} v_n &= \tilde{b_{n-1}} \\ a_n v_{n-1} + b_n v_n &= \tilde{b_n} \end{aligned}$$

As our b_i-values are equal to 2, and our a_i and c_i are equal to -1 with the exception of a_1 and c_n that are equal to 0, this will give us the same set of equations as in equation (1), remembering the boundary conditions v_0 and v_{n+1} being equal to 0.

[1] See appendices section 5.1

This leaves us with the matrix equation

$$\mathbf{Av} = \tilde{\mathbf{b}},$$

where

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

2.2 Algorithms

Two different algorithms, the tridiagonal matrix algorithm and the LU decomposition method are used for solving the one-dimensional Poisson equation. For the tridiagonal matrix algorithm, also known as the *Thomas Algorithm*[1], which is a simplified form of the Gaussian elimination, we will create two different versions: a general one which will solve any tridiagonal matrix problem and another one made specifically for solving our particular matrix.

2.2.1 The Tridiagonal Matrix Algorithm

Gaussian elimination is used to create the algorithm for solving general $n \times n$ tridiagonal matrices. In this example of how to find the algorithm we will be using a 3×3 matrix to save space. The algorithm will also work for larger matrices. Row reduction is used to firstly execute a forward substitution:

$$\left[\begin{array}{ccc|c} b_1 & c_1 & 0 & \tilde{b_1} \\ a_1 & b_2 & c_2 & \tilde{b_2} \\ 0 & a_2 & b_3 & \tilde{b_3} \end{array}\right] \sim \left[\begin{array}{ccc|c} b_1 & c_1 & 0 & \tilde{b_1} \\ 0 & b_2 - \frac{a_1 c_1}{b_1} & c_2 & \tilde{b_2} - \frac{a_1 \tilde{b_1}}{b_1} \\ 0 & a_2 & b_3 & \tilde{b_3} \end{array}\right]$$

To make the further calculations easier on the eye, some expressions are renamed:

$$\beta_1 = b_1,\ \beta_2 = b_2 - \frac{a_1 c_1}{b_1}\ and\ B_1 = \tilde{b_1},\ B_2 = \tilde{b_2} - \frac{a_1 \tilde{b_1}}{b_1}$$

$$\left[\begin{array}{ccc|c} \beta_1 & c_1 & 0 & B_1 \\ 0 & \beta_2 & c_2 & B_2 \\ 0 & a_2 & b_3 & \tilde{b_3} \end{array}\right] \sim \left[\begin{array}{ccc|c} \beta_1 & c_1 & 0 & B_1 \\ 0 & \beta_2 & c_2 & B_2 \\ 0 & 0 & b_3 - \frac{a_2 c_2}{\beta_2} & \tilde{b_3} - \frac{a_2}{\beta_2} B_2 \end{array}\right]$$

Again we rename:

$$\beta_3 = b_3 - \frac{a_2 c_2}{\beta_2}\ and\ B_3 = \tilde{b_3} - \frac{a_2}{\beta_2} B_2$$

$$\left[\begin{array}{ccc|c} \beta_1 & c_1 & 0 & B_1 \\ 0 & \beta_2 & c_2 & B_2 \\ 0 & 0 & \beta_3 & B_3 \end{array}\right]$$

which leaves us with these general expressions for β and B:

$$\beta_i = b_i - \frac{a_{i-1}c_{i-1}}{\beta_{i-1}} \quad \text{for } i = 2, \ldots, n. \tag{2}$$

and

$$B_i = \tilde{b_i} - \frac{a_{i-1}}{\beta_{i-1}} B_{i-1} \quad \text{for } i = 2, \ldots, n. \tag{3}$$

In Fortran the forward substitution looks like this:

```
! Forward substitution
DO i=3,n
   frac(i) = a(i-1)/b(i-1)          ! n flops
   b(i) = b(i)-(frac(i)*c(i-1))     ! 2n flops
   b_(i) = b_(i) - b_(i-1)*frac(i)  ! 2n flops
ENDDO
```

Note that we overwrite the values $b(i)$ and $b\_(i)$ which correspond to the values β_i and B_i from equations 2 and 3 respectively.

We have now created an upper triangular matrix by forward substitution. Next we want to make the matrix lower diagonal as well so that we can read out the solution. This is again done by row operations which we will use to find the backward substitution algorithm.

$$\left[\begin{array}{ccc|c} \beta_1 & c_1 & 0 & B_1 \\ 0 & \beta_2 & c_2 & B_2 \\ 0 & 0 & \beta_3 & B_3 \end{array}\right] \sim \left[\begin{array}{ccc|c} \beta_1 & c_1 & 0 & B_1 \\ 0 & \beta_2 & 0 & B_2 - \frac{c_2}{\beta_3}B_3 \\ 0 & 0 & \beta_3 & B_3 \end{array}\right] \sim$$

$$\left[\begin{array}{ccc|c} \beta_1 & 0 & 0 & B_1 - \frac{c_1}{\beta_2}(B_2 - \frac{c_2}{\beta_3}B_3) \\ 0 & \beta_2 & 0 & B_2 - \frac{c_2}{\beta_3}B_3 \\ 0 & 0 & \beta_3 & B_3 \end{array}\right] \sim \left[\begin{array}{ccc|c} \beta_1 & 0 & 0 & B_1 - \frac{c_1}{\beta_2}(B_2 - \frac{c_2}{\beta_3}B_3) \\ 0 & \beta_2 & 0 & B_2 - \frac{c_2}{\beta_3}B_3 \\ 0 & 0 & 1 & \frac{1}{\beta_3}B_3 \end{array}\right]$$

$$\sim \left[\begin{array}{ccc|c} \beta_1 & 0 & 0 & B_1 - \frac{c_1}{\beta_2}(B_2 - \frac{c_2}{\beta_3}B_3) \\ 0 & 1 & 0 & \frac{B_2 - \frac{c_2}{\beta_3}B_3}{\beta_2} \\ 0 & 0 & 1 & \frac{1}{\beta_3}B_3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & \frac{B_1 - \frac{c_1}{\beta_2}(B_2 - \frac{c_2}{\beta_3}B_3)}{\beta_1} \\ 0 & 1 & 0 & \frac{B_2 - \frac{c_2}{\beta_3}B_3}{\beta_2} \\ 0 & 0 & 1 & \frac{1}{\beta_3}B_3 \end{array}\right]$$

Lastly if we write out the matrix into a linear set of equations we get:

$$u_3 = \frac{1}{\beta_3}B_3, \quad u_2 = \frac{B_2 - \frac{c_2}{\beta_3}B_3}{\beta_2} = \frac{B_2 - c_2 u_3}{\beta_2}$$

$$u_1 = \frac{B_1 - \frac{c_1}{\beta_2}(B_2 - \frac{c_2}{\beta_3}B_3)}{\beta_1} = \frac{B_1 - c_1 u_2}{\beta_1}$$

This gives us the algorithms

$$u_n = \frac{1}{\beta_n}B_n \quad \textit{and} \quad u_i = \frac{B_i - c_i u_{i+1}}{\beta_i} \quad \text{for } i = n-1, n-2, \ldots, 1.$$

The backward substitution in Fortran then becomes:

```
! Backward substitution
 u(n) = b_(n)/b(n)
 DO i=n-1,2,-1
      u(i) = (b_(i)-c(i)*u(i+1))/b(i) ! 3n flops
 ENDDO
```

Adding the FLOPS (floating point operations) together gives a total of $8n$ FLOPS for these algorithms.

Because our matrix $\mathbf{A}$ is symmetric, the number of FLOPS needed for calculating the linear set of equations can be reduced. The forward substitution gives these simple expressions for β and B:

$$\beta_i = b_i - \frac{1}{\beta_{i-1}} \quad \text{for } i = 2, \ldots, n. \tag{4}$$

and

$$B_i = \tilde{b_i} - \frac{B_{i-1}}{\beta_{i-1}} \quad \text{for } i = 2, \ldots, n. \tag{5}$$

and the backward substitution gives these:

$$u_i = \frac{B_i + u_{i+1}}{\beta_i} \quad \text{for } i = n-1, n-2, \ldots, 1. \tag{6}$$

This results in an optimized algorithm needing only $4n$ FLOPS, which should make it twice as efficient as the general algorithm:

```
! Forward substitution
DO i=3,n
     b_(i) = b_(i) + b_(i-1)/b(i-1) ! 2n flops
ENDDO

! Backward substitution
u(n) = b_(n)/b(n)
DO i=n-1,2,-1
     u(i) = (b_(i)+u(i+1))/b(i) ! 2n flops
ENDDO
```

2.2.2 The LU Decomposition Method

A point of interest is to look at how our tridiagonal algorithm compares up to other ways of solving a linear set of equations. In this case we compare it up against the LU decomposition method which is based on rewriting the matrix A as the product between a lower diagonal matrix and an upper diagonal matrix.

$$\mathbf{Av} = \tilde{\mathbf{b}} \rightarrow \mathbf{LUv} = \tilde{\mathbf{b}}$$

$$\begin{bmatrix} l_{1,1} & 0 & \ldots & 0 \\ l_{2,1} & l_{2,2} & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n,1} & \ldots & \ldots & l_{n,n} \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \ldots & u_{1,n} \\ 0 & u_{2,2} & \ldots & \vdots \\ \vdots & 0 & \ddots & \vdots \\ 0 & \ldots & 0 & u_{n,n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \vdots \\ \tilde{b}_n \end{bmatrix}$$

Then defining $\mathbf{Uv}$ as $\mathbf{y}$ we can first solve $\mathbf{Ly} = \tilde{\mathbf{b}}$ to find $\mathbf{y}$, and then solve $\mathbf{Uv} = \mathbf{y}$ afterwards.

Solving the one-dimensional Poisson equation with the LU decomposition method in Fortran was done using the **lu_decompose** and **lu_linear_decompose** subroutines from the module **F90library**[2]. This module was translated from Fortran77 to Fortran90 by Morten Hjorth-Jensen and can be found at this GitHub adress.

The clear downside of using the LU decomposition method is that when used numerically it stores and does operations on the whole $n \times n$ matrix whereas the tridiagonal algorithm only stores a few vectors with n values each. Therefore the LU method is highly inefficient in this case where most of our matrix A consists of zeros which are not needed for the calculations. This severely affects the number of FLOPS, which is on the order of $\frac{2}{3}n^3$[3], a number that will quickly grow very big as n becomes large.

3 Results

3.1 Presentation of Data

Table 1 below shows the times the different algorithms took to run for various values of n. Notice that for $n > 10^4$ there are no values for the LU decomposition method. This is due to the fact that the computer doesn't have enough memory to run the calculations and therefore cannot execute them. Regardless of that the pattern is quite clear: the LU decomposition is a lot slower than the other two algorithms, while the difference in computing times for the general and optimized TD solvers is not very big, although it seems to increase with higher n values.

Comparison of computing times (in seconds)			
n	General TD solver	Optimized TD solver	LU decomposition
10	8.56×10^{-6}	8.51×10^{-6}	3.59×10^{-5}
10^2	1.35×10^{-5}	1.22×10^{-5}	2.78×10^{-3}
10^3	6.69×10^{-5}	4.42×10^{-5}	2.23
10^4	6.76×10^{-4}	4.62×10^{-4}	4408.60
10^5	5.44×10^{-3}	3.70×10^{-3}	N/A
10^6	3.82×10^{-2}	2.26×10^{-2}	N/A
10^7	3.51×10^{-1}	2.23×10^{-1}	N/A

Table 1: Comparison of computing times for the different algorithms.[2]

[2]All computing times except the one for LU decomp. for $n = 10^4$ has been run 10 times and then been averaged over. However, due to the long run time of the LU decomposition method for large n values, this is the value from only one time of running it, meaning that this value might not be as representable the other ones found in the table.

Figure 1 and 2 display the fit of our general and optimized algorithms compared to the closed form solution $u(x)$ for $n = 10$ and $n = 100$. For $n = 10$ the fit is good, but not great, while already for $n = 100$ it seems to give very good results.

Figure 3 shows the maximum relative error for different h-values for our optimized TD solver. As we would expect, the relative error decreases with smaller values of h, but at a certain point this trend stops and the relative error increase as h becomes even smaller. The cause of this is precision loss due to round-off errors in the computer when trying to represent very small numbers.

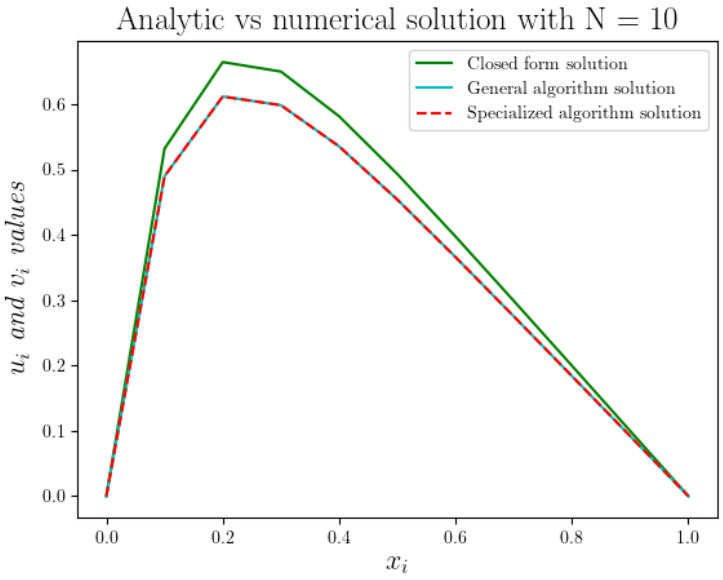

Figure 1: The closed form solution compared to our algorithms, calculated with N=10.

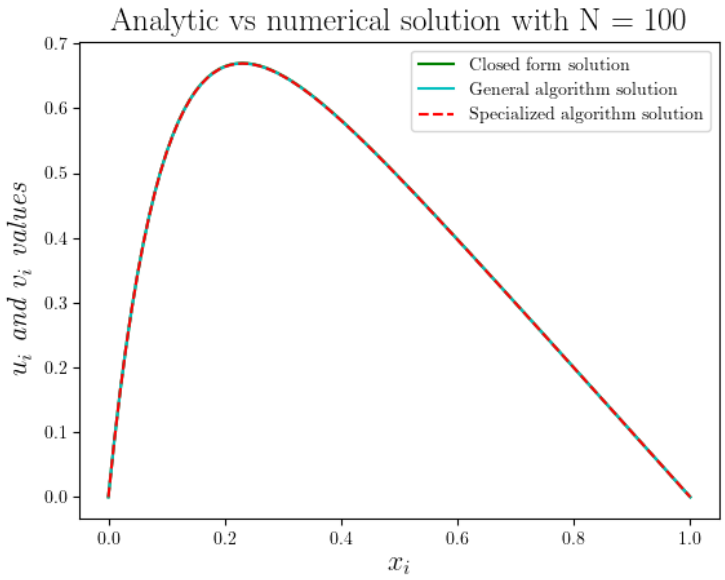

Figure 2: The closed form solution compared to our algorithms, calculated with N=100.

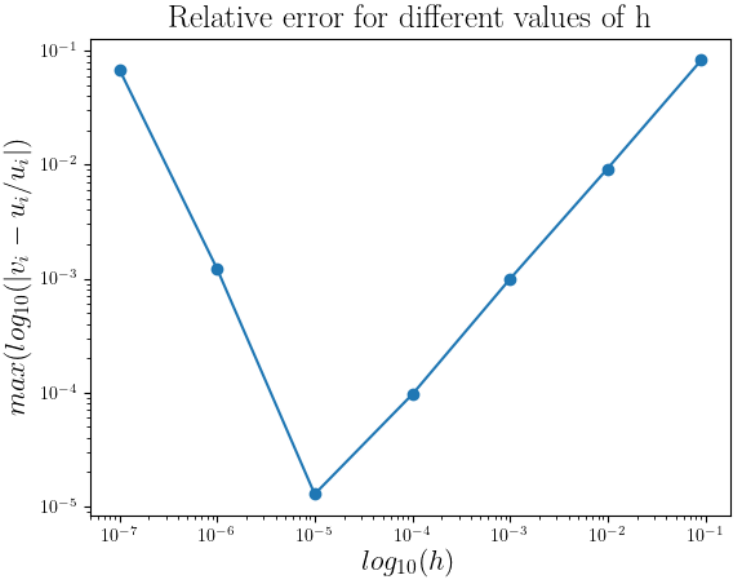

Figure 3: Plot of relative error for different values of h for our optimized tridiagonal matrix algorithm.

3.2 Discussion of Findings

The value of n decides how fast and precise our algorithms produce results. Therefore it is important to take time and try to understand which values of n will produce good enough results while still making the algorithms run relatively quick. In our case (for the optimized TD solver) it seems that a value of $n = 10^5$ gives the smallest numerical error while still being very efficient and not taking up much memory in the computer.

It seems reasonable to state that both our general and optimized tridiagonal matrix algorithms are the way to go when solving a problem like we have in this project. Both computing time and storage counts in favor of using our specially made algorithms rather than the LU decomposition method. The LU subroutines are great for solving smaller matrices that are not tridiagonal, but in our case they are not that useful.

4 Conclusions and Perspective

In this study we have experienced the great advantage it is to choose an efficient algorithm made especially for solving a particular case, as it saves both time and memory in the computer. We have experienced how computer memory is not infinite and that different computers run algorithms at different speeds. We now know that for solving a tridiagonal matrix problem, making a optimized algorithm is way more efficient than the LU decomposition method, as the optimized one uses $O(8n)$ FLOPS compared to the $O(\frac{2}{3}n^3)$ needed for solving it with the LU decomposition method.

5 Appedices

5.1 Taylor Approximation of the Second Order Derivative

Taylor expansion can be used to approximate the second order derivative of a function. One can represent the expression $u(x\pm h)$ like this:

$$u(x\pm h) = u(x)\pm hu' + \frac{h^2}{2!}u''\pm\frac{h^3}{3!}u''' + O(h^4)$$

which gives

$$u(x+h) + u(x-h) = 2u(x) + 2\frac{h^2}{2!}u'' + O(h^4).$$

This then yields the following expression for the second order derivative

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

and by using the assumption that $x_i = ih$, we end up with the expression as a linear set of equations:

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

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] Press, W. H. et. al., (1992). *Numerical Recipes in Fortran 77, The Art of Scientific Computing* 2nd ed. Press Syndicate of the University of Cambridge, New York.

[3] Hjorth-Jensen, M. (2015). *Computational Physics - Lecture Notes 2015.* University of Oslo.