

# Project 1

FYS4150 - Computational Physics

Even S. Håland

# Abstract

In this project an algorithm for solving the one-dimensional Poisson equation is developed, and the numerical solution is compared to an analytical one. A key point of the project is a study of the error of the numerical solution, from which we learn that we not necessarily gain anything by choosing extremely small step sizes in the iterations. We will also see that the problem can be very easily solved by using a library function, but that this is might not be a good idea.

## 1 Introduction

The purpose of this project is to develop an algorithm that will be used to find a numerical solution to the one-dimensional Poisson equation

$$-u''(x) = f(x), \quad (1)$$

where  $x \in (0, 1)$ , with the Dirichlet boundary conditions  $u(0) = u(1) = 0$ . It will be assumed that the source term ( $f(x)$ ) takes the form

$$f(x) = 100e^{-10x}. \quad (2)$$

The solution we obtain numerically will be compared to a closed-form solution given by

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

We can easily show that this is a solution to eq. (1) by taking the first and second derivatives:

$$\begin{aligned} u'(x) &= -(1 - e^{-10}) + 10e^{-10x} \\ \Rightarrow u''(x) &= -100e^{-10x} \\ \Rightarrow -u''(x) &= 100e^{-10x} = f(x). \end{aligned}$$

The algorithm will be developed in two different stages; first a general one, and then a simplified one that deals with the particular problem in this project. The two versions of the algorithm will be compared in terms of number of floating point operations and CPU time.

## 2 Discretization of the Poisson equation

To solve something numerically we need to make a discrete approximation to the problem. In this case we approximate  $u(x)$  by  $v(x_i) = v_i$ , with  $x_i = ih$  in the interval  $x_0 = 0$  to  $x_{n+1} = 1$ . The step size is defined by  $h = 1/(n + 1)$ . The boundary conditions are now given by  $v_0 = v_{n+1} = 0$ .

The second derivative is approximated by

$$u''(x) \approx \frac{v_{i+1} + v_{i-1} - 2v_i}{h^2}, \quad (4)$$

meaning that our problem can be written as

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \quad (5)$$

where  $f_i = f(x_i)$  and  $i = 1, \dots, n$ . To simplify the expression a little bit we multiply both sides by  $h^2$ , and define  $y_i = h^2 f_i$ .<sup>1</sup>

Let us now write eq. (5) explicitly for some values of  $i$  (and keep in mind that  $v_0 = v_{n+1} = 0$ ):

$$\begin{aligned} i = 1 & \Rightarrow -v_2 + 2v_1 = y_1 \\ i = 2 & \Rightarrow -v_3 - v_1 + 2v_2 = y_2 \\ i = 2 & \Rightarrow -v_4 - v_2 + 2v_3 = y_3 \\ & \vdots \\ i = n & \Rightarrow -v_{n-1} + 2v_n = y_n \end{aligned}$$

It is now relatively easy to see that this can be written as a matrix equation if we define

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \quad \text{and} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

The equation can then be written as  $\mathbf{A}\mathbf{v} = \mathbf{y}$ , where  $\mathbf{A}$  is the tridiagonal  $n \times n$ -matrix given by

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix},$$

meaning that to solve the problem we need to solve this equation for  $\mathbf{v}$ .

---

<sup>1</sup>In the project description it is suggested to use  $\tilde{b}_i = h^2 f_i$ , but I found that this can be quite confusing, as  $b$  is used to denote the diagonal elements of the matrix  $\mathbf{A}$ , meaning that  $\tilde{b}$  is very convenient to use when doing the Gauss elimination.

### 3 Developing the algorithm

When solving a matrix equation we make use of Gaussian elimination. A general tridiagonal matrix can be written in terms of vectors  $a$ ,  $b$  and  $c$  as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_1 & b_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & c_3 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_n \end{pmatrix},$$

so what we actually want to do is to eliminate all  $a$ 's in the matrix. We see that to eliminate  $a_1$  we must multiply the first row by  $\frac{a_1}{b_1}$ , and then subtract it from the second row, meaning that the second row becomes

$$\left( 0 \quad b_2 - \frac{c_1 a_1}{b_1} \quad c_2 \quad 0 \quad \cdots \quad 0 \right).$$

(Notice that the  $c$ 's are not affected by this procedure.) To simplify the expressions we would like to define

$$\tilde{b}_2 \equiv b_2 - \frac{c_1 a_1}{b_1}.$$

The procedure then continues in the same pattern, and we realize that the new diagonal elements ( $\tilde{b}$ 's) can be written as

$$\tilde{b}_i = b_i - \frac{a_i c_{i-1}}{\tilde{b}_{i-1}}. \quad (6)$$

We must also remember to do the same operations on the right hand side,

$$\tilde{f}_i = y_i - \frac{a_i \tilde{f}_{i-1}}{\tilde{b}_{i-1}}. \quad (7)$$

The procedure of updating  $\tilde{b}_i$  and  $\tilde{f}_i$  is referred to as *forward substitution*.

When the elimination is done we need to do a *backwards substitution* to actually solve the equation for each  $v_i$ . Since we now have a diagonal matrix we should start at the last equation, i.e.  $\tilde{b}_n v_n = \tilde{f}_n$ , which means that

$$v_n = \frac{\tilde{f}_n}{\tilde{b}_n}.$$

Now that  $v_n$  is known we can move to the second last equation, solve it for  $v_{n-1}$ , and so on. The general expression for  $v_i$  becomes

$$v_i = \frac{\tilde{f}_{i+1} - c_i v_{i+1}}{\tilde{b}_i}.$$

Before moving on to how I have coded this, and finally look at results, I would like to get all the "maths" done by looking at how we in our case can simplify the above expressions.

### 3.1 Simplifying the algorithm

We have now looked at how the matrix equation is solved when  $\mathbf{A}$  is a general tridiagonal matrix. However, the matrix we are studying is simplified quite a bit, as all  $a_i = c_i = -1$ , while all  $b_i = 2$ . The first thing to notice is that eq. (6) can be written as

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}}.$$

By writing the first few terms

$$\tilde{b}_2 = 2 - \frac{1}{2} = \frac{3}{2}, \quad \tilde{b}_3 = 2 - \frac{1}{3/2} = \frac{4}{3}, \quad \tilde{b}_4 = 2 - \frac{1}{4/3} = \frac{5}{4},$$

and so on, we realize that we can actually write the  $\tilde{b}$ 's as

$$\tilde{b}_i = \frac{i+1}{i}.$$

By doing the same for the  $\tilde{f}$ 's we find that

$$\tilde{f}_i = y_i + \frac{i-1}{i} \tilde{f}_{i-1} = y_i + \frac{\tilde{f}_i}{\tilde{b}_{i-1}},$$

and finally

$$v_{i-1} = \frac{i-1}{i} (\tilde{f}_{i-1} - v_i) = \frac{\tilde{f}_{i-1} - v_i}{\tilde{b}_{i-1}}.$$

When doing these simplifications the total number of floating point operations (FLOPS) is reduced quite a bit. Since  $\tilde{b}_i$  only depends on  $i$  the full  $\tilde{\mathbf{b}}$  can be calculated before starting the algorithm, which reduces the number of FLOPS from  $6(n-1)$  to  $2(n-1)$  in the forward substitution, and from  $3n$  to  $2n$  in the backwards substitution.

### 3.2 Coding the algorithm

At the moment of writing this report I have only written the code in Python, and the code can be found as `Project1.py` in the following git-repository:

<https://github.com/evensha/FYS4150/tree/master/Project1/Program>

The program take two input arguments:

1. An argument that specify if we want to run the general or the simplified algorithm ("g" for general and "s" for simplified).
2. The matrix dimension,  $n$ .

When writing the code I have chosen to let all arrays have  $n + 2$  elements, i.e.  $i = 0, 1, \dots, n, n + 1$ , to make the arrays correspond to the total number of grid points (including the boundary points), and to make the code as close to the mathematical expressions as possible.

After making the necessary arrays ( $a, b, c, x, f$  and  $v$ ) I make the four different loops, i.e. forward and backward substitution in both the general and the simplified way. The loops just update the values of the array elements of  $b, f$  and  $v$ , meaning that I have chosen not to make separate arrays called  $\tilde{b}, \tilde{f}$ , and so on. While running the algorithm I also calculate the CPU time. When the algorithm is done I make an array for the closed form solution, and plot it together with the numerical solution from the algorithm, and the (logarithm of) the relative error is calculated by the formula

$$\epsilon_i = \log_{10} \left( \left| \frac{v_i - u_i}{u_i} \right| \right). \quad (8)$$

All results are shown in the next section.

The last part of the program is to solve the problem by using library functions. The library `numpy.linalg` [1] contains a function called `solve()`, which simply takes  $\mathbf{A}$  and  $\mathbf{f}$  as input, and then "spits" out the solution,  $\mathbf{v}$  is found by calling  $\mathbf{v} = \text{solve}(\mathbf{A}, \mathbf{f})$ . (This function makes use of LAPACK [2], which solves the equation by LU decomposition.) The purpose of doing this is to compare the efficiency of the built-in functions with our algorithm, so I also calculate the CPU time spent on this operation.

## 4 Results

Figure 1 shows the numerical solution plotted together with the closed-form solution for 10, 100 and 1000 grid points. When  $n = 10$  we can see that the numerical estimate is somewhat different from the analytical solution, while for  $n = 100$  and  $n = 1000$  they seem to be very similar. (If one zooms closely in on the  $n = 100$  plot it is possible to see a slight difference around the maximum of the curve, while for  $n = 1000$  there seems to be no observable difference between the curves.)

The point of developing the simplified algorithm is of course to reduce the amount of time we spend on solving the problem. The CPU time for spent for the two different algorithms are given in Table 1. We see that for  $n = 10$  the CPU time is almost the same, while for larger values of  $n$  the simplified algorithm is about twice as fast as the general one.

In Table 2 the *log-log* relation between  $h$  and  $\epsilon_i$  (defined in eq. (8)) is given. By only considering the mathematical error in the approximation to the second derivative we expect that  $\epsilon_i$  increases by a factor of 2 when  $n$  is increased by a factor of 10. We see in the table that this holds up to and including  $n = 10^4$ . When further increasing  $n$  we can see that the error due to limited numerical precision starts to dominate.

At last our algorithm is compared to the `solve()`-function from `numpy.linalg()`. The results are given in Table 3, and we can clearly see why it is a bad idea to use library functions to solve this problem. Firstly, it is a lot slower because a full

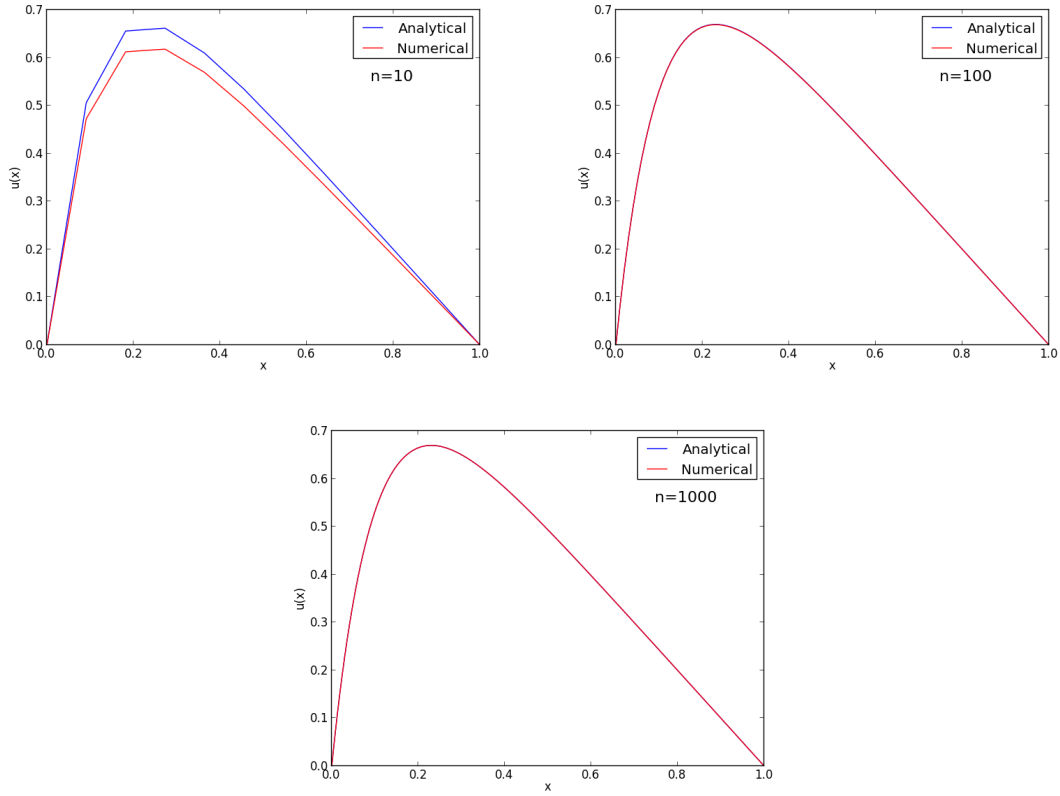


Figure 1: Analytical and numerical solutions for  $n = 10, 100$  and  $1000$ .

$n$	CPU time (s)	
	General alg.	Simplified alg.
$10^1$	$3.5 \cdot 10^{-5}$	$3.1 \cdot 10^{-5}$
$10^2$	$3.3 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$
$10^3$	$3.3 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$
$10^4$	$3.2 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$
$10^5$	0.33	0.17
$10^6$	3.28	1.66

Table 1: CPU time comparisons for the general and simplified algorithm.

LU decomposition is done, which requires  $\sim n^3$  FLOPS, while we found that our simplified algorithm only requires  $\sim 4n$  FLOPS. Secondly, at  $n = 10^5$  we run out of memory when trying to define  $\mathbf{A}$ . A matrix of size  $10^5 \times 10^5$  is simply too large to handle for the computer's memory.

## 5 Summary and conclusions

In this project we have seen how the Poisson equation can be written in a discrete way as a matrix equation,  $\mathbf{A}\mathbf{v} = \mathbf{f}$ , where  $\mathbf{A}$  was found to be a tridiagonal matrix.

$n$	$\log_{10}(h)$	$\epsilon_i$
$10^1$	-1.041	-1.180
$10^2$	-2.004	-3.088
$10^3$	-3.000	-5.080
$10^4$	-4.000	-7.079
$10^5$	-5.000	-8.843
$10^6$	-6.000	-6.075
$10^7$	-7.000	-5.525

Table 2:  $\epsilon_i$  as function of  $\log_{10}(h)$ .

$n$	CPU time (s)	
	<code>solve()</code>	Simplified alg.
$10^1$	$4.7 \cdot 10^{-4}$	$3.1 \cdot 10^{-5}$
$10^2$	$6.2 \cdot 10^{-3}$	$1.8 \cdot 10^{-4}$
$10^3$	$8.2 \cdot 10^{-2}$	$1.7 \cdot 10^{-3}$
$10^4$	16.6	$1.7 \cdot 10^{-2}$
$10^5$	MemoryError	0.17

Table 3: CPU time comparisons for the `solve()`-function in `numpy.linalg()` and our simplified algorithm.

First we developed an algorithm that solved the equation(s) for a general tridiagonal matrix, and then we saw how the algorithm could be speeded up because our matrix  $\mathbf{A}$  was particularly simple.

By comparing the numerical solution from our algorithm with an analytical one, and we saw that we were able to find a good approximation to the analytical solution when choosing small enough step size,  $h$ . However, we have also seen that even though we continue decreasing  $h$  the numerical solution does not necessarily become better and better, since the numerical precision is limited.

Finally we saw that a matrix equation can be solved using the `solve()`-function in the `numpy.linalg` library. This is a very simple way of solving the problem, but it is much slower than the algorithm we developed, and we get memory issues when  $\mathbf{A}$  is sufficiently large.

## References

- [1] *Linear algebra* (`numpy.linalg`), The Scipy community.  
<https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.linalg.html>
- [2] LAPACK - *Linear Algebra PACKage*  
<http://www.netlib.org/lapack/>