Project 1: Computational Physics - FYS3150

Fredrik Hoftun & Mikkel Metzsch Jensen

September 09, 2020

Contents

Abstract

The goal of this project were to...
What did we do?
What did we find?

1 Introduction

In this project we will investegate different numerical approaches for the solving of the one-dimensional Poisson equation with Dirichlet boundary conditions. This is a problem that is used in many different scientiffic applications. It is used to describe electrostatic and megnetostatic phenonema in a quantitative maner. It is also helps to understand diffusion and propagation problems. The solution is generally used in a wide range of fields such as engineering, physics, mathematics, biology, chemistry, etc. [1]. In this project we have used a second derivative approximation in order to rewrite the problem as a set of linear equations. We have written three different algorithms: A general and a specialiazed algorithm using gaussian elimination and then LU demposition. Theese are described individually in the method sections along with a calculation of the number of floating point operation for each of them. Since we know the analytical solution for our problem we have compaired relative error for different choices of stepsize h in the algorithm. We have also compaired of the CPU time of theese computationg. We found a significant difference in both precesion (due to round of erros) and CPU time, which are presented in the results section.

1. Motivate the reader
2. What have I done
3. The structure of the report
4. conlusion?

2 Method

2.1 Defining the problem

We are going to solve the one-dimensional Poisson equation with Dirichlet boundary conditions given as the following:

$$u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0$$

In our case we will use the function:

$$f(x) = 100e^{-10x}$$

Note that the algorithms for solving the problem will not be depedent on this specefic choice of f(x). The reason why we stick to this function is for the praticality of having the following analytical solution for f(x):

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

We can ensure that this is a valid Sslution by inserting it into the Poission equation. First we calculate double derivative of u(x):

$$u'(x) = -(1 - e^{-10}) + 10e^{-10x}, \quad u''(x) = -100e^{-10x}$$

We now see that the solution satisfies the Poisson equation:

$$-u''(x) = 100e^{-10x} = f(x)$$

We can therefore use this analytical solution to evaluate the precision of the numerical solutions for different choices of step length h.

2.2 Rewritting the problem as a set of linear equations

In order to solve the Poisson equation numerically we discretize u as v_i with $n+2$ grid points $x_i = ih$ for $i = 0, 1, \ldots, n+1$. To clarify we have $x_0 = 0$, $x_{n+1} = 1$ which are spaced with step length $h = 1/(n+1)$. The boundary conditions is then $v_0 = v_{n+1} = 0$. We use the following second derivative approximation

$$-u''(x_i) \approx -\tfrac{v_{i+1} + 2v_i - v_{i+1}}{h^2} = f(x_i) \quad \text{for } i = 1, \ldots, n$$

$\iff$

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f(x_i)$$

Notice that we cannot calculate the second derivative approximation at the end points $i = 0$ and $i = n+1$ since we need avaliable points $v_{\pm 1}$ for the calculation. We define the colum vector $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ and try to setup the equation for every step $i = 1, \ldots, n$. As we do this we see a usefull pattern appearing

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 f(x_1)$$

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f_n \end{bmatrix}$$

From this we see that we can write the problem as a linear set of equation:

$$\mathbf{Av} = \mathbf{g}$$

With the following definitions:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \quad , \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \quad , \tilde{\mathbf{g}} = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f_n \end{bmatrix}$$

2.3 General solution using Gausian elimination

We can solve our set of linear equations using Gaussian elimination on the matrix $\mathbf{Av} = \mathbf{g}$. In the beginning we assume a more generilazed matrix (A) as:

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

Here a_i are the elements below the diagonal, b_i are the elements on the diagonal and c_i are the elements above the diagonal. In order to solve this we use first a forward substitution and then a backwards substitution [2]. The implementation of this is showed in Algorithm 1:

Algorithm 1 General algorithm

```
for i = 2, ..., n do                 ▷ Forward substitution eliminating a_i
    b_i = b_i − c_{i−1} · a_i / b_{i−1}                          ▷ Update b_i
    g_i = g_i − g_{i−1} · a_i / b_{i−1}                          ▷ Update g_i
end for

v_0 = v_{n+1} = 0                   ▷ Backward substitution obtaining v_i
v_n = g_n / B_n;
for i = n − 1, ..., 1 do
    v_i = (g_i − c_i · v_{i+1}) / b_i
end for
```

By running this algorithm for decreasing step length we se that the numerical solution aproaches the analytical quite fast (see figure 1)

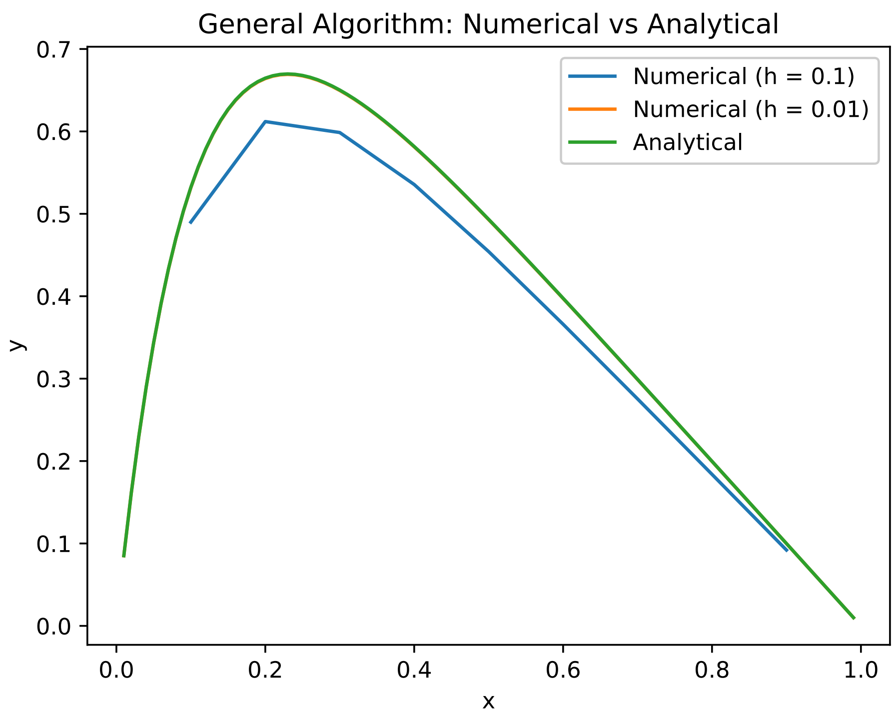

Figure 1: The figure shows the analytical solution and the numerical solution using the generel algorithm for $h = 0.1$ and $h = 0.01$ respectively. We see that the numerical solution lines up with the analytical solution already for $h = 0.01$.

We can calculate the algorithms number of Floating Point Operations. In our forward substitution we have $2 \cdot 3$ operations for each loop which runs for a total of n-1 times. In the backward substitution we have a single leading operation and then 3 operations which loops for a total of n-1. The total number of operations is then

$$(6+3)(n-1)+1 = 9n-8$$

For large n we can approximate this as 9n.

2.4 Simplified specific solution

Since we have fixed value $a = -1$, $b = 2$, $c = -1$ for the matrix A introduced in previous section, we can simplify our algorithm even more.

Now we can do the forward substitution before hand: **A**:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix}$$

$$\sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & 0 & 4/3 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & -1 & 2 & -1 \\ 0 & \cdots & & 0 & -1 & 2 \end{bmatrix} \sim \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 3/2 & -1 & 0 & \cdots & \cdots \\ 0 & 0 & 4/3 & -1 & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & 0 & i_n/i_n - 1 & -1 \\ 0 & \cdots & & 0 & 0 & i_n + 1/i_n \end{bmatrix}$$

We see that the updated diagonal element b_i follows the formula:

$$b_i = \frac{i+1}{i}$$

This means that the update of the diagonal element now use 2 floating point operations per n in where the general one used 3 per n. But the most important fact is that this can be precalculated and in practice excluded from the algorithm. We can therefore justify not to count theese floating point operations. The update of g_i with known values for a, b and c simplifies to:

$$g_i = g_i + g_{i-1}/b_{i-1}$$

[2]. This gives us the following simplified algorithm:

Algorithm 2 Special algorithm, where $a_i = -1,\ b_i = 2,\ c_i = -1$

for $i = 2, \ldots, n$ **do** ▷ Forward substitution eliminating a_i
  $b_i = i + 1/i$ ▷ Update b_i
  $g_i = g_i + g_{i-1}/b_{i-1}$ ▷ Update g_i
end for

$v_n = 0$ ▷ Backward substitution obtaining v_i
for $i = n-1, \ldots, 1$ **do**
  $v_i = \frac{g_i + v_{i+1}}{b_i}$
end for

Similarly to the general algorithm we can calculate total number of floating point operations (not including calculation of b_i):

$$(2+2)(n-2) = 4n - 8$$

For large n this can be approximated to $4n$.
PLOT OR EXAMPLE OF CODE WORKING?

2.5 LU decomposition

For the LU decomposition we ...
FLOPS $O(n^3)$ se `https://en.wikipedia.org/wiki/LU_decomposition`, søk etter "float"

2.6 Comparing precision and error

In order to compare the precesion of the different solutions we will use the relative error bwteen the numerical v_i and analytical $f(x_i) solutionasfollow$

2.7 Implementation

We used `C++` to implement our algorithms ... and we used Pythons library Matplotlib to plot our results.

3 Results

GITHUB LINK HERE
Max error:

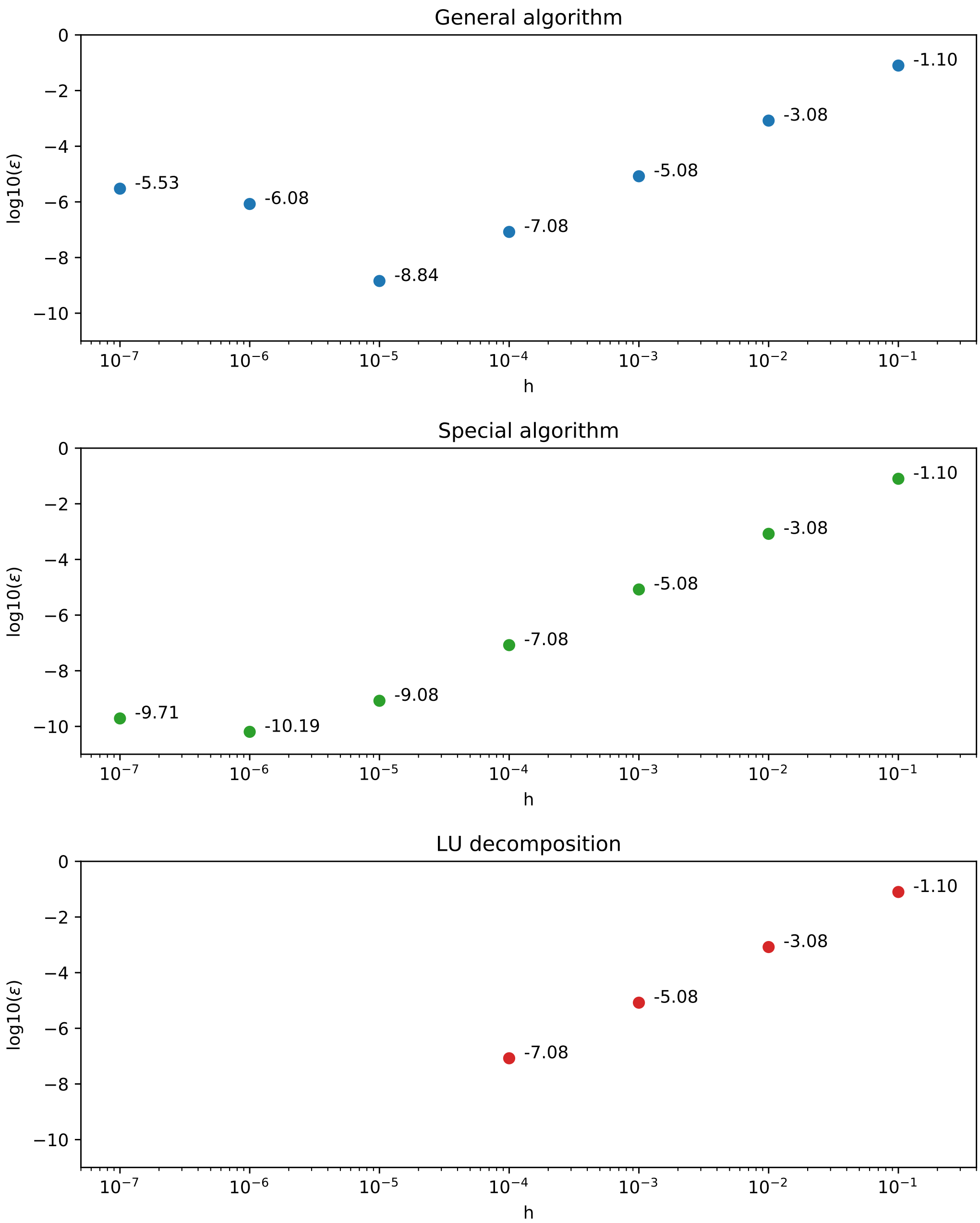

Figure 2: Log10 of the maximum error for each numerical solution compared to the analytical solution for different number of gridpoints n and different numerical methods.

Table 1: CPU Time

N	**General algorithm: t [s]**	**Special algorithm: t [s]**	**LU Decomposition: t [s]**
10^1	3×10^{-6}	3×10^{-6}	9.81×10^{-4}
10^2	4×10^{-6}	4×10^{-6}	1.96×10^{-4}
10^3	3.8×10^{-5}	3.7×10^{-5}	1.02×10^{-2}
10^4	3.41×10^{-4}	3.54×10^{-4}	2.45
10^5	3.79×10^{-3}	3.50×10^{-3}	nan
10^6	3.57×10^{-2}	3.24×10^{-2}	nan
10^7	3.16×10^{-1}	3.24×10^{-1}	nan

Computer specs:

CPU = 2.5 GHz Intel Core i5

3.1 General algorithm

3.2 Special algorithm

3.3 LU decomposition

4 Discussion

Error analysis

The computer should be able to make a maximum of $2.5e9$ floating point operations pr. second. We can inevestegate the number of FLOPS for the general algorithm with $n = 10^7$:

$$FP = 9 \times 10^7 - 17 \approx 9 \times 10^7$$

$$\text{CPU Time} = 0.316\ s$$

$$\text{FLOPS = FP / CPU Time} \approx 0.28\ GHz$$

We see that there are some room up the maximum capacity of the processor (2.5 GHz), but this might be somewhat expectable

5 Conclusion

In this report we have used three different ways of computing $\mathbf{Av} = \mathbf{g}$ and have seen that efficiency of the methods vary greatly. We have witnessed the importance of efficient implementation of algorithms ...

6 References

References

[1] S. B. Gueye, K. Talla and C. Mbow, "Solution of 1d poisson equation with neumann-dirichlet and dirichlet-neumann boundary conditions using the finite difference method", Journal of Electromagnetic Analysis and Applications, Vol. 6, No. 10, pp. 309, 2014.

[2] Hjort-Jensen, M., 2018. Computational Physics Lectures: Linear Algebra methods, accesable at course github repository. http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/pdf/linalg-print.pdf