Project 1 - 1-D Poisson Equation

FYS3150 at University of Oslo

Simen Løken

September 2020

1 Abstract

In this project we're going to be looking at different solutions of the one dimensional Poisson Equation. We're going to be using the Tridiagonal Matrix Algorithm (Thomas Algorithm), both in it's general and a special form uniquely fit to our matrix, and we're going to try brute-forcing a solution using an LU-decomposition, and compare FLOPs and computation time to better understand how we can maximize the efficiency of our programs and the limitations of hardware like memory. We find that we can optimize the general algorithm for and go from 9n FLOPs to 4n.

2 Introduction

The Poisson Equation is a flexible equation with a broad range of uses in physics, from electromagnetism to Newtonian gravity. In it's most general form it's given as:

$$\Delta\varphi = f \tag{1}$$

However, for our purposes, the one dimensional variation can be expressed as:

$$-u'' = f(x) \tag{2}$$

To properly solve this problem we've been given, we're going to be using Python. While solving the problem itself is quite trivial, the main objective of this project is to optimize and refine our method. Thus, we're going to be trying and comparing three solutions, a general Thomas Algorithm, a specialized Thomas Algorithm and an LU-decomposition.

3 Theory and Method

3.1 Theory

Our second order differential equation is given as

$$\frac{\delta^2 u(x)}{\delta x^2} = f(x) \tag{3}$$

with a Dirichlet boundary

$$x \in (0,1), u(0) = u(1) = 0$$

We assume that the function f(x) is given as:

$$f(x) = 100e^{-10x} \tag{4}$$

which gives us the analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{5}$$

Given that we now know f(x) and we're looking to find the second derivative, we can approximate it using a Taylor Expansion. This gives us:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n, \tag{6}$$

where h is the dynamic step-size given as $h = \frac{1}{n-1}$ given by n points and v_i is the discrete form of u. We can rewrite this to fit a matrix. For simplicity and generality, let's rename:

$$-\frac{a_i + c_i - 2b_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n, \tag{7}$$

This can be rewritten as the $n \times n$ matrix

$$\begin{bmatrix} b_1 & c_1 & 0 & \ldots & \ldots & \ldots \\ a_1 & b_2 & c_2 & \ldots & \ldots & \ldots \\ & a_2 & b_3 & c_3 & \ldots & \ldots \\ & \vdots & \vdots & \ddots & \vdots & \vdots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} = f_i h^2$$

We name our matrix A and generalize the equation above to be:

$$\mathbf{Av} = \mathbf{b} \tag{8}$$

where $\mathbf{v}$ are solutions to $\mathbf{b}$ and $\mathbf{b}$ is $h^2 f_i$

3.2 Method

Like previously mentioned, we're going to be employing a general solution given by the Thomas Algorithm. For simplicity's sake, let's assume we've got a 5×5 matrix:

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 \\ 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & a_4 & b_5 \end{bmatrix}$$

This gives us

$$\mathbf{Av} = \mathbf{f}$$

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 \\ 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & a_4 & b_5 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix}$$

Which we can rewrite as

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & f_1 \\ a_1 & b_2 & c_2 & 0 & 0 & f_2 \\ 0 & a_2 & b_3 & c_3 & 0 & f_3 \\ 0 & 0 & a_3 & b_4 & c_4 & f_4 \\ 0 & 0 & 0 & a_4 & b_5 & f_5 \end{bmatrix}$$

with which we can use Guassian elimination, giving us

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 & f_1 \\ a_1 & \tilde{b_2} & c_2 & 0 & 0 & \tilde{f_2} \\ 0 & a_2 & \tilde{b_3} & c_3 & 0 & \tilde{f_3} \\ 0 & 0 & a_3 & \tilde{b_4} & c_4 & \tilde{f_4} \\ 0 & 0 & 0 & a_4 & \tilde{b_5} & f_5 \end{bmatrix}$$

where $\tilde{b_i}$ and $\tilde{f_i}$ are given as:

$$\tilde{b_i} = b_i - \frac{a_{i-1} \cdot c_{i-1}}{\tilde{b}_{i-1}} \tag{9}$$

$$\tilde{f_i} = f_i - \frac{a_{i-1} \cdot \tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{10}$$

This is the forward substitution of our method. We've now got to do backwards substitution to finally find $\mathbf{v}$. For simplicity's sake, we redefine

$$f_1 = \tilde{f_1} \qquad b_1 = \tilde{b}_1$$

We can now eliminate the upper part of the matrix, which gives us

$$u_i = \frac{\tilde{f_i} - c_i \cdot u_{i+1}}{\tilde{b_i}} \tag{11}$$

We can use this algorithm in Python to solve our problem. The algorithm looks like this:

```
def gen(u,f,n,f_t):
    f_t[1] = f[1]
    for i in range(2,n+1):
        bV[i] = bV[i]-(aV[i]*cV[i-1])/bV[i-1]
        f_t[i] = f[i] - (aV[i]*f_t[i-1])/bV[i-1]
    u[n] = f_t[n]/bV[n]
    for j in range(n-1,-1,-1):
        u[j] = (f_t[j]-cV[j]*u[j+1])/bV[j]
    return u
```

We'll now be looking at the special case. While our solution above works for any tridiagonal matrix, we've not taken advantage of the fact that our matrix $\mathbf{A}$ is symmetrical along the diagonal.

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

This means that we can rewrite our previous equations, equation 9 and equation 10. Components now cancel each other out, and we get:

$$\tilde{b}_i = b_i - \frac{1}{\tilde{b}_{i-1}} \tag{12}$$

$$\tilde{f}_i = f_i - \frac{f_{i-1}}{\tilde{b}_{i-1}} \tag{13}$$

It then follows that we can simplify equation 11 to be:

$$v_i = \frac{\tilde{f}_i + v_{i+1}}{\tilde{b}_{i-1}} \tag{14}$$

This simplification means we can simplify our code too, which becomes:

```
def spec(u,f,n,d_t,f_t):
    f_t[1] = f[1]
    for i in range(2,n+1): #forward sub
        f_t[i] = f[i]+(f_t[i-1])/d_t[i-1]
    u[n] = (f_t[n])/d_t[n]
    for j in range(n-1,0,-1): #backwards sub
        u[j]= (f_t[j]+u[j+1])/d_t[j]
    return u
```

We will then be using an equation for relative error given as:

$$\epsilon_i = log_{10}|\frac{v_i - u_i}{u_i}| \tag{15}$$

Lastly we'll be looking at the LU-decomposition. This is the most generalist and brute-force approach for solving a set on linear equations out of the methods we'll be using here. An LU-decomposition exists for all square matrices and is given as

$$\mathbf{A} = \mathbf{LU} \tag{16}$$

where $\mathbf{A}$ is the matrix in question while $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper triangular matrix, respectively. There's also an expanded version of LU-decomposition, partial-pivot, which is expressed as:

$$\mathbf{PA} = \mathbf{LU} \tag{17}$$

Let's assume we wish to solve a linear equations system like equation 8. To do this, we can insert equation 8 into equation 17, which gives us:

$$\mathbf{LUv} = \mathbf{Pb}$$

We can then solve the equations

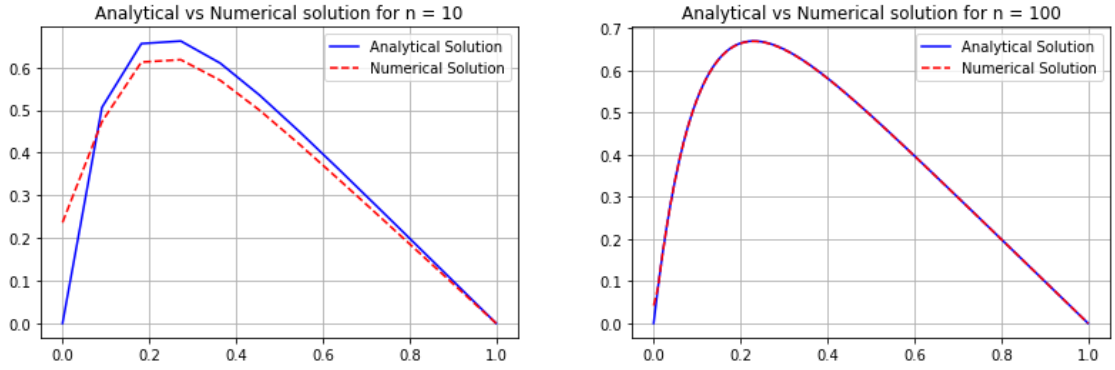
$$I: \quad \mathbf{Lu} = \mathbf{Pb}$$

$$II: \quad \mathbf{Uv} = \mathbf{y}$$

For this we'll be using the scipy.linalg library of functions, namely $lu\_factor()$ and $lu\_solve()$, which will factorize and solve our matrix $\mathbf{A}$, respectively.

4 Results

We'll start by going through the results given the general Thomas Algorithm. We were working with the Dirichlet boundary of $u(0) = u(1) = 0$, and as such we get this solution:



(a) General n = 10

(b) General n = 100

Figure 1: Comparison between the general Thomas Algorithm for n = 10 and 100 respectively

We can see that even for $n = 100$ we don't get a perfect fit. Compare this to the special solution for the Thomas Algorithm, which you can see in Figure 3 below. We see that the special solution is a much better fit already and we're only using $n = 100$. Additionally, if we count the FLOPs of our program manually, we'll find that the general solution uses 9 FLOPs throughout it's loops. Comparatively, the specialized solutions cuts this down to 4 FLOPs, since the algorithm runs $n - 1$ times, we get:

$$FLOPs_{gen} = 9(n-1) = 9n$$

$$FLOPs_{spec} = 4(n-1) = 4n$$

We've gauged the accuracy of our model using equation 15, which gives us (Figure 2):
We see that (maybe unsurprisingly) given a smaller stepsize we get more accurate results. The only exception is a sudden jump for $n = 10^6$ which I can't explain. Maybe a rounding error? Lastly we tried to solve the same problem with an LU decomposition. Above you'll find a table

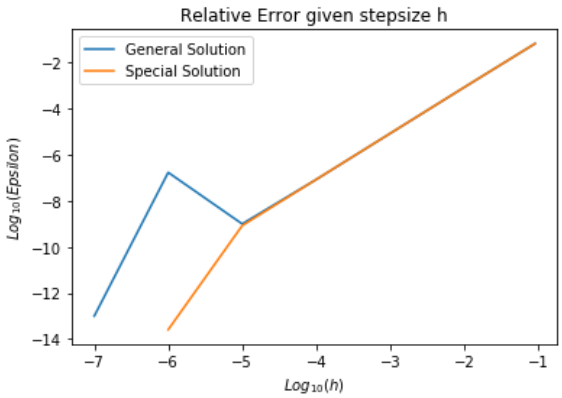
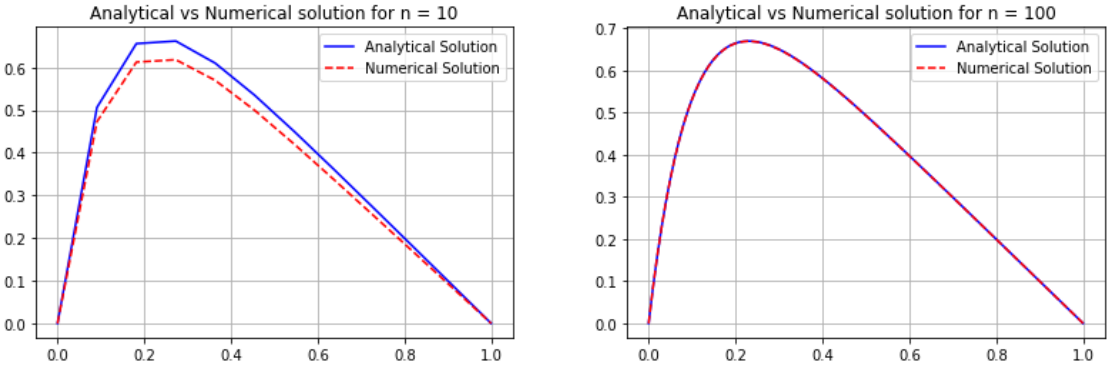


Figure 2: Relative Error given a timestep h



(a) Special n = 10

(b) Special n = 100

Figure 3: Comparison between the special Thomas Algorithm for n = 10 and 100 respectively

General [s]	Special [s]	LU [s]
2.7×10^{-5}	0.0088	0.098
0.0002	0.0088	0.00049
0.002	0.0097	0.016
0.0211	0.019	9.3
0.21	0.24	—
2.0	1.1	—
20	11	—

Table 1: Runtimes for every method in seconds

with the results (table 1) and a complimenting plot below (figure 4) We see that LU-decomposition isn't viable for big matrices, but it works very well and is by far the easiest to implement for small

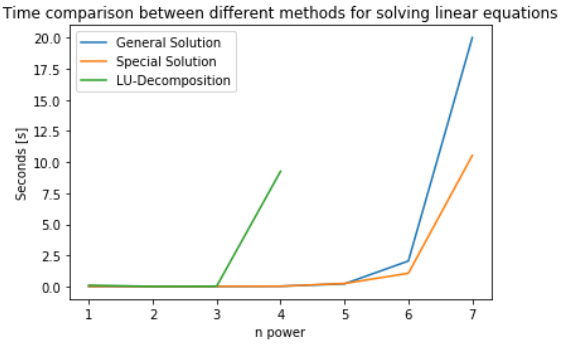


Figure 4: Time taken to run algorithm given a power of n

matrices. There are no measurements past $n = 100000$ because that would require more memory than I have.

5 Discussion

The first thing I'd like to discuss is error. I decided not to add error bars and find the average of benchmark times given that I had incredibly small deviances, with only a few outliers, which we're not supposed to include anyway. Secondly I'd like to discuss the equations 12 and 13. I realized this a bit too close to the deadline to act on it, but I could've saved a FLOP if I had combined $\frac{a_{i-1}}{\tilde{b}_i}$ and made them into some variable C as such:

$$C = \frac{a_{i-1}}{\tilde{b}_i}$$

So in actuality, we're going from 8n FLOPs to 4n when specializing our algorithm. Thirdly I'd like to discuss in a bit more detail the difference between LU-decomposition and the Thomas Algorithm. The primary reason why we're able to get away with much larger numbers should we TDMA is that we're only ever storing three vectors, as opposed to LU which requires an entire matrix. For our limit, $n = 100000$, we'd have a matrix with a size of 100000×100000. Given that floats in Python are double precision floats, every number will individually require 8 bytes, which would be 8×10^{10} bytes, or 80Gb, much more than I or any other commercially available computer has, unless you're a rather eccentric enthusiast.

6 Conclusion

In conclusion, we've shown and calculated the solution of the one dimensional Poisson Equation. We've used both a TMAD and an LU-decomposition, and weighted them against each other. We've also shown that, in some specific cases, you can optimize your TMAD for upto (in our case) a 100% increase in computing speed, from 8n FLOPs to 4n. We've also shown the limitations of LU-decomposition and the importance of effectivizing your algorithm to save memory (and time) when working on larger data sets.

7 Sources

M.Hjorth-Jensen, FYS3150/FYS4150 Lecture Notes 2020, 2020