Project 1

Anna Stray Rongve  
Amund Midtgard Raniseth  
Knut Magnus Aasrud

Mandag 9 September 2019

The programs referenced in this article are in a repository linked in the appendix.

# Abstract

In this project we have solved a one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations **Av=d** and under the following assumptions:

* A is nonsingular
* has a unique solution **x** for every **b** in $\mathbf R^n$

Then we solved the equations utilizing the Thomas algorithm, a special case of Gaussian elimination that has two steps - the forward- and backward substitution. Thereafter we made a special algorithm in order to reduce the number of FLOPs.

Thereafter we specialized the algorithm by choosing a specific matrix $\mathbf A$ in order to reduce the number of FLOPs and compared its CPU time with our general algorithm.

To measure deviation between the analytical() and numerical solution , the relative error was calculated by

By studying the number of floating point operations, FLOPs, we could predict which method would be the most efficient(in measured CPU time). However we realised quickly that computer-factors would play a big roll when increasing the size of the matrix(n).

# Introduction

The purpose of this project is to implement a numerically effective solution of the one-dimensional Poisson equation with Dirichlet boundary conditions

and to implement this in a programming language of choice (Python, in our case). This will be done using two different approaches - the general Thomas algorithm and a specialized Thomas algorithm - the speed of which is compared.

Crucially, the step size affects the results of these methods, and this is also put to the test. The methods are put up against the analytical solution and for one of the algorithms, the relative error is calculated for different step sizes. Lastly, our method is compared to one using LU-decomposition.

# Theory and technicalites

The Poisson equation we are going to solve reads as follows:

with the analytical solution:

## Project 1 a)

We start by discretizing to , with the boundary conditions :

For ,

For ,

For ,

Multiplying both sides by gives

Which you can rewrite as a linear set of equations where

$$\mathbf{A}=\left[\begin{matrix}2 & -1 & 0 & \dots & \dots & 0\\-1 & 2 & -1 & 0 & \ddots & \vdots \\0 & -1 &2 & -1 & 0 & \vdots \\ \vdots & \vdots & \ddots& \ddots & \ddots & \vdots\\0 & \dots & \dots & -1 & 2 & -1\\0 & \dots & \dots & 0 & -1 & 2\end{matrix}\right],$$

$$\mathbf{v}=\left[\begin{matrix}v\_{1}\\v\_{2}\\v\_{3}\\ \vdots \\ v\_{n-1}\end{matrix}\right],$$

and

$$\mathbf{d}=\left[\begin{matrix}d\_{1}\\ d\_{2}\\\ d\_{3}\\ \vdots \\ d\_{n-1}\end{matrix}\right],$$

with .

We see that is a tridiagonal matrix which we can employ the Thomas algorithm [@Hjorth-Jensen2018] on. This is done below.

## Project 1 b)

### General algorithm

We have a linear set of equations

In the general case, we can express any tridiagonal matrix

$$\mathbf{A}=\left[\begin{matrix}b\_1 & c\_1 & 0 & \cdots & \cdots & 0 \\a\_{1} & b\_2 & c\_2 & \ddots & \ddots &\vdots \\0 & a\_{2} &b\_3 & \ddots & \ddots &\vdots \\ \vdots& \ddots & \ddots& \ddots& c\_{n-2} & 0 \\0 & \dots & 0 & a\_{n-2} & b\_{n-1} & c\_{n-1}\\0 & \dots & \dots & 0 & a\_{n-1} & b\_n\end{matrix}\right]$$

just by the three vectors , where has length , and and have length .

### Forward substitution

Firstly, we want to eliminate the 's.

gives us these equations for the case of and

For the rest, we get

We can then modify (3) by subtracting (1), like this

Which gives

Notice that has been eliminated ( the first lower diagonal element has been eliminated).

This can be continued further - to eliminate all the 's - and is what we call *forward substitution*.

Its apparent that the vector elements get more and more complicated. To solve this, we make modified vectors and find their elements recursively. Furthermore, we ensure that the 's are 1 by normalizing with the modified diagonal elements.

### Backward substitution

If we look at the coefficients defined above, we see that they give these equations for every :

This is the *backward substitution* necessary to find the solution.

The whole algorithm runs using FLOPs, specifically . This is a major improvement on Gaussian elimination, which requires FLOPs [@Hjorth-Jensen2018].

## Project 1 c)

### Modified algorithm

In the case of the Poisson equation we can use our general algorithm for a tridiagonal matrix, derived above, and simply replace our variables .

$$\\
\mathbf{A}=\left[\begin{matrix}2 & -1 & 0 & \cdots & \cdots & \cdots\\-1 & 2 & -1 & 0 & &\\0 & -1 & 2 & -1 & 0 &\\\vdots&\vdots & \ddots& \ddots&\ddots &\vdots\\0 & & & -1 & 2 & -1\\0 & & & & -1 & 2\end{matrix}\right] \left[\begin{matrix}v\_1\\v\_2\\ \cdots\\\cdots\\\cdots\\v\_n\end{matrix}\right] = \left[\begin{matrix}d\_1\\d\_2\\ \cdots\\\cdots\\\cdots\\d\_n\end{matrix}\right]
\\$$

This translates into a simpler algorithm, were we're able to cut down the number of FLOPs.

### Forward substitution special case

Inserting the values of , and into the general algorith, we get this:

$$\\
\tilde{b}\_{i}=1\\
\tilde{c}\_{1}=-\frac{1}{2}\\
\tilde c\_i = -\frac{1}{2-(-1)a\_{i-1}} = -\frac{1}{2 + \tilde c\_{i-1}}\\
\tilde d\_1 = \frac{d\_1}{2}\\
\tilde d\_i = \frac{d\_i + \tilde d\_{i-1}}{2+ \tilde c\_{i-1}} \\
$$

### Backward substitution special case

The backward substitution will not be any different from the one in Project 1 b)

$$\\
v\_n = \tilde d\_i\\
v\_{i}=\tilde{d}\_{i}-\tilde{c}\_{i}v\_{i+1}
\\$$

This also runs using O(n) FLOPs, but by simplifying our algorithm the number of FLOPs decreases from **9n** to **6n**.

## Project 1 d)

### Relative error

The special Thomas algorithm is compared against the analytical solution and the relative error is calculated. This is done using this formula:

where is the numerical solution and is the analytical solution. For each step size, the maximum value of the 's is found and stored.

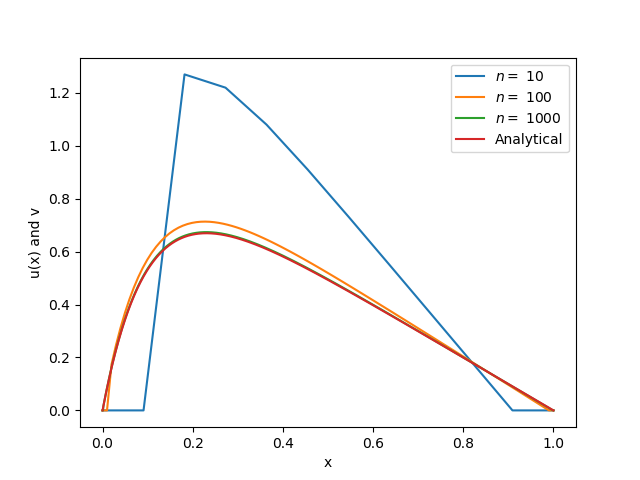
## Project 1 e)

To compare the TDMA function with an LU decomposition we first put both functions in one code to be ran at the same time. For the LU decomposition we decided to use *lu*factor\_ and *lu*decompose\_ from the *scipy.linalg* library. The execution time was counted with *clock()* from the *time* library in Python. The counting started at the start of the recursive algorithm, and were stopped immediately after.

# Results

## Project 1 b)

The program general\_tdma\_function.py is based on the general Thomas algorithm - solving our sample Poisson equation and plotting it at different step sizes. It gives this result:



**Figure 1**: General TDMA solution for , compared to the analytical solution.

Its quite clear that a smaller step size correlates to higher accuracy for these selections of step sizes.

## Project 1 c)

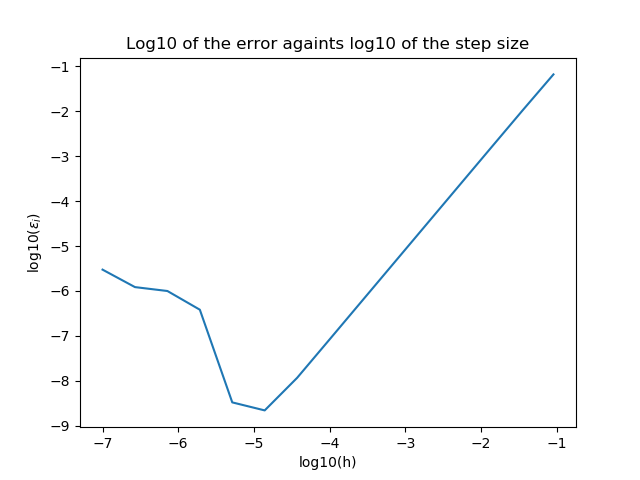
The program special\_tdma\_function.py is based on our specialized Thomas algorithm. In General-vs-special-tdma-test.py, we compare the time spent over the general and special algorithm and print them to the console at different step sizes (-values). The results are:

General TDMA, time spent on n = 100 is 0.000195 seconds  
Special TDMA, time spent on n = 100 is 0.0002599 seconds  
  
  
General TDMA, time spent on n = 1000 is 0.0046473 seconds  
Special TDMA, time spent on n = 1000 is 0.003015 seconds  
  
  
General TDMA, time spent on n = 10000 is 0.0573864 seconds  
Special TDMA, time spent on n = 10000 is 0.0366101 seconds  
  
  
General TDMA, time spent on n = 100000 is 0.307781 seconds  
Special TDMA, time spent on n = 100000 is 0.216879 seconds  
  
General TDMA, time spent on n = 1e+06 is 2.97735 seconds  
Special TDMA, time spent on n = 1e+06 is 2.56285 seconds

Its not fully apparent at small matrix sizes, but once they get big, the reduction in FLOPs makes a difference. This is because the overhead in the *scipy.linalg.lu*solve\_ function is relatively big for small 's.

## Project 1 d)

The program (relative\_error.py) gives these results:



**Figure 2**: log10() vs log10(h)

|  |  |
| --- | --- |
| Relative error |  |
| -1.17969778218 | -1.04139268516 |
| -1.97626186757 | -1.44715803134 |
| -2.8062343362 | -1.86332286012 |
| -3.65484240031 | -2.28780172993 |
| -4.50952402333 | -2.71516735785 |
| -5.36521193786 | -3.14301480025 |
| -6.22236646348 | -3.57159238336 |
| -7.07928513404 | -4.00004342728 |
| -7.93346055078 | -4.42858829767 |
| -8.65847844871 | -4.85715150269 |
| -8.48011516818 | -5.28571704599 |
| -6.41732032512 | -5.71428616043 |
| -6.00098101822 | -6.14285730089 |
| -5.91413531823 | -6.57142872052 |
| -5.52523001828 | -7.00000004343 |

We see that when the of the step size goes below -5, we are losing precision fast.

## Project 1 e)

The code is available in *Project-1/Code/Python/tdma*compare\_lu.py\_ in our github repository.

|  |  |  |
| --- | --- | --- |
|  | TDMA | LU decomposition |
|  |  |  |
|  |  |  |
|  |  |  |

The table is a bit confusing since for the LU decomposition is faster than the TDMA method, but the general trend is that the TDMA method is wildly superior.

If the LU decomposition is run with a matrix, we quickly run out of RAM. This is because every matrix element takes up 8Bytes, which in our case adds up to 80Gigabytes.

# Conclusion and perspectives

# Appendix

[Source Code](https://github.com/kmaasrud/Project-1/tree/master/Code/Python)

# References

::: {refs}