

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

We study the importance of computer algorithm optimization and compare different algorithms used for tridiagonal matrix equation sets. We find that a specialized algorithm for tridiagonal matrices are superior to a general algorithm when it comes to efficiency. We also demonstrate that using a general matrix LU-decomposition is a waste of resources if your matrix is tridiagonal.

Introduction

In this project we want to solve a one-dimensional Poisson equation with Dirichlet boundary conditions which reads as follows:

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

We will solve this numerically in a programming language of choice (in our case Python) in order to study the importance of stepsize and floating point operations, abbreviated FLOPs.

We approached this problem by rewriting the equation as a set of linear equations $\mathbf{A}\mathbf{v}=\mathbf{d}$ under the following assumptions:

- \mathbf{A} is $n \times n$ nonsingular,
- $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution \mathbf{x} for every \mathbf{b} in \mathbf{R}^n .

\mathbf{A} is also tridiagonal, which means we can solve the equations utilizing the Thomas algorithm (Hjorth-Jensen 2018), a special case of Gaussian elimination for tridiagonal matrices. It has two steps - the forward- and backward substitution. Thereafter we specialize our algorithm for the specific matrix \mathbf{A} the Poisson-equation gives us, in order to reduce the number of FLOPs, which we measure by comparing its CPU time with our general algorithm.

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 another, using *scipy*'s LU-decomposition and linear algebra solver.

Theory and technicalites

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

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

with the analytical solution:

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

Project 1 a)

We start by discretizing $u(x)$ to v_i , with the boundary conditions $v_0 = v_n = 0$:

For $i = 1$,

$$-\frac{v_2 + v_0 - 2v_1}{h^2} = f_1.$$

For $i = 2$,

$$-\frac{v_3 + v_1 - 2v_2}{h^2} = f_2.$$

For $i = n - 1$,

$$-\frac{v_n + v_{n-2} - 2v_{n-1}}{h^2} = f_{n-1}.$$

Multiplying both sides by h^2 gives

$$-v_2 + 2v_1 - v_0 = h^2 \cdot f_1,$$

$$-v_3 + 2v_2 - v_1 = h^2 \cdot f_2,$$

$$-v_n + 2v_{n-1} - v_{n-2} = h^2 \cdot f_{n-1}.$$

Which you can rewrite as a linear set of equations $\mathbf{A}\mathbf{v} = \mathbf{d}$ where

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

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \end{bmatrix},$$

and

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \end{bmatrix},$$

with $d_i = h^2 \cdot f_i$.

We see that \mathbf{A} is a tridiagonal matrix which we can employ the Thomas algorithm (Thomas 1949) on. This is done below.

Project 1 b)

General algorithm

We have a linear set of equations $\mathbf{A}\mathbf{v} = \mathbf{d}$

In the general case, we can express any tridiagonal matrix

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

just by the three vectors a, b and c , where b has length n , and a and c have length $n - 1$.

Forward substitution

Firstly, we want to eliminate the a_i 's.

$\mathbf{A}\mathbf{v} = \mathbf{d}$ gives us these equations for the case of $i = 1$ and $i = n$

$$b_1 v_1 + c_1 v_2 = d_1, \quad i = 1 \quad (1)$$

$$a_{n-1} v_{n-1} + b_n v_n = d_n, \quad i = n. \quad (2)$$

For the rest, we get

$$a_1 v_1 + b_2 v_2 + c_2 v_3 = d_2, \quad i = 2. \quad (3)$$

$$a_{i-1} v_{i-1} + b_i v_i + c_i v_{i+1} = d_i, \quad i = 2, \dots, n-1.$$

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

$$b_1 \cdot (3) - a_1 \cdot (1)$$

Which gives

$$(a_1 v_1 + b_2 v_2 + c_2 v_3) b_1 - (b_1 v_1 + c_1 v_2) a_1 = d_2 b_1 - d_1 a_1$$

$$(b_2 b_1 - c_1 a_1) v_2 + c_2 b_1 v_3 = d_2 b_1 - d_1 a_1.$$

Notice that v_1 has been eliminated (\Leftrightarrow the first lower diagonal element has been eliminated).

This can be continued further - to eliminate all the a_i '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 \tilde{b}_i 's are 1 by normalizing with the modified diagonal elements.

$$\begin{aligned} \tilde{b}_i &= 1 \\ \tilde{c}_1 &= \frac{c_1}{b_1} \\ \tilde{c}_i &= \frac{c_i}{b_i - \tilde{c}_{i-1} a_{i-1}} \\ \tilde{d}_1 &= \frac{d_1}{b_1} \\ \tilde{d}_i &= \frac{d_i - \tilde{d}_{i-1} a_{i-1}}{b_i - \tilde{c}_{i-1} a_{i-1}} \end{aligned}$$

Backward substitution

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

$$v_n = \tilde{d}_n$$

$$v_i = \tilde{d}_i - \tilde{c}_i v_{i+1}$$

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

The core of our Thomas algorithm code reads as follows:

```
for i in range(1,n-1):  #FORWARD substitution
    m=1.0/(b[i]-a[i-1]*c_tilde[i-1])  #saves 1 division, 1 multiplication and 1 subtraction
    c_tilde[i]=c[i]*m
    d_tilde[i]=(d[i]-a[i-1]*d_tilde[i-1])*m

v[n-1]=d_tilde[n-1]

for i in reversed(range(1,n-1)):  #BACKWARD substitution
    v[i]=d_tilde[i]-c_tilde[i]*v[i+1]
```

Which implements the recursive formulas found immediately above. The code can be found in `/Code/Python/general_tdma_function.py` in our repository linked here.

The whole algorithm runs using $O(n)$ FLOPs, specifically $9n$. This is a major improvement on Gaussian elimination, which requires $O(n^3)$ FLOPs (Hjorth-Jensen 2018).

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 a_i , b_i and c_i with respectively -1 , 2 and -1 .

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots \\ -1 & 2 & -1 & 0 & & \\ 0 & -1 & 2 & -1 & 0 & \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & & -1 & 2 & -1 \\ 0 & & & & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ \cdots \\ \cdots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdots \\ \cdots \\ \cdots \\ d_n \end{bmatrix}$$

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

Forward substitution special case

Inserting the values of a_i , b_i and c_i into the general algorithm, 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:

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

where v is the numerical solution and u is the analytical solution. For each step size, the maximum value of the ϵ_i '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:

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

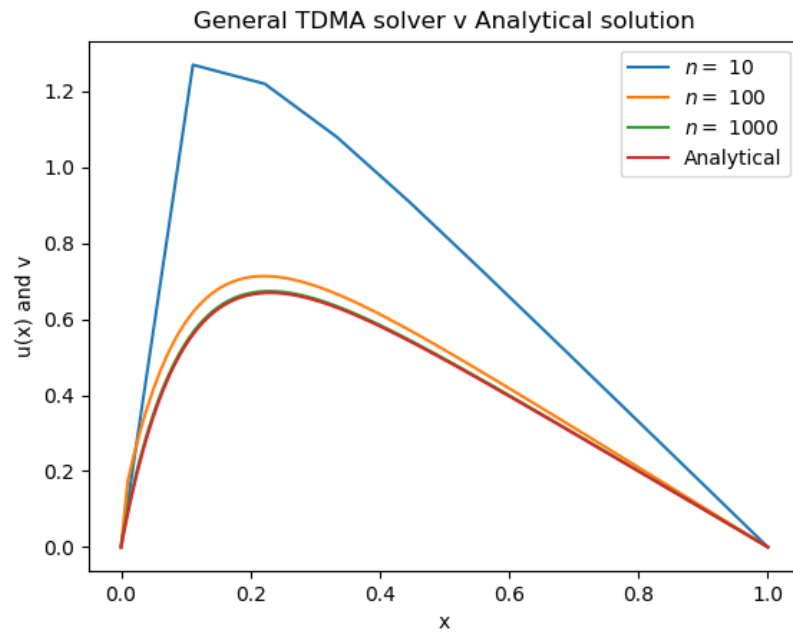


Figure 1: **Figure 1:** General TDMA solution for $n = 10, 100, 1000$, compared to the analytical solution.

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 (n -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. The difference in time fluctuates a bit, but this is probably due to background tasks taking up some of the memory.

Project 1 d)

The program (`relative_error.py`) gives these results:

Relative error	$\log_{10}(\text{Step size})$
-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

Relative error	$\log_{10}(\text{Step size})$
-5.91413531823	-6.57142872052
-5.52523001828	-7.00000004343

We see that when the \log_{10} 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
$n = 10$	0.000045s	0.000151s
$n = 100$	0.002442s	0.002045s
$n = 1000$	0.026847s	0.131573s

The table is a bit confusing since for $n = 100$ the LU decomposition is faster than the TDMA method, but the general trend is that the TDMA method is wildly superior. The cause of this contradiction is most probably caused by some sort of overhead in Python or the *scipy* libraries.

If the LU decomposition is run with a $10^5 \times 10^5$ matrix, we quickly run out of RAM. This is because every matrix element takes up 8Bytes(double precision variable), which in our case adds up to 80Gigabytes.

Conclusion and perspectives

Running the different programs we now understand the importance of algorithm optimization. By reducing the number of FLOPs from $9n$ to $6n$, our program would run 33% faster. This was not entirely apparent through the timing results though, since the background tasks interfered. When running *scipy*'s LU algorithm with a big matrix we realised the importance of reducing memory usage and unnecessary calculations as much as possible in order to reduce the running time and even to be able to run the code. With bigger calculations this would be of an even greater importance.

An important aspect of this project was to study the accuracy of our calculation by finding the limit for our stepsize in terms of computer precision. We see that when decreasing the stepsize the precision increases until the point where it starts to decrease. This might be due to round off errors.

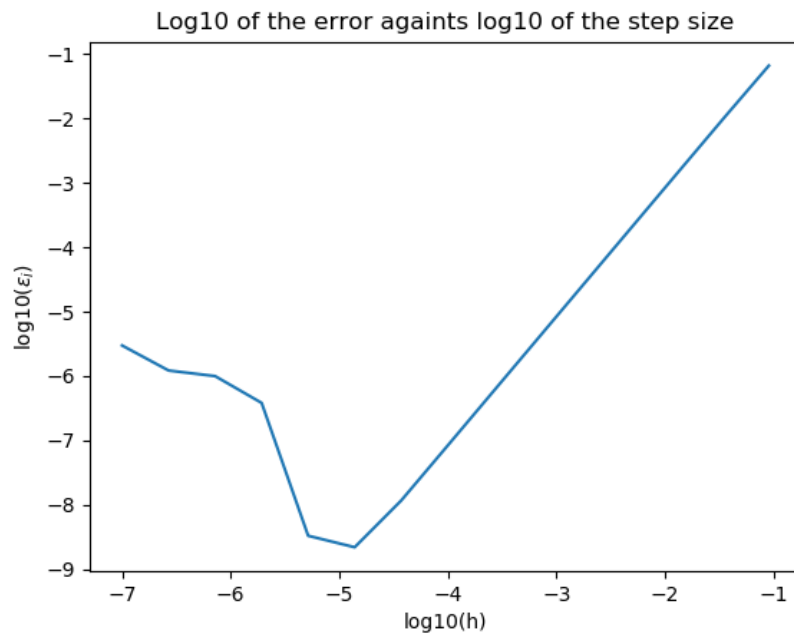


Figure 2: **Figure 2:** $\log_{10}(\epsilon)$ vs $\log_{10}(h)$

Appendix

Source Code

References

::: {refs}

Hjorth-Jensen, Morten. 2018. “Computational Physics Lectures: Linear Algebra methods Important Matrix and vector handling packages Basic Matrix Features Matrix properties reminder.” <http://www.netlib.org>.

Thomas, L. H. 1949. “Elliptic Problems in Linear Differential Equations over a Network, Watson Sci. Comput. Lab Report, Columbia University, New York.” <http://www.cs.tau.ac.il/~dcor/Graphics/adv-slides/Solving.pdf>.