

Project 1

Anna Stray Rongve Amund Midtgard Raniseth
Knut Magnus Aasrud

Mandag 9 September 2019

Abstract

Summary of project.

The abstract gives the reader a quick overview of what has been done and the most important results. Try to be to the point and state your main findings.

In this project we have solved a one-dimensional Poisson equation with Dirichlet boundary condition by rewriting it as a set of linear equations, $\mathbf{A}\mathbf{v}=\mathbf{d}$. 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 floating point operations and compared its CPU time with our general algorithm.

In the last part we computed the relative error for the exact function vs. the computed and how the error developed with increasing floating points. Lastly we compared our results from our previous calculations with a LU- decomposition.

Introduction

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

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

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 equation 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 (cite lecture notes) 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 $i = n$

$$b_1v_1 + c_1v_2 = d_1, \quad i = 1 \quad (1)$$

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

For the rest, we get

$$a_1v_1 + b_2v_2 + c_2v_3 = d_2, \quad i = 2. \quad (3)$$

$$a_{i-1}v_{i-1} + b_iv_i + c_iv_{i+1} = d_i, \quad i = 2, \dots, n-1.$$

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

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

Which gives

$$\begin{aligned} (a_1v_1 + b_2v_2 + c_2v_3)b_1 - (b_1v_1 + c_1v_2)a_1 &= d_2b_1 - d_1a_1 \\ (b_2b_1 - c_1a_1)v_2 + c_2b_1v_3 &= d_2b_1 - d_1a_1. \end{aligned}$$

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 :

$$\begin{aligned} v_n &= \tilde{d}_n \\ v_i &= \tilde{d}_i - \tilde{c}_i v_{i+1} \end{aligned}$$

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

The whole algorithm runs using $O(n)$ FLOPs, specifically $9n$. This is a major improvement on Gaussian elimination, which requires $O(n^3)$ FLOPs (cite lecture notes).

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 .

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

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**.

Relative error

Conclusion and perspectives

Results

Project 1 b)

Project 1 c)

Project 1 d)

The program (`general-vs-special-tdma-test.py`) gives this result:

Relative error	$\log_{10}(\mathbf{Step\ size})$
-0.545362450619	-2.00432137378
-0.483273092184	-3.00043407748
-0.477730509812	-4.00004342728
-0.477182121231	-5.00000434292
-0.477128070817	-6.00000043429
-0.477119426687	-7.00000004343

Project 1 e)

Appendix

[Source Code](#)

Bibliography