Project 1

Linus Ekstrøm, Joakim Flatby

August 2018

Two numerical methods for solving the one-dimensional Poisson equation were studied in this article: LU-decomposition and the Thomas algorithm. Comparing our solution to an exact analytic solution let us analyze both numerical errors and computational requirements for both our methods. The Thomas algorithm proved to be more efficient than LU-decomposition.

Contents

Introduction

In this article we studied a case of the one-dimensional Poisson equation. We discretize and approximate the second derivative and solve the Poisson equation in the form of a tridiagonal matrix equation. Two methods for solving our matrix equation were used: Thomas algorithm and LU-decomposition.

Theory

The One-dimensional Poisson Equation

The Poisson equation is a classic electromagnetic equation for the potential given a charge distribution. We wish to solve the one-dimensional Poisson equation for a source term

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

because this source has an exact closed form solution

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

which will let us test the accuracy of our numerical calculations. Rewritten the general one-dimensional Poisson equation reads

$$-\nabla^2 u(x) = f(x) \tag{1}$$

We will couple this equation with the Dirichlet boundary conditions

$$x \in (0,1) \text{ and } u(0) = u(1) = 0$$

so that our system of equations are solvable. Our first step is to discretize u as v_i with grid points $x_i = ih$ in the interval $x_0 = 0$ to $x_{n+1} = 1$, with $h = 1/(n+1)$. By Taylor expanding $u(x)$ and solving for $\frac{d^2u}{dx^2}$ we can then approximate the second derivative of $u(x)$. The Dirichlet boundary conditions are then $v_0 = v_{i+1} = 0$ By inserting this approximation into the Poisson equation we get

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \tag{2}$$

Multiplying with the h^2 term and defining $b_i = h^2 f_i$ we are left with

$$-v_{i+1} - v_{i-1} + 2v_i = b_i \tag{3}$$

where $i = 1, ..., n$, and we have now arrived at our set of linear equations. This can be written as

$$A\vec{v} = \vec{b} \tag{4}$$

where A is the $n \times n$ tridiagonal matrix

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

Looking at this we notice the pattern present in the matrix A as supplied in the task.

Solving a General Tridiagonal Matrix Equation

It is possible to rewrite our matrix A in terms of one-dimensional vectors $\vec{a}, \vec{b}, \vec{c}$ of length $1:n$. The equation then reads

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

To solve this general matrix equation we row reduce matrix $\mathbf{A}$ to echelon form. First operation is subtracting $\frac{a_1}{b_1}$Row I $-$ Row II, which yields

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ 0 & b_2' & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2' \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

where $b_2' = b_2 - \frac{a_1}{b_1}c_1$ and $\tilde{b}_2' = \tilde{b}_2 - \frac{a_1}{b_1}\tilde{b}_1$. Next we wish to remove a_2 in the same fashion, only this time we use our new values of b_2' and $\tilde{b}_2'$. This time we subtract $\frac{a_2}{b_2'}$Row I $-$ Row II and we continue this process all the way down. Generalizing our procedure we obtain the following relations for b_i' and $\tilde{b}_i'$

$$b_i' = b_i - \frac{a_{i-1}}{b_{i-1}'}c_{i-1} \qquad \tilde{b}_i' = \tilde{b}_i - \frac{a_{i-1}}{b_{i-1}'}\tilde{b}_{i-1}' \tag{5}$$

where we use $b_1' = b_1$ and $i \in (2, 3, .., n)$. Running over these i's gives us our tridiagonal matrix on row reduced echelon form

$$\mathbf{A} = \begin{bmatrix} b_1' & c_1 & 0 & \dots & \dots & \dots \\ 0 & b_2' & c_2 & \dots & \dots & \dots \\ & 0 & b_3' & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & b_{n-1}' & c_{n-1} \\ & & & & 0 & b_n' \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1' \\ \tilde{b}_2' \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n' \end{bmatrix}.$$

Next we notice that v_n is given by the bottom row: $v_n = \frac{\tilde{b}_n'}{b_n'}$, and from this we can back substitute to find the rest of the solution to our equation. The pattern for the back substitution is given by

$$v_i = \frac{\tilde{b}_i' - c_i v_{i+1}}{b_i'} \tag{6}$$

where we let i run backwards from $n-1$ to 1. The list of values we obtain from doing these steps is the solution to our matrix equation. This algorithm, backward and forward to solve a tridiagonal matrix equation, is called the general Thomas algorithm.

Specializing the Thomas Algorithm

Taking a look at our specific matrix A. we can see that in our case we always have $a_i = c_i = -1$, and $b_i = 2$. With these values in hand we can easily simplify our algorithm because we don't need to loop through values of a_i and c_i and multiply during runtime, we can just hardcode the multiplication by -1 beforehand. We can also find an analytic expression for b'_i. Again, since we have $a_i = c_i$ and $b_i = 2$, it simplifies to

$$b'_i = 2 - \frac{1}{b'_i - 1} \tag{7}$$

and looking at how this behaves as i increases we can see that it follows the same pattern as

$$b'_i = \frac{i+1}{i} \tag{8}$$

when $i \geq 1$. This means that we can precalculate b'_i as well since it now only depends on i.

Computing the Relative Error in Our Approximation

In this article we have examined the following source term for the one-dimensional Poisson equation¨

$$f(x) = 100e^{-10x} \tag{9}$$

which yields the closed-form solution

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

Computing the relative error as

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{11}$$

for $i \in (1, 2, ..., n)$

Comparing results with LU-decomposition

Using scipy's built-in LU-method we can solve the same equation by just passing in the matrix A and $\tilde{b}$. However, it might not be the most efficient way of doing it. Using the algorithm from the last section to calculate the relative error we can compare the results between the two methods. With Pythons time-function we can also compare the computation time for each method, to see which one takes the most out of the computer.

In addition to this, we can take a look at the amount of floating point-operations being executed in each of the different methods. Seeing that these are what mostly makes up the work the computer has to do when running through this program, we should see that the difference amount of floating point-operations between the three methods correspond with the computation difference in time recorded by python.

General Thomas Algorithm

The general Thomas algorithm consists of two equations for the forward substitution and one equation for the backward substitution. All three equations have 3 operations. So the total amount of floating point operations for the forward substitution is $3(n-1) + 3(n-1) = 6(n-1)$ and the backwards substitution $3(n-1)$ which adds up to $9(n-1)$ for the entire algorithm.

Specialized Thomas Algorithm

In the forward substitution there is now only one equation, containing three operations. The backwards substitution only has two operations now that c_i is known beforehand. This means we have $3(n-1) + 2(n-1) = 5(n-1)$ total operations.

LU-decomposition

The LU-decomposition has to go through a lot more operations than the Thomas algorithm because it doesn't have a method for singling out the parts of the matrix containing values. Instead it factors A (including all the zeros) into PLU and solves the entire system. This adds up to approximately $\frac{2}{3}n^3$ operations.

Method

We wrote the algorithms in Python using Python3 with NumPy, MatPlotLib and SciPy.

General Algorithm

This algorithm needs the values of a, b and c given as arrays where a and c is of length $n-1$ and b is of length n. This is because there is no a in the first row and no b in the last row of the matrix. We also need $\tilde{b}$ which is calculated from $h^2 f$, both of which only depend on n. Lastly the initial values are set and the loop themselves go as follows

```
b_prime[0] = b[0]
b_prime_tilde[0] = b_tilde[0]

# Forward substitution
for i in range(1, n - 1):
    b_prime[i] = b[i] - (a[i] / b_prime[i - 1]) * c[i - 1]
    b_prime_tilde[i] = b_tilde[i] - (a[i] / b_prime[i - 1]) * b_prime_tilde[i - 1]

# Backward substitution

for i in range(n-2, 0, -1):
    v[i] = (b_prime_tilde[i] - c[i] * v[i + 1]) / b_prime[i]
return v
```

Speciaized Algorithm

The specialized algorithm has the values for a, b and c hardcoded into the equations. This means that they are replaced with their corresponding values -1, 2 and -1 since we know they have the same value for all i. We see that b' can now be computed once beforehand instead of $(n-1)$ times and that several variables turn into numbers in the two remaining equations. All of which is what leads to fewer floating point operations.

```
b_prime = [(i+1)/i for i in range(1, n)]
b_prime_tilde[0] = b_tilde[0]

# Forward substitution
for i in range(1, n - 1):
    b_prime_tilde[i] = b_tilde[i] - (-1 / b_prime[i - 1]) * b_prime_tilde[i - 1]
```

```
7
8  # Backward substitution
9
10 for i in range(n-2, 0, -1):
11     v[i] = (b_prime_tilde[i] + v[i + 1]) / b_prime[i]
12 return v
```

LU-Decomposition

To compute using the LU-decomposition method we use python's built-in lu factor- and lu solve-functions. LU solve takes two arguments, A and $\tilde{b}$. We create A using for loops and $\tilde{b}$ using the get_b_tilde()-function.

```
1  def get_b_tilde(n):
2      h = 1 / (n+1)
3      x = np.array([i * h for i in range(n)])
4      f = 100*np.exp(-10*x)
5      return h*h * f
6
7  def LU_decomposition(n):
8      matrix = np.array([[0 for i in range(n)] for j in range(n)])
9
10     for i in range(n):
11         matrix[i][i] = 2
12         if(i!=0):
13             matrix[i][i-1] = -1
14         if (i!=n-1):
15             matrix[i][i+1] = -1
16
17     return scipy.linalg.lu_solve(scipy.linalg.lu_factor(matrix), get_b_tilde(n))
```

Relative Error

Relative error is calculated using $\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right)$ and then using numpy.max() on the resulting array ϵ

```
1  def get_b_tilde(n):
2      h = 1 / (n+1)
3      x = np.array([i * h for i in range(n)])
4      f = 100*np.exp(-10*x)
5      return h*h * f
6
7  def analytical(n):
8      h = 1 / (n + 1)
9      x = np.array([i * h for i in range(n)])
10     return u(x)
11
12 def relative_error(v, n):
13     return np.max(np.log10(np.abs((v[1:] - analytical(n)[1:])/analytical(n)[1:])))
```

Computation Time

The time used by each algorithm is found by using a simple implementation of pythons time.time().

```
1  t0 = time.time()
2  test = general_algorithm(n)
3  comp_time = time.time() - t0
```

Results and Discussion

Thomas algorithm

Results from the Thomas algorithm with a few different values for n, plotted with the analytical solution(with n=100)

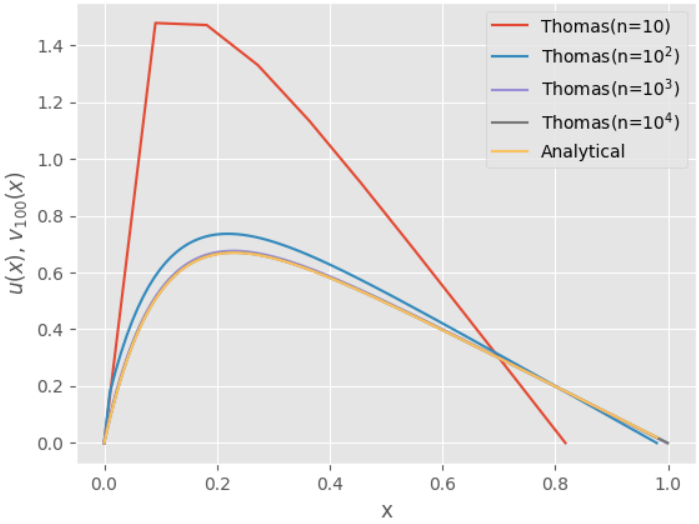

Figure 1: Thomas algorithm results vs. analytical

LU-Decomposition

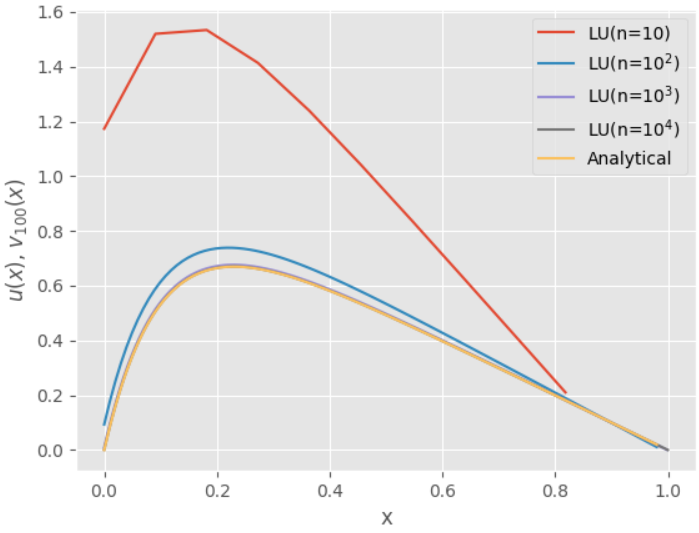

Figure 2: LU-decomposition vs analytical

Error Analysis

Below is a table for the relative error for $n \in (10^1, 10^7)$

n	General Thomas alg.	Special Thomas alg.	LU-decomposition
10^1	0.28332	0.28332	0.30174
10^2	0.03646	0.03645	0.03749
10^3	0.00375	0.00375	0.00385
10^4	0.00038	0.00038	*NA*
10^5	3.76382e-05	3.76384e-05	*NA*
10^6	3.91100e-06	3.76395e-06	*NA*
10^7	1.32374e-06	3.76464e-07	*NA*

Notice the error of the two Thomas algorithms is lower than for the LU-decomposition. Moreover, the special Thomas algorithm is a bit more accurate than the general one, but not by a substantial amount.

Computation Time

Below is a table for the computation time for $n \in (10^1, 10^7)$. While the difference in accuracy for the special and general Thomas algorithm was insubstantial the difference in computation time was fairly large.

n	General Thomas alg.	Special Thomas alg.	LU-decomposition
10^1	0.0003519	8.89e-05	0.0001841
10^2	0.0027337	0.0004251	0.0020370
10^3	0.0260372	0.0029838	0.1249039
10^4	0.2580581	0.0323858	*NA*
10^5	2.5139050	0.3043160	*NA*
10^6	26.623171	3.5876150	*NA*
10^7	253.45664	33.660682	*NA*