FYS3150 Project 1

Bendik Nyheim

September 7, 2020

Abstract

The runtime of the regular and specialized Thomas algorithm for $n = 10^6$ are respectively $0.069s$ and $0.048s$, and for $n = 10^7$ they are respectively $0.672s$ and $0.485s$. Solving the system using LU-decomposition took $1.972s$ for $n = 10^3$, and would not finish running for $n = 10^4$ and above. The max relative error decreases by a factor of 100 when the step size is decreased by a factor of 10.

1 Introduction

This project aims to investigate the accuracy and run time of numerical methods of solving mathematical problems. Specifically, the one-dimensional Poisson equation with Dirichlet boundary conditions is being solved by rewriting it as a set of linear equations. The equation reads

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0. \tag{1}$$

By defining the discretized approximation of u as v_i, I can rewrite the second derivative of u as

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

where h is the step length between the data points, and $f_i = f(x_i)$. The step length $h = 1/(n+1)$ and $i = 1, 2, 3, ..., n$. I will assume that $f(x) = 100e^{-10x}$. After defining the variables like this, I can easily write the differential equation as a set of linear equations, which I will show in the method section.

Throughout this project, there is one important aspect to keep in mind, which is dynamic memory allocation. Dynamic memory allocation means that I manually allocate space in memory for data

structures. This is mostly important to do for long arrays. When the arrays are no longer needed in the program, I manually delete the space in memory. If I don't do this, the computer might run out of memory when dealing with many long arrays.

2 Method

As mentioned in the introduction, I will solve (2) by rewriting it as a set of linear equations. If we input $i = 1, 2, ..., n$, we get

$$\boldsymbol{f} = \frac{1}{h^2}\begin{bmatrix} 2v_1 - v_2 \\ 2v_2 - v_1 - v_3 \\ 2v_3 - v_2 - v_4 \\ \dots \\ \dots \\ 2v_n - v_{n-1} \end{bmatrix} = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \frac{1}{h^2}A\mathbf{v} \tag{3}$$

To make it a little easier to program I also define $d_i = h^2 f_i$, and we end up with $A\mathbf{v} = \boldsymbol{d}$. In order to solve this system, I am implementing the tridiagonal matrix algorithm, also known as the Thomas algorithm. A tridiagonal matrix, is a matrix with only elements in the three middle diagonals. The general system looks like this

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_1 & b_2 & c_2 & & \\ & a_2 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_{n-1} & b_n \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}. \tag{4}$$

The algorithm consists of forward substitution, and then backwards substitution. First we remove the lower diagonal $(a_1, ..., a_n)$ by subtracting a multiple of the row above, and rewriting the elements

on the diagonals like this

$$\tilde{b}_2 = b_2 - a_1/b_1$$
$$\tilde{b}_3 = b_3 - a_2/b_2$$
$$\vdots$$
$$\tilde{b}_i = b_i - a_{i-1}/b_{i-1}$$

This will also change the right hand side, such that

$$\tilde{d}_2 = d_2 - a_1/b_1$$
$$\tilde{d}_3 = b_3 - a_2/b_2$$
$$\vdots$$
$$\tilde{d}_i = b_i - a_{i-1}/b_{i-1}$$

The first elements stay the say, such that $\tilde{b}_1 = b_1$ and $\tilde{d}_1 = d_1$. The system now looks like this

$$\begin{bmatrix} \tilde{b}_1 & c_1 & & & 0 \\ 0 & \tilde{b}_2 & c_2 & & \\ & 0 & \tilde{b}_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & 0 & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{d}_1 \\ \tilde{d}_2 \\ \tilde{d}_3 \\ \vdots \\ \tilde{d}_n \end{bmatrix}. \tag{5}$$

The backwards substitution then solves for all the unknown by first finding $v_n = \tilde{d}_n/\tilde{b}_n$, and then using

$$v_i = \frac{\tilde{d}_i - c_i v_{i+1}}{\tilde{b}_i}, \;\; i = n-1,\; n-2,\; \ldots,\; 1 \tag{6}$$

Now I'll show you how this is implemented in C++ code. Assume that the arrays a, b, c and d are defined.

```
b_tilde[0]=b[0]
d_tilde[0]=d[0]


//Forward subsitution
for (int i = 1; i < n; i++)
```

```
    {
        b_tilde[i] = b[i]-a[i-1]*c[i-1]/b_tilde[i-1];
        d_tilde[i] = d[i]-a[i-1]*d_tilde[i-1]/b_tilde[i-1];
    }

//Backwards substituition
v[n]=d_tilde[n]/b_tilde[n]
for (int i = n-1; i > 0; i--)
    v[i]=(d_tilde[i]-c[i]*v[i+1])/b_tilde[i];
```

I chose to solve this for $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$, to see how important the number of grid points are for approximating the analytical solution. In order to plot the analytical solution, I first need to know what it is. As mentioned in the introduction, I assume that $-u''(x) = f(x) = 100e^{-10x}$. I integrate it twice, and get

$$u'(x) = 10e^{-10x} + A$$
$$u(x) = -e^{-10x} + Ax + B$$

I use the boundary conditions, that are $u(0) = u(1) = 0$,

$$u(0) = -1 + B = 0 \implies B = 1$$
$$u(1) = -e^{-10} + A + 1 = 0 \implies A = e^{-10} - 1$$
$$\implies u(x) = 1 + (e^{-10} - 1)x - e^{-10x}$$

I wrote the numerical solution for all the different values of n to a file, and plotted them with the analytical solution. In addition to visualizing the difference with the numerical solutions and the analytical, I also calculated the maximum relative error between the numerical and analytical solution, for every value of n. For that I used the formula

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{7}$$

The reason for using a log_{10} is that the error from this numerical method is given by $O(h^2)$. When we decrease the step size by a factor of 10, the error is decreased by a factor of 100. Because of that, it is easier to present the error after taking the log_{10} of it.

An important thing to have in mind when solving math problems numerically, is the number of FLOPS(floating point operations) that your solving algorithm uses. The total number of FLOPS in the forward and backwards substitution used in this project, is $9n$. I find this by counting the number of operations (+ - * /) in the algorithm. In our specialized case, where the diagonal values are equal, we can do better. We got $a_i = c_i = -1$ for $i = 0, 1, \ldots, n$, which means calculating $a_{i-1} \cdot c_{i-1} = 1$ is unnecessary. Calculating $a_{i-1} \cdot \tilde{d}_{i-1} = -\tilde{d}_{i-1}$ is also unnecessary. If we look at $\tilde{b}$, we can actually find an analytical solution after removing $a \cdot c$. Everything that has an analytical solution can be calculated before the solving algorithm, and is therefore not counted in number of FLOPS. The expression for after removing the unnecessary operation is $\tilde{b}_i = 2 - 1/\tilde{b}_{i-1}$. In our case, $b_i = 2$ is constant. If we start at the beginning, we got

$$\tilde{b}_1 = b_1 = 2$$
$$\tilde{b}_2 = b_2 - 1/\tilde{b}_1 = 2 - 1/2 = 3/2$$
$$\tilde{b}_3 = b_3 - 1/\tilde{b}_2 = 2 - 2/3 = 4/3$$
$$\tilde{b}_4 = b_4 - 1/\tilde{b}_3 = 2 - 3/4 = 5/4$$

As we can see, there is a pattern, and the analytical solution for $b_i = (i+1)/i$. The C++ code for our special case, looks like this

```
//Forward subsitution
for (int i = 1; i < n; i++)
    d_tilde[i] = d[i]+d_tilde[i-1]/b_tilde[i-1];


//Backwards substituition
v[n]=d_tilde[n]/b_tilde[n]
for (int i = n-1; i > 0; i--)
    v[i]=(d_tilde[i]+v[i+1])/b_tilde[i];
```

The number of FLOPS are now reduced to $4n$. In order to see what difference this actually makes, I timed the regular and specialized algorithm 100 times for values of n up to $n = 10^7$, and took the average of all the run times. When allocating memory dynamically for n this big, and timing it 100

times, it is very important to delete the arrays from memory. If not, the memory will likely run out. This is an example of how to allocate memory and deleting it.

```
double *x = new double[n];
delete [] x;
```

Though I have investigated how the run time varies for this algorithm, what about other methods for solving the differential equation? Another method is by LU-decomposition, which is by writing a matrix and a product of an upper and lower triangular matrix. It is a common method, but the number of FLOPS for this method is $O(n^3)$. I will compare the computation of this method to the previously used Thomas algorithm.

3 Results

Figure 1 shows the numerical solutions for a few different number of grid points, along with the numerical solution. We can clearly see that for $n = 10$, the solution is far from the analytical. As we increase n by a factor of 10, the error decreases rapidly, as expected. Figure 2 shows a close up, which makes the it easier to see the difference of the numerical solutions. Table 1 presents the results of the run times for the different algorithms.

Table 1: Table showing the run time for the solving algorithms using both the regular and specialized Thomas algorithm, and LU-decomposition.

n	Regular	Specialized	LU-decomp
10	0	0	0
10^2	0	0	0
10^3	0	0	1.972s
10^4	0	0	-
10^5	0	0	-
10^6	0.069s	0.048s	-
10^7	0.672s	0.485s	-

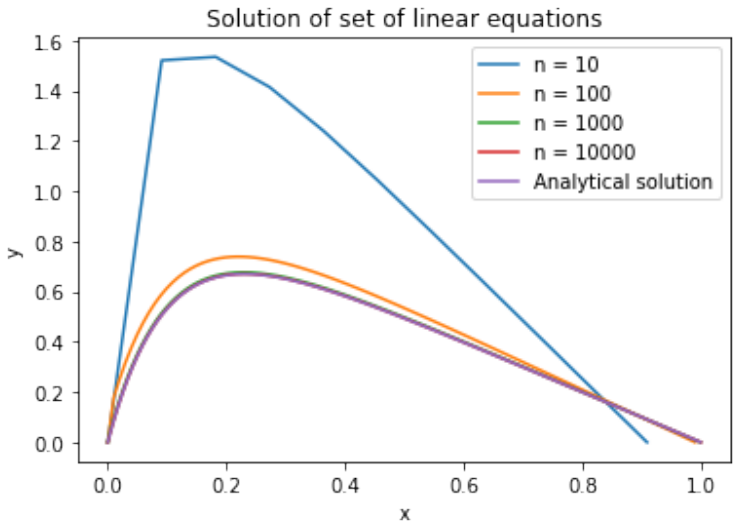

Figure 1: The numerical solutions of the system of equations solved for different numbers of grid points n. The analytical solution is also plotted.

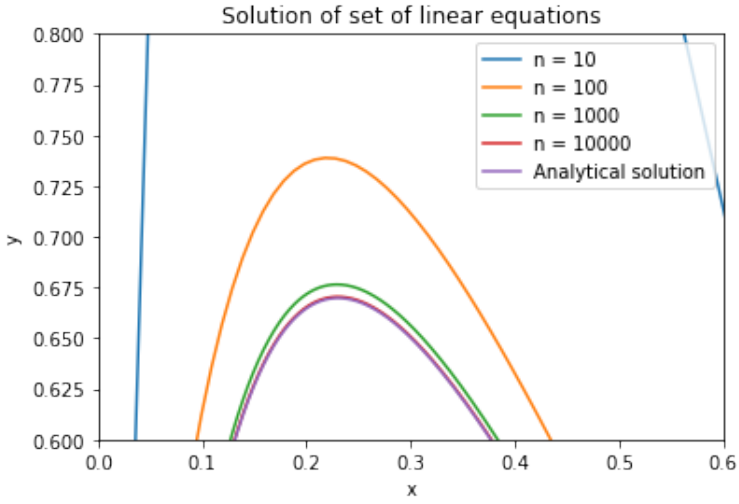

Figure 2: A close-up of figure 1, which makes it easier to see the difference of the different solutions.

4 Discussion

The results in table 1 are not good. The reason for this is the precision of the timer used in C++. For both the regular and specialized Thomas algorithm, it only returned 0 as runtime, which is

Table 2: log_{10} of the max relative error between the numerical and analytical solutions.

n	$log_{10}(\epsilon_{rel})$
10	$1.782 \cdot 10^{-1}$
10^2	$1.239 \cdot 10^{-2}$
10^3	$1.209 \cdot 10^{-3}$
10^4	$1.207 \cdot 10^{-4}$
10^5	$1.206 \cdot 10^{-5}$
10^6	$1.206 \cdot 10^{-6}$
10^7	$1.206 \cdot 10^{-7}$

inaccurate. For $n = 10^6$ and $n = 10^7$ we got some good results though. We can clearly see that the specialized algorithm has reduced the time significantly. If we used this algorithm for even higher n, the difference would be more significant. However for the LU-decomposition solution, the only recorded time I was able to get was for $n = 10^3$. For smaller n, the time was recorded as 0, and for higher n, the program haven't finished running for quite some time. This is not surprising, as the number of FLOPS for this method is $\sim O(n^3)$, as mentioned earlier. For $n = 10^3$, that is $O(10^9)$ FLOPS, which is why it used longer time than the Thomas algorithm used for $n = 10^7$. When $n = 10^4$, the number of FLOPS is increased to $O(10^{12})$, which is pretty significant. That is also why the LU-decomposition method using $n = 10^4$ won't finish running on my computer.

The max relative error between the numerical and analytical solutions presented in table 2, looks exactly as expected. As mentioned in the method section, the error should decrease by a factor of 100, when the step size is decreased by a factor of 10. This is exactly what we see in the table. It is also interesting to look at the figures 1 and 2. You may actually see that the difference between the analytical and numerical graph is decreased in a logarithmic fashion, which is what we would expect.

5 Conclusion

The Thomas algorithm is more efficient than LU-decomposition when solving a set of linear equations set up by a tridiagonal matrix. For the special case when the elements are equal along the diagonals, we can specialize the algorithm and reduce the number of FLOPS, and hence the computation time significantly. The relative error is decreased by a factor of 100, when the step size is decreased by a factor of 10.

References

[1] Morten Hjort-Jensen, *Computational Physics: Teach yourself C++*, `http://compphysics.github.io/ComputationalPhysics/doc/pub/learningcpp/html/learningcpp-bs.html`

[2] Morten Hjort-Jensen, *Computational Physics: Lecture Notes Fall 2015*, `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`