Computational Physics - FYS3150
Project 1

Sep 5, 2019

Abstract

The one dimensional Poisson equation is solved using algorithms based on Gaussian elimination and LU decomposition. The algorithms are compared based on efficiency and accuracy.

1 Introduction

Students are often told that there are many ways of getting from A to B in a problem. Often with the subtext that it does not matter what path you take, as long as you end up at the right place. However, what if you have to traverse the same path thousands or even millions of times? Choosing the shortest path, or rather minimizing the number of steps, is often extremely important when writing computer algorithms.

The aim of this project is to explore this idea through solving the one dimensional Poisson equation

$$-\frac{d^2u}{dx^2} = f(x) \tag{1}$$

with different algorithms. By looking at CPU time and estimating the error for each algorithm, the goal is to consider what features of an algorithm is advantageous in problems like this.

All code, produced data and figures are available at the GitHub repository HHaughom/FYS3150-Project1.

2 Theory

The first aim of the project is to solve the one dimensional Poisson's equation with the the range and boundaries given below.

$$-\frac{d^2u}{dx^2} = f(x) = 100e^{-10x}, \qquad x \in (0,1) \qquad u(0) = u(1) = 0 \tag{2}$$

When discretized, $u \longrightarrow u_i$ and $f(x) \longrightarrow f_i$, where the index $i \in [1, n]$ and the stepsize $h = \frac{1}{n}$. As viewed in the lecture notes (Hjort-Jensen, 2015), it is then possible to approximate the second derivative

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} \approx f_i. \tag{3}$$

The value of f_i is proportional to a weighted sum of the corresponding u_i and its next and previous terms u_{i+1} and u_{i-1}. This means that a matrix can be constructed where the diagonal keeps the coefficient of u_i, the left hand neighbour keeps the coefficient of u_{i-1}, while the right hand neighbour keeps the coefficient of u_{i+1}.

This way, all values u_i can be stored in a vector $\vec{u}$ and all $h^2 f_i$ in a vector $\vec{g}$, and the equations can be written as the linear transformation

$$A\vec{u} \equiv \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & \dots & \dots & \dots \\ 0 & -1 & 2 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 2 & -1 \\ 0 & \dots & \dots & \dots & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ u_{n-1} \\ u_n \end{bmatrix} = h^2 \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ f_{n-1} \\ f_n \end{bmatrix} \equiv \vec{g} \tag{4}$$

A straightforward way of approaching this problem is using Gaussian elimination. However, instead of jumping head first into a specialized algorithm, it is worth noting that matrix A is tridiagonal; only the diagonal and its neighbour elements are non-zero. For matrices on this form, there exists a simplified form of Gaussian elimination, known as the Thomas algorithm (Thomas, 1949). It is specialized for dealing with matrices on the form

$$(\mathbf{A}|\tilde{\mathbf{f}}) = \begin{bmatrix} d_1 & b_1 & 0 & \dots & \dots & \dots & g_1 \\ a_2 & d_2 & b_2 & \dots & \dots & \dots & g_2 \\ 0 & a_3 & d_3 & \dots & \dots & \dots & g_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & d_{n-1} & b_{n-1} & g_{n-1} \\ \dots & \dots & \dots & \dots & a_n & d_n & g_n \end{bmatrix} \tag{5}$$

In other words, the coefficients a_i, d_i and b_i may vary over the range of integration. Like in standard Gaussian elimination, the first aim of the Thomas

algorithm is to obtain an upper triangular matrix through row reduction. This will be on the form

$$(\mathbf{A}|\tilde{\tilde{\mathbf{f}}}) \sim \begin{bmatrix} d_1 & b_1 & 0 & 0 & 0 & \dots & g_1 \\ 0 & \tilde{d_2} & b_2 & 0 & 0 & \dots & \tilde{g}_2 \\ 0 & 0 & \tilde{d_3} & b_3 & 0 & \dots & \tilde{g}_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \tilde{d_{n-1}} & b_{n-1} & \tilde{g_{n-1}} \\ \dots & \dots & \dots & \dots & 0 & \tilde{d_n} & \tilde{g}_n \end{bmatrix},$$

where the b_i elements are unchanged and

$$\begin{aligned} \tilde{d_i} &= d_i - \frac{a_i}{\tilde{d_{i-1}}} b_{i-1}, & \tilde{d_1} &= d_1 = 2 \\ \tilde{g_i} &= g_i - \frac{a_i}{\tilde{d_{i-1}}} \tilde{g}_{i-1}, & \tilde{g_1} &= g_1. \end{aligned} \tag{6}$$

Then, by backwards substitution, the u_i's can be solved for

$$u_{i-1} = \frac{\tilde{g}_{i-1}}{\tilde{d}_{i-1}} - \frac{b_{i-1}}{\tilde{d}_{i-1}} u_i, \qquad u_n = \frac{\tilde{g_n}}{\tilde{d_n}}. \tag{7}$$

See the Appendix for a fleshed out derivation of the Thomas Algorithm. From equation (6) and (7) it is clear that this algorithm will require 9 floating point operations (FLOPS) per run.

While the general Thomas algorithm clearly encompass the special case of Poisson's equation, the number of FLOPS can be reduced by substituting $d_i = 2$ and $a_i = b_i = -1$. This gives the specialized version

$$\begin{aligned} \tilde{d_i} &= 2 - \frac{1}{\tilde{d}_{i-1}}, & \tilde{d_1} &= d_1 = 2 \\ \tilde{g_i} &= g_i + \frac{\tilde{g}_{i-1}}{\tilde{d}_{i-1}}, & \tilde{g_1} &= g_1 \end{aligned} \tag{8}$$

and

$$u_n = \frac{\tilde{g_n}}{\tilde{d_n}} \qquad u_{n-1} = \frac{\tilde{g}_{n-1}}{\tilde{d}_{n-1}} - \frac{u_n}{\tilde{d}_{n-1}}. \tag{9}$$

The equations in (7) and (8) contain 7 FLOPS total. Further manipulation of $\tilde{d_i}$ and $\tilde{g_i}$, makes it possible to reduce this to number to 4. Although this is not implemented in this project.

For comparison, the number of FLOPS of a general LU decomposition algorithm goes like n^3, where n is the number of steps (Hjort-Jensen, 2015).

3 Method

Both the general and the specialized algorithm presented in equations (6) - (9) were implemented in a C++ script. For comparison, a third set of solutions were produced by the LU decomposition functionality from the C++ Armadillo library (Curtin & Sanderson, 2016). In other words, the algorithms discussed are

1. The general Thomas algorithm
2. The specialized Thomas algortihm
3. Armadillo LU decomposition

Each of these are tested with seven different stepsizes

$$h = \frac{1}{n}, \qquad n = 10^k, \ k \in [1, 2, ..., 7],$$

amounting to a total of $3 \cdot 7 = 21$ different runs. For each of the 21 runs, the relative error

$$\epsilon_i = \log_{10}\left(\left|\frac{u_i - u(x_i)}{u(x_i)}\right|\right). \tag{10}$$

were measured. Where

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}. \tag{11}$$

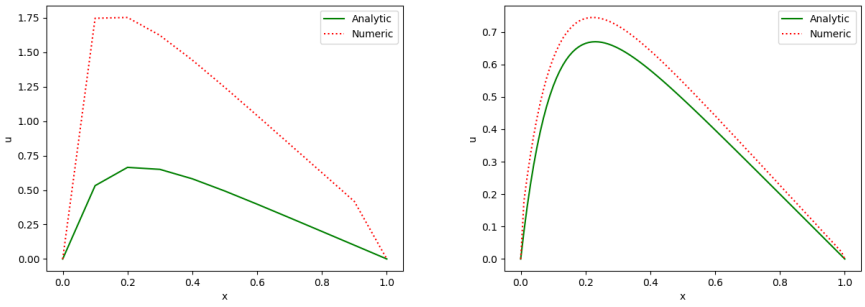
is the known analytical solution of Poisson's equation (2). In addition, The CPU time for each algorithm was recorded. For the general Thomas algorithm, both the the forward and backwards substitution (equation (6) and (7)) were included in the timing. For the special case, because the forward substitution can be simplified, we only time the backward substitution (equation 9). This feels a little bit like cheating, but the main purpose here is to get an idea of the CPU time in the optimized case. For the Armadillo algorithm, both the LU decomposition of the matrix and the subsequent solving is timed.

4 Results

Below follows plots of the numerical numerical estimation along with the analytical solution for the different algorithms and for a selection of stepsizes $h = 1/n$. All the referenced algorithm implementations can be found in "Algorithms.cpp" in the GitHub repository linked to in the introduction.

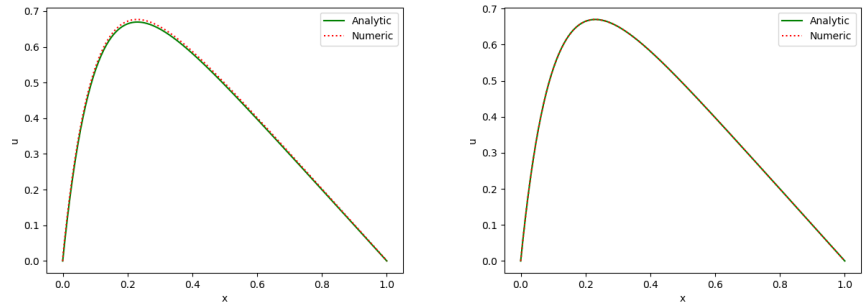
4.1 General Thomas algorithm

The plotted data is calculated from the *algo9n* function in "Algorithms.cpp".



(a) $n = 10$ (b) $n = 100$

Figure 1: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by algo9n function.

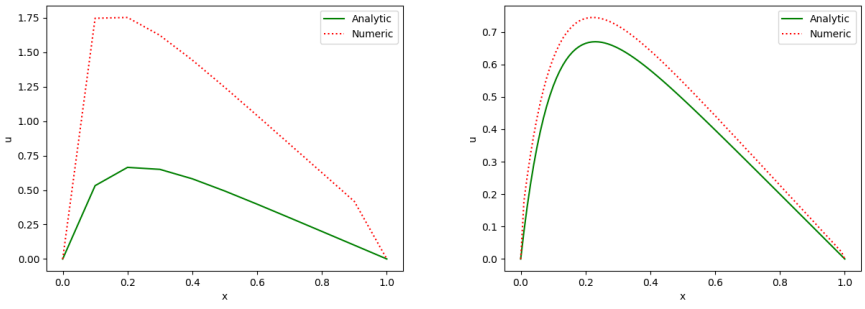


(a) $n = 1000$ (b) $n = 10000000$

Figure 2: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by algo9n function.

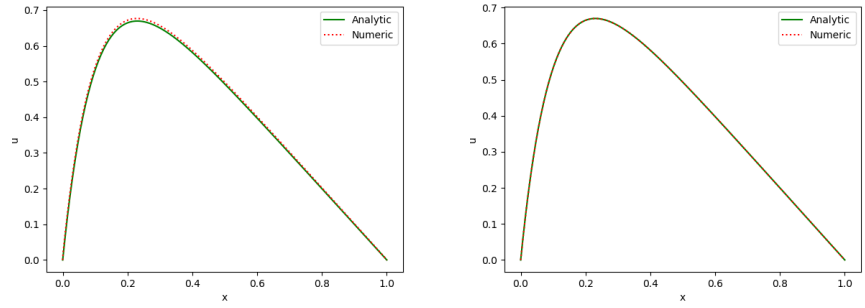
4.2 Specialized Thomas algorithm

The data is calculated from the *algo4n* function found in "Algorithms.cpp".



(a) $n = 10$ (b) $n = 100$

Figure 3: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by algo4n function.

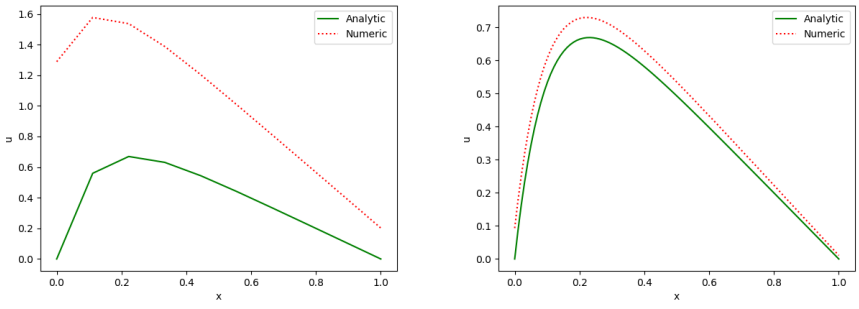


(a) $n = 1000$ (b) $n = 10000000$

Figure 4: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by algo4n function.

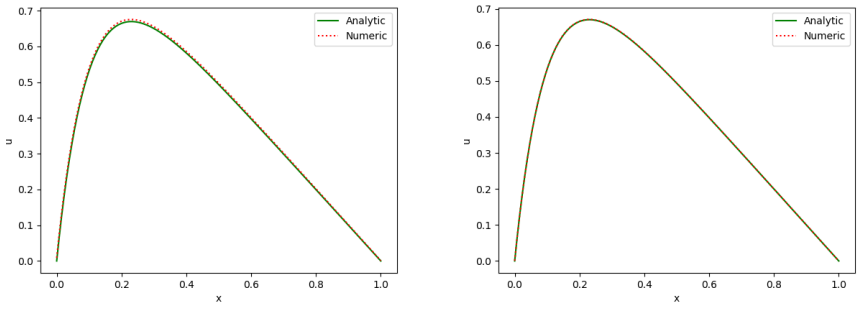
4.3 LU decomposition in Armadillo

The data is calculated from the *arma_lu* function found in "Algorithms.cpp".



(a) $n = 10$ (b) $n = 100$

Figure 5: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by Armadillo functionality.



(a) $n = 1000$ (b) $n = 10\,000$

Figure 6: Plot of numerical estimation compered to analytical solution with n integration points. Datapoints produced by Armadillo functionality.

4.4 CPU time

In Table 1 below, the CPU time of each algorithm is displayed. The numbers changed for each run, but kept its order of magnitude. The numbers are the median of three different runs who's output can be found in the "timing_outputs.txt" in the repository. The computer ran out of memory for the Armadillo algorithm at $N = 10^5$ steps and only the first four runs could be implemented.

Steps, $\log_{10}$ N	General [ms]	Specialized [ms]	Armadillo LU [ms]
1	0	0	5
2	1	1	2
3	11	10	166
4	114	126	18 972
5	830	834	-
6	9 110	9 108	-
7	119 178	119 947	-

Table 1: The typical CPU time for the different algorithms and for different stepsizes.

4.5 Relative error

In Table 2 below, the relative error for each run is displayed.

Steps, $\log_{10}$ N	General	Specialized	Armadillo LU
1	2.28278	2.28278	1.96439
2	1.09616	1.09616	1.07504
3	1.00936	1.00936	1.00735
4	1.00093	1.00093	1.00073
5	1.00009	1.00009	-
6	1.00001	1.00001	-
7	1.00000	1.00000	-

Table 2: The relative error for the different algorithms and for different stepsizes.

Note that the error for the general and the special Thomas algorithm is identical. Below is a logarithmic plot of the relative errors as a function of number of steps.

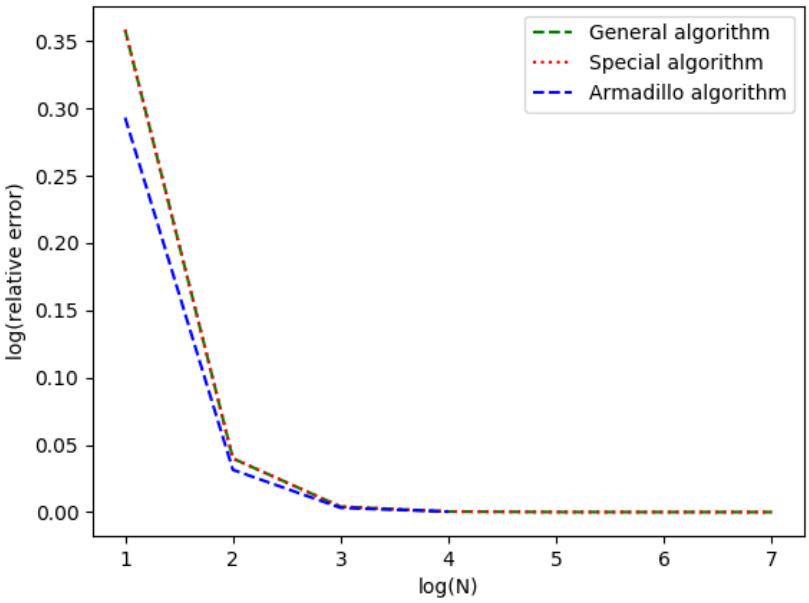


Figure 7: Logarithmic plot of relative error over the number of steps. The entries for the general and special algorithm are identical, while the Armadillo plot stops at log(N)=4.

5 Discussion

The CPU timing and error analysis gave several unexpected results.

First, the CPU times for the general and specialized algorithms are remarkably similar. This despite the the 1/3 ratio of the FLOPS. For the lower step numbers n, the time scale is too low to give credible results. Processes unaccounted for can affect the result significantly. However, even at the highest n's, the CPU time is slightly lower for the general algorithm. In other words, the FLOPS do not appear to be the dominating factor for the CPU time. It is unclear how much this is related to the script and how much it is related to the computer running it. The number of memory reads and writes included in the timing is even higher in the general algorithm and cannot explain the result. To explore this further, a higher number of test runs should be done, using a different computer and preferably a higher n.

All this said, the number of FLOPS are of the same order of magnitude for both algorithms. It is possible that a difference of higher order, for instance n^2, is required to see an effect on the CPU time.

The CPU time for the LU decomposition method is significantly higher than for the Thomas algorithms. The Armadillo functions used are a little like black

boxes, and without looking at the algorithm behind, it is hard to estimate the true number of FLOPS. If n^3 is taken as an estimate, the initial thesis of a correlation between number of FLOPS and CPU time, holds for this case. As seen for the Thomas algorithm, it is unclear how sensitive this correlation is. Still, it is clear that LU decomposition is not a good strategy for solving this problem. Not the least because of the enormous memory usage.

The relative error is strictly decreasing as a function of steps for all the algorithms. The smaller the step size, the better the accuracy. Something that is not immediately obvious, is why the error seems to approach exactly 1.0, regardless of the algorithm. However, upon writing out some of the values of both the analytic and numeric vectors, it is clear that the biggest error is always in the first $u = u_1$. For some reason, the order of magnitude of the numeric approximation is always one lower than the analytic solution at this point. In other words,

$$\epsilon_{max} = \epsilon_1 = \log_{10}\left(\left|\frac{u_i - u(x_i)}{u(x_i)}\right|\right) \approx \log_{10}\left(\left|\frac{-u(x_i)}{u(x_i)}\right|\right) = \log_{10}(1) = 0.$$

So even if the absolute error keeps declining, the relative error is approaching 1. It is possible that this is just a built-in feature in the algorithm. However, because the Armadillo algorithm behaves the same way, it is naturally to question if there is some logical error in the code. It is not very surprising that the Thomas algorithms have the same error. After all, they are based on the same expression. That said, a higher number of FLOPS makes the program more susceptible to floating point errors, and the error might not stay the same for higher n. The LU decomposition is slightly better at n<3, but the difference evens out after this point.

6 Conclusion

For the uncritical user, computers can at times feel like an unlimited source of power. Reality will hit you in the face soon enough with overflow or memory errors. It is then necessary to think carefully about what you are asking it to do. Although the Armadillo LU decomposition makes for cleaner looking code and its accuracy is competitive, its brute-force approach is not the right one in a problem like this. Although it is not obvious why the general Thomas algorithm performed as good as the specialized, it is clear that these are better alternatives for solving the set of equations. Even if factors like memory writes and function calls may have played in during the CPU timings, it is clear that Armadillos LU decomposition cannot compete with the linear evolution of FLOPS in the Thomas algorithm.

References

[1] Hjort-Jensen, M. Computational Physics. University of Oslo, 2015.

[2] Thomas, L.H. Elliptic Problems in Linear Differential Equations over a Network. Watson Science Computer Laboratory Report. Columbia University, 1949.

[3] Curtin, R. Sanderson, C. Armadillo: a template-based C++ library for linear algebra. Journal of Open Source Software, Vol. 1, pp. 26, 2016.

Appendix

Derivation of the Thomas algorithm

Forward substitution

Multiplying the first row by $\frac{a_2}{d_1}$ and subtracting it from the second row

$$(\mathbf{A}|\tilde{\tilde{\mathbf{f}}}) \sim \begin{bmatrix} d_1 & b_1 & 0 & 0 & \dots & g_1 \\ 0 & \tilde{d_2} & b_2 & 0 & \dots & \tilde{g_2} \\ & & & \dots & & \end{bmatrix} \qquad \tilde{d_2} \equiv d_2 - \frac{a_2}{d_1} b_1, \quad \tilde{g_2} = g_2 - \frac{a_2}{d_1} g_1$$

Repeating the process for the third row, multiplying the second row by $\frac{a_3}{\tilde{d_2}}$ and subtracting it from the third

$$(\mathbf{A}|\tilde{\tilde{\mathbf{f}}}) \sim \begin{bmatrix} d_1 & b_1 & 0 & 0 & 0 & \dots & g_1 \\ 0 & \tilde{d_2} & b_2 & 0 & 0 & \dots & \tilde{g_2} \\ 0 & 0 & \tilde{d_3} & b_3 & 0 & \dots & \tilde{g_3} \\ & & & \dots & & & \end{bmatrix} \qquad \tilde{d_3} \equiv d_3 - \frac{a_3}{\tilde{d_2}} b_2, \quad \tilde{g_3} = g_3 - \frac{a_3}{\tilde{d_2}} \tilde{g_2}$$

Continuing this process all the way to the nth row, we end up with the augmented matrix

$$(\mathbf{A}|\tilde{\tilde{\mathbf{f}}}) \sim \begin{bmatrix} d_1 & b_1 & 0 & 0 & 0 & \dots & g_1 \\ 0 & \tilde{d_2} & b_2 & 0 & 0 & \dots & \tilde{g}_2 \\ 0 & 0 & \tilde{d_3} & b_3 & 0 & \dots & \tilde{g}_3 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \tilde{d_{n-1}} & b_{n-1} & \tilde{g_{n-1}} \\ \dots & \dots & \dots & \dots & 0 & \tilde{d_n} & \tilde{g}_n \end{bmatrix},$$

where the b_i elements are unchanged and

$$\tilde{d_i} = d_i - \frac{a_i}{\tilde{d_{i-1}}} b_{i-1}, \qquad \tilde{d_1} = d_1 = 2$$
$$\tilde{g_i} = g_i - \frac{a_i}{\tilde{d_{i-1}}} \tilde{g}_{i-1}, \qquad \tilde{g_1} = g_1.$$

Backward substitution

With this upper diagonal matrix, the set of equations can now be solved with backwards substitution. Starting from the bottom, we have

$$u_n = \frac{\tilde{g_n}}{\tilde{d_n}}.$$

The next row gives

$$u_{n-1} = \frac{\tilde{g}_{n-1}}{\tilde{d}_{n-1}} - \frac{b_{n-1}}{\tilde{d}_{n-1}} u_n,$$

and in general the (i-1)th point of the solution is

$$u_{i-1} = \frac{\tilde{g}_{i-1}}{\tilde{d}_{i-1}} - \frac{b_{i-1}}{\tilde{d}_{i-1}} u_i.$$

To summarize, we solve the generalized equation by deciding the elements of the upper triangular matrix. Then we work our way up the reduced matrix and solve for each element u_i of the solution.