Numerical methods for solving second order linear differential equations

Daniel Pinjusic, Gudmund Gunnarsen and Ida Due-Sørensen

September 10, 2020

Abstract

For this project, the goal was to familiarize ourselves with C++ numerical programming, in particular dynamic memory allocation and matrix and vector operations. The topical problem was solving the Poisson equation using numerical methods, gaining a greater understanding of algorithm run times, numerical approximations and round off errors. We found that using a specialized Thomas algorithm written specifically for the problem at hand was the fastest option at 0.05 ms for $n = 10^3$ mesh-points. Meanwhile the much more general method of LU-decomposition, not taking into account the simple tridiagonal form of the matrix was much slower at 199.95 ms for the same number of mesh-points. This LU algorithm would simply fail at around $n = 10^5$ mesh-points because of memory issues, while the specialized alogorithm could run all the way up to around $n = 10^7$ mesh-points. At this point roundoff errors start accumulating: The relative error in this project decreased for up to about $n = 10^6$ mesh-points, but using $n = 10^7$ mesh-points increased the relative error, signifying a loss of numerical precision.

1 Introduction

In this project[1] we aim to numerically solve Poissons's equation, which is a partial differential equation on the form

$$\nabla^2 \Phi = f,$$

simplifying it to one dimension and using Dirichlet boundary conditions. It can be shown that Poisson's equation in one dimension can be written as a set of ordered linear equations, and that these linear equations can be written as a linear system, whose solution is given by $\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}$.

The matrix $\mathbf{A}$ is in this case a tridiagonal matrix, and we will be looking at different methods of solving such a linear system whose matrix is tridiagonal using forward and backwards substitution and LU decomposition. The method for forward and backwards substitution will be split into a general and special algorithm, making use of the fact that the diagonals of $\mathbf{A}$ are constants. These methods will then be compared with respect to both speed and accuracy, as well as difficulty of implementation and use case.

It is also our aim to familiarize ourselves with the C++ programming language, especially the handling of arrays and dynamic memory allocation, but also the usage of the armadillo library, which provides a tool kit for solving linear equations in C++.

2 Theory

2.1 The Poisson Equation

The well known Poisson's equation is a partial differential equation with broad utility in electrostatics, mechanical engineering, and theoretical physics. For the case of electrostatics in three dimensions, the equation is given by

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}). \tag{1}$$

where Φ is the potential field and $\rho(\mathbf{r})$ is the localized charge distribution. By assuming that both Φ and $\rho(\mathbf{r})$ is spherically symmetric simplifies equation (1) to a one dimensional equation in r,

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(\mathbf{r}).$$

We let $\Phi = \frac{\phi(r)}{r}$ to simplify the equation once more, which gives

$$\frac{d^2\phi}{dr^2} = -4\pi\rho(r).$$

We see that we can write the Poisson equation in a general form by letting $\phi \to u$, $r \to x$ and $f = -4\pi\rho(r)$,

$$-u''(x) = f(x). \tag{2}$$

3 Method

3.1 Discretization of the Poisson equation

The one-dimensional Poisson equation with Dirichlet boundary conditions was solved by rewriting it as a set of linear equations. Explicitly, we assumed that the source term is $f(x) = 100e^{-10x}$ with $x \in [0,1]$ and that $u(0) = u(1) = 0$. The exact, closed-form, solution of equation (2) can then be written as $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$. To solve the Poisson equation numerically, the boundary value problem is discretized to yield

$$\begin{aligned} x &\to x_i \in [x_0, x_1, \dots, x_{n+1}] \\ u(x) &\to v(x_i) = v_i \in [v_0, v_1, \dots, v_{n+1}] \\ f(x) &\to f(x_i) = f_i \in [f_1, f_2, \dots, f_n], \end{aligned}$$

where x is given by $x_i = x_0 + ih$ with step length, $h = \frac{x_{n+1}-x_0}{n+1} = \frac{1}{n+1}$. By applying the Dirichlet boundary conditions, we get

$$v_0 = v(x_0) = v(0) = 0, \quad v_{n+1} = v(x_{n+1}) = v(1) = 0.$$

Then we use the mathematical definition of the second order derivative of a function $u(x)$ given by

$$u''(x) = \lim_{h\to 0} \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}, \tag{3}$$

to see that the second order discrete approximation of equation equation (2) can be written as

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = f_i. \tag{4}$$

We rewrite this expression to

$$-v_{i+1} + 2v_i - v_{i-1} = \tilde{b}_i \quad \text{with} \quad \tilde{b}_i = h^2 f_i.$$

This is a set of linear equations

$$\begin{aligned}
i = 1: &\qquad & -v_2 - v_0 + 2v_1 &= \tilde{b}_1,\\
i = 2: &\qquad & -v_3 - v_1 + 2v_2 &= \tilde{b}_2,\\
\vdots &\qquad & &\vdots\\
i = n: &\qquad & -v_{n+1} - v_{n-1} + 2v_n &= \tilde{b}_n,
\end{aligned}$$

that can be written as the linear system

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \vdots \\ \vdots \\ \tilde{b}_n \end{bmatrix}. \tag{5}$$

As we can see, $\mathbf{A}$ is a tridiagonal-constant matrix, also called Töplitz matrix. For simplicity we can write the system above as

$$\mathbf{A} \cdot \mathbf{v} = \tilde{\mathbf{b}},$$

which is solvable by Gaussian elimination.

3.2 Tridiagonal matrix algorithm

In this section we use the the tridiagonal matrix algorithm, also known as the Thomas algorithm, to solve the linear system defined in equation (6). This algorithm is a simplified form of Gaussian elimination that can be used to solve tridiagonal systems of equations and consists of three steps: (1) decomposing the matrix to vectors, (2) forward substitution and (3) backward substitution. The tridiagonal matrix alorithm is a general algorithm, thus we introduce a general tridiagonal system for n unknowns may be written as

$$\mathbf{A} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix} \tag{6}$$

Step 1: We decompose the general tridiagonal matrix to three one-dimensional vectors of length n.

$$\mathbf{A} \to a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}, c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_n \end{bmatrix}$$

Step 2: To get a pivot column, we multiply the first row with a_1/b_1 and subtract the first row from the second row which gives us the matrix

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & c_2 & 0 & & \vdots \\ 0 & a_2 & b_3 & c_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & 0 & a_{n-1} & b_n \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{B}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_n \end{bmatrix}$$

where

$$d_2 = b_2 - c_1 \frac{a_1}{b_1} \quad \tilde{B}_2 = \tilde{b}_2 - \tilde{b}_1 \frac{a_1}{b_1}.$$

By repeating the procedure with the modified second equation on the third equation and so on, we find the general pattern

$$d_i = b_i - c_{i-1} \frac{a_{i-1}}{d_{i-1}}, \quad \tilde{B}_i = \tilde{b}_i - \tilde{B}_{i-1} \frac{a_{i-1}}{d_{i-1}}, \tag{7}$$

where $d_1 = b_1$ and $\tilde{B}_1 = \tilde{b}_1$.

Pivot elements have now successfully been produced in every column in the matrix and our problem is now on the form

$$\begin{bmatrix} d_1 & c_1 & 0 & \dots & 0 \\ 0 & d_2 & c_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & d_{n-1} & c_{n-1} \\ 0 & \dots & \dots & 0 & d_n \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{B}_1 \\ \tilde{B}_2 \\ \vdots \\ \vdots \\ \tilde{B}_n \end{bmatrix}.$$

Step 3: We perform a backward substitution to find the solution set $\mathbf{u}$. Writing out the matrix equation gives

$$\begin{aligned} d_1 v_1 + c_1 v_2 &= \tilde{B}_1 \\ d_2 v_2 + c_2 v_3 &= \tilde{B}_2 \\ &\vdots \\ d_{n-1} v_{n-1} + c_{n-1} v_n &= \tilde{B}_{n-1} \\ d_n v_n &= \tilde{B}_n. \end{aligned}$$

Then solving for v_n and the general pattern for $v_i,\ i \in \{1, 2, \dots, n-1\}$ gives

$$v_n = \frac{\tilde{B}_n}{d_n}, \quad v_i = \frac{\tilde{B}_i - v_{i+1} c_i}{d_i}. \tag{8}$$

Algorithm 1: Forward substitution

```
for $i = 1$ to $n$ do
    $d_i = b_i - c_{i-1}\frac{a_{i-1}}{d_{i-1}}$;
    $\tilde{B}_i = \tilde{b}_i - \tilde{B}_{i-1}\frac{a_{i-1}}{d_{i-1}}$;
end
```

Algorithm 2: Backward substitution

```
for $i = n$ to 1 do
    if $i = n$ then
        $v_i = \frac{\tilde{B}_n}{d_n}$;
    else
        $v_i = \frac{\tilde{B}_i - v_{i+1} \cdot c_i}{d_i}$;
    end
end
```

3.3 Optimizing the tridiagonal matrix algorithm

In this work, the tridiagonal matrix $\mathbf{A}$ have identical matrix elements along the diagonal and identical (but different) values for non-diagonal elements ($a_i = c_i = -1$ and $b_i = 2$). For this specific case we can optimize the tridiagonal matrix algorithm. We rewrite the general algorithm presented in subsection 3.2 to fit this specific case. Inserting the values for a_i, b_i and c_i in the expression for d_i (equation (7)) gives

$$d_2 = 2 - \frac{1}{2} = \frac{3}{2} \tag{9}$$

$$d_3 = 2 - \frac{1}{3/2} = \frac{4}{3} \tag{10}$$

$$\vdots \tag{11}$$

$$d_i = \frac{i+1}{i}. \tag{12}$$

By inserting the expression for d_i in equation (7) and equation (8) we obtain the optimized tridiagonal matrix algorithm.

Algorithm 3: Optimized forward substitution

```
for $i = 1$ to $n$ do
    $d_i = \frac{i+1}{i}$;
    $\tilde{B}_i = \tilde{b}_i + \tilde{B}_{i-1}\frac{(i-1)}{i}$;
end
```

Algorithm 4: Optimized backward substitution

```
for $i = n$ to 1 do
    if $i = n$ then
        $v_i = \tilde{B}_n \frac{n}{n+1}$;
    else
        $v_i = v_{i+1} + \tilde{B}_i \frac{i}{i+1}$;
    end
end
```

3.4 Error analysis

The relative error ϵ_i in the data set $i = 1, \ldots, n$ is defined by

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right), \tag{13}$$

where u_i and v_i are the analytic and numerical solutions, respectively, with $u_i = u(x_i)$. This quantity was used to evaluate the accuracy of the numerical solution, and to compare the accuracy of the different algorithms used.

3.5 LU decomposition

A more general solution to the matrix equation equation (5) is using LU decomposition. This does not take into account the tridiagonal form or the simple values, and we expect a slower algorithm. In LU decomposition, the matrix $\mathbf{A}$ is factored into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, as $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$. This is possible assuming the matrix $\mathbf{A}$ is not singular [2].

LU decomposition allows for separating equation equation (6) into two new equations,

$$\mathbf{L} \cdot \mathbf{w} = \tilde{\mathbf{b}},$$
$$\mathbf{U} \cdot \mathbf{v} = \mathbf{w},$$

defining also **w**. In matrix form, these equations are more easily solved by Gaussian elimination due to the triangular nature of **L** and **U**:

$$\begin{bmatrix} L_{11} & 0 & \dots & 0 \\ L_{21} & L_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ L_{n1} & \dots & \dots & L_{nn} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \vdots \\ \tilde{b}_n \end{bmatrix}, \tag{14}$$

$$\begin{bmatrix} U_{11} & U_{12} & \dots & U_{1n} \\ 0 & U_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & U_{nn} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} w_1 \\ \vdots \\ \vdots \\ w_n \end{bmatrix}. \tag{15}$$

The diagonal elements in **L** are scaled such that $L_{ii} = 1 \ \forall \ i$, which simplifies the algorithm for solving equation equation (14):

Algorithm 5: LU forward substitution

```
for $i = 1$ to $n$ do
    $w_i = \tilde{b}_i$
    for $j = 1$ to $i - 1$ do
        $w_i = w_i - L_{ij} w_j$
    end
end
```

Once w is found, the backward substitution is slightly more complex as the diagonal elements of **U** are not of the same simple form:

Algorithm 6: LU backwards substitution

```
for $i = n$ to $1$ do
    $v_i = w_i$
    for $j = n$ to $i + 1$ do
        $v_i = v_i - U_{ij} v_j$
    end
    $v_i = \frac{v_i}{U_{ij}}$
end
```

Using LU-decomposition is significantly more taxing on both memory and number of floating point operations than the simple tridiagonal solution above. The entire matrix is in the memory while performing the LU-decomposition, which limits the size of the matrix compared to the three vectors in the tridiagonal case.

3.6 Counting number of floating point operations

We are interested in the number of floating point operations performed in each algorithm, to give an estimate of execution time. In equation (7), we see that to calculate d_i and $\tilde{B}_i$, one subtraction, one multiplication and one division is required in each case. Since this is performed for each i in the matrix up to n, this yields $6n$ floating point operations for the forward substitution.

Then for equation (7) we see that to calculate u_i one subtraction, one multiplication and one division is once more required. Therefore the total number of floating point operations in the general tridiagonal case should be $9n$.

Meanwhile, for the optimized tridiagonal case with $a_i, c_i = -1,\ b_i = 2,\ d_i$ can be precalculated as in equation (12), which means that for both the forward and backward substitutions, only two floating point operations are necessary. One addition, and one division in both cases. In total there are therefore only $4n$ floating point operations in the specialized case.

4 Results

Figure 1 shows the numerical solution using the general algorithm from section 3.2 plotted against the analytical one for $n = \{10, 100, 1000\}$ mesh-points. For the latter two cases, the error is too small to locate visually.

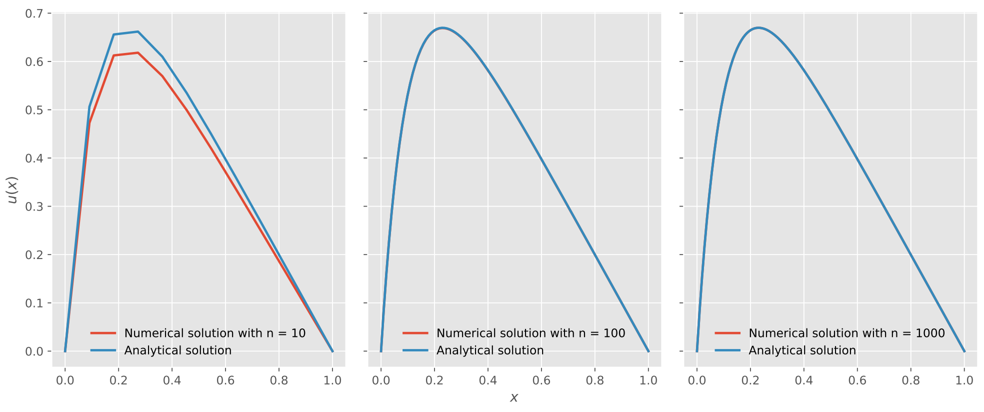

Figure 1: The numerical solution of the Poisson's equation with n mesh-points compared to the analytical solution.

4.1 Comparison of execution times

The execution times for small values of n are so small that they vary greatly with each execution of the program. The following values summarized in table 1 are therefore averages over 50 executions of each algorithm. For $n = 10^5$ and above, LU-decomposition fails as there is not enough memory available for the entire $10^5 \times 10^5$ matrix. Since each number in the matrix is a double precision floating point number occupying 8 bytes, a $10^5 \times 10^5$ matrix would require 8×10^{10} bytes or 80 GB of memory, which is beyond the average commercial computer.

Approximate execution time by algorithm [ms]			
n	General Tridiagonal	Special Tridiagonal	LU-decomposition
10^1	< 0.01	< 0.01	0.15
10^2	< 0.01	< 0.01	4.10
10^3	0.08	0.05	199.95
10^4	0.50	0.28	36741.21
10^5	4.34	2.73	-
10^6	42.23	27.11	-

Table 1: The approximate execution time in milliseconds for each of the three algorithms used.

4.2 Relative Error

The relative error is a useful measure for understanding how accurate the numeric results are. In table 2, the relative error from equation equation (13) is tabled, with only the maximum value of the error for a given value of n shown. In this case the special algorithm from section section 3.3 was used to calculate the numeric values v_i.

n	Relative Error [$\log_{10}$]
10^1	-1.1797
10^2	-3.0880
10^3	-5.0801
10^4	-7.0793
10^5	-9.0791
10^6	-10.1628
10^7	-9.0900

Table 2: The relative error in the special algorithm, for different values of n.

5 Discussion

The results show that all three methods discussed are capable of solving this set of linear equations, however, the efficiency of each method differs, and their general use case is also different. It is clear that both the general and special algorithm for forwards and backwards substitution are very accurate for big values of n, and they are both fast, with the special algorithm being slightly faster than the general case. LU decomposition on the other hand, is much slower due to its big demands on available memory, although it is just as accurate as the aforementioned methods. Although LU decomposition is slower, it will not always be the case that you will be dealing with a tridiagonal matrix, in which case LU decomposition might be the best solution available.

5.1 Algorithm Runtime

As is evident from table 1, LU-decomposition is enormously more time consuming than either of the tridiagonal algorithms. The difference between the special tridiagonal case and the general tridiagonal case however is much smaller. We know that LU-decomposition requires a number of floating point operations of the order $\mathcal{O}(\frac{2}{3}n^3)$ [2] for large n, while the execution time seems to increase approximately by a factor 100 for each row excepting the first. We should expect a factor 1000 for large n, but for comparatively low values such as $n \sim 10^3$ it seems that the forward and backwards substitution, as well as lower order dependent floating point operations in the decomposition itself are significant enough to impact the execution times. Testing with $n = \{2500, 5000, 10000\}$ yields approximate execution times of respectively 1022 ms, 5205 ms and 36741 ms, giving a factor of about 6, compared with the expected factor $2^3 = 8$.

On the other hand, the number of floating point operations for the tridiagonal cases should be, for the special and the general algorithms respectively, $4n$ and $9n$, as established previously. This means that the special case algorithm should take about half the time of the general case algorithm, while the results in table 1 are runtimes close to $\frac{2}{3}$ of the general case. This can be explained when one takes into account that the algorithms are timed in our case including initialization of values as well as precalculating d_i. Since precalculating d_i requires one addition and one division as in equation (12), the special algorithm including precalculations will increase to $6n$ floating point operations, and hence the approximate 1.5 times increase in calculation speed seems more reasonable.

5.2 Implementation of algorithms

The difficulty of implementing the general and special algorithms is quite easy and straightforward, with only a few lines of code needed for the actual calculation, ignoring the declaration of variables. Implementing an algorithm for the LU decomposition is, on the other hand not as trivial, but not too complicated either, however, we still chose to use the implementation built into armadillo.

Since the algorithm for LU decomposition was not made by us, but by the creators of armadillo, we can be confident that the implementation of the algorithm is quite optimal, so there is likely not much to be gained by tweaking this algorithm or creating our own, however, there could be some way of improving the algorithms coded by us, even if it could be argued that the implementations are good enough as is, and mostly bottle necked by memory size and allocation.

5.3 Error analysis

Looking closer at the relative error, we see from Table 2 that the relative error becomes smaller as n increases, but reaches a point near $n = 10^6$ where round off errors start building up, thereby increasing the relative error. This is an issue that might be fixed if we use more bits per number, but this would also increase memory usage and the speed of the implementation. For the simple calculations that were required for this project, there is no need to do such a thing as the results were quite good.

We see that as expected from the lectures [2], the slope of the relative error plotted against $\log_{10}(n)$ is approximately -2 up to about $n = 10^5$.

6 Conclusion

In conclusion, it is clear that it is very much worth the effort tailor make algorithms that are case specific when possible, as there is a lot of compute time to be saved. In this instance, even though the Thomas algorithm and LU decomposition work in similar ways, the compute time of both are vastly different because the LU decomposition method does not ignore all the rows of zeros in our matrix.

It was also made clear to us how important the chosen step size is, as it is important to choose a step size that balances numerical precision with round off errors.

7 Appendix

Github repository with codes and figures can be found at `https://github.com/idadue/ComputationalPhysics`.

References

[1] Hjort-Jensen, M: Project1,
`https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2020/Project1/pdf/Project1.pdf`

[2] Hjort-Jensen, M: Computational Physics,
`https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`