FYS3150 Project 1

Ole Petter Maugsten and Johan Emil Larsson

(Dated: September 10, 2020)

We solved a second order linear differential equation both analytically and numerically. We solved it numerically in three ways. The first two were a general and special implementation of the Thomas algorithm. The last one used library functions from armadillo. We did this to test the accuracy and effectiveness of the methods. We found that the Thomas algorithm were both faster and more accurate than the library functions. The specialised algorithm was faster because it needed less FLOPs, but since it used the same amount of grid points as the general, it had about the same accuracy.

I. INTRODUCTION

In this project we solve a second order linear differential equation both analytically and numerically. By comparing the numerical solutions to the analytical, we get a sense of how effective and accurate the methods are. For solving the problem efficiently, we rewrite the equation as a set of linear equations. We then use dynamic memory allocation and LU-decomposition for the matrices.

The problem we look at is the famous Poisson equation in electrostatics.

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

With spherical symmetry and a substitution $\Phi(r) = \phi(r)/r$, this reduces to a one-dimensional problem.

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

With $\phi \to u$ and $r \to x$, we write the general equation as

$$-u(x)'' = f(x) \tag{3}$$

We will solve this where $x \in (0, 1)$. For the numerical solution we need some fixed boundary conditions. These are given $u(0) = u(1) = 0$. The source term is assumed to be $f(x) = 100e^{-10x}$.

II. METHOD

The source term $f(x)$ is known, and finding an analytical solution is a matter of integration. We find it to be

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{4}$$

To solve the equation numerically, we discretize u to v_i with gridpoints x_i. The second derivative of u can be expressed by a three-point approximation such that

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \tag{5}$$

or

$$2v_i - v_{i+1} - v_{i-1} = h^2 f_i.$$

We rewrite $h^2 f_i = \tilde{b_i}$. Each element of $\underline{\tilde{b}}$ can be expressed

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

Here we have a set of linear equations. On vector form this translates to

$$\underline{\tilde{b}} = A\ \underline{v} \tag{6}$$

where A is the given $n \times n$ tridiagonal matrix. We solve equation (6) by implementing the Thomas algorithm. This is a special form of gaussian elimination for tridiagonal matrices such as A. To effectively solve this system we LU-decompose the matrix.

A. LU-decomposition

LU-decomposition is a method where we decompose a matrix A into two matrices L and U such that $LU = A$ and L is a lower triangular matrix and U an upper triangular matrix. To find U we simply row-reduce A until it is in upper-triangular form, and note the row-operations we perform. If we then perform the inverse of those row-operations on I the identity matrix, we obtain a lower-triangular matrix L. We illustrate this on the given tridiagonal matrix A,

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & 0 \\ a_1 & b_2 & c_2 & \dots & 0 \\ 0 & a_2 & b_3 & c_3 & \dots \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & a_{n-1} & b_n \end{bmatrix}$$

We propose that since A is tridiagonal, then L and U are both semi-tridiagonal. That is they have numbers along the diagonal and one of the offset-diagonals. We can verify this by performing the matrix multiplication of these two matrices and confirming that we indeed get a tridiagonal matrix. Let then,

$$U = \begin{bmatrix} d_1 & e_1 & 0 & \dots & 0 \\ 0 & d_2 & e_2 & \dots & 0 \\ 0 & 0 & d_3 & e_3 & \dots \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & 0 & d_n \end{bmatrix}$$

For L we know from the LU-decomposition method that its diagonal is filled with ones, such that,

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_1 & 1 & 0 & \dots & 0 \\ 0 & l_2 & 1 & 0 & \dots \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & l_{n-1} & 1 \end{bmatrix}$$

From the matrix multiplication of L and U we see that for the diagonal of A,

$$b_{i+1} = l_i e_i + d_{i+1} \tag{7}$$

for the other two diagonals we have, $c_i = e_i$ and

$$a_i = l_i d_i \tag{8}$$

Since $d_1 = b_1$ from the first row of L multiplied by the first column of U, we can then iterate through $i = 2, ..., n-1$ and find all d_i and l_i. The psuedocode for this is,

```
d[1] = b[1]
for i in 2...n:
    l[i-1] = a[i-1]/d[i-1]
    d[i] = b[i] - l[i-1]*c[i-1]
```

which is our algorithm for finding the LU-decomposition for A.

B. Solving the linear system $Av = b$

Using the LU-decomposition we can more simply solve the linear system $Av = b$, where $v, b \in \mathbb{R}^n$.

The reason why this method is very quick is because the LU-decomposition only needs to be found once. If you are solving the same system for many different b, and the LU-decomposition has already been found, the amount of operations is drastically reduced since you only need to then do the substituion part. We introduce the vector $u \in \mathbb{R}^n$ which solves $Lu = b$. Each element in u, u_i, is easily seen from the operations required to bring $[Lb]$ to $[Iu]$. For each row i, we need to subtract the row above multiplied by l_i, that is

$$u_i = b_i - u_{i-1} l_{i-1}$$

where $u_1 = b_1$ since no operations are needed on the first row. We know have the vector u and can solve the system $Uv = u$ by row-reducing $[Uu]$ to $[Iv]$. Similarly as before we look at which operations are need to bring U to I. We start from $i = n-1 ... 1$ and iterate backwards this time. For each row, we now need to subtract the row below multiplied by c_i and divide by d_i, that is

$$v_i = (u_i - v_{i+1} c_i)/d_i$$

where $v_n = d_n/a_n$. The psuedo-code for this algorithm is,

```
u[1] = b[1]

for i in 2...n:
    u[i] = b[i] - u[i-1]*l[i-1]

v[n] = d[n]/a[n]

for i in n-1...1
    v[i] = (u[i] - v[i+1]*c[i])/d[i]
```

Which solves our linear system $Av = b$. QED. This is what we will call the general algorithm. Since it can be used on any general tridiagonal matrix.

In this project since $a_i = c_i = -1$ and $b_i = 2$, we can optimize our algorithm somewhat. From equation 7 we can rewrite this as,

$$d_{i+1} - l_i = 2$$

Meanwhile equation 8 can be written as,

$$d_i = -1/l_i$$

combining these two equations gives,

$$d_{i+1} = 2 - 1/d_i$$

Since $d_1 = b = 2$ we can find the first few terms d_i, $d_2 = 2 - 1/2 = 3/2$, $d_3 = 2 - 2/3 = 4/3$, $d_4 = 2 - 3/4 = 5/4$, the pattern is simply $d_i = \frac{i+1}{i}$. This also gives $l_i = \frac{-i}{i+1}$. The algorithm above is then simplified since we do not need to find the LU-decomposition. The substitution can also be simplified,

$$u_i = b_i + u_{i-1}\frac{i-1}{i}$$

and

$$v_i = (u_i + v_{i+1})\frac{i+1}{i}$$

QED. We call this algorithm the specialized alogrithm since it is specific to a only one specific tridiagonal matrix.

C. Floating-point operations

We solve the numerical problem using different methods. It is interesting to see how fast each method is. To get a grip of this, we can count the floating-point operations (FLOPs). This is the number of arithmetic operations needed in the algorithms.

For inverting a general n times n matrix, the number of FLOPs needed is $O(n^4)$, where n is the size of the matrix. This is very demanding for computers in terms of allocating enough memory for large matrices.

A general LU-decomposition for a dense matrix is better in that the FLOPs needed are $O(n^3)$. It is $2/3n^3$ to be exact. Fortunately for us, the matrix A is only tridiagonal. It is also symmetric along the diagonal. This helps greatly for reducing FLOPs. By implementing the

Thomas algorithm (general solution), the total number of FLOPs reduces to $6n$. Since d_i can be pre-calculated $d_i = \frac{i+1}{i}$, it will not contribute with FLOPs. The total number of FLOPs needed for the specialised solution becomes $4n$, which is very good.

For a computer with 8GB of RAM, we could in theory calculate a solution using doubles (8 bytes per number in C++) with $n \sim 10^3$ grid points by LU-decomposing a dense matrix. If we instead use the Thomas algorithm, we could find a solution with $n \sim 10^9$. This will greatly improve the accuracy.

D. Execution of code

We have implemented both the generalized and the specialized algorithm in C++. Python is used as a wrapper to execute the code and time it. We time it using the python library function `time.time_ns` with a stated resolution of 84ns [3] on Linux-systems, which should be plenty for our applications. To get a representative value for the CPU-time of the program, we time each program 10 times and average the time used. Using the library armadillo for C++ [4], we used the functions `LU` and `solve` to find the solution of the linear system.

III. RESULTS

Solving the linear system given in the method section yields for $n = 10, ..., 1000$ the plot seen in figure 1. The analytical, exact solution of the equation 4, is labelled "Analytical". We found the CPU-time used by the generalized and specialized algorithms, plotted for different values of n in figure 2.

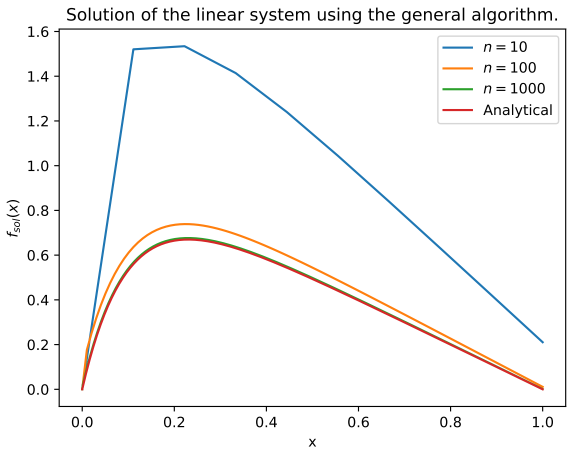

Figure 1. A plot of the solved system of equation using the general algorithm, compared with the analytical solution.

The relative error was found for the specialized algorithm and compared with the analytical solution, see figure 3. The first and last points of the results were thrown away, since the analytical solution at the end points is 0.

Curve	Peak	Error
$n = 10$	1.531	0.8613
$n = 100$	0.7383	0.0686
$n = 1000$	0.6764	0.0067
Analytical	0.6697	0.0

Table I. This table shows the difference between the numerical calculated peaks and the analytical solution.

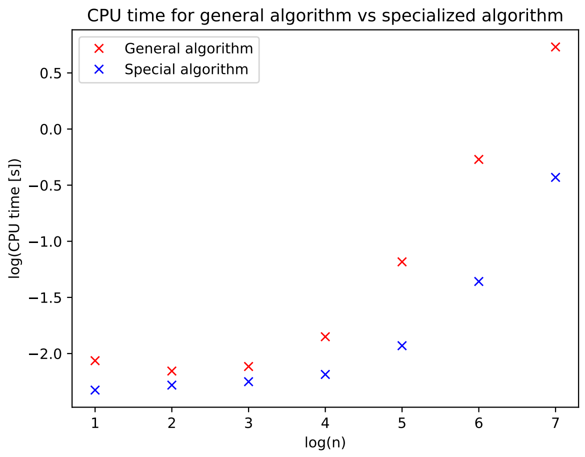

Figure 2. The elapsed CPU-time for the general and special algorithm.

We finally timed the specialized alogrithm and the library functions, see table II.

We note that for the generalized algorithm, we are able to solve with up to $n = 10^7$ grid points, for the specialized

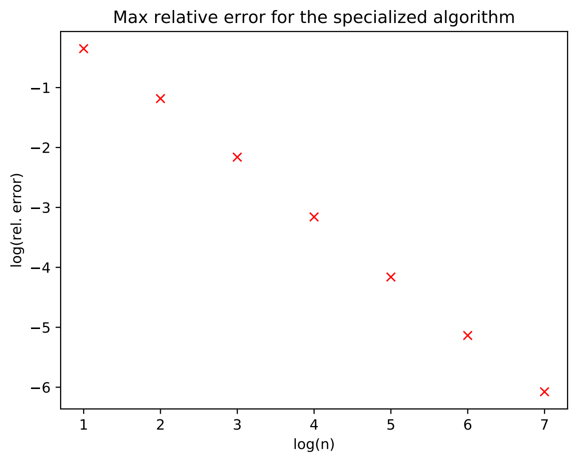

Figure 3. The relative error of the special algorithm using n points.

$\log(n)$	Special [s]	Library [s]
1	0.007	0.016
2	0.006	0.149
3	0.004	0.144
4	0.006	23.031

Table II. CPU-time of the specialized algorithm and the library functions (armadillo).

this value is $n = 10^8$ and using the library functions $n = 10^4$. Any larger values for n runs out of RAM on a linux computer with 8GB.

IV. DISCUSSION

From figure (4) we see how greatly the accuracy of the general solution improves when increasing the number n of grid points. As mentioned in the floating-point section, the error decreases with $O(n)$. Let us look at the peak of each solution, and calculate the difference from the analytical curve. The errors are listed in table (I). The error for the curve with $n = 10$ is about ten times greater than that of the curve with $n = 100$. The same goes for the curves $n = 100$ and $n = 1000$. This fits well with our prediction of the error decreasing as $O(n)$.

From figure (3) we see the relative error versus number of grid points. If we were to draw a line through the seven points its slope would be pretty accurately $a = -1$. Since figure (3) is a log-log plot, this means that the relative error decreases with $1/n$. We found that the special solutions would need $4n$ FLOPs, and that means the relative error should decrease as such. Fortunately it does, which substantiates the theory. From the figure, we can also see that the relative error in the first three points are about the same for the special and general solution. Since they respectively need $4n$ and $6n$ FLOPs, it means that the specialised solution is faster, but not more accurate than the general.

It is worth mentioning that with our computers with 8GB of RAM cannot calculate Thomas algorithm-solutions with more than $n \sim 10^8$ grid points. After that point, the double arrays would take up too much space for the memory to cope with.

If we look figure 2 we can compare the elapsed CPU-time consumed by the generalized algorithm compared with the specialized algorithm. We note that for $n = 4..6$ the general algorithm is slower than the special algorithm, for the smaller values of n they are practically equal. We note that these measurement were, particularly for small n, not very precise, which is why running the program multiple times and averaging the result was needed. This is likely due to effects from cold boot and task management from linux. The difference in CPU-time was about as expected, not having to perform the LU-decomposition for the specialized algorithm means that it needs less time to run. We can also see that the gap between the general and the special algorithm grows larger with n. Which is logical because the FLOPS required to perform the LU-composition grows faster than the FLOPS required to do only the substitution part of the algorithm (which is the main difference between the specialized and generalized algorithms). In table II we compare the library functions from Armadillo with the specialized algorithm. We see a stark difference between these two methods. Armadillo uses drastically more time than the specialized algorithm. We were not able to run for any larger values of n, so the precision of the results is somewhat debatable, however the difference from the specialized to library functions is greater than one magnitude for all values which fairly certainly means that the library functions are much slower. Particularly for the value $n = 4$, we can see from the resource usage on the system that it runs out of RAM and has to use the swap partition. This is why it is so much slower than the specialized algorithm. The armadillo implementation also does not assume a tridiagonal matrix, meaning that a lot of RAM is spent on storing the zeros of the matrix. As mentioned in the results, the armadillo implementation will not run for a matrix of size $10^5 \times 10^5$. This is despite the fact that mathematically, the generalized, specialized and armadillo implemented algorithm are the same. Meaning we have no mathematical error between the methods. The trade-off between the three methods is also clear. The specialized algorithm is naturally much more rigid than the generalized algorithm, which again is more rigid than the aramadillo implemented solution. If we wanted to solve a dense, non-tridiagonal matrix, we would be forced to use the slow armadillo implementation of the three.

V. CONCLUSION

We have in this project explored the solution $u(x)$ to the equation $-u''(x) = f(x)$ by discretizing u and solving the resulting linear system using LU-decomposition and substitution. Solving the linear system was done using three methods, first by a general method, which is able to solve any tridiagonal matrix, then by the specialized method which only solves the particular matrix corresponding to the discretized second derivative of u, and finally using the library functions from armadillo, which is able to solve for any real matrix. The specialized method was the fastest and required the least FLOPS, it is however much more rigid. The library functions from armadillo uses alot more resources, both CPU time and RAM.

VI. REFERENCES

1. M.H. Jensen. 2020. ComputationalPhysics, GitHub repository, https://github.com/CompPhysics/ComputationalPhysics

2. Lay, D.C., McDonald, S.R. & McDonald, Judi J, 2016. Linear algebra and its applications 5th ed., Harlow: Pearson.

3. Stinner, V. 2017. ”PEP 564 – Add new time functions with nanosecond resolution” https://www.python.org/dev/peps/pep-0564/

4. Sanderson, Conrad & Curtin, Ryan, 2016. Armadillo: a template-based C library for linear algebra. Journal of open source software, 1(2), p.26.l