Computational Physics
FYS4150

Written by:

Marius Enga, Patryk Krzyzaniak
Mohamed Ismail and Kristoffer Varslott

Department of Physics UiO

UiO : University of Oslo

Contents

Abstract

The one-dimensional Poisson equation can be discretized into a linear algebra problem, which previously was solved using LU-decomposition. However the algorithm for LU-decomposition uses $\frac{2}{3}n^3$ FLOPS, resulting in a slow algorithm and the computer running out of memory for large matrices. We developed two algorithms for solving this problem, with FLOPS in the order of n, a general algorithm (9n FLOPS) using arbitrary matrix elements and a specialised algorithm (4n) using precalculated elements. The method utilizes 3 vectors for describing the tridiogonal matrix from the discretized equation. The specialised algorithm, results in a fast algorithm with high precision up to a $10^6 \times 10^6$, where the general algorithm gets a loss of precision for lower dimensional matrices and is considerably slower due to higher number of FLOPS.

I. Introduction

In this project we are looking at a second order differential equation. This equation comes from the classical equation from electromagnetism:

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

This is called Poisson's equation. Where Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. We will rewrite the equation as a standard second order differential equation in one dimension:

$$-u''(x) = f(x). \tag{1}$$

This is our problem which we will evaluate. First off we are going to work towards a numerical approach, which we will implement within our code. We will then compare this result up against the (analytical solution for the problem) numerical calculation. The main task in this project is to write a program that takes into account numerous of details which minimizes the CPU-usage. For doing so we need to carefully construct the algorithm. We will also compare the algorithms to see which yields the best result. Evaluating the consisting errors occurring for different step lengths, as n is increasing. Further on we will look closely on the problem, and defining several algorithms for solving our problem.

Finally we will look at LU-decomposition and how the execution time is altered in comparison with the algorithms. Number of FLOPS will be essential as we are progressing in this project. We will look at how different algorithms behaves within our programs.

II. Method

As mentioned, we will numerically solve the second order differential equation:

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

Firstly we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$.
We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n, \tag{2}$$

where $f_i = f(x_i)$. Further on we will show that this is indeed a set of linear equations:

i. Linear equations

$$\mathbf{Av} = \mathbf{d}, \tag{3}$$

We can easily check that this is correct by inserting i = 1,2,3,...,n. into equation 2. Multiplying h^2 on both sides:

$$\begin{aligned}
-v_2 + 2v_1 &= h^2 f_1 \quad i = 1 \\
-v_3 - v_1 + 2v_2 &= h^2 f_2 \quad i = 2 \\
-v_4 - v_2 + 2v_3 &= h^2 f_3 \quad i = 3 \\
&\vdots \\
-v_{n-1} + 2v_n &= h^2 f_n \quad i = n
\end{aligned} \tag{4}$$

Where we have that $v_0 = v_{n+1} = 0$. Which gives us the matrix:

$$\begin{bmatrix} 2v_1 & -v_2 & 0 & \dots & \dots & 0 \\ -v_1 & 2v_2 & -v_3 & 0 & \dots & \dots \\ 0 & -v_2 & 2v_3 & -v_4 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -v_{n-2} & 2v_{n-1} & -v_n \\ 0 & \dots & & 0 & -v_{n-1} & 2v_n \end{bmatrix} = \begin{bmatrix} h^2 f_1 \\ h^2 f_2 \\ \dots \\ \dots \\ \dots \\ h^2 f_n \end{bmatrix}.$$

Extracting the discretized v_i from the matrix and defining $d_i = h^2 f_i$: We end up with:

$$\begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \dots \\ \dots \\ \dots \\ d_n \end{bmatrix}.$$

This confirms that the approximation of the second derivative can be expressed as a set of linear equations.

$$\mathbf{Av} = \mathbf{d},$$

Where A is the created matrix, also known as tridiagonal matrix. In this project we will use that $f(x) = 100e^{-10x}$, with the same interval and boundary conditions as given earlier. $x \in (0, 1)$ and $f(0) = f(1) = 0$. Also the analytical solution to the second order differential equation is $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$

ii. LU decomposition

Another interesting way to solve the problem can be done by implementing LU decomposition. By this the matrix A has to be rewritten as a product of L and U. The new matrices enable a fairly simple calculation that solves the problem for v_i.

$$LU = A$$

Where matrix L and U consist of an upper and lower triangular matrix respectively. Now there is another representation of A.

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & & \dots & \dots & 0 \\ L_{21} & 1 & 0 & \dots & \dots & \dots \\ & L_{32} & 1 & 0 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & L_{i-1,j-2} & 1 & 0 \\ L_{i,j} & \dots & \dots & \dots & L_{i,j-1} & 1 \end{bmatrix} \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & & \dots & \dots & U_{1,j} \\ 0 & U_{22} & U_{23} & \dots & \dots & \dots \\ & 0 & U_{33} & U_{34} & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & U_{i-1,j-1} & U_{i-1,j} \\ 0 & \dots & \dots & \dots & 0 & U_{i,j} \end{bmatrix}$$

This is the general expression for L and U matrices. Further on we need to solve for Y:

$$L\mathbf{Y} = \mathbf{d} \tag{5}$$

This linear set of equations is solvable, as our d_1 is known and we can calculate the next steps accordingly. Carrying on we need to find v_i. This can be done by the matrix U, yielding:

$$U\mathbf{v} = \mathbf{Y} \tag{6}$$

As we now have the solutions to Y and the matrix U we can calculate the vector $\mathbf{v}$. This is on the other hand done by backward substitution. Where we know the last element $U_{i,j}$ as well as the last element of $\mathbf{Y}$.

This way of solving the problem was done by using a library with LU-Decomposition called Armadillo [1] [2].

iii. General Tridiagonal Matrix

We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1 : n$.
Our linear equation reads

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \dots \\ \dots \\ \dots \\ d_n \end{bmatrix}.$$

To be able to write our linear equation in terms of one-dimensional vectors we first need to perform a row-reduction.

$$\mathbf{A} = \begin{bmatrix} \tilde{b}_1 & c_1 & 0 & \dots & \dots & \dots \\ 0 & \tilde{b}_2 & c_2 & \dots & \dots & \dots \\ & 0 & \tilde{b}_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & \tilde{b}_{n-1} & c_{n-1} \\ & & & & 0 & \tilde{b}_n \end{bmatrix} \quad \rightarrow \quad \begin{aligned} \tilde{b}_1 &= b_1 \\ \tilde{b}_2 &= b_2 - \frac{a_1}{\tilde{b}_1} c_1 \\ \tilde{b}_3 &= b_3 - \frac{a_2}{\tilde{b}_2} c_2 \\ &\cdot \\ &\cdot \\ &\cdot \\ \tilde{b}_n &= b_n - \frac{a_{n-1}}{\tilde{b}_{n-1}} c_{n-1} \end{aligned}$$

The matrix on the left hand-side simply becomes row-reduced where $\tilde{b}$ is defined as above. With the same row-operations we get an alternation in the right hand-side vector $\mathbf{g}$. $\mathbf{g}$ is translated and $\tilde{\mathbf{d}}$ is defined as:

$$\begin{bmatrix} \tilde{d_1} \\ \tilde{d_2} \\ \dots \\ \dots \\ \dots \\ \tilde{d_n} \end{bmatrix} \quad \rightarrow \quad \begin{aligned} \tilde{d_1} &= d_1 \\ \tilde{d_2} &= d_2 \frac{a_1}{\tilde{b}_1} \tilde{d_1} \\ \tilde{d_3} &= d_3 - \frac{a_2}{\tilde{b}_2} \tilde{d_2} \\ &\cdot \\ &\cdot \\ &\cdot \\ \tilde{d_n} &= d_n - \frac{a_{n-1}}{\tilde{b}_{n-1}} \tilde{d_{n-1}} \end{aligned}$$

Now we are left with a set of linear equations expressed as:

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & \dots & \dots & \dots \\ 0 & \tilde{b}_2 & c_2 & \dots & \dots & \dots \\ & 0 & \tilde{b}_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & \tilde{b}_{n-1} & c_{n-1} \\ & & & & 0 & \tilde{b}_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{d_1} \\ \tilde{d_2} \\ \dots \\ \dots \\ \dots \\ \tilde{d_n} \end{bmatrix}$$

The set of one dimensional vectors can be expressed as following:

$$\tilde{b}_1 v_1 + c_1 v_2 = \tilde{d}_1$$
$$\tilde{b}_2 v_2 + c_2 v_3 = \tilde{d}_2$$
$$\tilde{b}_3 v_3 + c_3 v_4 = \tilde{d}_3$$
$$.$$
$$.$$
$$.$$
$$\tilde{b}_n v_n = \tilde{d}_n$$

This linear set of equations contains everything we need to construct our $\mathbf{v}_i$. Firstly we fill up values for $\tilde{\mathbf{g}}$ and $\tilde{\mathbf{d}}$ as a forward calculation. Secondly we need to look at our new equation to find values for $\mathbf{v}_i$. To solve this we need to use backward substitution, by doing this we end up with the final algorithm:

$$v_{i-1} = \frac{\tilde{d}_{i-1} - c_{i-1} v_i}{\tilde{b}_{i-1}} \tag{7}$$

Where i = n. As we already have a solution for v_n, which is given by:

$$v_n = \frac{\tilde{d}_n}{\tilde{b}_n}$$

iv. Special Tridiagonal Matrix

The second method takes advantage of the special matrix and that its diagonal elements are $b_i = 2$ and the non-diagonal elements are $a_i = c_i = -1$.
This enables us to precalculate the updated diagonal matrix elements $\tilde{b}_i$, such that less FLOPS is necessary throughout our loop.

$$\tilde{b}_i = b_i - \frac{a_{i-1}}{\tilde{b}_{i-1}} c_{i-1} = 2 - \frac{1}{\tilde{b}_{i-1}} = \frac{i+1}{i}$$

while the non-diagonal elements $a_i = c_i$ are unchanged.

Thus the right hand can be written as:

$$\tilde{d}_i = d_i - \frac{a_{i-1}}{\tilde{b}_{i-1}} \tilde{d}_{i-1} = d_i + \frac{(i-1)}{i} \tilde{d}_{i-1}$$

The backward substitution gives then the final solution [3] as:

$$v_{i-1} = \frac{\tilde{d}_{i-1} - c_{i-1}v_i}{\tilde{b}_{i-1}} = \frac{i-1}{i}(\tilde{d}_i + v_i)$$

Where $v_n = \frac{\tilde{d}_n}{\tilde{b}_n}$

v. Number of FLOPS

FLOPS refer to floating point operations and are a measure of computing performance. Every arithmetic operation performed corresponds to one floating point operation. What this means is that when measuring how fast an algorithm is, the amount of FLOPS it uses is being measured.

First we will consider our general and special algorithm. The general algorithm consist of many operations to solve the problem, yielding 9n number of FLOPS. This can be shown by looking at

$$\begin{aligned} \tilde{b}_n &= b_n - \frac{a_{n-1}}{\tilde{b}_{n-1}}c_{n-1} \\ \tilde{d}_n &= d_n - \frac{a_{n-1}}{\tilde{b}_{n-1}}\tilde{d}_{n-1} \\ v_{i-1} &= \frac{\tilde{d}_{i-1} - c_{i-1}v_i}{\tilde{b}_{i-1}} \end{aligned} \tag{8}$$

By counting the number of operations needed to calculate, we find that there are indeed 9N FLOPS needed to run the general algorithm.

Implementing the special algorithm which takes into consideration that the diagonal elements and the right-hand side are precalculated, the number of operations for computing the second derivative v_i can be reduced drastically as these expressions can be calculated beforehand.

$$\tilde{b}_i = \frac{i+1}{i}, \qquad \tilde{d}_i = d_i + \frac{(i-1)}{i}\tilde{d}_{i-1}$$

These expressions are known and are not necessary to be included in the algorithm itself and lead to only the backward substitution being a part of the algorithm for solving v_i values.

$$v_{i-1} = \frac{i-1}{i}(\tilde{d}_i + v_i)$$

This leads to only 4n number of FLOPS for the special algorithm for tridiagonal matrix.

On the other hand, LU-decomposition requires approximately $\frac{2}{3}n^3$ FLOPS [3].

III. Implementation

Programs used in this project can be found on https://github.com/patrykpk/FYS4150/tree/master/Project_1 and a file there named "Reproducibility-Guide.md" explains how to run the scripts. All the plots and results used in this article are also included for files that did not exceed the size limit. All calculations are done in C++, while the plotting is done in Python.
Our C++ program is based on the codes found in the course's github repository https://github.com/CompPhysics/ComputationalPhysics [3].

IV. Numerical results

We have so far given a detailed explanation of our problem and our methods for solving the second order equation. An overview can be seen as a problem where we consider multiple methods, general and special algorithms and LU decomposition for solving the equation. We will here look at these methods and compare them for various n. As we mentioned, the number of FLOPS vastly differs in between these methods, as we will further investigate under this section. We will also consider the set precision for the analytic and numerical solution, seeing where the numerical method fails. The relative error becomes considerably important here, as we will show.

i. CPU time

Checking the CPU time for the general and special algorithm for matrices up to $n = 10^7$, as well as for LU decomposition. Acquiring the values for different methods required running the code several times before deducing at the mean value of these for the processing time.

Table 1 shows the time it takes the code to run through the for loop, for the different methods and is a result of taking mean value of execution time for 10 runs for each value of n. As we can see the special algorithm takes a clear advantage as n is increasing. Another point worth noticing are the results for LU-decomposition for values of n above 10^4 where the Armadillo library was used [1] [2].

Table 1: Execution time for various algorithms over for loops

Execution time [ms]			
n	*General algorithm*	*Special algorithm*	*LU-decomposition*
10	0.005	0.002	1.920
10^2	0.007	0.005	2.814
10^3	0.042	0.028	10.840
10^4	0.545	0.362	976.984
10^5	3.823	2.979	Out Of Memory
10^6	31.812	25.465	Out Of Memory
10^7	320.085	244.056	Out Of Memory

ii. Precision: Analytic and Numerical method

Precision is a vital factor when it comes to numerical analysis. Increasingly decreasing the step-length would by common knowledge increase the precision as well as decrease the relative error. We accommodated that was not the case. For some set of n, we could see that a discrepancy occurred where the error exploded.

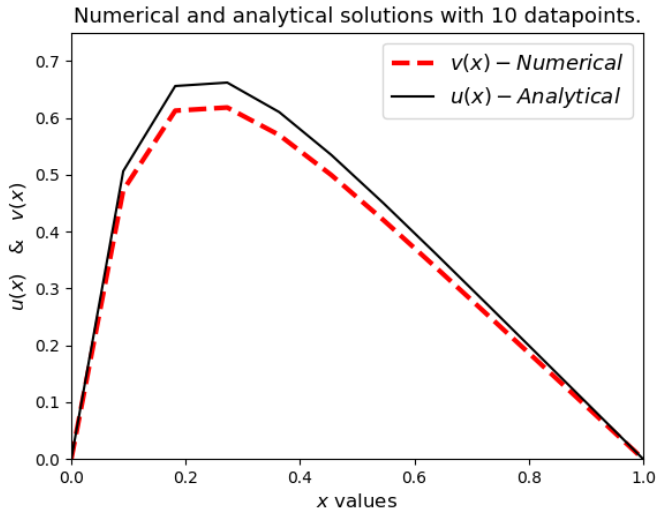

Figure 1: Plot of u(x) & v(x), with n = 10

Figure 1 show a lack of precision for n = 10. Where as mentioned $h = \frac{1}{1+n}$, which gives us a fairly big step-length. The error becomes present as we can see by the plot. It becomes clear that we have a discretized function with a limited number of points.

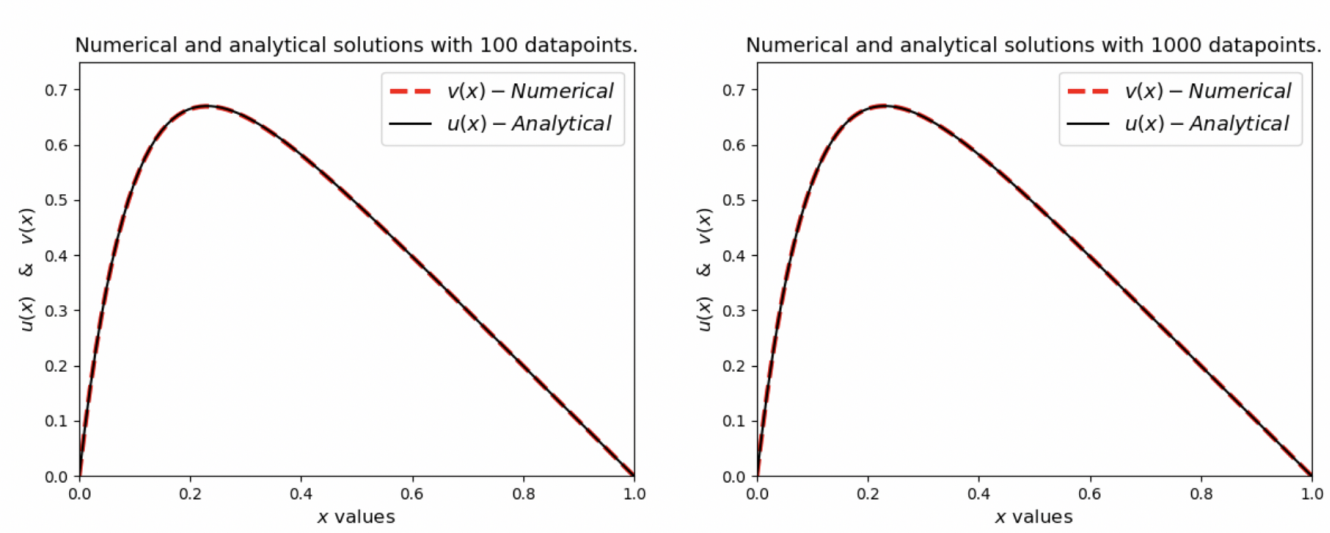

Figure 2: Plot of u(x) & v(x), with n = [100,1000]

Figure 2 is an entirely different case where the analytic and numerical method is overlapping quite satisfyingly. We can see that by only increasing number of points by a factor of ten drastically improved our results. Further on we need to calculate the relative error. We did this for our two main cases, for the general and special algorithm.

As seen from Table 2 the maximum relative error for both cases seems to be identical up to some point while still decreasing. Additionally the maximum error for the general and special case appears to differ from each other with increasing n and the error reaches a threshold where it no longer decreases.

Table 2: Logarithmic values of step-size and the relative error for general and special algorithm.

n	$\log_{10}(h)$	Max(ϵ) general	Max(ϵ) special
10	-1	-1.180	-1.180
100	-2	-3.088	-3.088
10^3	-3	-5.080	-5.080
10^4	-4	-7.079	-7.079
10^5	-5	-8.843	-9.079
10^6	-6	-6.076	-10.164
10^7	-7	-5.525	-9.090

Figure 3 gives a clear description of the relative error when n becomes significantly large. This is as seen in the figure given in a logarithmic scale, yielding n up to 10^7. We can see that there is indeed a discrepancy as n reaches 10^5 for the general algorithm and a tenth order later for the special algorithm. At these values our step length is $h = \frac{1}{1+n}$, which gives a step length in the order of 10^{-7}.

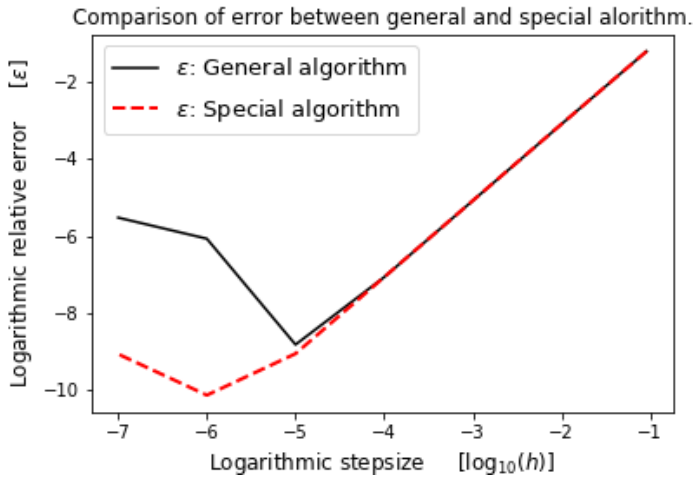

Figure 3: Plot of the logarithmic relative error for n up to 10^7 as a function of logarithmic step-size h

V. Discussion

Table 1 clearly shows that there is a big difference in CPU time between the three different algorithms. This can be explained by the difference in the number of floating point operations, where the special algorithm requires approximately $4n$ FLOPS, the general algorithm needs $9n$ FLOPS, while the LU requires $\frac{2}{3}n^3$ FLOPS.

The results also tells us that the LU cannot run when $n \geq 10^5$. This is because computers are not infinite and can only store finite quantity of information. For a $10^5 \times 10^5$ matrix with double precision, 64 bits or 8 bytes for every matrix element, one needs $10^5 \times 10^5 \times 8 \approx 100$ GB of memory. Therefore, one cannot run the LU-algorithm for matrices with dimensions $n \geq 10^5$ without the needed memory in one's computer.

Figure 1 illustrates that for low n-values our approximation fits terrible with the function $u(x)$. For larger n-values like $n = 100$ and $n = 1000$, illustrated in Figure 2, it is clearly shown that the approximation becomes better. This is also shown in Table 2 and Figure 3 where the log_{10} of the relative error ϵ is plotted against the log_{10} of the step-size h.
And this is true for both the general and the special algorithm.

But when $n \geq 5$ Figure 3 shows clearly that the relative error rises for the general algorithm and for the special algorithm this occurs when $n \geq 6$. Increasing the amount of data points and thus decreasing the step length should in theory lead to a better precision. Based on this the relative error should also decrease, but this is not the case.
The reason for why the error rises for smaller h-values is because the error is being dominated by round-off error due to our equipment. A machine can only represent a certain amount of numbers, so when the values stored in the computer is very low that they cannot be represented, information and precision will be lost.

VI. Conclusion

The second order differential equation has been evaluated in different ways throughout the text. It has been shown that the special algorithm is the way to go, as the number of FLOPS is less than the other algorithms. The CPU time differences is quite marginal for low n for the general and special algorithm. The difference becomes noticeable for $n = 10^4$. The LU decomposition is much

slower as expected, due to the number of FLOPS. By optimizing our algorithm we minimized the CPU time and also pushed the limit for precision for high n as shown in Figure 3. It is also important to be aware of the relative error abnormality as n becomes too high. By implementing our special algorithm to our problem we were able to minimize our relative error giving a better approximation to our problem.

References

[1] Conrad Sanderson and Ryan Curtin. Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation. *Mathematical and Computational Applications*, 24(3):70, 2019.

[2] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(3):26, 2016.

[3] M. Hjorth-Jensen. Computational Physics. *University of Oslo*, `https://github.com/CompPhysics/ComputationalPhysics`, 2013.