FYS3150 - Project 1

Git Repository: <https://github.com/henrikx2/FYS3150>

**Abstract**

This report reveals how ordinary 2nd order differential equations can be solved numerically using a matrix equation. It shows how the matrix equation can be simplified and specialized through algorithms to obtain less experimental errors and faster calculation than i.e. a solution with LU-decomposition or a standard matrix multiplication.

(Summary of work.)

1. **Introduction**

(Aim, what has been done. Summary of structure.)

This experiment aims to solve the one-dimensional Poisson equation with Dirichlet boundary conditions. Poisson’s equation is written as:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

And the one-dimensional equation to solve is in this case expressed as:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

To solve the equation, we will use a numerical approximation to the second derivative and express it as a matrix equation with a tridiagonal matrix. We will then solve the matrix equation generally by rewriting it as three one-dimensional vectors and then specialize it afterwards. In the end we will use LU-decomposition to solve the matrix equation and look at the difference in number of floating points, CPU-time and the relative error as function of iterations .

1. **Methods and theory**
   1. **Discretized values**

To do integration on a computer, we have to work with discretized variables. In this case the main function, , is a function of , and we therefore need to be discrete. In other words;

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

Where is the distance (step size) between each discretized and is the number of -values (mesh points) on the interval . In this explicit case we define and , this means that there is steps from to and that we may express the step size as:

|  |  |  |
| --- | --- | --- |
|  |  | (4) |

Furthermore, we denote the discretized approximation to as and .

* 1. **Second derivative approximation**

To calculate the second derivative numerically, we use an expression which evolves from tweaking two different taylor-expansions. Equation (5) shows the general expression of a taylor-expansion around a point :

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

Now, knowing that is discretized, we can calculate one step forward and one step backward using equation (5).

|  |  |  |
| --- | --- | --- |
|  |  | (6) |
|  |  | (7) | |

We also denote the discrete values as follows:

To find an expression for the 2nd derivative we add equation (6) and (7) together and get equation (8):

|  |  |  |
| --- | --- | --- |
|  |  | (8) |

Equation (8) gives the possibility to calculate the 2nd derivative by knowing the function values at each step . Furthermore; the error in this approximation will run like , which will be explained later from a logarithmic point of view.

* 1. **Tridiagonal matrix equation**

Now, using equation (8) to describe equation (2) gives:

|  |  |  |
| --- | --- | --- |
|  |  | (9) |

Which can be rewritten as:

|  |  |  |
| --- | --- | --- |
|  |  | (10) |

This equation consists of three trailing -values for every used. By the following set of equations it can be showed how Equation (10) looks like when evolves from to :

|  |  |  |
| --- | --- | --- |
|  |  | (11) |

Further, there is the fact that and that in every equation there is one multiplied with 2 and one and one multiplied with -1. This gives rise to a tridiagonal matrix of the type:

Such that the -dimensional equation system can be written as a matrix equation as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (12) |

Where:

If the diagonals in matrix is written as vectors , and , this can be expressed in a general way as:

|  |  |  |
| --- | --- | --- |
|  |  | (13) |

It’s important to note that the endpoints of the vectors are not included in this matrix, because they are already known from the Dirichlet boundary conditions.

* 1. **The analytical solution**

The source term used in this report is , which gives the solution of equation (2) as

|  |  |  |
| --- | --- | --- |
|  |  | (14) |

By derivation this can be shown to be correct:

This function will be used to compare the results of the numerical solutions.

* 1. **The General Algorithm**

To solve the matrix equation (13) the algorithm called the Thomas Algorithm[[1]](#footnote-1) will be used. The Thomas algorithm consists of a decomposition of the matrix into three arrays, a forward substitution and then a backwards substitution. It will look something like this (using Python notation):

|  |
| --- |
| **Forward substitution** |
|  |
| **Backwards substitution** |
|  |

The number of FLOPS required for this algorithm is approximately FLOPS in the forward substitution and FLOPS in the backwards substitution. Which gives a total of FLOPS. There are also 10 memory reads and 3 memory writes. These

* 1. **The Special Algorithm**

In the case when all the elements at each diagonal is the same, the calculations can be simplified to have less FLOPS. In this case, we can use the fact that the arrays and are just . Also, in the forward substitution section, the array can be precalculated. This makes the algorithm (in Python notation):

|  |
| --- |
| **Forward substitution** |
|  |
| **Backwards substitution** |
|  |

Which gives a total number of FLOPS of . The -values in this special algorithm are calculated prior to the for-loop with the algorithm derived by entering the values of and in the expression of . This results in the expression:

For increasing values if we get:

But by examining the answer, we can see that the expression for the -elements can be expressed:

This is a faster way of computing the d-values, rather than doing it recursively inside the for-loop. We would in other words expect this algorithm to run times faster than the general algorithm (based only on FLOPS).

* 1. **LU-Decomposition**

The last way to calculate and analyse the 2nd derivative in this report will be by using the LU-decomposition of the matrix A. The LU-decomposition of a matrix consists of a lower (L) and an upper (U) triangular matrix respectively. We will not be using pivoting, since the matrix A is a non-singular matrix and therefore division by zero is no risk. An LU-decomposition of A can be expressed as:

This way of solving the matrix equation will run with approximately FLOPS. This can be shown by writing out the whole set of equations and counting the FLOPS. So, this will (based on FLOPS) be a much slower way of computing the approximation , than both the general and the special method. However, we might encounter that the reads/writes in the previous algorithms plays a significant role for some values of . The LU-Decomposition method will be performed by the *lu\_factor()* and *lu\_solve()* functions in python.

* 1. **Relative error**

The relative error at the position in the approximation of the 2nd derivative is calculated with the expression:

|  |  |  |
| --- | --- | --- |
|  |  | (15) |

The error used in the error-plot will be the average of all the ’s for each step size . By plotting these values as a log-log-plot (log10) the result should in theory be a linear graph with a slope of 2. This slope comes from the fact that the error in the second derivative approximation is a function of the . This is described in Eq. (8) by .

1. **Results and discussion**
   1. **General algorithm**

Figure 3.1.1 shows the approximation calculated with the general algorithm for values compared to the analytical solution from Eq. (14).

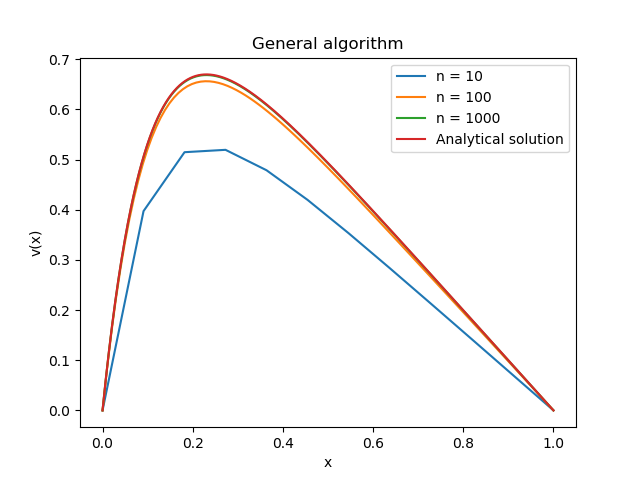


Figure 3.1.1: Solution with general algorithm for different values of n compared to the closed-form solution.

From this, we see that there is an increasing precision with increasing . Already when (green curve), we observe that the calculated curve is hidden underneath the analytical curve.

* 1. **Special algorithm vs. General algorithm**

Table 3.2.1 contains average CPU-time usage in both the general and the special algorithm. This average CPU-time is calculated from Table 5.1.1 and Table 5.1.2.

*Table 3.2.1 – Average CPU-time usage in the General and Special algorithm*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **General algorithm CPU-time** | **Special algorithm CPU-time** | **Speed fraction** |
| **n=10** |  |  |  |
| **n=102** |  |  |  |
| **n=103** |  |  |  |
| **n=104** |  |  |  |
| **n=105** |  |  |  |
| **n=106** |  |  |  |

In 2.5 and 2.6, we expected the special algorithm to be times faster than the general. We see from the speed fraction in Table 3.2.1 that this is not quite correct. The special algorithm is the fastest for every value of , but how much faster varies a lot. The reason for this may be that the program is not run isolated on the CPU. On a computer, the operation system (in this case Windows) will always run other tasks on the CPU simultaneously as we run an algorithm. Depending on the heaviness of these task, the CPU-time of the algorithms may vary.

* 1. **Comparing LU-Decomposition**

The approximation done by LU-Decomposition is compared to the general and the special algorithm in Table 3.3.1. The fastest and slowest algorithms are shown with green and red colours respectively.

*Table 3.3.1 – Average CPU-time usage in the General, Special and LU-Decomposition*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **General algorithm CPU-time** | **Special algorithm CPU-time** | **LU-Decomposition CPU-time** |
| **n=10** |  |  |  |
| **n=102** |  |  |  |
| **n=103** |  |  |  |
| **n=104** |  |  |  |
| **n=105** |  |  | Error |
| **n=106** |  |  | Error |

In section 2.7, we stated that the LU-Decomposition method ran with FLOPS. For high -values, this would imply a much slower calculation than both the general and special algorithm. This seems to be correct as the LU-Decomposition is the slowest one when for all -values.

In the cases of , the program will not calculate the LU-Decomposition method and returns an error. This error is due to the fact that a -matrix contains -elements. Every element in an array, matrix or variable on a computer, uses 8 bytes of random access memory (RAM). The computer must then allocate bytes of RAM. This much more than average people have on their computers.

One remark one could note is the fact that the actual LU-decomposition of the matrix A is included when calculating the speed of the method. This may seem a bit unfair, because in the special algorithm, we allowed the array elements on the diagonal to be precalculated. By inspecting how much time this actually takes, we find (in Table 5.1.2 under Diagonal calculation) that this time (depending on ) is a maximum of 40% of the time when pre calculation is excluded. The significance of the pre calculation does actually become smaller by increasing . In other words; adding the pre calculation time to the time in Table 3.1.1 would not change the main results.

* 1. **Relative error**

The relative error is described by the log-log plot in Figure 3.4.1 and shows how different values of the step size may impact the experimental results.

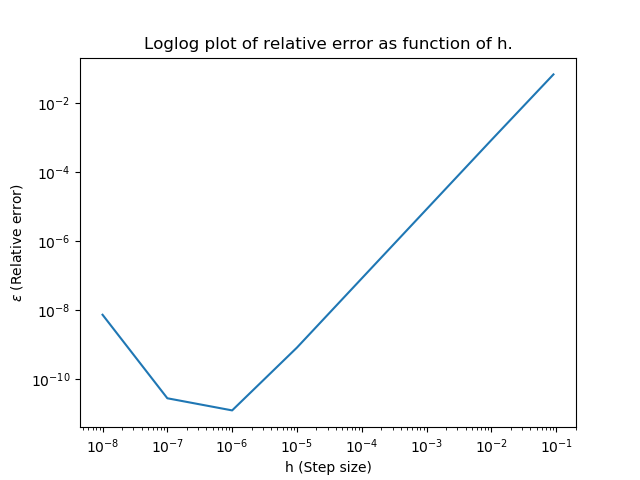


Figure 3.4.1: Log-log plot of the relative error as a function of step size h.

As expected from the last plot (Figure 3.1.1), there is an increasing precision as decreases ( increases) and there is a linear curve in the beginning. But as the error reaches its minimum (around ), the error increases rapidly. There is a loss of numerical precision due to truncation and round-off errors which appear when working with small numbers ( with double precision) on a computer. This is how a rather small step size may make large loss of numerical precision because the error run as .

The slope of the graph is obtained by use of the least square method in python. This gives a slope of , which is the expected value from Equation (8).

1. **Conclusion**
2. **Appendix**

**5.1 CPU-time data**

Table 5.1.1 and 5.1.2 contains 5 sets of CPU-time usage in both the General and the Special algorithm for values of . The CPU-time usage presented in Table 5.1.3 contains CPU-time usage in the LU-Decomposition.

*Table 5.1.1 – CPU-time in General algorithm from 5 different runs.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **General algorithm** | | | | | |
| **Run:** | 1 | 2 | 3 | 4 | 5 |
| **n=10** | 0.0001170 | 0.0001228 | 0.0000609 | 0.0000477 | 0.0000924 |
| **n=102** | 0.0003081 | 0.0002983 | 0.0005668 | 0.0005648 | 0.0005556 |
| **n=103** | 0.0040404 | 0.0038759 | 0.0072516 | 0.0062470 | 0.0068394 |
| **n=104** | 0.0483207 | 0.0326118 | 0.0699969 | 0.0639499 | 0.0618664 |
| **n=105** | 088 | 0.2814706 | 0.6291117 | 0.6299004 | 0.6598148 |
| **n=106** | 3.6926919 | 4.7178563 | 6.1608409 | 6.6048769 | 5.9710214 |

*Table 5.1.2 – CPU-time in Special algorithm from 5 different runs and CPU-time in calculation of the diagonal elements.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Special algorithm** | | | | | | |
| **Run:** | 1 | 2 | 3 | 4 | 5 | Diagonal calculation |
| **n=10** | 0.0000403 | 0.0000433 | 0.0000194 | 0.0000854 | 0.0000234 |  |
| **n=102** | 0.0003088 | 0.0001563 | 0.0002143 | 0.0008692 | 0.0002160 |  |
| **n=103** | 0.0020578 | 0.0018289 | 0.0033968 | 0.0088157 | 0.0038368 |  |
| **n=104** | 0.0204579 | 0.0197546 | 0.0377723 | 0.0693642 | 0.0367184 |  |
| **n=105** | 0.1669740 | 0.2652897 | 0.2802256 | 0.3940636 | 0.1939927 |  |
| **n=106** | 2.25802910 | 3.8945884 | 3.7121895 | 4.0959852 | 2.9244365 |  |

*Table 5.1.3 – CPU-time in LU-Decomposition from 5 different runs.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LU-Decomposition** | | | | | |
| **Run:** | 1 | 2 | 3 | 4 | 5 |
| **n=10** | 0.0002393 | 0.0002920 | 0.0002796 | 0.0002238 | 0.0002291 |
| **n=102** | 0.0020839 | 0.0020134 | 0.0035307 | 0.0018647 | 0.0045986 |
| **n=103** | 0.0257322 | 0.0275712 | 0.0256663 | 0.0243680 | 0.0267002 |
| **n=104** | 10.211161 | 10.085833 | 12.274161 | 10.730455 | 10.679242 |

**References**

1. **Thomas, L.H. (1949)**, *Elliptic Problems in Linear Differential Equations over a Network*, Watson Sci. Comput. Lab Report, Columbia University, New York. [↑](#footnote-ref-1)