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 specialize it afterwards. In the end we will use LU-decomposition to solve the matrix equation and look at the difference number of floating points, CPU-time and relative error.

1. **Methods**
   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 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 .

* 1. **Tridiagonal matrix equation**

Now, by using equation (8) to describe equation (2) we get:

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

Which we rewrite as:

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

This equation consists of three trailing -values for every used. We can show how this will look when evolves from to by the following set of equations:

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

We know 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 we can write the -dimensional equation system as a matrix equation as follows:

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

Where:

If we make the diagonals in matrix vectors , and , we can write the matrix equation in a general way as:

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

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

* 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) we will use the algorithm called the Thomas Algorithm (1). 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** |
|  |

We can see that 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.

* 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 on beforehand with the algorithm derived by entering the values of , in the expression of . This results in the expression:

Which can be rewritten as a function of :

This is a much faster way of computing the d-values, rather than doing it recursively inside the for-loop.

* 1. **LU-Decomposition**

The last way to calculate the 2nd derivative in this case 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.

1. **Results and discussion**
2. **Conclusion**
3. **Appendix**

**References**