FYS3150/4150
Computational Physics
Project 1

Magnus Ulimoen
Krister Stræte Karlsen

September 11, 2015

1 Introduction

A great deal of differential equations in the natural sciences can be written as a linear second-order differential equation on the form

$$\frac{d^y}{dx^2} + k^2(x)y = f(x) \tag{1.1}$$

Some examples include, Schrödinger's equation, the Airy function, Poisson's equation and a special case, Laplace's equation.

Being able to solve such equations optimally with respect to accuracy and efficiency is undoubtedly important. This report will compare different algorithms for solving them.

The algorithms to be investegated in this report are standard Gaussian Elimination, LU decomposition, sparse matrix decomposition and an algorithm specifically tailored for a tridiagonal system of linear equations.

1.1 A closer look at an example from Electromagnetism

In electrostatics Maxwells equations reduces to

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0} \tag{1.2}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{1.3}$$

$$\nabla \times \mathbf{E} = 0 \tag{1.4}$$

$$\nabla \times \mathbf{B} = 0 \tag{1.5}$$

As the curl of the electric field is zero, the field is conservative and can be written on the form

$$\nabla\Phi = -\mathbf{E} \tag{1.6}$$

This gives Poisson's equation

$$\nabla^2\Phi = -\frac{\rho(\mathbf{r})}{\epsilon_0} \tag{1.7}$$

Letting ρ be spherically symmetric leads to a symmetric Φ. This gives with a substitution $\Phi(r) = \phi(r)/r$

$$\frac{d^2\phi}{dr^2} = -\frac{r\rho(r)}{\epsilon_0} \tag{1.8}$$

Which is on the form of (1.1) with $k(x) = 0$, which is also the differential equation which is solved in this paper.

2 Method

The specific differential equation, with boundery values, being discussed in this report is:

$$-u''(x) = f(x), \quad x \in [0,1], \quad u(0) = u(1) = 0. \tag{2.1}$$

This boundery value problem can be discretized and written as a system of linear equations.

Letting the domain $x \in [0,1]$ be discretized into $n+2$ pieces,

$$x_0, x_1, \ldots, x_n, \ldots, x_{n+1} \tag{2.2}$$

where

$$h = \frac{x_{n+1} - x_0}{n+1} = \frac{x_{n+1}}{n+1} \tag{2.3}$$

$$x_i = x_0 + ih = ih \tag{2.4}$$

an then sampling the solution at the mesh points so that $u(x_i) \simeq v_i$.

And using the three-point formula from the symmetric Taylor-expansion for the second derivative of v,

$$\frac{d^2v}{dx^2} \simeq \frac{v_{i-1} + v_{i+1} - 2v_i}{h^2} \tag{2.5}$$

The discretized form of equation (2.1) can then be written as

$$\frac{v_{i-1}+v_{i+1}-2v_i}{h^2}=\tilde{f}_i, \quad i=1,2,\dots,n \tag{2.6}$$

with $\tilde{f}_i = f(r_i)h^2$.

With boundary conditions

$$v_0 = v_{n+1} = 0 \tag{2.7}$$

This can now be written as a system of linear equations on the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}, \tag{2.8}$$

Multipling the discretized equation (2.6) by h^2 we get:

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

Filling in for i and choosing $\tilde{b_i} = h^2 f_i$ we obtain the following set of equations:

$$\begin{aligned}
2v_1 - v_2 &= \tilde{b_1}\\
-v_1 + 2v_2 - v_3 &= \tilde{b_2}\\
&\vdots\\
-v_{i-1} + 2v_i - v_{i+1} &= \tilde{b_i}\\
&\vdots\\
-v_{n-1} + 2v_n &= \tilde{b_n}
\end{aligned}$$

Now one can easily see that this system of linear equations can written on the form of (2.8)

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} v_0 \\ v_1 \\ \dots \\ \dots \\ \dots \\ v_{n-1} \end{pmatrix}, \quad \tilde{\mathbf{b}} = \begin{pmatrix} \tilde{b}_0 \\ \tilde{b}_1 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_{n-1} \end{pmatrix}.$$

Having a complete set of linear equations we move on to how to solve them.

2.1 A tridiagonal matrix algorithm

We start by looking at the system of equations:

$$b_1v_1 + c_1v_2 = \tilde{b_1} \tag{2.9}$$

$$a_2v_1 + b_2v_2 + c_3v_3 = \tilde{b_2} \tag{2.10}$$

$$a_3v_2 + b_3v_3 + c_3v_4 = \tilde{b_3} \tag{2.11}$$

$$\vdots$$

$$a_nv_{n-1} + b_nv_n = \tilde{b_n} \tag{2.12}$$

If we solve (2.9) for v_1 and insert it into (2.10) we obtain the following "modified second equation":

$$(b_1b_2 - a_2c_1)v_2 + b_1c_2v_3 = b_1\tilde{b_2} - a_2\tilde{b_1}$$

Now having successfully removed v_1 from the second equation we can go on and solve it for v_2 and insert it into the third equation obtaining:

$$\begin{aligned}(b_3(b_1b_2 - a_2c_1) - a_3b_1c_2)v_3 + c_3(b_1b_2 - a_2c_1)v_4 = \\ (b_1b_2 - a_2c_1)\tilde{b_3} - a_3b_1\tilde{b_2} + a_2a_3\tilde{b_1}\end{aligned}$$

The two modified equations may be written as

$$v_2 = \frac{b_1\tilde{b_2} - a_2\tilde{b_1}}{b_1b_2 - a_2c_1} - \frac{b_1c_2}{b_1b_2 - a_2c_1}v_3 = \beta_2 + \gamma_2v_3$$

$$\begin{aligned}v_3 =& \frac{(b_1b_2 - a_2c_1)\tilde{b_3} - a_3(b_1\tilde{b_2} - a_2\tilde{b_1})}{b_3(b_1b_2 - a_2c_1) - a_3b_1c_2} - \frac{c_3(b_1b_2 - a_2c_1)}{b_3(b_1b_2 - a_2c_1) - a_3b_1c_2}v_4 \\ =& \frac{\tilde{b_3} - a_3\beta_2}{a_3\gamma_2 + b_3} + \frac{-c_3}{a_3\gamma_2 + b_3}v_4 = \beta_3 + \gamma_3v_4\end{aligned}$$

This prossess can be repeated up untill the last equation. This is the forward substitution step. From the last equation we compute v_n and get all we need to compute v_{n-1}, then v_{n-2}, and so on. This is the backward substitution part of the algorithm. A shrewd reader might see that the coefficiants, β and γ, take a recursive form

$$\beta_i = \frac{\tilde{b_i} - a_i\beta_{i-1}}{a_i\gamma_{i-1} + b_i}, \qquad \gamma_i = \frac{-c_i}{a_i\gamma_{i-1} + b_i},$$

and the equation for v_{i-1} reads:

$$v_{i-1} = \beta_{i-1} + \gamma_{i-1}v_i \tag{2.13}$$

It follows from (2.9) that $\beta_1 = \frac{\tilde{b_1}}{b_1}$ and $\gamma_1 = \frac{-c_1}{b_1}$. From combining (2.12) and (2.13) we get

$$v_n = \frac{\tilde{b_n} - a_n \beta_{n-1}}{a_n \gamma_{n-1} + b_n} = \beta_n.$$

Having all the nessecary ingredients the algorithm reads as follows.

Algorithm I

```
$a_i = c_i = -1, \quad i = 1, 2, 3, .., n$
$b_i = 2, \quad i = 1, 2, 3, .., n$
$\tilde{b_i} = h^2 f_i \quad i = 1, 2, 3, .., n$
$\beta_1 = \frac{\tilde{b_1}}{b_1}$
$\gamma_1 = \frac{-c_1}{b_1}$
for $i = 2, 3, .., n$
    $\beta_i = \frac{\tilde{b_{i-1}} - a_{i-1}\beta_{i-1}}{a_{i-1}\gamma_{i-1} + b_{i-1}}, \quad \gamma_i = \frac{-c_{i-1}}{a_{i-1}\gamma_{i-1} + b_{i-1}}$

$v_n = \beta_n$
for $i = n, n-1, \cdots, 2$
    $v_{i-1} = \beta_{i-1} + \gamma_{i-1} v_i$
```

This is often refered to as *Thomas Algorithm*, or *TDMA*, an algorithm for solving tridiagonal systems of linear equations.

2.1.1 FLOPS

This algorithm requires nine operations for each row for the forward substitution, and two backwards. This requires 11 FLOPS for each row to solve the system. With n rows this algorithm requires 11N FLOPS.

In big O notation the complexity of $\mathcal{O}(n)$.

2.1.2 Implementation

The algorithm in section 2.1 is implemented in C++. Since we do not have changing a_i we have coded $a_i = a$. The zero-indexing in C++ is used, and the elements accessed in the code is different from the algorithm.

2.2 Standard Gaussian Elimination

The Gaussian elimination is solving the system (2.8) by using row operations and getting a row echelon form and then backsubstituting.

This operation in principle is here shown for matrix A.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & y_1 \\ a_{21} & a_{22} & a_{23} & \cdots & y_2 \\ a_{31} & a_{32} & a_{33} & \cdots & y_3 \\ \cdots & & & & \\ a_{n1} & a_{n2} & \cdots & & y_n \end{pmatrix} \tag{2.14}$$

To get the matrix into the row echelon the first row needs to have a non-zero value as its first element. Then a constant $c_i = a_{i1}/a_{11}$ is calculated, and a subtraction of the first row,

$$a_{ij} = a_{ij} - c_i a_{ij} \tag{2.15}$$

is done. For the elements a_{i1}, this gives zero, and the matrix is row reduced to

$$A \sim \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ 0 & a_{22} - c_2 a_{22} & a_{23} - c_2 a_{23} & \cdots \\ 0 & a_{32} - c_3 a_{32} & a_{33} - c_3 a_{33} & \cdots \\ 0 & \cdots & & \\ 0 & a_{n2} - c_2 a_{n2} & \cdots & \end{pmatrix} \tag{2.16}$$

The operation can be repeated on the submatrix $A_{(2-N)(2-N)}$ until the decired triangular matrix is achieved.

The backsubstitution is simple when a row reduced matrix is achieved, by examining the matrix equation $\mathbf{Ux} = \tilde{\mathbf{y}}$

$$\begin{pmatrix} u_{11} & u_{12} & \cdots & \\ 0 & u_{22} & u_{23} & \cdots \\ & \cdots & \cdots & \\ 0 & \cdots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \cdots \\ \tilde{y}_{n-1} \\ \tilde{y}_n \end{pmatrix} \tag{2.17}$$

$$x_n = \frac{\tilde{y}_n}{a_{nn}} \tag{2.18}$$

$$x_{n-1} = \frac{\tilde{y}_{n-1} - a_{(n-1)n} x_n}{a_{(n-1)(n-1)}} \tag{2.19}$$

Where x_i is found from a recursive method extended from this.

2.2.1 FLOPS

For a general matrix with size $(N \times N)$, the algorithm requires N^2 operations per row reduction. This is repeated for all N rows, and the complexity is $\mathcal{O}\left(n^3\right)$ for the reduction. The backwards substitution has $\mathcal{O}\left(n^2\right)$ and is not of significance.

2.2.2 Implementation

The matrix solvers are used from the Armadillo library[1]. The matrix is created by indexing the non-zero elements and setting this equal to the tridiagonal elements. The calculation is done by a call to

Which returns the resulting vector.

2.3 LU decomposition

The matrix **A** can be decomposited into an upper triangular matrix **U** and a lower triangular matrix **L**. The elements on the diagonal of **L** are 1, and it is invertible.

By decomposition the result is

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{w} \tag{2.20}$$

Which is then solved for x by using a the inverse of L

$$\mathbf{Ux} = \mathbf{L^{-1}w} = \mathbf{y} \tag{2.21}$$

Since U is an upper triangular matrix the solution is found from back-substitution of the elements as described in the Gaussian Elimination.

2.3.1 FLOPS

The decomposition into the LU-matrix requires $\mathcal{O}\left(N^3\right)$ FLOPS, but has the advantage that backwards substitution just requires $\mathcal{O}\left(N^2\right)$ FLOPS to compute. This decomposition is also highly reusable if (2.8) is to be solved for more than one **x**, or the determinant is to be found.

[1] `http://arma.sourceforge.net/`

2.3.2 Implementation

The LU decomposition is achieved with the Armadillo library, by calling the lu()-method. This produces three matrices, $\mathbf{L}$, $\mathbf{U}$ and a transformation matrix $\mathbf{P}$, such that

$$\mathbf{A} = \mathbf{P}^\top \mathbf{L}\mathbf{U} \tag{2.22}$$

The vector y is then created by

$$\mathbf{y} = \left(\mathbf{P}^\top \mathbf{L}\right)^{-1} \tag{2.23}$$

Then the solver function in the Armadillo library is used.

2.4 Method using sparse

To prevent saving a very sparse matrix in memory a sparse method can be employed. This saves the non-zero matrix elements and location instead of the entire matrix. This saves memory and computational speed, as only these elements are manipulated.

2.4.1 FLOPS

For a sparse matrix (with suitable algorithms) the number of operations are considerably less than for the Gaussian reduction when the matrices are sparse. The amount of computations is very dependent upon the nature of the matrix.

2.4.2 Implementation

For our very sparse matrix which has $N - 3$ unused elements per row, the memory-space saved is $(N - 3)^2$. This helps keep the matrix in the CPU-cache and speeds it up for larger matrices. The implementation follows from the non-sparse solver, but a sparse matrix has to be constructed and a sparse matrix solver must be used.

The calculation is done by a call to

3 Results

3.1 Speed/Efficiency

The speed is measured in the program by counting the number of clock cycles that is executed between the start of the algorithm and the end of it. This is so divided by one million, to simulate a ”standard” computer. This is not a perfect measure, as random usage of the computer can divert CPU-time to other processes while the time is being measured.

A computer being run with no other processes and sequentially running through the parameters has produced a table for the computational speed of the algorithms[2].

The CPU-time and the number of mesh points are included in table 1, and plotted in figure 1. When the number of meshpoints gets bigger, certain properties of the algorithms can be seen. With log plots the first derivative gives the polynomial complexity. This shows that TDMA follows roughly $\mathcal{O}\left(n\right)$, LU-decomp $\approx \mathcal{O}\left(n^3\right)$, Gaussian $\approx \mathcal{O}\left(n^3\right)$ and the sparse $\approx \mathcal{O}\left(n^{(5/2)}\right)$.

It is worth noting that the LU-decomp takes a considerable amount of time more than the Gaussian, as the solver for the Gaussian is more effective. The LU-decomp has the advantage that it can be used for more vectors easily.

3.2 Accuracy/Error

$$err = \max\left(\log_{10}\left(\max\left[\frac{v_i - u_i}{u_i}\right]\right)\right) \tag{3.1}$$

The error is computed using (3.1).

A table and figure of the error is included in table 2 and plotted in figur 2. The error is very similar, and is simply overplotted. This is saying that the methods give the same results, and are using the same procedures.

[2]For reproduction use the script `compare_algs.py`

Table 1: Comparing the time, number of clock cycles per MHz for the different algorithms for different number of mesh points. TDMA: TriDiagonal Matrix algorithm

	Gauss Elimination	TDMA.	Sparse decompostion	LU decomposition
N	Time	Time	Time	Time
10	4.42000000e-04	2.50000000e-05	5.32000000e-04	1.86820000e-02
25	5.09000000e-03	3.40000000e-05	8.06000000e-04	5.11400000e-03
50	5.94300000e-02	7.70000000e-05	8.18000000e-04	1.01090000e-02
100	3.55500000e-03	4.80000000e-05	1.14700000e-03	1.07380000e-02
250	1.72380000e-02	1.19000000e-04	2.66800000e-03	8.30680000e-02
500	2.83505000e-01	2.08000000e-04	8.02000000e-03	2.14431000e-01
1000	3.55766000e-01	3.66000000e-04	1.68560000e-02	1.31517900e+00
2000	1.75258000e+00	9.27000000e-04	7.37350000e-02	9.11888400e+00
3000	4.48197300e+00	1.14700000e-03	2.08975000e-01	2.39678190e+01
4000	1.01286820e+01	1.87500000e-03	2.66886000e-01	5.86866600e+01
5000	1.93415310e+01	2.37900000e-03	3.97806000e-01	1.08909192e+02
6000	3.20998800e+01	2.22900000e-03	5.35781000e-01	1.89079185e+02

`Results/n_time.png`

Figure 1: Time taken for the different algorithms plotted against the number of mesh points

4 Discussion

5 Concluding remarks

We see that the algorithm developed in section 2.1 is well suited for this problem.

Table 2: Comparing accuray of different algorithms for a different number of mesh points. The error here is defined as log_{10} of the maximum relative error.

	Gauss Elimination	TDMA.	Sparse decompostion	LU decomposition
N	Error	Error	Error	Error
10	0.34469208	0.34469208	0.34469208	0.34469208
25	0.14844317	0.14844317	0.14844317	0.14844317
50	0.07648232	0.07648232	0.07648232	0.07648232
100	0.0388765	0.0388765	0.0388765	0.0388765
250	0.01571379	0.01571379	0.01571379	0.01571379
500	0.00788507	0.00788507	0.00788507	0.00788507
1000	0.00394968	0.00394968	0.00394968	0.00394968
2000	0.00197664	0.00197664	0.00197664	0.00197664
3000	0.00131816	0.00131816	0.00131816	0.00131816
4000	0.00098877	0.00098877	0.00098877	0.00098877
5000	0.00079109	0.00079109	0.00079109	0.00079109
6000	0.00065928	0.00065928	0.00065928	0.00065928

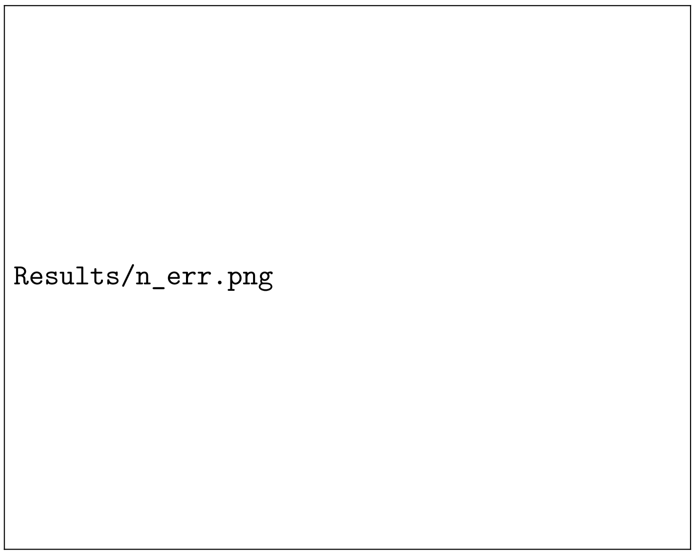

Figure 2: Relative error for the different algorithms plotted againt the number of meshpoints