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

I Schrödinger's equation

II Airy function

III Economics?

IV Poisson's equation

V Laplace's equation (Poisson's with f=0)

So 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 is standard Gaussian Elemination, LU decomposition, sparse matrix decomposition and an algorithem specifically tailored for this problem. 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).

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 $n+1$ pieces,

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

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

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

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

$$\mathbf{Av} = \tilde{\mathbf{b}}, \tag{2.8}$$

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

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

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)

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix}, \quad \tilde{\mathbf{b}} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}.$$

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:

$$b_1 v_1 + c_1 v_2 = \tilde{b}_1 \tag{2.9}$$

$$a_2 v_1 + b_2 v_2 + c_3 v_3 = \tilde{b}_2 \tag{2.10}$$

$$a_3 v_2 + b_3 v_3 + c_3 v_4 = \tilde{b}_3 \tag{2.11}$$

$$\vdots$$

$$a_n v_{n-1} + b_n v_n = \tilde{b}_n \tag{2.12}$$

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

$$(b_1 b_2 - a_2 c_1) v_2 + b_1 c_2 v_3 = b_1 \tilde{b}_2 - a_2 \tilde{b}_1$$

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:

$$\begin{aligned} (b_3(b_1 b_2 - a_2 c_1) - a_3 b_1 c_2) v_3 + c_3(b_1 b_2 - a_2 c_1) v_4 = \\ (b_1 b_2 - a_2 c_1)\tilde{b}_3 - a_3 b_1 \tilde{b}_2 + a_2 a_3 \tilde{b}_1 \end{aligned}$$

The two modified equations may be written as

$$v_2 = \frac{b_1 \tilde{b}_2 - a_2 \tilde{b}_1}{b_1 b_2 - a_2 c_1} - \frac{b_1 c_2}{b_1 b_2 - a_2 c_1} v_3 = \beta_2 + \gamma_2 v_3$$

$$\begin{aligned} v_3 = & \frac{(b_1 b_2 - a_2 c_1)\tilde{b}_3 - a_3 (b_1 \tilde{b}_2 - a_2 \tilde{b}_1)}{b_3 (b_1 b_2 - a_2 c_1) - a_3 b_1 c_2} - \frac{c_3 (b_1 b_2 - a_2 c_1)}{b_3 (b_1 b_2 - a_2 c_1) - a_3 b_1 c_2} v_4 \\ = & \frac{\tilde{b}_3 - a_3 \beta_2}{a_3 \gamma_2 + b_3} + \frac{-c_3}{a_3 \gamma_2 + b_3} v_4 = \beta_3 + \gamma_3 v_4 \end{aligned}$$

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 $i = 2, 3, .., n$
    $\beta_i = \frac{\tilde{b_{i-1}} - a_{i-1}\beta_{i-1}}{a_{i-1}\gamma_{i-1} + b_{i-1}}, \qquad \gamma_i = \frac{-c_{i-1}}{a_{i-1}\gamma_{i-1} + b_{i-1}}$

$v_n = \beta_n$
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*, an algorithm for solving tridiagonal systems of linear equations.

2.1.1 Implementation

The algorithm in section 2.1 is implemented in C++. This requires three arrays to be created, which holds β_i, γ_i and $\tilde{f_i}$. These arrays are zero-indexed, and the implementation follows this.

Our implementation assumes constant a, b, c, and the elements a_i used in the algorithm is not implemented in the code. This is easily changed for more general purposes.

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.

A matrix

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

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 looking at the matrix equation $Ux = y$

$$\begin{pmatrix} u_{11} & u_{12} & \cdots & \\ 0 & u_{22} & u_{23} & \cdots \\ & \cdots & \cdots & \\ 0 & \cdots & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_{n-1} \\ y_n \end{pmatrix} \tag{2.17}$$

$$x_n = \frac{y_n}{a_{nn}} \tag{2.18}$$

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

Where x_i is found from a recursive method.

2.2.1 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

```
arma::Col<double> x = solve(A, b);
```

2.3 LU decomposition

When the matrix equation (2.8) is solved, this is modified into an upper triangular and lower triangular matrix. The lower matrix (L) diagonal has elements with the value 1, and is invertible if A is non-singular.

The LU decomposition allows one to find the sets of linear equations,

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

Is then solved for x by using a the inverse of L (which exist due to its construction),

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

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 Implementation

KSK: Implementation of LU to be added here.

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 Implementation

The matrix solution above is very ineffective in both the amount of computations and memory usage. This is improved by using sparse matrices, which saves the non-zero elements by position and value.

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

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 calculation is done by a call to

```
arma::Col<double> x = spsolve(spA, b);
```

3 Results

3.1 Speed/Efficiency

Thomas alg.		Gaussian el.		Sparse		LU decomp.	
N	Time	N	Time	N	Time	N	Time
10	Insert	10	Insert	10	Insert	10	Insert
100	Insert	100	Insert	100	Insert	100	Insert
1000	Insert	1000	Insert	1000	Insert	1000	Insert
10000	Insert	10000	Insert	10000	Insert	10000	Insert

3.2 Accuracy/Error

Thomas alg.		Gaussian el.		Sparse		LU decomp.	
N	LMRE.	N	LMRE.	N	LMRE.	N	LMRE.
10	Insert	10	Insert	10	Insert	10	Insert
100	Insert	100	Insert	100	Insert	100	Insert
1000	Insert	1000	Insert	1000	Insert	1000	Insert
10000	Insert	10000	Insert	10000	Insert	10000	Insert

3.3 FLOPS

The number of floating point operations (FLOPS) per algorithm gives a useful metric of the scalability. The more FLOPS required the more time is required for larger systems. This is usually given in $\mathcal{O}(x)$ notation, where x gives approximately the number of calculations.

3.3.1 Thomas algorithm

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 $\mathcal{O}(n)$.

3.3.2 Gaussian decomposition

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.

For a sparse matrix (with suitable algorithms) this could be less, as some of the row operations are skipped. This leads to a lower amount of FLOPS needed for a sparse matrix.

3.3.3 LU decomposition

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.

4 Discussion

Can solve the equation (2.8) directly by LU-decomposition. The matrix A is then transformed into one upper triagonal matrix and one lower diagonal matrix. By doing this on a matrix this results in an easier way to solve several $Ax = y$ for different y.

Direct solver by arma::solve(A, y) is used to compare with the LU-method. This is ordinary quicker, since armadillo is a high-level wrapper around lapack etc.

4.1 FLOPS

The standard is $\mathcal{O}\left(N^3\right)$ complexity

5 Concluding remarks

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