FYS3150/4150
Computational Physics
Project 1

Magnus Ulimoen
Krister Stræte Karlsen

September 9, 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.

KSK: More specific info on which algorithms to be investigated.

1.1 A closer look at an example from Electromagnetism

The electrostatic potential Φ is generated from a charge distribution ρ. This gives

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}) \tag{1.2}$$

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

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r) \tag{1.3}$$

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

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

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

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

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)

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

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:

$$b_1v_1 + c_1v_2 = \tilde{b_1} \tag{2.9}$$
$$a_2v_1 + b_2v_2 + c_3v_3 = \tilde{b_2} \tag{2.10}$$
$$a_3v_2 + b_3v_3 + c_3v_4 = \tilde{b_3} \tag{2.11}$$
$$\vdots$$
$$a_nv_{n-1} + b_nv_n = \tilde{b_n} \tag{2.12}$$

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

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

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

KSK: More on implementation needed?

2.2 Standard Gaussian Elimination

KSK: A short description should be added here.

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,

$$Ax = LUx = w \tag{2.14}$$

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

$$Ux = L^{-1}w = y \tag{2.15}$$

Since U is an upper triangular matrix the solution is found from back-substitution of the elements,

$$\begin{pmatrix} u_{11} & u_{12} & \dots & \\ 0 & u_{22} & u_{23} & \dots \\ & \dots & \dots & \\ \dots & 0 & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & \dots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_{n-1} \\ y_n \end{pmatrix} \tag{2.16}$$

$$x_n = \frac{y_n}{a_{nn}} \tag{2.17}$$

$$x_{n-1} = \frac{y_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}} \tag{2.18}$$

And a recursive formula which takes $\mathcal{O}\left(N^2\right)$ calculations.

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.

3 Results

3.1 Speed/Efficiency

3.2 Accuracy/Error

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.

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, and the algorithm has a complexity of $\mathcal{O}(N)$.

3.3.2 Gaussian decomposition

The most general algorithm has a complexity of $\mathcal{O}(N^3)$.

3.3.3 LU decomposition

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

5 Implementation

5.1 Thomas algorithm

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.

5.2 Matrix

The matrix solvers are used from the Armadillo library[1]. This is a wrapper around lapack, and is using methods to reduce the number of FLOPS required.

5.3 Sparse matrix

The matrix solution above is very ineffective in both FLOPS and memory usage. This is improved by using sparse matrices, which saves the position and the value, and assumes everything else is zero, which saves a lot of space in the memory for our purpose ($(N-3)^2$ memory space saved).

6 Concluding remarks

Comparing the different algorithms for different h and against each other

Compare relative error. What is rel-error compared to FLOPS?

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