FYS3150 - Project 1

Daniel Heinesen, Halvard Sutterud, Gunnar Lange

September 25, 2016

Abstract

In this project we will study speed and numerical precision of linear algebra algorithms. Given a tridiagonal matrix, we used both a general solving algorithm for tridiagonal matrices and a special tailored 'ferrari' method for our specific problem. This is also compared to a cumbersome LU-decomposition of the matrix.

Using an analytic expression for the diagonal elements in our special case, we were able to reduce the total number of FLOPS down to a total of (INSERT N) computations, where N is the number of row and column elements in our matrix.

Contents

1 Introduction

We solve the one-dimensional Poisson equation with Dirichlet boundary conditions by reducing it to a set of equations on the form of a tridiagonal matrix.

2 Theoretical model

Discretizing the Poisson equation

Assume that $u(x)$ is a continuous, twice differentiable function, $u(x) \in \mathbb{C}^2$. Using Taylor polynomial to the second degree, a general approximation formula to the second derivative can be derived as:

$$\frac{d^2u}{dx^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + O(h^2)$$

Discretizing $u(x)$ at points $x_1, x_2, ..., x_n$, and introducing the convenient notation $u(x_i) = u_i$, this formula can be rewritten in discrete form as:

$$\frac{d^2u}{dx^2} \approx \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + O(h^2)$$

Where h is the distance between the grid points, given by $h = 1/(n+1)$. If one additionally imposes Dirichlet boundary conditions, i.e. that $u_0 = u_{n+1} = 0$, this formula is valid for all x_i, $i \in [1, n]$. Inserting this formula into the Poisson equation with right-hand side $f(x)$, and letting $f(x_i) = f_i$, gives:

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i \tag{1}$$

This can be rephrased as a linear algebra problem, of the form:

$$\mathbf{Au} = h^2\mathbf{f}$$

Where $\mathbf{A}$ is a $n \times n$ given by:

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots \\ 0 & 0 & -1 & 2 & -1 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & -1 & 2 \end{pmatrix}$$

Notice that $\mathbf{A}$ is a tridiagonal matrix. This makes the general solution algorithm much simpler.

Solving a tridiagonal matrix problem

A general tridiagonal matrix problem can be written as:

$$\begin{pmatrix} a_1 & b_1 & 0 & \dots & \dots & 0 \\ c_1 & a_2 & b_2 & 0 & \dots & 0 \\ 0 & c_2 & a_3 & b_3 & 0 & \dots \\ 0 & 0 & c_3 & a_4 & b_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & c_{n-1} & a_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_n \end{pmatrix}$$

Where, in the specific case above, $v_n = f_n/h^2$. This problem may be solved in two steps: a decomposition and forward substitution, and a backward substitution. The goal is to make each column a pivot column. For the forward substitution, it is easiest to first transform the matrix into an upper-diagonal matrix. Thus, c_1 needs to be zero. This can be achieved by subtracting c_1/a_1 times the first row from the second row. This gives:

$$\begin{pmatrix} a_1 & b_1 & 0 & \dots & \dots & 0 \\ 0 & \tilde{a_2} & b_2 & 0 & \dots & 0 \\ 0 & c_2 & a_3 & b_3 & 0 & \dots \\ 0 & 0 & c_3 & a_4 & b_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & c_{n-1} & a_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} v_1 \\ \tilde{v_2} \\ v_3 \\ v_4 \\ \vdots \\ v_n \end{pmatrix}$$

Where:

$$\tilde{a_2} = a_2 - \frac{c_1}{a_1}b_1, \quad \tilde{v_2} = v_2 - \frac{c_1}{a_1}v_1$$

Notice how only a and v changes in this step. Thus all parameters will be the same in the next time step, but the a and v from the previous time step, will now be $\tilde{a}$ and $\tilde{v}$. It is therefore easy to generalize the above formulae to:

$$\tilde{a}_{i+1} = a_{i+1} - \frac{c_i b_i}{\tilde{a_i}} \tag{2}$$

$$\tilde{v}_{i+1} = v_{i+1} - \frac{c_i}{\tilde{a_i}} \tilde{v_i} \tag{3}$$

These equations can now be iterated from $i = 1$ to $n-1$. We then end up with an upper triangular matrix, containing $\tilde{a_i}$ and b_i, given by:

$$\begin{pmatrix} a_1 & b_1 & 0 & \dots & \dots & 0 \\ 0 & \tilde{a}_2 & b_2 & 0 & \dots & 0 \\ 0 & 0 & \tilde{a}_3 & b_3 & 0 & \dots \\ 0 & 0 & 0 & \tilde{a}_4 & b_4 & \dots \\ \vdots & & & & \ddots & \vdots \\ 0 & & \dots & & 0 & \tilde{a}_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} v_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \\ \vdots \\ \tilde{v}_n \end{pmatrix}$$

Now we need a 1 in the last entry of the matrix. This is easily achieved by dividing by $\tilde{a}_n$. The other pivot elements can be obtained by backwards substitution, by employing the following algorithm.

- Subtracting b_i times the $i+1$th row from the ith row (because the $i+1$th row will have a 1 as a pivot).
- Divide the ith row by $\tilde{a_i}$

This will impact the $\tilde{v}_i$, converting them into w_i, according to:

$$w_i = \frac{\tilde{v}_i - b_i w_{i+1}}{\tilde{a_i}} \tag{4}$$

This defines a general algorithm for solving a tridiagonal matrix problem. The efficiency of this algorithm is surprisingly easy to calculate, as shown in the next section.

Counting the number of FLOPS

The general algorithm consists of equations 2, 3 and 4. Let us look at the required FLOPS for each of these equations:

Equation 2 This equation requires the computation of $x = c_i/\tilde{a}_i$. After this, we compute $y = x \cdot b_i$, and finally the computation $z = a_{i+1} - y$. This is done for $n-1$ points (as it is not required for the first row), giving a total of $3(n-1)$ FLOPS for this equation.

Equation 3 This equation requires the computation of $x = c_i/\tilde{a}_i$, but this has already been computed in the previous step, and therefore does not require additional FLOPS. Then, we need to compute $y = x\tilde{v_i}$ and finally $z = v_{i+1} - y$. This therefore requires a total of $2(n-1)$ FLOPS.

Equation 4 This equation requires the computation of $x = b_i w_{i+1}$, followed by the computation of $y = \tilde{v}_i - x$, and finally the computation $z = y/\tilde{a}_i$. This gives a total of $3(n-1)$ FLOPS.

Total Thus totally, the algorithm requires $8(n-1)$ FLOPS. However, we have ignored the division by $\tilde{a_n}$ to get w_n. This requires one additional FLOP, giving the total number of FLOPS as:

$$\text{FLOPS} = 8n - 7$$

Tailoring an algorithm

In the previous sections, we have developed and analyzed an algorithm for a general tridiagonal matrix. However, as seen from equation 1, the matrix in our case is significantly simpler. This comes from the fact that there are only three distinct numbers present in our matrix. This means that *all quantities* which only require a_i, b_i or c_i can be computed a priori. It is also possible to obtain an analytic expression for $\tilde{a}_i$, as follows.

Finding an analytical expression for $\tilde{a}_i$: The general formula for $\tilde{a}_{i+1}$ is given by equation 2. In our case, $c_i = b_i = -1$ and $a_{i+1} = 2$. Thus equation 2 can be rewritten as:

$$\tilde{a}_{i+1} = 2 + \frac{1}{\tilde{a}_i} \tag{5}$$

Note that $\tilde{a}_1 = a_1 = 2$. Trying out some iterations, it seems that:

$$\tilde{a}_{i+1} = -\frac{(i+2)}{i+1} \tag{6}$$

This is proved in appendix **??**. Thus, $\tilde{a}_i$ can also be pre-computed. This means that the number of FLOPS during runtime can be significantly reduced.

Countinger the number of FLOPS in the tailored algorithm

Equation 2 This can be pre-computed, and therefore does not contribute any FLOPS during runtime.

Equation 3 As $\tilde{a}_i$ is already pre-computed, we can also pre-computed $\tilde{a}_i^{-1}$. Thus this equation only requires the multiplication of $x = \tilde{v_i}\frac{c_i}{\tilde{a}_i}$, followed by the subtraction $y = v_{i+1} - x$, for a total of $2(n-1)$ FLOPS.

Equation 4 Here we only need to compute $x = \tilde{v}_i + w_{i+1}$ (since $b_i = -1$), followed by the computation $x/\tilde{a}_i$ for a total of $2(n-1)$ FLOPS.

Total This gives a total of $4n - 4$ FLOPS. However, we still need to compute $\tilde{v}_n/a_n$ as well, which gives on additional FLOP, for a total of:

$$\text{FLOPS} = 4n - 3$$

Which approximately cuts the running time by a factor 2.

3 Method - Implementation

All programs and benchmarks calculations can be found on our GIT repository[1].

4 Results

4.1 The general algorithm

The time taken for different grid points is shown below:

[1] https://github.com/dulte/Comp-Phys/tree/master/Project1

Number of points	Time taken (seconds) general	Time taken (seconds) tailored
10^1	$2 \cdot 10^{-6}$	$2.0 \cdot 10^{-6}$
10^2	$7 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$
10^3	$6.6 \cdot 10^{-5}$	$4.1 \cdot 10^{-5}$
10^4	$6.7 \cdot 10^{-4}$	$2.6 \cdot 10^{-4}$
10^5	$6.6 \cdot 10^{-3}$	$3.0 \cdot 10^{-3}$
10^6	$5.9 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$
10^7	$6.6 \cdot 10^{-1}$	$2.7 \cdot 10^{-1}$

4.2 The tailored algorithm

4.3 LU-Decomposition

5 Conclusion

6 Reference