Project 1: Vector and Matrix Operations in c++, Fys 4150

Ask J. Markestad, Thorbjørn V. Larsen, Ingrid A. V. Holm

Abstract

We study the Gaussian elimination method for solving a linearized second order inhomogeneous differential equation over an interval with Dirichlet boundary conditions, using both a general Gaussian elimination method for tridiagonal matrix and one tailored for our three point approximation to the second derivative and compare their speed. We also use preexisting packages, armadillo, to solve the same problem by LU-decomposition to show the difference in calculation time between a general, always applicable method and our problem specific method.

Introduction

Differential equations are fundamental in physics. The standard approach in most branches of physics is to describe a physical system in terms of it's symmetries and degrees of freedom through the formalism of Lagrange or Hamilton, i.e. by setting up equations of motion. It is thus a very important tool to be able to solve these differential equations as precisely and effectively as possible. In this project we explore algorithms for calculating second order differential equations using vector and matrix operations with the aim to better understand the computational demands of differential equation solutions and how to reduce these demands. Using the three point derivative approximation for the second order derivative we find that the matrix representing out set of coupled differential equations is a tridiagonal one. We use the Gaussian elimination method of forward and backward substitution to diagonalize this matrix for general matrix elements. We test the accuracy of this method with increasing number of points. The next step is then to use our specific form of our symmetric tridiagonal matrix with only three unique elements to create an even faster algorithm, allowing us to compare their speed difference. Finally we use the LU-decomposition method to solve our matrix equation. To do this we use already preexisting efficient code from the armadillo package, and then compare computational speed of this method for solving general matrix equations to our specialized algorithm.

Theory and Algorithms

We will be looking in detail at linear second-order differential equations.

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

More specifically, we will look at differential equations where $k^2(x) = 0$. A known physics example of this type of equation would be Poisson's equation in radial coordinates for spherically symmetric source term.

$$\frac{d^2\phi}{dr^2} = f(r)$$

In Newtonian gravity or in electromagnetism the source term is associated with a charge or mass distribution with a negative sign to explicitly indicate the attractiveness of the resulting potential. We will therefore look closely at the equation:

$$-u''(x) = f(x) \tag{1}$$

With Dirichlet boundary conditions, i.e. $x \in (0,1),\ u(0) = u(1) = 0$. To solve this equation numerically we need to discretize the functions u and f with grid points $x_i = ih$ from 0 to 1 in $n+1$ steps using step length $h = 1/(n+1)$. This means that we know the initial and final point of u through the boundary conditions, and that we can use the three point derivative formula to rewrite the equation as:

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = f_i \qquad \text{for } i = 1, 2, ..., n \tag{2}$$

The solution to this equation with the given boundary conditions is $u(x) = 1 - (1 - e^{-10}) - e^{-10x}$. For $x = 0$ we get $u(0) = 1 - (1 - e^{-10}) * 0 - e^{-10*0} = 1 - 1 = 0$, and for $x = 1$ we get $u(1) = 1 - 1 + e^{-10} - e^{-10} = 0$, so it satisfies the boundary conditions and by deriving it twice we get $u'(x) = -(1 - e^{-10}) + 10e^{-10x}$, $u''(x) = -100e^{-10x}$. So we have found the exact solution to the differential equation which gives us a value to compare to, to see the accuracy of our numerical method. The next step to solve the equation is to rewrite it as a linear set of equations:

$$\begin{pmatrix} 2 & -1 & 0 & ... & ... & 0 \\ -1 & 2 & -1 & 0 & ... & 0 \\ 0 & -1 & 2 & -1 & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & -1 & 2 & -1 \\ 0 & 0 & ... & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ ... \\ ... \\ u_n \end{pmatrix} = \begin{pmatrix} f_1h^2 \\ f_2h^2 \\ f_3h^2 \\ ... \\ ... \\ f_nh^2 \end{pmatrix} \tag{3}$$

From here on out we will absorb the h^2 term into the source function. We will now take a step back and look at a more general matrix $\mathbf{A}$ for this sort of linear problem and developing an algorithm for it. This allows us to see the difference in CPU time between the general algorithm and one made specifically for our very symmetric case. The more general matrix we will look at is a tridiagonal matrix:

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & ... & ... & 0 \\ a_1 & b_2 & c_2 & 0 & ... & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & ... & 0 & a_{n-1} & b_n \end{pmatrix} \tag{4}$$

We now see that this problem is on the type $\mathbf{Au} = \mathbf{f}$ were we know the values of $\mathbf{A}$ and $\mathbf{f}$. A solution for this problem is to form a diagonal matrix of $\mathbf{A}$, such that we easily can read out the values of $\mathbf{u}$. As the matrix is tridiagonal we can do this in two steps, forwardsubstitution where we remove the a values and backwardssubstitution were we remove the c values. Looking at the first 2 rows, we see that if we subtract the first row multiplied by a_1/b_1 we get

$$\begin{pmatrix} b_1 & c_1 & 0 & ... & ... & 0 \\ 0 & b_2 - \frac{a_1c_1}{b_1} & c_2 & 0 & ... & 0 \\ 0 & a_2 & b_3 & c_3 & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & ... & 0 & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ ... \\ ... \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 - \frac{f_1a_1}{b_1} \\ f_3 \\ ... \\ ... \\ f_n \end{pmatrix} \tag{5}$$

If we do this all the way to the end of the matrix, we get a new matrix with only non-zero elements on and above the diagonal. If we call the new values of the diagonal element $\tilde{b}$ we have an explicit expression for them as

$$\tilde{b_i} = b_i - \frac{a_{i-1}c_{i-1}}{\tilde{b_{i-1}}} \qquad i = 2, ..., n \tag{6}$$

We also have to update the $\mathbf{f}$ in a similar manner

$$\tilde{f_i} = f_i - \frac{a_{i-1}f_{i-1}}{\tilde{b_{i-1}}} \qquad i = 2, ..., n \tag{7}$$

In the end of this forward substitution we have

$$\begin{pmatrix} b_1 & c_1 & 0 & ... & ... & 0 \\ 0 & \tilde{b_2} & c_2 & 0 & ... & 0 \\ 0 & 0 & \tilde{b_3} & c_3 & 0 & ... \\ ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & 0 & \tilde{b_{n-1}} & c_{n-1} \\ 0 & 0 & ... & 0 & 0 & \tilde{b_n} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ ... \\ ... \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ \tilde{f_2} \\ \tilde{f_3} \\ ... \\ ... \\ \tilde{f_n} \end{pmatrix} \tag{8}$$

To continue we have to remove the elements above the diagonal. This time we first look at the elements in the last row that the first elements u_n is given by $\tilde{f_n}/\tilde{b_n}$. To find the next elements u_{n-1} we simply see that the equation for this row is

$$u_{n-1}\tilde{b_{n-1}} + c_{n-1}u_n = \tilde{f_{n-1}} \tag{9}$$

We know all values except u_{n-1} and solve for this value in an interative process upwards. This gives the algorithm

$$u_i = \frac{\tilde{f_i} - c_i u_{i+1}}{\tilde{b_i}} \quad i = n-1, n-2, ..., 1 \tag{10}$$

where the sequence for the i's are important(i.e starting with the known value in the bottom). If we count the flops without any further simplifications, we get $3 * 3n = 9n$ flops. Later in the algorithm section we see that we can actually reduce this value by 1 to get 8n flops.

We can then look at our specialized example were we know the coefficients of the matrix **A**. By precalculation some of the algorithms we can reduce the number of operations for a given gridsize n by a substantial factor. If we insert the numbers into the first forward substitution equation 6 we get

$$\tilde{b_i} = 2 - \frac{1}{\tilde{b_{i-1}}} \quad i = 2, ..., n \tag{11}$$

For n=2,3,4 we have

$$\tilde{b_2} = 2 - \frac{1}{2} = \frac{3}{2} \tag{12}$$

$$\tilde{b_3} = 2 - \frac{2}{3} = \frac{4}{3} \tag{13}$$

$$\tilde{b_4} = 2 - \frac{3}{4} = \frac{5}{4} \tag{14}$$

where we start to suspect a series on the form

$$\tilde{b_i} = \frac{i+1}{i} \quad i = 2, ..., n \tag{15}$$

We can see that this step only has 2n flops. The equations for **f** also simplify to

$$\tilde{f_i} = f_i + \frac{f_{i-1}}{\tilde{b_{i-1}}} \quad i = 2, ..., n \tag{16}$$

where we now equally have 2n flops for this step.
For the backward substitution we get

$$u_i = \frac{\tilde{f_i} + u_{i+1}}{\tilde{b_i}} \quad i = n-1, n-2, ..., 1 \tag{17}$$

which has 2n flops. This gives in total 6n flops for this specified algorithm, a significant improvement compared to the general case.

A. Memory handling and algorithms

To efficiently handle the memory and not generate lots of copies of the matrices, vectors are used as the algorithms act on a vector-type structure. For each round of the general algorithm, as in the forward substitution of b we can pass the array by reference and only update the values. The pseudocode for the forwards substitution would then be

```
//general forward algorithm
//start at 1 as the 0 elements is ok
for (int i=1; i<=n; i++)
        {
        b[i]=b[i]-c[i-1]*b[i-1]/b[i-1];
        f[i]=f[i]-c[i-1]*f[i-1]/b[i-1];
        }
```

We notice that the factor $a[i-1]/b[i-1]$ gets calculated twice. This is a waste of flops, so to reduce the number of flops by 1 we can calculate this value and save it as an intermediate step. This will now give

```
//general forward algorithm
//start at 1 as the 0 elements is ok
double intermediate;
for (int i=1; i<=n; i++)
        {
        intermediate = a[i-1]/b[i-1];
        b[i]=b[i]-c[i-1]*intermediate;
        f[i]=f[i]-f[i-1]*intermediate;
        }

//general backward algorithm
//start at 1 as the 0 elements is ok
for (int i=n-1; i>=0; i--)
        {
        u[i]=(f[i]+u[i+1])/b[i];
        }
```

We see that in total we have 8n flops for the general algorithm. We know that a in specialized case where we tailor the algorithm for the problem we usually can reduce the number of flops, so let's go through the specified program.

```
//specified forward algorithm
//start at 1 as the 0 elements is ok
for (int i=1; i<=n; i++)
        {
        b[i]=(i+1)/i;
        f[i]=f[i]+f[i-1]/b[i-1];
        }

//specified backward algorithm
//start at 1 as the 0 elements is ok
for (int i=n-1; i>=0; i--)
        {
        u[i]=(f[i]+u[i+1])/b[i];
        }
```

Giving 6n flops.

Results

Table 1: My caption

log10(N)	log10(ϵ)	Time General (s)	Time Special(s)	Time LU(s)
1	-1.097	$5.000 \cdot 10^{-6}$	$1.000 \cdot 10^{-6}$	INSERT
2	-0.004789	$3.000 \cdot 10^{-6}$	$2.000 \cdot 10^{-6}$	
3	-3.002	$2.400 \cdot 10^{-5}$	$2.000 \cdot 10^{-5}$	
4	-4.000	0.0002260	0.0001910	NA
5	-5.000	0.002342	0.001957	NA
6	-6.000	0.02318	0.02478	NA
7	-6.244	0.2693	0.2175	NA

Conclusion

References

[1] Morten Hjort-Jensen *Computaional Physics Lecture Notes Fall 2015* Department of Physics, University of Oslo 2015

`https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`