FYS3150 - Computational physics

Project 1

Tridiagonal matrix

Filip Henrik Larsen
filiphenriklarsen@gmail.com

Dato: 6. september 2014

Description

In this project I were to solve the one-dimensional Poissson equation,

$$-u''(x) = f(x).$$

, with Dirichlet boundary conditions:

$$u(0) = u(1) = 0.$$

I did this by rewriting it as a set of linear equations. This was accomplished by using the formula

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

for the second derivative. Here v is the position, and h is the position step lenght. I considered the intervall

$$x = [0, 1] \quad \text{with step length} \quad h = 1/(n+1)$$

The functions $u(x)$ and $f(x)$ were given as:

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \quad \& \quad f(x) = 100e^{-10x}$$

The goal was to write a code that could solve the linear equation to obtain a position vector, v, that should represent $u(x)$.

The Equation

The first step was to rewrite the equation

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i$$

as

$$-v_{i-1} + 2v_i - v_{i+1} = b_i \quad , b_i = h^2 f_i$$

Now, v is a vector, and the weights of their coefficient has to change with i. A simple equation for this problem can therefore be expressed with the tridiagonal matrix

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix}$$

So that the equation becomes:

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & \ldots \\ a_2 & b_2 & c_2 & \ldots & \ldots & \ldots \\ & a_3 & b_3 & c_3 & \ldots & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \ldots \\ \ldots \\ \ldots \\ \tilde{b}_n \end{pmatrix}$$

Leading up to the final equation

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i,$$

Here, the vector $\tilde{b}$, is known, and the and the coefficients a_i, b_i and c_i are $-1, 2$, and -1 respectively. In my code I chose to name the product vector x in order to remove confusion. This is the equation I had to solve. It was simple enough, but I did not come up with the mathematical solution by myself. I do understand it though, and I did code the algorithms from my interpretation of the mathematics. I will now try to explain it, but I will not show the entire prossess. I found a very good explanation at `http://www3.ul.ie/wlee/ms6021_thomas.pdf`, so check it out!

Solving the linear equation

To solve the equation there was really only two steps; Forward- and backwards substitution.

Forward substitution:

The first row will only consist of two values of $\vec{v}$: v_1 and v_2.

(1) The first equation read:

$$b_1 v_1 + c_1 v_2 = x_1$$

Now if I divide through by b_1 I obtain

$$v_1 + g_1 v_2 = p_1$$

With the substitution

$$g_1 = \frac{c_1}{b_1} \ , \quad p_1 = \frac{x_1}{b_1}$$

(2) The second equation read:

$$a_2 v_1 + b_2 v_2 + c_2 v_2 = x_2$$

Now I subtract a_2 times row 1 from row 2, giving:

$$a_2 v_1 + b_2 v_2 + c_2 v_2 - a_2(v_1 + g_1 v_2) = x_2 - a_2 p_1$$

$$(b_2 - a_2 g_1) v_2 + c_2 v_2 = x_2 - a_2 p_1$$

We now divide through by $(b_2 - a_2 g_1)$ to obtain

$$v_2 + g_2 v_2 = p_2$$

With the substitution

$$g_2 = \frac{c_2}{b_2 - a_2 g_1} \ , \quad p_1 = \frac{x_2 - a_2 p_1}{b_2 - a_2 g_1}$$

(i) We continue just as we did in (2) through all the values of i. This leads to the final linear equation:

$$\begin{pmatrix} 1 & g_1 & 0 & \dots & \dots & \dots \\ 0 & 1 & g_2 & \dots & \dots & \dots \\ & 0 & 1 & g_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & 1 & g_{n-1} \\ & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ \dots \\ \dots \\ \dots \\ p_n \end{pmatrix}$$

with

$$g_i = \frac{c_i}{b_i - a_i g_{i-1}} \ , \quad p_i = \frac{x_i - a_i p_{i-1}}{b_i - a_i g_{i-1}}$$

Backward substitution:

As we can see, the forward substitution ended with the last row revealing v_n! Since this becomes known, the only unknown element in the next to last row is v_{n-1}, making that and all the rest possible to find. We therefore start at $i = n$ and store the v-values with the formula

$$v_i = p_i - g_i v_{i+1}$$

The script

There isn't much left to tell about my algorithms. They are pretty much exactly as the last two algorithms above. It is quite compactly coded, so it is easy to follow. The loops for Forward- and Backward substitution are marked by comments in the script.

- Asking for a value, N, for the dimention of the vector/matrix/system?. This value determins the value of the step length, h.
- Creates vectors and filling the known ones. Also add the values $a_1 = 0$ and $c_n = 0$. Since these are not in the matrix, they should not interfere with the algorithms.
- Starts the clock for time measurement.
- Stores the values for the first substitution, and starts the forward substitution algorithm.
- Stores the initial value $v_0 = 0$, and starts the backwards substitution.
- Stops the the clock, calculate and print out the time duration of the program. From what I read this is the CPU-time and not some physical time unit.
- Prints out number of FLOPS.
- Writes all the elements in the vector v and the value of N to a .txt -file.
- Tells the system to run the python script which reads the .txt -file and stores the elements in an array. It then deletes the .txt -file.
- Python generates the values of x and u(x) and plots the the result. The plot gets stored as a .JPEG -file, then the python script ends.
- C++ calculates the relative error and prints the maximum relative error to screen.

In the code I have commented out the FLOPS-count, though to use it you only have to remove the commenting symbols.

Results

The python program receiving the calculated vector plots it as a function of x, and on the same figure it plots $u(x)$ as a comparison. I ran the program several times in order to use different values of N; 10, 100 and 1000.

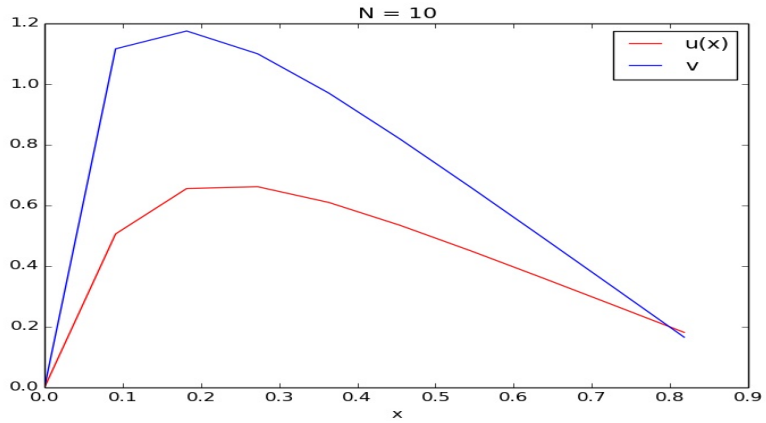

Figure 1: Calculated position values vs real position values, N=10.

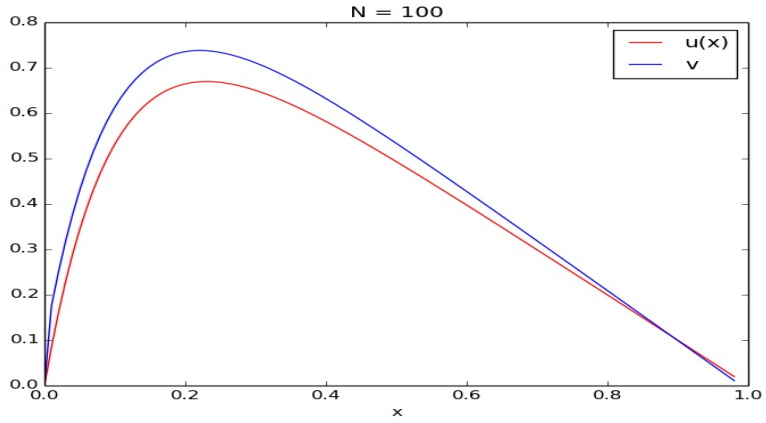

Figure 2: Calculated position values vs real position values, N=100.

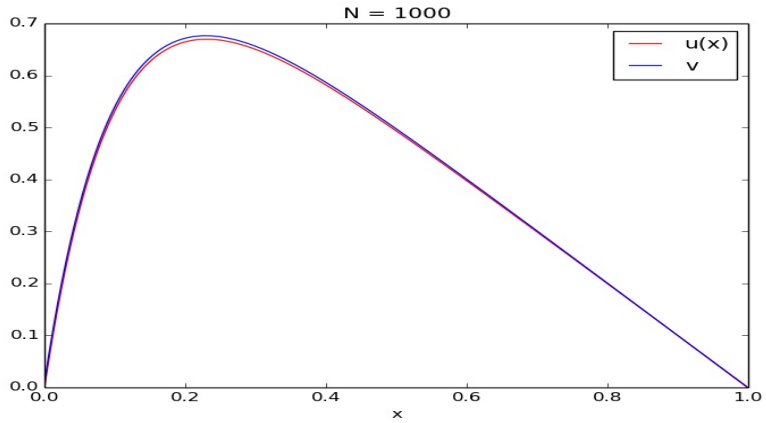

Figure 3: Calculated position values vs real position values, N=1000.

From the series of figures we notice that as N increases, from 10 to 1000, v becomes a better and better approximation of $u(x)$. This is a consequence of the position step length, h, becomeing smaller. One would expect the approximation becomeing better as the value of h decreases until h is so small that (serious) roundoff errors occur. However I did not reach this limit.

N	10	10^2	10^3	10^4	10^5
FLOPS	88	988	9988	99988	999988
Maximum relativ error	0.0812318	3.31901E-02	3.80117E-03	3.85444E-04	3.85981E-05

Table 1: Maximum relativ error and number of floating point operations corresponding to different values of N.

The number of FLOPS, as we can see, seems to follow a pattern. I found it difficult to write it mathematically, however it approximately goes as $10N$. This was more then I thought it would be. I had heard it would go as $9N$, but I guess what's most important is the dependence of the first order of N. In the lectures I was told that the number of FLOPS in the standard Gaussian elimination method is something like $\frac{2}{3}(N^3)$, which for lagre N, can be seriously big. The LU method was somewhat better with (only) $O(N^2)$, for a tridiagonal matrix that is.

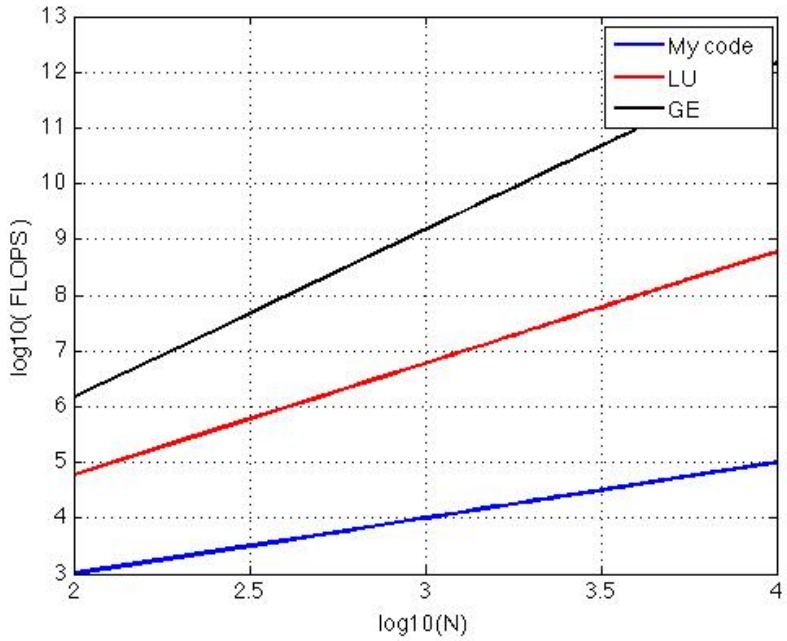

Figure 4: Comparison of FLOPS.

Obviously, since my program had to do a lot less calculations, it was faster then using the LU and *solve* functions in armadillo. I took the time measurement of both methods, and the result is in the table bellow.

N	10	10^2	10^3	10^4	10^5
My code	1.4e-05	9.3e-05	0.002165	0.011724	0.095005
LU	0.00071	0.054184	9.64154		

Table 2: Comparison of time-consumption.

The reason why I did not get any data on the last two tests for the LU-method is that it has to store at least $10^5 * 10^5 = 10^{10}$ elements for the initial matrix A. Considering that these elements are doubles, for precision purposes, that gives $10^{10} * 64\text{bits} = 80\text{Gb}$! It then has to construct two more matrices of the same size for the L and U matrices. If it can even comprehend the memory demands it will take a long time with the 10^{10} or so FLOPS, and therfor be unefficient.

The code I wrote, on the other hand, stores the diagonals (upper, middle and lower) as arrays, taking up just a fraction of the memory. And from there it takes the forward- and backwards substitutions without the mess with constructing big matrices.
So to sum it up, the code I wrote is far superior to the LU-method in order to solve the tridiagonal matrix problem. It has to do way less calculations and take up less memory and is therefor mush faster. On the downside it is very limited in its applications. It can only take on a tridiagonal matrix, while the other can consider other types as well.

Critique

As I am new to the C++ language I obviously learned a lot through doing this exercise. I have still to learn how to make nice plots though. I found it difficult to make up the algorithms for the forwards- and backwards substitutions myself, which is why I looked it up online. Afterwards I noticed a hint in the lecturenotes. A heads up would have been great, but now I know where to look for clues next time.