Project 1, FYS 3150 / 4150, fall 2013

Nathalie Bonatout and Odd Petter Sand

September 11, 2013

1 Introduction

In this project we will solve the one-dimensional Poissson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

To be more explicit we will solve the equation

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

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

where $f_i = f(x_i)$.

(Author's note: This text, and the text introducing the various exercises, is taken from the project description provided at the course website.)

2 Exercises

2.1 Exercise a)

We start with the given equation

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

where $i \in [1,n] \cap \mathbb{N}$ and $n \in \mathbb{N}$. We will assume that $n \geqslant 3$ and that $v_0 = v_{n+1} = 0$.

$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \equiv \widetilde{b}_i$

$$\begin{bmatrix} -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} v_{i-1} \\ v_i \\ v_{i+1} \end{bmatrix} = \widetilde{b}_i$$

Now we expand these vectors from 3 elements to n elements. Note that the ith element of the row vector should be 2 and the ith element of the column vector should be v_i after the expansion:

$$\begin{bmatrix} 0 & \cdots & -1 & 2 & -1 & \cdots & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_{i-1} \\ v_i \\ v_{i+1} \\ \vdots \\ v_n \end{bmatrix} = \widetilde{b}_i$$

Note that in the row vector 2 can very well be the first or last element, in which case the preceding presentation can be a little misleading. We further recognize the row vector as the ith row of the $n \times n$ matrix $\mathbf{A}$ (defined such that $A_{ij} = 2\delta_{ij} - \delta_{i(j-1)} - \delta_{i(j+1)}$) and get the inner product

$\mathbf{A}_i \cdot \mathbf{v} = \widetilde{b}_i$

and remembering that $i \in [1,n]$, by the definition of matrix multiplication

$$\mathbf{Av} = \widetilde{\mathbf{b}}$$

which is what we wanted to show.

2.2 Exercise b)

The algorithm we will use is as follows: First we do forward subsitution by subtracting from the current row a multiple of the row before it.

$A_i = A_i - x_i A_{i-1}$

Here, x_i is the factor that cause the term a_i to cancel out. Before the first step, the rows $i-1$ and i look like this (the * indicates a value that has been changed by the algoritm)

$$\begin{array}{cccccccc} \cdots & 0 & b^*_{i-1} & c_{i-1} & 0 & 0 & \cdots \\ \cdots & 0 & a_i & b_i & c_i & 0 & \cdots \end{array}$$

and our goal is that it look like this after the forward substitution

$$\begin{array}{cccccccc} \cdots & 0 & b^*_{i-1} & c_{i-1} & 0 & 0 & \cdots \\ \cdots & 0 & 0 & b^*_i & c_i & 0 & \cdots \end{array}$$

(note that c_i is unchanged). When we reach the bottom, we do a backward subsitution by adding to the current row a multiple of the row below it and then dividing the last element by itself to make the last element equal to 1:

$A_i = A_i - c_i A_{i+1}$

$A_i = \frac{A_i}{b^*_i}$

That is to say, we go from

$$\begin{array}{cccccccc} \cdots & 0 & b^*_i & c_i & 0 & \cdots \\ \cdots & 0 & 0 & 1 & 0 & \cdots \end{array}$$

to

$$\begin{array}{cccccccc} \cdots & 0 & 1 & 0 & 0 & \cdots \\ \cdots & 0 & 0 & 1 & 0 & \cdots \end{array}$$

Naturally, we also have to do equivalent operations on the vector $\widetilde{\mathbf{b}}$ on the other side of the equation. We will not calculate any unnecessary values, i.e. values

that will not be used by the program later on. Hence the algoritm looks like this (number of flops in parantheses):

For $i: 2 \longrightarrow n$

1. $x_i = \frac{a_i}{b^*_{i-1}}$ (n)
2. $b_i = b_i - x_i c_{i-1}$ $(2n)$
3. $\widetilde{b}_i = \widetilde{b}_i - x_i \widetilde{b}_{i-1}$ $(2n)$

$\widetilde{b}_n = \frac{\widetilde{b}_n}{b_n}$ (1)

For $i: n-1 \longrightarrow 1$

1. $\widetilde{b}_i = \widetilde{b}_i - c_i \widetilde{b}_{i+1}$ $(2n)$
2. $\widetilde{b}_i = \frac{\widetilde{b}_i}{b_i}$ (n)

This makes the total running time of the algorihm $8n$ (actually, it is $8(n-1)$, but we are mostly concerned with the performance for large n, where $n \approx n-1$).

If the matrix is symmetric, so that $a_i = c_{i-1}$, then $x_i c_{i-1} = \frac{a_i^2}{b^*_{i-1}}$.

If $a_i = c_{i-1} = k$ for all i, we have a special case where $k = \pm 1$. Then $x_i = \pm \frac{1}{b^*_{i-1}}$, $k^2 = 1$ and $x_i c_{i-1} = \frac{1}{b^*_{i-1}} = -x_i \equiv y_i$.

Furthermore, $c_i \widetilde{b}_{i+1} = k \widetilde{b}_{i+1} = \mp \widetilde{b}_{i+1}$ (note: opposite of the sign of k). The algorithm will then look like this for $k = -1$:

For $i: 2 \longrightarrow n$

1. $y_i = \frac{1}{b^*_{i-1}}$ (n)
2. $b_i = b_i - y_i$ (n)
3. $\widetilde{b}_i = \widetilde{b}_i + y_i \widetilde{b}_{i-1}$ $(2n)$

$\widetilde{b}_n = \frac{\widetilde{b}_n}{b_n}$ (1)

For $i: n-1 \longrightarrow 1$

1. $\widetilde{b}_i = \widetilde{b}_i + \widetilde{b}_{i+1}$ (n)

2. $\widetilde{b}_i = \frac{\widetilde{b}_i}{b_i}$ (n)

This gives a total running time of $6n$ for this implementation of the algorithm, which we ended up using for this project.

The following figures are the plots given after using our tridiagonal solver (in blue) and after computing the results with the following equation, given in the project description (in red here);

$$u(x) = 1 - (1 - e^{-10}) * x - e^{-10x}$$

We can see that the bigger number of points we take, the better accuracy we get, which is expected: by increasing our number of points, we increase the number of samples to do our computation. But we too increase the probability of meaningful round-off errors happening, since two consecutives results will be really close if we have a large number of points

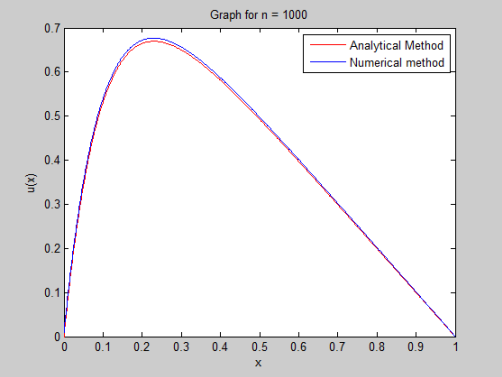

Figure 1: Analytical and numerical results for n = 1 000

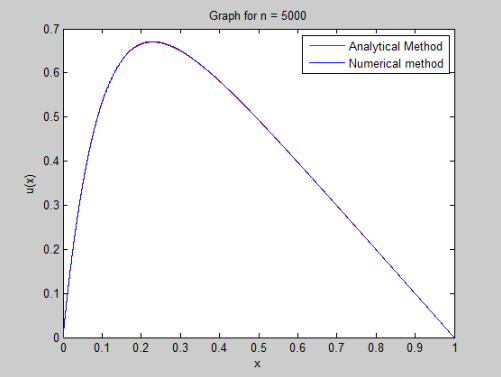

Figure 2: Analytical and numerical results for n = 5 000

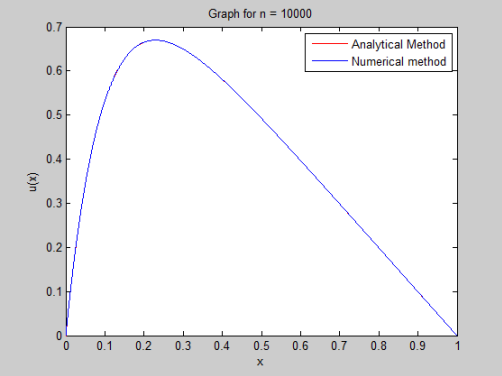

Figure 3: Analytical and numerical results for n = 10 000

2.3 Exercise c)

In this exercise we compute the relative error in the data set $i = 1, \ldots, n$, by setting up

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. Here, v_i is our numerical solution, given by our tridiagonal solver. u_i is the analytical solution.

Number of points	10000	25000	50000	100000
Step length	10^{-4}	4.10^{-5}	2.10^{-5}	1.10^{-5}
Max. relative error (%)	$0,07$	$0,03$	$0,01$	$0,007$

Table 1: Maximum relative error for different step lengths

The main point here, is that by increasing the number of sample in our interval, we can see that our results are getting more accurate. But by doing that, we induce a round-off error, which explains why we aren't getting any smaller error. The consecutive results are indeed getting closer and closer as the number of points grows,

But it is also important to underline that the points for which the relative error between the analytical and the numerical results is maximum, are the boundary points. Thus, wathever number of points we take, the relative error will remain on these points, significant.

2.4 Exercise d)

We will try to show here the differences between our tridiagonal solver, and the LU function implemented in Armadillo. We will especially look at two things: the time spent inside these functions, and the number of floating point operations -FLOPS- (in order to compute the elapsed time for each function, we will use the functions available in the library time.h).

In terms of FLOPS, to solve a n system, our tridiagonal solver needs

$$6n$$

With the LU function, decomposing a n*n matrix costs

$$\frac{2}{3}n^3$$

Dimension of the matrix	$(10 * 10)$	$(100 * 100)$	$(1000 * 1000)$
Time spent : tridiagonal solver (s)			
Time spent : LU from Armadillo (s)			

Table 2: Time usage between LU decomposition and our tridiagonal solver

We can already deduce that our tridiagonal solver will probably run faster than the LU function.

We can see that (probably ~~)our tridiagonal solver is faster than the LU function. This is a rather expected result: indeed, we implemented our solver especially for this system. We knew that the matrix we were going to compute was a tridiagonal matrix, and we even knew that it was symmetric, before writing our algorithm. Thus, we avoid some "irrelevant" steps done by the LU function, such as computing coefficient that, we know it, are null.

2.5 Exercise e)

In this exercise we are investigating matrix multiplication in row major order versus column major order with regards to running time.

The task here is to write a small program which sets up two random (use the ran0 function in the library lib.cpp to initialize the matrix) double precision valued matrices of dimension 5000×5000. (NOTE: The original value of $10^4 \times 10^4$ proved too memory intensive for our poor laptops.)

The multiplication of two matrices $\mathbf{A} = \mathbf{BC}$ could then take the following form in standard row-major order

```
for(j=0 ; j < n ; j++) {
    for(k=0 ; k < n ; k++) {
        a[i][j]+=b[i][k]*c[k][j]
    }
}
```

```
    }
```

and in a column-major order as

```
        for(i=0 ; i < n ; i++) {
            for(k=0 ; k < n ; k++) {
                a[i][j]+=b[i][k]*c[k][j]
            }
        }
    }
```

(NOTE: We implemented this using dynamic memory allocation, as lib.cpp proved difficult to add to the project in Visual C++.)

The output of our program:

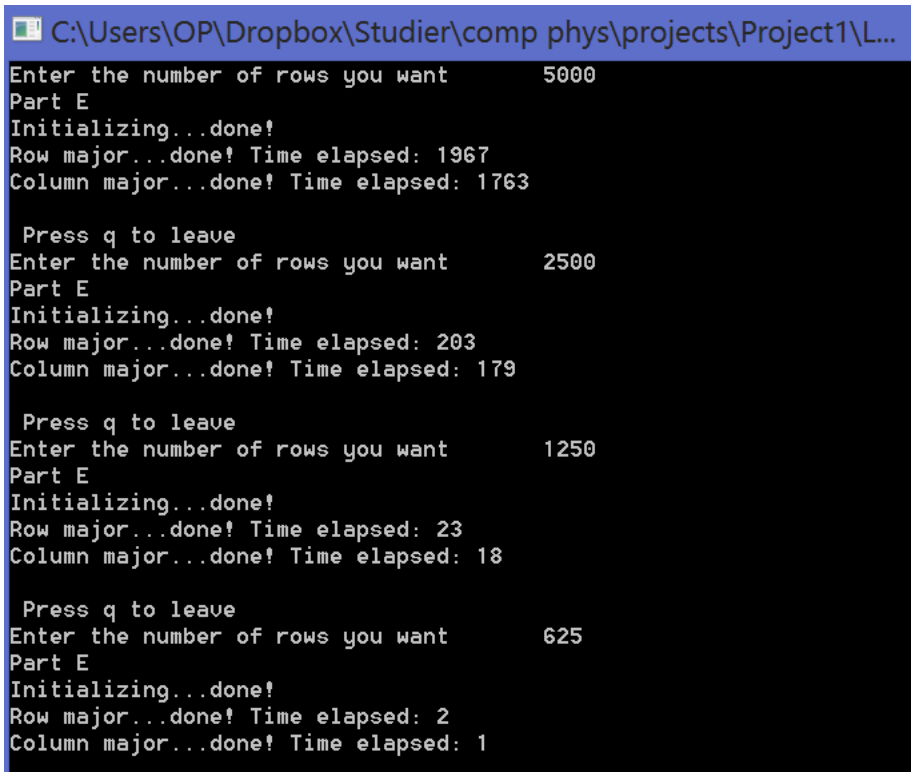

Figure 4: Timings of row and column major multiplications

TOTALLY BOGUS due to coffee break. :p

Figure N: (Note that the "Part B" is a typo. It should read "Part E".)

3 Conclusion

In this project we learned that managing and sparing our RAM is indeed important and that external libraries can be fishy, even in the nature of their parameters (byref ... !!!) é_è ! And maybe that it is, indeed, sometimes better to implment our own algorithms.

3.1 Critique

We would like to provide the following items of feedback for future versions of this project:

- lib.cpp sucks! Arma too, sometimes !!
- we need more RAM. you made our laptops feel sad.
- ???