Project 1, FYS 3150 / 4150, fall 2013

Nathalie Bonatout and Odd Petter Sand

September 14, 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);

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

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

Number of points	10	100	1000	10000	100000
Step length				10^{-4}	1.10^{-5}
Max. relative error (%)				$0,07$	$0,007$

Table 1: Maximum relative error for different step lengths

Screens/Graph_b/n=10.png

Figure 1: Analytical and numerical results for n = 10

Screens/Graph_b/n = 100.png

Figure 2: Analytical and numerical results for n = 100

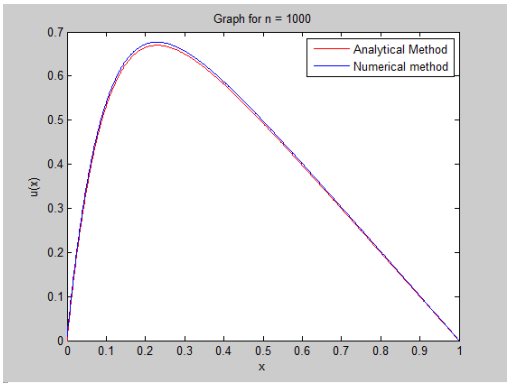

Figure 3: Analytical and numerical results for n = 1 000

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.

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.

TODO: Step length too small induces rounding errors.

2.4 Exercise d)

We will try to show here the differences between our tridiagonal solver, and the LU function implemented in Armadillo (as lib.cpp would not play nice in Windows, at least not for the moment). 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 $n \times n$ matrix costs

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

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

Dimension of $n \times n$ matrix	Time: Tridiagonal solver (s)	Time: LU decomposition (s)
$n = 10$	≈ 0	≈ 0
$n = 100$	0.001	0.001
$n = 500$	0.001	0.03
$n = 1000$	0.001	0.124
$n = 2500$	0.003	0.769
$n = 5000$	0.004	3.104
$n = 10^4$	0.006	OoM*
$n = 10^5$	0.057	OoM*
$n = 10^6$	0.575	OoM*
$n = 10^7$	5.769	OoM*
$n = 10^8$	OoM*	OoM*

Table 2: Running times for LU decomposition and tridiagonal algorithm. (*: Out of Memory error when initializing matrices.)

From our results, we make the following observations:

- Our tridiagonal solver is indeed $O(n)$, excepting some tiny overhead that is only significant when $n \lesssim 5000$.
- Given that $\frac{3.104}{0.769} \approx \frac{0.124}{0.03} \approx 2^2$ and $\frac{0.769}{0.124} \approx 2.5^2$, it is evident when comparing the values of n that Armadillo's LU implementation runs as $O(n^2)$, again with some tiny overhead for small n. This is significantly better scaling than we expect from the standard LU algorithm, and thus we conclude that Armadillo uses a more efficient implementation for this operation. Had we chosen to use lib.cpp instead, we would expect it to run as $O(n^3)$.

The reason for these differences are, naturally, that our tridiagonal solver is optimized for the problem at hand, using information about our matrix that is not available to the general LU algorithm. More to the point, the LU decomposition needs to calculate a lot of values that our tridiagonal method ignores, since we know that our matrix is very sparse.

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, so we were forced to scale it down.)

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

```
for(i=0;i<n;i++){
        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(j=0;j<n;j++){
        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 was as follows:

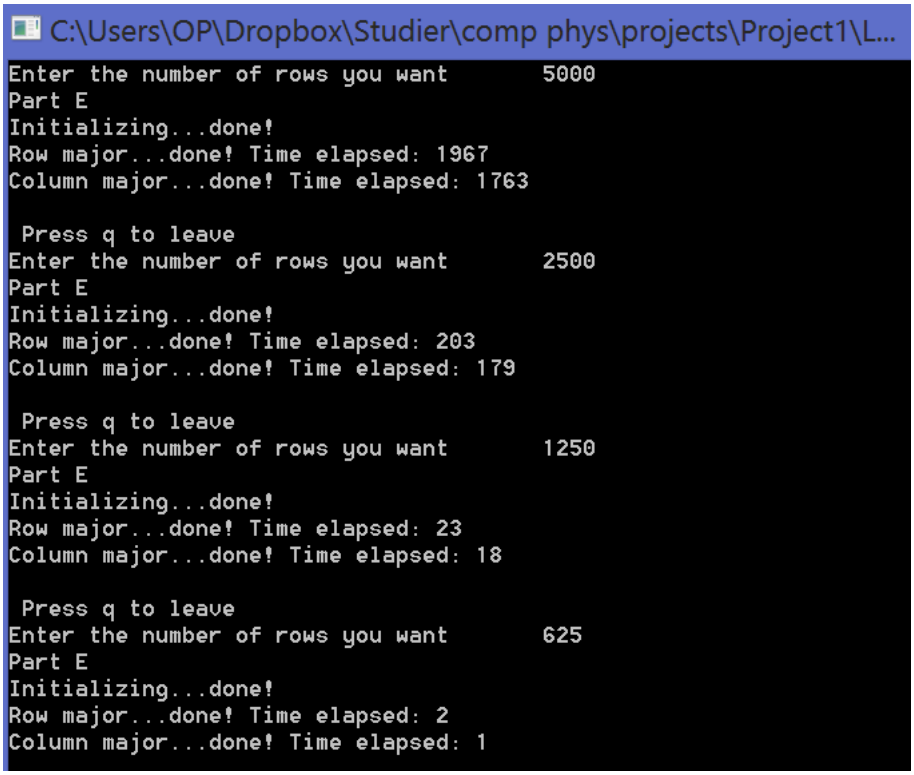

Figure 4: Elapsed time in seconds for row and column major multiplications

We see that row major order multiplication takes more than 10% longer to complete in all cases. We know that Visual C++ stores matrices in row major order (see http://msdn.microsoft.com/en-us/library/8ccd1wzb%28v=vs.110%29.aspx), and this appears to result in longer memory leaps when looking up the elements for the calculation.

(Author's note: We are a little surprised by this result. We would expect row major order to be more efficient, as the memory leaps are likely to be shorter on average. In particular, k affects rows and columns equally in total, whereas

i leaps would be longer (on average equal to the length of a row) than j leaps within the same row. We would appreciate some feedback on this to make sure we aren't misunderstanding something crucial.)

3 Source code

All our source code can be found at our GitHub repository: https://github.com/OPSand/Project1

4 Conclusion

From this project, we have learned the following things:

- Specialized algorithms can save significant computing time and memory.
- Error analysis shows that there is a tradeoff to consider when choosing a step size in a numerical method, aside from running time. A smaller step size is only an improvement up to a point, where a smaller step size may introduce loss of precision due to rounding errors - as such, there exists a "sweet spot" where one can get the lowest possible relative error. Analysis of the output compared to a known solution can help locate this sweet spot.
- Row major order vs. column major order matters when you do matrix multiplication, due to differing average memory jump length.

4.1 Critique

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

- We would be very thankful if lib.cpp were easier to include in our Visual C++ projects on Windows. We ended up using Armadillo instead, which worked well enough for our purposes (and even produced some interesting deviations from the expected result in exercise d), but for future projects it would be nice if we got it to work from the get-go. Is it possible to look into this?

- It would be nice if Armadillo was introduced even earlier - we ended up doing some things twice because we wanted to start early and hadn't gotten Armadillo to work yet.

- Even though we were able to make Armadillo work on Windows and shared the method, many other students found it cumbersome. We recommend putting together a guide on how to do this (and we're happy to share our experiences).

- We found GitGUI for Windows extremely cumbersome to use, especially when branching (we were forced to make local backups and clone the repository from scratch several times). We recently switched to GitHub for Windows (http://windows.github.com/) and never looked back. We would recommend mentioning its existence to students who want to code in Windows. It sports a very intuitive user interface, and we found that its changelog all but eliminates the need for keeping a separate log.