FYS3150 - Project 1

Alfred Alocias Mariadason

08.09.2014

Brief description of problem

In this project the task was to solve the one-dimensional Poisson equation by rewriting it as a set of linear equations, with other words, basically solving a second order differential equation of form:

$$A\vec{v} = f$$

Where A is an $n \times n$ tridiagonal matrix

Algorithm used

The algorithm used in this project was a simple algorithm consisting of a decomposition, forward elimination and then a backward substitution. This was possible because the matrix at hand is tridiagonal and the middle diagonal was dominant($b_i > a_i, c_i$).
The matrix at hand looks like this:

$$A = \begin{bmatrix} b1 & c1 & \dots & \dots & \dots & \dots \\ a2 & b2 & c2 & \dots & \dots & \dots \\ \dots & a3 & b3 & c3 & \dots & \dots \\ & & \ddots & \ddots & \ddots & \\ \dots & \dots & \ddots & \ddots & \ddots & \dots \\ \dots & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} \\ \dots & \dots & \dots & \dots & a_n & b_n \end{bmatrix} \tag{1}$$

For the forward elimination(After making a decomposition and only looking at the lower triangle of the matrix) one does a general gauss eliminiation process. Multiply row 1 with a_2 and row 2 with b_1, then subtract 1 from 2. Generally put, one multiplies n-1'th row with a_n and n'th row with b_{n-1} this basically eliminates all the a's. Multiply $\vec{v}$ into the matrix and we have:

$$\left.\begin{array}{c} a_2b_1v_1 + c_1a_2v_2 = \tilde{b} \\ (b_2b_1 - c_1a_2)v_2 + c_2v_3 = \tilde{b} \\ \vdots \\ b_{n-1}a_nv_{n-1} + c_{n-1}a_nv_n = \tilde{b}_{n-1} \\ (b_nb_{n-1} - c_{n-1}a_n)v_n = \tilde{b}_n \end{array}\right\} \begin{array}{l} v_2 = (\tilde{b}_1 - a_2b_1v_1)/c_1a_2 \\ v_3 = (\tilde{b}_2 - v_2(b_2b_1 - c_1a_2))/c_2 \\ v_n = (\tilde{b}_{n-1} - b_{n-1}a_nv_{n-1})/c_{n-1}a_n \\ v_n = \tilde{b}_n/(b_nb_{n-1} - c_{n-1}a_n) \end{array}$$

Now we go for the backward substitution. The only process needed here is to solve the last equation for v_{i-1} and then running a backwards recursive loop(start at end and run to 1). The algorithm itself(the way it's programmed) looks like this.

$btemp_i = b_1$
$v_1 = \tilde{b}_1/btemp$
$temp_1 = c_1/btemp_1$
for $i = 2, ..., n+1$:
  $btemp_i = b_i - a_i temp_{i-1}$
  $temp_i = c_i/btemp_i$
  $v_i = (\tilde{b}_i - a_i v_{i-1})/btemp_i$
end

for $i = n, ..., 0$:
  $v_i -= temp_i v_{i+1}$
end

Note that this algorithm assumes the boundary values(values at each end) is set explicitly before the loop is run.
We can see here that the exact number of flops needed is 8 if one counts division and subtraction as one flop(eventhough these two operations might prove to use more than that of one flop).

Rewriting the equation

Rewriting the Poisson equation as a linear set of equation is fairly straight forward. If we first use the matrix given in the project(matrix A). The only operation necessary is to transpose our $\vec{v}$, with other word we define:

$$\vec{v} = (u_1, ..., u_n)^T$$

Multiply this with A and we get the linear set of equations:

$$A\vec{v} = \tilde{b}$$

$\tilde{b} = h^2 f_i$

Test of program and plots

Using the program 2fe listed in last section the plots looks like this:

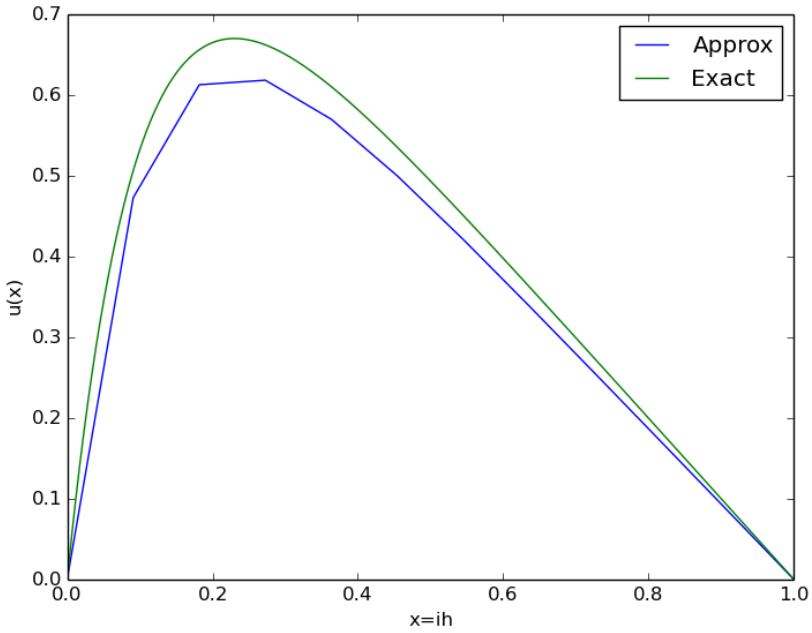

figure 1: n=10

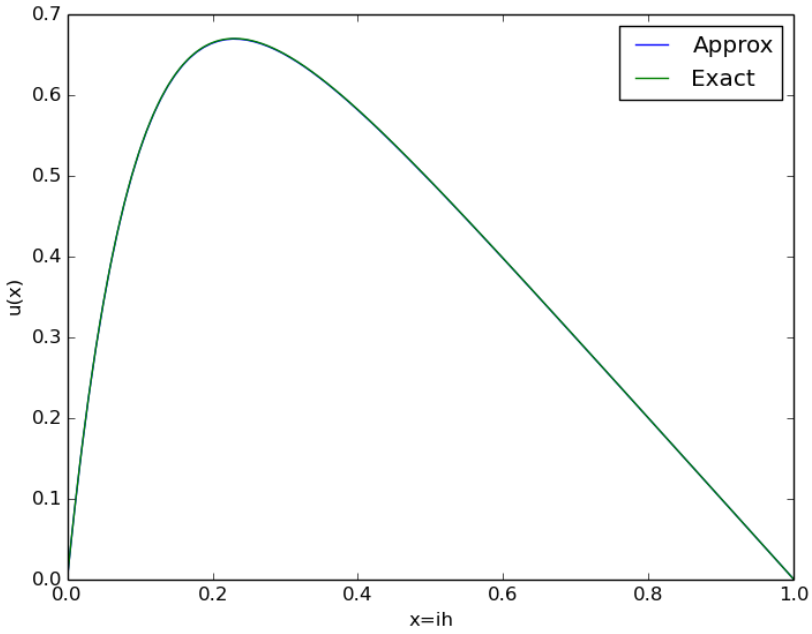

figure 2: n=100

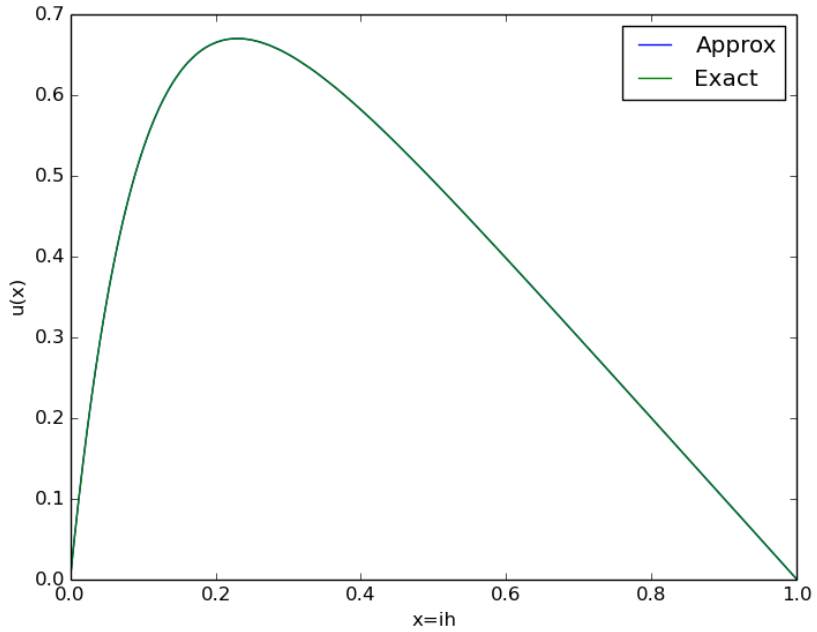

figure 3: n=1000

One can see here that increasing the number of iterations increases the accuracy of our approximation. The plot also suggests that by increasing N the graph quickly converges towards the exact solution. The difference between the numerical solution and the exact one is undistinguishable already at $n = 100$.

Relative error

The relative error looks something like this:

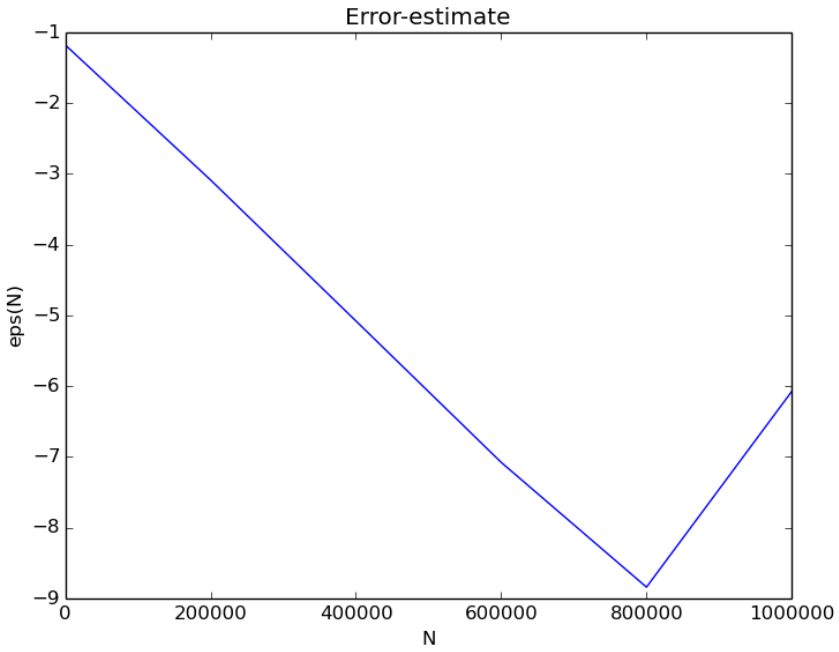

figure 4 error estimate. n=1E1,1E2,1E3,1E4,1E5,1E6

The error decreases drastically before increasing again. This increase in error is probably because of round off error on the computer. With other words error in floating number calculations. Note that we also take log10 of the relative error, this also causes problems when N is large.

Compare with LU

Running a performance check on the TDMA method and LU in armadillo the comparison table looks like this:

	TDMA	LU
n=10	4.1μs	172μs
n=100	29μs	1690μs
n=1000	293μs	74162μs

One can see here that the LU-method is alot slower than our algorithm. This is due to that the LU-method doesn't account for the zeros in the matrix. Armadillo doesnt explicitly tell the number of flops used, however one can see that the time usage increases with a factor of n^3. Running a $10^5 \times 10^5$-matrix gave an error. Armadillo didn't want to allocate the memory needed to make a matrix of that size, which is quite understandable(a $10^4 \times 10^4$-matrix gave a complete freeze).

Programs

`https://github.com/0o1Insane1oO/project1`
main program(c++): 2fe.cpp
plotter(python): 2fe-plot.py
Note: everything is made to run on linux.