Fys4150
Project 1

Peter Killingstad and Karl Jacobsen

September 5, 2017

Abstract

A 1D Poisson problem is solved numerically with three different algorithms in C++. The Thomas algorithm, a simplified version of this algorithm, and the LU decomposition are used. Dynamical memory allocation is used. Theoretical derivations of algorithm efficiency (FLOPS) and truncation and round-off error are derived. The numerical simulations confirms the analytical derivations, with a truncation error of order 2 and a given mesh size that gives a minumim error.

Contents

1 Introduction

Many real life physical problems are formed as differential equations. However, most of these problems are not possible to solve analytically. With the rise of computational power, many real life problems can now we solved numerically. Hence, knowledge of numerical approximation theory and about practical numerical simulations is very important.

Computational speed is a crucial variable when it comes to numerical solutions of differential equations. A fast computational language is necessary, and knowledge of a fast computer language is essential. In this project we are starting to learn the C++ language, which is one of the fastes programming languages. In the project we learn about how to implement dynamic memory allocation in C++. Dynamic memory allocation and pointers lets us take advantage of the computational potential of C++.

The first task when it comes to solving a differential equation numerically, is to discretize the equation. This is done in this project. Solving large problems on a computer, often ends in solving a linear system. In this project we learn how rewrite a numerical problem to a set of linear equations.

The linear systems resulting from differential equations are ofte large. Hence it is important with knowledge about different solution algorithms for linear systems, among other things about the speed of the algorithms. In this project we derive two algorithms, the Thomas algorithm and a simplified version of the Thomas algorithm, and we compute the FLOPS (number of floating point operations) for each algorithm. Finally, we program these algorithms and run them.

Often one does not write the algorithms, but use functinos written by others. In this project we learn how to implement externally given programs, in this case the famnous LU-decomposition.

A differential equation that shows up in many physical cases, is the Poisson equataion. Electromagnetism, heat transfer, and linear elasticity are example of importan areas where the Poisson equation arises. Knowledge about the numerical issues relating to solution of the Poisson equation is essential. We solve a Poisson equation in this project.

Numerical simulations has additional challanges compared to anaytical solutions. Firstly, the limited data implies that we will never get fully continuous solutions on a computed. We will only get an approximation. Knowledge about errors in these approximatinos is essential. There are several type of errors that may occour. Firstly there can be a truncation error in the numerical approximation. Secondly there can be errors du to limited precision on the computer. Both of these errors can destroy the numerical approximations, giving so large errors that the solutions are worhless. In this project we derive expressions for both truncation error and precision error, so that we know how our numerical solution is supposed to act. Most of the simulations fits well with the theoretical error estimates.

2 Theory

2.1 Setting up the linear system

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

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

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0. \tag{2}$$

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

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \qquad \text{for } i = 1, \ldots, n, \tag{3}$$

where $f_i = f(x_i)$.

Now we will show that the discrete problem can be rewritten to a linear system of the type

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}.$$

The reason for wanting the rewrite to a linear system, is that we know of methods that solve such linear systems. Hence, by solving the linear system we will also solve our discrete differential equation problem!

To obtain the linear system, we can write out the discretized approximation into a system of equations. We start by setting $i = 1$, multiply the discretize equation (3) with h^2 and write out the first couple of equations to look for a pattern

$$\begin{aligned} -2v_1 + v_2 &= f_1 h^2 \\ v_1 - 2v_2 + v_3 &= f_2 h^2 \\ v_2 - 2v_1 + v_4 &= f_3 h^2 \\ v_3 - 2v_2 + v_5 &= f_4 h^2 \\ \vdots &= \vdots \\ v_{n-1} - 2v_n &= f_n h^2. \end{aligned}$$

As the equation is written out, it is clear that the values $1, -2$ and 1 is repeating. We can now rewrite the system of equations into a matrix made up of the coefficients and two vectors, one containing the unknowns v_1 to v_n and the other containing the inhomogeneous terms f_1 to f_n. The system would then be

$$\underbrace{\begin{bmatrix} -2 & 1 & \cdots & 0 \\ 1 & -2 & 1 & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 1 & -2 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}}_{\mathbf{V}} = \underbrace{\begin{bmatrix} f_1 h^2 \\ f_2 h^2 \\ \vdots \\ f_n h^2 \end{bmatrix}}_{\tilde{\mathbf{b}}} \tag{4}$$

2.2 Linear system solvers

We will apply three different algorithms for solving the above linear system. These algorithms and their theoretical efficiency (FLOP-calculations) will be described below.

2.2.1 Gaussian elimination tridiagonal systems

We use standard Gaussian elimination, utlizing the sparsity of the matrix by ignoring the zero terms when doing the forward and backward substitution. This gives the Thomas algorithm. The algorithm can be seen in our code, which follows:

```
void gaussianTridiagonalSolver(double ** computed_tridiagonal_matrix, double * computed_right_hand_side,
    double * computed_numerical_solution, int N){
  double multiplicationFactor;

  for (int i = 1; i < N; i++){
     multiplicationFactor = computed_tridiagonal_matrix[i][i-1]/computed_tridiagonal_matrix[i-1][i-1];

     computed_tridiagonal_matrix[i][i] += - multiplicationFactor*computed_tridiagonal_matrix[i-1][i];

     computed_right_hand_side[i] += - multiplicationFactor*computed_right_hand_side[i-1];
  }

  for(int i = N-1; i > -1; i--){
```

```
      if(i == N-1)
         computed_numerical_solution[i] = computed_right_hand_side[i]/computed_tridiagonal_matrix[i][i];
      else
         computed_numerical_solution[i] = (computed_right_hand_side[i] -
             computed_tridiagonal_matrix[i][i+1]*computed_numerical_solution[i+1])
         /computed_tridiagonal_matrix[i][i];
   }
}
```

As can be seen from the code, the algorithm simply consist of two basic steps: one forward substitution and one backward substitution.

Now we want to calculate the number of floating points operations (FLOPS) needed to solve the linear system with the above algorithm. This is an extremely important topic, since the number of FLOPS typically will be very large in physics problems, and having an algorithm that reduces the FLOPS can be the difference between being able to solve the problem and not!

According to Wikipedia, the Thomas algorithm requires $\mathcal{O}(n)$ FLOPS. To check our understanding, we will try calculating the number of FLOPS ourself. Looking at the code above, we start by caunting FLOPS in the forward substitution, which is the first for loop. The variable 'mutliplicationfactor' equals a division, which counts as one FLOP. '`coputed_tridiagonal_matrix`' enveolves a sum of two terms, which gives one FLOP, and a multiplication, which gives another FLOP. So in total for '`computed_tridiagonal_matrix`' we have 2 FLOPS. Next the calculation of '`computed_right_hand_side`' envolves a multiplication (1 FLOP) and a summation (1 FLOP), giving 2 FLOPS for '`computed_right_hand_side`'. For each iteration in the loop, we have a total of 5 FLOPS. The loop goes from 1 to $N-1$, giving $N-1$ iterations. This, in total we have $5(N-1)$ FLOPS for the forward substitution.

On the backward substitution we have one multiplication (1 FLOP), one division (1 FLOP), and one subtraction (1 FLOP), ginving a total of 3 FLOPS per iteration. With $N-1$ iterations, we get $3(N-1)$ FLOPS on the backwards step.

Adding the FLOPS from the backwards step and the forwards step, we end with $8(N-1)$ FLOPS total, which is equal to $\mathcal{O}(N)$ FLOPS, which is the same as given on Wikipedia.

2.2.2 Gaussian elimination symmetric tridiagonal systems

If the tridiagonal system is symmetric, $A = A^T$, the Gaussian elimination method can be made even simpler than in the Thomas algorithm. The program below shows the algorithm and its implementation.

```
void gassianTridiagonalSymmetricSolver( double * computed_right_hand_side,
                          double * computed_numerical_solution, int N){
    double diagonal;
    double * f;
    f = new double[N];
    f[0] = computed_right_hand_side[0];
    diagonal = 2.;
    for (int i=1; i < N; i++){
        f[i] = computed_right_hand_side[i] + double(i)/(i+1.)*f[i-1];
    }
    diagonal = ((N-1)+2.)/((N-1)+1.);
    computed_numerical_solution[N-1] = f[N-1]/diagonal;
    for (int i = N-1; i >0; i--){
        computed_numerical_solution[i-1] = double(i)/(i+1.)*(f[i-1]
                            + computed_numerical_solution[i]);
    }

    delete [] f;
}
```

We can again calculate the FLOPS needed to solve the linear system with the above algorithm. For the forward substitution we have the variable f that contains a division, two summations and a multiplication. This leads to a flop count of $4(N-1)$ FLOPS. Outside of the two loops in the algorithm, we have to calculate the diagonal, that is needed to start of the backward substituion. The diagonal contains two summations and a division, giving us a count of three FLOPS. We need the last computed numerical solution to start of the backward substituion. It contains one division, giving one FLOP. In the backward substitution we have one division, two summations and a multiplication giving 4(N-1) FLOPS. In total we get $8(N-\frac{1}{2})$ FLOPS, which is equal to $\mathcal{O}(N)$ FLOPS.

2.2.3 LU

We will compare the speed of our algorithms with the speed of the LU-algorithm. According to Hjorth-Jensen [1] p. 173, solving a system with LU is $\mathcal{O}(n^2)$ requires FLOPS, but the composition itself requires $\mathcal{O}(n^3)$ FLOPS. Based on this, we expect our algorithms to outperform the LU-algorithm, independentlyu of whether one includes the composition itself in the comparison or not.

Here we will try ourself to compute the FLOP requirements for solving a system with LU. One starts by solving $Ly = w$ for y, where L is lower tridiagonal N X N with 1 on the diagonal. Seting up the first few iterations of the loop solving the system, we hopefully see a pattern:

$$y_1 = w_1 \tag{5}$$
$$y_2 = w_2 - L_{21}y_1 \tag{6}$$
$$y_3 = w_3 - L_{31}y_1 - L_{32}y_2 \tag{7}$$
$$\vdots \tag{8}$$
$$y_N = w_N - \sum_{i=1}^{N} L_{Ni}y_i \tag{9}$$

From the above, we see

$$y_i = w_i - \sum_{j=1}^{i} L_{ij}y_j \tag{10}$$

In terms of flop, the above gives

$$fl(y_i) = 1 - 2(i-1) = 3 - 2i \tag{11}$$

In total we get

$$fl(y) = \sum_{i=1}^{N} fl(y_i) \tag{12a}$$
$$\overset{(11)}{=} \sum_{i=1}^{N} 3 - 2i \tag{12b}$$
$$\overset{\sum_{i=1}^{N-1} i \approx n^2/2}{\approx} 3N - N^2 \tag{12c}$$
$$\approx \mathcal{O}(N^2) \tag{12d}$$

The approximation used in the 2nd last line, can be found in Watkins [2] (which according to the author is a book on numerical linear algebra in the gap between standard introduction texts and the numerical linear algebra texts by Trefethen and Bau, Goloub and Van Loan, and Demmel).

The next step in the algorithm for solving a system with LU, is to solve $Ux = y$ for y.

We skip some of the steps, and start with the following

$$x_i = \frac{(y_i - \sum_{j=i+1}^{N} A_{ij}x_j)}{A_{ii}} \tag{13}$$

We get

$$fl(x_i) \overset{13)}{=} 1 + 2(N - i) + 1 \tag{14}$$

$$= 2\bigg(N - i + 1\bigg) \tag{15}$$

In total we get

$$fl(x) = \sum_{i=1}^{N} 2\bigg(N - i + 1\bigg) \tag{16a}$$

$$\overset{\sum_{i=1}^{N-1} i \approx n^2/2}{\approx} 2\bigg(N^2 - N^2/2 + N\bigg) \tag{16b}$$

$$\approx \mathcal{O}(N^2) \tag{16c}$$

Now adding we see from (12d) and (16c) that LU uses $\mathcal{O}(N^2)$ FLOPS when solving the linear system.

2.3 Error calculations

Having knowledge about the errors of the numerical solutions is essential, because without this knowledge, it is difficult to know the validity of the results. Without knowledge of validity of the results, we are exiting the subject of (natural) science.

Leaving out probably the largest error source, our selves, there will be two possible sources of errors: truncation error due to numerical approximation of continuous operators (the 2nd derivative), and round-off errors due to limited computer precision. Below we will derive some expressions for these errors, giving us knowledge about what to expect from the numerical solutions.

2.3.1 Truncation error

In our differential equation, we approximate the 2nd derivative with a truncated Taylor-polynomial. A nice thing by using Taylor ploynomials is that we have knowledge about the truncation error, the difference between numerical and exact solution due to terms in the Taylor series being left out. It turns out that in our case the truncation error depends on the step size, h. Hence we know what to expect of the truncation error when varying h.

In our case the truncation error is of $\mathcal{O}(h^2)$. From this we expect the truncation error to depend on the step lenght, h, to a order of 2, meaning that a reduction of h by a factor of 10 should reduce the truncation error by a factor of 20.

2.3.2 Round-off errors

With double precision the computer is only accurate up to $\epsilon_{ro} = 10^{-16}$, where ϵ_{ro} stands for round of error. Now we will calculate an expression for the round-off error in our approximation of the 2nd derivative.

We approximate the 2nd derivative with

$$f'' = \frac{f_- + f_+ - 2f_0}{h^2} \tag{17}$$

$$= \frac{(f_+ - f_0) + (f_- - f_0)}{h^2} \tag{18}$$

For small h, the subtractions in the above expression is $(f_+ - f_0) \approx (f_- - f_0) \approx \epsilon_{ro}$. Insertion of this into the above equation gives

$$|f''| \leq \frac{2\epsilon_{ro}}{h^2} \tag{19}$$

For the truncation error we have the expression $\frac{f_0^{(4)}h^2}{12}$.

Using the expressions for truncation error and round-off error, we get the following expression for the total error

$$|\epsilon_{Total}| \leq \frac{2\epsilon_{ro}}{h^2} + \frac{f_0^{(4)}h^2}{12} \tag{20}$$

We see that the truncation error decreases in smaller h, while the round-off error increases. Since we are so luck having the analytical exprresion, we can calculate the truncation error. Being able to do this, we can find the optimal h, which minimizes the total error. Optimal h is found by differentiation of the above expression with respect to h. The differentiation yields

$$h^{optimal} = \frac{24\epsilon_{ro}}{f_0^{(4)}} \tag{21}$$

$$= \frac{24 \cdot 10^{-16}}{-10^4 e^{-10x}}, \tag{22}$$

where we in the last step have differentiated the expression for the anlytical solution four times. Now the optimal h depends on which value of x that is chosen. $x = 0$ gives $2 \cdot 10^{-5}$, while $x = 1$ gives $3 \cdot 10^{-4}$. From this we expect the optimal h to be found somewhere around $h \approx [10^{-5}, 10^{-4}]$.

2.3.3 Total error

We are more interested in the relative error, meaning the difference between the numerical and exact solution, relateive to the exact solution, than in the pure error. The reason for this, is that a small pure error could be physically insignificant if the quantities are big, while on the other hand a small pure error could be very significant if the quantities we are measuring are small. Example: 1 cm in error when simulating trajectories of planets is probably less physically significant than 1 cm error when doing simulations on molecular level.

For inspection of errors, and relating them to the expected convergence rates from Taylor series theory, it is neat to take the log the the errors and of the step lengths. Plotting this, the slopes could be directly compared with the expected convergence rate, the $\mathcal{O}$-term from the Taylor approximation of the 2nd derivative.

Our solutions will be vectors, so there will be errors at each vector element. We are interested in a rough error-number, so we want an error measure that represents all of the array-points. One way of doing this, is for each step length to extract the max value of the relative error. By this method, we know that the error we get is an upper bound for all local errors, local meaning errors at the different vector elements.

By the methods mentioned above, 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, and extract the maximum for each step length.

3 Results

3.1 Efficiency of the linear system solvers

The below table displays the times used by the different algorithms. Not that the LU-time includes also the time for the LU-decomposition, in addition to the time solving the system.

Thomas	Symmetric	LU
2.000000000000000E-06,	2.000000000000000E-06,	8.000000000000000E-06,
3.000000000000000E-06,	2.000000000000000E-06,	0.001498000000000000,
3.800000000000000E-05,	2.200000000000000E-05,	-
0.001763000000000000,	0.000163000000000000,	-
0.018432000000000000,	0.001550000000000000,	-

From the above table we that LU clearly uses more time than the other algorithms. This is in according to expectations, knowing that LU should be $\mathcal{O}(N^3)$ in FLOPS, while at least the tridiagaonal solver should be $\mathcal{O}(N)$ FLOPS.

The figure below may perhaps give a better impression on the times for the different algorithms, at least on how the results compare to the theoretical FLOP-estimations.

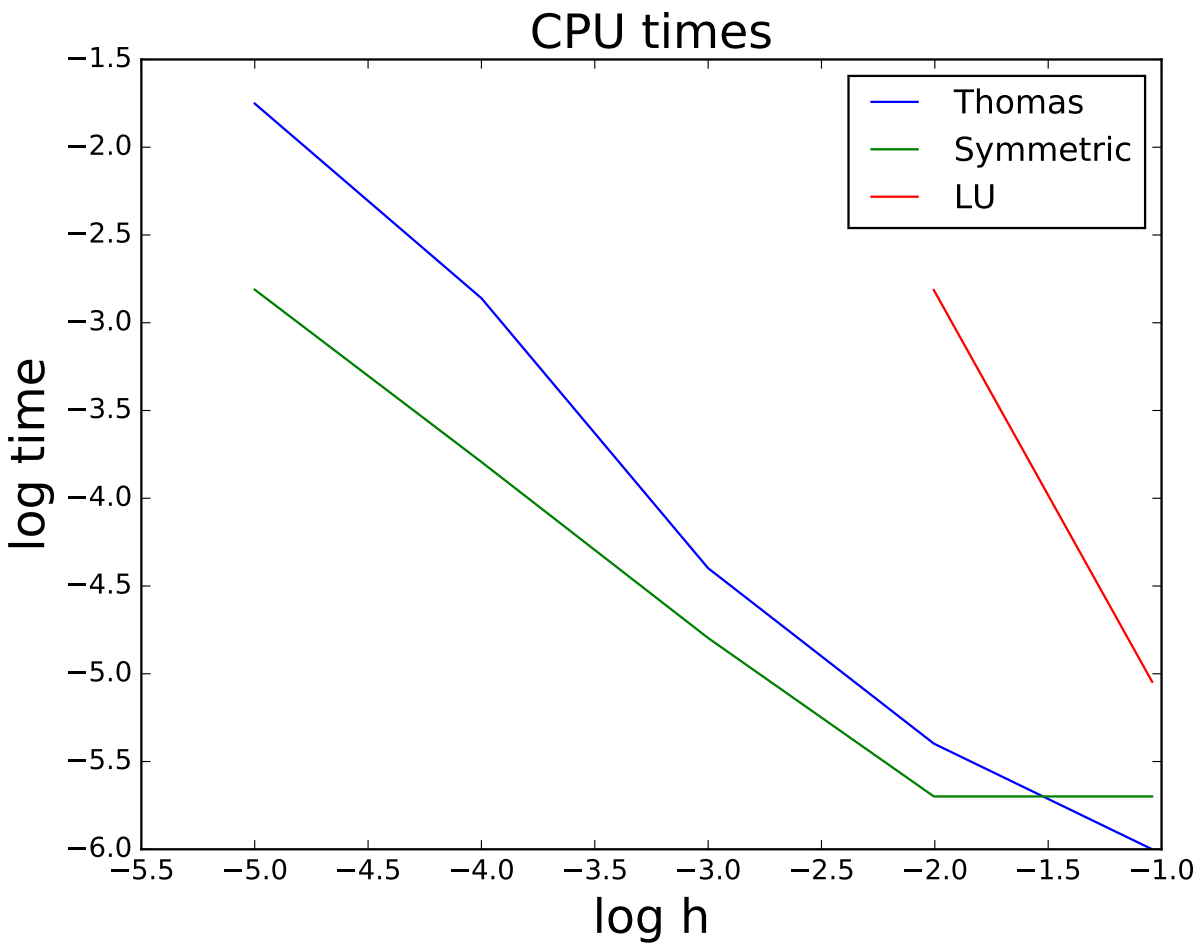

Figure 1: Log CPU-times

The figure above shows that the Thomas algorithm is $\mathcal{O}(N)$, the incline is about -1 in the figure. For the LU we get a slope somewhere between 2 and 3. Knowing that the decomposition is of order 3 and the solution of the system by LU is of order 2, we had expected an incline of 3 in our graph, since the largest N should dominate. We are not sure why we do not get a slope of 3 for LU.

Also the plot above for the Thomas algorithm and the simpplified algorithm does not correspond with our FLOP-calculations from the theory section. From thepry we had expected these to much closer.

A final note about the above figure is that the plots varied when we ran the simulations again. This we thought was very strange, but we guess the reason is that the state of the computer's CPU and memory are non-constant between different simulations, explaining the varying plots.

We were asked if it would be possible to run LU for $N = 10^5$. From before, we know that the Thomas algorithm, which is $\mathcal{O}(N)$, makes the computed crash for $N = 10^6$. Solving LU, which is $\mathcal{O}(N^3)$ for $N = 10^5$, would give FLOPS of order 10^15, which would never be solved on our computer.

3.2 Error analysis

The below table shows the errors for the different algorithms. It was expected that the algorithms should produce results equal to each other, whithin machine precision. All algorithms solves exactly the same system. The table below confirms our expectation.

Thomas	Symmetric	LU
-1.179697782181124,	-1.179697782181123,	-1.179697782181124,
-3.088036831552412,	-3.088036831558286,	-3.088036831552412,
-5.080051549975231,	-5.080051566878527,	-
-7.079285115156589,	-7.079267771744975,	-
-8.842972299542293,	-9.079091385416225,	-

Since the algorithms give the same error, we use only the results from one of the algorithms when analysing the error. The below figure shows the numerical results from the Thomas algorithm and the exact solution.
We see that except for the coarsest grid, $N = 10$, the fit is really good. Knowing that our approximator for the 2nd derivative is 2nd order, and the fact that the simulations in the figure has 10, 100 and 1000 grid points, it is not surprising that the error is reduced considerably when the grid is made 10 times finser in each step. With a convergence rate of 2, a ten fold refinement in the grid should lead to a 100 fold refinement in the truncation error.

Below follows figures for the relative error. Not that the scales on the x-axis on the two figures are not the same. The reason for different scale is that our computed crashed when we increased N tenfold from the last N in the right figure.

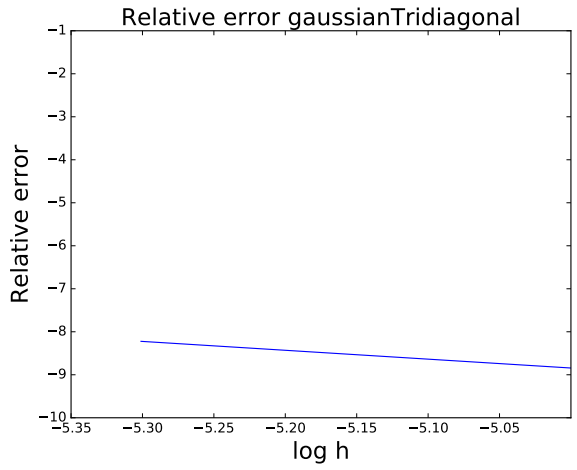

Figure 3: Results Tridiagonal Gaussian

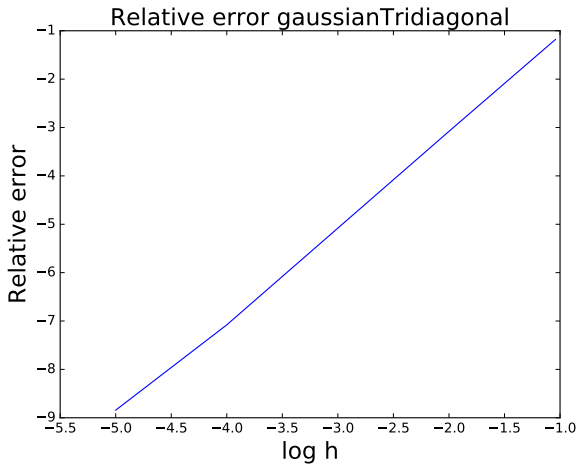

Figure 4: Results Tridiagonal Gaussian

Our computer crashes for $N = 10^6$, so the above figure to the right stops at $N = 10^5$. The figure shows that up to $N = 10^5$, the error seems to follow the expected truncation error, since the slope seems to be pretty close to 2.

The figure to the left above shows the error-development staring where the previous figure left off, but with smaller increases in N. The figures above suggest that the optimal h is around $h = 10^{-5}$, and this corresponds well with our expecations of optimal h in the area $[10^{-5}, 10^{-4}]$ derived in the theory section.

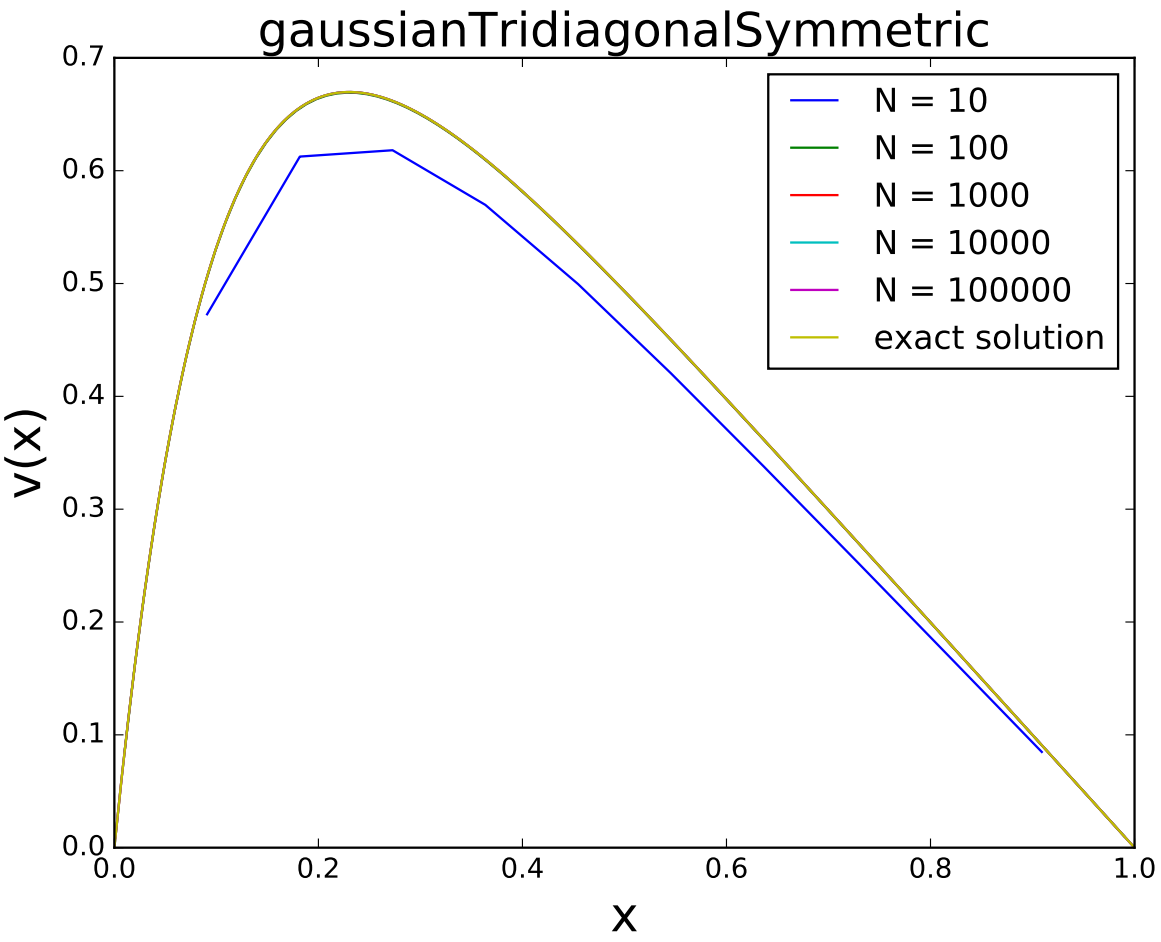

Figure 2: Results Tridiagonal Gaussian

We must however note that the results was not as clear for the symmetric algorithm. For the symmetric algorithm, the error still declined when we doubled the mesh refinement from the right to the left figure above. However, the slope was much lower than 2.

4 Conclusions

In this project we discretized the 1D Poisson equation, rewrote the problem to a linear system, and solved the linear system with three different algorithms in C++.

For all algorithms we calculated the number of FLOPS. The C++ simulations confirmed our theoretical FLOP calculations. Out of the three algorithms, we found that the symmetric algorithm was fastest, while the LU-decomposition was slowest.

Knowing the anaytical solution to our problem, we calculated the numerical error, represented by the sup-norm. We found that the numerical error was 2nd order up to a point, but that further mesh refinement after this point increased the error. All this is consistent with theoretical derivations we did for the total error.

5 Feedback

This project has been extremely educational. We learned about about c++, especially pointers and dynamic memory allocoation. Also which for us was a well forgotten subject, we learned about dangerous of numerical round-off errors.

We feel the size of the project is large, much larger than typical assignments in other courses. However, the quality and quantity of the teaching without a doubt made the workload managable. The detailed lectures, combined with the fast and good respones on Piazze helped a lot!

We think the project could have gone even smoother, if we on the 2nd lab-session had learned basic branching in Github. We used a considerable amount of time finding out of this.

All in all, two thumbs up!

6 Bibliography

[1] Hjorth-Jensen, M.(2015) *Computational physics. Lectures fall 2015.* `https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Lectures`

[2] Watkins, D.S.(2002) *Fundamentals of matrix computations. 2nd edition.*