Fys4150
Project 1

Peter Killingstad and Karl Jacobsen

September 4, 2017

Contents

1 Abstract

2 Introduction

Goals: Learn C++, dynamic memory allocation, discretizing differential equations and setting up a linear system, algorithms for solving tridiagonal linear systems, error-analysis (truncation and precision), efficiency analysis.

3 Theory

3.1 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

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

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

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

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 eauquation problem!

Task 1 We get to the linear system by setting in the first couple of values in the above equation with v's, and notice the pattern.

3.2 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-1] = 0;
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];
else
computed_numerical_solution[i] = computed_right_hand_side[i+1] -
    computed_tridiagonal_matrix[i][i+1]*computed_numerical_solution[i+1]/computed_tridiagonal_matrix[i][i];
}

}
```

Task 2 We can insert a few comments in the top of the function, explaining the algorithm.

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, which is equal to $\mathcal{O}(N)$ FLOPS, which is the same as given on Wikipedia.

3.3 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_tridiagonal_matrix, double *
    computed_right_hand_side, double * computed_numerical_solution, int N){
for (int i = 1 ; i < N; i++){
computed_tridiagonal_matrix[i][i] +=
    -computed_tridiagonal_matrix[i-1][i]*computed_tridiagonal_matrix[i-1][i]/computed_tridiagonal_matrix[i-1][i-1];
computed_right_hand_side[i] +=
    -computed_tridiagonal_matrix[i-1][i]*computed_right_hand_side[i-1]/computed_tridiagonal_matrix[i-1][i-1];
}

computed_numerical_solution[N-1] = computed_right_hand_side[N-1]/computed_tridiagonal_matrix[N-1][N-1];
for (int i = N-2; i > -1; i--){
computed_numerical_solution[i] = computed_right_hand_side[i] -
    computed_tridiagonal_matrix[i][i+1]*computed_numerical_solution[i+1]/computed_tridiagonal_matrix[i+1][i+1];
}
}
```

Task 4 The above listing must be made more pretty bby fixing the spaces, but that can wait tuil we know we have the final program. Also for this function, we should in the top add lines describing the algorithm.

Task 5 Here we should do, and show, FLOP-count for the above listed algorithm.

3.4 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.

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.

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 $i = 1, \dots, n$,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.

As an alternative we also computed the L2-error norm, which basically sums up all the local errors to one single measure. The reason for also including this norm, is that it can give some extra information about the overall performance of the numerical approximation over the whole domain.

3.5 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:

$$y_1 = w_1 \tag{2}$$
$$y_2 = w_2 - L_{21}y_1 \tag{3}$$
$$y_3 = w_3 - L_{31}y_1 - L_{32}y_2 \tag{4}$$
$$\vdots \tag{5}$$
$$y_N = w_N - \sum_{i}^{N-1} L_{Ni}y_i \tag{6}$$

Task 6 Insert code for importing LU from Armadillo and the commands used.

4 Results

4.1 Thomas algorithm

Task 7 Comment on the figure.

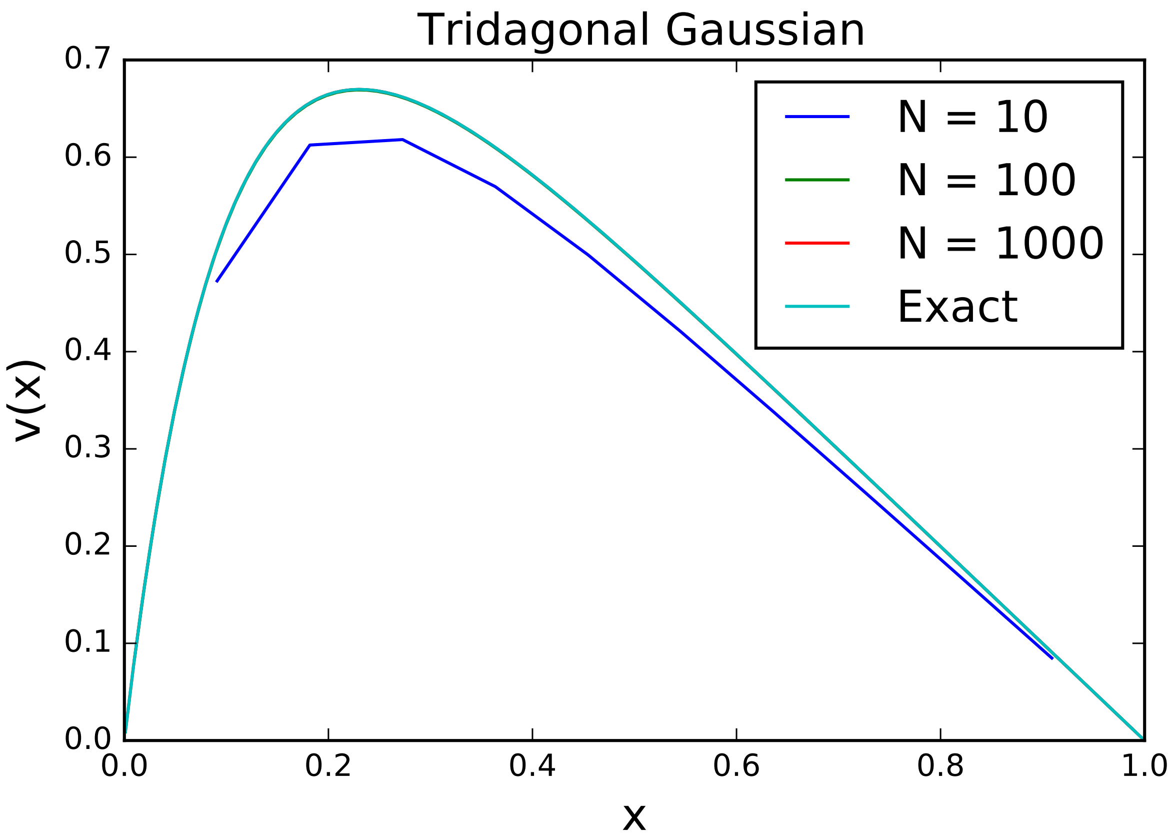

Figure 1: Results Tridiagonal Gaussian

4.2 Iterations times

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.

Tridiagonal	Symmetric	LU
3e-06	2e-06	9e-06
7e-06	6e-06	0.002699
0.000177	0.000107	1.551176
0.001408	0.001451	-

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.

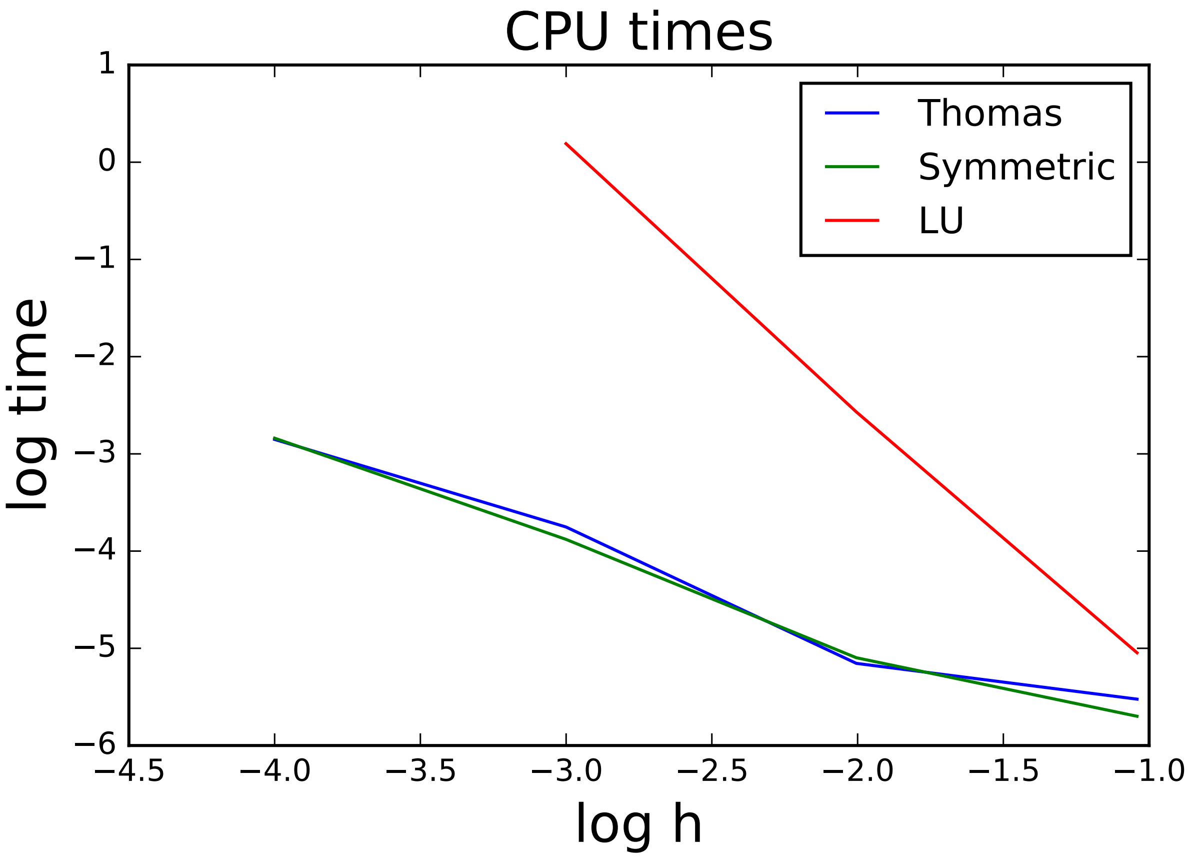

Figure 2: Log CPU-times

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.

4.3 Error

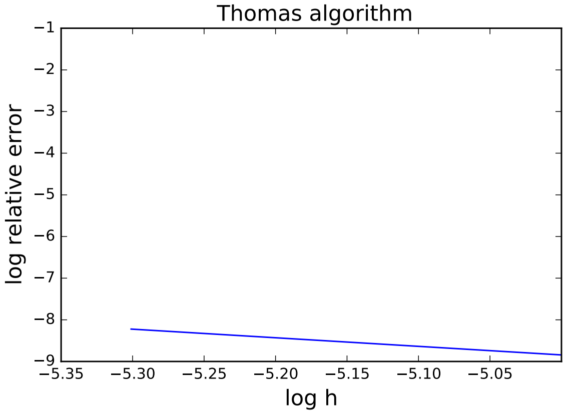

Figure 3: Results Tridiagonal Gaussian

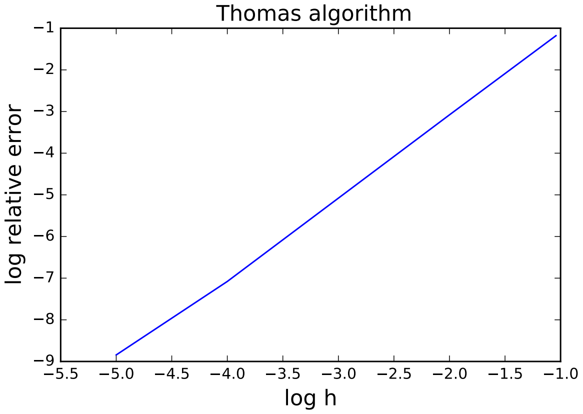

Figure 4: Results Tridiagonal Gaussian

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.

4.4 LU

Task 10 We compare the cpu time for LU with the CPU-times for the other two algorithms, for $N = 10, 100, 1000$.

Task 11 Copmute error for $N = 1000$ for LU and one of the other two algorithms. The results should be identical.
Task 12 LU possible $N = 10^5$? Find out.

5 Conclusions

Task 13 Sum up the results here.

6 Feedback

Task 14 What has been good, something that could have been done better?

7 Bibliography

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