Project 1

Tor-Andreas Bjone, Håvar Rinde Lund Jacobsen, Håkon Rinde Lund Jacobsen
(Dated: September 7, 2020)

In this project we have solved the one-dimensional Poisson equation by setting up a tridiagonal matrix equation. By applying Gaussian row reduction, we have derived a general and a special case algorithm for solving the linear equations. We have compared these methods in terms of efficiency (floating point operations, CPU-time and memory usage) among each other and to the LU-decomposition method for solving matrix equations. The special case algorithm was slightly faster than the general algorithm. LU-decomposition was the slowest among the methods, because of the simplicity of the problem.

I. INTRODUCTION

In this report we will look at numerically solving a set of n differential equations using matrix operations. We will start by setting up a general algorithm by using Jordan-Gauss elimination and then specialize it for our case with the second derivate. Last we will do a LU-decomposition. We will look at the different computing times and FLOPs(Floating Point Operations) for the different methods.

When dealing with large matrices, FLOPs becomes important to take into account, and specialized algorithms can sometimes be the only way to solve a set of differential equations if n gets to big. A large nxn matrix will take up a lot of space in the memory of a computer.

To validate our findings we will use the Poisson equation with the Dirichlet boundary conditions. The Poisson equation is

$$-\frac{d^2}{dx^2}u(x) = f(x), \tag{1}$$

where $f(x)$ is the value at a given x, and $u(x)$ is the closed-form analytical solution.

We let x be dimensionless, so that

$$0 \leq x \leq 1. \tag{2}$$

Then we give the Dirichlet boundary conditions.

$$u(0) = u(1) = 0. \tag{3}$$

II. METHOD

The second derivative of a function $u(x)$ can be discretized to a function $v(x)$ solve it numerically. Then

$$\frac{d^2}{dx^2}u(x) \approx -\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \tag{4}$$

for a solution $f(x)$ to the equation, and with step length

$$h = \frac{x_{n+1} - x_0}{n+2}.$$

We can then rewrite this to

$$-v_{i+1} + v_{i-1} - 2v_i = f_i h^2 = g_i,$$

where $g_i = h^2 f_i$. This can be represented by a matrix $A \in \mathbb{R}^{nxn}$, and vectors $\vec{g}$ and $\vec{v}$. We then get that

$$A\vec{v} = \vec{g}. \tag{5}$$

$$A = \begin{bmatrix} 2 & -1 & 0 & ... & 0 \\ -1 & 2 & -1 & 0 & ... \\ 0 & -1 & 2 & -1 & ... \\ ... & 0 & -1 & 2 & -1 \\ ... & ... & 0 & -1 & 2 \end{bmatrix} = \begin{bmatrix} b_1 & c_1 & 0 & ... & 0 \\ a_1 & b_2 & c_2 & 0 & ... \\ 0 & a_2 & b_3 & c_3 & ... \\ ... & 0 & a_{n-2} & b_{n-1} & c_{n-1} \\ ... & ... & 0 & a_{n-1} & b_n \end{bmatrix},$$

where $a_1 = a_2 = ...a_{n-1} = a_n = -1$, $b_1 = b_2 = ...b_{n-1} = b_n = 2$ and $c_1 = c_2 = ...c_{n-1} = c_n = -1$.

We start by solving this with Jordan-Gauss elimination. Our goal is then to a lower triangular matrix. When we do this on $A\vec{v} = \vec{g}$ we get the following algorithm:

$$\tilde{b}_i = \frac{\tilde{b}_{i-1}}{a_{i-1}} b_i - \tilde{c}_{i-1},$$

$$\tilde{c}_i = \frac{\tilde{b}_{i-1}}{a_{i-1}} c_i,$$

$$\tilde{g}_i = \frac{\tilde{b}_{i-1}}{a_{i-1}} g_i - \tilde{g}_{i-1},$$

with $\tilde{b}_0 = b_0$, $\tilde{c}_0 = c_0$ and $\tilde{g}_0 = g_0$. We let $i = 1, 2, ..., n-1$.

This is what we call the forward algorithm. This can then be solved by what we call the backward algorithm. We can use this because our matrix is lower triangular. We then get that

$$v_{j+1} = \frac{\tilde{g}_j - \tilde{c}_j \cdot v_{j+2}}{\tilde{b}_j}$$

, where we set $j = n - i$ and $i = 2, 3, 4, ..., n$ to make the solver go backward. Before we run the algorithm we solve element n: $v_n = \frac{\tilde{g}_{n-1}}{\tilde{b}_{n-1}}$.

When calculating this numerically it is beneficial to have as few CPU-operations as possible. We therefore count the number of FLOPs (Floating Point Operations). We can see that in the forward algorithm there is $8 \cdot n$ FLOPs and in the backwards algorithm there is $3 \cdot n$ FLOPs. So in the general solution we get a total of $11 \cdot n$ FLOPs.

We will check this algorithm by using the one dimensional Poisson equation with Dirichlet boundary conditions. The general Poisson equation in one dimension reads

$$-\frac{d^2}{dx^2}u(x) = f(x).$$

Then we put in the Dirichlet boundary conditions $u_0 = 0$, $u_{n+1} = 0$ and $x \in (0, 1)$. In our case we will assume that the source term is

$$f(x) = 100e^{-10x}.$$

Then the analytical solution becomes

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

We can then look at the relative error between the analytical solution and the numerical solution to validate our results.

Since $a_1 = a_2 = ... = a_n$, $c_1 = c_2 = ... = c_n$ and $b_1 = b_2 = ... = b_n$ we can simplify our algorithm to reduce the FLOPs. We call this the specialized algorithm.

$$\tilde{c}_i = \tilde{b}_{i-1},$$

$$\tilde{b}_i = -2\tilde{b}_{i-1} - \tilde{c}_{i-1},$$

$$\tilde{g}_i = -g_i \cdot \tilde{b}_{i-1} - \tilde{g}_{i-1}.$$

We can see that the forward solver now has $4 \cdot n$ FLOPs so that the specialized algorithm has a total of $7 \cdot n$ FLOPs.

To solve this numerically we add all the vectors and matrices using armadillo library[1] for c++.

We start by defining a vector x for the Dirichlet boundary condition and a vector v for the numerical solution. These vectors are $n + 2$ long.

We have the boundary conditions so we start by setting $v_0 = 0$ and $v_{n+1} = 1$. Then we make a function $f(x)$ to initialize vector g.

```
//f(x) test function
double matrix_solver::f(double x){
  return 100 * exp(-10*x);

//filling in values for g(x)
  for(int i=0;i<n;i++){
    g_vec(i) = pow(h,2) * f(x_vec(i+1));
  }
```

Vector g is of length n. We have to account for this when initializing it with x, so we use that $g_i = h^2 \cdot f_{i+1}$. We can then solve the forward and backward algorithm like this in c++:

```
void matrix_solver::forward_solver_general(){
  //Setting initial values for the tilde vectors
  b_tilde(0) = b_vec(0); c_tilde(0) = c_vec(0); g_tilde(0) = g_vec(0);

  //Solving the forward algorithm
  for(int i=1;i<n;i++){
    b_tilde(i) = b_vec(i)*b_tilde(i-1)*1./a_vec(i-1) - c_tilde(i-1);
    c_tilde(i) = c_vec(i)*b_tilde(i-1)*1./a_vec(i-1);
    g_tilde(i) = g_vec(i)*b_tilde(i-1)*1./a_vec(i-1) - g_tilde(i-1);
  }

void matrix_solver::backward_solver(){
  //Setting up the endpoint
  v_vec(n) = g_tilde(n-1)*1./b_tilde(n-1);

  //Solving the backward algorithm
  for(int i=2;i<n+1;i++){
    int j = (n-i);
    v_vec(j+1) = (g_tilde(j) - c_tilde(j) *v_vec(j+2))*1./b_tilde(j);
  }
```

And for the specialized solution we just change the $\tilde{b}$, $\tilde{c}$ and $\tilde{g}$ values. For both of the solutions we will look at different n values and find the computing time and relative error

$$\epsilon_i = log_{10}(|\frac{v_i - u_i}{u_i}|). \tag{6}$$

Then we will also do a LU-decomposition in armadillo and look at the CPU-times.

III. RESULTS

Table I. tabell

Floating point operations (FLOPs)			
Method	$n = 10$	$n = 100$	$n = 1000$
General ($11n$)	110	1100	11000
Special ($7n$)	70	700	7000
LU-decomposition ($(2/3)n^3$)	667	666667	666666667

Table (I) shows the number of FLOPs required for solving the linear equations using the different algorithms compared to the general LU-decomposition method. Because of the simplicity of this project, the general and special algorithms are the most efficient in terms of FLOPs and memory. The CPU-time for solving the equations also reflects this.

Figures (1), (2) and (3) shows the numerical solution and the analytical solution of the equation (1). For large n, these are in good alignment. Figures (4), (5) and (6) shows the same situation for the special algorithm. Figures (9), (10) and (11) shows the solution when using LU-decomposition on the matrix. For the same values of n, these different methods for solving

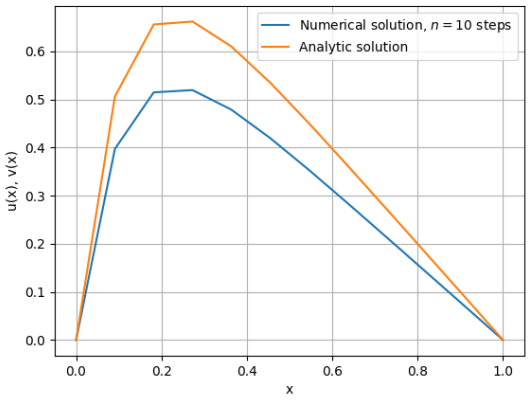

Figure 1. Solution of the general algorithm for $n = 10$

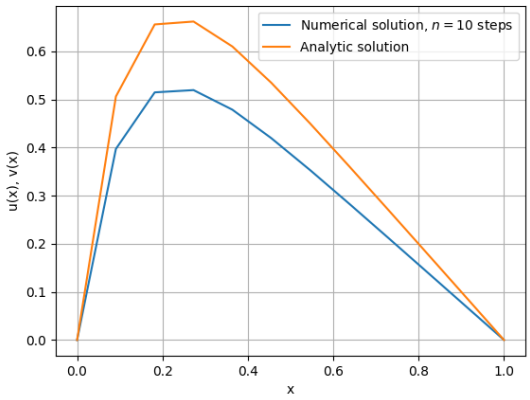

Figure 4. Solution of the specialized algorithm for $n = 10$

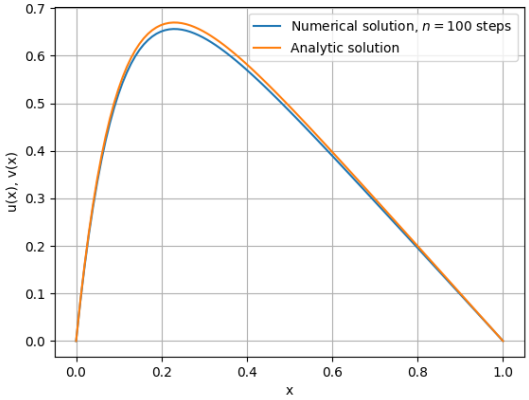

Figure 2. Solution of the general algorithm for $n = 100$

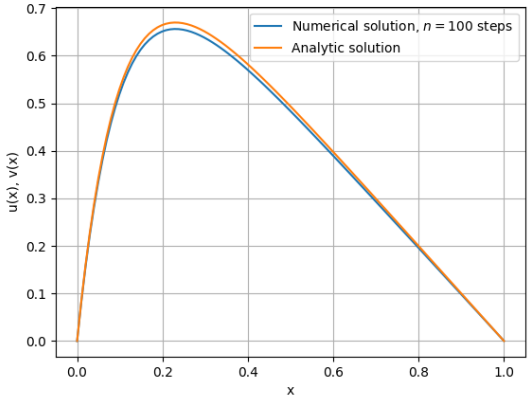

Figure 5. Solution of the specialized algorithm for $n = 100$

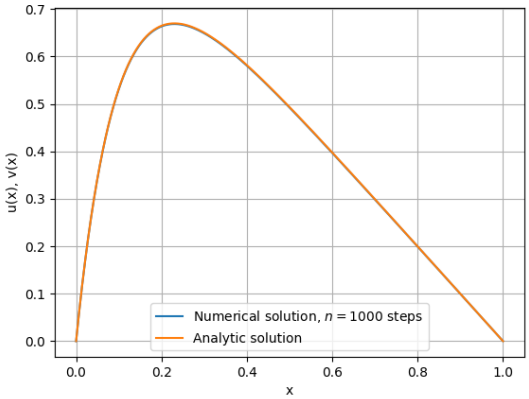

Figure 3. Solution of the general algorithm for $n = 100$

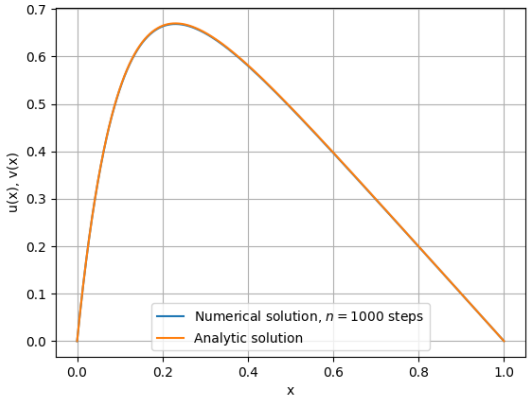

Figure 6. Solution of the specialized algorithm for $n = 1000$

the equation produces the same solution (no noticeable difference).

The relative error for the general algorithm and the special algorithm are shown in figures (7) and (8) on a logarithmic scale. As expected, the relative error

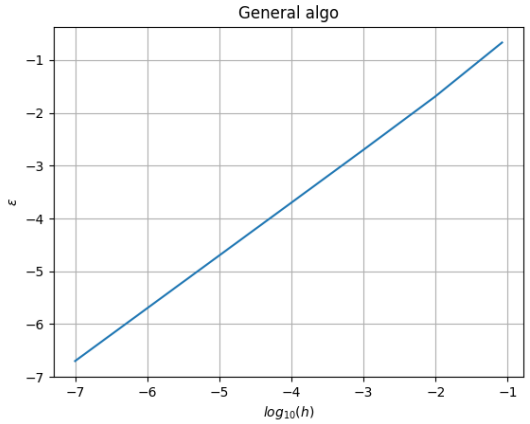

Figure 7. The relative error for the general algorithm.

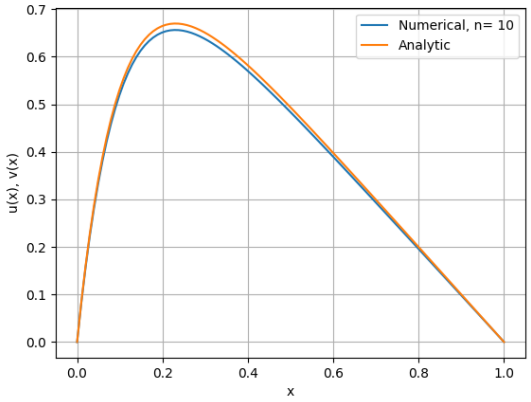

Figure 10. Solution of the LU-decomposition for $n = 100$.

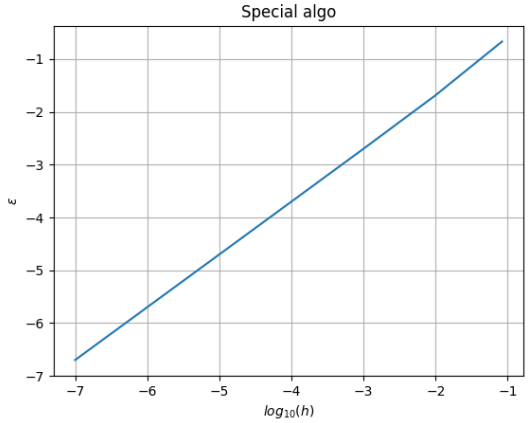

Figure 8. The relative error for the specialized algorithm.

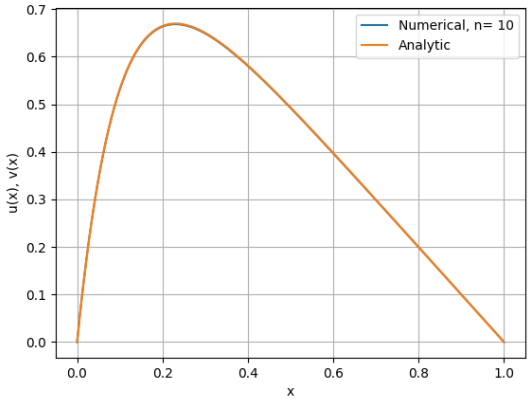

Figure 11. Solution of the LU-decomposition for $n = 1000$.

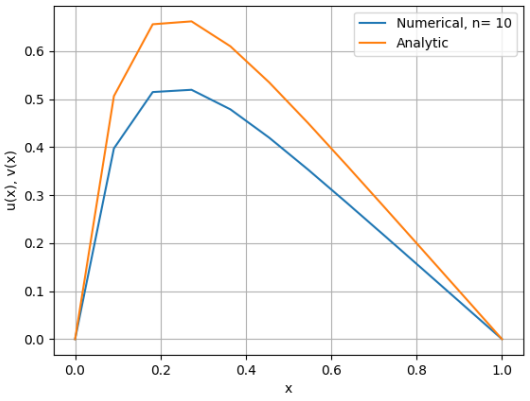

Figure 9. Solution of the LU-decomposition for $n = 10$.

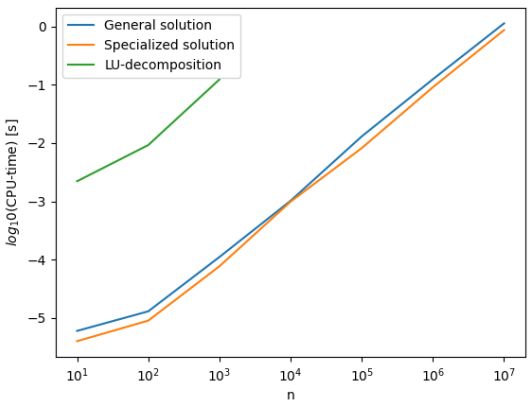

Figure 12. The CPU-time for the different solving methods. Taking the logarithm to see the lower values better.

reduces for smaller step sizes h (larger n equals smaller h). When calculating the relative error from equation (6), we have extracted the maximum value.

CPU-time usage (logarithmic) is shown in figure

(12) as a function of n. The special algorithm is consistently more efficient than the general algorithm, due to less FLOPs. Both algorithms were able to solve for $n = 10^7$. With the LU-decomposition however we were not able to exceed $n = 10^3$ due to memory issues. The CPU-time increases rapidly for large n.

IV. CONCLUSION

We have found numerical solutions to the one-dimensional Poisson equation (1) with Dirichlet boundary conditions (3) using the approximate of the second derivative (4) which are in good alignment with the closed-form analytic solution for $n > 100$. The special algorithm is the most efficient in the special case because of the reduction of FLOPs (less CPU-time and memory usage). The general algorithm is slightly slower, but has the advantage of being more accessible to solving other problems requiring other kinds of tridiagonal matrices. Because of the simplicity of this problem, the LU-decomposition is the slowest method for solving among those we have tested. In a more general case we would expect the LU-decomposition to be faster than the standard Gaussian row-reduction, which led us to the general and special algorithms.

When evaluating the error, we have extracted the maximum value from equation (6). As n grows large, the relative error reduces because it depends on the step size. The general and special algorithm does not require a lot of memory in order to solve the equations in this project, even when $n = 10^7$. This is not possible for the general LU-decomposition however.

REFERANSE

[1] Armadillo

APPENDIX

https://github.com/Trrn13P/FYS3150_project_1