FYS4150 Project 1

Eirik Thorsrud

September 10, 2018

1 Abstract

In this project we want to solve the one-dimensional Poisson equation numerically. By deriving a discretized approximation we are able to form a matrix which describes the approximated solution as a linear set of equations. As the matrix is tridiagonal, we are able to do a workaround to create an algorithm which is faster by using vectors for diagonal elements in the matrix instead of doing calculations with a matrix which is mostly made up by zeros. We will also look at specializing the algorithm to reduce the number of floating point operations.

2 Introduction

The equation we want to solve is

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0$$

We want to find $u(x)$ when $f(x)$ is given. First off, What we want to do is to define a discretized approximation to the second derivative of $u(x)$ with v_i. This is given by

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \tag{1}$$

With our approximation of the second derivative we form a linear set of equation on the form $\boldsymbol{Av} = \tilde{\boldsymbol{b}}$. Here $\boldsymbol{A}$ is a tridiagonal $n \times n$ matrix

$$\boldsymbol{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix}, \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} \tilde{\boldsymbol{b}} = \begin{bmatrix} \tilde{b_1} \\ \tilde{b_2} \\ \dots \\ \dots \\ \dots \\ \tilde{b_n} \end{bmatrix}$$

As we see multiplying $\boldsymbol{A}$ with $\boldsymbol{v}$ gives

$$2v_1 - v_2 = \tilde{b_1}, \qquad -v_1 + 2v_2 - v_3 = \tilde{b_2}, \qquad -v_2 + 2v_3 - v_4 = \tilde{b_3}$$

and so on. These are the same equations that we would get using (1). Here we have used that $\tilde{b_i} = h^2 f_i$. As we can see, we can transform our problem into an equation using matrix multiplication. Using C++, i will program a general solver for these set of equations using gaussian elimination on $\boldsymbol{A}$ to eliminate the 'sub diagonal' elements consisting of -1's beneath the diagonal consising of 2's. Then we can find a value for v_n. Having v_n we can find v_{n-1} and we can go all the way back to finding v_1 and we have our solution. This can also be done by defining vectors for the diagonal elements and will speed up the calculations which i will show in the following pages by comparing with a solution using LU decomposition. I will also check the relative errors of the resulting solution compared to the analytical solution as n increases.

3 Methods

3.1 Gaussian elimination, forward- and backward substitution

To create a general algorithm we suppose our tridiagonal matrix consists of unknown but equal elements over, under and along the diagonal. We call these c, d and e, so our matrix will look like this

$$\boldsymbol{A} = \begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & 0 \\ c_1 & d_2 & e_2 & 0 & \dots & \dots \\ 0 & c_2 & d_3 & e_3 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & c_{n-2} & d_{n-1} & e_{n-1} \\ 0 & \dots & \dots & 0 & c_{n-1} & d_n \end{bmatrix}$$

To be able to find the solution for the last element d_n we need to eliminate the c's under the diagonal. This is done by gauss elimination. We start off by adding a multiplicative of the first row to the second which will remove c_1. Multiplying the first row with $\frac{c_1}{d_1}$ and subtracting that from the second row eliminates c_1. Now since we have done this operation on one side of the equation, we need to do the same operation to our $\tilde{b}$ vector in our equation $\boldsymbol{Av} = \tilde{\boldsymbol{b}}$. After repeating this step through the whole matrix we remove the elements below the diagonal and are left with the same e's but our diagonal d's does now consist of c's e's. Now our matrix has the elements

$$\tilde{d_2} \to d_2 - \frac{e_1 c_1}{d_1}, \qquad \tilde{d_3} \to d_3 - \frac{e_2 c_2}{\tilde{d_2}}, \qquad \tilde{d_i} = d_i - \frac{e_{i-1} c_{i-1}}{\tilde{d}_{i-1}}$$

and our $\tilde{b}$ also gets changed in the following way

$$\tilde{b_i} \to b_i - \tilde{b}_{i-1} \frac{e_{i-1}}{\tilde{d}_{i-1}}$$

Now our equation looks like this

$$\begin{bmatrix} d_1 & e_1 & 0 & \dots & \dots & 0 \\ 0 & \tilde{d_2} & e_2 & 0 & \dots & \dots \\ 0 & 0 & \tilde{d_3} & e_3 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & \tilde{d_{n-1}} & e_{n-1} \\ 0 & \dots & \dots & 0 & 0 & \tilde{d_n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b_2} \\ \dots \\ \dots \\ \dots \\ \tilde{b_n} \end{bmatrix}$$

By using our modified matrix A and equally modified $\tilde{b}$ we can find a solution for v_n. This is simply $v_n = \frac{\tilde{b_n}}{\tilde{d_n}}$. From here, all we need to do is run our algorithm with indexes from $i = n-1$ to $i = 2$ to find all the solutions for v_i. The solutions for v_i is then given by

$$v_i = \frac{\tilde{b_i} - e_i v_{i+1}}{\tilde{d_i}}$$

The C++ program for the general solution is called Project1_general.cpp

3.2 Specializing the algorithm and number of floating point operations (FLOPS)

In our case, the diagonal elements are all equal. They consist of 2's in the diagonal and -1's over and under the diagonal. In the general algorithm we had the following operations for the forward substitution

$$\tilde{d_i} = d_i - \frac{e_{i-1}c_{i-1}}{\tilde{d}_{i-1}} \qquad \tilde{b_i} \to b_i - \tilde{b}_{i-1}\frac{e_{i-1}}{\tilde{d}_{i-1}}$$

The operations of the $\tilde{d_i}$ requires one multiplication, one division and one subtraction. Total number of FLOPS are $3(N-1)$ since the operation is done $N-1$ times. The same amount of FLOPS are used for the operations on $\tilde{b_i}$. This gives a total amound of $6(N-1) \sim 6N$ FLOPS for the forward substitution. The backward substitution, finding the solutions to v_i requires one multiplication, one division and one subtraction, therefore $\sim 3N$ FLOPS. Our total amount of FLOPS for the general algorithm is then $\sim 9N$.

We want to specialize our algorithm to our specific case. Since $d_i = 2$, $e_i = -1$ and $c_i = -1$, we can write our forward substitution as

$$\tilde{d_i} = 2 - \frac{1}{\tilde{d_{i-1}}} \quad \Rightarrow \quad \tilde{d_i} = \frac{i+1}{i}$$

$$\tilde{b_i} = b_i + \frac{\tilde{b}_{i-1}}{\tilde{d}_{i-1}}$$

And our backward substitution goes like

$$v_i = \frac{\tilde{b_i} + v_{i+1}}{\tilde{d_i}}$$

We have now reduced our floating point operations into $\sim 4N$ in the forward substitution and $\sim 2N$ in the backward substitution. A total of $\sim 6N$ FLOPS. *The C++ program for the special solution is called Project1_special.cpp*

4 Results and discussion

Figure 1 and 2 shows the solution to the algorithm with $n = 10$ and $n = 100$, where n decides the size of the matrix and vectors. As we can see from Figure 2 the solution fits good at $n = 100$.

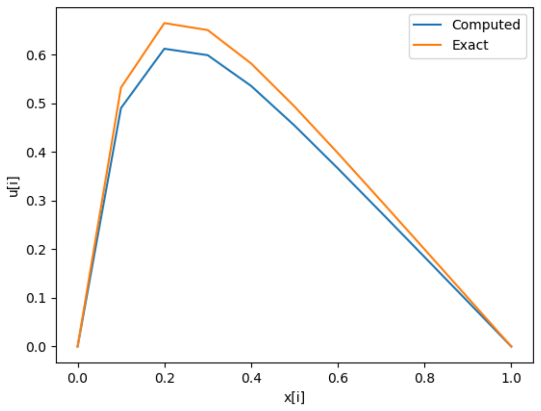

Figure 1: The solutions to v_i compared to the exact $u(x)$ with n = 10

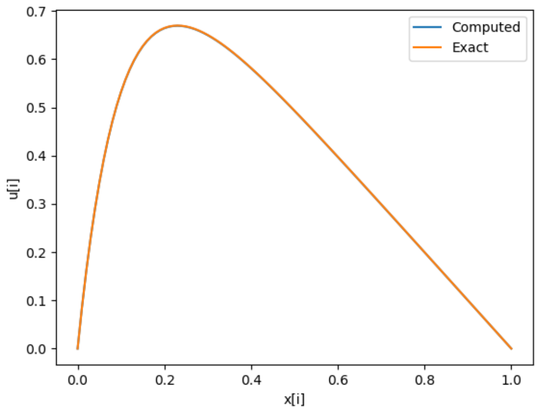

Figure 2: The solutions to v_i compared to the exact $u(x)$ with n = 100

In Methods section **3.2**, we made a specialized algorithm and reduced our number of FLOPS from $\sim 9N$ to $\sim 6N$. By comparing the time the CPU used to make the calculations in the general and specialized case with $n = 10^5$, 10^6 and 10^7 we get the results in table 1.

N	CPU time General case (seconds)	CPU time Special case (seconds)
10^5	0.003531	0.003172
10^6	0.036566	0.032564
10^7	0.385476	0.339343

Table 1: Comparing CPU time for general and special case

Each time we run the program, the CPU time varies, and table 1 shows six random tests. Taking and average of 10 runs we get 0.003645 seconds for the general case and 0.003234 for the special case for $n = 10^5$. We can clearly see that the special case has a slightly lower run time.

We can test how our code does in comparison to an LU decomposition done with the Armadillo library in C++. We test out for 10×10, 100×100 and 1000×1000 matrixes. By running our general solver the following output is made for the specific n's

10 $\times$ 10:
Tridiagonal general case took: 4e-06 Seconds
LU Decomposition took: 3.6e-05 Seconds

100 $\times$ 100:
Tridiagonal general case took: 7e-06 Seconds
LU Decomposition took: 0.001338 Seconds

1000 $\times$ 1000:
Tridiagonal general case took: 3.8e-05 Seconds
LU Decomposition took: 0.111587 Seconds

As we can see from the outputs above, our tridiagonal general solver is a lot more faster than the armadillos LU decomposition. This has to do with the fact that our matrix has identical elements along the diagonals which makes us able to use vectors for the diagonals instead of the whole matrix.

We can now take a look at the relative errors related to each n. For each calculated result we have an error compared to the exact solution. Taking the maximum of all the relative errors for each n we are able to make a log-log plot which is shown in Figure 3. As we see as h is getting smaller the relative error decreases. This goes on to a point where the relative error is getting larger at $h = 10^{-7}$. The reason for the loss of numerical precision may be due to the computers representation of numbers. When the error gets so small that the computer can not represent the error right, it can make the error rise.

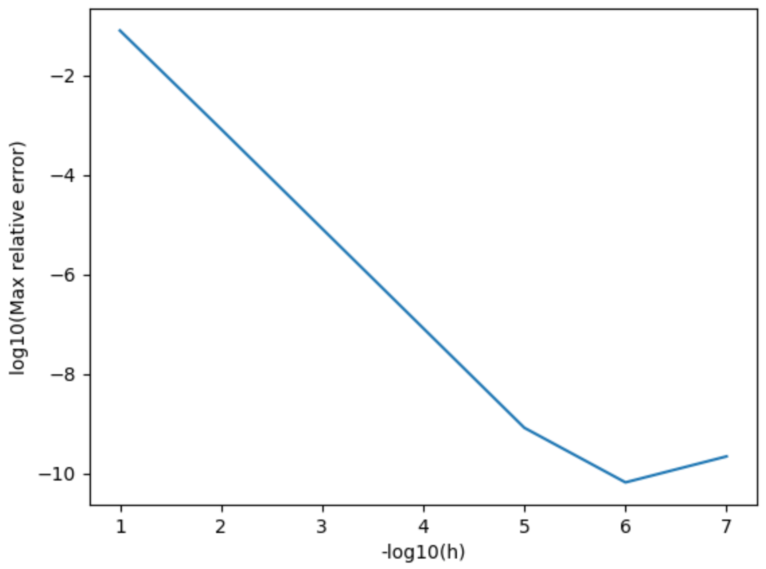

Figure 3: The logarithm of the relative error compared to the logarithm of stepsizes h

5 Conclusion

Starting with a discrete solution of the second derivative of a function, we have shown how we can set up an equation using a matrix $\boldsymbol{A}$. Starting off with a general solver, we reduced the number of floating point operations by specializing our code for our special case. Our special solver showed a slightly reduced CPU time compared to the general solver. Compared to the armadillos LU solver, our solver did the calculations faster by a factor of 10^3. When it comes to relative errors in the calculated solutions compared to the exact solution, there seemed to be a minimal with a stepsize $h = 10^{-6}$ which corresponds to an $n = 10^6$.

6 Appendix

Project1_general.cpp
Project1_special.cpp
Project1_general.o
Project1_special.o
terminal_outputs.txt