Project 1

Ragnhild Wisted
FYS3150, fall semester 2020
(Dated: September 10, 2020)

We have developed two tridiagonal matrix algorithms for solving the one-dimensional general Poisson equation, one general and one specific. We then used these algorithms to study the correlation between the number of FLOPs required and the CPU time, and compared them to a LU decomposition method. The general algorithm required 1.5 times more FLOPs than the specific, and this was reflected in our results. For $n = 10^7$, the CPU times for the general solver was $1.28924 \cdot 10^6$ microseconds, and for the specified solver it was $2.89226 \cdot 10^5$ microseconds. While we expected the numerical precision to decrease as $n > 10^4$, this did not happen in our calculations. This is suspicious, and might indicate that something has gone wrong in our code.

I. INTRODUCTION

The second derivative of a function $f(x)$ can be expressed as

$$f''(x) = \lim_{h \to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \tag{1}$$

where h is the step length between derivation points. Mathematically we would say that as h becomes infinitely smaller, the approximation becomes more accurate. Nowadays however, these kinds of calculations are usually done on computers, and there is a limited amount of numbers that can be represented exactly in a computer. This leads to round-off errors, which will accumulate when the number of integration points on a given interval increases, i.e. when the step length h decreases. This means that paradoxically, as the theoretical mathematical precision increases, we see an increasing loss of numerical precision. In other words, just decreasing the step length h will not necessarily lead to more accurate results.

The goal of this project is explore how numerical errors can accumulate, while also becoming more familiar with concepts like pointers and memory allocation in C++, how the number of floating point operations (FLOPs) impact CPU time, and getting started with writing programs in C++. To achieve this, we will study the general one-dimensional Poisson equation,

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

We will solve this for $x \in [0, 1]$, with the boundary conditions $u(0) = u(1) = 0$. We assume that the source term is

$$f(x) = 100e^{-10x}. \tag{3}$$

This means that equation 2 has a closed-form solution given by

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}. \tag{4}$$

We will write two algorithms for solving $u(x)$, one general and one for our specific case. To check if our algorithms work as intended we will plot the numerical solutions with the exact solution. The plots will be presented in section III.

We will then compare the maximum relative error of the numerical calculations from our two algorithms and study the correlation between the step length h and the maximum value of the relative error.

Finally we will then study whether the number of floating point operations (FLOPs) required in two different algorithms impact the CPU time, and compare them to a LU decomposition method using the two functions **ludcmp()** and **lubksb()** from the external library **lib.cpp** found at the course's github repository.[1]

II. METHOD

A. Discretize the equation

We define the discretized approximation to u as v_i, and use equation 1 to approximate the second derivative of u with the following three point formula,

$$-v_i'' = -\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \qquad \text{for } i = 1, \ldots, n, \tag{5}$$

where $f_i = f(x_i)$. This discretized approximation has n grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$, with the step size being defined as $h = 1/n$. Our previous boundary conditions have now become $v_0 = v_{n+1} = 0$.

We multiply equation 5 with h^2:

$$v_{i+1} + v_{i-1} - 2v_i = h^2 f_i. \tag{6}$$

We rename the above right side g_i:

$$h^2 f_i = g_i. \tag{7}$$

Now we see that we end up with a set of equations

$$\begin{aligned} i = 1 &\rightarrow v_2 - 2v_1 + u_v = g_1 \\ i = 2 &\rightarrow v_3 - 2v_2 + v_1 = g_2 \\ &\vdots \\ i = n &\rightarrow v_{n+1} - 2v_n + v_{n-1} = g_n \end{aligned} \tag{8}$$

We remember that $v_0 = v_{n+1} = 0$.

Now define a tridiagonal Töplitz matrix $\mathbf{A}$, which we see arises from the discretization of the second derivative:

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & 0 & 0 & 1 & -2 \end{bmatrix}$$

Now we define two vectors

$$\mathbf{v^T} = [\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}] \tag{9}$$

and

$$\mathbf{g^T} = [\mathbf{g_1}, \mathbf{g_2}, \ldots, \mathbf{g_n}] \tag{10}$$

Now we see that equations 6 is equivalent to

$$\mathbf{Av} = \mathbf{g}. \tag{11}$$

B. The general algorithm

Now we want to create an algorithm for solving this linear set of equations. We begin by looking at $\mathbf{A}$ for the case $n = 4$:

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{bmatrix}$$

We see that all the information we need can be stored in two vectors $\mathbf{d}$ and $\mathbf{e}$:

$$\mathbf{d^T} = [\mathbf{d_1}, \mathbf{d_2}, \ldots, \mathbf{d_n}] \tag{12}$$

$$\mathbf{e^T} = [\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_{n-1}}] \tag{13}$$

Written out explicitly, $\mathbf{Av} = \mathbf{g}$ is then

$$\begin{aligned} d_1 \cdot v_1 + e_1 \cdot v_2 + 0 + 0 &= g_1 \\ e_1 \cdot v_1 + d_2 \cdot v_2 + e_2 \cdot v_3 + 0 &= g_2 \\ 0 + e_2 \cdot v_2 + d_3 \cdot v_3 + e_3 \cdot v_4 &= g_3 \\ 0 + 0 + e_3 \cdot v_3 + d_4 \cdot v_4 &= g_4 \end{aligned} \tag{14}$$

If we leave out $\mathbf{v}$ this can be written as a matrix

$$\begin{array}{cccc|c} d_1 & e_1 & 0 & 0 & g_1 \\ e_1 & d_2 & e_2 & 0 & g_2 \\ 0 & e_2 & d_3 & e_3 & g_3 \\ 0 & 0 & e_3 & d_4 & g_4 \end{array}$$

After Gaussian elimination this matrix has the form

$$\begin{array}{cccc|c} d_1 & e_1 & 0 & 0 & g_1 \\ 0 & \tilde{d}_2 & e_2 & 0 & \tilde{g}_2 \\ 0 & 0 & \tilde{d}_3 & e_3 & \tilde{g}_3 \\ 0 & 0 & 0 & \tilde{d}_4 & \tilde{g}_4 \end{array}$$

where the new vectors $\tilde{\mathbf{d}}$ and $\tilde{\mathbf{g}}$ are found by forward substitution:

$$\begin{aligned} \tilde{d}_i &= d_i - e_{i-1}^2 / \tilde{d}_{i-1} \\ \tilde{g}_i &= g_i - e_{i-1} \cdot \tilde{g}_{i-1} / \tilde{d}_{i-1} \end{aligned} \tag{15}$$

with initial conditions $\tilde{d}_1 = d_1$ and $\tilde{g}_1 = g_1$.

The solution u_i can then be found by using backwards substitution:

$$u_i = (\tilde{g}_i - e_i u_{i+1}) / \tilde{d}_i. \tag{16}$$

Our complete general algorithm (which is a version of the Thomas algorithm) is then

$$\begin{aligned} \tilde{d}_i &= d_i - e_{i-1}^2 / \tilde{d}_{i-1} \\ \tilde{g}_i &= g_i - e_{i-1} \cdot \tilde{g}_{i-1} / \tilde{d}_{i-1} \\ v_i &= (\tilde{g}_i - e_i v_{i+1}) / \tilde{d}_i \end{aligned} \tag{17}$$

These three steps require $3(n-2)$ (as the initial conditions $v_0 = v_{n+1} = 0$ are known) FLOPs each (one subtraction, one multiplication and one division). When n is a high number, this is $\sim 3n$ per step, for a total of $\sim 9n$ FLOPs.

C. The spesific algorithm

In the case of our specific triangular matrix, we know that all entries $e_i = 1$. This means that $\tilde{d}_i$ is simply found by $\tilde{d}_i = d_i - 1/\tilde{d}_{i-1}$. This has an analytical expression,

$$\tilde{d}_i = -\frac{(i+1)}{i}. \tag{18}$$

$\tilde{\mathbf{g}}$ is then found by the now simplified forward substitution,

$$\tilde{g}_i = g_i - \tilde{g}_{i-1}/\tilde{d}_{i-1} \tag{19}$$

with the initial condition $\tilde{g}_1 = g_1$. The backwards substitution is then

$$v_i = (\tilde{g}_i - v_{i+1})/\tilde{d}_i \tag{20}$$

If $\tilde{\mathbf{d}}$ was pre-calculated, this would bring the number of FLOPs down to $4(n-2) \sim 4n$. However, in our program we ended up calculating $\tilde{d}_i$ in each iteration, and thus the total number of FLOPs is $\sim 6n$.

D. Evaluating the relative error

We have an analytical solution to our equation, which means that we can use the computed solution and the exact solution to evaluate the relative error ϵ in our solvers:

$$\epsilon = \left| \frac{u_{\text{computed}} - u_{\text{exact}}}{u_{\text{exact}}} \right| \tag{21}$$

We compute the logarithm of the relative error,

$$\epsilon_i = \log_{10}\left(\left| \frac{v_i - u_i}{u_i} \right| \right), \tag{22}$$

and study it as a function of $\log_{10}(h)$. Here, v_i is the computed solution and u_i is the exact, analytical solution. We choose to use the specific solver for this. The results are presented in table I.

E. Implementation

In our program we defined six vectors for the general solver, $\mathbf{g}$, $\tilde{\mathbf{g}}$, $\mathbf{d}$, $\tilde{\mathbf{d}}$, $\mathbf{e}$ and $\mathbf{u}$. The specific solver only needed four, namely $\mathbf{g}$, $\tilde{\mathbf{g}}$, $\tilde{\mathbf{d}}$, and $\mathbf{u}$. We created a function for the general and one for the specific algorithm. In order to compare the numerical results with the exact solution (equation 4), we created a function to write the following variables to a data file: n, $\log_{10}(h)$, x_i, the exact solution $u(x_i)$, the general solution $v_{general}(x_i)$, its maximum relative error $\epsilon_{general}$, the specific solution $v_{specific}(x_i)$, its maximum relative error $\epsilon_{specific}$, the solution from the LU decomposition method v_{LU}, and its maximum relative error ϵ_{LU}. This was done to be able to plot our results with matplotlib in python.

The CPU time of each method was found by using the chrono function

`high_resolution_clock,`

and the time was found in microseconds (ms). In order to get a more accurate sense of the CPU time, we ran the calculations 10 times and used the median time. To find the median, we used a pre-made function to find the median of an unsorted array.[2] To spending unnecessary time writing to file, we made sure that only the final run wrote the data to file.

III. RESULTS

The results of comparing our numerical solutions to the exact solution of $u(x)$ are shown in figures 1, 2 and 3. The maximum relative error ϵ is presented as a function of $\log_{10}(h)$ in table I. The different CPU times are shown in table II.

n	$\log_{10}(h)$	Maximum ϵ
10	-1.0	0.327572
10^2	-2.0	0.037856
10^3	-3.0	0.003853
10^4	-4.0	0.000386
10^5	-5.0	3.9e-05
10^6	-6.0	4e-06
10^7	-7.0	1e-06

Table I. Maximum value of the relative error ϵ as a function of the step size h. The general algorithm was used to extract the values.

n	CPU time LU decomposition [ms]	CPU time general solver [ms]	CPU time specified solver [ms]
10	0	0	0
10^2	$1.985 \cdot 10^3$	0	0
10^3	$2.91087 \cdot 10^6$	0	0
10^4	-	$4.965 \cdot 10^2$	0
10^5	-	$6.483 \cdot 10^3$	$2.9895 \cdot 10^3$
10^6	-	$5.44835 \cdot 10^4$	$2.8424 \cdot 10^4$
10^7	-	$1.28924 \cdot 10^6$	$2.89226 \cdot 10^5$

Table II. CPU time for the different solvers, given in microseconds.

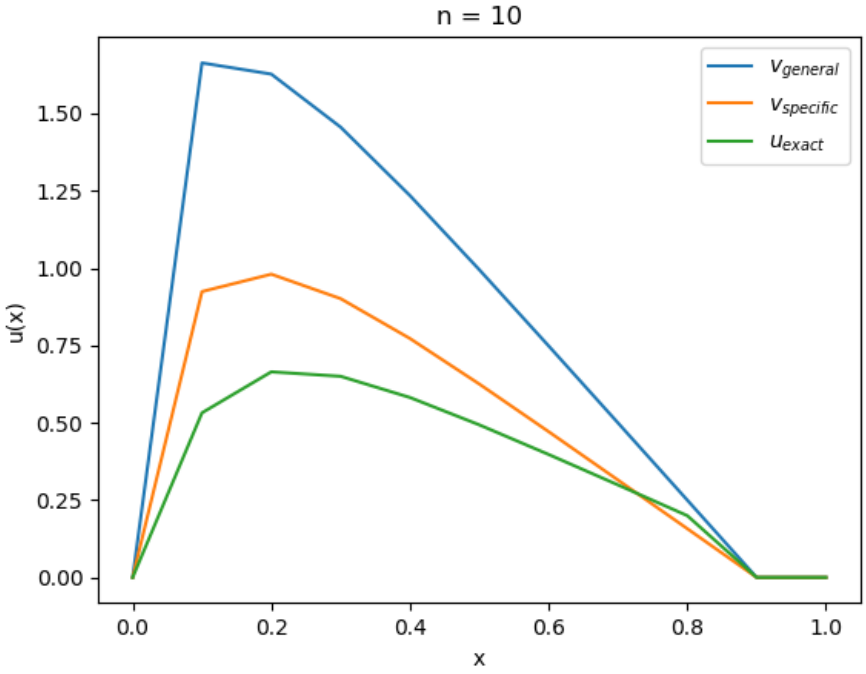

Figure 1. Comparison of the numerical and exact solution with $n = 10$.

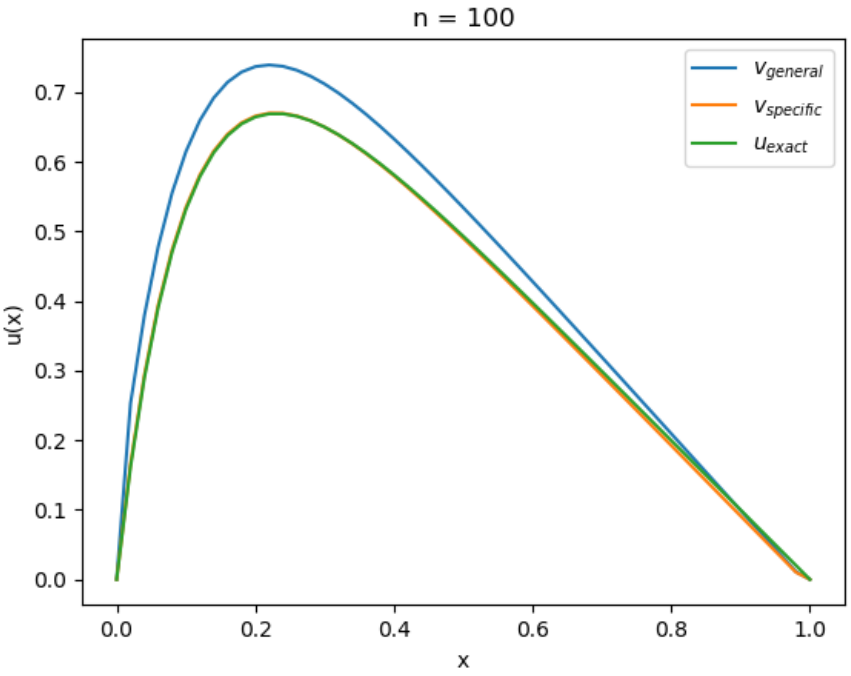

Figure 2. Comparison of the numerical and exact solution with $n = 10^2$.

IV. DISCUSSION/CONCLUSION

From the plots of the numerical and analytical solutions of the equation, we can see that both our algorithms are good approximations, especially for $n \geq 1000$. In figure 1 something happens that makes all the functions go to 0 one step too early, this could be a mistake in the method for extracting the data.

The relative error ϵ as function of step size h in figure 4 is unexpected. What we expect is that the relative error has its minimum around $\log_{10}(h) = -4$, and increases for smaller step lengths due to accumulation of numerical imprecision. In our plot the relative error appears to stay small for increasingly smaller values of h. This is a bit worrying, and might indicate that something is wrong in our code. From table I we see that ϵ behaves as expected until $n = 10^4$, but for the higher values of n they seem too small.

As for the CPU time, we see that the LU decomposition method, which requires an entire $n \times n$ matrix, is substantially slower than our solvers. In our code, the LU decomposition method was not viable for $n \geq 10^4$. We also see that of our methods, the specific solver (which requires less FLOPs) is faster than the general solver.

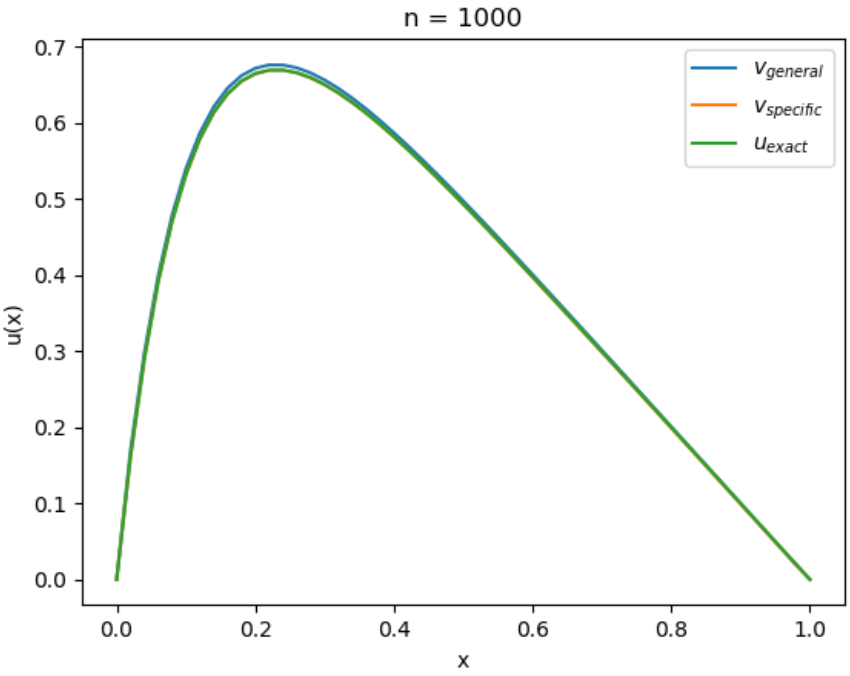

Figure 3. Comparison of the numerical and exact solution with $n = 10^3$.

REFERENCES

[1] Course git repository: https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Programs/LecturePrograms/programs/LinAlgebra/cpp
[2] Median function: https://xoax.net/cpp/ref/cpp_examples/incl/mean_med_mod_array/

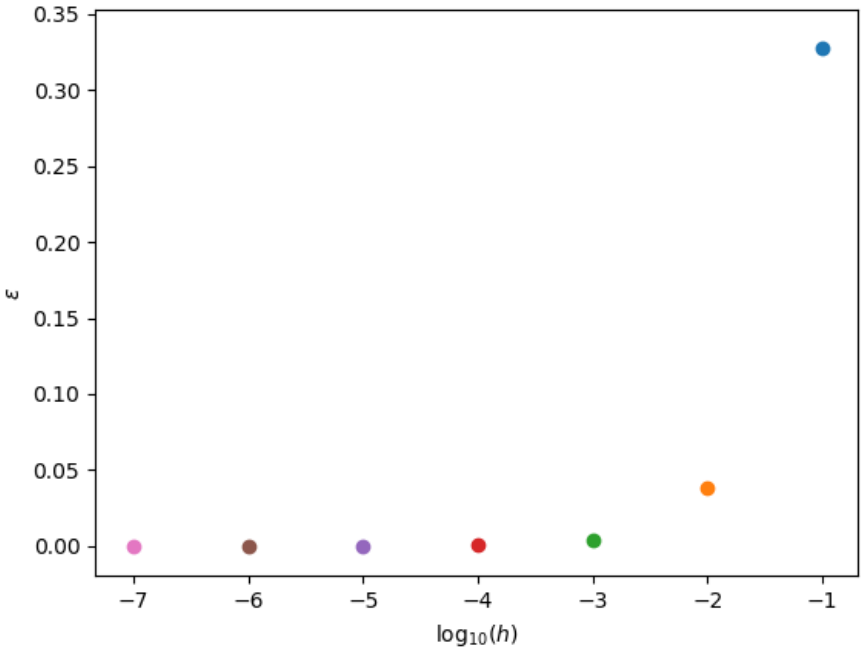

Figure 4. Logarithmic plot of the maximum relative error *epsilon* as a function of h.