Analysis of algorithms for solving one-dimensional Poisson's equation

Project 1, FYS4150
Autumn 2019

Jowita Borowska

ABSTRACT

The following project focuses on solving one-dimensional Poisson's equation (rewritten as a set of linear equations) with Dirichlet boundary conditions. We take three different approaches: the general tridiagonal matrix solver via Thomas algorithm, specialized algorithm (employing the fact that the diagonal elements of the coefficient-matrix are identical) and LU decomposition of the matrix. One of the aspects of this project is finding the most efficient method, which appears to be the specialized algorithm, associated with the shortest CPU time (on average) and the smallest number of floating-point operations needed to perform computations ($4n$). Error-analysis based on comparison between the numerical and analytical solution shows that careless increasing the number of grid points, n, eventually leads to loss of numerical precision (which happens around $n \approx 10^5$ for the general algorithm).

1 Introduction

The Poisson's equation is widely used in many branches of both engineering and physics, for instance to describe the potential arising from a localized charge or mass density distribution. Mathematically, its general one-dimensional form is given by

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

where f is the source term (inhomogeneous term), here assumed to be $f(x) = 100e^{-10x}$. We define domain of the function as $x \in (0, 1)$ and apply Dirichlet boundary conditions, so that $u(0) = u(1) = 0$. This gives the analytical solution on the form $u(x) = 1-(1-e^{-10})x-e^{-10x}$.

In the following project the Poisson's equation will be solved numerically and subsequently compared to the analytical closed-form solution. We will apply three different approaches of solving linear sets of equations (explained thorougly in the Method section) and discuss their accuracy, as well as the important aspect of efficiency (Results and discussion).

The main program (project1_main.cpp) is written in C++, while all the plots included in this report are made with use of Python-code that reads output files generated by the main program (project1_read_plot.py). The files can be found in the Github repository: github.com/jowborowska/CompPhysics

2 Methods

2.1 Discretization and linear set of equations

Discretization is required in order to solve Eq. (1) numerically. We use grid points $x_i = ih$ $(i = 0, 1, ..., n+1)$ on the interval from $x_0 = 0$ to $x_{n+1} = 1$, which corresponds to the spacing equal $h = 1/(n+1)$ and source term discretized as $f_i = f(x_i)$. Furthermore, $u(x_i)$ becomes v_i, so that the boundary conditions translate to $v_0 = v_{n+1} = 0$ and the second derivative (for i = 1,2,...,n) can be approximated as

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

which becomes

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

$$\begin{aligned}
2v_1 - v_2 &= h^2 f_1 \quad \text{for } i = 1, \\
-v_1 + 2v_2 - v_3 &= h^2 f_2 \quad \text{for } i = 2, \\
-v_{n-2} + 2v_{n-1} - v_n &= h^2 f_{n-1} \quad \text{for } i = n-1, \\
-v_{n-1} + 2v_n &= h^2 f_n \quad \text{for } i = n.
\end{aligned}$$

This linear set of equations can clearly be written in the form of matrix-vector multiplication, where our solution becomes the vector, $\vec{v} = [v_1, v_2, ..., v_n]^T$, and the coefficients in front of its elements, v_i, are stored in the tridiagonal, $n \times n$ matrix, so that

$$\begin{bmatrix}
2 & -1 & 0 & \dots & \dots & 0 \\
-1 & 2 & -1 & 0 & \dots & \dots \\
0 & -1 & 2 & -1 & 0 & \dots \\
 & \dots & \dots & \dots & \dots & \dots \\
0 & \dots & & -1 & 2 & -1 \\
0 & \dots & & 0 & -1 & 2
\end{bmatrix}
\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix}
=
\begin{bmatrix} h^2 f_1 \\ h^2 f_2 \\ \dots \\ \dots \\ \dots \\ h^2 f_n \end{bmatrix}.$$

We name the above matrix $\mathbf{A}$ and elements $h^2 f_i = \tilde{b}_i$, so equation takes the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}. \tag{2}$$

In general, we can rewrite the matrix $\mathbf{A}$ in terms of one-dimensional vectors. Here $\vec{b}$ contains elements of the main diagonal, $\vec{a}$ - the first diagonal below this, and $\vec{c}$ - the first diagonal above the main one (all the remaining matrix-elements are equal zero),

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

Each of the three vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ contains $n+1$ elements (indexed from 0 to n), to match the mathematical notation, dimensions and boundary conditions. We set then $a_0 = b_0 = c_0 = 0$ and $a_n = c_n = 0$, as these elements do not entry the matrix $\mathbf{A}$.

2.2 Algorithms

General (Thomas) algorithm

Following QuantStart (with some modifications increasing computational efficiency), we will develop a general algorithm for solving sets of linear equations on the form of Eq. (2). This method is known as the Thomas Algorithm and applies Gaussian elimination to a tridiagonal matrix, such as our previously defined matrix $\mathbf{A}$. We start with the set of linear equations, as shown before

$$\begin{aligned} b_1 v_1 + c_1 v_2 &= \tilde{b}_1 \\ a_1 v_1 + b_2 v_2 + c_2 v_3 &= \tilde{b}_2 \\ &\vdots \\ a_{i-1} v_{i-1} + b_i v_i + c_i v_{i+1} &= \tilde{b}_i \\ &\vdots \\ a_{n-1} v_{n-1} + b_n v_n &= \tilde{b}_n \end{aligned}$$

where coefficients a_i, b_i, c_i and $\tilde{b}_i = h^2 f(x_i) = 100 h^2 e^{-10 x_i}$ are known and we seek the solution elements v_i.

The fist step is performing a forward substitution by replacing some elements with the modified ones, formed as follows (new coefficients are denoted with primes):

$$\begin{aligned} \text{for} \quad & i = 1, 2, ..., n : \\ & m = b_i - a_{i-1} c_{i-1}^{'} \\ & c_i^{'} = \frac{c_i}{m} \\ & \tilde{b}_i^{'} = \frac{\tilde{b}_i - a_{i-1} \tilde{b}_{i-1}^{'}}{m} \end{aligned}$$

where factor m is precalculated in each step in order to decrease the number of necessary floating-point operations.
The solution is then obtained by the backward substitution:

$$\begin{aligned} \text{for} \quad & i = n, n-1, ..., 1 : \\ & v_i = \tilde{b}_i^{'} - c_i^{'} v_{i+1} \end{aligned}$$

The same notation and indexing is followed in the program, in order to avoid any confusion.

Specialized algorithm

It is worth noticing that matrix $\mathbf{A}$ has identical values of all main diagonal elements and the same values across the first diagonal below and above the main one, i.e. $b_1, b_2, ..., b_n = 2$ and $a_1, a_2, ..., a_{n-1} = c_1, c_2, ..., c_{n-1} = -1$. This observation allows us to introduce a significant

simplification to the general algorithm. We implement these known values into the forward subsitution algorithm described in the previous subsection and get

$$\begin{aligned}
\text{for} \quad & i = 1, 2, ..., n : \\
& m = 2 - (-1)c_{i-1}^{'} = 2 + c_{i-1}^{'} \\
& c_i^{'} = \frac{-1}{m} = -\frac{1}{2 + c_{i-1}^{'}} \\
& \tilde{b}_i^{'} = \frac{\tilde{b}_i - (-1)\tilde{b}_{i-1}^{'}}{m} = \frac{\tilde{b}_i + \tilde{b}_{i-1}^{'}}{2 + c_{i-1}^{'}}
\end{aligned}$$

which can be further simplified by rewritting $c_i^{'}$ as a function of i only, not dependant on the previous entries:

$$\begin{aligned}
c_1^{'} &= -\frac{1}{2+0} = -\frac{1}{2} \\
c_2^{'} &= -\frac{1}{2 - \frac{1}{2}} = -\frac{2}{3} \\
c_3^{'} &= -\frac{1}{2 - \frac{2}{3}} = -\frac{3}{4} \\
\Rightarrow c_i^{'} &= -\frac{i}{i+1}.
\end{aligned}$$

Therefore, the forward substitution in the specialized algorithm becomes

$$\begin{aligned}
\text{for} \quad & i = 1, 2, ..., n : \\
& \tilde{b}_i^{'} = \frac{\tilde{b}_i + \tilde{b}_{i-1}^{'}}{2 + c_{i-1}^{'}} = \frac{\tilde{b}_i + \tilde{b}_{i-1}^{'}}{2 - \frac{i-1}{i}}
\end{aligned}$$

and backward substitution:

$$\begin{aligned}
\text{for} \quad & i = n, n-1, ..., 1 : \\
& v_i = \tilde{b}_i^{'} - c_i^{'} v_{i+1} = \tilde{b}_i^{'} + \frac{i v_{i+1}}{i+1}
\end{aligned}$$

Lower-upper (LU) decomposition

Another method for finding the solution to the problem we are interested in, is to rewrite our matrix $\mathbf{A}$ as the product of a lower triangular matrix, $\mathbf{L}$, and an upper triangular matrix, $\mathbf{U}$, so that Eq. (2) becomes

$$\mathbf{Av} = \mathbf{LUv} = \tilde{\mathbf{b}}.$$

The equation can now be calculated in two steps:

$$\mathbf{Lw} = \tilde{\mathbf{b}},$$

$$\mathbf{Uv} = \mathbf{w}.$$

We will not describe the details of this algorithm, as the program uses functions lu and solve from library armadillo in order to perform the operations above (see Hjorth-Jensen for more thorough mathematical description).

2.3 Error calculation

In order to determine the accuracy of our numerical approximation, we want to compute the relative error in the data set as

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u(x_i)}{u(x_i)}\right|\right) \quad \text{for } i = 1, 2, ..., n,$$

where $u(x_i) = 1 - (1 - e^{-10})x_i - e^{-10x_i}$ is the exact closed-form solution.
Thereafter, we find its maximum value in a given data set, ε_{max}, for different number of grid points, n. Increasing n implies decreasing the stepsize, as $h = 1/(n + 1)$. We will then show how the maximal relative error changes as a function of $\log_{10}(h)$.

2.4 Numerical implementation

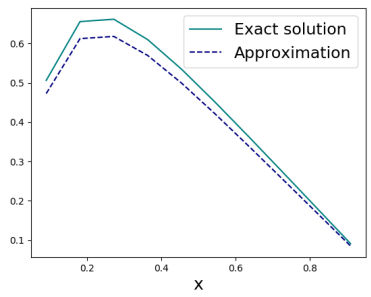
As already briefly mentioned in the Introduction, the main program has been written in C++. It takes two parameters as an input from terminal - the first one is the number of grid points, n, the second one is the name of the algorithm we want to use (either general, special or LU). We set up necessary variables and vectors, using dynamic memory handling. After computations are performed, the solution is written to an output file (solution.outn for a provided n) on the form of 4 columns with consecutively x_i, v_i, $u(x_i)$ and ε_i values for each point on the grid, $i = 1, 2, ..., n$. In case of the general and specialized algorithm we find the maximal relative error in a given data set. We also measure the CPU time for all methods to investigate the efficiency of different algorithms. An exemplary run of the program in the terminal looks then as follows:

```
>> ./ project1_main 1000 special
Using special algorithm with 1000 grid points.
Time elapsed for special-case algorithm: 7.8e-05 s
Maximal relative error for the step-size log(h) = -3.00043 is -5.08005.
```

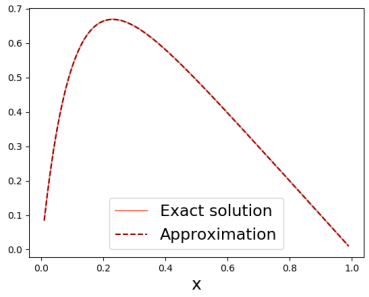
The output files may subsequently be read by a Python program that contains a function plotting the numerical approximation and the exact analytical solution for a given data set (project1_read_plot.py). This program is also used to generate the plot of maximal relative error as a function of stepsize (based on manually defined arrays, read off from terminal-output).

3 Results and discussion

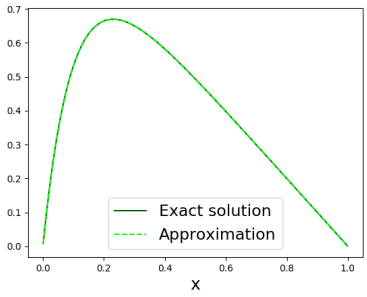
Running the programs with use of the general Thomas algorithm for $n = 10$,$n = 100$ and $n = 1000$ results in the plots shown on figure 1. (however, all of the three methods produce similar results for these n values). We can see that for $n = 10$ grid points, the numerical solution deviates from the exact one. Nevertheless, as we increase n, precision of the approximation gets much better, so that the plots overlap, being almost indistinguishable from each other.



(a) $n = 10$ grid points.



(b) $n = 100$ grid points.



(c) $n = 1000$ grid points.

Figure 1: Numerical approximation in comparison with the exact closed-form soluion for different number of grid points, n.

3.1 Error analysis

It is clear that increasing the number of grid points (i.e. decreasing the stepsize, h) leads to more accurate solution, that resembles its exact analytical form more closely. Indeed, by analyzing the maximal relative error as a function of stepsize (see figure 2. and table 1.), we can see that as h decreases, the relative error gets smaller as well. However, the situation changes when n reaches the magnitude 10^5 ($\log_{10}(h) = -5$). We get to the point of loss of numerical precision, and the relative error starts increasing again.

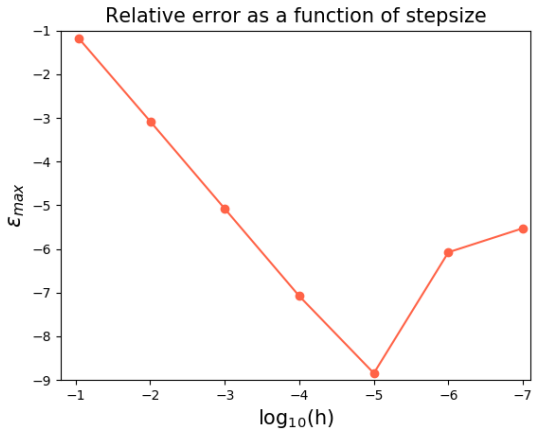


Figure 2: Maximal value of the (logarithmic) relative error, ε_{max}, as a function of $\log_{10}(h)$.

$\log_{10}(h)$	ε_{max}
-1.04	-1.18
-2.00	-3.09
-3.00	-5.08
-4.00	-7.08
-5.00	-8.84
-6.00	-6.08
-7.00	-5.53

Table 1: Maximal value of the relative error for the data sets with different stepsizes, h.

3.2 Efficiency

Table 2. collects some parameters that allow us to compare the three discussed algorithms with regard to their efficiency. CPU time given in table 2. is averaged over 3 consecutive runs of the program (as it is not the same for each execution of the program). We see that the number of floating point operations needed in order to perform computations using the

general and specialized algorithm is of order n, while LU-decomposition requires significantly more, $(2/3)n^3$. This obviously translates to much longer CPU time and even out of memory error for $n \geq 10^5$, making execution of the program impossible.

The number of FLOPS for the specialized algorithm is given under the assumption that the i-dependent factor is precalculated (which can be done since it does not depend on any other elements). We follow the same strategy in the program, not taking into account the CPU time needed to perform this precalculation. In that case, the specialized algorithm seems to be the most efficient one.

		Algorithm		
		General	Specialized	LU
FLOPS		8n	4n	$(2/3)n^3$
CPU time [s]	$n = 10^4$	0.00135	0.00068	65.168
	$n = 10^5$	0.01515	0.01158	-
	$n = 10^6$	0.11339	0.04708	-

Table 2: Comparison between three algorithms discussed in the project, taking into account number of floating point operations (FLOPS) and averaged CPU time for different number of grid points, n. "-" for LU-decomposition indicates that execution of the program was impossible.

4 Conclusions

There are many mathematically correct methods for solving linear sets of equations, like the one-dimensional Poisson's equation discussed throughout this project. Nevertheless, not every method is suitable for numerical computations. It is definitely worth spending some extra time and effort to modify the existing algorithm, in order to "customize it" to the specific case that we are considering. A good approach may save a lot of computational time, whereas a bad one may even make computations impossible to perform (like in case of LU-decomposition for a big number of grid points). It is also necessary to keep track of numerical precision and not get carried away while decreasing the spacing between grid points too much.

BIBLIOGRAPHY

Hjorth-Jensen, M. (2018, September 6). Computational Physics Lectures: Linear Algebra Methods. Retrieved from
http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/pdf/linalg-print.pdf

QuantStart Team. "Tridiagonal Matrix Solver via Thomas Algorithm." *QuantStart*, n.d., www.quantstart.com/articles/Tridiagonal-Matrix-Solver-via-Thomas-Algorithm.