Matrix algorithms for solving the one-dimensional Poisson equation with Dirichlet boundary conditions

Benedicte B. Nyheim, René A. Ask and Trine K. Olafsen

(Dated: September 9, 2019)

We've solved the one-dimensional Poisson equation with Dirichlet boundary conditions. This was done by discretization of the the second derivative and conversion of the equation to a tridiagonal linear matrix equation. This matrix equation was then solved by three different algorithms; A generalized, a specialized Thomas algorithm and a generalized Thomas algorithm combined LU-decomposition.

We derived an approximation to the optimal choice of step size which we then compare with the global minimum of the computed relative error of the solutions pertaining to each algorithm. We found that the global minimum was in agreement with the estimate for the specialized algorithm. For the two other algorithms we argued that their global minima were in agreement with the estimate as well if rounding-errors were taken into account.

I. INTRODUCTION

Differential equations show up in all branches of physics. While there exists a subset of such equations that permit closed-form solutions, the differential equations that emerge from the vast majority of real world problems require us to resort to approximation schemes simply because no such solution exists. This lack of existence prompts us to develop and estimate the error of numerical methods in order to study and assert the validity of our results.

In this article we shall study, compare and contrast three methods of solving Poisson's equation by converting the problem into a linear tridiagonal matrix equation. The first algorithm we'll look at is a general one developed to solve a tridiagonal matrix equation where we do not assume much about the matrix elements aside from requiring that the matrix is indeed invertible. We will further specialize the algorithm for our particular matrix and benchmark the two methods for comparative purposes. We will also compare these two methods towards a more intricate method using LU-decomposition in combination with the normal forward- and backward-substitutions required by the special form of Gaussian elimination we'll need. Finally we'll compare and contrast the three algorithms using benchmarks such as time used and computed relative error.

II. REPRODUCIBILITY

All codes used to produce the results found in this article can be found at [1]. To run the code, simply write the following in a standard linux terminal:

```
python3 main.py <exponent_of_the_higest_number_of_gridpoints> <which_algorithm>
```

By the exponent of the highest number of gridpoints we simply mean that if you type in 7, you'll run the code for $n = 10, 10^2, ..., 10^7$. There are three different algorithms you can choose from. Simply type *general* to get algorithm 1, *special* to run algorithm 2 or *lu* to run algorithm 3. You might have to change the name of some directories on the computer it's run at in order for the code to work properly with your computer. These can be manually changed in the .py files in the codes found at the github repository.

III. METHOD

The general form of the differential equation (DE) with Dirichlet boundary conditions we wish to solve is

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

To this end, we'll derive an approximation to the second derivative of a general continuous function $f(x)$ before we estimate the total error of the approximation. For simplicity, we'll assume that this function is analytic, that is, it has derivatives of all orders.

A. Derivation of the approximation scheme and an upper-bound estimate on its total error

The derivation in this section is based on the work found here [4]. We start by Taylor expanding about some point x

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(\xi_1), \tag{2}$$

and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(\xi_2), \tag{3}$$

where $\xi_1 \in (x, x+h)$ and $\xi_2 \in (x-h, x)$. Adding the two equations and rearranging a bit we get

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h}{3!}\left[f'''(\xi_1) - f'''(\xi_2)\right]. \tag{4}$$

To find the signed truncation error, we rewrite the equation as

$$f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = -\frac{h}{3!}\left[f'''(\xi_1) - f'''(\xi_2)\right]. \tag{5}$$

Before we proceed with the error analysis, we might as well pick $\xi \in (x-h, x+h)$ such that $f(\xi) = \max\{f(\xi_1), f(\xi_2)\}$ and multiply it by a factor 2 at the right-hand side (RHS), that is

$$f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = -\frac{2h}{3!}f'''(\xi) \tag{6}$$

Due to the fact that computers are unable to represent numbers exactly, the computed values can be written as $\bar{f}(x+h) = f(x+h)(1+\epsilon_1)$, $\bar{f}(x) = f(x)(1+\epsilon_2)$ and $\bar{f}(x-h) = f(x-h)(1+\epsilon_3)$. where ϵ_1, ϵ_2 and ϵ_3 are small real numbers, roughly of the magnitude 10^{-16}. An upper-bound estimate of the global total error, that is, the truncation error and the error due to loss of precision is

$$\begin{aligned}
\epsilon &= \left|f''(x) - \bar{f}''(x)\right| \\
&= \left|f''(x) - \frac{\bar{f}(x+h) - 2\bar{f}(x) + \bar{f}(x-h)}{h^2}\right| \\
&= \left|f''(x) - \frac{f(x+h)(1+\epsilon_1) - 2f(x)(1+\epsilon_2) + f(x-h)(1+\epsilon_3)}{h^2}\right| \\
&= \left|f''(x) - \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{f(x+h)\epsilon_1 - 2f(x)\epsilon_2 + f(x-h)\epsilon_3}{h^2}\right| \\
&\leq \left|-\frac{2h}{3!}f'''(\xi) - \frac{\epsilon_1 - 2\epsilon_2 + \epsilon_3}{h^2}\max_{\zeta\in(x-h,x+h)} f(\zeta)\right| \\
&\leq \frac{h}{3}\max_{\xi\in(x-h,x+h)}|f'''(\xi)| + \frac{|\epsilon_1| + 2|\epsilon_2| + |\epsilon_3|}{h^2}\max_{\zeta\in(x-h,x+h)}|f(\zeta)| \\
&\leq \frac{h}{3}\max_{\xi\in(0,1)}|f'''(\xi)| + \frac{4\epsilon_M}{h^2}\max_{\zeta\in(0,1)}|f(\zeta)|.
\end{aligned} \tag{7}$$

Furthermore, we set $\epsilon_M = \max\{|\epsilon_1|, |\epsilon_2|, |\epsilon_3|\}$. Now let $M_1 \equiv \max_{\xi\in(0,1)}|f'''(\xi)|$ and $m_2 \equiv \max_{\zeta\in(0,1)}|f(\zeta)|$. Then the upper-bound estimate of the total error can neatly be written as

$$\epsilon \leq \frac{h}{3}M_1 + \frac{4\epsilon_M}{h^2}M_2. \tag{8}$$

Solving $\mathrm{d}\epsilon/\mathrm{d}h = 0$ with respect to h yields the optimal choice of step-size h^* given as

$$h^* = \left(\frac{24\epsilon_M M_2}{M_1}\right)^{1/3}. \tag{9}$$

We need to determine M_1 and M_2 to estimate h^* for our particular closed-form solution to the differential equation. This function is

$$f(x) = 1 - (1 - e^{-10})x - e^{-10x}, \tag{10}$$

and solving $\mathrm{d}f/\mathrm{d}x = 0$ yields

$$\zeta = -\frac{\ln\left(1 - e^{-10}\right) - \ln(10)}{10}, \tag{11}$$

such that $M_2 = |f(\zeta)|$. Furthermore, $f^{(4)}(x) = -10^4 \exp(-10x)$, hence we see that $M_1 \leq \lim_{\xi \to 0} |f'''(\xi)|$. For simplicity, we might as well just set $M_1 = |f'''(0)|$. Then we obtain

$$h^* \approx \left(\frac{24\epsilon_M |f(\zeta)|}{|f'''(0)|}\right)^{1/3} \approx 2.51 \times 10^{-6}, \tag{12}$$

where we set $\epsilon_M = 10^{-15}$. For our purposes, it's more convenient to express it in terms of the logarithm

$$\log_{10}(h^*) \approx -5.6 \tag{13}$$

B. Approximate solution of the differential equation by the Thomas algorithm

From the arguments above it's obvious that we can approximate the second derivative of a function $f(x)$ by

$$f''(x_i) \approx \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2} \equiv \frac{f_{i+1} - f_i + f_{i-1}}{h^2}, \tag{14}$$

Now, for our particular DE, we want to approximate the true solution $u(x)$ by the a function $v(x)$. Using the discretization from above, we can thus rewrite our DE as

$$-\frac{-v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \quad i = 1, 2, ..., n, \tag{15}$$

which may be rearranged into

$$2v_i - v_{i+1} - v_{i-1} = f_i h^2 \equiv q_i. \tag{16}$$

From (16) we can rewrite this into a matrix equation as follows

$$\begin{pmatrix} 2v_1 - v_2 \\ -v_1 + 2v_2 - v_3 \\ \vdots \\ 2v_n - v_{n-1} \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ \vdots & & & & \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{pmatrix}. \tag{17}$$

To solve the matrix equation we'll first develop a general algorithm based on forward- and backward substitution to solve an equation $A\boldsymbol{x} = \boldsymbol{q}$ where A is a tridiagonal matrix as below. Performing one step of Gaussian elimination on the following matrix yields

$$A = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ a_1 & b_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix} \sim \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ 0 & b_2 - \frac{a_1}{b_1} c_1 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix}. \tag{18}$$

Defining $\tilde{b_2} \equiv b_2 - (a_1/b_1)c_1$, we can perform another step

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ 0 & \tilde{b_2} & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix} \sim \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ 0 & \tilde{b_2} & c_2 & 0 & \cdots & \cdots \\ 0 & 0 & b_3 - \frac{a_2}{\tilde{b_2}}c_2 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix}, \tag{19}$$

and at this point it's pretty obvious that the general pattern for the forward-substitution is

$$b_i = b_i - \frac{a_{i-1}}{b_{i-1}}c_{i-1}, \qquad \text{for} \qquad i = 2, 3, ..., n, \tag{20}$$

$$q_i = y_i - \frac{a_{i-1}}{b_{i-1}}y_{i-1}, \qquad \text{for} \qquad i = 2, 3, ..., n, \tag{21}$$

where the second equation is simply performing the exact same operation on the RHS of the equation.

Once this process in completed, we need to perform back-substitution to find $\boldsymbol{x}$. At this point our equation will look as follows:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ 0 & \tilde{b}_2 & c_2 & 0 & \cdots & \cdots \\ 0 & 0 & \tilde{b}_3 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \tilde{b}_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & \tilde{b}_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1x_1 + c_1x_2 \\ \tilde{b}_2x_2 + c_2x_3 \\ \tilde{b}_3x_3 + c_3x_4 \\ \vdots \\ \tilde{b}_{n-1}x_{n-1} + c_{n-1}x_n \\ \tilde{b}_nx_n \end{pmatrix} = \begin{pmatrix} \tilde{q}_1 \\ \tilde{q}_2 \\ \tilde{q}_3 \\ \vdots \\ \tilde{q}_{n-1} \\ \tilde{q}_n \end{pmatrix}, \tag{22}$$

so the general algorithm for the back-substitution is

$$x_i = \frac{q_i}{b_i}, \qquad \text{for} \qquad i = n, \tag{23}$$

$$x_i = \frac{q_i - c_ix_{i+1}}{b_i}, \qquad \text{for} \qquad i = n-1, n-2, ..., 1. \tag{24}$$

For our particular matrix, we know that $b_i = 2$ and $c_i = a_i = -1$, so we can specialize the algorithm. To see this, we first motivate the formula by

$$b_1 = b = 2, \tag{25}$$

$$b_2 = b - 1/b_1 = 2 - \frac{1}{2} = \frac{3}{2}, \tag{26}$$

$$b_3 = b - 1/b_2 = 2 - \frac{1}{3/2} = 2 - \frac{2}{3} = \frac{4}{3}, \tag{27}$$

$$\vdots \tag{28}$$

$$b_i = \frac{i+1}{i}, \qquad \text{for} \qquad i = 1, 2, ..., n, \tag{29}$$

$$q_i = q_i + \frac{i-1}{i}q_{i-1}, \qquad \text{for} \qquad i = 1, 2, ..., n. \tag{30}$$

For the solution $\boldsymbol{x}$ we get the recursive relations

$$x_n = \frac{n}{n+1}q_n, \tag{31}$$

$$x_i = \frac{i}{i+1}(q_i + x_{i+1}), \qquad \text{for} \qquad i = n-1, n-2, ..., 1. \tag{32}$$

C. LU-decomposition, Forward- and Back-substitution

We will heavily lend inspiration from this source [3] in this section. We want to solve the equation $A\boldsymbol{x} = \boldsymbol{q}$ by using a LU-decomposition $A = LU$. To this end we begin by explicitly writing out our tridiagonal matrix A, the lower-triangular matrix L and the upper-triangular matrix U. We also note that the bandwidth of the banded matrix must be conserved, which we accomplish by writing the following general matrices:

$$A = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\ a_1 & b_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_2 & b_3 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \ell_2 & 1 & \cdots & \cdots & \cdots & 0 \\ 0 & \ell_3 & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & \ell_{n-1} & 1 & 0 \\ 0 & 0 & \cdots & \cdots & \ell_n & 1 \end{pmatrix} \begin{pmatrix} d_1 & u_1 & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 & u_2 & \cdots & \cdots & 0 \\ 0 & 0 & d_3 & u_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & d_{n-1} & u_{n-1} \\ 0 & 0 & \cdots & \cdots & 0 & d_n \end{pmatrix} = LU, \tag{33}$$

and performing matrix multiplication we get

$$LU = \begin{pmatrix} d_1 & u_1 & \cdots & \cdots & \cdots & 0 \\ \ell_2 d_1 & \ell_2 u_1 + d_2 & u_2 & \cdots & \cdots & 0 \\ 0 & \ell_3 d_2 & \ell_3 u_2 + d_3 & u_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ell_{n-1} d_{n-2} & \ell_{n-1} u_{n-2} + d_{n-1} & u_{n-1} \\ 0 & 0 & \cdots & \cdots & \ell_n d_{n-1} & \ell_n u_{n-1} + d_n \end{pmatrix}, \tag{34}$$

which yields the following general relations:

$$d_i = b_i, \qquad u_i = c_i, \qquad \text{for} \qquad i = 1, \tag{35}$$

$$\ell_i = \frac{a_{i-1}}{d_{i-1}}, \qquad \text{for} \qquad 1 < i \leq n, \tag{36}$$

$$d_i = b_i - \ell_i u_{i-1}, \qquad \text{for} \qquad 1 < i \leq n, \tag{37}$$

requiring $2n$ multiplications and n additions. In other words, the total floating point operations involved in finding LU is $3n$.

We can then write $A\boldsymbol{x} = LU\boldsymbol{x} = L\boldsymbol{y} = \boldsymbol{q}$ where $\boldsymbol{y} \equiv U\boldsymbol{x}$. Explicitly, we can write this as

$$L\boldsymbol{y} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ \ell_2 & 1 & \cdots & \cdots & \cdots & 0 \\ 0 & \ell_3 & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & \ell_{n-1} & 1 & 0 \\ 0 & 0 & \cdots & \cdots & \ell_n & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \ell_2 y_1 + y_2 \\ \ell_3 y_2 + y_3 \\ \vdots \\ \ell_{n-1} y_{n-2} + y_{n-1} \\ \ell_n y_{n-1} + y_n \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ \vdots \\ q_n \end{pmatrix} \tag{38}$$

which yields the following procedure:

$$y_1 = q_1, \tag{39}$$

$$y_i = q_i - \ell_i y_{i-1}, \qquad \text{for} \qquad i = 2, 3, ..., n. \tag{40}$$

giving $n-1$ multiplications and $n-1$ additions. Thus the added computational cost is $2(n-1)$.

Finally, to determine $\boldsymbol{x}$, we perform back-substitution by solving $U\boldsymbol{x} = \boldsymbol{y}$. Writing it out explicitly yields

$$\begin{pmatrix} d_1 & u_1 & \cdots & \cdots & \cdots & 0 \\ 0 & d_2 & u_2 & \cdots & \cdots & 0 \\ 0 & 0 & d_3 & u_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & d_{n-1} & u_{n-1} \\ 0 & 0 & \cdots & \cdots & 0 & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 v_1 + u_1 v_2 \\ d_2 v_2 + u_2 v_3 \\ \vdots \\ \vdots \\ d_{n-1} v_{n-1} + u_{n-1} v_n \\ d_n v_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix}, \tag{41}$$

which yields the following procedure:

$$x_n = \frac{y_n}{d_n}, \tag{42}$$

$$x_i = \frac{y_i - u_i x_{i+1}}{d_i}, \qquad \text{for} \qquad i = n-1, n-2, ..., 1. \tag{43}$$

Here we got $2n-1$ multiplications and $n-1$ additions, so the number of floating point operations are $3n-2$. Putting all of these together we get $4(2n-1) \approx 8n$ floating point operations.

As a measure of the error between the closed-form solution $u(x)$ and the approximated solution $v(x)$, we shall use

$$E(x) = \log_{10}\left(\left|\frac{u(x) - v(x)}{u(x)}\right|\right). \tag{44}$$

D. The Algorithms and their Computational Cost

The general algorithm can be summarized as follows

Similarly, the special algorithm is

Algorithm 1 General Thomas algorithm

Step 1: Forward substitution
for $i = 1, 2, ..., n$ **do**
  $b_i = b_i - a_{i-1}c_{i-1}/b_{i-1}$
  $q_i = q_i - a_{i-1}q_{i-1}/b_{i-1}$
Step 2: Backward substitution
for $i = n, n-1, ..., 1$ **do**
  if $i = n$ **then**
    $x_i = q_i/b_i$
  else
    $x_i = (q_i - c_i x_{i+1})/b_i$

counting the number of floating-point operations (FLOPS), we get $6n$ from the forward-substitution and $3n$ from the backward-substitution. Thus the algorithm requires roughly $9n$ FLOPS.

The specialized algorithm is summarized below

Algorithm 2 Specialized Thomas Algorithm

Step 1: Forward substitution
for $i = 1, 2, ..., n$ **do**
  $q_i = q_i + [(i-1)/i]q_i$
Step 2: Backward substitution
for $i = n, n-1, ..., 1$ **do**
  if $i = n$ **then**
    $x_i = [i/(i+1)]q_i$
  else
    $x_i = [i/(i+1)](q_i + x_{i+1})$

The forward substitution requires $2n$ FLOPS and the backward substitution uses $2n$ FLOPS as well. Thus the total flop-count is $4n$ for this algorithm [2].

The algorithm developed using LU-decomposition is

Algorithm 3 Thomas algorithm with LU-decomposition

Step 1: LU-decomposition $A = LU$
for $i = 1, 2, ..., n$ **do**
  if $i = 1$ **then**
    $d_i = b_i$
    $u_i = c_i$
  else
    $\ell_i = a_{i-1}/d_{i-1}$
    $d_i = b_i - \ell_i u_{i-1}$
Step 2: Forward substitution, solve $L\boldsymbol{y} = \boldsymbol{q}$
for i = 1,2,...,n **do**
  if i = 1 **then**
    $y_i = q_i$
  else
    $y_i = q_i - \ell_i y_{i-1}$
Step 3: Backward substitution, solve $U\boldsymbol{x} = \boldsymbol{y}$
for $i = n, n-1, ..., 1$ **do**
  if $i = n$ **then**
    $x_i = y_i/d_i$
  else
    $x_i = (y_i - u_i x_{i+1})/d_i$

The LU-decomposition requires roughly $3n$ FLOPS, the forward substitution demands about $2n$ FLOPS and the backward-substitution needs about $3n$ FLOPS. Thus the algorithm's total computational cost is about $8n$ FLOPS, the same as the specialized algorithm.

IV. RESULTS

A. The numerical solution

The solution computed using algorithm 1 for $n \in \{10, 100, 1000\}$ grid points is shown in figure 1.

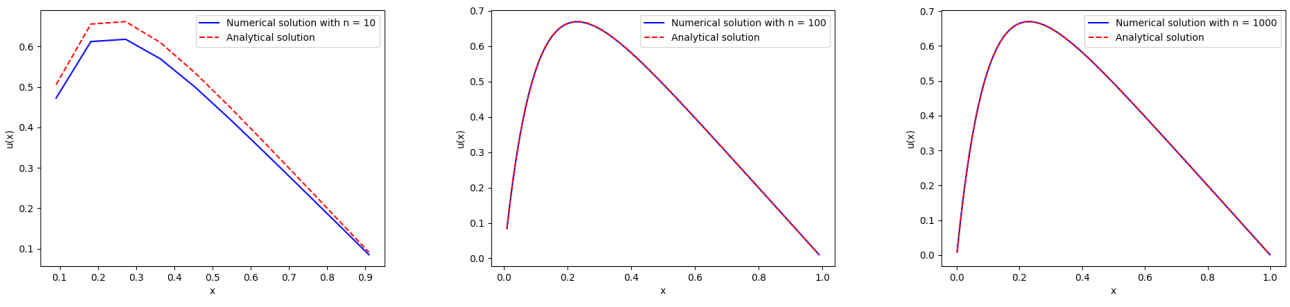

FIG. 1. The numerical solution $v(x)$ computed using the algorithm 1 and the closed-form solution $u(x)$ on the interval $(0, 1)$ for $n \in \{10, 100, 1000\}$.

The solution computed with algorithm 2 for $n \in \{10, 100, 1000\}$ is plotted in figure 2.

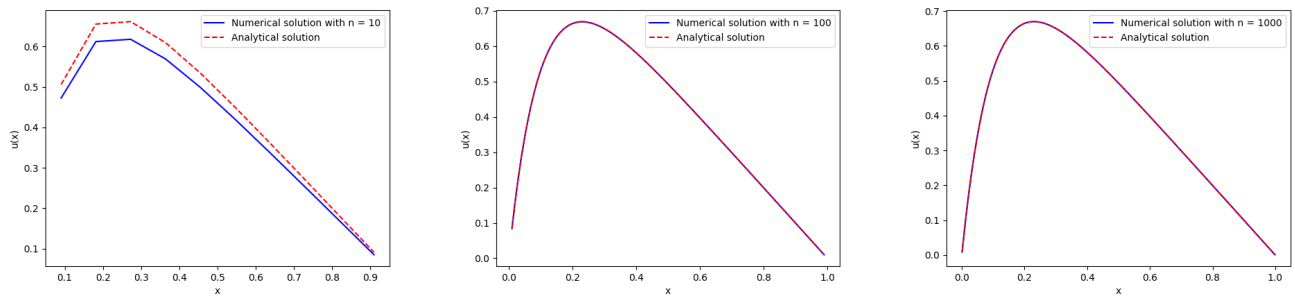

FIG. 2. The numerical solution $v(x)$ computed using algorithm 2 and the closed-form solution $u(x)$ on the interval $(0, 1)$ for $n \in \{10, 100, 1000\}$.

Finally, the computed solution using algorithm 3 for $n \in \{10, 100, 1000\}$ is shown in figure 3.

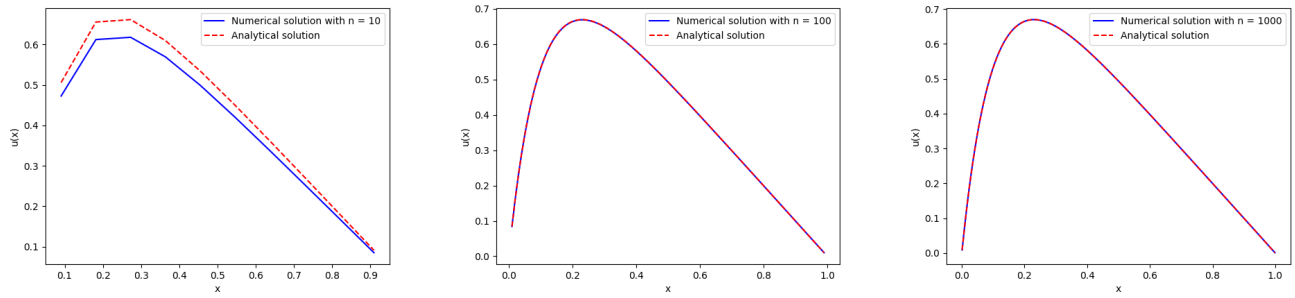

FIG. 3. The numerical solution $v(x)$ computed using algorithm 3 and the closed-form solution $u(x)$ on the interval $(0, 1)$ for $n \in \{10, 100, 1000\}$.

B. The maximum error

First we computed the maximum error using equation (8) for the purpose of comparison. It's shown in figure 4,

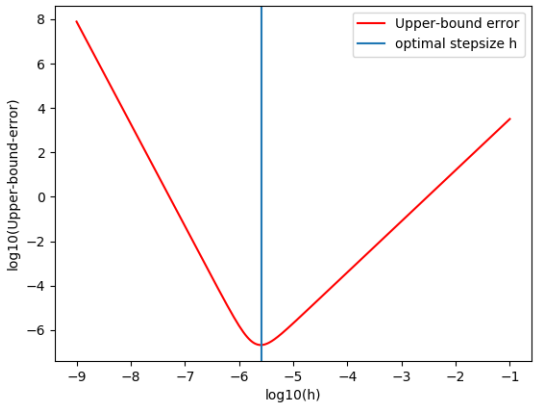

FIG. 4. The upper-bound error computed using eq. (8). We've also included the estimated optimal stepsize h^* which corresponds to the global minimum of the function.

The computed maximum error using equation (44) is shown in figure 5

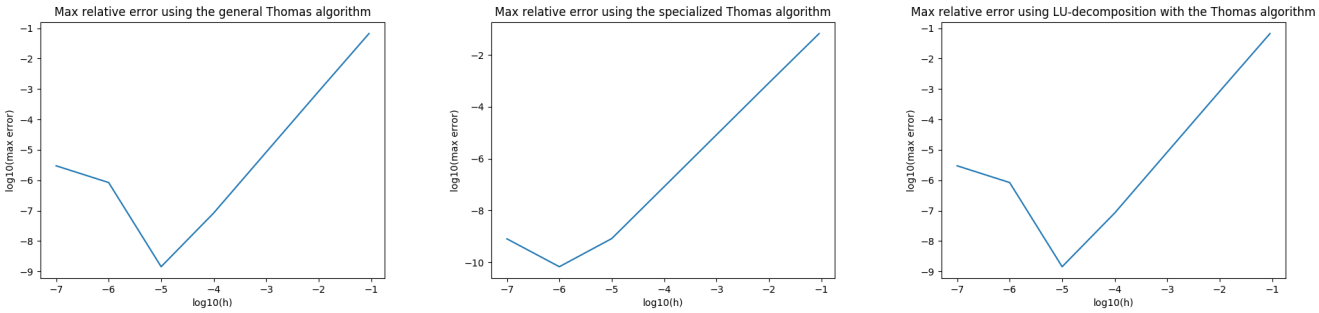

FIG. 5. The computed total error using (44) for all three algorithms.

C. The number of gridpoints vs. time

The computed time in seconds for each algorithm is shown in table I

n	Algorithm 1 [s]	Algorithm 2 [s]	Algorithm 3 [s]
10	0.000014	0.000014	0.000014
10^2	0.000013	0.000011	0.000012
10^3	0.000026	0.000016	0.000051
10^4	0.000146	0.000071	0.000279
10^5	0.001457	0.000832	0.002621
10^6	0.014909	0.007167	0.026178
10^7	0.152232	0.068831	0.259516

TABLE I. The number of gridpoints vs the time to complete the computations in seconds for all three algorithms.

V. DISCUSSION

A. Time vs FLOP-count

From table I we see that algorithm 2 is roughly twice as fast as algorithm 1 for large n which is consistent with the fact that the latter has about twice as large flop-count as the former. One unexpected anomaly is that the algorithm using the LU-decomposition uses longer time than the other algorithms for large n even though its flop-count is lower than the general algorithm. This prompted us to attempt optimizing the code by reducing the number of for-loops in algorithm 3 by one, essentially by moving the forward-substitution inside the for-loop pretaining to the LU-decomposition. This reduced the time used by a factor of 0.88. In other words, the algorithm still runs slower than algorithm 1. This at least indicates that the flop-count of an algorithm is not sufficient to determine which one will be faster. We do not however understand exactly why there's such a big difference between the two and we suggest that this would be an excellent problem to study further in detail.

B. Error

If we inspect figure 5, we see that algorithm 1 and 3 yielded the same result with a global minimum at roughly $\log_{10}(h) = -5$. This might be consistent with the fact that they have roughly the same flop-count and is possibly indicating that the matrix algorithms themselves accumulate about the same rounding errors. We can also note that algorithm 2 has a global minimum at roughly $\log_{10}(h) = -6$ which is consistent with the estimate we made regarding the optimal choice of step size in equation (13). However we do argue that the computed relative error of the two other algorithms are consistent with our estimate for the following reason: since the algorithms has a higher flop-count, they will indeed accumulate more numerical rounding errors throughout the computations. This implies that we'd expect the optimal choice of the step size to be higher since the error resulting from the computer itself, not the mathematics, will begin to dominate earlier than what you'd expect from our somewhat naive prediction. By earlier, we mean that you gradually decrease the step size towards the global minimum. Before this point (large h), the mathematical error

dominates and beyond this point (for very small h), loss of precision and rounding errors begins to dominate the total error. An interesting suggestion for further investigation here would be to predict the contribution from the rounding errors and make a better prediction of the optimal choice of step size.

VI. CONCLUSION

We've solved the one-dimensional Poisson equation by converting it to a tridiagonal linear matrix equation which we solved using three different algorithms. We implicitly obtained an estimate on the optimal choice of step size by deriving an upper-bound estimate on the total error of the approximation scheme for the double derivative. This estimate was then to be compared with the computed relative error between the approximate and the closed-form solution.

We found that the relative time used by algorithm 1 and 2 were consistent with their relative flop-count, while algorithm 3 presented an anomaly that we couldn't properly explain. We conclude that this is an interesting prospective problem to investigate further.

We found that the global minimum of the error pertaining to the solution computed with algorithm 2 was in agreement with the estimated optimal choice of step size. Furthermore, we argued that the global minima of the other two algorithms were in agreement as well if rounding errors were taken into account. We suggest that estimating the rounding-errors in order to obtain a better estimate on the optimal step size could be a fruitful problem to study further.

---

[1] https://github.com/benedibn/comphys/tree/master/projects/project1/codes.

[2] Morten Hjorth-Jensen. *Computational Physics, Lecture Notes Fall 2015.* 2015.

[3] George Em Karniadakis and Robert M. Kirby II. *Parallell Scientific Computing in C++ and MPI.* Cambridge University press, 2003.

[4] Knut Mørken. *Numerical Algorithms and Digital Representation.* 2017.