Solving the 1D Poisson Equation using Finite Difference Methods

Bekkevik M, Steinnes L.

Institute of Physics, University of Oslo

m.g.bekkevik@fys.uio.no, lasse.steinnes@fys.uio.no

September 10, 2020

In solving the one dimensional Poisson equation with Dirichlet conditions three different algorithms are tested. First a general algorithm with forward and backward substitution for an equation with a tridiagonal matrix, then a simplified version of that adjusted to a specific case with identical entries on the diagonals. Finally a LU decomposition which made use of armadillo, an external library package. All the numerical solutions are compared to the analytic solution. The LU approach performs worse in accuracy and CPU-time. The forward and backward substitution algorithms are equal in relative error $8.7772 \cdot 10^{-4}$ for 1000 grid points against $6.0287 \cdot 10^{-2}$ for LU. The special, simplified approach is the most CPU-time efficient with 21.3 μs for 1000 grid points.

I Introduction

Differential equations play an important role in many disciplines including engineering, physics, economics, and biology. Not all differential equations are analytically solvable, and in those cases numerical methods become indispensable. Unfortunately as with any calculations done by computers there is always errors. The aim is to keep the error as small as possible and within acceptable limits.

This report represents a simple algorithm for solving a one-dimensional Poisson equation by expressing the differential equation as a matrix equation which in turn can be solved by forward or backward substitution or LU decomposition. Since this equation has an exact solution, this provides a useful way of evaluating the algorithm by means of relative error and L2-norm as well as regular unit testing.

Under the Theory section is a presentation of the basic concepts and also a short deduction of the used algorithms in solving the Poisson equation is included. Then follows a description of the methods before the results are presented. Subsequently a discussion of the results and suggestions for improvements. The paper ends with a conclusion and bibliography.

II Theory

The theory section is mainly based upon Morten Hjorth-Jensen's lecture notes in the Computational Physics course (FYS4150) at the University of Oslo [1].

II.I Poisson's equation

The Poisson equation describes the relation between an electrostatic potential, $\boldsymbol{\phi}$ and a localized charge distribution $\rho(\boldsymbol{r})$ (eq. 1).

$$\nabla^2 \boldsymbol{\phi} = -4\pi\rho(\boldsymbol{r}) \tag{1}$$

Solving of this equation offers a description of the electromagnetic field through the electrostatic potential. In the case of a spherical symmetric potential, the equation is reduced to a one dimensional problem. Using spherical coordinates yields

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\boldsymbol{\phi}}{dr}\right) = -4\pi\rho(r). \tag{2}$$

By substituting $\boldsymbol{\phi}(\boldsymbol{r}) = \phi(r)/r$, the equation to solve becomes

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r). \tag{3}$$

II.II Matrix representation

The discrete method for solving the ordinary differential equation can be represented in matrix form. In our case the Poisson equation is one-dimentional $-u''(x) = f(x)$, with $x \in [0,1]$ and the Dirichlet (non-diffentiated) boundary conditions $u(0) = u(1) = 0$. $u(x)$ is a continous function. This equation is analytically solvable as well as numerical. To do an numerical analysis the function needs to be discretized. The discretized function is called $v(x)$ with x still in the intervall from 0 to 1 but with n discrete values separated

with a step length $h = (x_{n-1} - x_0)/(n+1) = 1/(n+1)$. The boundary condition is now expressed as $v_0 = v_n = 0$. We approximate the derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, ..., n \tag{4}$$

$$f_i = f(x_i). \tag{5}$$

We multiply by h^2 and get

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

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \tag{7}$$

With the n different values of x this equation is really a set of n linear equations, which can be expressed on matrixform: $Av = b_i$, where $b_i = h^2 f_i$. Here

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

This is a tridiagonal Tôplitz matrix which simplifies things quite a lot. This means that the computer don't have to store and update a whole matrix, but only three arrays - one for the diagonal elements one for the one-step-off diagonal elements and one for the b-values, here. The first algorithm does not require that the diagonal entries to be the same, nor the one-step-off the diagonal. The equation $Av = b$ can be solved by inverting $v = A^{-1}b$, but more efficiently is the method of Gaussian elimination or forward substitution.

II.III Forward and backward substitution

To solve such a matrix system requires two steps: 1) Forward substitution to reduce the matrix to upper triangular form, and 2) backward substitution to get the unknown parameter values (citation needed). These two operations corresponds to Gaussian elimination, also known as row reduction (citation needed).

Forward substitution involves row-wise substraction so that

$$a_{jk}^{(m+1)} = a_{jk} - \frac{a_{jm}^{(m)} e_{mk}^{(m)}}{a_{mm}^{(m)}}. \tag{8}$$

The same operations must also be performed on the right hand side of the matrix equation. Thus

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}}. \tag{9}$$

The resulting matrix will be on upper triangular form. Hence, backward substitution is needed to obtain the x_m's,

$$x_m = \frac{1}{b_{mm}} \left(y_m - \sum_{k=m+1}^{n} b_{mk} x_k \right) \tag{10}$$

for $m = (n-1, n-2, ..., 1)$. A general extended, symmetric, tridiagonal matrix is

$$\begin{bmatrix} d_1 & e_1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & b_1 \\ e_1 & d_2 & e_2 & 0 & \cdots & \cdots & \cdots & 0 & b_2 \\ 0 & e_2 & d_3 & e_3 & 0 & \cdots & \cdots & 0 & b_3 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & e_{n-1} & d_n & b_n \end{bmatrix}.$$

To get the matrix on upper triangular form, the e_1 (the first element in the second row) must be removed by subtracting $\frac{e_1}{d_1}$ times the first row from the second. This leaves the second diagonal element to be $d_2 - \frac{e_1^2}{d_1}$, which is called $\tilde{d}_2$ from now on. The last element in the second row will become $b_2 - \frac{b_1 e_1}{d_1}$ which is now called $\tilde{b}_2$. This continues down through all the rows of the matrix. The general expressions for the resulting diagonal elements and the last element is

$$\tilde{d}_i = d_i - \frac{e_{i-1}^2}{\tilde{d}_{i-1}} \tag{11}$$

$$\tilde{b}_i = b_i - \frac{e_{i-1} \tilde{b}_{i-1}}{\tilde{d}_{i-1}} \quad \text{i=2, ..., n-1.} \tag{12}$$

The 0th and nth element is not included since they are constantly equal to 0. The first is not updated in the algorithm. The off-diagonal elements above is unchanged. After the Gaussian elimination the extended matrix is

$$\begin{bmatrix} d_1 & e_1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & \tilde{b}_1 \\ 0 & \tilde{d}_2 & e_2 & 0 & \cdots & \cdots & \cdots & 0 & \tilde{b}_2 \\ 0 & 0 & \tilde{d}_3 & e_3 & 0 & \cdots & \cdots & 0 & \tilde{b}_3 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & \tilde{d}_n & \tilde{b}_n \end{bmatrix}.$$

By the method of backward substitution the solution is obtained. By starting at the bottom with $\tilde{d}_n v_n = \tilde{b}_n \implies v_n = \frac{\tilde{b}_n}{\tilde{d}_n}$ you can unambiguously find $v_{n-1} = \frac{\tilde{b}_3 - e_3 v_4}{\tilde{d}_3}$ and continuing up through the rows of the matrix until all the v_i's are found. The general expression is

$$v_i = \frac{\tilde{b}_{n-1} - e_{n-1} v_{n-1}}{\tilde{d}_{n-1}}. \tag{13}$$

II.IV Simplified algoritm

With the numerical expression for the second derivative, the matrix will have identical entries on the diagonal band. This means that the algorithn can be simplified. The expressions for $\tilde{d}_i$ and $\tilde{b}_i$ can be written as

$$\tilde{d}_i = \frac{i+1}{i} \tag{14}$$

$$\tilde{b}_i = b_i + \frac{i-1}{i} b_{i-1} \tag{15}$$

II.V LU decomposition

LU decomposition or lower upper factorization is a decomposition of a matrix A into a lower triangular matrix L and an upper triangular matrix U. If $det(A) \neq 0$ A has an LU-factorization. If $A \in \mathbb{R}^{n \times n}$ is also non-singular, the LU-factorization is unique and the determinant is given by the diagonal elements of U. It can be used to solve a system of linear equations. Any solution v of the matrix equation $Av = b$ is also the solution to the equation $LUv = b$. The U matrix is found by Gaussian elimination of A. By noting all the elementary and other elimination matrices that A is multiplied with to become upper triangular, the lower triangular matrix can be found. For instance if U is obtained by multiplying A with 3 elimination matrices:

$$E_3 E_2 E_1 A = U \quad \text{you can solve for } A, \tag{16}$$

$$A = \underbrace{E_1^{-1} E_2^{-1} E_3^{-1}}_{L} U \quad \text{and that means} \tag{17}$$

$$L = E_1^{-1} E_2^{-1} E_3^{-1}. \tag{18}$$

To see how the computer algorithm goes consider the 4x4-matrix

$$A = LU \tag{19}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & {}_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix} \tag{20}$$

From the first column the elements u_{11}, l_{21}, l_{31} and l_{41} can be determined. From this all the unknown elements in the equations involving the second column in A can be found. And so fourth. Generally the steps are column by column (j) determine the first element by $u_{1jj} = a_{1j}$. Then determine all $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$ for $i = 2, \cdots, j-1$. Next calculate the diagonal elements $u_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk}u_{kj}$. Ultimately find the $l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right)$.

II.VI Analytic solution

The differential equation does have an exact solution. The right hand side is of the equation is in this case $f(x) = 100e-10x$.

$$-\frac{d^2u}{dx^2} = 100e^{-10x} \tag{21}$$
$$-\frac{du}{dx} = \int 100e^{-10x}dx \tag{22}$$
$$-\frac{du}{dx} = -10e^{-10x} + A \tag{23}$$
$$-u(x) = \int -10e^{-10x} + Adx \tag{24}$$
$$-u(x) = e^{-10x} + Ax + B \tag{25}$$
$$u(x) = -e^{-10x} - Ax - B. \tag{26}$$

A and B are some constants. Using the boundary condition:

$$u(0) = -1 - B = 0 \tag{27}$$
$$B = -1 \tag{28}$$
$$u(1) = -e^{-10} - A + 1 = 0 \tag{29}$$
$$A = 1 - e^{-10}u(x) = 1 - (1 - e^{-10})x - e^{-10x}, \tag{30}$$

which is the analytic solution. This is useful for evaluating the numerical solution.

III Methods

To solve the one-dimensional Poisson equation $-u''(x) = f(x)$, with $x \in [0, 1]$, and the Dirichlet (non-differentiated) boundary conditions $u(0) = u(1) = 0$, the method of forward an backward substitution is used as well as LU decomposition. All methods are solved with n= (10, 100 and 1000) points in the numerical solution. However, results for

n up to 10^6 is presented under time usage, but no further analysis are made of those results. A test is included for a quadratic function, since a central difference scheme for the second derivative is exact for $p(x) = c_0 + c_1 x + c_2 x^2$.

The algorithms are implemented in the low level programming language C++, results are written to csv-files. Results are plotted in the high level programming language Python, by using `pandas` and `matplotlib` packages in the program `test-visualizermain.py`, which applies the functions implemented in `postprocessing.py`. All implementations can be found at https://github.com/lasse-steinnes/FYS4150-Project1.

III.I Thomas algorithm

The forward and backward substitution is implemented for a case where the diagonal elements is not required to be equal in the program `generalsolver.cpp`. The same algorithm is tailored to the problem at hand (see sec.II) and can be found in a simplified version, `simplesolver.cpp`. The values written to file are the step size (h), x_i's, numerical solution v(x_i), known analytical solution u(x_i) and the time usage.

III.II LU-decomposition

Two libraries are used to attain a solution from LU-decomposition. From the eigen library `<Eigen/Dense>`, an eigen-type matrix is created suited to a central difference method. The problem is simply solved by `Ainv = A.lu().solve(I)`, followed by `vnum = Ainv*rhs`. Wherein I is the identity matrix. In the same way, using the library `<armadillo>`, then `vnum = solve(A,rhs)`. The same output is written to file as for the mentioned Thomas algorithm implementation (sec. III.I).

III.III Error estimates and stability criteria

Since the equation has an analytical solution, this provides an excellent way of evaluating the methods by means of the relative error and the L2-norm as a function of step size.

The relative error gives is given by

$$\varepsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right), \tag{31}$$

where v_i is an element from the numerical solution, and u_i is from the analytical solution. From each experiment with n-point solutions, the maximum was chosen as the final rela-

tive error. This error estimate was used to evaluate the effect of n on the order of magnitude for the error compared to the exact solution up to machine precision.

While the relative error gives a ratio of how large the absolute error ($e_i = v_i - u_i$) is compared to the exact solution, L2-norm/h^2 is a stability criteria for the numerical method with $O(h^2)$ error terms. It is defined as

$$L2/h^2 = \frac{E}{h^2} \tag{32}$$

where,

$$E = \sqrt{h \sum_{i=0}^{n-1} (e^2)}, \tag{33}$$

and h is the step size. When the numerical solution is stable, the $L2/h^2$ should remain approximately the same for a range of h-values. A natural extension would be to calculate convergence rate r, however, it was not performed in this implementation.

III.IV

IV Results

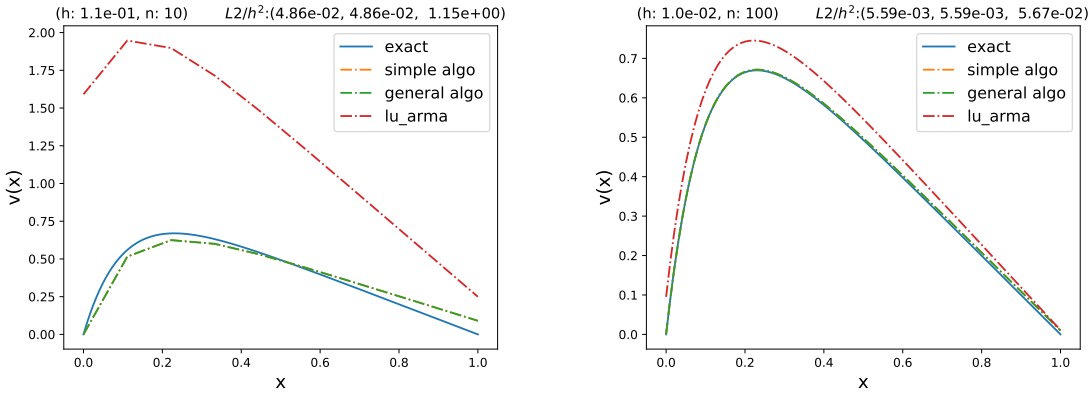
The results from finite difference methods performed on a 1D Poisson equation are presented in section IV.I. An analysis of the relative error, L2-norm and time usage are produced, and is shown in section IV.II.

IV.I Forward and backward substitution

Two algorithms for forward and backward substitution were used to solve the numerical solution, together with LU-decomposition provided by the armadillo library. The results were compared with the analytical solution in Fig. 1 - 2 (See section II for details on the equation to be solved).

Numerical solutions with low resolution (Fig. 1a-1b) fail to produce the exact behaviour of the analytical solution. The own-implemented code `simple algo` and `general algo` perform equally well, but underestimates the exact function. However, the LU-decomposition provided by the armadillo library, overestimates the physical solution by order of magnitude 10^0 at the most ($n = 10$).

The high resolution numerical solution ($n = 1000$) of the 1D problem, lies closer to the analytical solution (Fig. 2), when compared to numerical solutions with $n = (10, 100)$.



(a) n = 10

(b) n = 100

Figure 1: **Low resolution numerical solution to a 1D Poisson equation**: Results from two algorithms of different complexity, and LU-decomposition, are compared to the analytical solution (see section II and III). **a)** Solutions for resolution of 10 inner points (n = 10) **b)** Solutions for resolution of 100 inner points (n = 100). Parameters: h - step size, L2 - the L2-norm (left to right: simple algo, general algo, lu arma).

The L2-norm/h^2 does not remain stable between experiments n = $(10, 100, 1000)$ for each given method, wherein the LU-decomposition method goes from L2/h^2 = 1.15 (n = 10) to 5.32e-03 (n= 1000). The L2/h^2 should optimally be the same for different step sizes, h.

IV.II Error estimates and time usage

The time usage spans from microseconds (μs) to milliseconds (ms), depending on how many internal points are used in the iterative solution (tab. 1). In general the time usage goes down by a power of ten with a tenfold increase in iteration points (n). The time usage is in general lower for the Thomas algorithm tailored to solving the Poisson equation with a central difference scheme. However, there are some stochasticity prevalent in time usage as seen in tab 1, so that the general algorithm performs better for some larger n's. The generated solutions from the library-provided LU-decomposition perform worse in comparison the former methods, in which the eigen LU-solver is the worst all together (1.11e4 ms compared to 2.13e-2 ms (simple algo) for n = 1000).

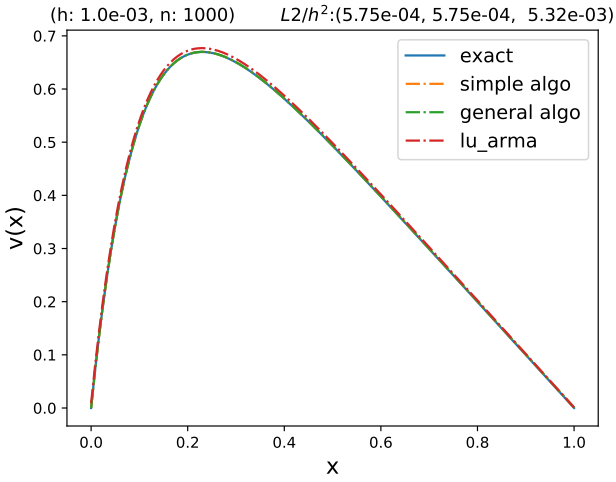


Figure 2: **High resolution numerical solution to a 1D poisson equation:** Results from two algorithms of different complexity, and LU-decomposition, are compared to the analytical solution. A resolution of 1000 inner points (n= 1000) is used. Parameters: h - step size, L2 - the L2-norm (left to right: simple algo, general algo, lu arma).

Table 1: **Time usage**: Time usage (ms, 10^{-3} s) of four different methods with different step sizes, $h = (x_n - x_0)/n$) for $x \in [x_0, x_n]$. The simple algorithm was tailored to solve the particular 1D Poisson equation in question, whereas the general method was implemented as a general Thomas algorithm. These are compared with LU-decomposition methods provided by the armadillo and eigen libraries.

n	Simple algo	General algo	LU-arma	LU-eigen
10	3.58e-4	5.23e-4	2.21e-1	2.34e-1
10^2	2.05e-3	2.50e-3	4.16e0	2.02e1
10^3	2.13e-2	1.97e-2	8.87e0	1.11e4
10^4	2.97e-1	2.73e-1	1.02e3	-
10^5	2.70e0	2.23e0	-	-
10^6	3.40e1	2.29e1	-	-

A trade-off occur between time usage and relative error. As time usage goes up, due to larger array operations as n increases, the relative error diminish (tab. 2). Although the simple algorithm performs better on time usage and memory allocation than a general algorithm, the two methods have the same relative error for all experiments (n = 10, 100, 1000 shown in tab.2). Both LU-decomposition methods had the same relative error.

However, again, they performed worse than the two own-implemented algorithms, with a larger relative error compared to the latter, especially for low values of n. Thus, a more stable performance are provided by the simple and general algorithm, as show in Fig. 3 (Appendix).

Table 2: **Relative error**: Relative error of four different methods with different step sizes, $h = (x_n - x_0)/n)$ for $x \in [x_0, x_n]$. The simple algorithm was tailored to solve the particular 1D Poisson equation in question, whereas the general method was implemented as a general Thomas algorithm. These are compared with LU-decomposition methods provided by the armadillo and eigen libraries, which in this case performed equally well.

n	Simple algo	General algo	LU-decomposition (arma/eigen)
10	2.0335e-01	2.0335e-01	5.4313e-01
10^2	9.4982e-03	9.4982e-03	6.7097e-02
10^3	8.7772e-04	8.7772e-04	6.9287e-03

V Discussion

The results obtained by solving a differential equation, using a finite difference scheme (sec. IV), is discussed and compared with the analytical solution. First, the performance of the different algorithms are discussed in sec. V.I. On this basis, possible improvements are suggested in sec. V.II.

V.I Method Performance

The general and simple algorithm perform equally well in terms of relative error and $L2/h^2$ as a function of datapoints n (Fig. 2 and 3, tab. 2). Even for relatively small values of n, i.e. n = 100, the numerical method has a small relative error ($\varepsilon = 9.4e-03$) in comparison to the LU-decomposition-based solution ($\varepsilon = 6.7e-02$). A solution using LU-decomposition is unable to replicate the analytical analytical solution for $n \leq 100$ (Fig. 1). This artefact of the LU-solution is explained by an unstable $L2/h^2$ for low values of n (Fig. 3).

Time usage is highest for the LU-decomposition generated solution provided libraries `<Eigen/Dense>` and `<armadillo>`. Thus, the Thomas algorithm both have a higher accuracy, i.e. a lower relative error, and a lower time usage, wherein the problem-tailored "simple" algorithm performed best in all benchmarks. Hence, the simple algorithm should

be favored as a method for this specific case, above the other methods.

The higher time usage and higher relative error of LU-decomposition-based methods, followed by the general algorithm, can be explained by number of floating point operations (FLOPs) and memory usage. Regarding time usage, the general and simple algorithm takes in arrays of length n+2 (includes start and endpoint as boundary conditions), instead of a matrix with dimension n$\times$n, which easily will exceed the Random Access Memory (RAM) of the computer. For example, running the LU-programs with n = 10^5 gives an error message, whereas the Thomas algorithms run without problems. As for FLOPs, it can be shown that the general algorithm, simple algorithm and LU-decomposition method has O(9n), O(4n) and $0(2/3n^3)$ FLOPs respectively [1]. Thus, the LU method takes a longer time and is more prone to accumulate round-off errors.

V.II Suggested Improvements

Due to a larger number of FLOPs in the general Thomas algorithm, compared to the simple, it is expected to take longer time than the latter. However, as shown in tab. 1, this in not always the case. By taking a mean of many experiments with same n-values, the stochasticity in time usage would probably even out.

Another small adjustment would be to run the programs with h = (b-a)/n, instead of h = (b-a)/(n-1), to align step sizes of the last (u_n) and next to last (u_{n-1}) solution points. A more important improvement would be to use makefiles and make the code more accessable, through functions, structs and classes.

VI Conclusion

In the quest of solving a differential equation, the one dimensional Poisson equation, three different algorithms are tested and compared to each other and the analytic solution. First a general algorithm with forward and backward substitution for a tridiagonal matrix, then a simplified version of that adjusted to a specific case with identical entries on the diagonals and finally a LU decomposition which made use of armadillo, an external library package. The general and special forward and backward substitution algorithm prove to be more accurate than the LU decomposition who are more prone to accumulate round-off errors. Even with a grid of $n = 100$ points the relative error is quite low 9.4982e-03. The special simplified algorithm has the shortest CPU-time with 21.3 μs for 1000 grid points.

References

[1] Morten Hjorth-Jensen. Computational Physics Lectures: Linear Algebra methods, 2020. Accessed: August 2020.

VII Appendix

VII.I L2-norm/h^2

The L2-norm is plotted as a stability criterion for the simple and general general algorithm, and for the LU-decomposition method provided by armadillo (See sec. II, III and IV.II).

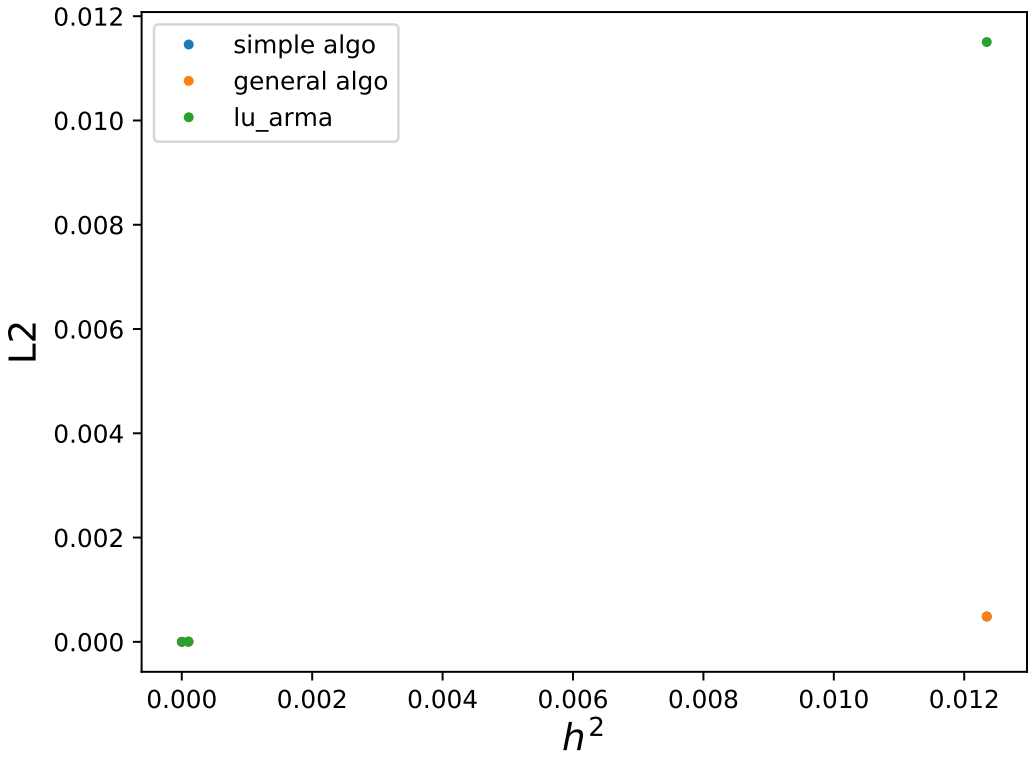


Figure 3: **L2-norm/h^2 for the numerical solutions to a 1D poisson equation:** The L2 is plotted as a function of h^2. Equal L2 values in a range of step sizes (h) indicate a stable algorithm. Two algorithms of different complexity, and LU-decomposition, are compared to the analytical solution. The LU decomposition provided by the armadillo library is especially sensitive to large step values. Parameters: h - step size, L2 - the L2-norm (left to right: simple algo, general algo, lu arma).