FYS4150: Project 1

Jonathan Brakstad Waters

September 7, 2019

Abstract

This project provides an algorithm for solving the one dimensional Poisonn equation. Comparisons are made with other numerical methods, with the conclusion that the specially designed algorithm is superior both with respect to running time and accuracy.

Introduction

In this project, I have looked at the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} = f(x)$$

for a given f, and the solution to this as given by a numerical solver of a system of eqiations, using forward and backward substitutions. In the process, I have investigated different numerical methods and the time they take to execute. I have also compared my numerical solutions to the exact solution, and analyzed how the relative error changes with the step size of the discretisation. In the following, I will present in detail my methods and proceedings, and share my results. Python is used as programming language throughout.

Description of the problem

The basis for this project is the one dimensional Poisson equation with Dirichlet boundary conditions:

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

For this project, we take $f(x)$ to be $f(x) = 100e^{-10x}$, which yields the exact solution $u_{\text{exact}}(x) = 1 - (1 - e^{-10})x - e^{-10x}$. This is useful for later comparision and error analysis.

To solve the equation, we define the discretized approximation to u as v_i with $N + 1$ grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_N = 1$. This gives a step size of $h = \frac{1}{N}$. Thus, the second derivative is approximated as:

$$u''(x) \approx \frac{v_{i-1} - 2v_i + v_{i+1}}{h^2}$$

The Poisson equation then takes the form:

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

$$\implies -v_{i-1} + 2v_i - v_{i+1} = h^2 f_i(x) \coloneqq b_i$$

With a bit of staring, one can easily convince oneself that with i running from 0 to N, the resulting set of linear equations can be expressed as

$$\mathbf{Av} = \mathbf{b}$$

with

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ 0 & 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

Note that the first and last row (corresponding to v_0 and v_N) are omitted, since these values are already given by the boundary conditions. Now, the task to be handled is to solve this set of equations numerically.

Methods

The matrix $\mathbf{A}$ is a tridiagonal matrix. As one can see, most of it's entries are zero, and so we do not need to store the whole matrix, merely the diagonal and off-diagonal non-zero entries. These can be stored in one array each. We will start by making a numerical solver for the general case, where our non-zero elements take arbitrary values. We will call the "main diagonal" entries d_i $(i = 1 \cdots (N-1))$, the "lower diagonal" entries c_i $((i = 1 \cdots (N-2))$ and the "upper diagonal" entries e_i $((i = 1 \cdots (N-2))$.

Having defined these vectors, together with the discrete version of $f(x)$, we can solve the set of equations using the Thomas algorithm for tridiagonal matrices (Thomas, 1949):

$$\begin{aligned} \tilde{d}_1 &= d_1 & \tilde{f}_1 &= f_1 \\ \tilde{d}_i &= d_i - e_{i-1}\frac{c_{i-1}}{\tilde{d}_{i-1}} \\ \tilde{f}_i &= f_i - \tilde{f}_{i-1}\frac{c_{i-1}}{\tilde{d}_{i-1}} \\ v_{N-1} &= \frac{\tilde{g}_{N-1}}{\tilde{d}_{N-1}} \\ v_i &= \frac{\tilde{g}_i - e_i u_{i+1}}{\tilde{d}_i} \end{aligned}$$

Figure 2 shows the results of this algorithm, for different values of N. You can see that the algorithm becomes quite accurate for relatively small values of N.

Counting the floating point operations (FLOPS) needed by this algorithm, we see that they scale as $9N$. However, we can reduce this number to increase the effiency of the program. Note that all the main diagonal entries in $\mathbf{A}$ are equal to 2, and that all the off-diagonal entries are equal to -1. For one thing this means that we can save memory, because we do not need to store arrays for $\mathbf{c}$, $\mathbf{d}$ and $\mathbf{e}$. Furthermore, the Thomas algorithm for the diaogonal terms now yields:

$$\tilde{d}_i = d_i - e_{i-1}\frac{c_{i-1}}{\tilde{d}_{i-1}} = 2 - (-1)\frac{(-1)}{\tilde{d}_{i-1}} = 2 - \frac{1}{\tilde{d}_{i-1}}$$

If you calculate some of these terms, you will quickly realise that

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

This means that we can precalculate this vector, and it does not need to add to our FLOPS count. The fact that our terms for $\mathbf{c}$ and $\mathbf{e}$ are all -1 means that we can simplify further in the rest of the algorithm as well. In total then, these adjustments reduce the total numer of floating point operations to $4N$. In table 3 you can see the resulting change in running time for the program, for values of N ranging from 10^1 to 10^6. The actual results for $\mathbf{v}$ yielded by the more efficient program are nigh identical with those from the general solver, and I omit showing the plots.

It is interesting to look at how the error of our approximation varies with the size of our step size, h. To do this, we run a loop over different values of N. For every N-value, we calculate the solution, and calculate the relative error by the formula:

$$\epsilon_i^{rel} = \left|\frac{v_i - u_i}{u_i}\right|$$

I then plot the maximum value of ϵ_i^{rel} against the step size h, using logarithmic scaling. Running through the loop, this provides the plot as shown in figure 1.

Another option for solving our set of linear equations, is to use an LU decomposition solver for a general NxN matrix. The LU decomposition decomposes a matrix $\mathbf{A}$ into one lower triangular matrix $\mathbf{L}$ and one upper triangular matrix $\mathbf{U}$. These two matrices can then be used to solve equations on the form $\mathbf{Ax} = \mathbf{b}$. To do this, I use the functions lu_factor and lu_solve from the scipy.linalg library. In figure 3 I have plotted the LU solution together with the tridiagonal solution and the exact solution for some values of N. In table 2 you can see how the LU solver compares to the tridiagonal solver running time wise, whilst in table 1, I have compared the error made by the two solvers.

Results and discussion

The goal of this report is to determine what algorithm is best suited for solving the one dimensional Poisson equation. To do this I have looked at the run time and the error for the general Thomas algorithm, the tridiagonal solver for a 2nd derivative matrix, and the LU solver for an N×N matrix.

I will start by rejecting the general Thomas algorithm right away. This algorithm is identical to the special solver, except for the larger amount of FLOPS, and therefore less efficiency. The special case solver provides the same precision with lower runtime, and is therefore a "no brainer". The following is a comparison between the special case tridiagonal solver and the LU solver.

Looking at table 1, you can see that for larger values of N (corresponding to smaller step size), the LU solver seems to be a tiny bit more precise than the tridiagonal solver, and so this might at first glance seem to be a better method. However, there are two things that should give pause for thought.

Firstly, table 2 shows us that the LU solver takes considerably more time to run (for $N = 10^4$ it differs by a factor 10^3), and one has to consider if such a difference in runtime is worth an increase of 10^{-5} in precision.

Secondly, to use the LU solver, one has to use a full matrix, with N×N elements, all taking up 8 bytes of working memory (RAM). For a matrix of size $N = 10^4$, this is no problem - all the numbers only add up to 0,8 gigabytes. Increase N by a factor 10, and your matrix suddenly requires 80 gigabytes of space. This is more than what's available on most normal computers (most of which has 8 or 16 gigabytes of RAM available). This means that the error of 10^{-7} is the smallest error you can possibly get with the LU solver under normal circumstances.

N	$log_{10}(\epsilon_{tri})$	$log_{10}(\epsilon_{LU})$	$log_{10}(\epsilon_{tri}) - log_{10}(\epsilon_{LU})$
10^1	$-1,1005822227587367$	$-1,1005822227587367$	$0,00000000e+00$
10^2	$-3,0793983614099818$	$-3,0793983614025606$	$-7,42117479e-12$
10^3	$-5,0791834174211967$	$-5,0791833992507156$	$-1,81704811e-08$
10^4	$-7,0791811315291371$	$-7,079198287911642$	$1.71563825e-05$

Table 1: *Error made by the tridiagonal solver and the LU solver, and the difference between the two, for increasing values of N.*

The observant reader will have noted that the tridiagonal solver can be run for much larger values of N, taking considerably shorter time, as shown in table 3. Running the tridiagonal solver for $N = 10^6$ only takes a tenth of the time it takes to to run the LU solver for $N = 10^4$. And in fact, this is all we need - the runtimes for larger N are more or less uninteresting.

N	Tridiagonal Solver	LU Solver
10^1	$1,7 \cdot 10^{-5}$	10^{-4} - 10^{-2}
10^2	$1,7 \cdot 10^{-4}$	10^{-3} - $2,5 \cdot 10^{-2}$
10^3	$1,8 \cdot 10^{-3}$	$3,3 \cdot 10^{-2}$
10^4	$1,6 \cdot 10^{-2}$	$1,34 \cdot 10^{1}$

Table 2: *Table showing run times for the 2nd derivative tridiagonal solver and the LU solver, respectively. The time values are given in seconds.*

Looking at figure 1, we can see that when we increase N, the error reduces with a slope of 2 on the logarithmic scale. If you plot a function $f(x) = x^2$ with logarithmic axes, you get a straight line with slope 2, and so this result corresponds well with our expected result for the mathematical error, which should go as h^2 (Hjort-Jensen, 2015). However, note that the slope flattens out, and actually turns at $N = 10^6$. This is not due to the mathematical error. Rather, it is due to loss of numerical precision. When the step size becomes too small, the computer is no longer able to represent numbers with sufficient precision, and this leads to erros in the calculations. Therefore there is no point in using a value for N larger than 10^6.

With this value for N, the tridiagonal solver has an error of approximately 10^{-10} (figure 1), a factor of 10^3 less than the LU solver for $N = 10^4$.

N	General Case	Special Case
10^1	$1,0 \cdot 10^{-4}$	$1,7 \cdot 10^{-5}$
10^2	$3,2\dot{1}0^{-4}$	$1,7 \cdot 10^{-4}$
10^3	$4,8 \cdot 10^{-3}$	$1,8 \cdot 10^{-3}$
10^4	$4,2 \cdot 10^{-2}$	$1,6 \cdot 10^{-2}$
10^5	$2,6 \cdot 10^{-1}$	$1,7 \cdot 10^{-1}$
10^6	$2,6 \cdot 10^{0}$	$1,4 \cdot 10^{0}$

Table 3: *Table showing run times for the general tridiagonal solver and the 2nd derivative tridiagonal solver, respectively. The time values are given in seconds.*

One more thing to take note of, is that the time consuming part of the LU solver is the decomposition function. This function is not dependent on the right hand side of the equation, and so one only needs to perform it once, for a number of different right hand side expressions. The lu_solve function takes less time to perform. One could then ask if maybe the LU solver is better than the tridiagonal solver for problems with a large number of different right hand sides. Note that the precision of the tridiagonal solver is still better, due to the impossibility of running an LU solver for large values of N.

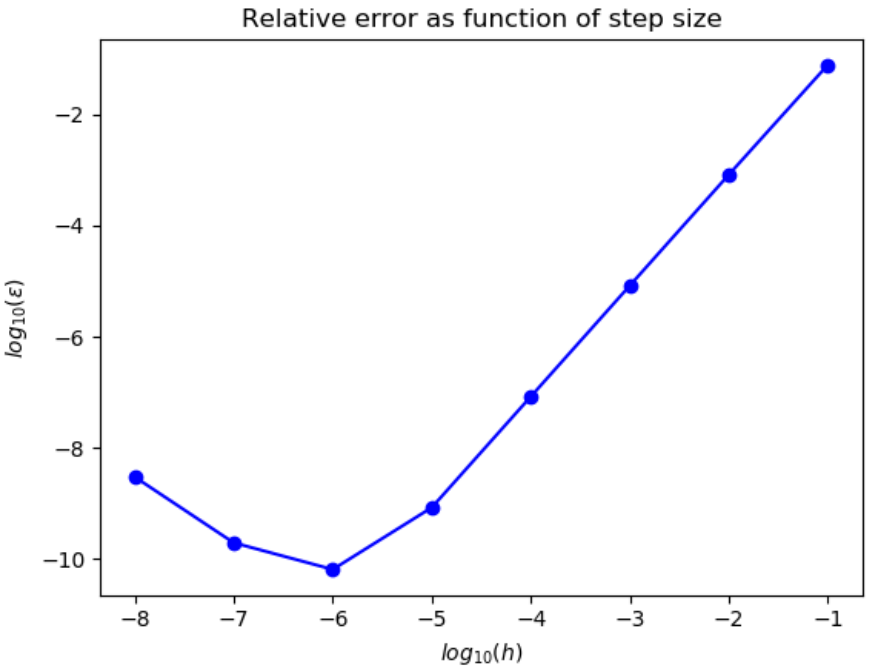

Figure 1: *Relative error plotted as function of step size. We observe that the error decreases with bigger N, until a certain point, where the error starts increasing againg.*

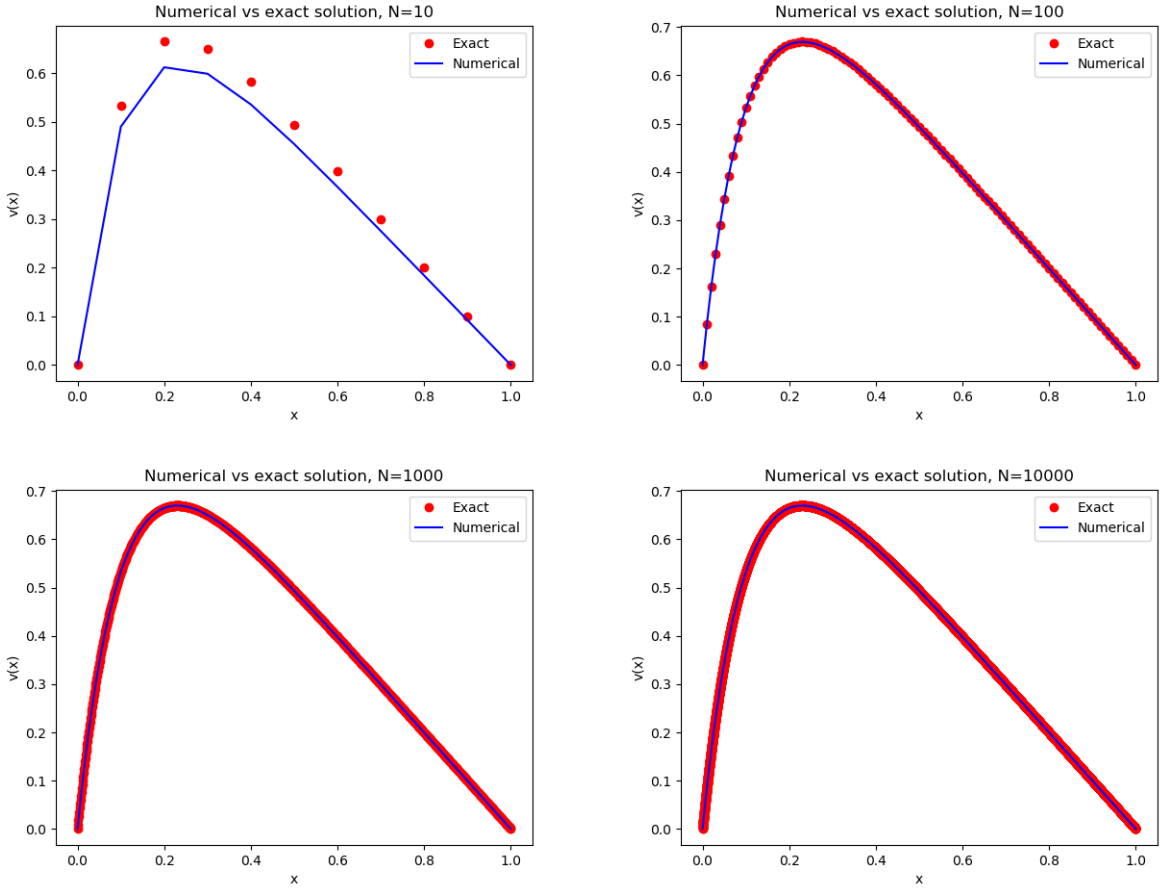

Figure 2: *Comparison of numerical solution against the exact analytical solution for the equation* $-u''(x) = 100e^{-10x}$*, using the Thomas algorithm for a general tridiagonal matrix*

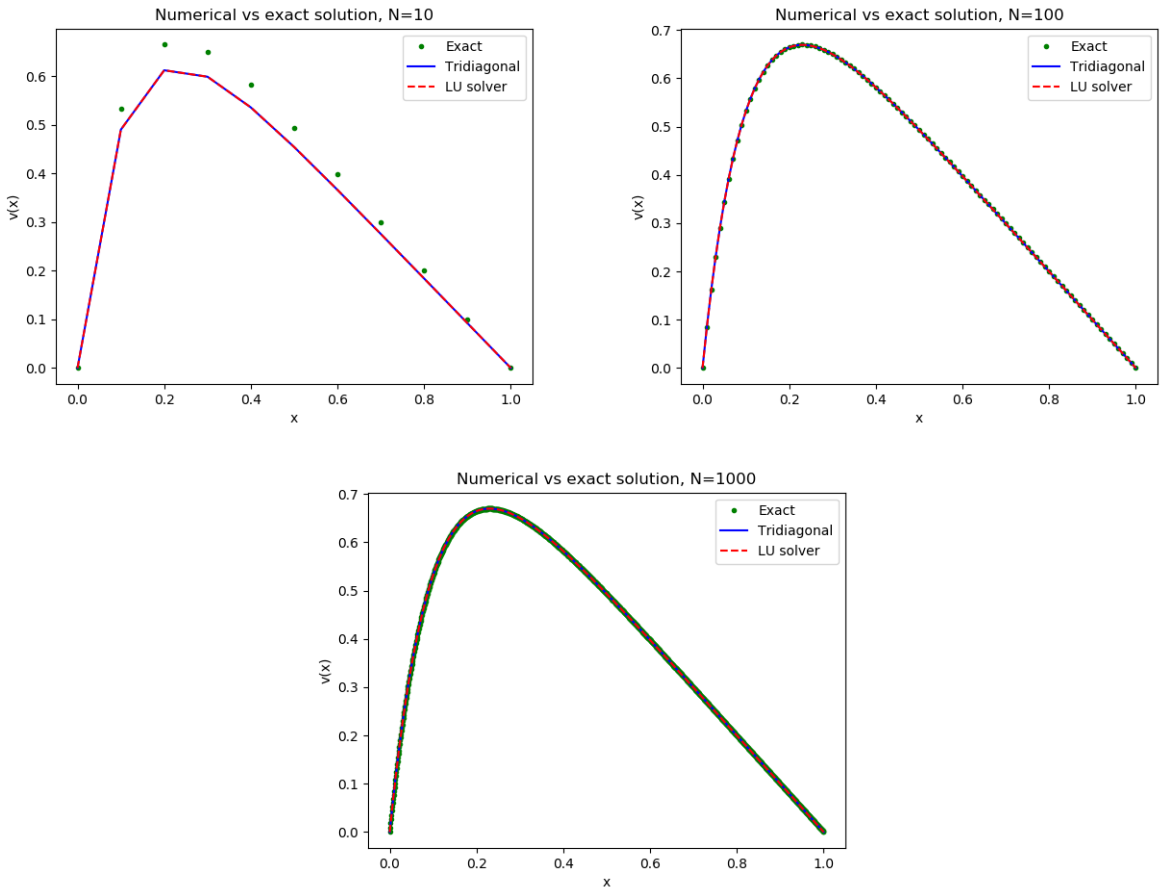

Figure 3: *Solution yielded by the LU solver in scipy, together with the specialized tridiagonal solver and the exact solution.*

Conclusion

There seems to be no doubt that for solving the one dimensional Poisson equation, the specialized tridiagonal solver is superior to a general tridiagonal or LU solver - both in terms of effiency and precision. That being said, there is possibility that the LU solver is more useful if you plan to change your right hand side function frequently.

For further work, it could be interesting to look closer at the algorithms presented, streamline them, and make a set of functions for different uses, depending on efficiency and precision. Also, one could investigate if different right hand sides or boundary conditions affect the precision of the various algorithms.

References

Hjorth-Jensen, M (2015) Lecture Notes. Department of Physics, University of Oslo. Downloaded 07.09.2019 from:
https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Lectures

Thomas, L.H. (1949) Elliptic Problems in Linear Difference Equations over a Network. Watson Sc. Comp. Lab. Rep., Columbia University, New York

Developed Code files

Code files can be found in github repository at:
https://github.com/HolyWaters95/FYS4150-Project-1

- General Solver.py
- Special Case Solver.py
- Error Analysis.py
- LU Solver.py
- Error Analysis.py