FYS4150 - Computational physics
Project 1

Fred Marcus John Silverberg

September 10, 2018

Abstract
The basis of the project is to study numerical solutions for the Poisson equation in one dimension with the restrictions of Dirichlet boundary conditions. Since the equation is fairly known and simple to solve, the emphasis lie on the efficiency and accuracy of the methods. The later is possible due to knowledge of the analytical solution. Included numerical methods are the Thomas algorithm as well as LU decomposition. Results will vary with the number of grid points and the deepest conclusion is drawn from the fact that an increase of grid points do not increase the accuracy of the approximations.

Table of contents

1. Introduction
The Poisson equation in one dimension is classified as a linear second-order differential equation. Our source function and its relevant derivatives are defined on the interval [x] ∈ [0, 1]. This allows us to analyse the problem as a set of linear equations, in the form of [Au=b]. Here [A] will turn out to be a tridiagonal matrix of constants, [u] our unknown vector and [b] our known source function in discretized form. The chosen methods are applied through python code for various grid points in the interval [n] ∈ [10, 10^(7)]. An accuracy comparison is done visually by appended graphs as well as numerically by computing the maximal relative error for various [n]. The final step in the analysis is to evaluate the efficiency in terms of computational time and the total number of flops required.

2. Method

The source function is chosen as[$f(x) = 100e^{-10x}$], on the interval x ∈ [0, 1], with the analytical solution given by [$u(x) = 1 - (1 - e^{-10})x - e^{-10x}$]. Boundary conditions are: [u(0) = u(1) = 0]

- Sprung from the Taylor expansion we have: $f''(x) \approx (f(x+h) + f(x-h) - 2(fx))/h^2 + O(h^2)$
- Introduction of grid points [n] and step size [h = 1/(n+1)]
- Discretizing above leads to: $f''(x) \approx f_i = -(u_{i+1} + u_{i-1} - 2u_i)/h^2$, for [i] ∈ [0,1,..,n]
- Written as a set of linear equations [Au=b] for $b_i = (f_i h^2)$, [x] ∈ (0, 1) we have:

2	-1	0	0	*	u_1	=	b_1
-1	2	-1	0		u_2		b_2
0	-1	2	..		u_3		b_3
0	0	..	..		..		..

[1]

From the above derivation a tridiagonal matrix appear [A]. This allows us to simplify our gaussian based algorithm. But I will start by implementing the Thomas algorithm with the assumption that the values of the elements in respective diagonal of [A] are unknown.

Measurement of computational time will be done by using the inbuilt time function in python.

2.1 Thomas algorithm for a general tridiagonal matrix

For a general tridiagonal problem we then have:

b_1	c_1	0	0	*	u_1	=	b_1	**[1]**
a_1	b_2	c_2	0		u_2		b_2	
0	a_2	b_3	..		u_3		b_3	
0	0	..	..		..		..	

The first operation in solving **[1]** with an unknown matrix [A] is to row reduce the system.
A first step is to multiply row(1) by (a_0/b_0), a second to subtract row(1) from row(2). Continuation of this process would eliminate the lower diagonal, hence yield an expression for u_{n-2}. By using the expression for u_{n-2} the system can be solved by substitution. The idea is that since fulfilling this pattern yields the solution, one can identify the governing operations and use them to solve [Au=b].

- Decomposition phase: [i] ∈ [1,..,n]
  $m = a_i / b_{i-1}$
  $b_i = b_i - (c_{i-1})*m$
  $b_i = b_i - (b_{i-1})*m$

[1] Full derivation shown in Appendix 1

- Backward substitution phase: [i] = n
  $u_{i-1} = \tilde{b}_{i-1} / b_{i-1}$
- Backward substitution phase: [i] ∈ [n-2,n-3,...,0]
  $u_i = (\tilde{b}_i - c_i * u_{i+1}) / b_i$

Given the above operations the efficiency can be estimated by calculating the number of flops required for solving the system for a specific [n]. A flop can be clarified as an operation of either addition, subtraction, multiplication or division.

- Number of flops for a general tridiagonal matrix problem:
  Decomposition phase: [5*(n-1)] flops
  Substitution phase: [1+3*(n-1)] flops
  Total: [1+8*(n-1)] flops

The information above has been implemented in python.[2]

2.2 Thomas algorithm for our special tridiagonal matrix

For our specific case of the one dimensional Poisson equation, where diagonals are known as $a_i = c_i = -1$ and the main diagonal $b_i = 2$, the above algorithm can be further simplified.

- Decomposition phase: [i] ∈ [1,..,n]
  $b_i = (i+1)/i$
  $\tilde{b}_i = b_i + (\tilde{b}_{i-1})/b_{i-1}$
- Backward substitution phase: [i] = n
  $u_{i-1} = \tilde{b}_{i-1} / b_{i-1}$
- Backward substitution phase: [i] ∈ [n-2,n-3,...,0]
  $u_i = (\tilde{b}_i + u_{i+1}) / b_i$

For this specific case the efficiency is drawn from:

- Number of flops for this special tridiagonal matrix problem:
  Decomposition: [3*(n-1)] flops
  Substitution: [1+2*(n-1)] flops
  Total: [1+5*(n-1)] flops

The information above has been implemented in python.[3]

[2] Code for general case is to be found in Appendix 2
[3] Code for specific case is to be found in Appendix 3

2.3 LU decomposition

For the special case, matrix A is indeed invertible, which allows me to evaluate the problem by LU decomposition. The idea behind LUD is to decompose matrix A into L*U, where L and U are triangular matrices. L simply holds the lower diagonals and U holds the upper, as shown below.

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

Following this decomposition [Au=*b*] => [L(Uu)=*b*] => [Lw=*b*] , [Uu = w].
As seen this results is a system of two linear sets of equations. By solving [Lw=*b*] for w one can later solve [Uu=w] for u and a final solution is simply found.

For this case the efficiency is drawn from:

- Number of flops by LU decomposition:
  Decomposition: 2/3*(n^3) flops
  Substitution: 2*(n^2) flops
  Total: 2/3*(n^3) + 2*(n^2) flops

I have implemented the idea by using pre built functions in the library scipy.[4]

2.4 Relative error estimation

The calculation for the relative error is straightforward.

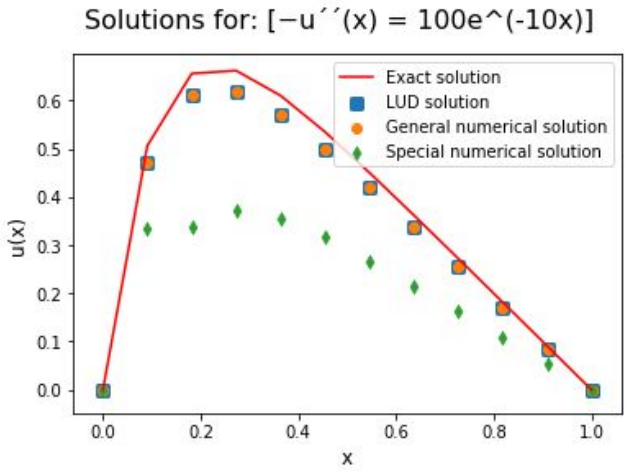
[$E(i) = \log_{10}(|\text{numerical solution}(i) - \text{exact solution}(i)| \,/\, |\text{exact solution}(i)|)$]

The maximum of the relative errors for specific inputs of [n] is obtained in a table(1) under Results.

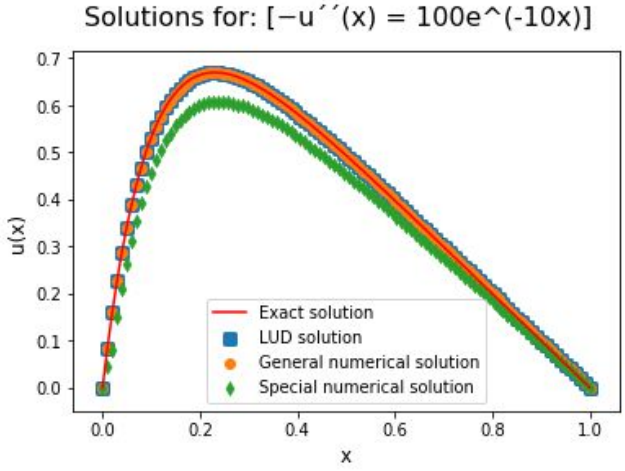
[4] LU decomposition code is to be found in Appendix 4

3. Results

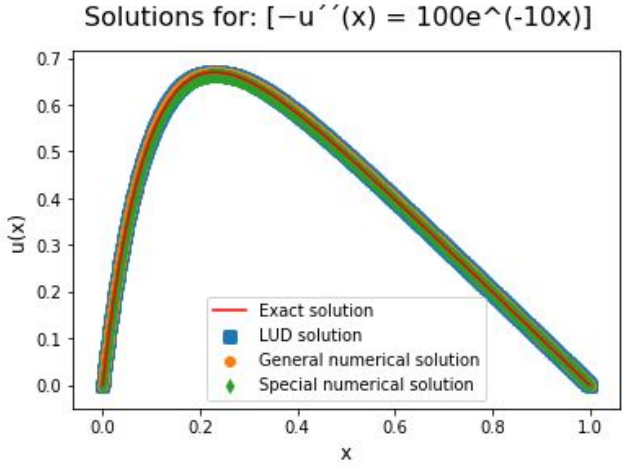
3.1 Graphical comparison of numerical solutions versus the exact solution.



Figure(1): Grid points = n = 10

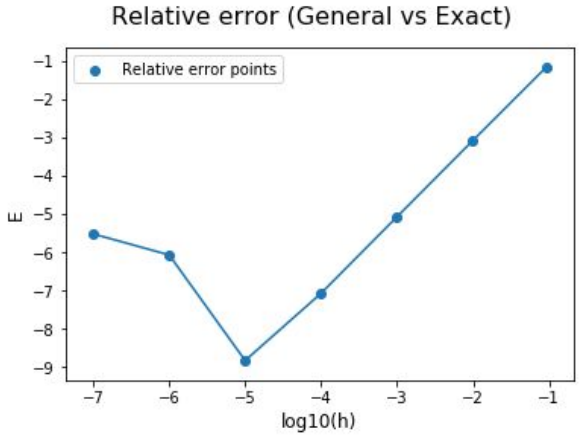


Figure(2): Grid points = n = 100



Figure(3): Grid points = n = 1000

3.2 Relative error



Figure(4): Relative error measured in logarithmic scale by equation from [2.3], see table below.

n	log10(h)	Max. relative error
10^1	-1.04	-1.18
10^2	-2.00	-3.09
10^3	-3.00	-5.08
10^4	-4.00	-7.08
10^5	-5.00	-8.84
10^6	-6.00	-6.08
10^7	-7.00	-5.53

Table(1): Table with $\log_{10}$ values of step length [h = 1/(n+1)] with the maximum relative errors for respective [n] value. The relative errors is between the general solution and the exact solution.

3.3 Total number of flops

n	General algorithm	Special algorithm	LU Decomposition
10^1	73	46	867
10^2	793	496	6.67*10^(5)
10^3	7993	4996	6.67*10^(8)
10^4	79993	49996	6.67*10^(11)
10^5	799993	499996	6.67*10^(14)
10^6	7999993	4999996	6.67*10^(17)

Table(2): Calculated with the formulas from the method section.

3.4 CPU time

n	General algorithm	Special algorithm	LU Decomposition
10^1	1.90*10^(-5)	1.64*10^(-5)	2.27*10^(-4)
10^2	2.54*10^(-4)	2.03*10^(-4)	7.06*10^(-4)
10^3	1.50*10^(-3)	1.16*10^(-3)	0.0341
10^4	0.0148	0.0118	27.475
10^5	0.1396	0.1030	'MemoryError'
10^6	1.1761	0.9476	'MemoryError'

Table(3): This summarize the time it takes for computing the problem for each numerical method. Calculated with built in code for python, measured in [Seconds].

4. Discussion

Figure(1) and figure(2) easily identifies the general algorithm [2.1], where the elements in each diagonal is unknown, as a much more accurate approximation compared to the special algorithm [2.2]. However, as the number of grid points grow it becomes harder to visually separate them. The method of LU decomposition [2.3] comes out to be identical to the general. The fact that they become identical further strengthen both their reliability.

From table(1) there is a pattern for [n] as it grows up to about 10^(5). The slope is roughly 2 from n=10^1 to n=10^5, which is what to be expected from the "Big O" of the taylor expansion, where we limit our information. For n=10^6 the relative error increases, which indicates that an increase of grid points beyond n~10^5 would only contribute negative. This contradict the general idea that we can make n infinite small. The reason behind this is the numerical round off errors in the computation.

A quick look at the total number of flops and CPU time, [table 2,3] reveals that the special algorithm is considerably more efficient and less time consuming. In combination with the less accurate approximation, this is a trade of one would have to consider as [n] grows. Well, at least if your computer is from the stone age or if you have multiple sets to calculate. The amount of flops required for a LU decomposition escalates as n grows, which is why my computer with 8GB of ram had no chance of calculating n=10^5.

5. Conclusions

Both Thomas algorithms [2.1],[2.2] approximates the analytical solution in a satisfactory manner and as expected from the Taylor expansion. It is also obvious from the figures(1,2,3) that as [n] increases from 10 to 1000, the numerical solutions becomes more accurate and close to identical. But a further analysis in relative error against the analytical solution reveals that as [n] grows above n=10^5 the methods experience a loss of precision. Since that is the case, it is not necessary to go beyond n~10^5.

The method of LU decomposition [2.3] comes out as a bad option for a case where the matrix is of the tridiagonal type, since it is both less efficient and more time consuming. This while the general algorithm [2.1] generates identical results in combination with higher efficiency and lower

computational time. It is also apparent that the method of LU decomposition is in a higher degree limited by the amount of RAM in the computer.

Fast check table:

Measurement:	General algorithm [2.1]	Special algorithm [2.2]	LU decomposition [2.3]
Max. relative error	First place	Third place	First place
Nr of Flops	Second place	First place	Third place
CPU time	Second place	First place	Third place
RAM limitation	Second place	First place	Third place

6. References

Morten. Hjorth-Jensen. Computational Physics, Lecture notes fall 2015. Department of Physics, University of Oslo, 2015.

7. Appendices

1
https://github.com/silverberg89/FYS4150/blob/master/Project1/Taylor_expansion.jpg
2
https://github.com/silverberg89/FYS4150/blob/master/Project1/Thomas_algo_general.py
3
https://github.com/silverberg89/FYS4150/blob/master/Project1/Thomas_algo_special.py
4
https://github.com/silverberg89/FYS4150/blob/master/Project1/LUD.py
5
Access to all material can be found at: https://github.com/silverberg89/FYS4150/tree/master/Project1