Project 1

Poisson's Equation in One Dimension

FYS4150 Computational Physics

Link to github repository for codes:
https://github.uio.no/sidrar/FYS4150-H20_project1

Shafaq Naz Sheikh, Sidra Rashid
6th September 2020

ABSTRACT In this project we study Poisson's equation in one dimension, with Dirichlet boundary conditions. We will solve the equation using two methods, the Thomas algorithm and the LU decomposition.

INTRODUCTION

Many important differential equations in Science can be written as linear second-order differential equations

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x)$$

Where f is normally called the inhomogeneous term, and k^2 is a real function. A classical equation from electromagnetism is Poisson's equation. The electrostatic potential Φ is generated by a localized charge distribution ρ(**r**). In three dimensions it reads

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r})$$

With a spherically symmetric Φ and ρ(**r**) the equations simplifies to a one dimensional equation in r, namely

$$\frac{1}{\mathrm{r}^2}\frac{\mathrm{d}}{\mathrm{dr}}\left(\mathrm{r}^2\frac{\mathrm{d}\Phi}{\mathrm{dr}}\right) = -4\pi\rho(r)$$

which can be rewritten via a substitution $\Phi(r) = \phi(r)/r$ as

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

The inhomogeneous term f or source term is given by the charge distribution ρ multiplied by r and the constant -4π. We will rewrite this equation by letting $\phi \to u$ and $r \to x$. The general one-dimensional Poisson equation reads then

$$-u''(x) = f(x)$$

We will solve the equation with Dirichlet boundary conditions, that is

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

In this specific case, we will assume that the source term is $f(x) = 100e^{-10x}$, and keep the same interval and boundary conditions, so the equation becomes;

$$-u''(x) = 100e^{-10x}, \quad x \in (0,1), \quad u(0) = u(1) = 0$$

This equation has a closed-form solution given by

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}$$

This will help us figure out how good of an approximation our numerical solution is to the analytical solution.

In order to discretize the equation, we chose n grid points for the x-values. We approximate $u(x)$ by u_i, with $x_i = ih$ with $x_0 = 0$ and $x_n = 1$. The step length, h , is defined by $h = 1/(n+1)$. We discretize the boundary conditions by taking $u_0 = u_{n+1} = 0$. The next step is to approximate the second derivative, this can be done by a Taylor expansion.

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2!}u''(x)$$
$$u(x-h) = u(x) - hu'(x) + \frac{h^2}{2!}u''(x)$$

Now adding the expressions on both sides gives,

$$u(x+h) + u(x-h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + u(x) - hu'(x) + \frac{h^2}{2}u''(x)$$
$$u(x+h) + u(x-h) = 2u(x) + h^2\, u''(x)$$
$$\Rightarrow\ u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

So the approximation becomes $u''(x) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$ for $i = 1, .., n$. The approximation becomes;

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = f_i$$

Where $f_i = f(x_i)$. We can rewrite this as a set of linear equations, if we first multiply by h^2 on both sides, we get,

$$u_{i+1} - 2u_i + u_{i-1} = h^2 f_i$$

Let us see what happens to this equation for different values of i,

$$\begin{array}{ll} i = 1 & u_2 - 2u_1 + u_0 = g_1 \\ i = 2 & u_3 - 2u_2 + u_1 = g_2 \\ \ldots\ldots & \\ \ldots\ldots & \\ i = n-1 & u_n - 2u_{n-1} + u_{n-2} = g_{n-1} \end{array}$$

Where $g_i = h^2 f_i$ for $i = 1, \ldots, n-1$. From the boundary conditions we know that u_0 and u_n are zero, so we have rows of variables. We can write this in a matrix A as

$A = \begin{bmatrix} -2 & 1 & \ldots & 0 \\ 1 & -2 & 1 & \ldots \\ \ldots & 1 & -2 & 1 \\ 0 & \ldots & 1 & -2 \end{bmatrix}$ This is a tridiagonal matrix, it is an approximation of the second derivative. The equation now becomes

$$\begin{bmatrix} -2 & 1 & \ldots & 0 \\ 1 & -2 & 1 & \ldots \\ \ldots & 1 & -2 & 1 \\ 0 & \ldots & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ \vdots \\ g_{n-1} \end{bmatrix}$$

which can be written as

$$A\boldsymbol{u} = \boldsymbol{g}$$

The matrix A has dimensionality $A \in \mathbb{R}^{n-1 \times n-1}$ and the vectors **u** and **g** have dimensionality $\boldsymbol{u}, \boldsymbol{g} \in \mathbb{R}^{n-1}$.

ALGORITHMS

To solve this system, we will use two different algorithms, the Thomas algorithm and the LU decomposition.

1. Thomas algorithm

For simplicity let us look at the case for $n = 4$. Then the general augmented matrix will look like

$$A = \begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ a_1 & b_2 & c_2 & 0 & g_2 \\ 0 & a_2 & b_3 & c_3 & g_3 \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix}$$

we perform Gaussian elimination on this matrix. First we want to all the elements below the main diagonal to be zero, that is, we want all the a_i's to be zero, then we will have an upper triangular matrix, and can use backward substitution to find the unknown u. The algorithm for achieving this is demonstrated below;

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ a_1 & b_2 & c_2 & 0 & g_2 \\ 0 & a_2 & b_3 & c_3 & g_3 \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix}$$

The first row remains unchanged, since we want a_1 to be zero. To get $a_1 = 0$, perform $row(2) - (\frac{a_1}{b_1}\ row(1))$, the resulting matrix looks like this. Note that the rows 3 and 4 remain unchanged.

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ 0 & b_2 - \frac{c_1 a_1}{b_1} & c_2 & 0 & g_2 - \frac{g_1 a_1}{b_1} \\ 0 & a_2 & b_3 & c_3 & g_3 \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix}$$

We rewrite element a_{22} and a_{25}, with this notation, the matrix can also be rewritten as it can be seen below.

$$a_{22} : \widetilde{b_2} = b_2 - \frac{c_1 a_1}{b_1}\ and\ a_{24} : \widetilde{g_2} = g_2 - \frac{g_1 a_1}{b_1}$$

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \widetilde{g_2} \\ 0 & a_2 & b_3 & c_3 & g_3 \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix}$$

In order to get $a_2 = 0$ we have to perform $row(3) - (\frac{a_2}{\tilde{b}_2}\ row(2))$. This leads to the matrix below. Similar to the previous step, we rewrite element a_{33} and a_{35} .

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \widetilde{g_2} \\ 0 & a_2 & b_3 - \frac{c_2 a_2}{\tilde{b}_2} & c_3 & g_3 - \frac{\widetilde{g_2} a_2}{\widetilde{b_2}} \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix} \sim \begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \widetilde{g_2} \\ 0 & 0 & \tilde{b}_3 & c_3 & \widetilde{g_3} \\ 0 & 0 & a_3 & b_4 & g_4 \end{bmatrix}$$

In the final step we want get $a_3 = 0$ we have to perform $row(4) - (\frac{a_3}{\widetilde{b_3}}\ row(3))$. This leads to the matrix below. Similar to the previous step, we rewrite element a_{44} and a_{45}, then the final matrix becomes :

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & g_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \widetilde{g_2} \\ 0 & 0 & \tilde{b}_3 & c_3 & \widetilde{g_3} \\ 0 & 0 & 0 & \tilde{b}_4 & \widetilde{g_4} \end{bmatrix}$$

The general formula for the b's and g's is:

$\tilde{b}_1 = b_1$ $\tilde{b}_i = b_i - \frac{c_{i-1} a_{i-1}}{\tilde{b}_{i-1}} \quad for\ i = 2, \dots, n-1$	$\tilde{g}_1 = g_1$ $\tilde{g}_i = g_i - \frac{\tilde{g}_{i-1} a_{i-1}}{\tilde{b}_{i-1}} \quad for\ i = 2, \dots, n-1$

The next step is to perform the backward substitution, and solve for u.

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 \\ 0 & 0 & 0 & \tilde{b}_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ \tilde{g}_2 \\ \tilde{g}_3 \\ \tilde{g}_4 \end{bmatrix}$$

The general formula for finding u is,

$$u_{n-1} = \frac{\tilde{g}_{n-1}}{\tilde{b}_{n-1}}$$
$$u_i = \frac{(\tilde{g}_i - c_i u_{i+1})}{\tilde{b}_i} \quad for\ i = n-2, \dots, 1$$

In this specific case the algorithm can be simplified by considering the fact that the matrix A is quite sparse, and has identical values on the main diagonal, and identical values above and below the main diagonal. The matrix A occurs as a result of the discretization of the second derivative. All the $b_i = 2$ and $a_i = c_i = -1$. The algorithm then becomes;

Forward substitution	Backward substitution
$\tilde{b}_1 = b_1\ ;\ \tilde{g}_1 = g_1; \quad for\ i = 2, \dots, n-1$ $\tilde{b}_i = \frac{(i+1)}{i}$ $\tilde{g}_i = g_i - \frac{\tilde{g}_{i-1}}{\tilde{b}_{i-1}}$	$u_{n-1} = \frac{\tilde{g}_{n-1}}{\tilde{b}_{n-1}}\ , \quad for\ i = n-2, \dots, 1$ $u_i = \frac{(\tilde{g}_i + u_{i+1})}{\tilde{b}_i}$

The algorithm can be stated as

Forward part
b_tilde[1] = b1 ; g_tilde[1] = g1
For i in range(2,n-1):
b_tilde[i] = (i+1)/i
g_tilde[i] = g[i] - (g_tilde[i-1]/ b_tilde[i-1])
end

Backward part
u[n-1] = g_tilde[n-1]/b_tilde[n-1]
For i in range(n-2,1):
u[i] = (g_tilde[i]-u[i+1])/b_tilde[i]
end

2. LU Decomposition

If the matrix A is non-singular(invertible), then A can be written as a product of two matrices, for example consider the case with n=4, then the matrix A can be written as:

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

$$\boldsymbol{A} \qquad\qquad \boldsymbol{L} \qquad\qquad \boldsymbol{U}$$

$$A = LU$$

Where L is a lower triangular matrix and U is an upper triangular matrix. This method helps us solve a system of linear equations $Ax = b$ in an elegant way.

$Ax = b$ can be rewritten by using the LU decomposition of A,

$$A\boldsymbol{x} = L(U\boldsymbol{x}) = L\boldsymbol{w} = \boldsymbol{b}$$
$$1.\ L\boldsymbol{w} = \boldsymbol{b}$$
$$2.\ U\boldsymbol{x} = \boldsymbol{b}$$

First solve $L\boldsymbol{w} = \boldsymbol{b}$ for **w**, this can easily be done by forward substitution as L is a lower triangular matrix. Then solve $U\boldsymbol{x} = \boldsymbol{b}$ for **x**, this can easily be done by backward substitution as U is an upper triangular matrix.

Comparing the different algorithms

Let us count the number of FLOPs in each algorithm, a FLOP is a floating point operation, and includes addition, subtraction, division and multiplication.

- The general Thomas algorithm
  For the forward substitution, we have to calculate $\tilde{b}_i$, this involves one multiplication, one division and one subtraction, so it requires 3 FLOPs. By the same

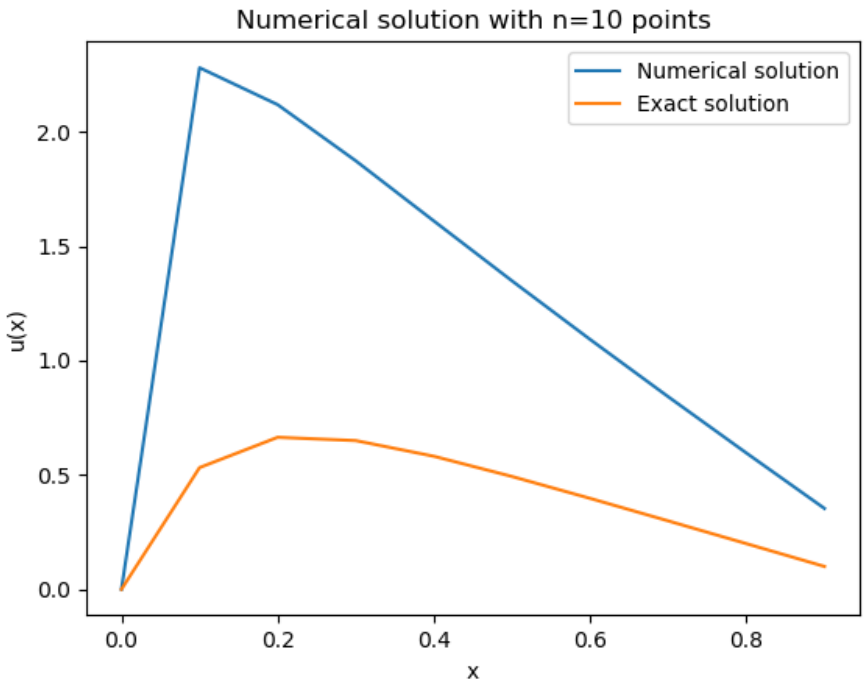
argument, calculating $\tilde{g}_i$ also requires 3 FLOPs, in total the forward substitution requires $6(n-2) = 6n$ FLOPs as the loop runs n-2 times. The backward substitution requires 3n FLOPs.
Total = $6n + 3n = 9n = O(n)$

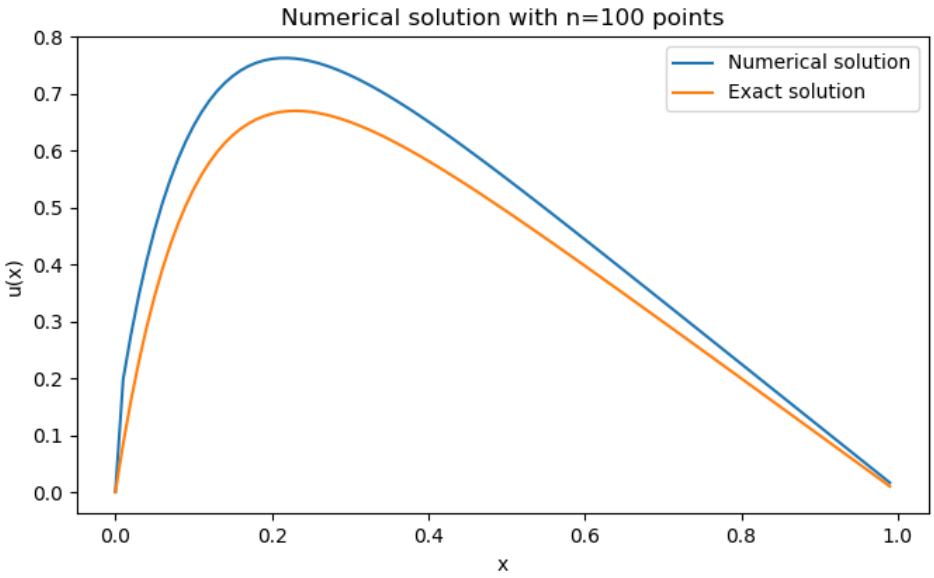
- The Thomas algorithm for the tridiagonal matrix
  Forward substitution: $4n$ FLOPs
  Backward substitution: $2n$ FLOPs
  Total = $4n + 2n = 6n = O(n)$

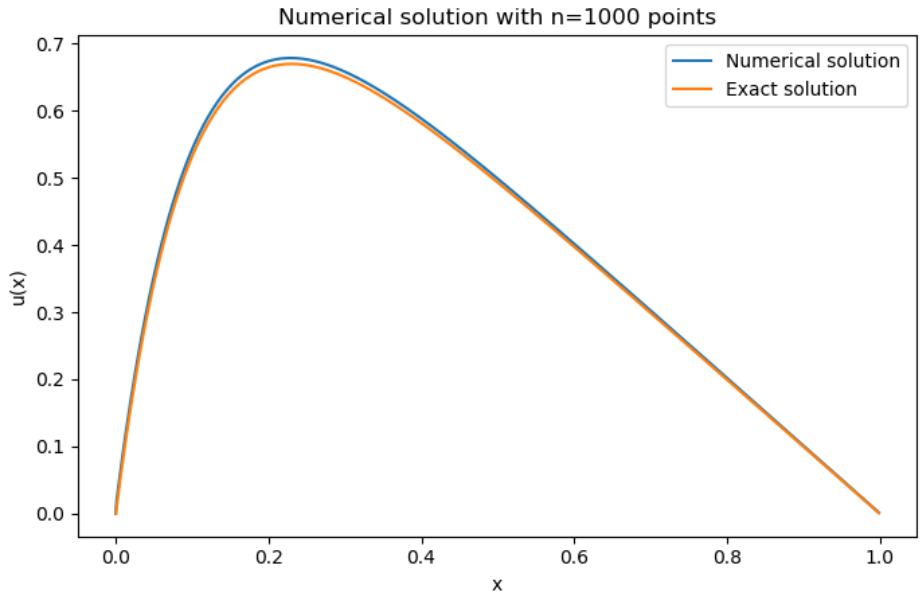
- The LU decomposition
  Total = $O(\frac{2}{3}n^3)$

RESULTS AND CONCLUSION

Here are the results that we got for n=10, n=100 and n=1000. We plotted our results along with the analytical solution to see how accurate the numerical solution is.







As it can be seen from the plots above, as n increases, the numerical solution becomes a better approximation to the analytical solution. Although the three algorithms solve for the unknown u(x), they have different runtimes, let us take a look at the running time of the algorithms.

Run time in seconds				
n	n with exponent	General algorithm	Tridiagonal case	LU decomposition
10	10	0.0000061	0.0000042	0.0865198
100	10^2	0.0000113	0.0000076	0.0774842
1000	10^3	0.0001067	0.0000582	0.134051
10000	10^4	0.0014177	0.000678	5.0226
100000	10^5	0.0160685	0.0081873	-
1000000	10^6	0.0160685	0.0732718	-

The best algorithm would be to use the Thomas algorithm as compared to the LU decomposition. Firstly we can see that the Thomas algorithm allows us to run the program for n as large as10^6 while the LU decomposition can only go up to10^4, at least this is the case for an average computer. When we use the Thomas algorithm to find the unknown u, the program only has to store three vectors of size n, one for the diagonal elements, one for the elements above the diagonal, and for the elements below the diagonal, we then use forward substitution and backward substitution to find the unknown u. In the special case where the matrix A is the tridiagonal, as it is in this problem, the computations are further simplified since all the elements above and below the diagonal are identical. This reduces the number of FLOPs thus making the program run slightly faster. However if we try to solve $A\boldsymbol{u} = \boldsymbol{g}$ by using the LU decomposition, the runtime is much slower. In order to compute the LU decomposition, the program has to first store the original matrix, which will have size $n \times n$, so even for small values of n, the amount of space needed is n squared. The program has to store one matrix for L and one for U, this requires a lot of memory and thus is only good enough for small values of n. The Thomas algorithm runs approximately in linear time, while the LU decomposition method runs in cubic time. In this specific case, the matrix A is sparse, and mostly contains zeroes, therefore storing the zeroes would be inefficient as they remain unchanged and do not contribute to finding the solution u. The best algorithm to use would be the tridiagonal matrix version of the Thomas algorithm, as we only need to store the elements on, above and below the main diagonal. The fastest algorithm to find the solution vector **u** would be to use the tridiagonal algorithm. This can also be seen by the runtime table shown above, for example for $n = 10^4$, the general algorithm takes 0.0014177 sec while the specialized tridiagonal algorithm takes 0.000678 seconds, this shows that our specialized algorithm is the most efficient algorithm to apply in this specific case. When it comes to the error, let us take a look at the maximum relative error for each step length.

Relative error		
n	n with exponent	Maximum relative error
10	10	0.363824
100	10^2	-1.00057
1000	10^3	-2.01591
10000	10^4	-3.01745
100000	10^5	-4.0176
1000000	10^6	-5.01762
10000000	10^7	(Segmentation fault)

In the table above, the relative error has been calculated for the specific algorithm. The relative error starts out small, and we get a good approximation, but for large values of n, the error begins to accumulate, and the results we get are not reliable anymore. This happens mainly due to loss of numerical precision. Numbers are represented in the binary system in the computer, and representing very small numbers is challenging.