SOLVING THE POISSON-EQUATION IN ONE DIMENSION: TRIDIAGONAL MATRIX ALGORITHM AND LU-DECOMPOSITION

FYS3150: COMPUTATIONAL PHYSICS

SIGURD SANDVOLL SUNDBERG
GITHUB.COM/SIGURDSUNDBERG

ABSTRACT. Abstract write last.

CONTENTS

1. Introduction

The use of differential equation has a sentral role in every branch of science. Linear second-order differential equations cover many of the important differential equations. Many of which can be manipulated to be solved as a linear algebra problem, for which we can develop algorithms to solve to problems numerically. There are limitations when it comes to numerically solutions. If the number of floating point operations(FLOPs) becomes too big the calculations can get exceedingly slow. One can also risk numerical inaccuracies simply due to a high amount of FLOPs. This puts the emphasis on developing algorithms which are stable, accurate and quick. In this project we will take a look at solving the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. Excplicitly we want to solve the following equation

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

We will device an algorithm for a tridiagonal matrix, both the general case and a specialized case. Including, we will take a look at the more general method of solving matrix equations using LU-decomposition, comparing run times and space between the different algorithms. In addition to looking at relative error and finding the mesh-grid which gives the best approximation. Lastly, the findings will be discussed.

2. Theory

2.1. Poisson equation.

A classic exmaple of linear second-order differential equation is the Poisson's equation from electromagnetism. The electrostatic potential Φ is generated by localized charge distribution $\rho(\mathbf{r}$. The equation read

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

If we have spherical symmetric Φ and $\rho(\mathbf{r}$ the equations simplifies to one-dimensional equation in r, namely

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

By substituting $\Phi(r) = \phi(r)/r$ we get

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

This is a linear second-order differential equation on the form in equation 1 with $\phi \to u$, $r \to x$ and $f(x) = 4\pi r\rho(r)$. We are left with

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

In our study the source term will be $f(x) = 100e^{-10x}$ and and the analytical solution is given by $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$, this can be verified easily by taking the second derivative of $u(x)$.

2.2. Approximating the second derivative.

To solve one-dimensional Poisson equiation numerically we will need to discretize the equation. Instead of having continous functions, we get

$$\begin{aligned} x &\to x_i \in [x_0, x_1, \ldots, x_i, \ldots, x_{n+1}] \\ u(x) &\to u(x_i) = u_i \in [u_0, u_1, \ldots, u_i, \ldots, u_{n+1}] \\ f(x) &\to f(x_i) = f_i \in [f_0, f_1, \ldots, f_i, \ldots, f_{n+1}], \end{aligned} \tag{2}$$

using grid-points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$ since we have an open interval. The step length h is defined as $h = 1/(n+1)$. We then have the boundary conditions $x_0 = v_{n+1} = 0$. The approximation for the second derivative we need from 1, can be found through Taylor expansion of $u(x \pm h)$ around x.

$$u(x \pm h) = u(x) \pm hu'(x) + \frac{h^2 u''(x)}{2!} \pm \frac{h^3 u'''(x)}{3!} + \mathcal{O}(h^4) \tag{3}$$

If we look at the two equation we get from $u(x+h)$ and $u(x-h)$, we can see that the first and third derivatives cancel eachother out when we add the two equations together. If we solve the resulting equation for $u''(x)$ we get

$$u''(x) = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2). \tag{4}$$

Using our discretization of the continous functions given by equation 2, we have the following equation

$$-u''(x) \simeq f_i = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} + \mathcal{O}(h^2). \tag{5}$$

for $i = 1, \cdots, n$. If we define $f_i^* = h^2 f_i$, equation 5 becomes $-u_{i+1} - u_{i-1} + 2u_i = f_i^*$. If we try inserting values of $i = 1, 2, 3, 4, \ldots, n-1n$ we get the following

$$\begin{aligned}
-u_2 - u_0 + 2u_1 &= f_1^* \quad i = 1\\
-u_3 - u_1 + 2u_2 &= f_2^* \quad i = 2\\
-u_4 - u_2 + 2u_3 &= f_3^* \quad i = 3\\
-u_5 - u_3 + 2u_4 &= f_4^* \quad i = 4\\
&\vdots\\
-u_n - u_{n-2} + 2u_{n-1} &= f_{n-1}^* \quad i = n-1\\
-u_{n+1} - u_{n-1} + 2u_n &= f_n^* \quad i = n
\end{aligned} \tag{6}$$

With the Dirichlet boundary conditions $u(0) = u(n+1) = 0$ we see resembles a matrix with 2 along the diagonal and -1 directly above and below the leading diagonal. We see that we can solve as a linear algebra problem, where we want to find $\vec{x}$, on the following form.

$$\mathbf{A}\vec{x} = f^*,$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we found in 6.

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \ddots & \ddots & \ddots & \dots \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{bmatrix} \tag{7}$$

3. Algorithms

For all programs, benchmark calulations and plots, see:
github.com/SigurdSundberg/FYS3150

3.1. Tridiagonal Matrix Algorithm.

Looking at a general case of solving a tridiagonal matrix on the form $\mathbf{A}\vec{x} = f^*$ we have

$$\begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & 0 & \dots & \dots \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} \\ \dots & \dots & \dots & \dots & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ \dots \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} f_1^* \\ f_2^* \\ \dots \\ \dots \\ \dots \\ f_n^* \end{bmatrix} \tag{8}$$

Let's take general case for a 4×4 tridiagonal matrix and use Gaussian elimination to see if we can spot a pattern. Our system can then be written

$$\mathbf{A}\vec{x} = f^*$$

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1^* \\ f_2^* \\ f_3^* \\ f_4^* \end{bmatrix} \tag{9}$$

The set of equations we will use Gaussian elimination on is therefore

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1^* \\ a_1 & b_2 & c_2 & 0 & f_2^* \\ 0 & a_2 & b_3 & c_3 & f_3^* \\ 0 & 0 & a_3 & b_4 & f_4^* \end{array}\right] \tag{10}$$

First we will use forward substitution to remove all the elements along the diagonal below the leading diagonal. We start with $\mathrm{II} - \mathrm{I}(a_1/b_1)$ to remove a_1 from II, where I refers to row one, II refers to row two and so on. We now how

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & b_2 - c_1a_1/b_1 & c_2 & 0 & f_2^* - f_1^*a_1/b_1 \\ 0 & a_2 & b_3 & c_3 & f_3^* \\ 0 & 0 & a_3 & b_4 & f_4^* \end{array}\right] \tag{11}$$

We will define two new variables $\tilde{b}_2 = b_2 - c_1a_1/b_1$ and $\tilde{f}_2 = f_2^* - f_1^*a_1/b_1$ to simplify the matrix and continue the forward substitution with $\mathrm{III} - \mathrm{II}(a_2/\tilde{b}_2)$.

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & b_3 - c_2a_2/\tilde{b}_2 & c_3 & f_3^* - \tilde{f}_2a_2/\tilde{b}_2 \\ 0 & 0 & a_3 & b_4 & f_4^* \end{array}\right] \tag{12}$$

Using the same definition as above as prior $\tilde{b}_i = b_i - c_{i-1}a_{i-1}/b_{i-1}$ and $\tilde{f}_i = f_i - f_{i-1}a_{i-1}/b_{i-1}$ we are left with

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1^* \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & a_3 & b_4 & f_4^* \end{array}\right] \tag{13}$$

Repeating this one more time we can eliminate a_3. We see however that we see that a pattern emerges for the forward substitution namely

$$\begin{aligned} \tilde{b}_i &= b_i - \frac{c_{i-1}a_{i-1}}{\tilde{b}_{i-1}} \\ \tilde{f}_i &= f_i^* - \frac{\tilde{f}_{i-1}a_{i-1}}{\tilde{b}_{i-1}}. \end{aligned} \tag{14}$$

If we define $\tilde{b}_1 = b_1$ and $\tilde{f}_1 = f_1$ and get the follwing matrix

$$\left[\begin{array}{cccc|c} \tilde{b}_1 & c_1 & 0 & 0 & \tilde{f}_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & 0 & \tilde{b}_4 & \tilde{f}_4 \end{array}\right] \tag{15}$$

We can now solve for $\vec{x}$ by doing a backwards substitution. We find x_4 by doing $\tilde{f}_4/\tilde{b}_4$, giving us

$$\left[\begin{array}{cccc|c} \tilde{b}_1 & c_1 & 0 & 0 & \tilde{f}_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & \tilde{f}_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right] \tag{16}$$

Doing a backwards substitution on matrix 16 by taking III $-$ IV(c_3) and dividing III by $\tilde{b}_3$ to normalize the row. We then define $x_3 = (\tilde{f}_3 - x_4c_3)/\tilde{b}_3$, which gives us

$$\left[\begin{array}{cccc|c} \tilde{b}_1 & c_1 & 0 & 0 & \tilde{f}_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & \tilde{f}_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right] \tag{17}$$

Repeating the process on row II and using the same definition as above we get

$$\left[\begin{array}{cccc|c} \tilde{b}_1 & c_1 & 0 & 0 & \tilde{f}_1 \\ 0 & 1 & 0 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right] \tag{18}$$

we see that a pattern emerges, namely

$$x_i = \frac{\tilde{f}_i - x_{i+1}c_i}{\tilde{b}_i} \tag{19}$$

which we will call backwards substitution from now.

Assuming we have pre initialized vectors a, b, c, f^* and defining $\tilde{b}_1 = b_1$ and $\tilde{f}_1 = 1$, our general algorithm from respectivly 14 and 19 will read as follows

Algorithm 1: Forward substitution

```
initialization;
for $i$ = 2, 3, ..., $n$ do
    $quotient = a_{i-1}/tildeb_{i-1}$;
    $\tilde{b}_i = b_i - c_{i-1} \cdot quotient$;
    $\tilde{f}_i = f^*_i - \tilde{f}_{i-1} \cdot quotient$;
end
```

Algorithm 2: Backward substitution

```
$x_n = \tilde{f}_n/\tilde{b}_n$;
for $i$ = n-1, n-2, ..., 1 do
    $x_i = (\tilde{f}_i - x_{i+1}c_i)/\tilde{b}_i$;
end
```

We are also interested in the number of FLOPs for our general algorithm. From the algorithm 1 we see that for each iteration i of the loop we have 5 FLOPs and in algorithm 2 we have 3 FLOPs for each iteration i of the loop. Each loop does $n-1$ iterations, thus we have in total $(8(n-1)+1)$ FLOPs, the $+1$ comes from the initial calculation in algortihm 2. So we have in total $8n-7$ FLOPs, for large n we can ignore the cosntant terms and we are left with $8n$ FLOPs for the general algorithm. For very large n this will come out to be $\mathcal{O}(n)$, however for comparison purposes we will use $8n$ FLOPs for the general algorithm.

3.2. **Specialized algorithm.** In our study we have matrix on the form seen in matrix 7 which as 2 along the leading diagonal and -1 along two other diagonals. Since we have a symmetrical matrix we make it more efficient at solving the problems. If we look at our algortihms 14 and 19 we can insert $b_i = 2$ and $a_i = c_i = -1$ for all i and we get the following expressions

$$\tilde{b}_i = 2 - \frac{1}{\tilde{b}_{i-1}} \tag{20}$$

$$\tilde{f}_i = f_i^* + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{21}$$

$$x_i = \frac{\tilde{f}_i + x_{i+1}}{\tilde{b}_i} \tag{22}$$

If we take a look at equation 20, we can show that the expressions can be rewritten on the form

$$\tilde{b}_i \equiv d_i = \frac{i+1}{i}. \tag{23}$$

We see this from inserting $i = 2, 3, 4, \ldots$. If we take a look at it we get

$$\begin{aligned}
d_1 &= 2 \\
d_2 &= 2 - \frac{1}{2} = \frac{3}{2} \\
d_3 &= 2 - \frac{1}{\frac{3}{2}} = \frac{4}{3} \\
d_4 &= 2 - \frac{1}{\frac{4}{3}} = \frac{5}{4} \\
&\vdots \\
d_i &= \frac{i+1}{i}
\end{aligned}$$

this can be shown to be true through, for example induction. This can be found before our algorithm reducing the number of FLOPs needed in our loops. Psuedocode for our specialized algorithm will be as follows

Algorithm 3: Specialized algortihm

```
initialization;
for $i$ = 2, 3,..., $n$ do
    $\tilde{f}_i = f_i^* + \tilde{f}_{i-1}/d_{i-1}$;
end
$x_n = \tilde{f}_n/d_n$;
for $i$ = n-1, n-2, ..., 1 do
    $x_i = (\tilde{f}_i + x_{i+1})/d_i$
end
```

If we take a look at the number of FLOPs in our specialized algorithm we see that we have 2 FLOPs for each of the loops, running $n-1$ times. Thus our specialized algorithm has in total $4(n-1)+1$ FLOPs $= (4n-3)$ FLOPs. For large n we have $4n$ FLOPs. If we compared this to the generalized algorithm, we see that it is a factor 2.

3.3. **LU Decomposition.** The general method for solving dense non-singular matrix is using LU decomposition. The method involves taking a matrix **A** and rewriting it into the product of a lower triangular matrix **L** and an upper triangular matrix

U.

$$\mathbf{A} = \mathbf{LU}$$

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

This gives us a new way of solving $\mathbf{A}\vec{x} = \vec{b}$, where the problem can be solved in two steps

$$\mathbf{LU}\vec{x} = \vec{b} \rightarrow \mathbf{L}\vec{w} = \vec{b}, \quad \mathbf{U}\vec{x} = \vec{w} \tag{24}$$

We can solve $\mathbf{L}\vec{w} = \vec{b}$ by forward substitution and $\mathbf{U}\vec{x} = \vec{w}$ by backward substitution.

Here we will not develop or implementing an algorithm for LU decomposition. We will use the high level library Armadillo and implement the course functions which can be found in the code folder on github, named lib.h. We are interested in the difference in runtime between the general algorithm, specialized algorithm and LU decomposition. Also included is the Armadillo function *solve* which is adaptable and choose the quickest possible solution to the linear system. We remember that for the general algorithm the number of FLOPs needed goes like $(8n)$FLOPs and for the specialized we were able to reduce it to $(4n)$FLOPs. For LU decomposition, as we are not working with vectors, but entire matricies. The number of FLOPs needed to LU decompose a matrix is $(2/3n^3)$FLOPs[1]. We would expect that using LU decomposition to solve the problem would result in a solver execution time going n^2.

4. Results

4.1. Relative error. Here we will take a look at the relative error between the algorithms implemented.

Table 1. Table covering runtimes from semi-cold starts for all four algorithms used. Thoose being General, Specialized, Armadillos *solve* and the library functions from lib.hpp. Data for Armadillo *solve* and lib.hpp are not included for $n > 4$ as it is limited by memory and could not be run in this test. All times are listed in seconds.

10^n	General	Specialized	*solve*	lib.hpp
1	0.00000024	0.00000021	0.00303977	randomdata
2	0.00000164	0.00000140	0.00008639	randomdata
3	0.00001927	0.00001484	0.00644989	randomdata
4	0.00021870	0.00014394	0.58261636	randomdata
5	0.00223254	0.00147353	n/a	n/a
6	0.02110298	0.01508435	n/a	n/a

4.2. CPU time.

5. Discussion

Discussion of the report.

6. Conclusion

Conclusion of the report.

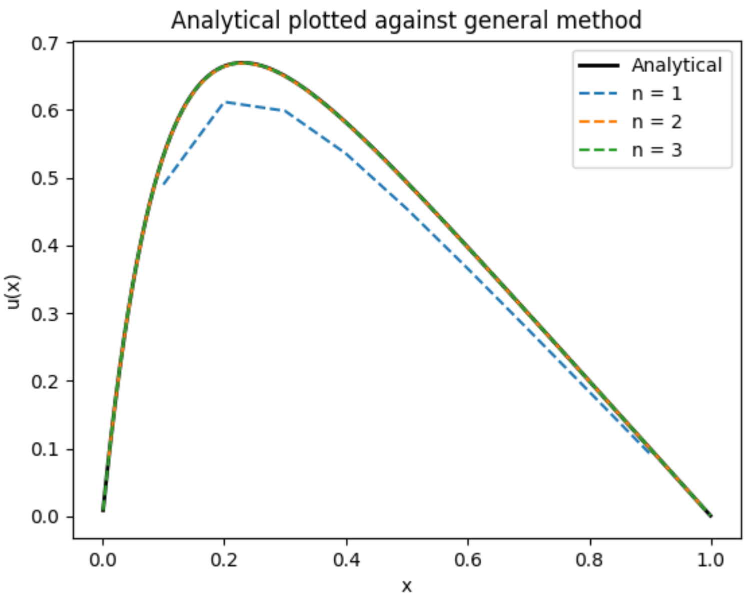

FIGURE 1. Plot of the specialized algorithm against the analytical solution. The plot shows runs for of the specialized algorithm with $n = 1, 2, 3$. End points are skipped in the plot for numerical solutions, as they do not affect the calculations and are set as $u(0) = u(n) = 0$.

REFERENCES

[1] Morten Hjorth-Jensen. Computational physics, lecture notes fall 2015. *Department of Physics, University of Oslo*, page 173, 2015.

[2] Sigurd Sandvoll Sundberg. Fys3150. `https://github.com/SigurdSundberg/FYS3150/tree/master/project1`, 2020.