Numerical Methods for Solving the One Dimensional Poisson Equation

Håkon Fossheim

Contents

1 Introduction

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

$$\frac{d^2y}{dx^2} + k(x)^2 y = f(x) \tag{1}$$

The Poisson equation is a linear, inhomogeneous second-order differential equation. It describes the potential field caused by a given charge- or mass density. When applied in the context of electromagnetism, it takes the form

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

Where ϕ is the electrostatic field, and ρ is the charge density. It is also used in the context of Newtonian gravity

$$\nabla^2 \Phi = 4\pi G\rho(\vec{r}) \tag{3}$$

Where ϕ is the gravitational potential and ρ the mass density. This project is focused on solving the one-dimensional electrostatic case numerically. The equation is solved using two different algorithms, namely the tridiagonal matrix algorithm(TDMA) and LU decomposition. A comparison of the efficiency of the algorithms is made.

2 Theory

2.1 The Poisson Equation

When ϕ and ρ are spherically symmetric, equation (2) is reduced to one dimension.

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

Substituting $\Phi(r) = \frac{\phi(r)}{r}$ gives

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r) \tag{5}$$

Rewriting using $\phi = u$ and $r = x$ yields

$$-\frac{d^2u}{dx^2} = g(x) \tag{6}$$

Where $g(x) = 4\pi x\rho(x)$. Equation (6) is to be solved numerically with Dirichlet boundary conditions.

$$-\frac{d^2u}{dx^2} = g(x), \quad x \in (0,1), \quad u(0) = u(1) = 0 \tag{7}$$

There are $n+2$ grid points: $x_i \in [x_0, x_1, ..., x_n, x_{n+1}]$, with corresponding numerical solutions v_i. The step size is $h = \frac{x_{n+1}-x_0}{n}$, giving $x_i = ih$.
The numerical approximation is found using the three-point centered-difference formula

$$\frac{-v_{i+1} - v_{i-1} + 2v_i}{h^2} = g(x) \tag{8}$$

Setting $f(x) = h^2 g(x)$ gives

$$-v_{i+1} - v_{i-1} + 2v_i = f(x) \tag{9}$$

This is a linear set of $n+2$ equations, and can therefore be written as a matrix equation

$$\mathbf{A}\vec{v} = \vec{f} \tag{10}$$

$$\begin{bmatrix} 2 & -1 & 0 & ... & ... & 0 \\ -1 & 2 & -1 & 0 & ... & ... \\ 0 & -1 & 2 & -1 & 0 & ... \\ & ... & ... & ... & ... & ... \\ 0 & ... & & -1 & 2 & -1 \\ 0 & ... & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ ... \\ v_n \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ ... \\ f_n \\ f_{n+1} \end{bmatrix} \tag{11}$$

In this project $f(x)$ is set to be $f(x) = 100e^{-10x}$. The analytical solution to equation 10 is then $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$.

2.2 Gaussian Elimination

Equation (9) can be generalized to

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = f(x) \tag{12}$$

The endpoints are known. They can therefore be excluded when setting up the algorithm. The 4x4 matrix equation corresponding to equation (12) is

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & a_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \tag{13}$$

By performing forward Gaussian elimination on this set of equations, the lower diagonal elements are eliminated.

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ a_1 & b_2 & c_2 & 0 & f_2 \\ 0 & a_2 & b_3 & c_3 & f_3 \\ 0 & 0 & a_3 & b_4 & f_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ 0 & b_2 - \frac{a_1}{b_1}c_1 & c_2 & 0 & f_2 - \frac{a_1}{b_1}f_1 \\ 0 & a_2 & b_3 & c_3 & f_3 \\ 0 & 0 & a_3 & b_4 & f_4 \end{array}\right] \tag{14}$$

The first row was multiplied by $\frac{a_1}{b_1}$ and subtracted from the second row, eliminating a_1. Because of this, the values of b_2 and f_2 are altered to $\tilde{b_2} = b_2 - \frac{a_1}{b_1}c_1$ and $\tilde{f_2} - \frac{a_1}{b_1}f_1$. Performing another forward substitution yields

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ 0 & \tilde{b_2} & c_2 & 0 & \tilde{f_2} \\ 0 & a_2 & b_3 & c_3 & f_3 \\ 0 & 0 & a_3 & b_4 & f_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ 0 & \tilde{b_2} & c_2 & 0 & \tilde{f_2} \\ 0 & 0 & b_3 - \frac{a_2}{\tilde{b_2}}c_2 & c_3 & f_3 - \frac{a_2}{\tilde{b_2}}\tilde{f_2} \\ 0 & 0 & a_3 & b_4 & f_4 \end{array}\right] \tag{15}$$

Let $\tilde{b_3} = b_3 - \frac{a_2}{\tilde{b_2}}c_2$ and $\tilde{f_3} = f_3 - \frac{a_2}{\tilde{b_2}}\tilde{f_2}$.

$$\left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ 0 & \tilde{b_2} & c_2 & 0 & \tilde{f_2} \\ 0 & 0 & \tilde{b_3} & c_3 & \tilde{f_3} \\ 0 & 0 & a_3 & b_4 & f_4 \end{array}\right] \sim \left[\begin{array}{cccc|c} b_1 & c_1 & 0 & 0 & f_1 \\ 0 & \tilde{b_2} & c_2 & 0 & \tilde{f_2} \\ 0 & 0 & \tilde{b_3} & c_3 & \tilde{f_3} \\ 0 & 0 & 0 & b_4 - \frac{a_3}{\tilde{b_3}}c_3 & f_4 - \frac{a_3}{\tilde{b_3}}\tilde{f_3} \end{array}\right] \tag{16}$$

By setting $\tilde{b_1} = b_1$ and $\tilde{f_1} = f_1$, the following general formulae hold:

$$\tilde{b_i} = b_i - \frac{a_{i-1}}{\tilde{b}_{i-1}}c_{i-1} \tag{17}$$

$$\tilde{f_i} = f_i - \frac{a_{i-1}}{\tilde{b}_{i-1}}\tilde{f}_{i-1} \tag{18}$$

```
        //Forwards substitution, general case
for (int i = 1; i < n; i++) {
        double ratio = a[i - 1] / b[i - 1]; //1 FLOP
        b[i] = b[i] - (ratio)*c[i-1]; //2FLOPS
        f[i] = f[i] - (ratio)*f[i-1]; //2FLOPS
        }
```

In the special case given by equation (9), these general formulae are reduced to

$$\tilde{b_i} = b_i - \frac{1}{\tilde{b}_{i-1}} \tag{19}$$

$$\tilde{f_i} = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}} \tag{20}$$

```
        //Forwards substitution, special case
for (int i = 1; i < n; i++){
    b[i] = b[i] - 1/b[i-1];    //2FLOPS
    f[i] = f[i] + (f[i-1] / b[i-1]);   //2FLOPS
        }
```

The matrix A has now been reduced to an upper diagonal matrix

$$\begin{bmatrix} \tilde{b_1} & c_1 & 0 & 0 \\ 0 & \tilde{b_2} & c_2 & 0 \\ 0 & 0 & \tilde{b_3} & c_3 \\ 0 & 0 & 0 & \tilde{b_4} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \end{bmatrix} \tag{21}$$

Where

$$\begin{aligned} v_1 &= (\tilde{f}_1 - c_1 v_2)/\tilde{b_1} \\ v_2 &= (\tilde{f}_2 - c_2 v_3)/\tilde{b_2} \\ v_3 &= (\tilde{f}_3 - c_3 v_4)/\tilde{b_3} \\ v_4 &= \tilde{f}_4/\tilde{b_4} \end{aligned}$$

The general formula is

$$v_i = \left(\tilde{f}_i - c_i v_{i+1}\right)/\tilde{b_i} \tag{22}$$

This also holds for $i = n$, because $v_{n+1} = 0$, giving $v_n = \tilde{f}_n/\tilde{b_n}$. By calculating v_n first, all the elements of $\vec{v}$ can be found by inserting v_{i+1} into the expression for v_i.

```
    // Backwards substitution, general case
v[n-1] = f[n-1]/b[n-1]; //1 FLOP
for (int i = n-2 ; i >= 0; i--) {
        v[i] = (f[i] - c[i]*v[i+1])/b[i]; //3 FLOPS
        }
```

The formula for the special case given by equation (9) is

$$v_i = \left(\tilde{f}_i + v_{i+1}\right)/\tilde{d}_i \tag{23}$$

```
    //Backwards substitution, special case
v[n-1] = f[n-1]/b[n-1]; //1FLOP
for (int i = n-2; i >= 0; i--) {
         v[i] = (f[i] + v[i+1])/b[i]; //3FLOPS
        }
```

From the code shown above, the total numbers of FLOPS for the general and special cases are

- General: $[5(n-2)] + [3(n-2)+1] = 8(n-2)+1 = \underline{8n-15}$
- Special: $[4(n-2)] + [3(n-2)+1] = 7(n-2)+1 = \underline{7n-13}$

2.3 LU Decomposition

Another way of solving equation (11) is using LU decomposition. This involves decomposing **A** into a lower triangular matrix, **L**, and an upper triangular matrix, **U**.
By adding a multiple of one row to a row below it, **A** can be reduced to the upper triangular matrix **U**. Each row operation corresponds to multiplying **A** on the left side with a corresponding elementary matrix.

$$\mathbf{E_k}...\mathbf{E_2}\mathbf{E_1}\mathbf{A} = \mathbf{U} \tag{24}$$

This can also be written as

$$\mathbf{A} = \mathbf{E_k^{-1}}...\mathbf{E_2^{-1}}\mathbf{E_1^{-1}}\mathbf{U} \tag{25}$$
$$\mathbf{A} = \mathbf{LU} \tag{26}$$

Since each row operation only involved adding a multiple of one row to a row below it, the elementary matrices are all lower triangular with ones on the diagonal. The corresponding inverse elementary matrices will also have this form. **L** will therefore take the form seen in the following equation, which is a 4x4 example.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \tag{27}$$

Inserting equation (26) into equation (10) gives

$$(\mathbf{LU})\,\vec{v} = \vec{f} \tag{28}$$
$$\mathbf{L}\,(\mathbf{U}\vec{v}) = \vec{f} \tag{29}$$

Now let

$$\mathbf{U}\vec{v} = \vec{w} \tag{30}$$

Inserting $\vec{w}$ into equation (29) gives

$$\mathbf{L}\vec{w} = \vec{f} \tag{31}$$

$\vec{v}$ is found by first solving equation (31) for $\vec{w}$ and inserting $\vec{w}$ into equation (30) and solving it for $\vec{v}$.

```
lu(L, U, A);
vec w = solve(L, f);
v = solve(U, w);
```

The Armadillo library was used to LU decompose **A** and subsequently solve $\vec{v}$

2.4 Implementation

All the functions needed for solving this project are implemented in "problems.cpp".
The function "problem1b" solves the general case. For n gridpoints, the user may specify $\vec{a}$, $\vec{b}$ and $\vec{c}$ by replacing the first n rows of the file "matrixdata.txt" with the desired values of a, b and c. Alternatively, the user may specify these values through the terminal by uncommenting the appropriate code. If "matrixdata.txt" is not found, the values will automatically be set to that of the special case.

The Chrono library was used to time the algorithms, and the functions "problem1b", "problem1c" and "problem1e" return the run-time for the general TDMA, specialized TDMA and LU decomposition respectively. Each of them take the number of gridpoints, n, as arguments. In addition to this, "problem1b" and "problem1c" take a bool as argument to determine whether to write the results to file or not.

3 Results

The numerical result presented in figures 1-3 were produced using the specialized TDMA in the function "problem1c".
Figure 1 shows the analytical and numerical solution of equation (7) for 10 gridpoints.

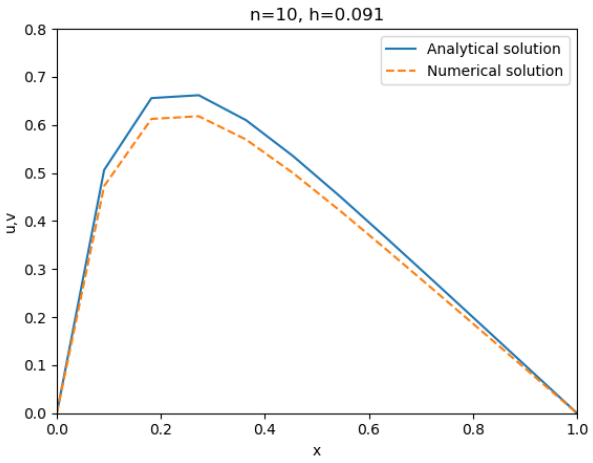

Figure 1: Analytical solution vs Gaussian elimination with 10 gridpoints

As seen, the numerical solution in figure 1 differs substantially from the analytical one. This is not suprising, as the error term in the three-point

centered-difference formula goes as $\mathcal{O}(h^2)$, which will be quite large for this step size.

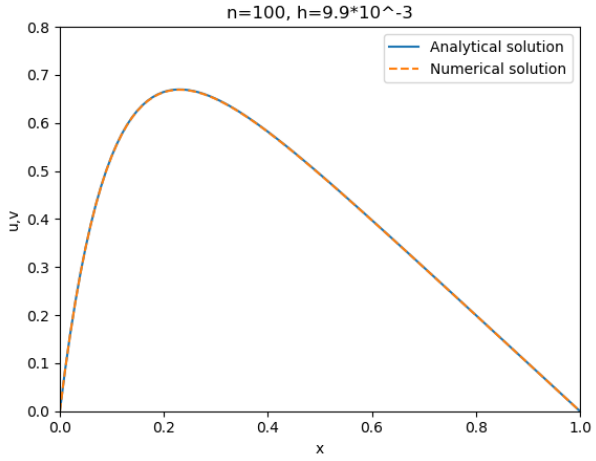

Figure 2: Analytical solution vs Gaussian elimination with 100 grid points

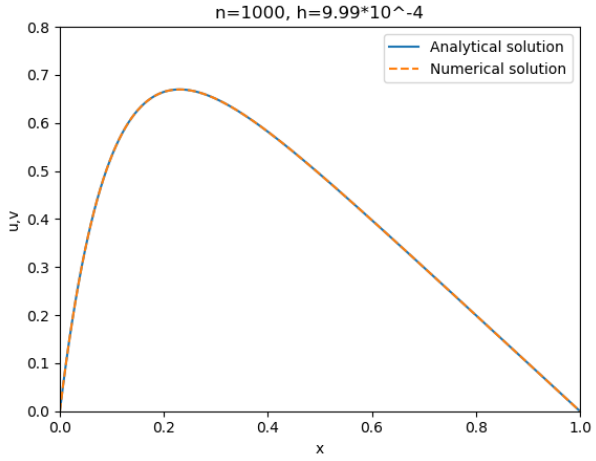

Figure 3: Analytical solution vs Gaussian elimination with 1000 grid points

When the number of grid points is increased to 100 and above, figures 1 and 2 seem indicate that the numerical solution overlap the analytical one. As table 1 and figure 4 show, this is not the case. The relative error was calculated for each grid point using

$$\epsilon = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{32}$$

and finding the maximum of these values for the given step size.

Relative error, Gaussian		
n	$\log_{10}(h)$	$max\left(\log_{10}\left(\left\|\frac{v_i-u_i}{u_i}\right\|\right)\right)$
10	-1	-1.1797
10^2	-2	-3.08804
10^3	-3	-5.08005
10^4	-4	-7.07936
10^5	-5	-9.0049
10^6	-6	-6.77137
10^7	-7	-13.007
$10^{7.2}$	7.2	-7.25184

Table 1:

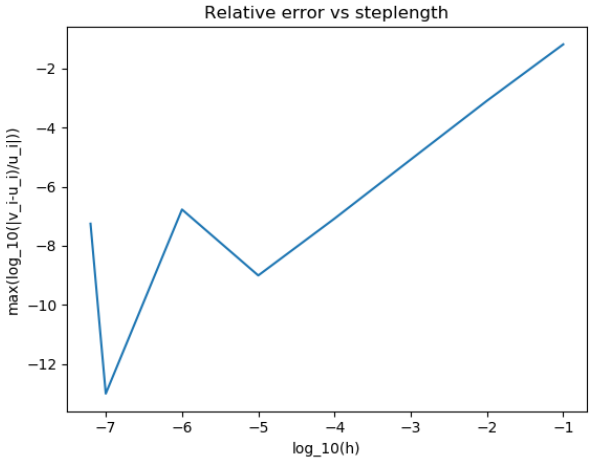

Figure 4: relative error vs step size, plotted on a log-log scale

Figure 4 shows that the relative error has a slope of two for step sizes larger than 10^{-5}. This is in accordance with the mathematical error, which goes as $\mathcal{O}(h^2)$.

As the step size gets smaller than 10^{-5} the computer will start accumulating round-off errors. This causes the error to behave unpredictably and in this case oscillate. The optimal step size is therefore 10^{-5}, since it is the smallest step size in the linear regime.

A for-loop was used to find the CPU times presented in table 2 and 3.

```
double sum = 0;
for (int i = 0; i < 10; i++) {
        sum+=problem1b(n,0);
}
double time_avrg = sum / 10;
cout << "The␣average␣CPU␣time␣was:␣" <<
   time_avrg*pow(10,6) << "␣microseconds."
   << endl;
```

The function "problem1b" returns the run-time of the algorithm. The for-loop then calculates the time average of 10 runs. This is done because the CPU time varies depending on what the machine is doing. A breakpoint inside the for loop allows for sampling at more spaced out time intervals.

CPU time, generalized vs specialized TDMA		
n	General	Specialized
10	0.474 µs	1.24 µs
10^2	2.4431 µs	3.0999 µs
10^3	25.1624 µs	24.1779 µs
10^4	0.22431 ms	0.21558 ms
10^5	2.24164 ms	1.92405 ms
10^6	20.9408 ms	19.2493 ms

Table 2:

Table 2 shows the CPU time for the generalized vs specialized TDMA. The specialized algorithm is slightly faster for large n. Both algorithms scale linearly with n, becoming about 10 times slower when n increases with a factor of 10. This is in agreement with theory.

CPU time, specialized TDMA vs LU decomposition		
n	Gaussian	LU decomposition
10	1.24 µs	136.898 µs
10^2	3.0999 µs	476.434 µs
10^3	24.1779 µs	22686.5 µs
10^4	0.21558 ms	2.43512 s

Table 3:

Table 3 shows that the Gaussian algorithm is far more efficient, especially for larger n. Whilst the run-time for the Gaussian algorithm scales linearly with n, the run-time for LU decomposition scales as n^2 according to the table.

4 Discussion

The round-off errors encountered for small step sizes are due to the lack of precision in floating point numbers. The computer used for this project uses a 64-bit operating system. Double-precision floating point numbers are therefore used, giving 52 bits to the mantissa and 11 bits to the exponent.
Fractions of 10 cannot be represented fully in binary as they are infinitely recurring numbers.

$$0.1_{10} = 0.00011001100110011..._2 \tag{33}$$

The number therefore has to be approximated. If we assume a constant absolute round off error,

$$|h_i - \tilde{h}_i| = \epsilon_a \tag{34}$$

then the relative error

$$\frac{|h_i - \tilde{h}_i|}{|h_i|} = \epsilon_r \tag{35}$$

will be very small for large values of h, and the mathematical error will dominate. As h gets smaller, the relative error will increase, and at some point be of significance. This explains the behavior seen in figure 4.

The run-time for LU decomposition goes as n^2. This is odd, because the number of FLOPS goes as n^3. This disagreement with theory might come from the fact that the computer used has 4 CPU cores, and so can perform parallel computing.