FYS3150 - Project 1

Vebjørn Gilberg and Trond Wiggo Johansen

September 2016

Abstract

We are going to solve the one-dimensional Poisson equation in two ways starting off with a simplified Gaussian elimination method and then by LU decomposition. The approximation of the second derivative is done by a three-point formula and we solve the problem with a discretized variable using a set of linear equations. This type of numerical solution to a DE relies heavily on the method used. With the LU decomposition we will see that the time required to run the program and get the solution increases extremely fast as n gets bigger, while the tridiagonal method performs a lot better.

Git repository: `https://github.com/VebjornG/FYS3150/tree/master/Prosjekt_1`

1 Introduction

In 1813, Siméon Denis Poisson studied the interior of attracting masses, in particular he studied the potential in the interior of the masses. His work later had an impact on electrostatics[1] and one of his results was the Poissons equation which we will solve.

Given a source function $f(x)$ we will solve the one-dimensional Poisson equation for a discretized set of variables $x \in (0, 1)$ with Dirichlet boundary conditions. Our first mission is to set a basis for the problem in terms of theory. If we play our cards right, we can represent the discretization of the second derivative by a set of linear equations on the form $\boldsymbol{Av} = \tilde{\boldsymbol{f}}$ and thus solve the equation numerically by Gaussian elimination.

The neat thing about our problem is that we will find ourselves performing Gaussian elimination on a tridiagonal matrix, which simplifies the task greatly. We start by choosing $n = 10$ grid points which corresponds to a 10×10 matrix, and then go on to $n = 100$ and $n = 1000$ which we implement in a C++ program.

The relative error in the data set can be computed by $\epsilon_i = log_{10}\left(|\frac{v_i - u_i}{u_i}|\right)$, for which we will find the maximum value in each step. Comparing this to the grid point size, n, for the different values will give us an idea of how well this method works. At the end we will compare this whole algorithm to the LU decomposition for all of the cases using library functions.

[1] *"Poisson has too much talent to apply it to the work of others. To use it to discover what is already known is to waste it ..."*

- Joseph Fourier after Poisson had a particularly productive period as he published works on a variety of topics along with electricity and magnetism.

2 Theory

For a three dimensional charge distribution the Poisson equation reads $\nabla^2\Phi = -4\pi\rho(\boldsymbol{r})$. What this equation tells us is that wherever there is charge around the potential accelerates, it grows ever steeper. Assuming that we're dealing with spherically symmetric potentials we can simplify the equation to

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

If we yet again simplify the expression by letting $\phi \to u$ and $r \to x$ we're left with the final differential equation that we will solve

$$-\frac{d^2u(x)}{dx^2} = f(x) \quad \text{with Dirichlet boundary conditions:} \quad u(0) = u(1) = 0$$

We will define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. Our step length is $h = 1/(n+1)$ and the boundary conditions will be $v_0 = v_{n+1} = 0$. We will assume that the source term is $f(x) = 100e^{-10x}$ and keep the same interval and boundary conditions. The differential equation has a solution given by $u(x) = 1-(1-e^{-10})x-e^{-10x}$. We can approximate the second derivative of u with the three point formula

$$-\frac{d^2u}{dx^2} \approx -\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i \qquad \text{for} \qquad i = 1,\dots,n$$

where $f_i = f(x_i)$. This equation is possible to rewrite as a set of linear equations of the form

$$\boldsymbol{Av} = \tilde{\boldsymbol{f}}$$

where $\mathbf{A}$ is a $n \times n$ tridiagonal matrix, $\mathbf{v}$ and $\tilde{\boldsymbol{f}}$ is a $n \times 1$ matrix.

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \cdots \\ 0 & -1 & 2 & -1 & 0 & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix}, \quad \mathbf{v} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{i-1} \\ v_i \\ v_{i+1} \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix} \quad \text{and} \quad \tilde{\mathbf{f}} = h^2 \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{i-1} \\ f_i \\ f_{i+1} \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix} = h^2\mathbf{f}$$

By rewriting the equation in general terms we get

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \equiv \tilde{f}_i$$

If we look at some values of i we might get an idea of how we can build our matrices

$$\begin{aligned} i = 1: &\qquad -v_0 + 2v_1 - v_2 = \tilde{f}_1 \\ i = 2: &\qquad -v_1 + 2v_2 - v_3 = \tilde{f}_2 \\ i = 3: &\qquad -v_2 + 2v_3 - v_4 = \tilde{f}_3 \\ &\vdots \end{aligned}$$

Discarding the endpoints $i = 0$ and $i = n + 1$ gives us an equation on the form $\mathbf{Av} = \tilde{\mathbf{f}}$, hence

$$\mathbf{Av} = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & \cdots \\ 0 & a_3 & b_3 & c_3 & 0 & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \vdots \\ \vdots \\ \tilde{f}_{n-1} \\ \tilde{f}_n \end{pmatrix} = \tilde{\mathbf{f}}$$

3 Algorithms and methods

3.1 Gaussian elimination

The problem at hand needs to be looked at carefully before we start to write the program. That is, we have to develop the algorithm starting off with the matrices as seen at the end of the theory section. The observant reader will notice that we can write he tridiagonal system on a more general form, namely

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i \qquad \text{for} \qquad i = 1, 2, \ldots, n$$

With a well established tradition of doing linear algebra in mind, we will do the Gaussian elimination with forward and backward substitution. Naturally, we start with forward substitution. Also, we can simplify the problem by setting $n = 4$. In the first step we multiply the first row by $\frac{a_1}{b_1}$ and subtract it from the second row, i.e.

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{pmatrix} \sim \begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b_2 - \frac{a_2 c_1}{b_1} & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{pmatrix}$$

What we want to do is to eliminate all the a_i's from the matrix, and so we have to repeat this process two more times. Finally we will find ourselves in the

situation where we have

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 \\ 0 & 0 & 0 & \tilde{b}_4 \end{pmatrix}$$

Where $\tilde{b}_2 = b_2 - \frac{a_2 c_1}{b_1}$ which leads to $\tilde{b}_i = b_i - \frac{a_i c_{i-1}}{\tilde{b}_{i-1}}$. Furthermore, we must define new functions $\tilde{f}_i$, so $\tilde{f}_2 = f_2 - \frac{a_2 f_1}{b_1}$,
$\tilde{f}_3 = f_3 - \frac{a_3 \tilde{f}_2}{\tilde{b}_2}$ and $\tilde{f}_4 = f_4 - \frac{a_4 \tilde{f}_3}{\tilde{b}_3}$ which leads to $\tilde{f}_i = f_i - \frac{a_i \tilde{f}_{i-1}}{\tilde{b}_{i-1}}$. When we put it together we get

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 \\ 0 & 0 & 0 & \tilde{b}_4 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \end{pmatrix}$$

When it comes to floating point operations we can observe that we need three of them for the $\tilde{b}_i$'s as well as 3 more when we include $\tilde{f}_i$, so now we're up to $6n$ FLOPS and we still have to perform the backward substitution. Now we have to work our way backwards, and if we look at the last matrix equation, in particular the last element $\tilde{b}_4$ we can see that $\tilde{b}_4 v_4 = \tilde{f}_4 \rightarrow v_4 = \frac{\tilde{f}_4}{\tilde{b}_4}$. When we repeat the process we begin to sense a pattern, i.e. The standard Gaussian elimination method requires $\frac{2}{3}n^3 + \mathcal{O}(n^2)$[1] which is a lot more than we need in our general solver.

$$\tilde{b}_3 v_3 + c_3 \underbrace{\frac{\tilde{f}_4}{\tilde{b}_4}}_{v_4} = \tilde{f}_3 \rightarrow v_4 = \left(\tilde{f}_3 - \tilde{b}_3 v_3\right) \Big/ c_3$$

$$\tilde{b}_2 v_2 + c_2 v_3 = \tilde{f}_2 \rightarrow v_3 = \left(\tilde{f}_2 - \tilde{b}_2 v_2\right) \Big/ c_2$$

Hence the algorithm for v_i can be written as

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

Now that we've done the backward substitution, we can see that we get a total of $9n$ FLOPS[2] due to the fact that we have 3 operations in every algorithm $\tilde{f}_i$, $\tilde{b}_i$ and $\tilde{v}_i$. In the special case of the Gaussian elimination where

$$b_i = 2 \qquad \text{and} \qquad a_i = c_i = -1$$

[2]Im aware that we can get $8n$ flops, but I'm not sure how.

we get the following iteration

$$\tilde{b}_2 = 2 - \frac{1}{2} = \frac{3}{2}$$
$$\tilde{b}_3 = 2 - \frac{1}{3/2} = 2 - \frac{2}{3} = \frac{4}{3}$$
$$\tilde{b}_4 = 2 - \frac{1}{4/3} = 2 - \frac{3}{4} = \frac{5}{4}$$

Evidently, the generalization of this is

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

One could argue that we don't need $\tilde{b}_i$, and that would be a well founded argument because we can remove it from the main algorithm and calculate the $\tilde{b}_i$'s beforehand. Now, if we include the forward and backward substitutions we get

$$\tilde{f}_i = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}}$$
$$\tilde{v}_i = \frac{\tilde{f}_i - v_{i-1}}{\tilde{b}_i}$$

So now we're up to $4n$ floating point operations. With this we can expect our special algorithm to run about twice as fast as the general.

3.2 The LU decomposition

The lower upper decomposition[3] factors a symmetric, invertible matrix $\boldsymbol{A}$ into an upper and lower traingular matrix, and as an example we'll use a $n \times n$ matrix. That is

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

Indeed, $\boldsymbol{A}$ can be written as $\boldsymbol{A} = \boldsymbol{LU}$ which means that we can reduce the problem to

$$\boldsymbol{Ly} = \tilde{\boldsymbol{b}} \qquad \text{and} \qquad \boldsymbol{Uv} = \boldsymbol{y}$$

[3]This decomposition was actually introduced by Alan Turing in 1948 in his paper 'Rounding-off errors in matrix processes'.

The next step involves going via $\boldsymbol{Ly} = \tilde{\boldsymbol{b}}$ using forward substitution to find $\boldsymbol{y}$, which we then can use to find $\boldsymbol{v}$ by backward substitution. We can visualize the problem in a more direct way by writing the equations out, hence

$$\begin{aligned} y_1 = \tilde{b}_1 \rightarrow l_{21}y_1 + y_2 = \tilde{b}_2 \rightarrow \quad \ldots \quad \rightarrow l_{n1}y_1 + l_{n2}y_2 + \ldots + y_n = \tilde{b}_n \\ u_{11}v_1 + u_{12}v_2 + \ldots + u_{1n}v_n = y_1 \rightarrow \quad \ldots \quad \rightarrow u_{(n-1)(n-1)}v_{n-1} + u_{(n-1)n}v_n = y_{n-1} \\ \implies u_{nn}v_n = y_n \end{aligned}$$

We could go through the process of making new arrays and whatnot in order to solve it. However, some kind people have made library functions that we can use which simplifies the process greatly. The downside is the number of FLOPS required by this method since the backward and forward substitution each requires n^2 floating point operations. The decomposition itself requires another $\frac{2}{3}n^3$ FLOPS so we're up to $\mathcal{O}_{LU} = \frac{2}{3}n^3 + 2n^2$ which is quite an increase!

4 C++ implementation

In our general solver (gen_solve) we have implemented the forward substitution as

```
for (int i = 2; i < n+1; i++) {
        b_tilde[i] = b[i] - (a[i-1]*c[i-1]/b_tilde[i-1]);
        f_tilde[i] = f[i] - (a[i-1]*f_tilde[i-1]/b_tilde[i-1]);
    }
```

and the backward substitution as

```
for (int i = n-1; i >= 1; i--) {
        v[i] = (f_tilde[i] - v[i+1]*c[i])/b_tilde[i];
    }
```

For our special solver (special_solve) the forward substitution is written as

```
for (int i = 2; i < n+1; i++) {
        f_tilde[i] = f[i] + (z[i]*f_tilde[i-1]);
    }
```

and the backward substitution as

```
for (int i = n-1; i >= 1; i--) {
        v_[i] = (f_tilde[i] + v_[i+1])/b_tilde[i];
    }
```

We computed the b_i's before running the forward and backward substitutions in our special solver. When comparing the CPU time for the general solver and the special solver we only clocked the time over the forward and backward substitutions.[4]

[4] We know that this is bad practice, that is, we should bake parts of the code into the report where they're needed. The thing is, it's quite late, so we'll do it on the next report!

5 Results

As we can see in *figure 1* the numerical solution fits the analytic solution better when N increases as we would expect. The C++ program iterates through the forward and backward substitution to ultimately produce v_i which is what we're plotting with N points along with the analytic solution $u(x)$ in a python script. Furthermore, we've included a timetracker to check the runtimes for the general and special case in our algorithm.

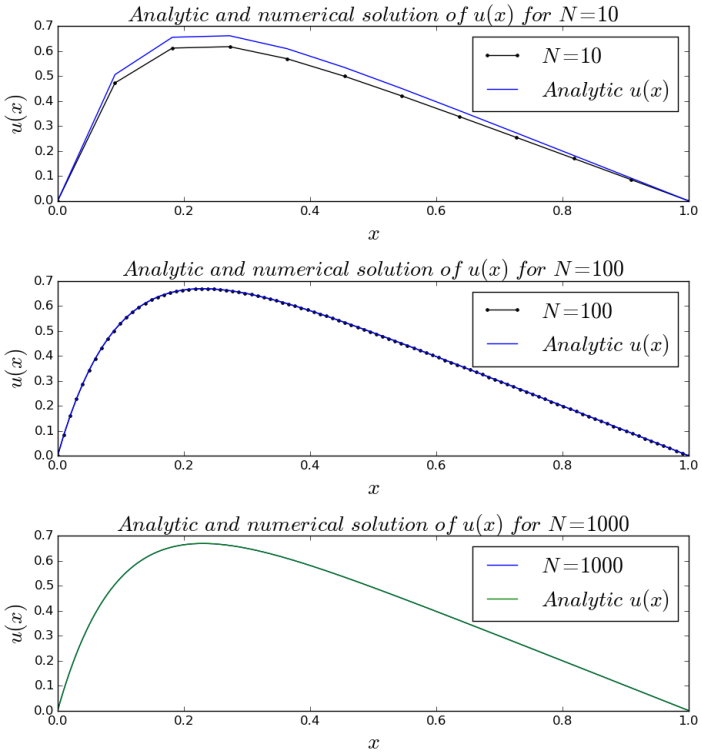

Figure 1

In the last section we said that we expected about twice as long runtime for the general case due to the FLOP-count being twice as large as the special case. We can see in the table below that our prediction wasn't completely correct, this may be due to the fact that there's not a correspondence in the number of flops calculated analytically and in the actual program.

n	Runtime - general	Runtime - special
10^1	0.00	0.00
10^2	$3.99 \cdot 10^{-6}$	$1.99 \cdot 10^{-6}$
10^3	$1.80 \cdot 10^{-5}$	$2.79 \cdot 10^{-5}$
10^4	$3.37 \cdot 10^{-4}$	$1.92 \cdot 10^{-4}$
10^5	$2.82 \cdot 10^{-3}$	$1.77 \cdot 10^{-3}$
10^6	0.024	0.0158

Table 1 Runtime for general solver and special solver.

The relative error of the method can be calculated by $\epsilon_i = log_{10}\left(|\frac{v_i - u_i}{u_i}|\right)$, and we were to find the maximum of this function for each n. When we increase n, we decrease the step length and so the maximum value of the error in each step should also decrease until round off errors take over such as when $n = 10^7$.

n	$\log_{10}(h)$	$\log_{10}(max(\epsilon_i))$
10^1	-1.0413927	-1.1796978
10^2	-2.0043214	-3.0880368
10^3	-3.0004341	-5.0800516
10^4	-4.0000434	-7.07927
10^5	-5.0000043	-9.0805589
10^6	-6.0000004	-13.370027
10^7	-7	-inf

Table 2 Matrix size n, log of step length, $\log_{10}(h)$ and log of maximal relative error, $\log_{10}(max(\epsilon_i))$ where $\epsilon_i = \log_{10}\left(|\ \frac{v_i - u_i}{u_i}\ |\right)$.

In general, when we want to see if something high or low etc., we need something to compare it to. In our case that's the runtime of our methods. We will compare it to the LU decomposition that we talked about earlier and see if the Gaussian elimination is faster or slower. Our prediction is that it's a lot faster than the LU decomposition. We can structure the times in a table as we did above, hence

n	Runtime - tridiagonal	Runtime - LU
10^1	0.00	0.000154
10^2	$3.99 \cdot 10^{-6}$	0.002392
10^3	$1.80 \cdot 10^{-5}$	0.094346
$2 \cdot 10^3$	$3.37 \cdot 10^{-4}$	0.472595
10^4	$2.82 \cdot 10^{-3}$	43.673426

Table 3 Runtime for general solver and the LU decomposition.

The runtimes for $n = 10^3$ were slightly faster than expected. If we try to run this at 10^5 the program will crash due to the fact that we get a matrix which is too large to handle. Needless to say, the LU decomposition is quite useless for $n \geq 10^4$, and it should be useless for $n \geq 2 \cdot 10^3$ as well.

6 Discussion

In method one we implemented the Gaussian elimination algorithm and checked the runtime and how the error evolved with increasing N. The error decreases with step size as one would expect. The worrying part about our relative error is that loss of numerical precision never takes over. That is, we should see an increase in the error. The reason we should see this increase is that when h gets smaller we subtract numbers that are almost equal in our algorithm which means that the numerical precision works up to a certain point. Our error however is infinitely small, which on a computer, is nonsensical, i.e. there's something wrong with our program.

Another thing that might confuse poor bachelor students is that the analytic solution is greater than the numerical solution for $n = 10$, however as n increases the numerical solution gets very close to the analytic solution. This might arise from the Taylor expansion where we neglect certain terms. The comparison of the Gaussian elimination and the LU decomposition in terms of runtime is also quite an eyeopener. As n increases, the runtime for the LU decomposition increases vastly. That is, if we were to choose on of the methods for a large $n \times n$ matrix the LU decomposition would simply take too long to run.

7 Conclusion

As we've seen in this project, the Gaussian elimination algorithm we implemented outperforms the LU decomposition when it comes to runtime. Hence, when one is given a task of solving the Poisson equation and is given a tridiagonal matrix A and a function f, one should think thoroughly through what method to implement and how many FLOPS that method requires. Our algorithm in section 3.1 required $\mathcal{O}(9n)$ FLOPS which is considerably less than the LU decomposition which requires about $\mathcal{O}(n^3)$. When we tried to find the log of the maximum error for each n we expected to see a decrease up to a certain point, what we got, however, was a solution of infinite precision and we clearly don't have that. The LU decomposition works for any invertible matrix and is a standard mathematical tool for solving a set of linear equations. Numerically it's of no use when n gets larger than about 10^3, reaching runtimes of over a minute even at that stage.

References

[1] Morten Hjort-Jensen. Computational physics, lecture notes fall 2015. Department of Physics, University of Oslo, 2015. `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf`.