Project 1 in FYS3150

Christian Lo Virik
Department of Physics, University of Oslo
(Dated: September 9, 2020)

This paper compares numerical solutions to Poisson's equation in one dimension with Dirichlet boundary conditions. We look at LU-decomposition, and also algorithms specialized to the matrix which turns out to be tri-diagonal. We found that LU-decomposition was ∼15,000 times slower than the specialized methods. We also found that the ideal number of grid points was $n \sim 10^{4.8}$, after which round-off errors contaminate the result.

Link to github: https://github.com/clvirik/Project1.git

I. INTRODUCTION

In modern science, no matter the field, using computers to numerically solve differential equations is incredibly useful. In this paper we are going to look at solving Poisson's equation from electromagnetism with Dirichlet boundary conditions[1] numerically in C++.

We will solve it in multiple ways, using LU-decomposition and optimizing it in a special case. We will look at the difference in floating point operations (FLOPs), the computational time, and also how round-off effects the relative error.

The motivation for this project is to determine the most efficient, and the optimal way to solve such a differential equation. This knowledge will help us better understand how computers solve differential equations, and also what their limitations are. We will then be better prepared for solving other problems in the future, so that we can work more efficiently and produce better, more reliable results.

II. METHOD

The differential equation we are going to solve in this paper is Poisson's equation from electromagnetism:

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}) \tag{1}$$

In the case where we have a symmetric charge distribution (independent of θ and ϕ), Poisson's equation becomes:

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

Which with the substitution $x = r$ and $\Phi(r) = u(x)/x$ reads:

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

where $f(x) = 4\pi\rho(x)$.

We will solve this general linear first-order Poisson equation with Dirichlet boundary conditions, i.e.

$$x \in [0,1], \qquad u(0) = u(1) = 0$$

Let us discretize. Let $x_i = ih$ be the grid-points such that $x_0 = 0$ and $x_{n+1} = 1$, and v_i be our approximation to $u(x_i)$. The step length will be $h = 1/(n+1)$, and our boundary conditions will be $v_i = v_{n+1} = 0$. The second derivative can then be approximated with:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f(x_i) \tag{3}$$

for $i = 1, \ldots, n$.

This can be rewritten as a vector equation with $\vec{v} = v_i\vec{e}_i$ and $\vec{g} = h^2 f(x_{i+1})\vec{e}_i$ (summation over i is implied):

$$\mathbf{A}\vec{v} = \vec{g} \tag{4}$$

where $\mathbf{A}$ is the tri-diagonal matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & \vdots \\ 0 & -1 & 2 & -1 & 0 & \vdots \\ \vdots & & & \ddots & & 0 \\ \vdots & \cdots & 0 & -1 & 2 & -1 \\ 0 & & \cdots & 0 & -1 & 2 \end{bmatrix} \tag{5}$$

Since $\vec{g}$ is known, and $\vec{v}$ is the solution to the differential equation, we have turned a calculus problem into a linear algebra problem. We can for instance solve it using Gaussian elimination, LU-decomposition, or by inverting $\mathbf{A}$. We can also (more efficiently) solve it using the special condition that $\mathbf{A}$ is tri-diagonal.

A. LU-Decomposition

An efficient way to solve any system of equations in linear algebra is using LU-decomposition. LU-decomposition consists of factoring the matrix $\mathbf{A}$ into lower- and upper-triangular matrices $\mathbf{L}$ and $\mathbf{U}$. This can always be done if $\mathbf{A}$ is a square matrix. That is

[1] Dirichlet boundary conditions mean that the function evaluates to 0 at the endpoints (0 and 1).

because **U** is just the row echelon form of **A** (which is always possible to find), and **L** is the inverse of the row operations necessary to put **A** in row echelon form. **L** will be lower-triangular because the matrix containing the row operations will be lower-triangular.

When we have the **L** and **U** matrices, our equation to solve (4) reads:

$$\mathbf{L}\mathbf{U}\vec{v} = \vec{g}$$

This means, of course, that we now just need to solve two pretty straightforward sets of linear equations:

$$\mathbf{L}\vec{w} = \vec{g} \tag{6}$$
$$\mathbf{U}\vec{v} = \vec{w} \tag{7}$$

These are easily solved using the Armadillo library in C++:

```
#include <armadillo>

mat A, L, U;
vec w, v, g;

// set A and g values

lu(L, U, A);

w = solve(trimatl(L), g);
v = solve(trimatu(U), w);
```

Now, according to Wikipedia [2], solving a system of linear equations using LU-decomposition takes $2/3n^3$ FLOPs. This is of course the same as Gaussian elimination, as it can be used to perform LU-decomposition. It also makes it more efficient than using QR-decomposition or inversion, but not as efficient as the more specialized methods we will look at next.

In addition we will have to store an entire $(n-1)\times(n-1)$ matrix. If we use doubles we will have to store $8n^2$ bytes, just for the matrix. This limits the number of grid points we can use substantially, as a typical PC only has about 8 GB of RAM (all of which is not accessible). This can also be dramatically improved in the specialized methods.

B. Solving the Tri-Diagonal Case

Since the matrix we are concerned with isn't just any matrix, but a tri-diagonal one, we do not have to store all the terms, just three diagonals. This means that the memory required for the matrix will go like $8\cdot 3n = 24n$. In addition we will have to store the x_i points and the v_i points for a total of $40n$ bytes. That is 4 GB for $n = 10^8$, which is probably as high as we will be able to go at max capacity.

Now let us look at how to solve the tri-diagonal case:

Our general equation (4) now looks like:

$$\mathbf{A} = \begin{bmatrix} d_0 & r_0 & 0 & \cdots & \cdots & 0 \\ l_0 & d_1 & r_1 & 0 & \cdots & \vdots \\ 0 & l_1 & d_2 & r_2 & 0 & \vdots \\ \vdots & & & \ddots & & 0 \\ \vdots & \cdots & 0 & l_{n-4} & d_{n-3} & r_{n-3} \\ 0 & & \cdots & 0 & l_{n-3} & d_{n-2} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} g_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ g_{n-2} \end{bmatrix} \tag{8}$$

The indices are the way they are in order to reflect how they will later be implemented.

A can now be put into row echelon form:

$$\mathbf{A} = \begin{bmatrix} \tilde{d}_0 & r_0 & 0 & \cdots & \cdots & 0 \\ 0 & \tilde{d}_1 & r_1 & 0 & \cdots & \vdots \\ 0 & 0 & \tilde{d}_2 & r_2 & 0 & \vdots \\ \vdots & & & \ddots & & 0 \\ \vdots & \cdots & 0 & 0 & \tilde{d}_{n-3} & r_{n-3} \\ 0 & & \cdots & 0 & 0 & \tilde{d}_{n-2} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} \tilde{g}_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \tilde{g}_{n-2} \end{bmatrix} \tag{9}$$

The equations for $\tilde{d}_i$ and $\tilde{g}_i$ will (through Gaussian elimination) be given by:

$$\tilde{d}_i = d_{i-1} - l_{i-1}r_{i-1}/\tilde{d}_{i-1}$$
$$\tilde{g}_i = g_i - l_{i-1}\tilde{g}_{i-1}/\tilde{d}_{i-1}$$

Since we are always dividing by $\tilde{d}_i$, we pre-calculate it $\breve{d}_i = 1/\tilde{d}_i$, and we get:

$$\breve{d}_i = \left(d_{i-1} - l_{i-1}r_{i-1}\breve{d}_{i-1}\right)^{-1} \tag{10}$$
$$\tilde{g}_i = g_i - l_{i-1}\tilde{g}_{i-1}\breve{d}_{i-1} \tag{11}$$

The initial values are given by $\breve{d}_0 = 1/d_0$ and $\tilde{g}_0 = g_0$.

The equation is then solved through backward substitution:

$$v_i = \left(\tilde{g}_{i-1} - v_{i+1}r_{i-1}\right)\breve{d}_{i-1} \tag{12}$$

Where $v_{n-1} = \tilde{g}_{n-2}\breve{d}_{n-2}$.

We see that the total number of FLOPs from equations (10), (11), and (12) is about 9 or 10 n depending on how you count. This method is implemented as the "Integrate_proper"-method in the C++ code.

C. Our special tri-diagonal case

Given **A** from (5), we can improve the equations from subsection II B. We now have that $d_i = 2$ and that $l_i = r_i = -1$. It can then easily be shown that equations

(10), (11), and (12) can be rewritten as:

$$\breve{d}_{i-1} = \frac{i}{i+1} \tag{13}$$

$$\tilde{g}_i = g_i + \tilde{g}_{i-1}\breve{d}_{i-1} \tag{14}$$

$$v_i = (\tilde{g}_{i-1} + v_{i+1})\,\breve{d}_{i-1} \tag{15}$$

We see that the number of FLOPs drops to 6 to 8 n, depending on whether we store $\breve{d}_i$ in a vector, or recalculate them both times.

If we choose to recalculate the $\breve{d}_i$-values, we only need to store the $\tilde{g}_i$-values in a vector. Then the memory requirement drops from $40n$ to $24n$, as we only need 3 n-dimensional vectors in total.

This method is implemented as the "Integrate_efficient"-method in the code.

D. Relative error

In order to find the relative error, it would be good if we knew the exact solution to or differential equation. We will therefore set the inhomogeneous term to:

$$f(x_i) = 100e^{-10x_i} \tag{16}$$

This gives our differential equation (given the boundary conditions) the exact solution:

$$u_i = 1 - \left(1 - e^{-10}\right) x_i - e^{-10x_i} \tag{17}$$

We can then calculate the relative error between our numerical solutions and the exact one by:

$$\epsilon_i = \log_{10}\left(\left|1 - \frac{v_i}{u_i}\right|\right) \tag{18}$$

This gives us a good way of measuring how accurate our numerical solution is. If we find the maximum relative error from many different step lengths (many different ns), we can then see how accurate different step lengths are, and when round-off errors become significant.

The way we extract the maximum relative errors in this paper is to take the relative error of the second to last point ϵ_n. This is because it seems to always be the largest. More generally the maximum could be extracted by looping through all the grid points in order to find the largest value. Either way we get a good picture of how the relative error changes with n.

III. RESULTS

No matter which one of the methods we use, the final result will be the same. In figure 1 we see that we get very good graphical precision already with $n = 100$.

In figure 2 we see a plot of the maximum relative error ϵ, as a function of n. We see that we get more precise results as n goes up. This lasts all the way until $n \sim 10^{4.8}$, where the relative error starts increasing with n.

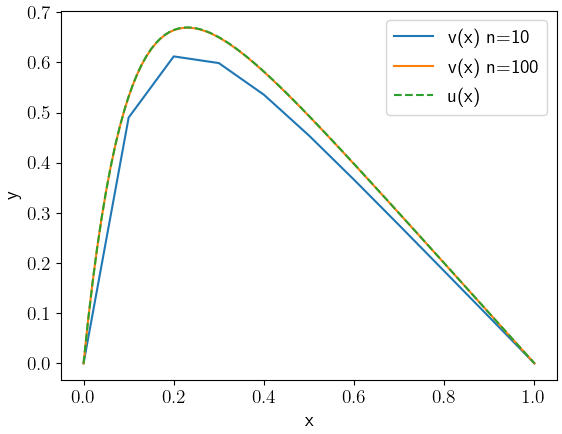

Figure 1. Plot of the exact solution $u(x)$, and the numerical solutions with $n = 10$, and $n = 100$.

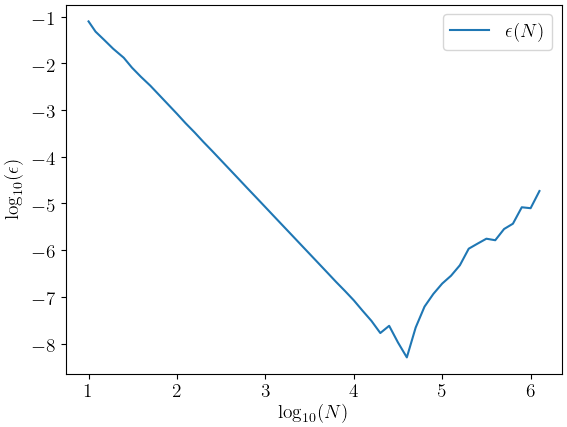

Figure 2. log-log plot of the maximum relative error $\epsilon(n)$.

Moving on to table I, we see how the methods compare in time taken to solve the system. The times were gotten by running each method a large number of times (100 for LU_decomp, and 10,000 for the others).

Method	Time (ms)
Integration_proper	0.0682
Integration_efficient	0.0603
LU_decomp	951.03

Table I. The time taken by the different methods to solve the $n = 1000$ case.

The "LU_decomp"-method runs my out of memory as early as at $n = 10^4$, which would correspond to about 1 GB. The "Integrate_efficient"-method, on the other

hand, is able to perform all the way up to $n = 10^7$ (fails for 10^8). This corresponds to about 240 MB.

IV. DISCUSSION

The rise in ϵ after $n \sim 10^{4.8}$ seen in 2 is due to round-off errors. That means that in this particular case, there is no point in going above this value of grid points. We should also note that even though the errors do get worse after this, they stay very low (below 10^{-4} or 0.01%) in the range of n we have looked at.

In table I we see that the "efficient" method did slightly better than the "proper" method as expected. The LU-decomposition, however, did a lot better than expected. Based on the FLOPs estimate, it should have taken about 1,000,000 times longer than the others (not ∼15,000 as the results show). This is because the FLOPs of LU-decomposition was proportional to n^3, while the others were only proportional to n. From the data it seems however, that the FLOPs of LU-decomposition only goes with n^2. This might be because armadillo has a well optimized LU-decomposition function, able to recognize our special case. It could also have another explanation in the hardware used, or the way the timing was done.

V. CONCLUSION

In this paper we have studied how to solve Poisson's equation in one dimension with Dirichlet boundary conditions. We have seen how storing an entire matrix massively reduces the amount of grid points we can use, and also how LU-decomposition is much slower than other methods in the case of a tri-diagonal matrix. We have also studied how the relative error changes with the number of grid points, and found that round-off errors start becoming noticeable around $n \sim 10^{4.8}$.

[1] Computational Physics I FYS3150/FYS4150. 2020. Project 1, Deadline September 9.
[2] Wikipedia contributors. Lu decomposition — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=LU_decomposition&oldid=965448339, 2020. [Online; accessed 7-September-2020].