FYS3150 - Project 1
Numeric algorithms for solving Poisson's equation with linear algebra

Bror Hjemgaard, Håkon Olav Torvik

September 2020

Abstract

The aim of this paper is to study the properties and effectiveness of different implementations of a numerical solver for the Poisson equation by discretizing the domain through three methods: A general solver based on LU-decomposition, a specialized, optimized version of this general solver and another general solver from the `C++` library `armadillo`. Their computation times were, respectively, of sizes $\mathcal{O}(n^3)$, $\mathcal{O}(n)$ and $\mathcal{O}(n^k)$, where $k \approx 1.2$. The different methods were able to handle different sizes of matrices of sizes $(10^n \times 10^n)$, with their upper limits being, respectively, $n = 4$, $n = 8$ and $n = 4$. We compared the numerical solutions to the analytical one, and compared their maximum relative error. For all three methods the error was of size $\mathcal{O}(n^2)$. This is due to the mathematical lower bound rounding error.

1 Introduction

The Poisson equation is an equation of great importance in many scientific models. In one dimension it can generally be written

$$\frac{d^2u(x)}{dx^2} = f(x). \tag{1}$$

In this project we will rewrite it as a set of linear equations, and implement different algorithms to solve these. We will compare the methods in respect to speed and accuracy.

We will let

$$f(x) = 100e^{-10x}, \tag{2}$$

and use Dirichlet boundary conditions,

$$u(0) = u(1) = 0,$$

to solve the differential equation on the interval

$$x \in (0, 1).$$

This equation has an analytical solution, which we will compare our numerical results with

$$u(x) = 1 - (1 - e^{10})x - e^{-10x}, \tag{3}$$

1.1 Linear set of equations

In order to solve equations numerically, we need to discretize the domain. We let $x \to \{x_0, x_1 \dots x_i \dots x_n\}$, where $x_0 = 0$, and $x_n = 1$. The step length between each point is $h = 1/(n + 1)$. The second derivative of our function u can then be approximated as

$$u_i' = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i, \qquad \text{for } i = 1, \dots, n \tag{4}$$

where $u_i = u(x_i)$ and $f_i = f(x_i)$ denotes the function values of $u(x)$ and $f(x)$ at $x = x_i$. This arises from Taylor-expanding $u(x)$ to the second term, and the truncation error goes as $\mathcal{O}(h^2)$, meaning that (4) can easily be rewritten as

$$-u_{i-i} + 2u_i - u_{i+1} = h^2 f_i. \tag{5}$$

Then, by letting

$$\mathbf{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{bmatrix} \quad \text{and} \quad \tilde{\mathbf{b}} = h^2 \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix},$$

we see that we can "extract" the 3 neighbouring u-values, as in (5), by constructing a tridiagonal matrix $\mathbf{A}$ on the form

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

This is a tridiagonal Töplitz matrix (The three diagonals are constant). Multiplication with $\mathbf{u}$ would yield vector elements equal to (5) for $i = 1, \dots, n$. To solve our initial problem (1) and estimate $\mathbf{u}$ at n points, we only have to solve the matrix problem

$$\mathbf{A}\mathbf{u} = \tilde{\mathbf{b}}. \tag{6}$$

2 Method

Because the original equation, (1) is continuous, and (6) is discrete, we rename the discrete solution $\mathbf{u}$ to $\mathbf{v}$. One method of solving the linear equation (6) would be finding the inverse of $\mathbf{A}$ and use matrix multiplication to solve for $\mathbf{v}$. We will instead use LU-decomposition and then forward and backward substitution to solve the equation, as shown below.

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}} \qquad \text{Original equation} \tag{7}$$

$$\mathbf{L}\mathbf{U}\mathbf{v} = \tilde{\mathbf{b}} \qquad \text{LU-decompose } \mathbf{A} \tag{8}$$

$$\mathbf{L}\mathbf{y} = \tilde{\mathbf{b}} \qquad \text{forward substitution to find } \mathbf{y} \tag{9}$$

$$\mathbf{U}\mathbf{v} = \mathbf{y} \qquad \text{backward substitution to find } \mathbf{v} \tag{10}$$

$\mathbf{L}$ is a lower-diagonal matrix, where all the elements above the diagonal is 0, and $\mathbf{U}$ is an upper-diagonal matrix, where all the elements below the diagonal is 0.

This process is firstly completed for a general $\mathbf{A}$, and thereafter optimized to a specialized algorithm suited the specific tridiagonal Töplitz matrix.

2.1 General case

To obtain the general LU-decomposition of $\mathbf{A}$ we implement the Doolittle algorithm [1]:

```
for $i = 0, 1, \ldots, n-1$
    for $k = i, i+1, \ldots, n-1$
        $U_{i,k} = A_{i,k} - \sum_{j=0}^{i} L_{i,j} U_{j,k}$
    for $k = i, i+1, \ldots, n-1$
        $L_{k,i} = \frac{1}{U_{k,k}} \left( A_{k,i} - \sum_{j=0}^{i} L_{k,j} U_{j,i} \right)$
```

This algorithm is very inefficient for large n, and the number of Floating Point OPerations (FLOPS) goes as $\frac{2}{3}n^3$[2]. When assessing algorithmic efficiency only the largest term is used, as this is the limit as $n \to \infty$. This means that the general LU-decomposition algorithmic efficiency is $\mathcal{O}(n^3)$.

We then perform the substitutions. The forward substitution gives us the elements of $\mathbf{y}$ as follows:

$$y_i = \frac{\tilde{b_i} - \sum_{j=1}^{i-1} L_{i,j} y_j}{L_{i,i}} \quad \text{for} \quad i = 1, 2, \ldots, n-1 \tag{11}$$

Counting the exact FLOPS for this sub-algorithm yields $n^2 + 3n$.

Then backward substitution finally gives us the elements of $\mathbf{v}$ as follows

$$v_i = \frac{\tilde{b_i} - \sum_{j=i+1}^{n} U_{i,j} v_j}{U_{i,i}} \quad \text{for} \quad i = n-1, n-2, \ldots, 1 \tag{12}$$

This sub-algorithm uses the same amount of FLOPS as the forward substitution, $n^2 + 3n$. The total number of FLOPS for the general solver of the problem is the sum of all these,

$$\frac{2}{3}n^3 + 2n^2 + 6n. \tag{13}$$

Only counting the largest term means this algorithm has an efficiency of $\mathcal{O}(n^3)$.

2.2 Special case

Because we are dealing with a tridiagonal Töplitz matrix we do not have to perform all the calculations of a full LU-decomposition. We also do not need to store a full $(n \times n)$ matrix in the memory for $\mathbf{A}$, $\mathbf{U}$ and $\mathbf{L}$. It is easy to observe that

$$A_{i,k} = \begin{cases} -2 & k = i \\ 1 & k = i \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, a general LU-decomposition has infinitely many solutions, where the elements of $\mathbf{L}$ along the diagonal are all the same. To solve this it is common to let them all be equal to 1.

To study the case, we first print out the matrix U for $n = 5$, using the general algorithm:

$$U = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ 0 & 1.5 & -1 & 0 & 0 \\ 0 & 0 & 1.33333 & -1 & 0 \\ 0 & 0 & 0 & 1.25 & -1 \\ 0 & 0 & 0 & 0 & 1.2 \end{bmatrix} \approx \begin{bmatrix} \frac{2}{1} & -1 & 0 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 & 0 \\ 0 & 0 & \frac{4}{3} & -1 & 0 \\ 0 & 0 & 0 & \frac{5}{4} & -1 \\ 0 & 0 & 0 & 0 & \frac{6}{5} \end{bmatrix}.$$

A pattern emerges:

$$U_{i,k} = \begin{cases} \frac{i+1}{i} & k = i \\ -1 & k = i+1 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

We methodically tested this inferred pattern for large n, finding no irregularities. Similarly, for L:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -0.5 & 1 & 0 & 0 & 0 \\ 0 & 0.66667 & 1 & 0 & 0 \\ 0 & 0 & -0.75 & 1 & 0 \\ 0 & 0 & 0 & -0.8 & 1 \end{bmatrix} \approx \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 & 0 \\ 0 & 0 & 0 & -\frac{4}{5} & 1 \end{bmatrix}. \tag{15}$$

We tested this for larger n as well, and the pattern holds. We thus infer that

$$L_{i,k} = \begin{cases} 1 & k = i \\ -\frac{i}{i+1} & k = i-1 \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

The proof that this holds generally is trivial and left as an exercise to the reader.

We see that **U** is an upper-diagonal matrix with the elements on the second upper diagonal equalling -1, with only the diagonal elements changing from index to index and the rest of the elements being zero. Thus, **U** only contains n *unique* elements, meaning we can reduce its computational representation from an $(n \times n)$ matrix to a vector U' of size n. Similarly, **L** only stores $n-1$ *unique* elements along its lower diagonal and can be represented by a vector L' of size $n-1$.

Each computation of (14) and (16) require 3 FLOPS[1]. However, as the k^{th} element is independent of other elements, this is considered prior, and not part of the algorithm for solving the problem. By using both of the analytical expressions (14) and (16) as well as representing the matrices as vectors, the entire LU-decomposition only uses $6n$ FLOPS and 8 bytes $\times 2n = 16n$ bytes of memory.[2]

Using the above-mentioned methods, it is also possible to optimize the forward- and backwards substitutions in our algorithm. Invoking (16) on the sum in (11), only the last term is non-zero, giving the expression

$$y_i = \tilde{b_i} - L'_i y_{i-1} \quad \text{for} \quad i = 1, 2, \ldots, n-1, \tag{17}$$

requiring $2n$ FLOPS. Similarly, the optimized backward substitution becomes

$$v_i = \frac{\tilde{b_i} + v_{i+1}}{U'_{i+1}} \quad \text{for} \quad i = n-2, n-3, \ldots, 0, \tag{18}$$

again requiring $2n$ FLOPS.
This algorithm now uses $4n$ FLOPS, giving an efficiency of $\mathcal{O}(n)$, which is extremely efficient compared to the general algorithm of $\mathcal{O}(n^3)$. Also memory usage should be drastically reduced, due to the vector representation of the matrices.
We have effectively arrived at the Thomas Algorithm.

3 Implementation

The equations were solved numerically with C++ and the results visualized with `python`.
All code can be found at `https://github.com/BrorH/FYS3150/tree/master/project1/code`.
We implemented both methods as described in the section above. We also made a program to solve the matrix equation (6) by using the `C++` package `armadillo`.

[1]Presumably calculating the elements of U should, according to (14), only be 2 FLOPS, but due to some non-optimal indexing, it was increased by one in our code.

[2]The vectors are filled with doubles of size 8 bytes.

3.1 Benchmarking

For each step we also calculate the relative error ϵ_i between the computed numerical value, and the analytical value.

$$\epsilon_i = \log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right). \tag{19}$$

The greatest of these errors, dubbed ϵ_{max}, is stored for each n to be studied and plotted. This error is used to make sure the solution is correct. As mentioned, the second derivative approximation used in (4) gives a truncation-error of magnitude $\mathcal{O}(h^2)$. This means we expect the log-plot of $\epsilon_{max}(h)$ to be a straight line, with slope 2. We have instead chosen to plot $\epsilon_{max}(n) = \epsilon_{max}\left(\frac{1}{h}\right)$, and expect therefore a slope of -2.

For both our general and optimized algorithms, we plotted the maximum relative error ϵ_{max} and the elapsed CPU time, both on a logarithmic axis. As we assumed both the relative error and the CPU time to be of scale $\mathcal{O}(n^k)$, for some k, these logarithmic plots should therefore be linear in nature, so linear regression was applied in order to estimate a fitting linear function on the form $ax + b$, where $x = \log(n)$. This was also applied to the `armadillo` method.

4 Results

4.1 General algorithm

Using our general algorithm for LU-decomposition for $\log n = 1, 2, 3$, we compared the numerical solutions v_i to that of the analytical in (3), illustrated in Figure 1. We also measured the elapsed CPU time during the solving, as well as calculating ϵ_{max}, (19), for each n. See Table 1.

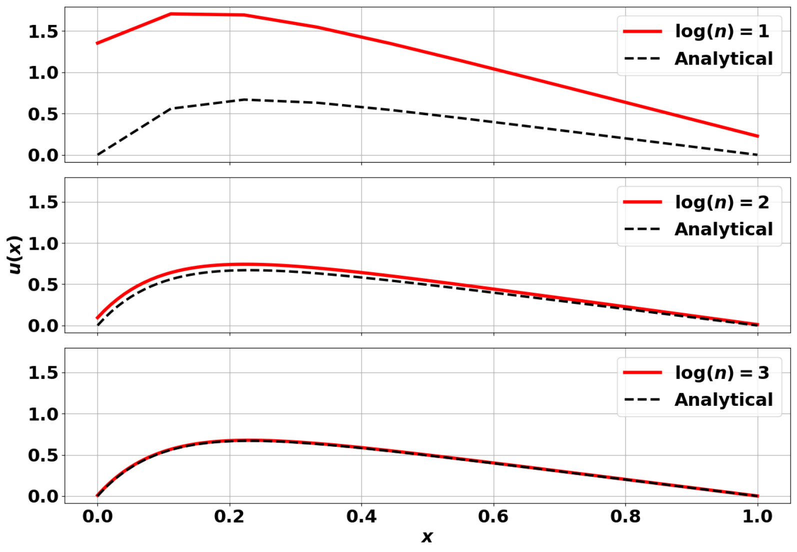

Figure 1: GENERAL: Plot of the estimated solution to the problem, plotted against the analytical solution, for 3 successively larger n

As seen in 1 the solution steadily approaches the analytical one for higher n. We did not plot these solutions for the other methods/higher n-values, as they are all effectively identical plots.

Table 1: GENERAL: CPU time and ϵ_{max}

$\log(n)$	CPU time [s]	ϵ_{max}
1	5.00e-06	1.42e-02
2	4.76e-04	9.09e-05
3	6.70e-01	8.73e-07
4	4.94e+03	8.69e-09

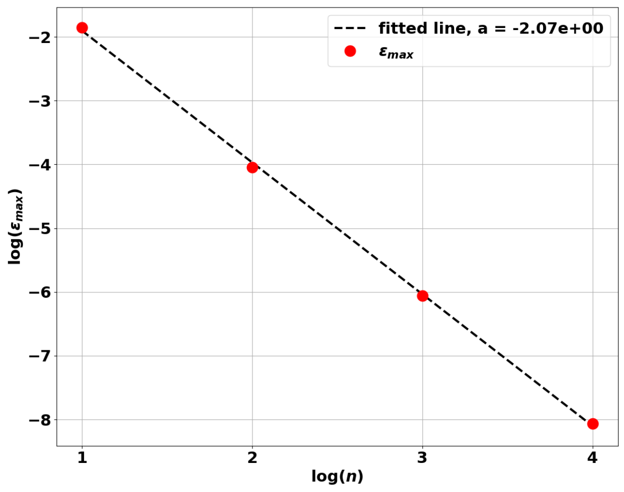

Figure 2: GENERAL: Logarithmic plot of ϵ_{max}

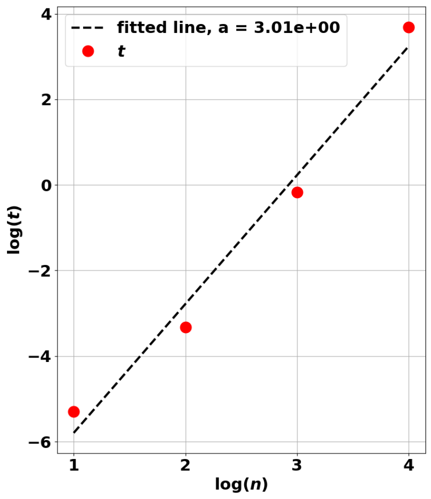

Figure 3: GENERAL: Logarithmic plot of CPU time

The figures above hint to errors decreasing as $\mathcal{O}(n^{-2})$ and time increasing as $\mathcal{O}(n^3)$. Its linearity is as expected.
Theoretically, we estimated the the total number of flops for the general algorithm to be $\frac{2}{3}n^3 + 2n^2 + 6n$ for a matrix of size $n \times n$ (eqn. (13)). In order to test this, we added a basic FLOP-counter to the general algorithm, and compared them to the estimated. The results are in Table 2 below.

Table 2: Predicted vs. counted FLOPS in general LU-decomposition

$\log(n)$	Predicted	Counted	Relative difference
1	927	825	11%
2	687267	681750	0.80%
3	668672667	668167500	0.076%
4	666866726667	666816675000	0.0075%

A neat decimal pattern is quickly visible in the "counted" column. However, there are some interesting discrepancies for $\log n = 1$. The relative difference quickly converges.

4.2 Optimized algorithm

Using the optimized method discussed earlier, we were able to solve the system for sizes up to $\log(n) = 8$. Thereafter our 32 GB RAM computers were insufficient.

The solutions were also calculated *much* faster (compare $\log(n) = 4$ for general method). The results in Table 3 below, The logarithmic plots as shown in Figures 4 and 5.

Table 3: OPTIMIZED: CPU time and ϵ_{max}

$\log(n)$	t [s]	ϵ_{max}
1	1.00e-06	1.42e-02
2	3.00e-06	9.09e-05
3	2.50e-05	8.73e-07
4	2.13e-04	8.69e-09
5	1.98e-03	8.69e-11
6	2.17e-02	8.69e-13
7	2.06e-01	8.68e-15
8	2.13e+00	9.64e-17

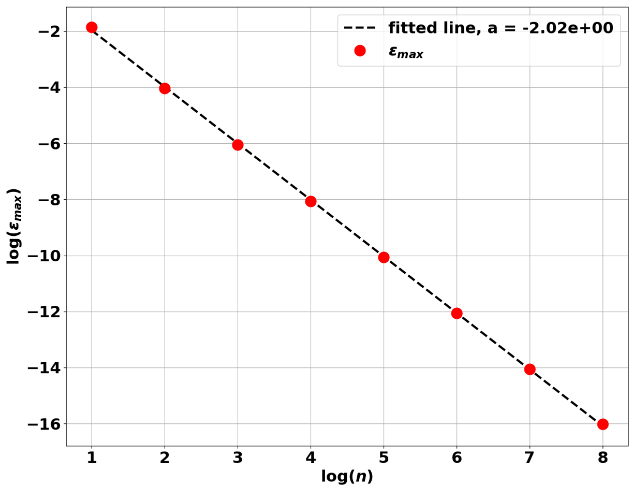

Figure 4: OPTIMIZED: Logarithmic plot of ϵ_{max}

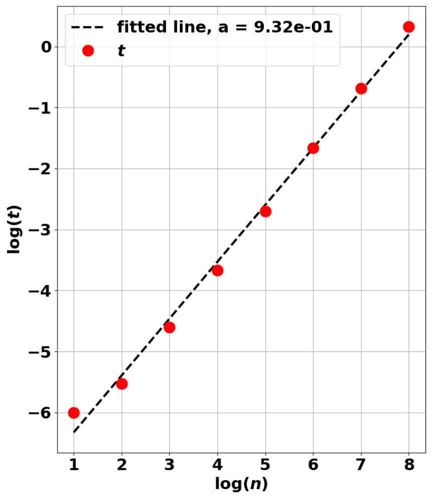

Figure 5: OPTIMIZED: Logarithmic plot of CPU time

As expected, the error decreases with $\mathcal{O}(n^{-2})$. The time, however, seems to be *much* lower than the general case.

4.3 Solving with armadillo

In order to further study the precision and computation time of our specialized solution, we compared it to those of the `C++` library `armadillo`, that specializes in linear algebra and matrix operations, and includes a `solve`-function that solves linear systems on the form $\mathbf{Ax} = \mathbf{b}$. We time the CPU-time and record the error for solutions up to $\log(n) = 4$. The results are listed below in Table 4.

Table 4: ARMADILLO: CPU time and ϵ_{max}

$\log(n)$	t [s]	ϵ_{max}
1	5.84e-04	1.42e-02
2	1.03e-03	9.09e-05
3	1.01e-01	8.73e-07
4	1.23e+00	8.69e-09

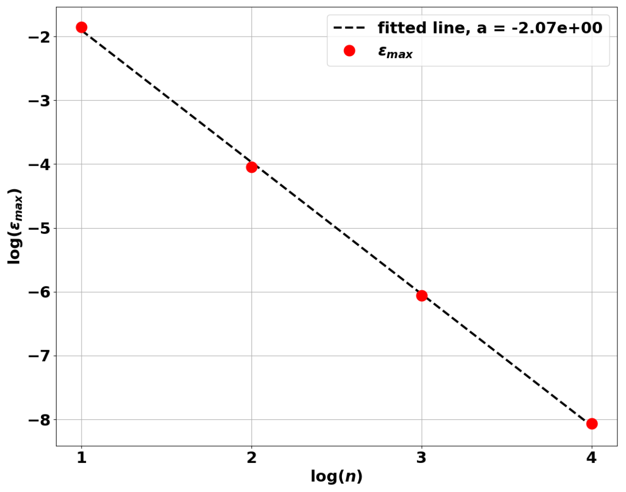

Figure 6: ARMADILLO: Logarithmic plot of ϵ_{max}.

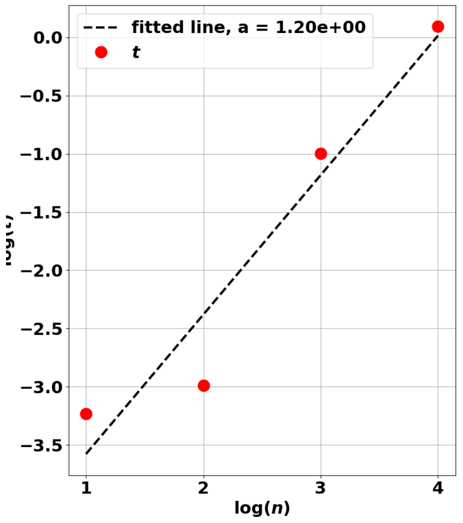

Figure 7: ARMADILLO: Logarithmic plot of elapsed time.

Following the pattern, the error goes like $\mathcal{O}(n^{-2})$ here as well. The first two data points in the time plot seem more erratic than the latter two.

5 Discussion

Predictably, all the error plots in Figures 2, 4 and 6 follow that of $\mathcal{O}(n^{-2}) = \mathcal{O}(h^2)$. This is of no surprise as the mathematical computation is identical amongst the three methods, leaving room for nothing but rounding point errors to distinguish them.
However, curiously enough, we never encountered any rounding point errors, as shown in Figure 4. The graph is linear all the way through. Presumable we should have hit some threshold where the numerical errors are larger than the mathematical ones.
By Figure 3 the time is of shape $\mathcal{O}(n^3)$, effectively confirming our assessment of the general LU-decomposition being of shape $\mathcal{O}(n^3)$.

For our optimized solver the time in Figure 5 is of size $\mathcal{O}(n^{0.9})$, which is less than the theoretical minimum of $\mathcal{O}(n)$. Our wrong value probably comes from the deviations from the calculations of lower order n's. To study this, we re-drew Figure 5 for only the last 6 points. See Figure 8. Or optimized method has approached a computation time of scale $\mathcal{O}(n)$, which is the lowest theoretically possible.

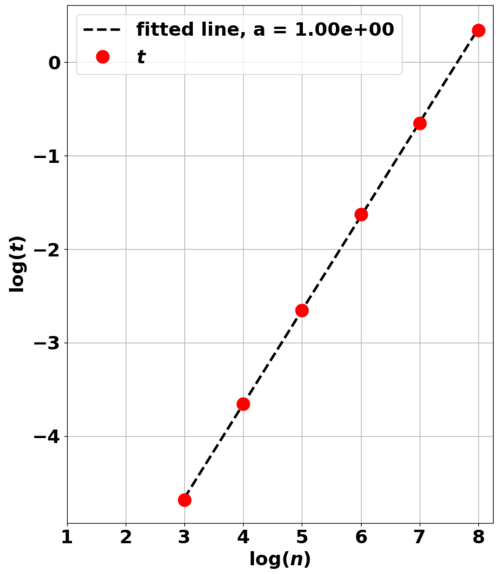

Figure 8: OPTIMIZED: Elapsed time with linear model fitted only along the last 6 points. The elapsed time then clearly follows $\mathcal{O}(n)$.

Lastly, the elapsed time for the `armadillo` method was, by Figure 7, approximately of size $\mathcal{O}(n^{1.2})$. We cannot confidently say if this is a fair estimation of its size, as 4 data points is not enough to get an accurate representation. It does, however, seem that the first two points are more erratic than the latter two, maybe hinting to some converging integer k in $\mathcal{O}(n^k)$. For $\log(n) \geq 5$ the `armadillo` library runs out of memory. This is because it tries to store a $(10^5 \times 10^5)$ matrix of doubles in memory. Doubles have a size of 8 bytes, meaning this matrix would require 8 bytes $\times 10^{10} = 80$ GB. In general, the memory needed by the $(10^n \times 10^n)$ matrix is $8 \cdot 10^{2n}$ bytes. According to the documentation, `solve` generally has an efficiency of $\mathcal{O}(n^3)$, which is equal to that of our general solver. However, comparing Table 1 and Table 4, our general solver is *faster* than `solve`. This is most likely due to the general method-of-solution used by `solve`, as it not necessarily uses LU-decomposition.
Comparing our tabular results from Table 1, Table 3 and Table 4 we see that the shape of ϵ_{max} remains constant throughout our three methods of solution. This tells us that the assumed patterns in (14) and (16) seem to hold equally well for large and small n. Furthermore, the identical errors from `armadillo`, which most likely does *not* use LU-decomposition, tells us that our errors probably come from the discretization of the function, and the accompanying mathematically inevitable inaccuracy.
Table 2 shows that the general LU-decomposition algorithm uses effectively the same number of FLOPS as we predicted using (13). We see that it is better for larger n, possibly due to small sub-algorithms whose usage is independent of n. For $\log(n) = 1$ the relative difference is much larger than for higher $\log(n)$. It is worth noting, that for the sake of simplicity that when counting FLOPS we do not count index arithmetic like, x[i + 1], as a FLOP. This is partially due to their absence in the mathematical algorithm.

6 Conclusion

We successfully managed to numerically solve the Poisson equation using linear algebra with a general algorithm, and with an optimized algorithm as well using packages which did all the calculations for us.

For all methods we observed that the maximum relative error compared to the analytical solution went as $\mathcal{O}(n^{-2})$. This was as expected from the mathematical truncation error in discrediting the domain. Curiously enough, our best method did not encounter rounding point errors.

The limiting factor for how large n we could run was memory-usage for all the methods. However, the general method was very slow, which also gave an effective limitation lower than the theoretical one, at $\log(n) = 3$ instead of 4.[3] It was interesting how we, in the optimized method, were able to drastically decrease the memory needed for solving the problem.

We estimated the number of FLOPS the algorithms used for an arbitrary n to high precision, achieving a relative difference of 0.008% for $\log(n) = 4$.

References

[1] GeeksforGees *Doolittle Algorithm: LU Decomposition* [website] Availible on https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/ [Visited 31.08.2020]

[2] Wikipedia *LU-decomposition using Gaussian elimination* Availible on https://en.wikipedia.org/wiki/LU_decomposition#Using_Gaussian_elimination [Visited 07.08.2020]

[3] In Table 1 calculating for $\log(n) = 4$ took 1 h 20 m. This is much longer than anything else we simulated.