Draft version September 8, 2019
Preprint typeset using LaTeX style emulateapj v. 12/16/11

COMPARING EFFICIENCY OF DIFFERENT ALGORITHMS FOR SOLVING SETS OF LINEAR EQUATIONS

Bruce Chappell and Markus Bjørklund

Draft version September 8, 2019

ABSTRACT

In this letter we analyze and compare the efficiency of a general, tailored and a LU-decomposition algorithm to solve a set of linear equations. The problem we will focus on is a discretized approximation to the one dimensional Poisson equation. The results show that, for n equations, we get $8n-7$ floating point operations (FLOPS) for the general algorithm, $4n-3$ FLOPS for the tailored algorithm and $\frac{2}{3}n^3$ FLOPS for the LU-decomposition method. We found that the optimal step length is $h = 10^{-7}$. The results also show that the LU decomposition can not handle cases where the matrix is bigger or equal to $n = 10^5$, whilst the other algorithms handle this very well.

Subject headings: Algorithm efficiency — Linear algebra: simulation — methods: computational

1. INTRODUCTION

Solving differential equations is an integral part of most research today. For most differential equations there exists no analytic solution, so we have to use a computational approach. Generally, the more realistic the model is, the more complex it is, and therefore more computationally demanding. There would be a lot to gain to increase efficiency on the algorithms used to solve such equations, thereby allowing for more efficient memory usage to solve more complex problems or increase the resolution of a given discretization. We have presented theory for the different algorithms in the method section, provided results for CPU runtime and error estimation in the results section, and drawn some conclusions. We have also proposed future steps in the further research section.

In this letter we will adopt a numerical approach to solving the one dimensional Poisson equation, which shows up in numerous areas of science.

2. METHOD

This section will describe the methods we utilized to acquire our results.

2.1. *Model*

We want to solve the one dimensional Poisson equation shown in equation 1.

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

We confine the equation to a normalized interval and with Dirichlet boundary conditions, namely

$$x \in [0,1]\ ,\ u(0) = u(1) = 0. \tag{2}$$

For reference, we assume the source term to be

$$f(x) = 100e^{-10x}. \tag{3}$$

The equation then has an analytic solution in

$$u(x) = 1 - \left(1 - e^{-10}\right)x - e^{-10x}. \tag{4}$$

markus.bjorklund@astro.uio.no
[1] Institute of Theoretical Astrophysics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway

We discretize the interval to $n+2$ points, so that $x_0 = 0$ and $x_{n+1} = 1$, giving a step length of $h = \frac{1}{n+1}$. We approximate the second derivative to the second order, giving equation 5 for a point $f(x_i)$:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \tag{5}$$

This can be rewritten as

$$-v_{i-1} + 2v_i - v_{i+1} = f_i \cdot h^2 \tag{6}$$

for $i = 1, ..., n$ and the end points decided by the boundary conditions. We then get a set of n equations like following

$$\begin{aligned}
-v_0 + 2v_1 - v_2 &= f_1 \cdot h^2 \\
-v_1 + 2v_2 - v_3 &= f_2 \cdot h^2 \\
-v_2 + 2v_3 - v_4 &= f_3 \cdot h^2 \\
&\vdots
\end{aligned}$$

Written as a matrix equation, this produces the equation

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{f}} \tag{7}$$

with $\tilde{\mathbf{f}} = \mathbf{f} \cdot h^2$ and the tri-diagonal $n \times n$ matrix

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{pmatrix}. \tag{8}$$

2.1.1. *General case*

We first address the general case, where we don't assume the values of the upper, lower and main diagonal. Adopting separate names for the three diagonals, **a**,**b** and **c**, we can represent the equation by the agumented matrix

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & & 0 \\ a_1 & \ddots & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{f}_1 \\ \vdots \\ \vdots \\ \tilde{f}_n \end{pmatrix}. \quad (9)$$

We solve this by first a forward substitution, I.E removing the elements below the pivot elements (**a**) in the augmented matrix, to obtain an upper triangular matrix. The diagonal elements will be updated as

$$\tilde{b_2} = b_2 - \frac{a_1 \cdot c_1}{b_1}, \quad (10)$$

and the source term will be updated as

$$\tilde{\tilde{f_2}} = \tilde{f_2} - \frac{a_1 \cdot \tilde{f_1}}{b_1}. \quad (11)$$

Generally, the scheme will become

$$\tilde{b_i} = b_i - \frac{a_{i-1} \cdot c_{i-1}}{\tilde{b}_{i-1}}, \quad (12)$$

and

$$\tilde{\tilde{f_i}} = \tilde{f_i} - \frac{a_{i-1} \cdot \tilde{f}_{i-1}}{\tilde{b}_{i-1}} \quad (13)$$

iterated from 2 to n. We then obtain an upper triangular matrix with updated elements, shown as the following augmented matrix.

$$\mathbf{C} = \begin{pmatrix} b_1 & c_1 & & 0 & \tilde{f}_1 \\ 0 & \ddots & \ddots & & \vdots \\ & \ddots & \ddots & c_{n-1} & \vdots \\ 0 & & 0 & \tilde{b_n} & \tilde{\tilde{f_n}} \end{pmatrix} \quad (14)$$

Now, we normalize the diagonal, which gives us

$$u_i = \frac{\tilde{\tilde{f_i}}}{\tilde{b_i}}, \quad (15)$$

and

$$\tilde{c}_i = \frac{c_i}{\tilde{b_i}}, \quad (16)$$

We now perform the backward substitution, I.E removing the elements above the main diagonal ($\tilde{\mathbf{c}}$). The main diagonal stays unchanged, but the source terms updates as follows

$$\tilde{u}_{i-1} = u_{i-1} - \tilde{c}_{i-1} u_i. \quad (17)$$

However, if we insert for u_{i-1} and $\tilde{c}_{i-1}$ from equations 15 and 16 respectively, we see that we can write this as one operation:

$$\tilde{u}_{i-1} = \frac{\tilde{\tilde{f}}_{i-1}}{\tilde{b}_{i-1}} - \frac{c_{i-1}}{b_{i-1}} u_i = \frac{\tilde{\tilde{f}}_{i-1} - c_{i-1} u_i}{b_{i-1}}. \quad (18)$$

We iterate this from n to 2.

2.1.2. *Special case*

We return to the special case illustrated in equation 8, where the diagonals have the same elements. Looking back to equation 12, we now know that all a's and c's will be -1, and all b's will be 2. We then get

$$\tilde{b_i} = 2 - \frac{1}{\tilde{b}_{i-1}}. \quad (19)$$

With $b_1 = 2$, we see that we get $b_2 = \frac{3}{2}$, $b_3 = \frac{4}{3}$, $b_4 = \frac{5}{4}$, and we can write equation 21 as

$$\tilde{b_i} = \frac{i+1}{i}. \quad (20)$$

Similarly, comparing back to equation 13, we now get

$$\tilde{\tilde{f_i}} = \tilde{f_i} + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}}. \quad (21)$$

However, inserting from our result in equation 20, we get

$$\tilde{\tilde{f_i}} = \tilde{f_i} + \frac{\tilde{f}_{i-1}}{\frac{(i-1)+1}{(i-1)}} = \tilde{f_i} + \frac{(i-1)\,\tilde{f}_{i-1}}{i}. \quad (22)$$

This is iterated from 2 to n.

Finally, the backwards substitution. We look back to equation 18, and see that it becomes

$$u_{i-1} = \frac{\tilde{\tilde{f}}_{i-1} + u_i}{b_{i-1}}. \quad (23)$$

and substituting in the result from 20 again, we get

$$u_{i-1} = \frac{\tilde{\tilde{f}}_{i-1} + u_i}{\frac{(i-1)+1}{(i-1)}} = \frac{(i-1)}{i}\left(\tilde{\tilde{f}}_{i-1} + u_i\right). \quad (24)$$

Thus, we can precalculate the factor $\frac{i-1}{i}$, and will only count as a constant multiplication (one flop) in each iteration.

2.1.3. *LU decomposition*

A standard way to solve linear equations is the LU decomposition. For an equation $\mathbf{Ax} = \mathbf{b}$, we decompose A into an upper- and lower triangular matrix, which produces the equation

$$\mathbf{LUx} = \mathbf{b}. \quad (25)$$

Now grouping $\mathbf{U}$ and $\mathbf{x}$ to form a new vector, we get the two equations

$$\mathbf{Ux} = \mathbf{y}, \quad (26)$$

$$\mathbf{Ly} = \mathbf{b}. \quad (27)$$

These equations can be solved in succession for each point i.

3. RESULTS

3.1. *Number of floating point operations in each algorithm*

The amount of FLOPS for the first algorithm will in total be: $3(n-1)$ from equation 12, $2(n-1)$ from 13

TABLE 1

n	General	Tailored	LU
10	$1 \cdot 10^{-6}$	$3 \cdot 10^{-6}$	$1.130 \cdot 10^{-4}$
10^2	$2 \cdot 10^{-6}$	$7 \cdot 10^{-6}$	$8.190 \cdot 10^{-4}$
10^3	$1.1 \cdot 10^{-5}$	$4.800 \cdot 10^{-5}$	$6.233 \cdot 10^{-2}$
10^4	$1.880 \cdot 10^{-4}$	$4.530 \cdot 10^{-4}$	$1.889 \cdot 10^{2}$
10^5	$1.925 \cdot 10^{-3}$	$1.694 \cdot 10^{-3}$	Insufficient memory
10^6	$1.880 \cdot 10^{-2}$	$1.601 \cdot 10^{-2}$	Insufficient memory
10^7	$1.907 \cdot 10^{-1}$	$1.160 \cdot 10^{-1}$	Insufficient memory

NOTE. — CPU time for each algorithm. All values are given in seconds. Note: only the algorithm, I.E the forward and backward substitution, is timed.

(since $\frac{a_{i-1}}{b_{i-1}}$ is already calculated), and $3\,(n-1)$ from equation 18. We will also require one final flop to normalize the last diagonal element $\tilde{\tilde{b}}_n$. So in total, we get $8n-7$ FLOPS.

The resulting FLOPS for the tailored algorithm will for equation 22 be $2\,(n-1)$ and for equation 24 we get $2\,(n-1)$. We also need to normalize u_n before we start the backwards substitution, so this adds one flop. In total we get $4n-3$ FLOPS.

The amount of FLOPS for the LU-decomposition method has already been thoroughly stated in the literature, and turns out to be $\frac{2}{3}n^3$ (Hjorth-Jensen, 2018). The CPU time in seconds is shown in table 1.

3.2. *Numerical error*

These methods converge to the solution quickly. Already for $n = 100$ we are really close to the analytic solution. A plot comparing the analytic and numerical solution for the general algorithm is shown in figure 1 and 2

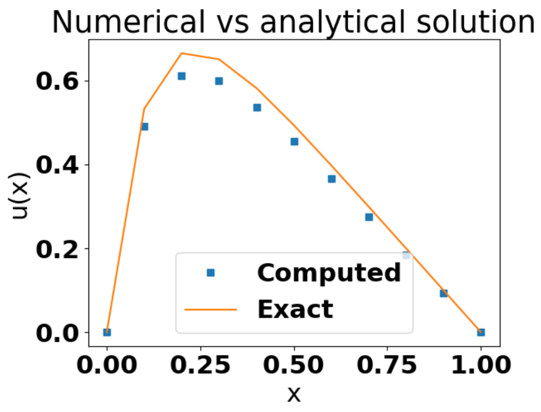

FIG. 1.— Numerical solutions plotted against the analytic solution for n=10. All figures are available in the Github repository

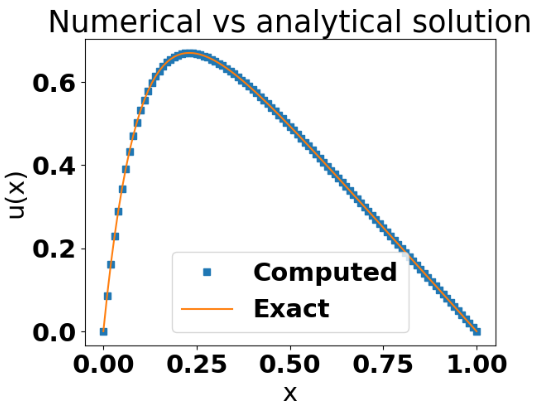

FIG. 2.— Numerical solutions plotted against the analytic solution for n=100. All figures are available in the Github repository

The error as a function of step size is also shown in figure 3. We can see the error goes as a linear graph with slope 2, which shows in a logarithmic plot that the error scales as h^2.

This shows that a smaller step size is not always better, as we reach a minimum of error at step size $h = 10^{-5}$. This is because for very small step sizes, the error in adding and subtracting nearly equal numbers increase, I.E. $u(x+h)+u(x-h)-2u(x)$ for our numerical derivative. We again divide this by h^2, further blowing up the error for small step sizes h, leading to loss of numerical precision.

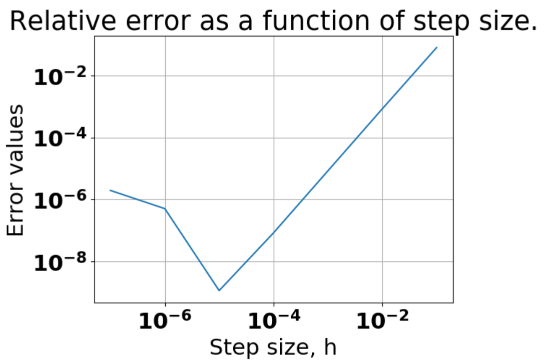

FIG. 3.— Logarithmic plot of relative error plotted against step size.

4. CONCLUSIONS

The results in this letter shows that when faced with a set of linear equations, one has a lot to gain to tailor your algorithm to the special case you are dealing with. Saving CPU time by a factor of 2 is substantial, and comparing to the LU-decomposition, for which the CPU time differs by a factor of n^2, tailoring the algorithm is absolutely essential.

We have also seen that for some cases, the problem is not even solvable by the standard LU-decomposition. For matrices with $n = 10^5$ or above, we did not have enough memory to perform the algorithm. However, the tailored methods work just fine, since we do not have to construct a whole matrix.

4.1. *Further research*

In this letter we have focused on the Poisson equation, with a specific second order approximation. However, similar analysis can be done for several other types of equations, or different approximations. There would be a lot to gain by gaining as efficient methods as possible for each specific case, so that bigger and more complex systems can be simulated and analyzed.

REFERENCES

Hjorth-Jensen, Morten Sep 6 2018, "Computational Physics Lectures: Linear Algebra" methods, http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/pdf/linalg-beamer.pdf

5. APPENDIX

All source code, data and figures can be found at the github repository: https://github.com/bruce-chappell/FYS4150/tree/master/project1