Numerical integration using the Thomas algorithm

Anders P. Åsbø
(Dated: September 10, 2019)

The focus of this paper was the specific case of numerically integrating a linear, second order differential equation as a system of linear equations by deriving, and implementing the Thomas algorithm[1], as well as the precision and computational speed of the algorithm.

It proved to be a efficient, and sufficiently precise method for integrating a second order, linear differential equation. The specialized variant of the algorithm proved especially memory efficient. Both the general, and specialized Thomas algorithms are preferable for the specific usecase in this paper, as they used significantly less memory, and time than a general LU-decomposition solver.

CONTENTS

I. INTRODUCTION

Numerical integration is an important cornerstone of computational physics, and as such it is important to understand its limits in terms of numerical precision and time spent computing. In this report I have looked at the specific case of numerically integrating a linear, second order differential equation as a system of linear equations, with the pretext of solving a one-dimensional variant of Poisson's equation. The focus of this paper will be on deriving, and implementing the Thomas algorithm[1], as well as the precision and computational speed of the implementation.

II. FORMALISM

A. Underlying theory

If I have a charge distrobution $\rho(\vec{r})$, as a function of the position $\vec{r}$, Possion's equation gives the electrostatic potential Φ

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

[2]. If I then assume both $\rho(\vec{r})$, and Φ to be spherically symetric, the equation can be simplified to

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

where $r = |\vec{r}|$. Substituting $\Phi(r) = \frac{1}{r}\phi(r)$ gives

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

which, for simplicity, can be written as

$$-u''(x) = f(x),$$

with $u = \phi$, $f = -4\pi\rho$, and $x = r$[2].

Having arrived at a simple formulation of the initial problem, I can now discretize it using the second order differential approximation

$$-\frac{u(x+h) + u(x-h) - 2u(x)}{h^2} = f(x).$$

Since I will be opperating with a discreete set of variables $x_i \in [0,1]$ where $x_i = ih$ for $i = 1, 2, 3, ..., N$, I can write the afformentioned approximation as

$$-\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f_i,$$

where $u_i = u(x_i)$. The step-size is defined as $h = \frac{x_N - x_0}{N}$, and I impose the Dirichlet boundary conditions $u(0) = u(1) = 0$[2].

B. Deriving an algorithm for square, tridiagonal matricies

The discretized approximation obtained in II.B can be rearranged to

$$-1u_{i-1} + 2u_i - 1u_{i+1} = h^2 f_i,$$

which describes a tridiagonal, $N \times N$-matrix

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

and with the unkowns in a vector

$$\vec{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix},$$

and the values of $h^2 f_i = g_i$ in a vector

$$\vec{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_N \end{bmatrix},$$

I can express the integration problem as a matrix equation

$$A\vec{u} = \vec{g}.$$

For a general matrix

$$A = \begin{bmatrix} d_1 & a_1 & 0 & \dots & \dots & 0 \\ b_1 & d_2 & a_2 & \ddots & \ddots & \vdots \\ 0 & b_2 & d_3 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a_{N-1} \\ 0 & \dots & \dots & \dots & b_{N-1} & d_N \end{bmatrix}$$

[2]. A way to solve the matrix equation is by Gaussian elimination. I start by subtracting row 1 multiplied by $\frac{b_1}{d_1}$ from row 2, and continue by subtracting row 2 multiplied by $\frac{b_2}{d_2}$ from row 3. This continues all the way down such that a general algorithm can be written as

$$d_i^* = d_i - \frac{b_{i-1}a_{i-1}}{d_{i-1}^*}, \tag{1}$$

with the condition that $d_1^* = d_1$. The same pattern aplies to $\vec{g}$, such that

$$g_i^* = g_i - \frac{b_{i-1}g_{i-1}}{d_{i-1}^*}, \tag{2}$$

where $g_1^* = g_1$, such that we get a new vector $\vec{g}^*$ with the adjusted values. The two above algorithems constitutes the decomposition and forward substitution of the matrix, and gives $u_N = \frac{g_N^*}{d_N^*}$. The remaining values of u can then be calculated recursivly as

$$u_i = \frac{g_i^* - a_i u_{i+1}}{d_i^*}, \tag{3}$$

with $i = N-1, ..., 2, 1$. This algorithm constitutes the backward substitution.

The complete algorithm is known as the Thomas algorithm, and is named after Llewellyn Thomas[1],.

III. IMPLEMENTATION

A. Implementing the general Thomas algorithm

To execute the numerical integration using (1), (2), and (3), I wrote the program "project.py"(A.1) which takes the number of step points N, as well as a label for the data files, as input from the user, and initializes an array of linearly spaced values of $x_i \in [0, 1]$, as well as the step-size. Furthermore, it calls a custom module "data_generator.py"(A.3), which generates saves an array of f_i values to a textfile. "data_generator.py"(A.3) also generates a seperate textfile containing an array of the analytical solution to the differential equation, since I am using the function

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

$$u'(x) = (1 - e^{-10}) - 10e^{-10x},$$

$$u''(x) = -100e^{-10x} = -f(x).$$

Both the textfile containing f_i, and the analytical solution are read by "project.py"(A.1), which stores them as arrays. The program also initializes empty arrays for $\vec{u}$, $\vec{d}^*$, and $\vec{g}^*$. The tridiagonal matrix was initialized as three arrays, one holding b_i, one holding d_i, and one holding a_i. Thus I avoid filling memory with the zero-elements.

"project.py"(A.1) continues, by setting the boundary conditions, and looping through (1), and (2) in one loop, and (3) in a second loop. For the decomposition and forward substitution, the factor $\frac{b_{i-1}}{d_{i-1}^*}$ was calculated before including it in (1), and (2), reducing the number of FLOPs from $6(N-1)$ to $5(N-1)$. Due to the boundary conditions, the backward substitution requires $3(N-2)$ FLOPs, resulting in a total of $8N - 11 \approx 8N$ FLOPs for sufficently large values of N.

Finally, "project.py"(A.1) saves the numerial solution $\vec{u}$ to a textfile, and plots it against the analytical solution.

B. Implementing a specialized version of the Thomas algorithm

Because the specific tridiagonal matrix in the problem has all diagonal elements equal to 2, and all non-zero, off-diagonal elements equal to -1, there was some optimization to be done, thus a specialized version of the integration program can be found in "project_specialized.py"(A.1).

Substituting in the known values, (3) can be reduced to

$$g_i^* = g_i + \frac{g_{i-1}^*}{d_i^*},$$

reducing the backward substitution to $2(N-2)$ FLOPs. Furthermore, the adjusted, diagonal elements can be expresed by an explicit formula

$$\frac{1}{d_i^*} = \frac{2i}{2(i+1)}.$$

Utilizing NumPy's elementwise opperations, "project_specialized.py"(A.1) precalculates all $\frac{1}{d_i^*}$ in parallell, reducing the decomposition and forward substitution to $2(N-1)$ FLOPs. This effectivly halves the number of FLOPs of the total algorithm to $4N-6 \approx 4N$, the arrays for the initial matrix elements were removed to as they are no longer needed.

IV. ANALYSIS

A. Plotting the genneral Thomas algorithm

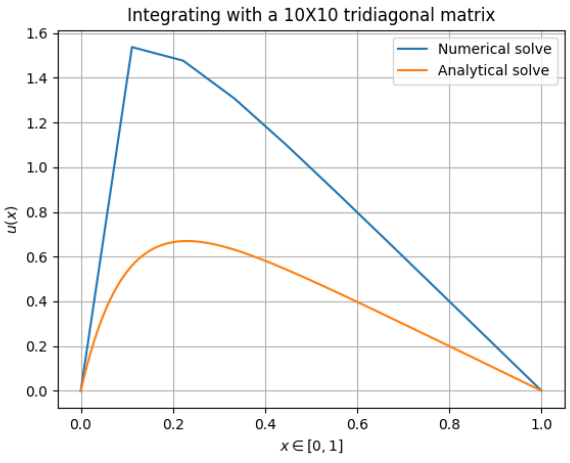

Figure 1. Plot of numerical, and analytical solution, using the general Thomas algorithm with $N = 10$.

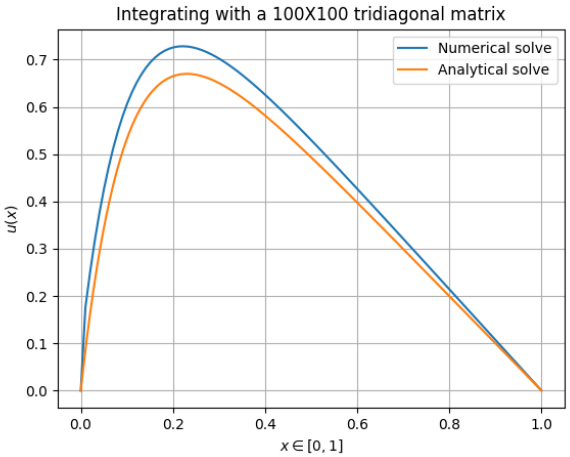

Figure 2. Plot of numerical, and analytical solution, using the general Thomas algorithm with $N = 100$.

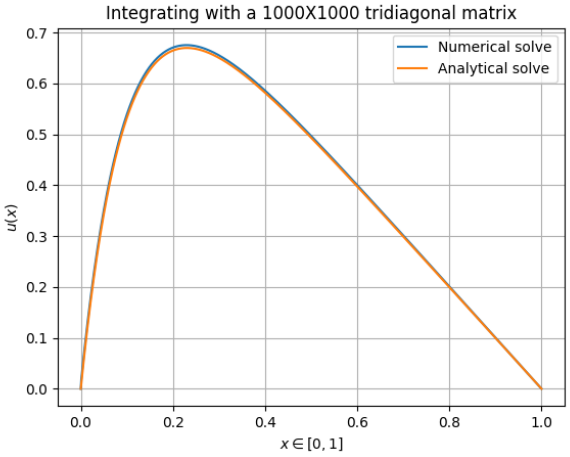

Figure 3. Plot of numerical, and analytical solution, using the general Thomas algorithm with $N = 1000$.

Using "project.py"(A.1), I plotted the numerical and analytical solutions to Possion's equation for a spherically symetrical charge distrobution, using a $N \times N$ matrix. From comparing figure 1 with figure 2, and figure 3, I see that the correspondence with the analytical values decreases as the number of matrix elements increases. It was rather unsuprising that a greater N, and therfore smaller step size h, leads to greater accuracy.

B. Benchmarks of the general and specializeed algorithms

N	Time [s]
1×10^1	$3.596\,200 \times 10^{-4}$
1×10^2	$6.853\,430 \times 10^{-4}$
1×10^3	$6.667\,227 \times 10^{-3}$
1×10^4	$6.290\,169 \times 10^{-2}$
1×10^5	$6.611\,870 \times 10^{-1}$

Table I. Table of time elapsed in seconds on the general Thomas algorithm for "project.py" with corresponding value of N

N	Time [s]
1×10^1	4.5119×10^{-5}
1×10^2	6.6955×10^{-5}
1×10^3	$7.452\,46 \times 10^{-4}$
1×10^4	$7.311\,16 \times 10^{-3}$
1×10^5	$7.114\,03 \times 10^{-2}$

Table II. Table of time elapsed in seconds on the specialized Thomas algorithm for "project_specialized.py" with corresponding value of N

I mesaured the time both "project.py", and "project_specialized.py" used to execute the general and specialized versions of the Thomas algorithm. This was done by saving the time on the CPU-clock directly before and after executing the two sequential loops that constitute the complete algorithm.

Table I shows the time it took for "project.py", and table II shows the time it took for "project_specialized.py". The two affomentioned tables show a decrease in computation time by approximatly one order of magnitude. This was rather suprising given that the number of FLOPs was only halved. However there might be extra overhead associated with retrieving values from arrays that affects the general Thomas algorithm, since it has the elements of the original matrix stored in three arrays, while the specialized Thomas algorithm does not, since any factor containing said elements are precalculated.

There are other possible factors at play, such as seperate processes taking up CPU cycles. Both programs where benchmarked with as few as possible background processes. It was also a posibility that variations in CPU clockspeed might influence the results. A better method of benchmarking would simply have been to take the averages of multiple runs per N-value.

C. Error analysis of the specialized Thomas algorithm

$\log_{10} h$	$\log_{10} \epsilon_{i,max}$
-1	-0.0176522
-2	-0.842497
-3	-1.59133
-4	-2.41863
-5	-3.29336
-6	-4.19081
-7	-5

Table III. Table of $\log_{10} h$, and the corresponding $\log_{10} \epsilon_{i,max}$, where h is the step-size used, and $\epsilon_{i,max}$ is the coresponding maximum of the relative error

To evaluate the accuracy of the specialized Thomas algorithm, I ran "project_specialized.py" for values of $N = 10^i$ where $i = 1, 2, ..., 7$. To compare the numerical and analytical values, I used "erroranalysis.py"(A.4) which reads each data set, and stores a 7×2-matrix in a textfile containing the $\log_{10}$ of the maximum relativ error for each value of N, and the log_{10} of the coresponding step size. The resulting values in table III shows that the maximum relativ error of the numerical solution decreases by approximatly one order of magnitude when the step size was decreased by one order of magnitude.

"erroranalysis.py" did output "RuntimeWarning: divide by zero encountered in log10". It was found to happen for $N = 10 \times 10^5$ Ĺ, and by saving and opening the corresponding array of relativ errors, I found that some of the values were 0. This does not appear to have affected the resulting maximum error.

D. Comparison with LU-decomposition

N	Time [s]
1×10^1	$2.777\,14 \times 10^{-4}$
1×10^2	$1.737\,73 \times 10^{-3}$
1×10^3	$3.077\,32 \times 10^{-2}$
1×10^4	$6.144\,22$
1×10^5	NaN

Table IV. Table of time elapsed in seconds on LU decomposition and solve for "LUdecomp.py" with corresponding value of N

For comparison, I used the LU-decompositon included in the SciPy library. Thdivide by zero in numpy log10e program "LUdecomp.py" which times both the LU factorisation, and solving the resulting system of equations. By comparing table IV with table I and table II, it becomes apparent that the LU-decompositon was significantly slower.

There was some overhead in the function calls, that "project.py" and "project_specialized.py" does not have,

as they are timed inside their respective functions. However, the more probable reason for the slow down was the increase in FLOPs from $8N$ and $4N$ for the general and specialized Thomas-algorithm, to $\frac{2}{3}N^3$ FLOPs for the LU-decomposition[3], since the LU-decomposition was dealing with all $N \times N$ elements of the input matrix.

The final entry on table IV is listed as "not a number" due to "LUdecomp.py" running out of memory and crashing, when atempting to allocate space for all matrix elements when $N = 1 \times 10^5$. This was not suprising, as an $10^5 \times 10^5$-matrix where each element is a 64bit floatingpoint number would require 80GB of random access memory. The system I used only had 16GB. By comparison, "project.py" uses only 2.4MB to represent the same matrix, allowing for much greater values of N, and "project_specialized.py" uses a mere 800kB representing the precalculated $1/d_i^*$ factors.

V. CONCLUSION

The Thomas algorithm proved to be a efficient, and sufficiently precise method for integrating a second order, linear differential equation. The specialized variant of the algorithm proved especially memory efficient, as discussed in IV.B. Both the general, and specialized Thomas algorithms are preferable for the specific usecase in this paper, as they used significantly less memory, and time than a general LU-decomposition solver.

Appendix A: Program files

All code for this report was written in Python 3.6, and the complete set of files can be found at https://github.com/FunkMarvel/CompPhys-Project-1.

1. project.py

https://github.com/FunkMarvel/CompPhys-Project-1/blob/master/project.py

2. project_specialized.py

https://github.com/FunkMarvel/CompPhys-Project-1/blob/master/project_specialized.py

3. data_generator.py

https://github.com/FunkMarvel/CompPhys-Project-1/blob/master/data_generator.py

4. erroranalysis.py

https://github.com/FunkMarvel/CompPhys-Project-1/blob/master/erroranlaysis.py

5. LUdecomp.py

https://github.com/FunkMarvel/CompPhys-Project-1/blob/master/LUdecomp.py

[1] B. N. Datta and Society for Industrial and Applied Mathematics., *Numerical linear algebra and applications* (Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2010) p. 530.

[2] Department of Physics, *Project 1 - Computational Physics I FYS3150/FYS4150*, Tech. Rep. (University of Oslo, 2019).

[3] M. Hjorth-Jensen, *Computational Physics Lectures: Linear Algebra methods*, Tech. Rep. (Department of Physics, University of Oslo, Oslo, 2018).