Solving the one-dimensional Poisson equation numerically

Camilla Marie Baastad Niels Bonten

September 10, 2018

We compare Thomas algorithm to LU decomposition for solving the one-dimensional Poisson equation with Dirichlet boundary conditions numerically. Using a three point formula for the second derivative, we obtain a tridiagonal matrix equation, $Av = d$, where A is an $n \times n$ matrix. Because the main diagonal of A consists only of 2's and the and the diagonal directly below and above consists only of -1's, we are able to reduce the number of FLOPS required by the Thomas algorithm from $\approx 10n$ to $\approx 4n$. In doing so, however, we appear to lose numerical precision for large n. As LU decomposition requires $\approx \frac{2}{3}n^3$, this method is less efficient. Using Thomas algorithm, the matrix may be stored as 3 length n vectors, and so this method consumes far less memory than LU decomposition.

All code and data used in this project can be found at: `https://github.com/CamillaBa/Project_1`.

1 Introduction

The Poisson equation is a differential equation that occurs in several branches of physics, including electromagnetism, thermodynamics and fluid mechanics. In this project, we are solving the one-dimensional Poisson equation with Dirichlet boundary conditions numerically, and comparing the results to a known analytic solution. More precisely, we approximate the second derivative to obtain a set of linear equations which we solve using Thomas algorithm and LU decomposition. The analytic solution is valuable to us since it provides a precise measure on how well these methods work.

We have two aims in mind. One is to reduce the number of floating point operations (FLOPS) in our code, so that it runs as efficiently as possible. The other is to compare speed and memory usage of the Thomas algorithm to those of LU decomposition methods.

Unsurprisingly we find that the more tailored method, Thomas algorithm, beats the more general method of LU decomposition with respect to speed and memory for the particular set of linear equation that we seek to solve. In terms of memory this is achieved since our tridiagonal $n \times n$ matrix can be determined completely by just 3 vectors of length $n-1$, $n-1$ and n respectively. For Thomas algorithm this means that we ideally only need to store roughly $3n$ numbers, whereas for LU decomposition we need $n \times n$. For big n, say 10^5 and up, this task becomes huge for the average computer.

We also consider FLOPS, and Thomas method is the least expensive in terms of CPU power as well. However, it should be pointed out that due to loss of numerical precision, there is is really no point in increasing n further than about 10^4. While it is true that Thomas algorithm beats LU decomposition in terms of FLOPS and memory, LU decomposition still runs in a matter of seconds for a matrix of this size. If you can afford those extra seconds, there is really no need to prefer one to the other.

2 Theory

2.1 The one-dimensional Poisson equation

We consider the one-dimensional Poisson equation with Dirichlet boundary conditions. That is, we are given a function $f\colon [0,1] \to \mathbb{R}$ and seek a function $u\colon [0,1] \to \mathbb{R}$ such that

$$-u''(x) = f(x), \text{ with } u(0) = u(1) = 0. \tag{1}$$

To attack this problem, we do the following. We fix a positive integer n and let $h := 1/(n+1)$. Then we partition $[0,1]$ into $n+1$ subintervals $[x_i, x_{i+1}]$ such that $x_i = ih$ for all $i = 0, \ldots, n+1$. Observe that the length of each interval is given by $h := 1/(n+1)$. Assuming that u is an exact solution to eq. (1), we find an approximation to u which we label v. The function v will be a function $v\colon [0,1] \to \mathbb{R}$ such that v is linear on each interval $[x_i, x_{i+1}]$ and such that $v(x_i) \approx u(x_i)$ for all i.

Our task is to find an ordered set of points $\{v_0, \ldots, v_{n+1}\}$ such that $v_i \approx u(x_i)$. Then $v(x_i) := v_i$ determines the approximation v completely.

2.2 Approximation to $u''(x)$

We now derive an approximation to $u''(x)$ using Taylor expansions. Consider the taylor expansions of u about x_i:

$$u(x_i \pm h) = u(x_i) \pm u'(x_i)h + \frac{u''(x_i)h^2}{2} + \mathcal{O}(h^3)$$

It is seen from adding $u(x_i + h)$ and $u(x_i - h)$ that:

$$u''(x_i) = \frac{u(x_{i+1}) + u(x_{i-1}) - 2u(x_i)}{h^2} + \mathcal{O}(h^2)$$

Assuming that we have discretized u and consider the points $u_i := u(x_i)$, we can define a discrete axpproximation to $u''(x_i)$ as

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

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

We now reformulat eq. (1) in terms of the approximation of $u''(x)$. Let $f_i = f(x_i)$ for $i = 0, \ldots, n+1$. Then eq. (1) becomes:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i, \quad v_0 = v_{n+1} = 0, \quad i = 1, \ldots, n \tag{2}$$

Solving this set of linear equations yields the ordered set $\{v_0, \ldots, v_{n+1}\}$ that we are after. The aim now is to solve this set of linear equations and obtain the approximation v to u.

2.3 The one dimensional Poisson equation as a matrix equation

Consider eq. (2). Multiplying both sides by h^2 yields the equivalent set of linear equations:

$$2v_i - v_{i+1} - v_{i-1} = h^2 f_i, \quad i = 1, \ldots, n$$

Because we assume $v_0 = v_{n+1} = 0$, we can discard the equations that would otherwise be needed to solve for v_0 and v_n. In other words, we may discard the equations corresponding to $i = 1$ and $i = n$.

We want to reformulate the remaining equations ($1 < i < n$) into a matrix equation. We define the $n \times n$ matrix

$$A := \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

and the vectors $v := (v_1, \ldots, v_n)$ and $d := h^2(f_1, \ldots, f_n)$. We have

$$Av = d, \tag{3}$$

as is seen from taking the dot product of the i'th row in A with v.

2.4 An analytical solution to the one dimensional Poisson equation

Assume that $f(x) = 100e^{-10x}$. We now show that $u(x) = 1 - (1 - e^{-10})x - e^{-10x}$ is an analytical solution to eq. (1) by insertion. We have

$$u'(x) = 1 - e^{-10} + 10e^{-10x}$$
$$u''(x) = -100e^{-10x}$$

Hence $-u''(x) = f(x)$.

Remark 2.1. The above solution will be a point of reference for the numerical solution, so that we may verify the validity of the proposed numerical solution. That is, we shall compare the approximation v to u for different values of n.

2.5 Thomas algorithm

A **tridiagonal matrix** is a square matrix whose only nonzero entries are the entries along the main diagonal and the entries along the diagonal directly above and the diagonal directly below the main diagonal.

Let temporarily A be any $n \times n$ tridiagonal matrix with $n \geq 2$ and v and d be two n-dimensional vectors. Note that

$$A = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix}$$

is uniquely determined by three vectors $a = (a_1, \ldots, a_{n-1})$, $b = (b_1, \ldots, b_n)$, and $c = (c_1, \ldots, c_{n-1})$.

We seek to solve an equation of the form $Av = d$. That is, we seek to solve:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_1 & b_2 & c_2 & \ddots & \vdots \\ 0 & a_2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{pmatrix} \tag{4}$$

This can be accomplished by row reducing the matrix:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 & d_1 \\ a_1 & b_2 & c_2 & \ddots & \vdots & \vdots \\ 0 & a_2 & \ddots & \ddots & 0 & d_{n-2} \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} & d_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & b_n & d_n \end{pmatrix} \sim \begin{pmatrix} \beta_1 & c_1 & 0 & \cdots & 0 & \delta_1 \\ 0 & \beta_2 & c_2 & \ddots & \vdots & \vdots \\ \vdots & 0 & \ddots & \ddots & 0 & \delta_{n-2} \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} & \delta_{n-1} \\ 0 & \cdots & \cdots & 0 & \beta_n & \delta_n \end{pmatrix}$$

Now we need to rid of the a_i's one by one. The first row operation needed is then clearly $R_2 \to R_2 - \frac{a_1}{b_1} R_1$. And then we wish to generalize this pattern further. The general algorithm for this row reduction can be described as follows. We inductively update each row accordingly, and one at the time starting from above:

Algorithm 1. (Row reduction)

(i) The initial steps are $\beta_1 := b_1$ and $\delta_1 := d_1$.

(ii) Let $\epsilon_i := a_i/\beta_i$ for $i = 1, \ldots, n-1$. The inductive step is:

$$\beta_i := b_i - \epsilon_{i-1} c_{i-1}, \quad \text{and } \delta_i := d_i - \epsilon_{i-1}\delta_{i-1}, \quad \text{for } i = 2, \ldots, n$$

Remark 2.2. The number of floating point operations needed to run the above algorithm is $7(n-1)$. Indeed, (i) requires no floating point operations and there are $n-1$ steps left. In each step we need to do 2 times a calculation of the form $a - b * c$ and one times a calculation of the form a/b, yielding a total of 7.

Equation (4) is now equivalent to the following set of linear equations in the variables $v_0, \ldots, v_{n+1}$:

$$\begin{aligned} \beta_{i-1} v_{i-1} + c_{i-1} v_i = \delta_i, \quad \text{for } i = 1, \ldots, n-1 \\ \beta_n v_n = \delta_n \end{aligned}$$

The algorithm to solve these equations becomes:

Algorithm 2. (Solve equations)

(i) The initial step is $v_n = \delta_n/\beta_n$.

(ii) The inductive step is:

$$v_i = \frac{\delta_i - c_i v_{i+1}}{\beta_i}, \quad \text{for } i = n-1, ..., 1$$

Remark 2.3. By a counting argument similar to the one in remark 2.2, the number of floating point operations needed for the above algorithm is $1 + 3(n-1)$.

2.6 The tridiagonal matrix equation for $-u''(x) = f(x)$

The special case we seek to solve is:

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{pmatrix} \tag{5}$$

We apply the algorithms described in section 2.5. Note first that by definition $\epsilon_i = -1/b_i$. Algorithm 1 then reduces to the recursive relation:

$$\beta_i = 2 - \frac{1}{\beta_{i-1}}, \quad \beta_1 = 2, \quad \text{for } i = 2, \ldots, n$$

We spot the general pattern by examining small i:

$i:$	1	2	3	4	$\cdots$	n
$\beta_i:$	2	$\frac{3}{2}$	$\frac{4}{3}$	$\frac{5}{4}$	$\cdots$	$\frac{n+1}{n}$

It follows by the definition of ϵ_i that $\epsilon_i = -\frac{i}{i+1}$ for all $i = 1, \ldots, n-1$. As with the general case, solving 5 amounts to the row reduction:

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & d_1 \\ -1 & 2 & -1 & \ddots & \vdots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 & d_{n-2} \\ \vdots & \ddots & \ddots & \ddots & -1 & d_{n-1} \\ 0 & \cdots & 0 & -1 & 2 & d_n \end{pmatrix} \sim \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & \delta_1 \\ 0 & 3/2 & -1 & \ddots & \vdots & \vdots \\ 0 & 0 & 4/3 & \ddots & 0 & \delta_{n-2} \\ \vdots & \ddots & \ddots & \ddots & -1 & \delta_{n-1} \\ 0 & \cdots & 0 & 0 & \frac{n+1}{n} & \delta_n \end{pmatrix}$$

Note that $\epsilon_i = -1/\beta_{i-1}$ and recall that by algorithm 1 that:

$$\delta_i = d_i + \frac{\delta_{i-1}}{\beta_{i-1}}, \quad \text{for } i = 2, \ldots, n,$$

To solve the set of linear equations we modify algorithm 2:

(i) The initial step is $v_n = \delta_n/\beta_n$.

(ii) The inductive step is:

$$v_i = \frac{\delta_i + v_{i+1}}{\beta_i}, \quad \text{for } i = n-1, ..., 1$$

Remark 2.4. Counting the total number of FLOPS in this special algorithm we see that algorithm 1 reduces to $2(n-1)$ FLOPS, and algorithm 2 reduces to $1 + 2(n-1)$ FLOPS. We have a total number of $1 + 4(n-1)$ FLOPS.

2.7 LU decomposition

Let $n \geq 1$, let A be an $n \times n$ matrix, and let d be an n-dimensional colum vector. The equation

$$Av = d,$$

may sometimes be solved for v by means of LU decomposition. An **LU factorization** is a (not necessarily unique) factorization $A = LU$ such that L is a lower triangular matrix and U is an upper triangular matrix. Then solving $Av = d$ is equivalent to solving $LUv = d$. So solving for v can be accomplished by first solving $Ly = d$ for y and then solving $Uv = y$ for v, where by definition $y := Uv$.

In our case we assume A to approximate the second derivatrive u'', so A becomes

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

A sufficient condition for an $n \times n$ matrix A to admit an LU factorization is that $\det A(1:k, 1:k) \neq 0$ for all $k = 1, \ldots, n-1$. [GV12]
We have:

$$\alpha_1 := \begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} = 4 - 1 = 3$$

$$\alpha_2 := \begin{vmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{vmatrix} = 2\alpha_1 + \begin{vmatrix} -1 & -1 \\ 0 & 2 \end{vmatrix} = 2\alpha_1 - 2 > \alpha_1 > 0$$

$$\alpha_3 := \begin{vmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{vmatrix} = 2\alpha_2 + \begin{vmatrix} -1 & -1 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{vmatrix} = 2\alpha_2 - \alpha_1 > \alpha_2 > 0$$

$$\alpha_4 := \begin{vmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{vmatrix} = 2\alpha_3 + \begin{vmatrix} -1 & -1 & 0 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{vmatrix} = 2\alpha_3 - \alpha_2 > \alpha_3 > 0$$

$$\vdots$$

The conclusion is (by induction) that $\alpha_n > 0$ for all $n > 0$. Therefore, the matrix A admits an LU decomposition, and we can use LU decomposition to solve $Av = d$.

Remark 2.5. Using LU decomposition for solving $Av = d$, where A is an $n \times n$ matrix, requires $\approx \frac{2}{3}n^3$ FLOPS. [TB97]

2.8 Relative error and numerical precision

Given a vector u and an approximation v to u, the relative error is

$$\epsilon := \frac{|u - v|}{|u|}.$$

Here $|u|$ is the standard Euclidean norm. That is, if we write $u = (u_1, \ldots, u_n)$, we have

$$|u| := \sqrt{u_1^2 + \cdots + u_n^2}.$$

This is the measure that we will use to determine the quality of our approximation to the analytical solution of $-u''(x) = f(x)$.

The following is taken from [Hjo11]. In a computer, a real number x is represented by a floating point number $\mathrm{fl}(x)$, such that $\mathrm{fl}(x) = x(1 \pm \epsilon_x)$, where $|\epsilon_x| \leq |\epsilon_M|$ is the error of x and ϵ_M is the assigned precision. A challenge when dealing with numerical calculation is numerical error. There are four major error sources. These may roughly be summarized as follows, although these may be discussed in more detail.

(i) Underflow: There is a limitation the smallest number that can be represented.

(ii) Overflow: There is a limitation to the biggest number that can be represented.

(iii) Round off: Binary operations return rounded answers.

(iv) Numerical precision: Loss of significant figures. This happens when you attempt to subtract two numbers r_1 and r_2 such that $r_1 \approx r_2$. If the digits r_1 and r_2 can not be represented accurately due to round off errors, their difference may yield a very wrong result. To illustrate this simply, assume $r_1 = 123$ and $r_2 = 122$ and that you only allow two figures to represent these numbers, that is, $r_1 \approx 1.2 \times 10^2$ and $r_2 \approx 1.2 \times 10^2$. Their difference $r_1 - r_2$ which should be 1 instead returns $12 - 12 = 0$, which is way off.

The situation is that the total error of the numerical calculations will be due to both mathematical error in approximations and numerical error. An example of mathematical error is the approximation for u'', with an error proportional to h^2. A place where numerical error can occur is in Thomas algorithm, where for example algorithm 1 (ii) and algorithm 2 (ii).

3 Results

Figure 1 shows how the numerical approximation fit the exact solution for increasing n. As is seen from the figure, choosing 100 grid points already appears to yield a fairly good fit.

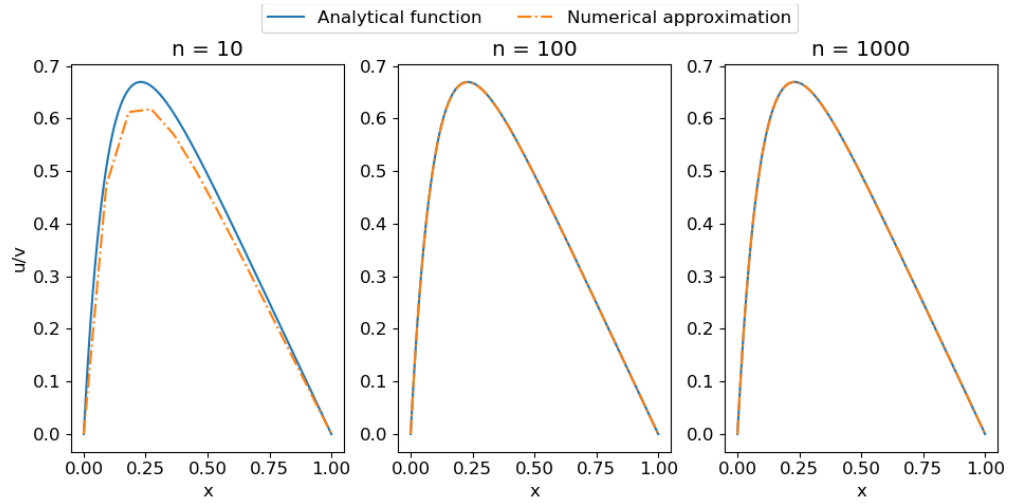

Figure 1: The analytic solution, found in section 2.4, compared to the numerical approximation, using our modified algorithm, for different numbers of grid point. Note that the approximation seems to converge to the exact solution as n increases.

While the plots in fig. 1 were produced using the optimized version of Thomas algorithm, the general algorithm would produces indistinguishable results for these values of n. However, as is seen in fig. 2, the two algorithms starts to differ as n grows larger.

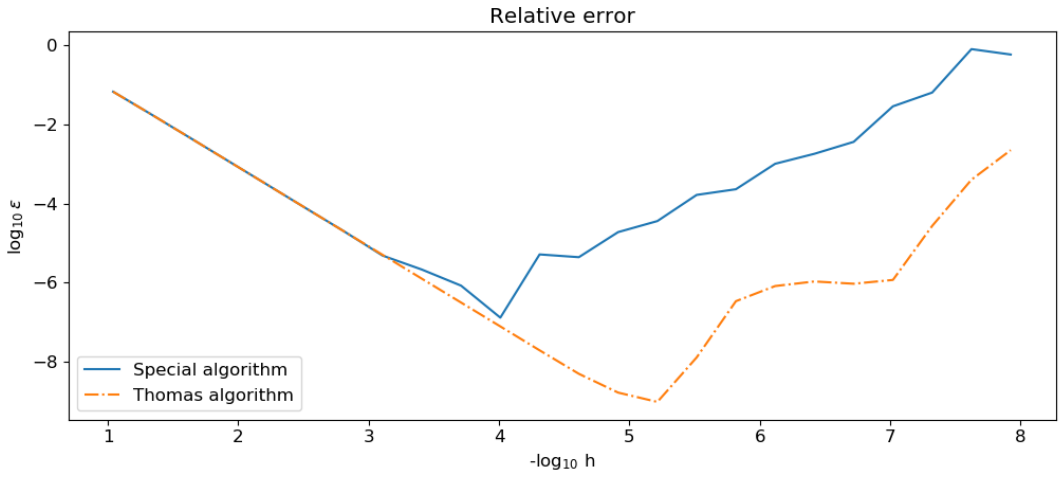

Figure 2: The relative error using Thomas algorithm and the modified version plotted against step size.

As fig. 2 shows, the relative error is smallest when using $\approx 10^4$ grid points with the modified algorithm. Further increasing the number of grid points leads to a larger relative error. This is most likely due to loss of numerical precision, as explained in section 2.8.

The relative error of the general method is about the same as the error of the modified one until $n \approx 10^4$. For larger n however, the error of the general method appears to yield a better approximation.

The relative error of the general Thomas algorithm does not increase until the number of grid points exceeds 10^5, and remains smaller than the error produced by the special method at least until $n \approx 10^8$. In any case, the optimal number of grid points for solving the problem lies somewhere between 10^4 and 10^5.

The elapsed time using the general Thomas algorithm, our optimized version and LU decomposition for several numbers of grid points is shown in table 1.

Table 1: Execution time for the three algorithms for different values of n. The LU decomposition algorithm could not be used for matrices of size $10^5 \times 10^5$ or greater, due to large memory consumption.

n	Execution time [s]		
	Thomas algorithm	Modified algorithm	LU decomposition
10	0.000	0.000	0.000
10^2	0.000	0.000	0.001
10^3	0.000	0.000	0.090
10^4	0.000	0.001	17.541
10^5	0.003	0.002	-
10^6	0.028	0.020	-
10^7	0.284	0.196	-
10^8	2.803	1.984	-

Table 1 lacks results for the LU decomposition method for matrices larger than $10^4 \times 10^4$. If we want to solve Poisson equation numerically using LU decomposition, then there is a firm limitation to the number of time steps n possible. To store a matrix of dimension $10^5 \times 10^5$ corresponding $n = 10^5$, one would need 8×10^{10} bytes = 80 Gb of memory. Already in terms of memory, this is no feasible task for the average computer. But this is not the only department in which LU decomposition performs less than optimal. In terms of computing power LU decomposition is quite costly. As our results show, LU decomposition gives less bang for the buck for our purpose, when compared to the special algorithm derived from Thomas algorithm.

The optimized version of Thomas algorithm seems to spend about 2/3 of the time spent by the general algorithm. This was not unexpected, seeing that the modified version computes about half the amount of FLOPS computed by the general version. The likely explanation that it is not twice as fast is probably that there is more going on under the hood in our program than flops.

4 Conclusion

By carefully inspecting the given problem one may find a more efficient and less memory consuming algorithm. This becomes apparent when comparing Thomas algorithm to LU factorization, which works also for non tridiagonal matrices. By adapting Thomas algorithm to our specific problem, we were able to cut the number of floating point operations required in half, leading to a more efficient algorithm.

One might expect that a smaller step size would lead to a better approximation. As we have seen, this is not always the case. Choosing $n \approx 10^4$ seems to be the ideal number of grid points for solving this

problem. If you surpass this, the memory consumption when using LU decomposition would become to great for the average computer, while the relative error produced by the modified version of Thomas algorithm would begin to increase. In this case, the general Thomas algorithm would be your safest bet.

The reason that the special algorithm performs worse than Thomas algorithm is a result of numerical precision. In order to find exactly what is at fault, it is possible to study the sequence of operations used in the two versions and look for cases where one might subtract approximately equal number in the special algorithm, whereas one avoids it in the general one. This would explain why the relative error for the modified version is greater than that of the general version when n is large. It would be interesting to examine the code and try to figure out exactly why this happens. Specifically, one could look for instances of almost equal numbers being subtracted, for example by devising an if test to see when this difference is close to zero.

References

[GV12] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU Press, 2012.

[Hjo11] Morten Hjorth-Jensen. "Computational physics". In: *Lecture notes* (2011).

[TB97] Lloyd N. Trefethen and David Bau III. *Numerical linear algebra*. Vol. 50. Siam, 1997.