Increasing accuracy and speed in numerical solutions of differential equation

Ove Haugvaldstad, Eirik Gallefoss

September 9, 2019

Abstract

Introduction

There is no denying the influence the computing has had on science. Where we today are at a point where the computer has become almost unanimous with the scientist. In a time where the computer can solve all your problems, one might ask if the old fashioned pen and paper method still necessary? We will argue for the necessity old pen and paper thinking, and demonstrate how pen and paper and computing supplement each other.

For our demonstration we will look at a how to solve a second order differential equation, specifically the general one dimensional Poisson's equation eq. (2). We will show how not thinking about the problem at hand and just computationally brute force the solution can lead to huge performance penalties.

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

Numerical differentiation

A computer can only operate in discrete steps, which means that variables are stored as discrete variables. A discrete variable defined over a particular range, would have a step length h between each value and can not represent any values in between. This means that how well a discrete variable would approximate the continuous variable depends on the size of the step length. The step length h can either be set manually or it can be determined from the start and end point of our range, $h = \frac{x_n - x_0}{n}$. Where n is the number of points we choose to have in our range.

The simplest way to compute the derivate numerically is to use what is called forward difference method eq.(2) or equivalently backward difference method (eq.3). If we include the limit $\lim_{h\to 0}$ we obtain the classic definition of the derivate.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{2}$$

$$f'(x) \approx \frac{f(x-h) + f(x)}{h} \tag{3}$$

Since numerical differentiation always will give an approximation of the derivate, we would like to quantify our error. The error can be derived if we do a taylor series expansion of the $f(x+h)$ term in around x.

$$f(x+h) = f(x) + h'f(x) + \frac{h^2 f''(x)}{2} + \frac{h^3 f'''(x)}{6} + \dots \tag{4}$$

If we next insert this taylor expansion into eq.(4) we get:

$$f'(x) = f'(x) + \frac{hf''(x)}{2} + \frac{h^2 f'''(x)}{6} + \dots \tag{5}$$

Our approximation of the derivate includes $f'(x)$ and some terms which are proportional to $h, h^2, h^3 \dots$ and since h is assumed to be small the h terms would dominate. The error is said to be of the order h.

To get a numerical scheme for the second derivate we would just take the derivate of eq. (2) except for a slight modification. Instead of looking at $f''(x) \approx \frac{f'(x+h)-f'(x)}{h}$ we would use $f''(x) \approx \frac{f'(x)-f'(x-h)}{h}$. it is simple to prove that the two expression are equivalent [1].

$$f''(x) \approx \frac{f(x+h) - f(x) - f(x-h+h) + f(x-h)}{h^2} \tag{6}$$

Then after a bit of a clean up we get an approximation for the second order derivate (eq. (7)).

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{7}$$

Then to quantify the error we proceed as for the first order derivate, by expanding $f(x+h)$ and $f(x-h)$.

$$f(x-h) = f(x) - hf'(x) + \frac{h^2 f''(x)}{2} - \frac{h^3 f'''(x)}{6} \dots \tag{8}$$

Next we substitute the two taylor expansion eq. (8) and eq. (4) into the expression for second order derivate eq. (7).

$$f''(x) \approx f''(x) + \frac{h^2 f^{(4)}(x)}{4!} + \frac{h^4 f^{(6)}(x)}{6!} + \dots \tag{9}$$

Then we see that leading error term is for the second derivate is $\mathcal{O}(h^2)$.

Methods

Building upon the previously described concepts of numerical derivatives, we will now describe how to solve our differential equation eq. (1) numerically by rewriting it as a set of linear equations.

To be explicit, the differential equation we will solve is:

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0$$

The first step is to define the discrete approximation v_i to $u(x)$, with grid points $x_i = ih$ in the range from $x_0 = 0$ to $x_{n+1} = 1$. Choosing the step length to be defined as $h = 1/(n+1)$. Including boundary conditions as $v_0 = 0$ and $v_{n+1} = 0$. The second derivate is approximated according to eq. (7), but rewritten in shorthand notation eq. (10).

$$f_i = -\frac{v_{i-1} - 2v_i + v_{i+1}}{h^2} \quad \text{for } i = 1, 2, 3 \ldots, n \tag{10}$$

To see how eq. (10) can be represented as matrix equation, we will first multiply each side by h^2.

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

Next we represent the v_i's and the g_i's as a vectors,

$$\mathbf{v} = [v_1, v_2, v_3, \ldots, v_n], \quad \mathbf{g} = [g_1, g_2, g_3, \ldots, g_n]$$

Then if we transpose our two vectors we only need to find the $n \times n$ matrix $\mathbf{A}$ and our matrix equation is complete. The matrix $\mathbf{A}$ would in our case looks like this.

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & 0 \\ 0 & -1 & 2 & -1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{bmatrix}$$

It is easy to verify that $\mathbf{A}v = \tilde{\mathbf{g}}$ would give us eq. (10), simply by doing the matrix multiplication. The matrix $\mathbf{A}$ has some particular nice features, primarily it a tridiagonal matrix which means that we can use the efficient Thomas algorithm to solve our linear system of equation, secondly it has constant values along the diagonals, which we'll exploit in our specialized Toeplitz algorithm.

Thomas Algorithm

The Thomas Algorithm is an general algorithm for solving tridiagonal sets of linear equations. The algorithm is quite straight forward to implement and

requires two steps only. We will demonstrate this algorithm with a 4×4 matrix, and our equation is $\mathbf{T} \cdot \mathbf{u} = \mathbf{g}$, see eq. (11).

$$\begin{bmatrix} d_1 & c_1 & 0 & 0 \\ a_1 & d_2 & c_2 & 0 \\ 0 & a_2 & d_3 & c_3 \\ 0 & 0 & a_3 & d_4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} \tag{11}$$

Then the first step is to rewrite our matrix T as upper triangular matrix. Starting with the first row we multiply by it by $\frac{a_1}{d_1}$ and subtract it from the second row.

$$0 + \left(d_2 - \frac{c_1 a_1}{d_1} u_2 \right) + c_2 u_3 = g_2 - g_1 \frac{a_1}{d_1}$$

Next we define $\tilde{d_2} = d_2 - a_1 c_1 / d_1$, $\tilde{g_2} = g_2 - g_1 c_1 / d_1$ and repeat the same proses for the third row. One also might see the general pattern emerging:

$$\tilde{d_i} = d_i - a_{i-1} c_{i-1} / d_{i-1}, \quad \tilde{g_i} = g_i - g_{i-1} c_{i-1} / d_{i-1} \tag{12}$$

After doing this forward sweep the matrix equation does now look like this,

$$\begin{bmatrix} d_1 & c_1 & 0 & 0 \\ 0 & \tilde{d_2} & c_2 & 0 \\ 0 & 0 & \tilde{d_3} & c_3 \\ 0 & 0 & 0 & \tilde{d_4} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} \tilde{g_1} \\ \tilde{g_2} \\ \tilde{g_3} \\ \tilde{g_4} \end{bmatrix}$$

and now we can solve our equation with respect to $\mathbf{u}$ by starting from the bottom row. We can directly read of $u_4 = \tilde{g_4} / \tilde{d_4}$. Working from the bottom and up we get $u_3 = \left(\tilde{g_3} - c_3 u_4 \right) / \tilde{d_3}$. Then we have the following general pattern

$$u_i = \left(\tilde{g_i} - c_i u_{i+1} / \tilde{d_i} \right) \tag{13}$$

Our implementation of the Thomas algorithm, is available through our github[1].

To get an idea of how an algorithm performs, we can count the number of floating point operations per second (FLOPS). When counting FLOPS we only look at the mathematical operations and only count those inside our for-loops. Counting the number of FLOPS for the Thomas algorithm we get $9N$ FLOPS, which is considerably less than the standard LU decomposition which has on the order of N^3 FLOPS. Memory usage is another important consideration to make when choosing algorithms. For instance with the Thomas algorithm we

[1] https://github.com/Ovewh/Computilus/tree/master/Project1/src/linalg.py

only need to store the values along the tridiagonal, since all the other elements are zero. Then we only need tree arrays to store the entire matrix. If we would instead use the general LU decomposition algorithm which requires an $N \times N$ matrix, we would need to store the entire matrix in memory. For instance if we had a $10^5 \times 10^5$ matrix, which means 10^10 matrix elements. To store this matrix in memory when every matrix element is 8 bytes, would require on the order of 10^{11} bytes, about 100 gigabytes to store. An amount of memory clearly beyond any ordinary laptop.

Specialized Algorithm

To create a specialized version of Thomas algorithm for our tridiagonal Toeplitz matrix we will take advantage of the fact that elements along the diagonals are constants. This means that we can pre-calculate the new diagonal elements $\tilde{d}$, reducing the number of FLOPS from $9N$ to $4N$.

Experimental setup

To measure the performance difference between the Thomas, specialized Toeplitz and the general LU-decomposition, we ran each algorithm for different values of n ranging by power of 10 from 10 to 10^7. In order to achieve more accurate timings, we ran each algorithm 10 times for each n and then taking the average time. We also analysed the numerical error and how the error varied with step size, by computing the maximum relative error.

Results

Algorithm run time

From the summary (table 1) we immediately see that the TDCMA is slower than TDMA and the LU is considerably slower than both TDMA and TDCMA.

To see how the algorithm time for our different methods scales with N we divide all timings with N (table 2). Both TDMA and TDCMA run times are of the same order, as was expected from the counting of flops. The times for LU show an increase of two orders of magnitude for each magnitude increase in n. This is consistent with our expectations of the LU algorithm time being proportional to n^3.

N	TDMA	TDCMA	LU
10	1.380e-05	1.800e-05	2.350e-04
100	2.530e-04	1.380e-04	8.270e-02
1000	1.860e-03	9.380e-04	7.610e+01
10000	1.420e-02	8.010e-03	
100000	2.020e-01	1.220e-01	
1000000	1.700e+00	7.390e-01	
10000000	1.460e+01	7.150e+00	

Table 1: Average (20 runs) algorithm run times in seconds.

N	TDMA	TDCMA	LU
10	1.380e-06	1.800e-06	2.350e-05
100	2.530e-06	1.380e-06	8.270e-04
1000	1.860e-06	9.380e-07	7.610e-02
10000	1.420e-06	8.010e-07	
100000	2.020e-06	1.220e-06	
1000000	1.700e-06	7.390e-07	
10000000	1.460e-06	7.150e-07	

Table 2: Algorithm times divided by n.

Comparing the run times of TDMA and TDCMA (table 3) we see they are the same order of magnitude. Theoretically we would expect TDCMA to be $\frac{9FLOPS}{4FLOPS} \approx 2.25$ times as fast as TDMA. When counting FLOPS we did not consider memory operations, so getting as these values close to 2.25 is considered to be in agreement with the theory. We also see that solving the problem by using LU decomposition with N = 1000 takes 80000 times as long as with TDCMA.

N	TDMA/TDCMA	LU/TDCMA
10	7.667e-01	1.306e+01
100	1.833e+00	5.993e+02
1000	1.983e+00	8.113e+04
10000	1.773e+00	
100000	1.656e+00	
1000000	2.300e+00	
10000000	2.042e+00	

Table 3: Run times of algorithms compared to TDCMA.

Error in numerical approximation

Figure 1 shows that we get quite good agreement with the analytic solution when N goes to 1000. To investigate further we look at the relative error $\epsilon_i = \left|\frac{u_i - v_i}{u_i}\right|$ in fig. 2. When increasing N (decreasing the step size h) up to 10^6 the relative error decreases proportionally, until N = 10^7, which might be a sign that we are starting to experience some round off errors.

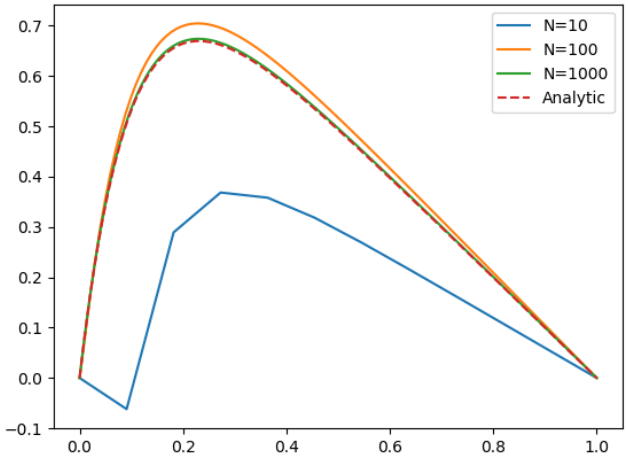

Figure 1: Comparison of analytic solution and numerical approximations with TDMA.

Summary

By understanding the problem we were able to simplify the calculations leading to speedups of the order 10^5 and the ability to achieve significantly higher numerical precision by reducing the memory needed for the calculations, thus allowing us to use smaller stepsizes. We expect that increasing the step size beyond 10^-7 will lead to round off errors and increases in the relative error, but were not able to confirm this.

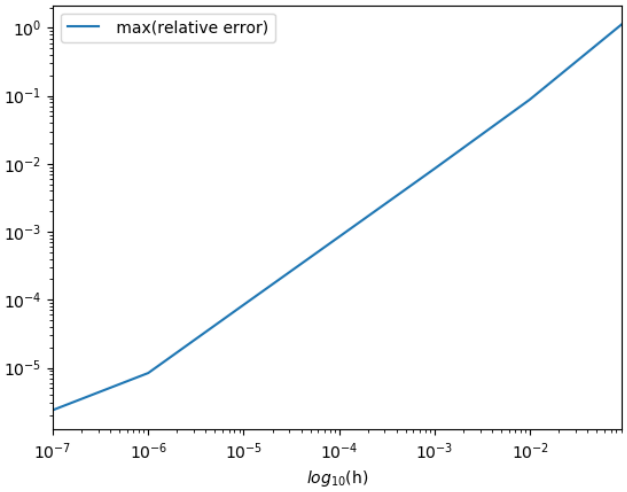

Figure 2: Maximum relative error between analytical and TDMA solution.

References

1. Scott, B. M. *Second derivative formula derivation* https://math.stackexchange.com/q/210269. (accessed: 05.09.2019).
2. Lee, W. T. *Tridiagonal Matrices: Thomas Algorithm* http://www.industrial-maths.com/ms6021_thomas.pdf. (accessed: 07.09.2019).