Project 1

Ove Haugvaldstad, Eirik Gallefoss

September 7, 2019

Abstract

Introduction

There is no denying the influence the computer has had on science. Computing has rendered scientist able to unravel the deeper mysteries of nature. Today for instance a scientist can explore the violent nature of a hurricane while at the same time enjoying a cup of coffee. With current rate in increase of computational performance, we are diminishing the computational limit. Still, to do computational problem solving both efficiently and with high precision requires both an deep understanding of the problem at hand, but also the inner workings of a computer.

Our aim is to demonstrate the benefits of understanding the problem at hand and to apply that knowledge to solve the problem in an efficient way. For our demonstration with we will look at a how to solve a second order differential equation, specifically the general one dimmensional Poisson's equation eq. (2).

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

Numerical differentiation

Computers 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 based on the start and end point of our particular 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:

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

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 $f''(x) \approx \frac{f'(x) - f'(x-h)}{h}$, which are equivalent to each other [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 $f(x-h)$.

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

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.

Explicitly, we will solve the differential equation:

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

We will define the discrete approximation to $u(x)$ as v_i with grid points $x_i = ih$ in the range from $x_0 = 0$ to $x_{n+1} = 1$, and the step length is defined as $h = 1/(n+1)$. The boundary conditions is $v_0 = 0$ and $v_{n+1} = 0$. The second derivate we approximate according to eq. (7) and also introducing the short-hand notation we get eq. (10).

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

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

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

Next we represent the v_i's and the $\tilde{g}_i$'s as a vectors,

$$\mathbf{v} = [v_1, v_2, v_3, \ldots, v_n], \quad \tilde{\mathbf{g}} = [\tilde{g}_1, , \tilde{g}_2, \tilde{g}_3, \ldots, \tilde{g}_n]$$

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 look 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 later.

Thomas Algorithm

The Thomas Algorithm quite straight forward to implement and based on the more general LU decomposition. The full implementation of the Thomas algorithm can be found on our github[1] here we will just outline the general idea.

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

Similarly to the LU decomposition, the idea is to decompose our matrix $\mathbf{A}$ into an upper triangular matrix $\mathbf{U}$ and a lower triangular matrix $\mathbf{L}$. Then we can solve the system of linear equations by setting $\mathbf{U}\mathbf{v} = \tilde{\mathbf{g}}$ and then first solving $\mathbf{L}\tilde{\mathbf{g}} = \mathbf{g}$ for $\tilde{\mathbf{g}}$ and then $\mathbf{U}\mathbf{v} = \tilde{\mathbf{g}}$ for $\mathbf{v}$ [2]. The Thomas algorithm consist of two steps. The first is where both we both do the decomposition and solve $\mathbf{L}\tilde{\mathbf{g}} = \mathbf{g}$ for $\tilde{\mathbf{g}}$ all in one downward sweep. The second step we solve $\mathbf{U}\mathbf{v} = \tilde{\mathbf{g}}$ for $\mathbf{v}$ with a upward sweep.

Results

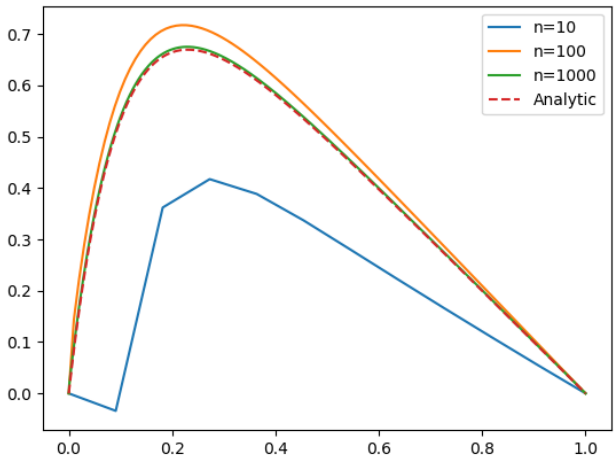

Figure 1: Comparison of analytic solution and numerical approximations

log_{10}(h)	max(relative error)
-1.04	2.82e-02
-2.00	-1.50e-01
-3.00	-2.21e-02
-4.00	-3.24e-03
-5.00	-4.33e-04
-6.00	-5.43e-05

Table 1: Maximum relative error between analytic and numeric solution.

After running the thomas and toeplitz algorithm on matrixes from 10x10 to 1000000 x 1000000, and the lower/upper decomposition + backward subtitution

(LU) on matrixes from 10x10 to 1000x1000 ten times and taking the average we got the results in table 3.

n	thomas	toeplitz	lu
10	2.560e-05	2.670e-05	3.580e-04
100	2.550e-04	2.160e-04	1.450e-01
1000	4.090e-03	1.220e-03	1.230e+02
10000	2.600e-02	1.170e-02	
100000	2.590e-01	1.220e-01	
1000000	2.680e+00	1.280e+00	

Table 2: Summary of algorithm times.

To see how the algorithm time for our different methods scales with n we divide all timings with n (table 3). Both the thomas and toeplitz algorithm times (normalized by n) are of the same order, as was expected from the counting of flops. The times for LU show an increase of to 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	thomas	toeplitz	lu
10	2.560e-06	2.670e-06	3.580e-05
100	2.550e-06	2.160e-06	1.450e-03
1000	4.090e-06	1.220e-06	1.230e-01
10000	2.600e-06	1.170e-06	
100000	2.590e-06	1.220e-06	
1000000	2.680e-06	1.280e-06	

Table 3: Algorithm times divided by n.

Comparing the algorithm times of thomas and toeplitz (table 4) we see they are the same order of magnitude. Theoretically we would expect the toeplitz algorithm to be $\frac{9FLOPS}{4FLOPS} \approx 2.25$ times as fast as toeplitz, and our results for larger values of n are quite close.

n	thomas/toeplitz	lu/toeplitz
10	9.588e-01	1.341e+01
100	1.181e+00	6.713e+02
1000	3.352e+00	1.008e+05
10000	2.222e+00	
100000	2.123e+00	
1000000	2.094e+00	

Table 4: Algorithm times of thomas divided by that of toeplitz.

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).