Fys3150 - Project 1

Sakarias Frette
Anders Vestengen
William Hirst

September 10, 2020

Abstract

Almost every numerical calculation contains error. Most of the time it is therefore not a question of eliminating the error, but controlling it. Up until now, we have increased the accuracy of a numerical solution by increasing the amount of time-steps. In this rapport we will see that it is not always a matter of increasing the iterations, but a question of the right amount. We will attempt to solve a differential equation using three different methods. When doing so we will be monitoring the accuracy and computation time of each method. From the result we concluded that the ideal number of time steps was $n = 10^6$ and that the fastest way of solving the equation was by taking advantage of its non-singular and symmetric nature.

1 Introduction

Most systems can be described with a set of differential equations, and in order to solve those problems efficiently we use programming. However, such a method creates a problem. Differential equations contain either an integral and/or a derivative, and mathematically they may require us to store and evaluate numbers which are infinitely long (irrational numbers), anything infinite is impossible because the computer memory is finite. We represent them as accurately as we can, but we end up creating tiny errors when the computer eventually has to truncate these numbers. These considerations are what we will be exploring in this report.
We will try to solve a differential equation with three different methods; assuming we know nothing about the matrices, taking advantage of the

qualities of the matrices and by LU-decomposition. We will then examine the amount of computational power and time each method needs to solve the equation. We will also briefly examine the error of some of the methods.

2 Method

2.1 The general case

We have the differential equation

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

where x is defined with $x \in [0,1]$ and the boundary conditions are given as $u(0) = u(1) = 0$. We can approximate this differential equation as

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

using finite differentiation, for $i = 1, ..., n$.
We will first look into the case where $i = 1$ and $-h^2 f_i = b_i$. We get the following equation

$$u_2 + u_0 - 2u_1 = b_1 \tag{3}$$

Similarly we get with $i = 2, .., n-1$

$$u_3 + u_1 - 2u_2 = b_2 \tag{4}$$

$$\vdots$$

$$u_n + u_{n-2} - 2u_{n-1} = b_{n-1} \tag{5}$$

We see here that this can be rewritten as a matrix equation $A\vec{v} = \vec{b}$, where

$$\vec{v}^T = [v_1, v_2, ..., v_{n-1}] \tag{6}$$

$$\vec{b}^T = [b_1, b_2, ..., b_{n-1}] \tag{7}$$

and

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

2.2 Specialized case

Now, the C++ pack Armadillo has a solver for inverting matrices to find, what in our case is $\vec{v}$. But inverting matrices, especially large ones take a lot of computing power and memory, and so if we know how the matrix will look, as we do in this case, we can row reduce the matrix to find a set of vectors used to find $\vec{v}$. We start by calling the diagonal elements for $d_1, d_2, ..., d_n$ and the elements parallel to the diagonal for $e_1, e_2, ..., e_n$. Our matrix will then look like this:

$$A = \begin{pmatrix} d_1 & e_1 & 0 & \cdots & 0 \\ e_1 & d_2 & e_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots & \vdots \\ 0 & \cdots & e_{n-1} & d_{n-1} & e_n \\ 0 & \cdots & 0 & e_n & d_n \end{pmatrix}$$

We will now rowreduce matrix Av=b. We start by subtracting row 1 by row two times e_1/d_1. Our new matrix now looks like this:

$$\begin{pmatrix} d_1 & e_1 & 0 & \cdots & 0 & b_1 \\ 0 & d_2 - \frac{e_1^2}{d_1} & e_2 & \cdots & 0 & b_2 - b_1\frac{e_1}{d_1} \\ \vdots & \vdots & \ddots & \cdots & \vdots & \vdots \\ 0 & \cdots & e_{n-1} & d_{n-1} & e_n & b_{n-1} \\ 0 & \cdots & 0 & e_n & d_n & b_n \end{pmatrix}$$

This gives us two new numbers $\tilde{d_2} = d_2 - \frac{e_1^2}{d_1}$ and $\tilde{b}_2 = b_2 - \frac{b_1 e_1}{d_1}$. Now we could do this for the next rows in the matrix, but we see a pattern that repeats, which is this general relation:

$$\tilde{d_i} = d_i - \frac{e_{i-1}^2}{\tilde{d_{n-1}}} \tag{8}$$

and

$$\tilde{b_i} = b_i - \frac{e_{i-1}\tilde{b_{i-1}}}{\tilde{d_{n-1}}} \tag{9}$$

This is called forward substitution. We now have this matrix equation

$$\begin{pmatrix} d_1 & e_1 & 0 & \cdots & 0 \\ 0 & \tilde{d_2} & e_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \cdots & \vdots \\ 0 & \cdots & 0 & \tilde{d_{n-1}} & e_n \\ 0 & \cdots & 0 & 0 & \tilde{d_n} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \tilde{b_2} \\ \vdots \\ \vdots \\ b_n \end{pmatrix} \tag{10}$$

From this we can see that

$$\tilde{d_n} v_n = \tilde{b_n}$$

and

$$\tilde{d_{n-1}} v_{n-1} + e_{n-1} v_n = \tilde{b_{n-1}}$$

This gives the general relation

$$v_i = \frac{\left(\tilde{b_i} - e_i v_{i+1}\right)}{\tilde{d_i}} \tag{11}$$

This solution is called backward substitution.

2.3 LU-decomposition

Another way to solve the matrix equation above is by using LU-decomposition. We start with $A\vec{v} = \vec{b}$, where as above, A is a matrix and $\vec{v}$ and $\vec{b}$ are vectors. Both A and $\vec{b}$ are known. We can decompose A into two new matrices L and U, where L is a lower triangular matrix and U is an upper triangular matrix. Now, since $A = LU$ we get the new equation

$$LU\vec{v} = \vec{b} \tag{12}$$

Now, we need to find $\vec{v}$ so we can solve $U\vec{v} = \vec{y}$. We find $\vec{y}$ by solving:

$$L\vec{y} = \vec{b} \tag{13}$$

This equation is easy to solve, as L and $\vec{b}$ are known and L is lower triangular. When $\vec{y}$ is found, we find $\vec{v}$ by solving:

$$U\vec{v} = \vec{y} \tag{14}$$

Now that we have both U and $\vec{y}$, we can solve and find $\vec{v}$.

2.4 Errors

In our case, we can test the accuracy of or numerical solution (v_i) against the analytical solution (u_i) to the differential equation. To measure this, we calculate the relative error, which we find using this formula:

$$\epsilon_i = \log_{10}\left(\mid \frac{v_i - u_i}{u_i} \mid \right) \tag{15}$$

We could compare the relative error of the entire solution, but to get an overview of our precision we use only the largest error for each value of n-iterations. We plot this against $\log_{10} h$, where $h = \frac{1}{n+2}$. This will show us the error as a function of step size, so that we can see how the step size affects our model.

3 Implementation

3.1 General solution of the differential equation

Since the structure of the implementation was laid out rigidly, we chose to have the input data be command-line arguments. Then we used for-loops to encapsulate the process of declaring and populating the A matrix, b and x vectors. This allows the elements of A to be different, and is therefore a more general solution. Finally we use Armadillo to solve the linear-algebra problem. Armadillos solve-function will analyse the provided matrix and choose the most efficient way to solve the equation. Since this is a boundary problem we iterated through $x = (h, 1-h)$, instead of the whole length, $x = (0, 1)$. Since we know the first and last part of the solution, we add these after the solver has finished.

3.2 Spesific solution of the differential equation

Using the math from the method section about the special case we design our algorithm like this. Note that the dynamic allocation of memory is only needed if one chooses not to use a package like Armadillo. The general algorithm looks like this.

Initialization:;
Set n, allocate memory for vectors $\vec{v}$,$\vec{b}$,$\tilde{b}$,$\tilde{d}$,$\vec{d}$,$\vec{e}$;
Forward part;
for *i = 1, n-1* **do**
  update $\tilde{b_i}$, $\tilde{d_i}$
end
Backward part;
$u_{n-1} = \tilde{b_{n-1}} / \tilde{d_{n-1}}$;
for *i = n-2, 1* **do**
  update u_i
end
Write to file

Algorithm 1: Special case algorithm

3.3 Relative error

As explained in the method section, we design an algorithm to find the maximum value of relative error for each n.

Initialization:;
Set vectors relativeerror, maxvalueerror, stepsize;
Set exponent;
for *p = 1, exponent* **do**
  Set $n = 10^{exponent}$;
  Set $h = \frac{1}{n+1}$;
  for *i = 0, n* **do**
    Update *relativeerror*;
  end
  Find $\log_{10}(relativeerror)$;
  Find $\log_{10}(h)$;
  Update maxvalueerror;
  Update stepsize;
end
Write to file

Algorithm 2: Relative error algorithm

3.4 Solution using LU-decomposition

To solve the the equation using LU-decomposition we apply the method described in the LU-section above. We first create A and $\vec{b}$ using for-loops (as previously). The we use Armadillos LU-decomposition function, splitting matrix A into LU. Then we use Armadillos solve function (as in the general case). First solving

$$L\vec{y} = \vec{b}$$

with solve. Then solving

$$U\vec{x} = \vec{y}$$

Throughout the project we are able to compare our numerical approximation to an analytical solution, and we see that for an increased number of steps the approximation improves in accuracy.

3.5 FLOPS

When looking at the FLOPS for solving the general solution algorithm we estimate the amount of Floating Point Operations we compute each cycle, and then we look at how these scale. So for most of this project we use a forward and a backward substitution and either an LU-decomposition or Gaussian elimination, both given by the lecture notes [1] [2].

Forward substitution

$= \mathrm{d}_i - \frac{e_{i-1}^2}{\tilde{d}_{i-1}} = 3n \;\; FLOPS$

$= \mathrm{g}_i - e_{i-1}\frac{-\tilde{b}_{i_i}}{\tilde{d}_{i-1}} = 3n \;\; FLOPS$

Backward substitution

$\mathrm{u}_i = \frac{(\tilde{b}_i - e_i - u_{i+1})}{\tilde{d}_i} = 3n \;\; FLOPS$

Total (forward/backward)= 9n FLOPS

LU-decomposition = $2\frac{}{3n^3 FLOPS}$
Gaussian Elimination = $\mathrm{n}^3 FLOPS$

4 Results

We tested several methods for solving the matrix equation. First we tried solving a general matrix with Armadillos solve, and got the following solution for $n = 10, 100, 1000$:

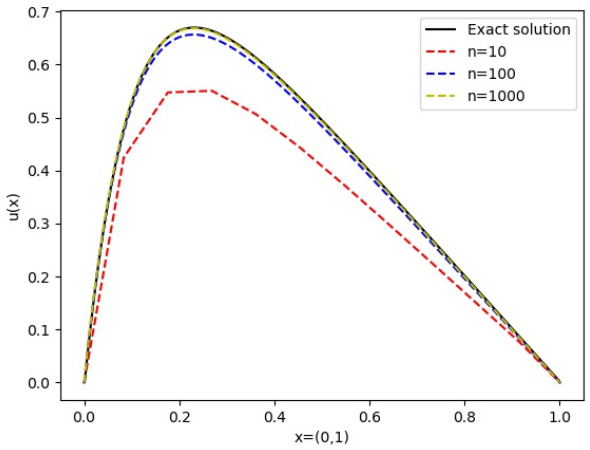

Figure 1: Inverting of matrix with $n = 10, 100, 1000$

From the plot we can see that the methods accuracy increases with increasing N. When N=1000 the numerical solution is basically identical to analytic solution.
We then tried with the specific solution, and got the following result:

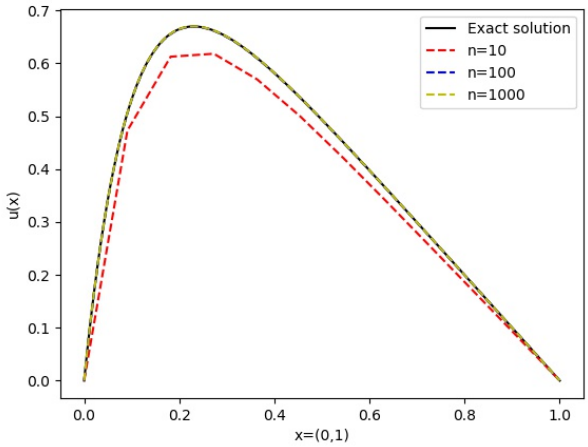

Figure 2: Special solution with $n = 10, 100, 1000$

The plot above seems to be more accurate than the previous solution for all values of N.
We then plot the relative error between the analytic solution and the solution for our specific case (2) as a function of N. We get the following result:

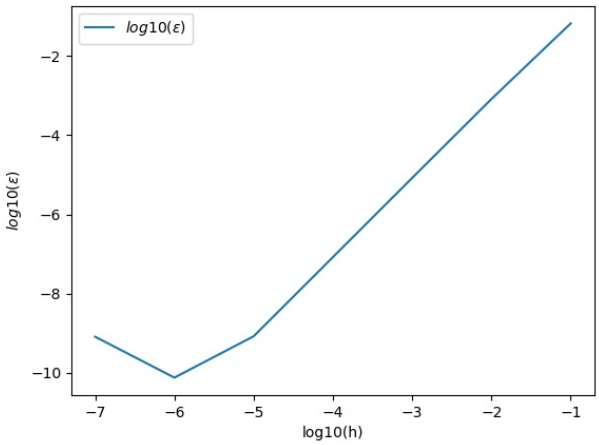

Figure 3: Relative error for specific solution for $n = 10^k, k \in [1, 2, 3, 4, 5, 6, 7]$

From the plot we can see that the relative error decreases linearly with increasing h up until N=10^5. At this point the relative error decreases at a

lower rate, and from N=10^6 to N=10^7 the relative error increases.
With LU-decomposition we get the following result:

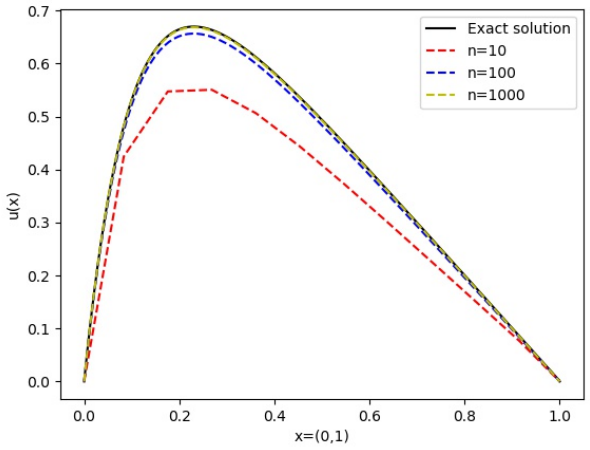

Figure 4: LU-decomp $n = 10, 100, 1000$

We can see from the plot that this method also gives an accurate solution when N is increased.

Finally we measured the time it took to solve the equations for the special and general method. The measured times were as followed:

N	10	100	10^3	10^4
General	1e-04s	9.7e-05s	6.2e-03s	4e-01s
Special case	2.4e-05s	1e-05s	6.1e-05s	5.9e-04s

From the table we can see that for lower N-values, the time it took to calculate the solution was relatively equal. When N is increased further, the special solution is faster than the general.

5 Discussion and analysis

5.1 Time spent on solution

From the results we can see that the accuracy of the models are relatively similar, at least when we increase N. By studying the plots, it does seem

like the specific solution is slightly more accurate than the general. This could be attributed to the fact that the general solution has more FLOPS, and could mean more computational errors. But, when N is increased we can see that they all converge towards the analytic solution. Therefore, we must look elsewhere to compare the models. And that is why we chose to measure the computed time for all the calculations.

The above chart is a run-time comparison between the two different cases of matrix solvers. Initially the two are very similar, we equate this to the fact that our general solution uses Armadillo, which has several built in optimizations that we assume somewhat obfuscates the results. However, this benefit seems to go away for much larger matrices. For $n > 10^4$ we encounter significant slowdown, while our special-case solution could compute for vectors up to a power of at least $n = 10^8$ before being killed by the compiler for exceeding ram-limitations.

5.2 Increasing error for decreasing h

From the plot of the relative error we discover something interesting. The relative error decreases when we increase the amount of time steps, which is what we would expect. But, when the amount of time-steps is increased from 10^6 to 10^7 the relative error starts increasing. This may look counter intuitive. Generally the accuracy of a numerical model increases when the amount of time-steps is increased. From our model we can see that after a certain point, this is not the case. Why is that so?

As we mentioned in the beginning of the report, a computational model is prone to certain unavoidable errors. Truncation-errors is one of those errors. For most calculations the truncation-error is not relevant on account of its error being relatively small. But, when we increase the amount of time-steps we also decrease h, meaning that we decrease the amount of change from one time-step to the next ($\Delta v = v_i - v_{i-1}$). Normally this means that the smaller the Δv, the more accurate the model. But, when Δv decreases to a certain limit, the truncation-error becomes relevant. This is why the relative error increases for N=10^5. The truncation error becomes large enough (compared to Δv), and will therefore contribute to a small error for each time-step calculation, and with enough time-steps, will result in a considerable error.

6 Conclusion

As shown in the results, our accuracy is limited by the computer having to truncate its numerical result every cycle. And so we find that for any meaningful solution, we must contend with a time-step no smaller than $h = 10^{-}6$. From comparing the time spent calculating with the different methods, we also discovered that a solution tailored for a specific equation can be faster. We contribute the decrease in time to fewer FLOPS per cycle.

References

[1] Morten Hjort Jensen. *Håndskrevne notater 27.Aug*. URL: https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/HandWrittenNotes/NotesAugust27.pdf. (accessed: 09.09.2020).

[2] Morten Hjort Jensen. *Håndskrevne notater 3.Sept*. URL: https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/HandWrittenNotes/NotesSeptember3.pdf. (accessed: 09.09.2020).