University of Oslo

Project 1

FYS3150

Bendik Dalen and Ulrik Seip
9-10-2018

Introduction

In this project the aim is to explore different numerical methods for lowering computation times and memory loads while maintaining precision. The Poisson equation is solved with Dirichlet boundary conditions by rewriting it as a set of linear equations, and then the aforementioned exploration of numerical methods is applied on these linear equations.

Theory

The Poisson equation

The following equation will be solved, which is a general representation of the Poisson equation with Dirichlet boundary conditions:

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0, \qquad h = \frac{1}{n+1}$$

Mathematicians hate this simple trick

With a 2nd order Euler approximation a differential equation can be converted into a tridiagonal matrix.

$$f_i = -\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2}$$

$$\tilde{b}_i = h^2 f_i = -v_{i+1} - v_{i-1} + 2v_i$$

$$b = \left[\tilde{b}_1, \tilde{b}_2 \dots \tilde{b}_n\right]$$

This gives is the following matrix:

$$A = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & 0 & \dots & \dots \\ 0 & a_2 & b_3 & c_3 & \dots & \dots \\ \dots & 0 & \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & a_{n-2} & b_{n-1} & c_{n-1} \\ \dots & \dots & \dots & 0 & a_{n-1} & b_n \end{bmatrix}$$

The resulting matrix can be represented by the three vectors a, b and c, which is beneficial if we want to store the matrix on our computer. This way the memory requirement would be $64 \cdot 3n$ bits rather than $64 \cdot n^2$ bits. Reducing the required ram storage opens the possibility of $n < 1e9$ rather than $n < 1e5$ approximately on a standard home computer.

Because of the nature of the diagonal matrix we can now use the simple following algorithm solve for the second derivative:

With the intent of performing $rref(A)$ we first remove the a's with the following row operations on matrix A with b_i' as the modified b_i:

$$II - \frac{a_1}{b_1} I, \qquad III - \frac{a_2}{b_2'} II, \qquad IV - \frac{a_3}{b_3'} III, \dots$$

Likewise, we substitute back to remove the c's

$$I - \frac{c_1}{b_2'} II, \qquad II - \frac{c_2}{b_3'} III, \qquad III - \frac{c_3}{b_4'} IV, \dots$$

The resulting matrix then looks like this:

$$\begin{bmatrix} b_1{}' & 0 & \dots & \dots & \dots & \tilde{b}_1{}' \\ 0 & b_2{}' & 0 & \dots & \dots & \tilde{b}_2{}' \\ \dots & 0 & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & b_{n-1}{}' & 0 & \tilde{b}_{n-1}{}' \\ \dots & \dots & \dots & 0 & b_n{}' & \tilde{b}_n{}' \end{bmatrix}$$

With the final equations for b_i' and $\tilde{b}_i'$ looking like this:

$$b_i' = b_i - \frac{a_{i-1}}{b_i} c_{i-1}$$

And

$$\tilde{b}_i' = \tilde{b}_i - \frac{a_{i-1}}{b_i} \tilde{b}_{i-1}{}'$$

We then use backwards substitution to obtain the solution to the original equation using $k_n = \tilde{b}_n{}'$:

$$k_{i-1} = \tilde{b}_{i-1}' - \frac{c_{i-1} k_i}{b_{i-1}{}'}$$

Counting the flops

The most important factor in calculating how long it should take to run our program, for a given n, is the amount of floating point operations (flops) per loop of the algorithm. Therefore, we want to optimise our algorithm to perform as few flops as possible.

In the proposed algorithm we have the following for the forward substitution followed by the backwards substitution:

$$b_i' = b_i - \frac{a_{i-1}}{b_{i-1}{}'} c_{i-1}, \qquad 3\ flops$$

$$\tilde{b}_i' = \tilde{b}_i - \frac{a_{i-1}}{b_{i-1}} \tilde{b}_{i-1}', \qquad 3\ flops$$

$$k_i = \tilde{b}_i' - \frac{c_i k_{i+1}}{b_i{}'}, \qquad 3\ flops$$

We can reduce this iteration from 9 flops to 8 flops by solving $\frac{a_{i-1}}{b_{i-1}{}'}$ first, because its is used twice. Solving it once means we do just that, solving it once, and so we save one flop.

Specific solution

In our case we don't just want to solve a general 2nd order differential equation, but also a specific one. We will follow the same modus operandi, but wherever possible we will cut corners to save computing time.

Firstly, lets set up the matrix:

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

Then, lets plug this into the general equations and eliminate anything superfluous:

$$b_i' = 2 - \frac{-1}{b_{i-1}}(-1) \Rightarrow 2 - \frac{1}{b_{i-1}}, \qquad 2flops$$

$$\tilde{b}_i' = \tilde{b}_\iota - \frac{-1}{b_{i-1}}\tilde{b'}_{i-1} \Rightarrow \tilde{b}_\iota + \frac{\tilde{b'}_{i-1}}{b_{i-1}}, \qquad 2\ flops$$

$$k_i = \tilde{b}_i' - \frac{(-1)k_{i+1}}{b_i'} \Rightarrow \tilde{b}_i' + \frac{k_{i+1}}{b_i'}, \qquad 3\ flops$$

As we can see our algorithm is reduced from 8 to 7 flops per iteration. Considering there are other processes that also affect the total runtime of the program this isn't too much of a difference when we are limited by the memory of normal laptops, but there should still be a detectable difference for higher numbers of iterations.

Error

Because a 2nd order Euler approximation is used we expect to find an error no greater than the third link in the Euler equation chain. If we call the error ϵ, this means that $\epsilon \propto n^2$. However, we do have another possible source of error, and that is the computational error. When performing several million loops it is quite possible to lose some accuracy due to round off errors, in the sense that a float only stores so many digits. If n becomes too large the errors may stack up, resulting in potentially sporadic and catastrophic errors. To explore this, we can run the algorithms above for as large n's as possible. Numerical errors are hard to predict, but $n \approx 1e8$ should produce a non-trivial numerical error. This means we will need to strike a balance between mathematical accuracy and numerical accuracy to obtain the lowest possible relative accuracy.

The relative error is expressed:

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

Where v_i is the computed result, and u_i is the analytical result.

Something about LU-decomposition

If we don't have a simple matrix that can be broken down into a couple of vectors we can perform an LU decomposition, L and U meaning lower and upper diagonal matrices respectively. The LU decomposition is a way to effectively row reduce a matrix A by, as the name suggests, decomposing it into two matrices $A = LU$. Exactly how such a decomposition is performed is not that important because there are applicable functions including in basic packages of most useful programming languages, but once L and U have been obtained we have to do the remainder as a loop for solving the equation that created the matrix.

$$\tilde{b}_i' = \tilde{b}_i - \tilde{b}_{i-1}' L_{i,i-1}$$

As before we now find the solution using backwards substitution:

$$k_i = \tilde{b}_i' - \frac{U_{i,i+1}k_{i+1}}{U_{i,i}}$$

Results

The basic equation

We observe a clear improvement in the accuracy of the approximation with an increase in n.

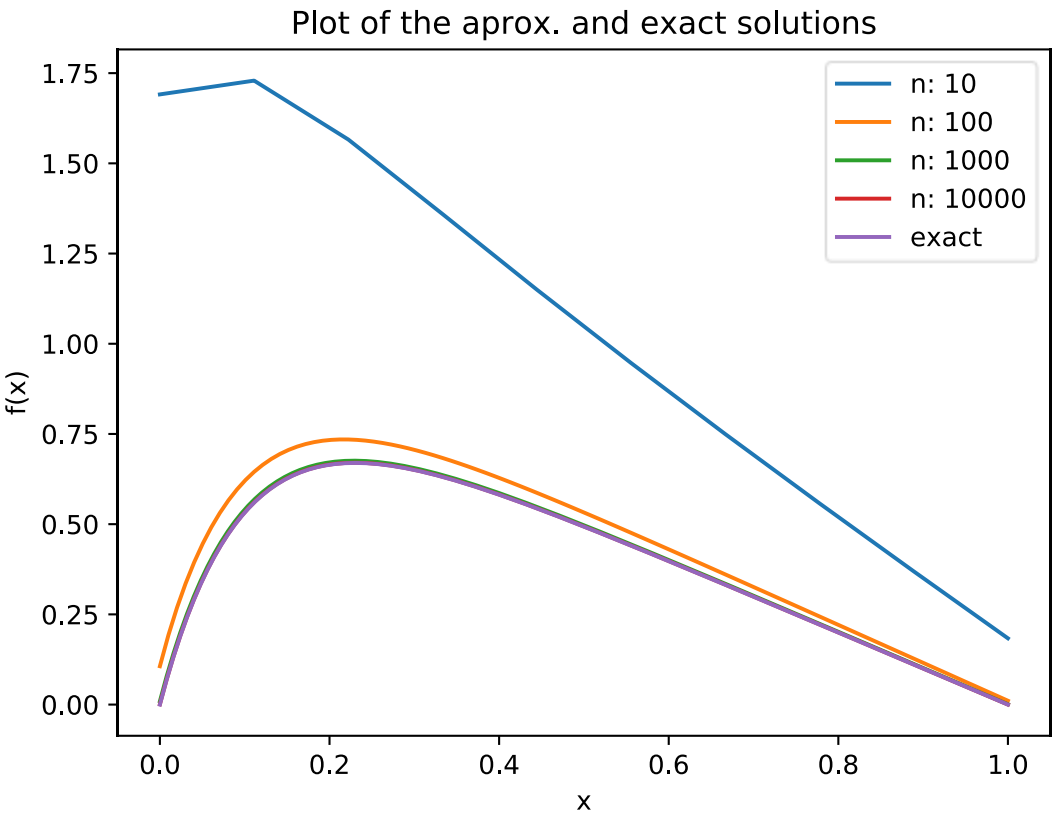

Figure 1 The numerical solution rapidly approaches the exact solution as n increases.

Runtime

The general and particular solution are almost identical in runtime, with a respective runtime of 1.31s and 1.13s for $n = 1e8$

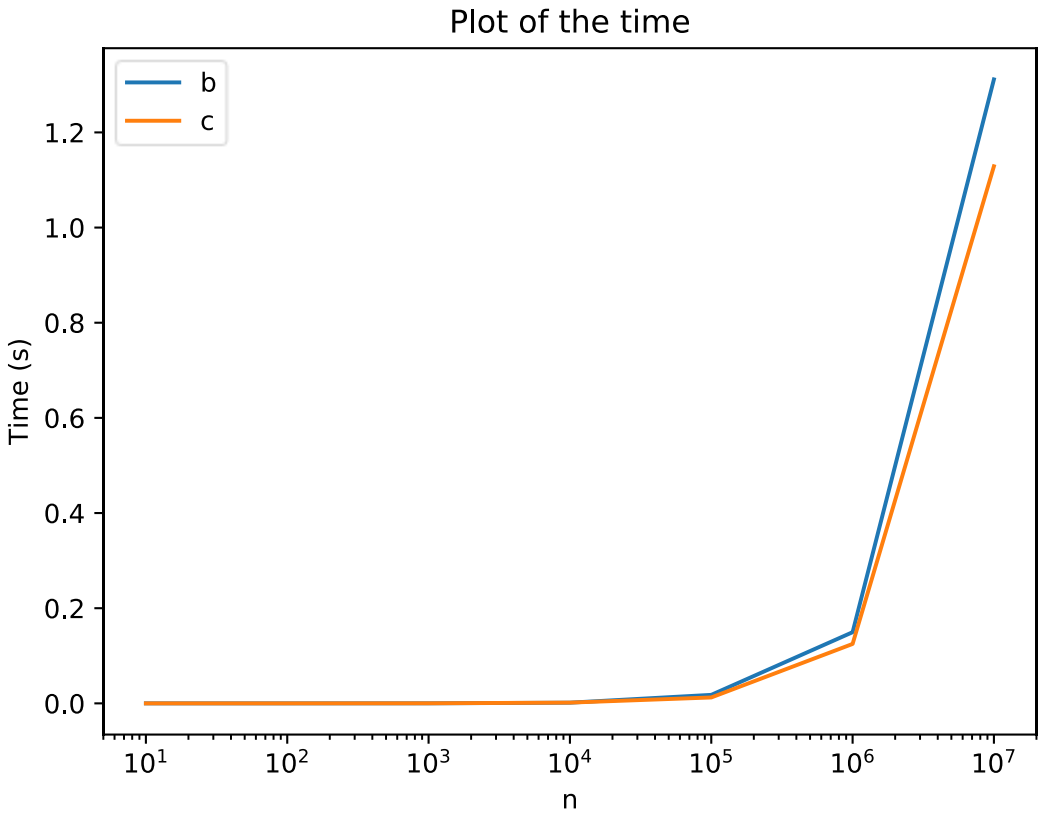

Figure 2 Logarithmic plot of computation time for different n.

The relative error

We expected to find a relative error that followed the truncation error $\propto n^2$, and then at some point arount $1e7$ the error should start to jump sporadically because of the computational error. However, as we can see from figure 3, there was no such computational error.

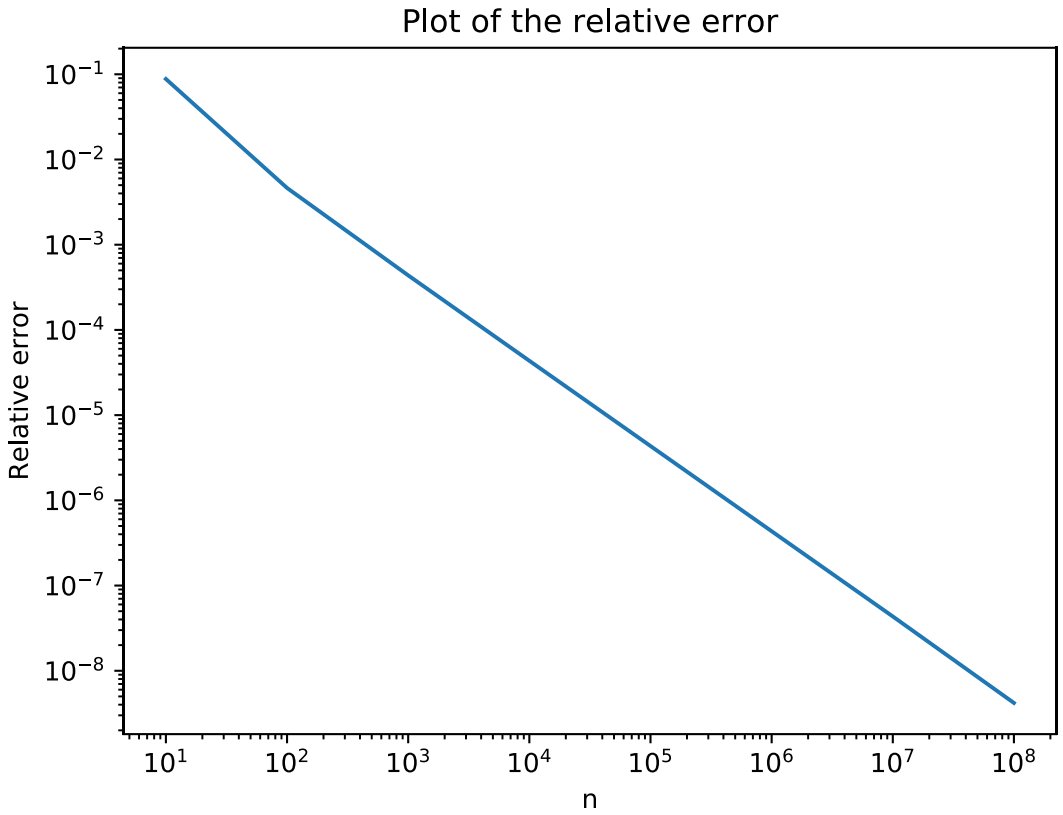

Figure 3 A logarithmic plot of the relative error of the general solution.

LU-decomposition

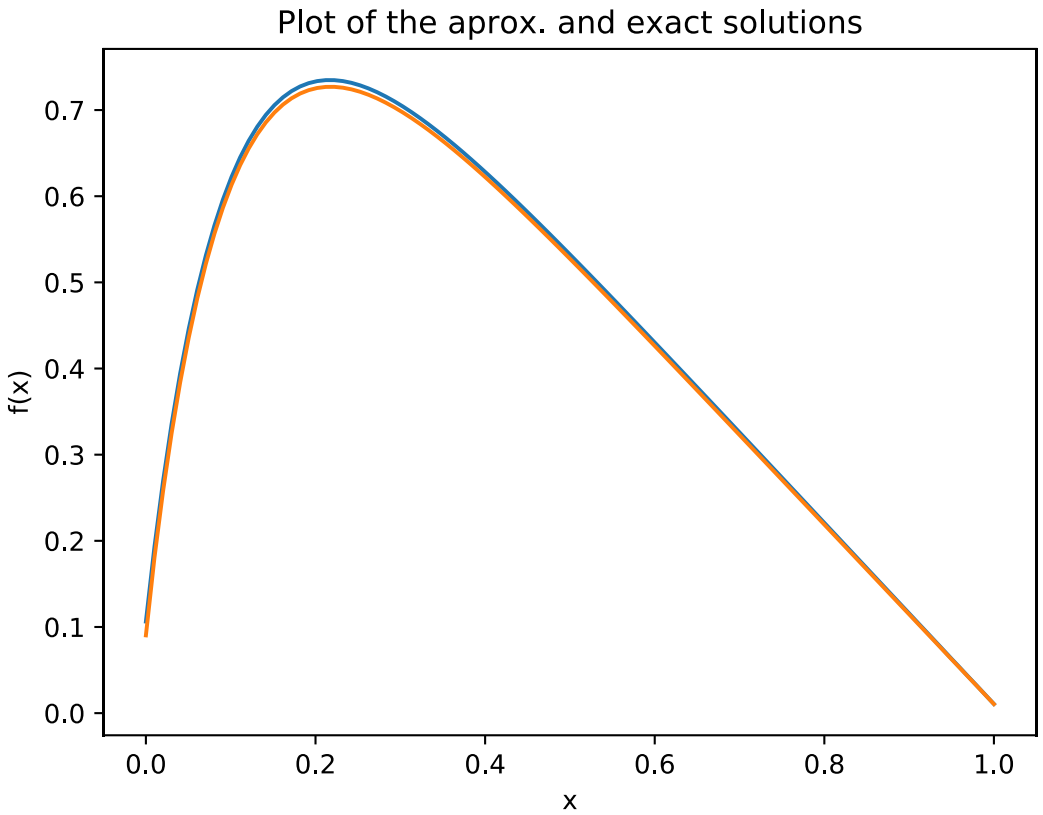

Figure 4 Plot of computed and exact solutions with LU-decomposition. The blue graph is the computed one, and the orange is the exact.

Conclusions

By tailoring the algorithm to the specific equation, the memory requirements are reduced from $64n^2$ to $64 \cdot 3n$. This is especially convenient when we observe no computational error. If we have the memory for it we should be able to increase n as much as we want. It could also mean that there is a flaw in our program somewhere.

When it comes to computation time there is a decrease about 14% when applying the particular algorithm. This is expected from the 8 to 7 flops ratio.