Draft version August 27, 2018
Preprint typeset using LaTeX style emulateapj v. 12/16/11

FYS3150 PROJECT 1

Sigbjørn L. Foss

Draft version August 27, 2018

ABSTRACT

We implement three different algorithms for solving a second order linear differential equation and compare the exactness of the solutions and the CPU time for the execution. We find that using a standard LU decomposition for this particular equation is very CPU intensive, and memory considerations also restricts the resolution in the discretization. The most efficient algorithm is able to do the calculations with $6n$ floating point operations which is an order of 2 lower than the $O(2/3n^3)$ operations required in the LU decomposition algorithm.

1. INTRODUCTION

In this project, we use the solving of a linear second order equation in demonstrating central programming and algorithmic concepts. We rewrite the equation to a matrix form, to be solved by C++ implementation. When implementing the program, we make use of dynamic memory handling to allow us to easily change the dimensions of the matrix we are working with, corresponding to changing the resolution of the discretized equation. The goal is to minimize the CPU time of the calculations. In achieving this, we start by looking at a general tridiagonal matrix, and derive an algorithm to solve the equation this represents. We then specialize to the case of a tridiagonal Toeplitz matrix, which is the representation we find for the equation we are trying to solve. We find an algorithm for solving such a system, and show that the number of floating point operations required for this case is lower than in the general case. We compare the CPU times of the general algorithm, the specialized algorithm and an algorithm involving the C++ armadillo functions for solving an LU-decomposed system. In all of these cases we also make and compare error estimates.

The methods section of this report will go through the algorithms used in solving the exercises, the results section will go through the end product of the algorithms and finally, we discuss the results in the conclusion section.

2. METHODS

2.1. *Rewriting the equation in matrix form*

We start by rewriting the given equation

$$-\frac{v_{i+1}+v_{i-1}-2v_i}{h^2} = f_i \quad \text{for } i = 1,\dots,n, \qquad (1)$$

where $f_i = f(x_i)$ as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix,

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

with $\tilde{b}_i = h^2 f_i$.
The set of equations read

$$\begin{aligned} i = 1 &: -v_0 + 2v_1 - v_2 = f_1 h^2 \\ i = 2 &: -v_1 + 2v_2 - v_3 = f_2 h^2 \\ &\vdots \quad \vdots \\ i = n &: -v_{n-1} + 2v_n - v_{n+1} = f_n h^2 \end{aligned}$$

where $v_0 = 0$ and $v_{n+1} = 0$. Adding some zeros to the equations,

$$\begin{aligned} i = 1 &: \quad 2v_1 - v_2 + 0 + 0 + 0 + 0 + \cdots + 0 = f_1 h^2 \\ i = 2 &: \quad -v_1 + 2v_2 - v_3 + 0 + 0 + \cdots + 0 = f_2 h^2 \\ i = 3 &: \quad 0 - v_2 + 2v_3 - v_4 + 0 + \cdots + 0 = f_3 h^2 \\ &\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ i = n-1 &: \quad 0 + \cdots + 0 - v_{n-2} + 2v_{n-1} - v_n = f_{n-1} h^2 \\ i = n &: \quad 0 + \cdots + 0 + 0 + 0 - v_{n-1} + 2v_n = f_n h^2 \end{aligned}$$

we recognize that the left hand side may be written as a product of a matrix containing the coefficients,

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

and the vector

sigbjolf@student.matnat.uio.no

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

The right hand side is then contained in the vector

$$\tilde{\mathbf{b}} = \begin{bmatrix} f_1 h^2 \\ f_2 h^2 \\ \vdots \\ f_n h^2 \end{bmatrix},$$

and the equation may then be written on the form

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 h^2 \\ f_2 h^2 \\ f_3 h^2 \\ \vdots \\ f_{n-1} h^2 \\ f_n h^2 \end{bmatrix}$$

$$\Rightarrow \mathbf{A}\mathbf{v} = \tilde{\mathbf{b}}.$$

The equation to be solved is

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

with $f(x) = 100e^{-10x}$.
This has a known solution $u(x)$ and derivatives,

$$\begin{aligned} u(x) &= 1 - \left(1 - e^{-10}\right) x - e^{-10x} \\ u'(x) &= \left(1 - e^{-10}\right) + 10e^{-10x} \\ u''(x) &= -100e^{-10x}. \end{aligned}$$

Inserting these into the Poisson equation, we find

$$-u''(x) = -(-100)e^{-10x} = 100e^{-10x} = f(x).$$

2.2. *Algorithm for a general tridiagonal matrix*

The previous section showed that we may write the Poisson equation with Dirchlet boundary conditions as a set of linear equations represented with a tridiagonal matrix. For simplicity we will define an equation matrix with rows E_i,

$$E = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & 0 & \tilde{b_1} \\ a_1 & b_2 & c_2 & 0 & \cdots & 0 & \tilde{b_1} \\ 0 & a_2 & b_3 & c_3 & \cdots & 0 & \tilde{b_1} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n-1} & b_{n-1} & c_{n-1} & \tilde{b}_{n-1} \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_n & \tilde{b_n} \end{bmatrix}.$$

To solve these equations, we will perform a forward and a backward substitution to get the matrix on a diagonal form. We define a new matrix where the first row is equal to the first row of the matrix E, $B_1 = E_1$. Performing the operation

$$B_{i+1} = E_{i+1} - \frac{a_i}{b_i} B_i$$

on each row for $i < n-1$ results in the upper triangular matrix

$$B = \begin{bmatrix} b'_1 & c_1 & 0 & \cdots & 0 & 0 & \tilde{b'_1} \\ 0 & b'_2 & c_2 & 0 & \cdots & 0 & \tilde{b'_2} \\ 0 & 0 & b'_3 & c_3 & \cdots & 0 & \tilde{b'_2} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & b'_{n-1} & c_{n-1} & \tilde{b'}_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & b'_n & \tilde{b'_n} \end{bmatrix}.$$

The new diagonal elements are found by

$$b'_{i+1} = b_{i+1} - \frac{a_i c_i}{b'_i}$$

and the elements in the last column to the right are

$$\tilde{b'}_{i+1} = \tilde{b}_{i+1} - \frac{a_i}{b'_i} \tilde{b'_i}$$

. The backwards substitution is done in a similar way. We define a matrix C where $C_n = B_n$ and perform the operation

$$C_{i-1} = B_{i-1} - \frac{c_{i-1}}{b'_i} C_i$$

for $i > 1$. The matrix is then

$$C = \begin{bmatrix} b'_1 & 0 & 0 & \cdots & 0 & 0 & \tilde{b''_1} \\ 0 & b'_2 & 0 & 0 & \cdots & 0 & \tilde{b''_2} \\ 0 & 0 & b'_3 & 0 & \cdots & 0 & \tilde{b''_2} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & b'_{n-1} & 0 & \tilde{b''}_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & b'_n & \tilde{b''_n} \end{bmatrix}.$$

The elements in the last column to the right are then given by

$$\tilde{b''_i} = \tilde{b'}_{n-1} - \frac{c_{n-1}}{b'_n} \tilde{b''_n}.$$

The solution is then found by

$$\tilde{b''_i} = b'_{i-1} u_{i-1} = \tilde{b'}_{n-1} - \frac{c_{n-1}}{b'_n} b'_i u_i$$

$$u_{i-1} = \frac{\tilde{b'}_{n-1} - c_{i-1} u_i}{b'_{i-1}}.$$

We see that it is sufficient to implement loops over

$$b'_{i+1} = b_{i+1} - \frac{a_i c_i}{b'_i},$$

$$\tilde{b'}_{i+1} = \tilde{b}_{i+1} - \frac{a_i}{b'_i} \tilde{b'_i},$$

and

$$u_{i-1} = \frac{\tilde{b'}_{i-1} - c_{i-1} u_i}{b'_{i-1}}.$$

2.3. *Algorithm for the special case*

For the special case, we have $c_i = a_i = -1$ and $b_i = 2$. The diagonal elements may then be pre calculated by

$$b'_{i+1} = 2 - \frac{1}{b'_i} = \frac{i+2}{i+1} \Rightarrow b'_i = \frac{i+1}{i}.$$

The rightmost column elements in the forward algorithm are then given by

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

and the solution is found by

$$u_{i-1} = \frac{\tilde{b'}_{i-1} + u_i}{\frac{i}{i-1}} = \frac{i-1}{i}\left(\tilde{b'}_{i-1} + u_i\right).$$

2.4. *LU decomposition*

An equation on the form

$$\mathbf{Ax} = \mathbf{b}$$

may be solved by decomposing the matrix $\mathbf{A}$ into a product of a lower- and upper diagonal matrix,

$$\mathbf{A} = \mathbf{LU}.$$

The system is then solved by

$$\mathbf{Ly} = \mathbf{b},$$

$$\mathbf{Ux} = \mathbf{y}.$$

This process in implemented for a 4x4 matrix and a four dimensional vector with random elemets in armadillo as following:

```
mat A = randu<mat>(4, 4);
vec b = randu<vec>(4);
mat L, U, P;
lu(L, U, P, A);
vec y = zeros<vec>(n);
y = solve(L, b);
vec solution = zeros<vec>(n);
solution = solve(U, y);
```

2.5. *Error analysis*

The Error analysis is fairly straight forward. The logarithm of the error in each step in the numerical approximation is given by

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

where v_i and u_i are the steps in the numerical and analytical solutions respectively. This is implemented as a loop:

```
mat error = zeros <vec>(n-2);
    for (int i=1; i<n-1 ; i++){
      error(i) = log10(abs((solution(i+1)
      - exact(x(i+1)))/exact(x(i+1))));
    }
```

2.6. *Timing of the algorithms*

To record the CPU time of the algorithms, we use the 'time.h' functionality in C++:

```
clock_t start, finish;
  start = clock();
<Algorithm is executed here >
  finish = clock();
  double timeused=(double)(finish - start)/
                  ((double)CLOCKS_PER_SEC);
```

This stores the CPU time in seconds in the variable 'timeused'. The time resolution of this function is limited however, so it is only useful when using a fairly large n, depending on which algorithm is timed.

2.7. *Reading of the results*

Most of the algorithms used print the results to a file which is in turn read by a short python script which plots the results, this script is found (along with the others) in the referenced github repository.

3. RESULTS

3.1. *Comparison of the general algorithm result and the colsed form solution*

As expected, for relatively low values of n, that is $n \leq 1000$, the error of the numerical solution diminishes, as shown in figure 1. For $n = 10$, the error of each point is more erratic, and the result isn't necessarily applicable.

3.2. *Floating point operations and CPU usage*

Looking at the terms in the general algorithm, we may count the number of floating point operations. The expressions were

$$b'_{i+1} = b_{i+1} - \frac{a_i c_i}{b'_i},$$

$$\tilde{b'}_{i+1} = \tilde{b}_{i+1} - \frac{a_i}{b'_i}\tilde{b'_i},$$

and

$$u_{i-1} = \frac{\tilde{b'}_{i-1} - c_{i-1}u_i}{b'_{i-1}}.$$

We see that each of these steps need three floating point operations each. They are implemented $n-1$ times in the algorithm, so the total number of operations needed is $9(n-1) \approx 9n$ for large n. The specialized algorithm precalculates the diagonal elements b'_i, so we only need the following expressions

$$\tilde{b'}_{i+1} = \tilde{b}_{i+1} + \frac{i}{i+1}\tilde{b'_i} = \tilde{b}_{i+1} + \frac{\tilde{b'_i}}{b'_i}$$

and

$$u_{i-1} = \frac{\tilde{b'}_{i-1} + u_i}{b'_{i-1}}.$$

Each of these operations perform three floating point operations each, so using this algorithm reduces the total number of flops to $6(n-1) \approx 6n$. Lastly, we know that the LU-decomposition needs $\mathcal{O}(2/3n^3)$ operations, so the first two algorithms constitute a significant leap in efficiency.

Assuming that each operation uses the same amount of CPU time, we would expect that the specialized algorithm should be 3/2 times faster than the general one, while the time expenditure should be larger than these by an exponential factor of two. However, using the "time.h" functionality, I've been unable to get a time resolution high enough to register times less than 0.01 seconds, so the resolution limit in my implementation of the timer means that the CPU time of the LU-method and the other two cannot be compared directly. C++

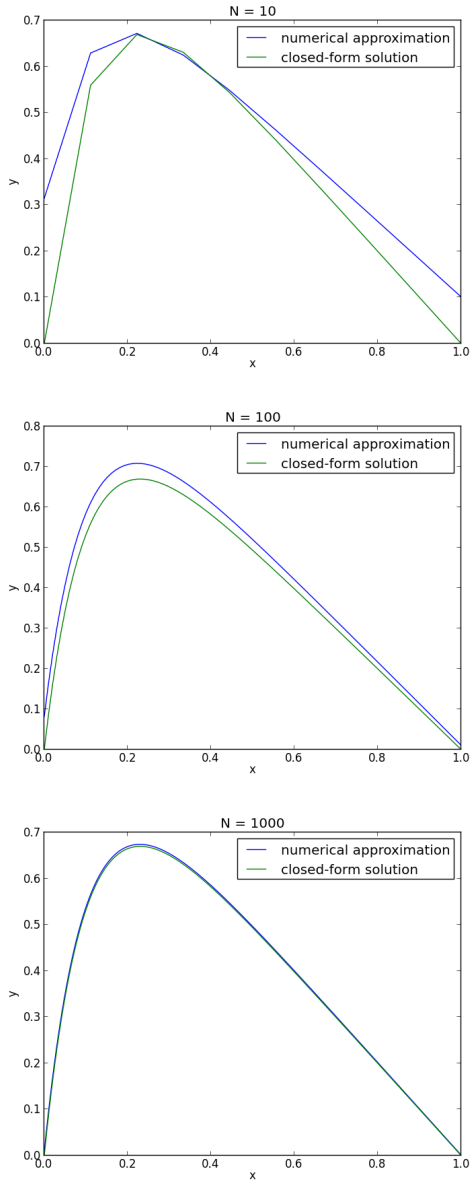

FIG. 1.— Comparison of the numerical and analytical solutions for $n = 10$, $n = 100$ and $n = 1000$

will not allow initializing a matrix with dimension 5. The results are shown in table 1. We can compare the times of the special and the general algorithms approximately for $n = 10^7$ however. We see that the ratio between the times for the special (t_s) and general (t_g) algorithms is

$$\frac{t_g}{t_s} = \frac{0.45}{0.26} \approx 1.73$$

which is slightly higher than the expected value of 1.5. This discrepancy is likely a result of the assumption that different floating point operations take an equal amount of time.

3.3. *Error estimates for the different algorithms*

Table 2 shows the error when running the programs for different values of n. We see that the error, exept for $n = 10$, decreases linearly when increasing n. The algorithm thus doesn't run into a problem with loss of precision even with up to 10^7 steps in the integration loop. We also see that the error of the specialised algorithm is somewhat smaller than for the LU algorithm. This is likely due to the fact that there are a lot more steps in the LU algorithm, so the round of errors propagate through a larger chain of operations.

TABLE 1
CPU TIME OF THE THREE ALGORITHMS IN SECONDS

$log_{10}(n)$	General algorithm	Specialized algorithm	LU algorithm
1	0	0	0
2	0	0	0
3	0	0	0.03
4	0	0	3.19
5	0.01	0.01	
6	0.04	0.03	
7	0.45	0.26	

TABLE 2
LOGARITHMIC ERROR OF THE ALGORITHMS

$log_{10}(n)$	mean value of $log_{10}(\epsilon_i)$	mean value of $log_{10}(\epsilon_i)$ (LU)
1	-1.2919	0.032847
2	-1.07838	-1.01779
3	-2.02083	-2.01497
4	-3.01442	-3.01384
5	-4.01388	
6	-5.01696	
7	-6.05417	

4. CONCLUSIONS

The main takeaway is that the developed algorithm specialized for the equation we are solving is faster, less CPU intensive and gives more accurate results. The algorithm holds for a finer discretization than I had predicted, even with step sizes as small as 10^{-7}. There were a fair amount of issues during development however. I have yet to start using GitHub in its intended manner, so I will only be uploading finished versions of the programs I've used. Additionally, the way I've structured the files in the project makes it quite tough to easily run the programs with different functionality. it's also made the workflow much less comfortable, and has resulted in the final product lacking in certain aspects. Moving forward, I need to wrok more systematically, so I can retain an overview of the project. The github page is found here: https://github.com/sifoss/project1-