DRAFT VERSION SEPTEMBER 11, 2018
Preprint typeset using LaTeX style emulateapj v. 11/26/04

FYS3150 COMPUTATIONAL PHYSICS, PROJECT 1

STIG-NICOLAI FOYN, CHRISTER DREIERSTAD

Draft version September 11, 2018

ABSTRACT

Comparing methods for solving second order differential equations this study will present that producing a specialized method reduces the run time of a numerical solver. When solving a second order differential equation we find that tailoring a method to the problem at hand will result in a shorter CPU time compared to a general method similar to the specialized. Further comparison proves that even slightly tailoring the algorithm (by the general method) will prove to be faster than a standard LU decomposition. For large scale calculations the LU decomposition is found to use more memory than the computer has available, while the general and specialized methods produces results, the latter being faster. The specialized method will focus on reducing the floating point operations (FLOPS) by tailoring the equations prior to the numerical calculations.

Subject headings: computational science — numerical methods: error estimation, LU decomposition, forward/backward substitution — methods: analytical, numerical, statistical

1. INTRODUCTION

This study on generalized and specialized numerical algorithms will focus on how a differential equation is solved numerically most efficiently while not reducing the accuracy of the calculations. By comparing the run time and observing the numerical error of the calculations . To specify, the equation in question is the one dimensional Poisson equation:

$$\nabla^2\Phi = -4\pi\rho(r),$$

where Φ is the electrostatic potential generated by a localized charge distribution and ρ is the charge density. This equation is used in electromagnetism, but in this case it was only necessary to analyze a normalized equation on the same form. Our normalized one dimensional Poisson equation can then be written as:

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

Further rewriting this as a set of linear equation will allow us to solve it using our numerical algorithms. As generalized algorithm we will be using a pre-built function for LU-decomposition. Specializing our algorithm further for the problem we will make a generalized algorithm for any solving any tridiagonal matrix. Lastly we will make an algorithm completely specialized to our problem where the matrix elements are already specified.

2. METHOD

For the study we consider a solution for the source term $f(x)$ to be

$$f(x) = 100e^{-10x},$$

which has a closed form solution

$$u_{cf}(x) = 1 - (1 - e^{10})x - e^{-10x}, \tag{2}$$

which yields $-f(x)$ when taking the second derivative.

Electronic address: stignicf@student.matnat.uio.no, christdre@student.matnat.uio.no
[1] Institute of physics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway

We consider the one dimensional Poisson equation to uphold Dirichlet boundary conditions; $u(0) = u(1) = 0$, so $x \in [0,1]$. The second order derivative $u(x)$ can be represented as

$$-\frac{du^2}{dx^2} = -\left(\frac{u(x+h) + u(x-h) - 2u(x)}{h^2} + \mathcal{O}(h^2)\right), \tag{3}$$

where h is the step size.

2.1. *Numerical representation*

When creating matrices and vectors in C++ they are created as dynamic arrays, so that the length of the array is decided based upon the problem. This is done to reduce the memory usage.

By approximating equation (3) and discretizing over x we get

$$-\frac{du^2}{dx^2} = -\frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = f(x_i) = f_i.$$

Where the step size $h = 1/(n+1)$. When calculating $x_i = i \times h$ the calculations runs over 1 to $n+1$, such that the last

$$x_{n+1} = (n+1)h = (n+1)/(n+1) = 1, \tag{4}$$

which ensures that $x \in [0,1]$. To simplify the calculations and reduce the FLOPS, we multiply the above equation by h^2 and rewrite $f_i' = h^2 f_i = h^2 100e^{-10x_i}$, such that the final discretized equation for solving equation (1) is

$$-(u_{i-1} - 2u_i + u_{i+1}) = f_i'. \tag{5}$$

The linear combination above can be represented by a matrix vector multiplication. The matrix will consist of the coefficients of the u_i's along the central, upper and lower diagonal, making up a tridiagonal matrix. The coefficient, after taking the outer sign of the left hand side of equation (5) into consideration, are -1, 2 and -1 respectively. The vector will consist of the values of the u_i's. The matrix representation will therefore be on the following form:

$$\boldsymbol{Au} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_1 & b_2 & c_2 & 0 & \dots & \dots \\ 0 & a_2 & b_3 & c_3 & 0 & \dots \\ \dots & 0 & a_3 & b_4 & \dots & 0 \\ \dots & \dots & 0 & \dots & \dots & c_{n-1} \\ 0 & \dots & \dots & 0 & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{bmatrix} = \begin{bmatrix} f_1' \\ f_2' \\ \dots \\ \dots \\ \dots \\ f_n' \end{bmatrix} = \boldsymbol{f_i'}.$$

Since we are considering fixed end points for u we do not include $i = 0$ and $i = n+1$ in the calculations for updating the u_i's.

2.2. *General algorithm*

We consider the matrix $\boldsymbol{A}$ to consist of three vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ consisting of all a_i, b_i and c_i respectively. To solve the matrix equation we perform a forward and a backward substitution of $\boldsymbol{A}$ and $\boldsymbol{f}'$ respectively.

In order to forward substitute $\boldsymbol{A}$, the matrix must be upper triangular, which is done by Gaussian elimination. An example with a $4x4$-matrix $\boldsymbol{E}$ and two $4x1$ vectors x and g follows to show the algorithm that are to be presented later. Consider

$$\boldsymbol{Ex} = \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix} = \boldsymbol{g}.$$

Performing Gaussian elimination we multiply the first row of $\boldsymbol{E}$ by a_1/b_1

$$\frac{a_1}{b_1}\begin{bmatrix} b_1 & c_1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a_1 & \frac{a_1 c_1}{b_1} & 0 & 0 \end{bmatrix},$$

and then subtract this from the second row:

$$\begin{bmatrix} a_1 & b_2 & c_2 & 0 \end{bmatrix} - \begin{bmatrix} a_1 & \frac{a_1 c_1}{b_1} & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & b2 - \frac{a_1 c_1}{b_1} & c_2 & 0 \end{bmatrix}.$$

With these operations the matrix $\boldsymbol{E}$ now reads

$$\boldsymbol{E} = \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & b2 - \frac{a_1 c_1}{b_1} & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{bmatrix},$$

where the substitution $\tilde{b}_2 = b_2 - \frac{a_1 c_1}{b_1}$ is introduced. This is performed on each of the lower triangular elements such that the end result is a upper triangular matrix

$$\boldsymbol{E} = \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 \\ 0 & 0 & 0 & \tilde{b}_4 \end{bmatrix},$$

where in general

$$\tilde{b}_i = b_i - \frac{a_{i-1} c_{i-1}}{\tilde{b}_{i-1}}, \tag{6}$$

for $i \in [2, n+1]$. The above equation applies directly to $\boldsymbol{A}$. Since $\boldsymbol{E}$ is altered, the same operations must be applied to $\boldsymbol{g}$. Multiplying the first element of $\boldsymbol{g}$ by a_1/b_1 and subtracting this from the second element

$$\begin{bmatrix} g_2 \end{bmatrix} - \begin{bmatrix} \frac{g_1 a_1}{b_1} \end{bmatrix} = \begin{bmatrix} g_2 - \frac{g_1 a_1}{b_1} \end{bmatrix}.$$

We then substitute $\tilde{g}_2 = g_2 - \frac{g_1 a_1}{b_1}$. Applying this to f' gives the general expression

$$\tilde{f}_i = f'_i - \frac{\tilde{f}_{i-1} a_{i-1}}{\tilde{b}_{i-1}}, \tag{7}$$

for $i \in [2, n+1]$. The values for $\tilde{b}_i$ and $\tilde{f}_i$ are calculated through forward substitution. For $\tilde{b}_1$ and $\tilde{f}_1$ the value is not changed from the original b_1 and f'_1 respectively. Therefore the calculations are done over the interval $i \in [2, n+1]$ given the aforementioned initial conditions.

To calculate the values for u_i we perform a backward substitution. We return to the example $\boldsymbol{Ex} = \boldsymbol{g}$, considering the Gauss eliminated matrix equation. For $i = 4$ the equation reads

$$\tilde{b}_4 x_4 = \tilde{g}_4 \Rightarrow x_4 = \frac{\tilde{g}_4}{\tilde{b}_4},$$

and for $i = 3$

$$\tilde{b}_3 x_3 + a_3 x_4 = \tilde{g}_3 \Rightarrow \tilde{x}_3 = \frac{\tilde{g}_3 - a_3 x_4}{\tilde{b}_3}.$$

In general and applied to $\boldsymbol{Au} = \boldsymbol{f}'$ the equation for u_i reads:

$$u_i = \frac{\tilde{f}_i - c_i u_{i+1}}{\tilde{b}_i}. \tag{8}$$

Following is pseudo code for the solution of the general algorithm.

```
//Allocate dynamic memory for variables
double *a = new double[n+1]
//similar for other variables b, c, f-tilde, u and
    x (an array of step sizes)

//Define the step size as h and make an array of
    steps from 0 to n+2
for i = 0; i < n+2; i++{
    x[i] = h*double(i)
}

//Updating b-tilde and f-tilde using the equations
    we found using forward substitution
for i = 2; i < n+2; i++{
    btilde[i] = update
    ftilde[i] = update
}

//Updating u using the equations we found using
    backward substitution
for i = n; i > 0; i--{
    u[i] = update
}
//write the results to a file
```

2.3. *Specific algorithm*

Further the goal is to improve upon the general algorithm by reducing the FLOPS and making sure there are only one calculation for each FLOP.

Since a_i, b_i and c_i are constant for all i the value can be inserted into equations (6), (7) and (8):

$$\tilde{b}_i = b_i - \frac{a_{i-1} c_{i-1}}{\tilde{b}_{i-1}} = 2 - \frac{1}{\tilde{b}_{i-1}},$$

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

$$u_i = \frac{\tilde{f}_i - c_i u_{i+1}}{\tilde{b}_i} = \frac{\tilde{f}_i + u_{i+1}}{\tilde{b}_i}.$$

In the above equations $1/\tilde{b}_{i-1}$ is calculated twice, which results in unnecessary calculations. Introducing a temporary variable $b'_i = 1/\tilde{b}_{i-1}$ changes the above equations for $\tilde{b}_i$ and $\tilde{f}_i$ to

$$\tilde{b}_i = 2 - b'_{i-1},$$

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

Having reduced the FLOPS to minimize the CPU time the calculations are timed to compare with the general algorithm. Timing is done with a built in function within C++ and only ran over the calculations of $\boldsymbol{Au} = \tilde{\boldsymbol{f}}$. Pseudo code is presented below

```
//Allocate dynamic memory for variables, but a, b
    and c now have specific values
double a = -1; double b = 2; double c = -1
```

```
//Updating b and f-tilde using the equations we
    found using forward substitution
for i = 2; i < n+2; i++{
    btilde[i] = update (a and c are now constants)
    ftilde[i] = update (a and c are now constants)
}
```

2.4. *Relative error*

The relative error is calculated by

$$\epsilon_i = \log_{10}\left(\left|\frac{u_i - u_{cf,i}}{u_{cf,i}}\right|\right). \tag{9}$$

When computing the relative error the indices are reduced such that division by zero is avoided, that is when $u_{cf,i} \to 0$. Following is pseudo code:

```
eps_max //calculating the initial value, assuming
    this is largest
for i=n/10+1:(9*n/10)+1 //ignoring end points
    calculate epsilon
    if epsilon > eps_max
        eps_max = epsilon
```

2.5. *LU-decomposition*

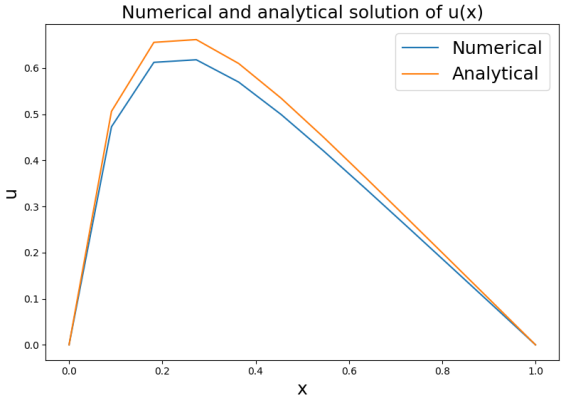
Lastly we compare the previous results to a solution of the matrix equation $\boldsymbol{Au} = \boldsymbol{f}$ by an LU decomposition. We decompose the matrix such that $\boldsymbol{A} = \boldsymbol{Lu}$, where $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{u}$ is an upper triangular matrix:

$$\boldsymbol{Lu} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & \dots \\ \dots & \dots & \dots & 0 \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & u_{nn} \end{bmatrix}.$$

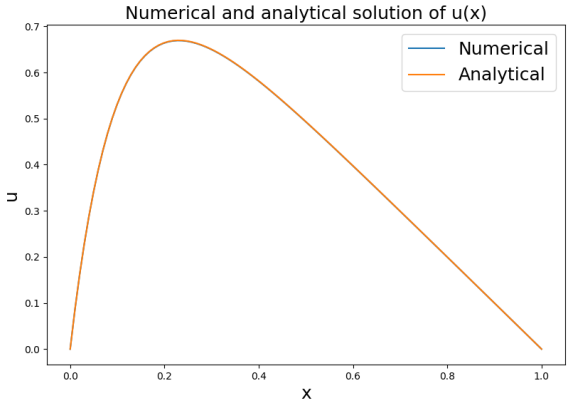
For the LU decomposition a function "solve" from the C++ package Armadillo is used.

3. RESULTS

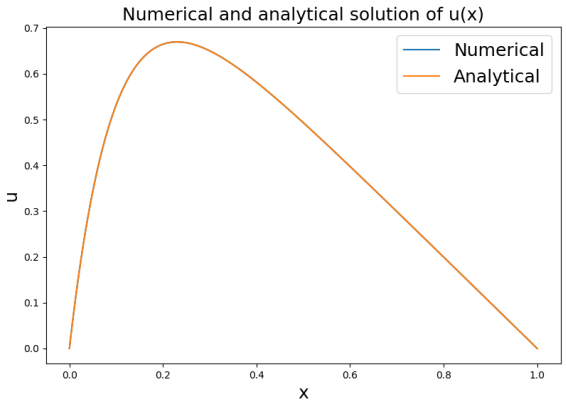
3.1. *Numerical error*



(a) The resolution of our step size is low and as a result there is a visible deviation from the analytical solution.



(b) Lowering the step size renders the deviation from the analytical solution invisible in the plot.



(c) Further lowering the step size the only visible difference is in the resolution of the curve in our plot.

FIG. 1.—: The solution of the differential equation using a generalized algorithm at matrix sizes (a): $n = 10$, (b): $n = 100$ and (c): $n = 1000$.

3.2. *Program CPU time*

Testing the time we ran the three different programs 10 times for different sizes n, listing longest, shortest and average CPU times. In Table 2 the results we have listed for timing our programs with $n = 1000$. Comparably for larger matrices $n = 10^6$ the programs yield the timings presented in Table 3.

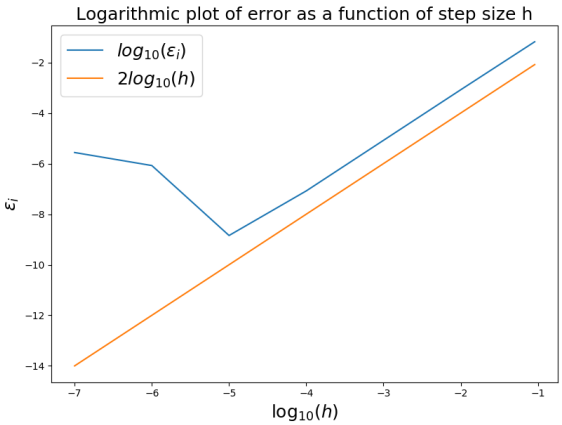


FIG. 2.—: Logarithmic plot of the error as function of the step size, to illustrate the $\mathcal{O}(h^2)$ dependency we have plotted $log_{10}(h^2) = 2log_{10}(h)$, which in the plot doubles as $y = 2x$, the expected value of our error.

TABLE 1:

n	Error
10^1	$6.6114 \cdot 10^{-2}$
10^2	$8.1651 \cdot 10^{-4}$
10^3	$8.3167 \cdot 10^{-6}$
10^4	$8.3311 \cdot 10^{-8}$
10^5	$1.4356 \cdot 10^{-9}$
10^6	$8.3973 \cdot 10^{-7}$
10^7	$2.7487 \cdot 10^{-6}$

NOTE. — Table of the error, for $n \in [10^1, 10^7]$. The table reveals that the numerical error increases when $n > 10^5$, while the error for $n \leq 10^5$ follow the expected value $\mathcal{O}(h^2)$.

TABLE 2:

	General	Specialized	Armadillo
Highest	0.000333s	0.000237s	0.021747s
Lowest	0.000191s	0.000074s	0.012255s
Average	0.000249s	0.000167s	0.015978s

NOTE. — This table contains highest, lowest and average measured CPU times for the different algorithms used at n = 1000, armadillo being overall, while our general and specialized algorithm timings both withing the same order of magnitude.

Note that the LU-decomposition runs out of memory at $n = 10^5$ meaning that we will not be able to test the Armadillo made algorithm for larger matrices. Our general algorithm runs at $8n$ FLOPS, while our specialized algorithm is optimized to only run at $6n$ FLOPS.

4. CONCLUSIONS

4.1. *Error*

Reviewing the plots from Fig. 1 we can see that the error is greater for $n = 10$ than it is for $n = 100$ points. Which is expected of our algorithm considering the error is of the order h^2, where h is the step size. However between these step sizes we already reach the maximum resolution of a side by side comparison, and between $n = 100$ and $n = 1000$ the only visible change is in the resolution of the plot itself. Further we compute the relative error of the specialized algorithm compared to the closed form solution and plot it in a logarithmic scale as seen in Fig. 2. By Fig. 2 we can see that the error goes as expected until $log_{10}(h) = -5$ and the error increases. This is the point where our numerical model becomes unstable due to the step size being too small, called loss of numerical accuracy, which is a direct result from n being too large.

TABLE 3:

	General	Specialized	Armadillo
Highest	0.085026	0.073819s	Out of memory
Lowest	0.074668	0.064465s	Out of memory
Average	0.079288s	0.067805s	Out of memory

NOTE. — This table contains highest, lowest and average measured CPU times for the different algorithms used at $n = 10^6$, armadillo runs out of memory, while the general and specialized algorithms staying within the same order of magnitude.

4.2. *CPU time*

While CPU time of the specialized and the general algorithms are of same order of magnitude, there is a clear difference for $n = 1000$. Analyzing the number of FLOPS in the different algorithms, it becomes clear that the specialized algorithm should be $8n/6n = 3/2$ times faster. Comparing average CPU times we see that this is highly accurate for $n = 1000$, in which case its $0.000249/0.000167 = 1.49$times faster. However for $n = 10^6$ this does not seem to be the case entirely, as we find it to be $0.079288/0.067805 = 1.17$ times faster. This can partially be attributed to the small sample size and the imprecise method when performing the calculations. The testing environment for calculating was a laptop, we expect that for large n the calculations take more time and there is a higher load on the CPU. This load results in a higher temperature which on a standard laptop could cause the performance to drop.

4.3. *LU decomposition*

The LU decomposition is as expected a slower and more demanding algorithm. Since the algorithm stores the variables as matrices instead of vectors, the computer runs out of memory for larger n i.e $n > 10^5$. The reason that the LU decomposition runs out of memory is that it saves every value of the matrix, including the zeros on the lower diagonal $\boldsymbol{L}$ and the upper diagonal of u which takes up a lot of memory when storing the $n \times n$ matrices. The number of FLOPS using LU decomposition is proportional to n^2 since we are dealing with nxn matrices. This has a considerable effect and even for smaller n, we get a slower CPU time, due to the large number of operations required in our calculations.

4.4. *Comments*

We both found this project to be great as an introduction to FYS3150 Computational Physics and as a way to start learning C++. We did not have many critiques, other than

the fact that in section 1b the variables written in the matrix and the ones in the vector b and $\tilde{b}$ was a bit unclear at first glance, and it is suggested that $\tilde{b}$ in the matrix $\boldsymbol{A}$ is renamed to $\tilde{f}$.

REFERENCES

[1] M. H. Jensen, *Project 1*, https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2018/Project1/pdf/Project1.pdf (10.09.18)