# FYS3150 - project 1

Sverre Wehn Noremsaune & Frida Marie Engøy Westby
(Dated: September 13, 2021)

https://github.uio.no/comPhys/FYS3150/tree/project1/project1

## PROBLEM 1

We have the one-dimensional Poisson equation

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

where $f(x)$ is known to be $100e^{-10x}$. We also assume $x \in [0,1]$, that the boundary condition are $u(0) = 0 = u(1)$ and $u(x)$ is

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{2}$$

where $u(x)$ is an exact solution to Eq. (1). We can check this analytically by differentiating $u(x)$ twice.

$$\begin{aligned} -u''(x) &= f(x) \\ u''(x) &= -f(x) \\ u(x)' &= 10x^{-10x} - 1 + \frac{1}{e} \\ u''(x) &= -100e^{-10x} = -f(x) \quad \blacksquare \end{aligned}$$

## PROBLEM 2

See FIG 1. wich shows the plot of the exact Poisson solution. See 'poisson_exact.cpp' and 'algorithms.cpp' in the github repository, for source code.

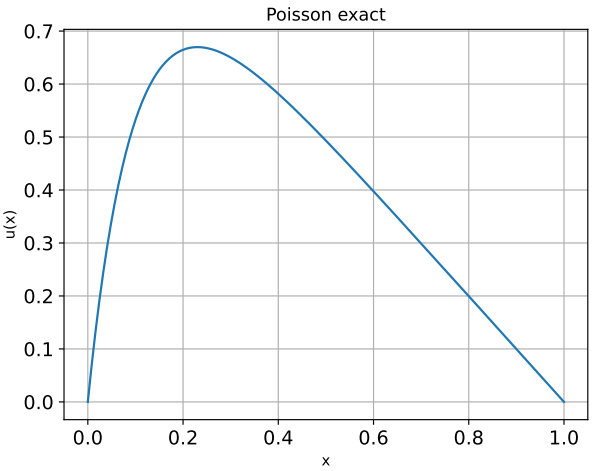


FIG. 1. A plot of the exact solution for the Poisson equation from 1 for $x \in [0,1]$

## PROBLEM 3

We are discretizing the Poisson equation from 1.
Discretizing x and setting up some notation:

$$\begin{aligned} x &\to x_i \\ u(x) &\to u_i \\ i &= 0, 1, \ldots, n \\ h &= \frac{x_{max} - x_{min}}{n} \\ x_i &= x_0 + ih \end{aligned}$$

Next we're using the three-point formula to find the second derivative:

$$\frac{du^2}{dx^2} = u'' = \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + O(h^2)$$

$$f_i = -\left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + O(h^2)\right)$$

Then we approximate and change the notation, $v_i \approx u_i$ and get

$$f_i = \frac{-v_{i-1} + 2v_i - v_{i+1}}{h^2} \tag{3}$$

## PROBLEM 4

The equation we got in 3 isn't the most ergonomic for setting up a matrix equation, so we can rewrite it to

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \tag{4}$$

Equation 4 is a set of equations for every i

$$\begin{array}{lccccc} i = 1 & -v_0 & 2v_1 & -v_2 & & = h^2 f_i \\ i = 2 & & -v_1 & 2v_2 & -v_3 & = h^2 f_2 \end{array}$$

and so on and so forth. We can see that $v_0$ and $v_n$ will end up and alone on their columns, and since we know what they are we simply move them over.

$$\begin{array}{lccccc} i = 1 & 2v_1 & -v_2 & & & = h^2 f_1 + v_0 \\ i = 2 & -v_1 & 2v_2 & -v_3 & & = h^2 f_2 \end{array}$$

This can then be easily rewritten as a matrix equation $A\vec{v} = \vec{g}$

$$\begin{bmatrix} 2 & -1 & 0 & 0 & \ldots & 0 & 0 \\ -1 & 2 & -1 & 0 & \ldots & 0 & 0 \\ 0 & -1 & 2 & -1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & -1 & 2 & -1 & 0 \\ 0 & 0 & \ldots & 0 & -1 & 2 & -1 \\ 0 & 0 & \ldots & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{n-3} \\ v_{n-2} \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ gn-3 \\ gn-2 \\ g_{n-1} \end{bmatrix}$$

where $g_i$ is $h^2 f_i$ ($+v_0$ for $f_1$ and $+v_n$ for $f_{n-1}$).

## PROBLEM 5

### Problem a

We can see that n is related to m since we're "dropping" 2 columns, which gives $n = m - 2$

### Problem b

We will find $\vec{v_i^*}$ for $1..(n-1)$, meaning everything but the boundary points.

## problem 6

Here will we look at a general tridiagonal matrix, and not the Poisson equation from earlier.

### Problem a

**Algorithm 1** Algorithm for solving general tridiagonal matrix

```
arrays a, b, c, u, f, temp of length n

btemp = b[1]
u[1] = f[1]/btemp

for  i = 2,3,...,n  do
    temp[i] = c[i-1] / btemp
    btemp = b[i] - a[i] * temp[i]
    u[i] = (f[i] - a[i] * u[i-1]) / btemp
for i = n-1, n-2, ..., 1 do
    u[i] -= temp[i+1] * u[i+1]
```

### Problem b

When we analyse the algorithm above, we can see that we get $1 + 6(n-1) + 2(n-1) = 1 + 8(n-1) = 8n - 7$ FLOPs.

## PROBLEM 7

In this problem and the problems below, will we once again look at the Poisson Equation.

### Problem a

See 'general_tridiag.cpp' and 'algorithms.cpp' in the github repository for the program that solves the Poisson equation with the general algorithm.

### Problem b

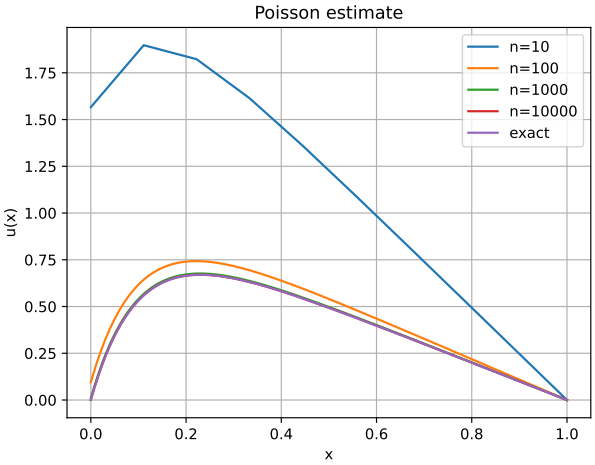


FIG. 2. Exact vs numerical comparison for the solution of equation 1

## PROBLEM 8

### Problem a

See FIG 3 for the polt showing the (logarithm of) the absolute error.

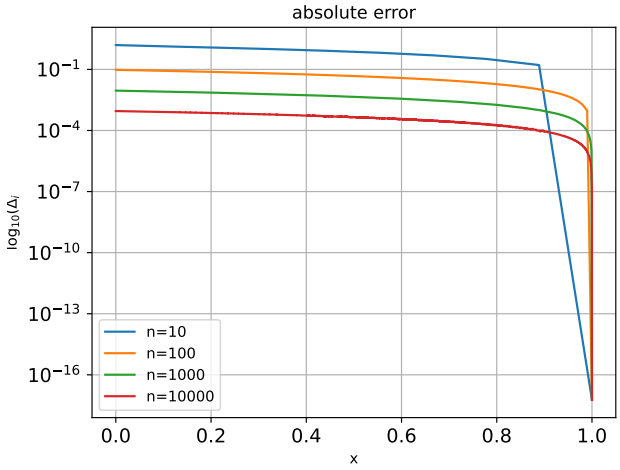


FIG. 3. The absolute error for the general tridiagonal alogrithm in $log_{10}$

### Problem b

See FIG 4 for the polt showing the (logarithm of) the relative error.

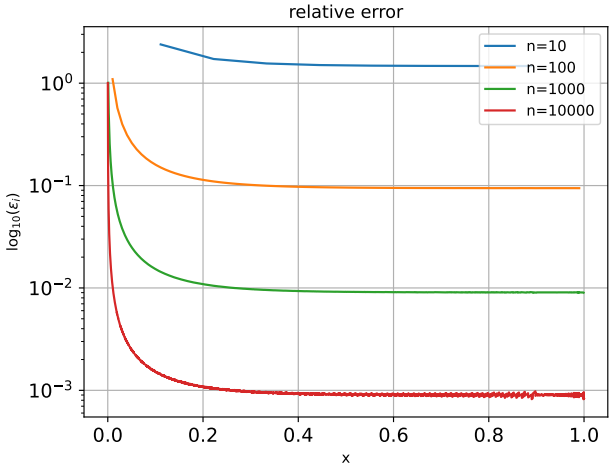


FIG. 4. The relative error of the general tridiagonal algorithm

**Problem c**

See FIG 5 for the plot for the table showing the maximum relative error. We see that for highter $n$ the closer we get to 1.0, and from $n = 1000000$ the max relative error becomes 1.0. Combined with the plot of the relative error above, we can see that the approximate and exact value differs quite dramatically close to $x = 0$ in comparison to the rest.

| n | max rel error |
|---:|---:|
| 10 | 2.3903877077005538 |
| 100 | 1.0946543221364102 |
| 1000 | 1.0091303813999173 |
| 10000 | 1.0009115263620092 |
| 100000 | 1.000088892839682 |
| 1000000 | 1.0 |
| 10000000 | 1.0 |

FIG. 5. Table with the biggest relative error for each iteration

## PROBLEM 9

Since the values of $a, b, c$ never changes, we can just replace the arrays with a constant, saving us from repeatedly calculating $-(-1) \cdot something$.

### Problem a

**Algorithm 2** Algorithm for solving special tridiagonal matrix

```
arrays u, f, temp of length n

btemp = 2
u[1] = f[1]/ 2

for  i = 2,3,...,n  do
    temp[i] = -1 / btemp
    btemp = b[i] + temp[i]
    u[i] = (f[i] + u[i-1]) / btemp
for i = n-1, n-2, ..., 1 do
    u[i] -= temp[i+1] * u[i+1]
```

### Problem b

FLOPs: $1 + 4(n-1) + 2(n-1) = 1 + 6(n-1) = 6n - 5$

### Problem c

See the function 'special_tridiag' in 'data_algorithms.cpp' in the github repository for the special algorithm.

## PROBLEM 10

| n | general (s) | special (s) |
|---|---|---|
| 100 | 8.516e-07 | 7.594e-07 |
| 1000 | 8.666e-06 | 7.3036e-06 |
| 10000 | 0.00011016 | 7.2997e-05 |
| 100000 | 0.0013352 | 0.00074125 |
| 1000000 | 0.013795 | 0.011789 |
| 10000000 | 0.14217 | 0.11722 |

FIG. 6. A table showing the running time of the general and special algorithm for a given $n$

By pure FLOP count the special should be about 25% faster, but we're not seeing quite that big gains. There are some outliers, but it is probably safe to assume that is due to some of the data being in cache. Assuming loading from memory is the bottleneck the speed up could also be due to the special algorithm loading less data.

## PROBLEM 11

LU decomposition has a complexity of $O(N^3)(+O(N^2))$ vs our algorithm that runs in $O(N)$. for $n = 10^5$ I would expect to:

1. Run out of memory, since you need to store $8 * N * N = 8 * 10^5 * 10^5 = 8 * 10^{10} \approx 80$GB.
2. To be very slow. The alogrithm's $O$ factor is two orders of magnitude larger, so probably around 100 times as long.