PROJECT 1

Computational Physics - FYS3150

September 13, 2021

Duncan Wilkins | Fuad Dadvar | Don Phillip Bullecer | Erling Nupen
https://github.com/dondondooooon/FYS3150

Introduction

The overall topic of this project is numerical solution of the one-dimensional Poisson equation. This is a second-order differential equation that shows up in several areas of physics, e.g. electrostatics. In future projects we will pay close attention to scaling of dimensional physics equations, but for this first project we start directly from the differential equation after scaling to dimensionless variables

The one-dimensional Poisson equation can be written as

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

Here $f(x)$ is a known function (the source term). Our task is to find the function $u(x)$ that satisfies this equation for a given boundary condition. The specific setup we will assume is the following:

- source term: $f(x) = 100e^{-10x}$
- x range: $x \in [0, 1]$
- boundry condition: u(0)=0, u(1) = 0

Problem I

In this task we're to check if that an exact solution to equation(1) is given by

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

If $u(x)$ is a solution to equation (1) in the problem text, then the following relation must hold:

$$\frac{d^2}{dx^2}((1 - (1 - e^{-10})x - e^{-10x})) = -100e^{-10x} = f(x) \tag{3}$$

Its quite straight forward to differentiate $u(x)$. Since it is a second order derivative, all first order terms of x will not survive the derivation. So we end up with $u''(x) = -10 \cdot (-10)(-10)e^{-10x}$ which is exactly equal to $f(x)$ and therefore is a solution to our equation.

Problem II

We wrote a program in C++ shown in that evaluates equation(2) which is the exact solution u(x) for a given array of x values of length N = 1000.
The program generates a data text file, and we use a short plotting script such as in Listing(1) in python to read our data file and generate the plot shown under in figur(1)

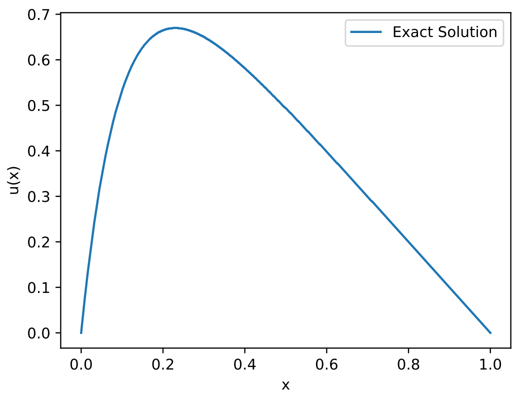

Figure 1: Plot of x vs. u(x)

```cpp
#include "head.hpp"

double u(double x);                          // Declaration of Equation 2 Function

int main(){
    //Makes a file.txt for the data points:
    string filename = "prob2.txt";
    ofstream ofile;                          //Create and open the output file
    ofile.open(filename);                    //Connect it to filename

    //Parameters for matrix
    double N = 1000;                          //Number of data points that will be used
    mat A = mat(N,2);           // Creates an Nx2 matrix
    A.col(0) = linspace(0,1,N);       // Creates N linearly spaced vector from start to end
    vec sz = A.col(0);                //Used to get the size of one column in the matrix

    //The Loop
    for (int i=0; i < sz.size() ; i++){     //Loop through x vector indexes
        A(i,1) = u(A(i,0));                  //Insert the function values into second column in matrix A

        //Saving the matrix into a data file
        ofile << setw(12) << setprecision(2) << scientific << A(i,0);            //Upload x-values
        ofile << setw(12) << setprecision(2) << scientific << A(i,1) << endl;     //Upload y-values
    }

    ofile.close();  //Close output file

    return 0;
}

//Calculating the y-values
double u(double x){                                  // Takes in an element x[i] from the loop
        return ( 1-(1-exp(-10))*x-exp(-10*x) );     //Calculates and spits out the solution using that value
}
```

Listing 1: C++ program

```
import numpy as np
import matplotlib.pyplot as plt

# Unload text file
x,y = np.loadtxt('prob2.txt', usecols=(0,1), unpack=True)

# Figure size (inches)
figwidth = 5.5
figheight = figwidth / 1.33333
plt.figure(figsize=(figwidth, figheight))

# Plot x vs. y (u(x))
plt.plot(x,y, label='Exact Solution')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.legend()
plt.savefig('Exact_Solution_plot.pdf')
plt.show()
```

Listing 2: Python plot script

Problem III

We have the equation for Taylor series that's defined:

$$f(x+h) = \sum_{x=0}^{\infty} \frac{1}{h!} f^n(x) h^n \tag{4}$$

Forward Taylor equation gives us:

$$f_{(x+h)} = f(x) + f'(x)h + \frac{1}{2}f'(x)h^2 + \frac{1}{6}f''(x)h^3 + O(h^4) \tag{5}$$

Backward Taylor equation gives us:

$$f_{(x-h)} = f(x) - f'(x)h + \frac{1}{2}f'(x)h^2 - \frac{1}{6}f''(x)h^3 + O(h^4) \tag{6}$$

Let's add Eq(5) and Eq(6) and simplify the equation:

$$f_{(x+h)} + f_{(x-h)} = 2f(x) + f''(x)h^2 \Leftrightarrow f''(x) = \frac{f_{(x+h)} - 2f(x) + f_{(x-h)}}{h^2} + O(h^2) \tag{7}$$

Now that we've generated a second derivative formula for an arbitrary function shown in Eq(7) we can switch out $f''(x)$with $u''(x)$ and set this equal to our function shown in Eq(1):

$$-u''(x) = f(x) \tag{8}$$

At the next step we'll change the x to x_i and so on $u(x_i)$ $f(x_i)$ with u_i and x_i so we'll have a discritized function:

$$f_i = -\Big[\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2)\Big] \tag{9}$$

Let's find the Approximation for the discritized function.

We need to rewrite our $u_i \approx v_i$: and we'll have:

$$f_i = -\Big[\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2}\Big] \tag{10}$$

Problem IV

We now have the discretized "approximated"? form of the Poisson equation:

$$f_i = -\Big[\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2}\Big] \tag{11}$$

Moving forward we want to rewrite the our discretized Poisson equation in the form of a matrix equation on the form:

$$A \cdot \vec{v} = \vec{g} \tag{12}$$

First of all, we'll rewrite the discretized Poisson equation (11) in a more intuitive way:

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i \tag{13}$$

Now we'll examin more closely what we get if we calculate different points for f_i in a range from i = 1 to i = 4.

$$\begin{aligned}
&(i=1) && -v_0 + 2v_1 - v_2 && = h^2 f_1 \\
&(i=2) && -v_1 + 2v_2 - v_3 && = h^2 f_2 \\
&(i=3) && -v_2 + 2v_3 - v_4 && = h^2 f_3 \\
&(i=4) && -v_3 + 2v_4 - v_5 && = h^2 f_4 \\
&. \\
&. \\
&. \\
&(i=n) && ....v_{n-1} + 2v_n - v_{n+1} && = h^2 f_n
\end{aligned}$$

Where we now see, that we can write our system of equations as a matrix which is zero everywhere except the sub- and super diagonal, and the diagonal itself. Before we write out our full matrix equation we'll rewrite the first and last equation, by moving them over to the other side of the equation as follows:

$$
\begin{aligned}
&(i = 1) && 2v_1 - v_2 && = h^2 f_1 + v_0 \\
&(i = 2) && - v_1 + 2v_2 - v_3 && = h^2 f_2 \\
&(i = 3) && - v_2 + 2v_3 - v_4 && = h^2 f_3 \\
&(i = 4) && - v_3 + 2v_4 - v_5 && = h^2 f_4 \\
&. \\
&. \\
&. \\
&(i = n) && .... + 2v_n - v_{n+1} && = h^2 f_n + v_{n-1}
\end{aligned}
$$

The matrix equation below is how our discretized Poisson equation look like on matrix equation form:

$$
\begin{bmatrix}
2 & -1 & 0 & 0 & 0 & \cdots & 0 \\
-1 & 2 & -1 & 0 & 0 & \cdots & 0 \\
0 & -1 & 2 & -1 & 0 & \cdots & 0 \\
0 & 0 & -1 & 2 & -1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\
0 & 0 & 0 & 0 & 0 & \cdots & 2
\end{bmatrix}
\cdot
\begin{bmatrix}
v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_n
\end{bmatrix}
= h^2 \cdot
\begin{bmatrix}
f_1 + v_0 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_n + v_{n-1}
\end{bmatrix}
\tag{14}
$$

Problem V

a.

The relation between m and n is as follows:

$$
n = m - 2 \tag{15}
$$

This relation comes from the fact that we moved over two elements from $\vec{m}$, and added them to the $\vec{g}$, as you can see in the second matrix in task 4. More precisely $\vec{v}^* = (0, \vec{v}, 0)$. In matrix form, this is the same as the first and second column is zero, and since the dimensions of the matrix A and $\vec{v}$ must be the same, the vector v must be of length m - 2 and the matrix A looses two columns. This is quite fortunate, because if the endpoints were not known, then in the case where i = 4, we would have 6 variables and only 4 equations, and we would not be able to solve the systems of equations. The matrix equation below represents our matrix equation which is the complete solution, where A has m (but two are zero, so we can just remove them) columns and v has m-2 elements.

$$\begin{bmatrix} 0 & a & b & 0 & 0 & \cdots & 0 \\ 0 & c & a & b & 0 & \cdots & 0 \\ 0 & 0 & c & a & b & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ \\ \vec{v} \\ \\ 0 \end{bmatrix} = h^2 \cdot \begin{bmatrix} f_1 + v_0 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_n + v_{n-1} \end{bmatrix} \tag{16}$$

b.

Solving for $\vec{v}$ yields the solution of $\vec{v}^*$ except the endpoints. As we defined above, $\vec{v}^*$ is equal to $(0, \vec{v}, 0)$

Problem VI

In this task we're to write down an algorithm for solving $A\vec{v} = \vec{g}$

$A\vec{v} = \vec{g}$ is the following equation beneath:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}$$

We row reduce the matrix by applying $R_2 \to R_2 - (\frac{a_2}{b_1})R_1$ operation:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & (b_2 - \frac{a_1}{b_1}c_1) & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix} \cdot \begin{bmatrix} g_1 \\ (g_2 - \frac{a_2}{b_1}g_1) \\ g_3 \\ g_4 \end{bmatrix}$$

New Notation

$$\tilde{b}_1 = b_1 \tag{17}$$

$$\tilde{g}_1 = g_1 \tag{18}$$

$$\tilde{b}_2 = b_2 - \frac{a_2}{\tilde{b}_1} c_1 \tag{19}$$

$$\tilde{g}_2 = g_2 - \frac{a_2}{\tilde{b}_1} \tilde{g}_1 \tag{20}$$

With the new notation we'll get these following algorithm's for $\tilde{b}_i$ and $\tilde{g}_i$ by forward substituting:

$$\tilde{b}_i = b_i - \frac{a_i}{\tilde{b}_{i-1}} c_{i-1} \tag{21}$$

$$\tilde{g}_i = g_i - \frac{a_i}{\tilde{b}_{i-1}} \tilde{g}_{i-1} \tag{22}$$

Let's backward substitute and find the algorithm for v_i.

We'll write out our new matrix including the new notation mark, we also do matrice operations aswell:

$$\begin{bmatrix} \tilde{b}_1 & c_1 & 0 & 0 \\ 0 & \tilde{b}_2 & c_2 & 0 \\ 0 & 0 & \tilde{b}_3 & c_3 \\ 0 & 0 & 0 & \tilde{b}_4 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} \tilde{g}_1 \\ \tilde{g}_2 \\ \tilde{g}_3 \\ \tilde{g}_4 \end{bmatrix}$$

That gives us with the following row reduction:

$$\begin{array}{c} \\ \xrightarrow{R_4 \to \frac{R_4}{\tilde{b}_4}} \\ \xrightarrow{v_4 = \frac{\tilde{g}_4}{\tilde{b}_4}} \\ \xrightarrow{R_4 \to \frac{(R_3 - c_3 R_4)}{\tilde{b}_3}} \end{array} \left[\begin{array}{cccc|c} \tilde{b}_1 & c_1 & 0 & 0 & v_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & v_2 \\ 0 & 0 & \tilde{b}_3 & c_3 & v_3 = \frac{(\tilde{g}_3 - c_3 v_4)}{\tilde{b}_3} \\ 0 & 0 & 0 & \tilde{b}_4 & v_4 = \frac{\tilde{g}_3}{\tilde{b}_4} \end{array} \right]$$

We'll continue this process and finally get the resolved algorithm for v$_i$:

$$v_i = \frac{(\tilde{g}_i - c_i v_{i+1})}{\tilde{b}_i} \tag{23}$$

Algorithms	**FLOPs**
$\tilde{b}_1$	$3_n - 1$
$\tilde{g}_1$	$3_n - 1$
v_i	$3_n - 1$

Table 1: When the value of the number n starts to increase we'll slowly neglect the value of the constant 1, since it won't do any effect on the equation for larger numbers.

Problem VII

Using the general algorithm from Problem VI we can write a program to solve $\mathbf{A}\vec{v} = \vec{g}$ where $\mathbf{A}$ is the tridiagonal matrix from Problem 4.

This program shown in *Listing 3* calculates the approximate solution $\vec{v}$, the exact solution $u(x)$, and their corresponding $\vec{x}$ for a range of different values for n and saves them into a data file.
The python script in *Listing 4* reads this data file and plots the results shown in *Figure 2*.

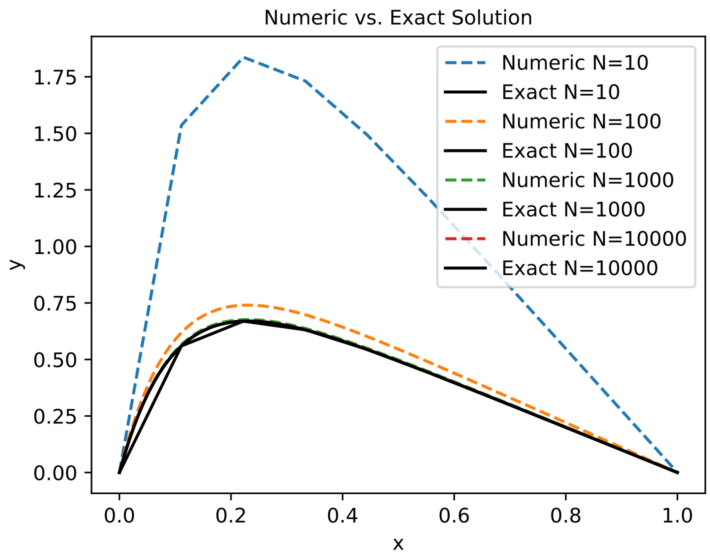

Figure 2: Plot of Numeric vs. Exact solutions of x

```
#include "head.hpp"
```

```

//Declaration of Functions
double u(double x);                                     // Analytical Exact Function
double f(double x);                                     // Original 2nd order derivative function

int main(){

    for(int i = 1; i < 5; i++){                         //Iterate N from 10 to 10^7

        int N = pow(10,i);                              //Number of data points that will be used
        double Nnew = N-2;                              //Length minus the two boundary points

        //Makes a file.txt for the data points:
        string filename = "prob7" + to_string(N) + ".txt";
        ofstream ofile;                                 //Create and open the output file
        ofile.open(filename);                           //Connect it to filename

        //Initialize vectors
        vec x = linspace(0,1,N);                        // Creates N linearly spaced vector from start to end
        vec y = vec(N);                                 // Initialize a vector for f''(x) of size N
        vec ux = vec(N);                                // Initialize a vector for u(x) of size N
        vec g = vec(N);                                 // g vector
        vec b = vec(Nnew).fill(2.);                     // Main-diagonal vector
        vec a = vec(Nnew).fill(-1.);                    // Sub-diagonal vector
        vec c = vec(Nnew).fill(-1.);                    // Super-diagonal vector
        vec bt = vec(Nnew);                             // b-tilde vector
        vec gt = vec(Nnew);                             // g-tilde vector
        vec vt = vec(N);                                // v-stjerne vector (The last two indexes for this vector will be empty)
        double h = x(1)-x(0);                           // h-Steps

        //Loop for y-values, ux-values and g-values
        for (int i=0; i < x.size(); i++){               // Loop through x vector indexes
            y(i) = f(x(i));                             // Fill in equation 1
            ux(i) = u(x(i));                             // Fill in equation 2
            g(i) = pow(h,2)*y(i);                       // Fill in indexes in g-vector (h^2*f_i)
        }

```

```
        //Initialze Boundaries
        g(0) = g(0) + ux(0);                            // g initial
        g(N-1) = g(N-1) + ux(N-1);                      // g end
        bt(0) = b(0);                                   // b-tilde initial
        gt(0) = g(0);                                   // g-tilde initial

        //Loop for b- and g-tilde vectors
        for (int i=1; i < Nnew ; i++){                  // Loop through N-2 indexes starting from 1
            bt(i) = b(i) - (a(i)/bt(i-1))*c(i-1);      // b-tilde vector def
            gt(i) = g(i) - (a(i)/bt(i-1))*gt(i-1);     // g-tilde vector def
        }

        //Initialize end element for v-stjerne vector
        vt(Nnew-1) = gt(Nnew-1)/bt(Nnew-1);

        //Loop for v-stjerne vector
        for (int i = Nnew-2; i >= 0 ; i--){             // Loop through N-2 indexes downwards starting from end index
            vt(i) = (gt(i)-c(i)*vt(i+1))/bt(i);        // v-stjerne vector def
        }

        // Writing them into the file
        for (int i=0; i<x.size(); i++){
            ofile << setw(15) << setprecision(5) << scientific << x(i);             // x-values
            ofile << setw(15) << setprecision(5) << scientific << vt(i);            // approx values
            ofile << setw(15) << setprecision(5) << scientific << ux(i) << endl;    // exact values
        }

        ofile.close();                                  // Close file
    }

    //Ferdig!
    cout << "Done!" << endl;
    return 0;
}

//Function for calculating the y-values
double f(double x){                                     // Takes in input x-element/index
    return (100*exp(-10*x));                            // From equation 1
}

//Function for calculating u(x)
double u(double x){                                     // Takes in input x-element/index
    return ( 1-(1-exp(-10))*x-exp(-10*x) );             // From equation 2
```

```
}
```

Listing 3: C++ program

```python
import numpy as np
import matplotlib.pyplot as plt

# Figure size mod
figwidth = 5.5
figheight = figwidth / 1.33333
plt.figure(figsize=(figwidth, figheight))

for i in range(1,5):
    N = 10**i                       # Iterate through the different N-values
    filename = f"prob7{N}.txt"      # Set name properly for text
    #print(filename) for checking purposes
    x,v,u = np.loadtxt(filename, usecols=(0,1,2), unpack=True)  # Unload correct text file
    v = np.roll(v,1)    # Shift once to the right to get end boundaries 0
    plt.plot(x,v,'--',label=f'Numeric N={N}')    # Plot v_n(x) vs. x
    plt.plot(x,u,'-',label=f'Exact N={N}',c='black')      # Plot u_n(x) vs. x

#Plots numeric vs. exact solution with different N values in the same plot
plt.title("Numeric vs. Exact Solution", fontsize=10)
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig("Num.vs.Exact_Ns.pdf")
plt.show()
```

Listing 4: Python plot script

Problem VIII

Problem VIII a.

In this task we're to make a plot that shows the absolute error, by using equation(24), if so we'll get this plot that shows us the absolute error as a function of x_i, and to show different choices N.

$$log_{10}(\Delta_i) = log_{10}(|u_i - v_i| \tag{24}$$

Problem VIII b.

And in this task we're to make the similarly plot of the relative error:

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

We'll have the following graph in figur(4) that shows us the complete solution

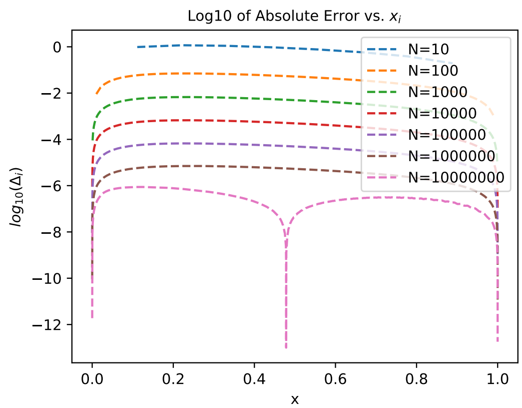

Figure 3: $log_{10}(\Delta_i)$ as a function of x_i, for multiple choices of N

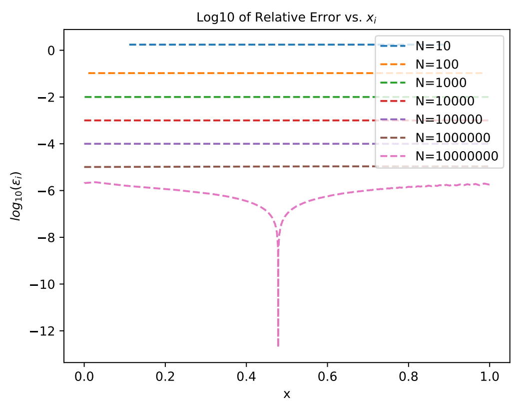

Figure 4: Relative error as a function of x_i for N[10,10^7]

Problem VIII c.

In this task we're to visualize the maximum relative error $max(\epsilon_i)$ for each choice of up to N = 10^7.

For each choice of n from $n = 10$ to 10^7 the plots in *Figure 5* show the corresponding maximum relative error.

Based on our results, we see that the higher value for the choice of N is, the lower the value for absolute and relative error. Therefore, the more reliable the results of the approximation becomes. However, higher choice of N also results in more data points to be evaluated resulting in unnecessary longer running time for the program.

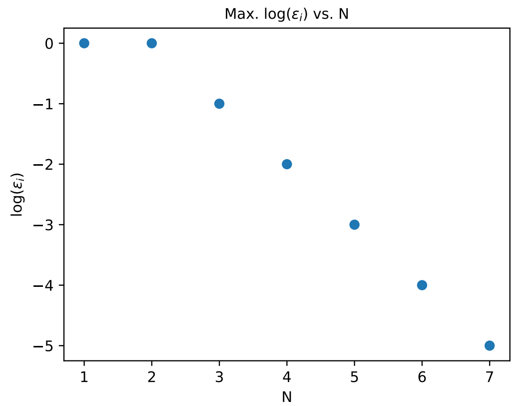

Figure 5: Maximum realtive error as a function of N, as for N to each step from 10^1 till 10^7

Problem IX

Now we're to specialize our algorithm from problem for our special case where **A** is specified by the signature $(-1, 2, -1)$, that is with

$$\vec{a} = [-1, -1, \dots, -1]$$

$$\vec{b} = [\;\; 2, \;\; 2, \dots, 2 \;\;]$$

$$\vec{c} = [-1, -1, \dots, -1]$$

ProblemIX a. b.

That gives us our matrix **A**:

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{bmatrix}$$

We can use the same procedure as we used in Problem 6 to solve this matrix-vector equations. However, since we know the following main, sub- and super-diagonal elements, we can just simplify our algorithm and reduce the FLOPs, we'll have:

New resulting algorithm

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

$$\tilde{g}_i = g_i + \frac{1}{\tilde{b}_1} \cdot \tilde{g}_{i-1}$$

$$v_N = \frac{\tilde{g}_N}{\tilde{b}_N}$$

$$v_i = \frac{\tilde{g}_i + v_{i+1}}{\tilde{b}_i}$$

So that means that the total FLOPs is $7n$ due to that ± 1 for a large n is insignificant. That means that we'll have $7n$ FLOPs for our new algorithm.

ProblemIX c.

The code that implements this new special algorithm is shown below in *Listing 5*. The difference between this new code vs. the previous one is on the lines where we defined $\tilde{b}_i$, $\tilde{g}_i$, v_N, v_i and the fact that we do not need to define and create the vector arrays $\vec{a}$, $\vec{b}$, and $\vec{c}$

```cpp
#include "head.hpp"

//Declaration of Functions
double u(double x);                                     // Analytical Exact Function
double f(double x);                                     // Original 2nd order derivative function

int main(){

    for(int i = 1; i < 6; i++){                         //Iterate N from 10 to 10^7

        int N = pow(10,i);                              //Number of data points that will be used
        double Nnew = N-2;                              //Length minus the two boundary points

        //Makes a file.txt for the data points:
        string filename = "prob9" + to_string(N) + ".txt";
        ofstream ofile;                                 //Create and open the output file
        ofile.open(filename);                           //Connect it to filename

        //Initialize vectors
        vec x = linspace(0,1,N);                        // Creates N linearly spaced vector from start to end
```

```
    vec y = vec(N);                             // Initialize a vector for f''(x) of size N
    vec ux = vec(N);                            // Initialize a vector for u(x) of size N
    vec g = vec(N);                             // g vector
    vec bt = vec(Nnew);                         // b-tilde vector
    vec gt = vec(Nnew);                         // g-tilde vector
    vec vt = vec(N);                            // v-stjerne vector (The last two indexes for this vector will be empty)
    double h = x(1)-x(0);                       // h-Steps

    //Loop for y-values, ux-values and g-values
    for (int i=0; i < x.size(); i++){           // Loop through x vector indexes
        y(i) = f(x(i));                         // Fill in equation 1
        ux(i) = u(x(i));                         // Fill in equation 2
        g(i) = pow(h,2)*y(i);                   // Fill in indexes in g-vector (h^2*f_i)
    }

    //Initialze Boundaries
    bt(0) = 2.;                                 // b-tilde initial
    gt(0) = g(0);                               // g-tilde initial
    g(0) = g(0) + ux(0);                        // g initial
    g(N-1) = g(N-1) + ux(N-1);                  // g end

    //Loop for b- and g-tilde vectors
    for (int i=1; i < Nnew ; i++){              // Loop through N-2 indexes starting from 1
        bt(i) = (double(i)+1)/double(i);        // b-tilde vector def
        gt(i) = g(i) + (1/bt(i-1))*gt(i-1);     // g-tilde vector def
    }

    //Initialize end element for v-stjerne vector
    vt(Nnew-1) = gt(Nnew-1)/bt(Nnew-1);

    //Loop for v-stjerne vector
    for (int i = Nnew-2; i >= 0 ; i--){         // Loop through N-2 indexes downwards starting from end index
        vt(i) = (gt(i)+vt(i+1))/bt(i);          // v-stjerne vector def
    }

    // Writing them into the file
    for (int i=0; i<x.size(); i++){
        ofile << setw(19) << setprecision(10) << scientific << x(i);              // x-values
        ofile << setw(19) << setprecision(10) << scientific << vt(i);             // approx values
```

```
            ofile << setw(19) << setprecision(10) << scientific << ux(i) << endl;    // exact values
        }

        ofile.close();                                    // Close file
    }

    //Ferdig!
    cout << "Done!" << endl;
    return 0;
}

//Function for calculating the y-values
double f(double x){                                    // Takes in input x-element/index
    return (100*exp(-10*x));                           // From equation 1
}

//Function for calculating u(x)
double u(double x){                                    // Takes in input x-element/index
    return ( 1-(1-exp(-10))*x-exp(-10*x) );            // From equation 2
}
```

Listing 5: C++ Program for Special Algorithm

Problem X

We run timing tests on our general and special algorithm based code. In *Listing 6*, we run both of the algorithms into a for loop to repeat the run for each choice of N for reliable timing results.

```
#include "head.hpp"

//Declaration of Functions
double u(double x);                        // Analytical Exact Function
double f(double x);                        // Original 2nd order derivative function

int main(){

for (int j=1; j<11; j++){            // Iterate 10 for reliable results
// Makes a .txt file for time measurements
string filename = to_string(j) +"_timedata.txt";
ofstream ofile;
ofile.open(filename);
    for(int i = 1; i < 7; i++){          //Iterate N from 1 to 10^6

        int N = pow(10,i);               //Number of data points that will be used
        double Nnew = N-2;               //Length minus the two boundary points
```

```
    vec x = linspace(0,1,N);         // Creates N linearly spaced vector from start to end
    vec y = vec(N);                  // Initialize a vector for y-values of size N
    vec ux = vec(N);                 // Initialize a vector for u(x) of size N
    vec g = vec(N);                  // g vector
    vec bt = vec(Nnew);              // b-tilde vector
    vec gt = vec(Nnew);              // g-tilde vector
    vec vt = vec(N);                 // v-stjerne vector
    vec bt2 = vec(Nnew);              // b-tilde vector
    vec gt2 = vec(Nnew);              // g-tilde vector
    vec vt2 = vec(N);                 // v-stjerne vector
    double h = x(1)-x(0);            // h-Steps
    //Loop for y-values, ux-values and g-values
    for (int i=0; i < x.size(); i++){    // Loop for y-values and g-values
        y(i) = f(x(i));                  // Fill in equation 1
        ux(i) = u(x(i));                 // Fill in equation 2
        g(i) = pow(h,2)*y(i);            // Fill in indexes in g-vector (h^2*f_i)
    }
    //Initialze Boundaries
    gt(0) = g(0);                     // g-tilde initial
    gt2(0) = g(0);                    // -"-
    g(0) = g(0) + ux(0);              // g initial
    g(N-1) = g(N-1) + ux(N-1);        // g end

    // Start measuring time for GENERAL
    auto gen_t1 = chrono::high_resolution_clock::now();
    //Initialize vectors
    vec b = vec(Nnew).fill(2.);      // Main-diagonal vector
    vec a = vec(Nnew).fill(-1.);     // Sub-diagonal vector
    vec c = vec(Nnew).fill(-1.);     // Super-diagonal vector
    bt(0) = b(0);                    // b-tilde initial

    //Loop for b- and g-tilde vectors
    for (int i=1; i < Nnew ; i++){               // Loop through N-2 indexes starting from 1
        bt(i) = b(i) - (a(i)/bt(i-1))*c(i-1);   // b-tilde vector def
        gt(i) = g(i) - (a(i)/bt(i-1))*gt(i-1);  // g-tilde vector def
    }
    //Initialize end element for v-tilde vector
    vt(Nnew-1) = gt(Nnew-1)/bt(Nnew-1);
    //Loop for v-stjerne vector
    for (int i = Nnew-2; i >= 0 ; i--){         // Loop through N-2 indexes downwards starting from end index
        vt(i) = (gt(i)-c(i)*vt(i+1))/bt(i);     // v-stjerne vector def
    }
    // Stop measuring time for general
    auto gen_t2 = std::chrono::high_resolution_clock::now();


    // Start measuring time for SPECIAL
```

```
        auto sps_t1 = chrono::high_resolution_clock::now();
        bt2(0) = 2.;                                    // b-tilde initial

        //Loop for b- and g-tilde vectors
        for (int i=1; i < Nnew ; i++){                  // Loop through N-2 indexes starting from 1
            bt2(i) = (double(i)+1)/double(i);           // b-tilde vector def
            gt2(i) = g(i) + (1/bt2(i-1))*gt2(i-1);      // g-tilde vector def
        }
        //Initialize end element for v-stjerne vector
        vt2(Nnew-1) = gt2(Nnew-1)/bt2(Nnew-1);
        //Loop for v-stjerne vector
        for (int i = Nnew-2; i >= 0 ; i--){         // Loop through N-2 indexes downwards starting from end index
            vt2(i) = (gt2(i)+vt2(i+1))/bt2(i);      // v-stjerne vector def
        }
        // Stop measuring time for special
        auto sps_t2 = std::chrono::high_resolution_clock::now();

        // Calculate elapsed time
        double duration_seconds1 = chrono::duration<double>(gen_t2 - gen_t1).count();
        double duration_seconds2 = chrono::duration<double>(sps_t2 - sps_t1).count();
        // Write time results into file
        ofile << setw(15) << setprecision(5) << duration_seconds1;    // time taken for gen.
        ofile << setw(15) << setprecision(5) << duration_seconds2 << endl; // time taken for spes.
    }
ofile.close();                              // Close file
}
    // Ferdig
    cout << "Done!" << endl;
    return 0;
}

//Function for calculating the y-values
double f(double x){                                 // Takes in input x-element/index
    return (100*exp(-10*x));                        // From equation 1
}

//Function for calculating u(x)
double u(double x){                                 // Takes in input x-element/index
    return ( 1-(1-exp(-10))*x-exp(-10*x) );         // From equation 2
}
```

Listing 6: C++ Program for Time Test

Using a python script such as *Listing 8*, we get a table showing the time results for both algorithms.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

G = np.empty([1,6])
S = np.empty([1,6])

for i in range(1,11):
    # Iterate 1 to 11 and find correct .txt file
    filename = f"{i}_timedata.txt"
    tg,ts = np.loadtxt(filename, usecols=(0,1), unpack=True) # Unload the 2 columns
    G = np.vstack([G,np.transpose(np.log10(tg))])
    S = np.vstack([S,np.transpose(np.log10(ts))])

G = np.delete(G,(0), axis=0)
S = np.delete(S,(0), axis=0)

print("The average time taken for general algorithm from n=10 to n=10^7 respectively is: ", G.mean(axis=0))
print("The average time taken for special algorithm from n=10 to n=10^7 respectively is: ", S.mean(axis=0))

#This Plots the Table
"""
# Plot 1
fig, ax1 = plt.subplots(figsize=(14,4))
fig.patch.set_visible(False)
ax1.axis('off')
ax1.axis('tight')

dfg = pd.DataFrame(G, columns=np.logspace(1,6,num=6,base=10,dtype='int'))
ax1.table(cellText=dfg.values, colLabels=dfg.columns, loc='center')

ax1.set_title('General Algorithm')
fig.tight_layout()
plt.savefig('TimeTest_General.pdf')
plt.show()

# Plot 2
fig, ax2 = plt.subplots(figsize=(14,3))
fig.patch.set_visible(False)
ax2.axis('off')
ax2.axis('tight')

dfs = pd.DataFrame(S, columns=np.logspace(1,6,num=6,base=10,dtype='int'))
ax2.table(cellText=dfs.values, colLabels=dfs.columns, loc='center')

ax2.set_title('Special Algorithm')
plt.savefig('TimeTest_Special.pdf')
plt.show()
"""
```

Listing 7: Python table plot script

Listing 7 prints out the averages of the time results for the repeated runs for each N used. These results are shown below.

N	$\log_{10}$(Time[s])
10^1	-5.43
10^2	-4.65
10^3	-3.81
10^4	-2.78
10^5	-1.76
10^6	-0.82

(a) General Algorithm

N	$\log_{10}$(Time[s])
10^1	-5.93
10^2	-5.10
10^3	-4.11
10^4	-3.12
10^5	-2.09
10^6	-1.13

(b) Special Algorithm

Table 2: Time Results

Based on *Table 2*, we see that the run time used for the Special Algorithm is less than the one for the General Algorithm.

This means that the Special Algorithm runs faster. This is due to the fact that we have reduced the number of FLOPs in our Special Algorithm.

Problem XI

In this case we're to solve our matrix equation using the general LU decomposition approach. For the decomposition step to be N = 10^5 we expect the laptop to go slower. Since a floating point number has 8 bytes and we have a matrix that stores NxN we'll have $8 \cdot 10^10$ bytes. That requires a severe amount of memory.