What should be included: Answers to all questions, as well as the proof that Jacobi converges for this problem, the Jacobi-method code, and the plot showing the unsatisfactory solution for few iterations with N = 5.

Lab 5 - Part 3

> Implement a python function for the Jacobi-method in your code from Lab 5, part 2. The function should take the matrix A the right-hand side, an integer number being the number of iterations (note, we are not using a more advanced stopping criteria here), and an initial guess L_0 and return the solution L_k .Hints: You can create the diagonal matrix D like this $D =$ `np.diagflat(np.diag(A))`. Other useful python functions are `numpy.tril(), numpy.triu()` and `numpy.matmul()`. Note that `numpy.tril()` and `numpy.triu()` includes the diagnoal of the original matrix, so for your purposes you need to subtract D . Include the code in your hand in.
>
> Use your function for solving the ODE-BVP from Lab 5 part 2, for N=5 and a maximum number of iterations equal to 100. Make sure that the result is close to the analytical solution (e.g. by looking at your plot).
>
> Decrease the number of Jacobi iterations until you can see in the plot that the solution is less accurate. How few iterations did you use? Include the plot in your hand in.
>
> Set N=10 instead. Do you need more iterations for the solution to be acceptable? How many? Note: It is possible to prove that the number of iterations depend on N

We have according to the method:

- $M = D$ where D is the diagonal matrix of A, so simply fetch the diagonal of A
- $K = L + U$ where L is the lower triangular part of A and U is the upper triangular part of A
- $-L, -U$ strictly lower and upper triangular parts of A respectively Can then write

$$x_k = D^{-1}(L + U)x_{k-1} + D^{-1}b$$

The function will take A, b, N and L_0 as input and return the solution L_k after N iterations.

We have

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & -2 & 1 \\ 2 & 1 & -4 \end{bmatrix}$$

In [177…

```
import numpy as np
import matplotlib.pyplot as plt
from math import pi
import time
from scipy.linalg import lu_factor, lu_solve


def assemble_A(N):
    h = (pi / 2) / (N + 1)
    A = np.zeros((N, N)) # Coefficient matrix of NxN dimension

    # main diag
    np.fill_diagonal(A, -2 + h**2)

    # sub diag, super diag
    # A[i, i-1] = 1 for i = 1 to N-1
    A[1:, :-1][np.eye(N-1, dtype=bool)] = 1
    # A[i, i+1] = 1 for i = 0 to N-2
    A[:-1, 1:][np.eye(N-1, dtype=bool)] = 1

    return A



def assemble_F(N, leftbc, rightbc):
    F = np.zeros(N)

    F[0] = -leftbc

    F[N - 1] = -rightbc
    return F
```

In [178…

```
def jacobi_method(A, b, L0, max_iterations):
    D = np.diagflat(np.diag(A))
    D_inv = np.linalg.inv(D)
    L =  np.tril(D - A)  # need to get rid of main diagonal
    U =  np.triu(D - A)  # same thing
    for i in range(max_iterations):
        L_next = D_inv @ (L + U) @ L0 + D_inv @ b
        L = L_next

    return L
```

In [179…

```
def ComputeAnalyticalSolution(N, leftbc, rightbc):
    x = np.linspace(0, (pi / 2), N + 2)
    y = np.cos(x) + 2* np.sin(x)
    return x, y
```

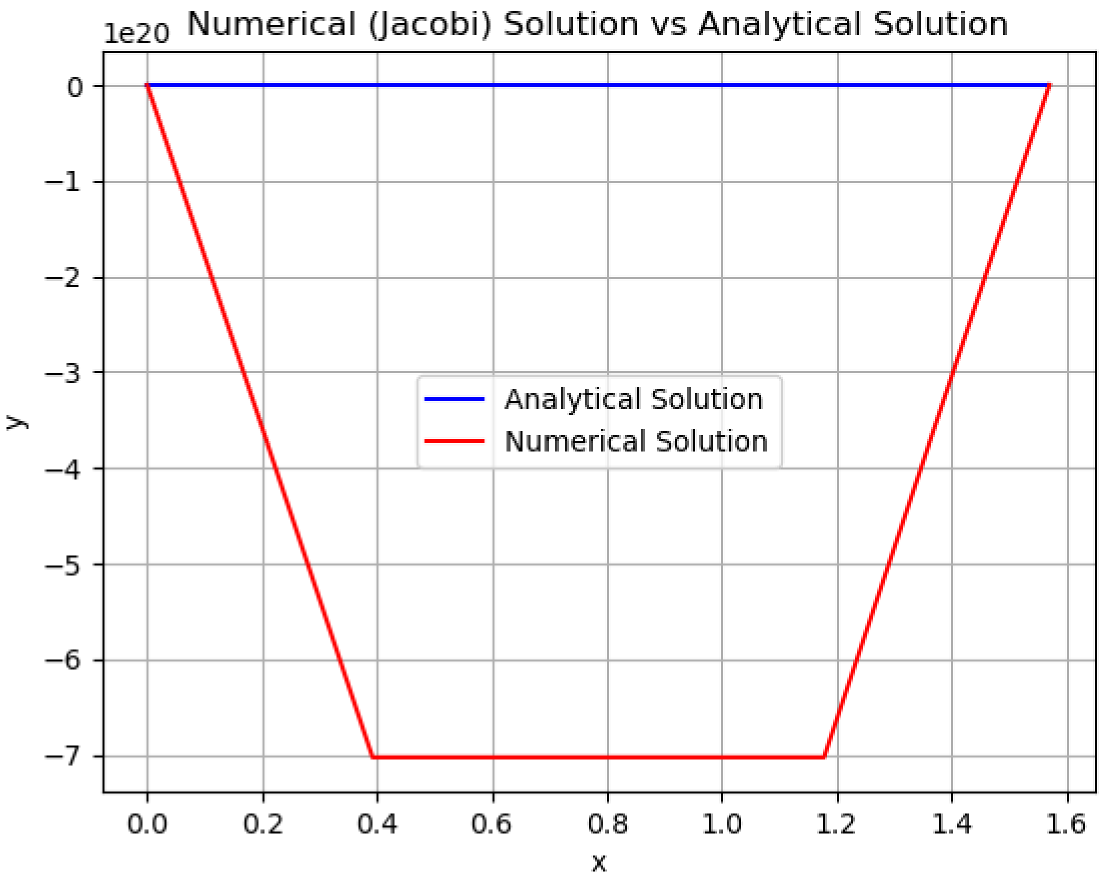
In [180…

```
def ComputeNumericalSolution(N, leftbc, rightbc):
    h = (pi / 2) / (N + 1)
    x = np.linspace(0, (pi / 2), N + 2)
    A = np.zeros((N, N)) # Coefficient matrix of nxn dimension
```

```
    F = np.zeros(N) # The result vector such that AU = F of size n
    # make initial a matrix of ones
    L0 = np.ones(N)
    A[0, 0] = -2 + h**2
    A[0, 1] = 1
    F[0] = -leftbc
    for i in range(1, N - 1):
        A[i, i - 1] = 1
        A[i, i] = -2 + h**2
        A[i, i + 1] = 1
        F[i] = 0
    A[N - 1, N - 2] = 1
    A[N - 1, N - 1] = -2 + h**2
    F[N - 1] = -rightbc

    y_h_int = jacobi_method(A, F, L0, 100)
    y_h = np.zeros(N + 2)
    y_h[0] = leftbc
    y_h[1:N + 1] = y_h_int
    y_h[-1] = rightbc
    return x, y_h, h
```

In [ ]:

```
N = 5
x = np.linspace(0, (pi / 2), N + 2)
leftbc = 1
rightbc = 2
y = np.cos(x) + 2* np.sin(x)

x_h, y_h, h = ComputeNumericalSolution(N, leftbc, rightbc)

# plot analytical and numerical
plt.plot(x, y, label='Analytical Solution', color='blue')
plt.plot(x_h, y_h, label='Numerical Solution', color='red')
plt.title('Numerical (Jacobi) Solution vs Analytical Solution')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend()
plt.show()
```


Numerical (Jacobi) Solution vs Analytical Solution
1e20
0
−1
−2
−3
−4
−5
−6
−7
y
Analytical Solution
Numerical Solution
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
x