Lab 5 - Part 2

Getting acquainted with LU-factorization

> Compute (with pen and paper) the LU factorization of the matrix, with partial pivoting
>
> $$A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & -2 & 1 \\ 2 & 1 & -4 \end{bmatrix}$$
>
> Include the derivation in your hand in. Check if $PA = LU$ to ensure that you did it correctly. Use scipy.linalg.lu_factor to perform the same calculation and check that it is the same. You don't need to include anything from this task in the hand in.

Here is the shortened solution for the LU factorization with partial pivoting:

Initial matrices:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad U = A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & -2 & 1 \\ 2 & 1 & -4 \end{bmatrix}$$

step1: Process first column.

- Pivoting: The largest absolute value in the first column is 2 (in R1 or R3). We choose U_{11} as the pivot. No row swap is performed.
- Elimination: Eliminate entries below U_{11}.
  - $R_2 \leftarrow R_2 - (1/2)R_1$. Store $l_{21} = 1/2$ in L.
  - $R_3 \leftarrow R_3 - (1)R_1$. Store $l_{31} = 1$ in L.

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & -3/2 & 1/2 \\ 0 & 2 & -5 \end{bmatrix}$$

step2: Process second column (from row 2 downwards).

- Pivoting: The elements are $(-3/2, 2)$. The largest absolute value is $|2|$ in R3. Swap R2 and R3.
  - Swap R2 and R3 in U.
  - Swap R2 and R3 in P.
  - Swap the multipliers in column 1 of L corresponding to rows 2 and 3 (L_{21} and L_{31}). After swap:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1/2 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 2 & -5 \\ 0 & -3/2 & 1/2 \end{bmatrix}$$

- Elimination: Eliminate the entry below the new pivot $U_{22} = 2$.
  - $R_3 \leftarrow R_3 - (-3/4)R_2$. Store $l_{32} = -3/4$ in L.

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1/2 & -3/4 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 2 & -5 \\ 0 & 0 & -13/4 \end{bmatrix}$$

The final matrices are:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1/2 & -3/4 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 2 & -5 \\ 0 & 0 & -13/4 \end{bmatrix}$$

Code

Consider the BVP that you worked in in the last lab, but with a time-dependent boundary condition

$$\begin{aligned} &y''(x) + y(x) = 0 \\ &y(x = 0) = 1 \\ &y(x = \pi/2) = 2\sin(\pi t) \end{aligned}$$

You will now solve this for $\iota = t_1, \iota_2, \iota_3, \ldots \iota_{M-1}, t_M = 1$, for $\mathrm{M} = 100$

1. Start by moving all code (from the previous lab) regarding the assembly of A, into its own subroutine. The subroutine should return A and have N as an input value. Also move all code regarding the assembly of the right hand side into another subroutine, which should take N and the boundary values as an input, and return F .
2. Create a for-loop looping over the time-values t_i. You can assemble A outside the for-loop, but you need to assemble the right hand side and solve the system inside the for-loop.
3. Set $N = 1000$ and measure the time it takes for the for-loop to run, e.g. by using timeit like below

```
{python}
import timeit
starttime=timeit.default_timer()
#your code...
print('solvetime is'+str(endtime-starttime))
```

4. Now utilize the power of LU-factorization. You can instead of using numpy. linalg. solve use scipy.linalg.lu_factor followed by scipy.linalg.lu_solve. The function

scipy.linalg.lu_solve takes care of the forward and backward substitution step. Think about if you have to put scipy.linalg.lu_factor inside or outside the loop.

5. Time your lu-version of the code. How does it compare to the timing of numpy. linalg .solve? Explain your result.
6. What is numpy.linalg.solve actually doing under the hood? Google.

In [22]:
```
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

    # Discretized equation: y_{i-1} + (-2 + h^2)y_i + y_{i+1} = 0
    # The matrix has 1 on the sub-diagonal and super-diagonal,
    # and (-2 + h^2) on the main diagonal.

    # 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) # The result vector such that AU = F of size N

    # Based on: y_{i-1} + (-2 + h^2)y_i + y_{i+1} = 0
    # Equation for i=1 (first interior point): y_0 + (-2 + h^2)y_1 + y_2
= 0
    # y_0 is the left boundary condition
    # F[0] = -y_0 = -leftbc
    F[0] = -leftbc

    # Equation for i=N (last interior point): y_{N-1} + (-2 + h^2)y_N +
y_{N+1} = 0
    # y_{N+1} is the right boundary condition
    # F[N-1] = -y_{N+1} = -rightbc
    F[N - 1] = -rightbc

    # For intermediate points (i=2 to N-1), y_{i-1} + (-2 + h^2)y_i +
y_{i+1} = 0
    # The right side is 0, which is already the default value in F

    return F

leftbc = 1
N = 1000  # Number of interior points for timing (as requested)
```

```
M = 100   # Number of time steps
time_points = np.linspace(0, 1, M)
t = time_points
rightbc = 2 * np.sin(pi * t)  # The right boundary condition y(pi/2)
will be 2 * sin(pi * t)
```

In [23]:
```
print("Timing with np.linalg.solve (N={})".format(N))
starttime = time.time()

A = assemble_A(N) # Assemble A matrix once outside the time loop

# Solve the system for each time step
for t in time_points:
    current_rightbc = 2 * np.sin(pi * t)
    F = assemble_F(N, leftbc, current_rightbc) # Assemble F for current
time
    y_h_int = np.linalg.solve(A, F) # Solve the system A * y_h_int = F

endtime = time.time()
print('Solve time (np.linalg.solve) is ' + str(endtime - starttime) + '
seconds')
```

```
Timing with np.linalg.solve (N=1000)
Solve time (np.linalg.solve) is 11.845492124557495 seconds
```

In [24]:
```
print("\nTiming with scipy.linalg.lu_factor/lu_solve (N={})".format(N))
starttime = time.time()

# Factorize A once outside the time loop. A is independent of time.
lu, piv = lu_factor(A)

# Solve the system for each time step using the LU factors
for t in time_points:
    current_rightbc = 2 * np.sin(pi * t) # Calculate the time-dependent
BC
    F = assemble_F(N, leftbc, current_rightbc) # Assemble F for current
time
    y_h_int = lu_solve((lu, piv), F) # Solve using LU factors

endtime = time.time()
print('Solve time (scipy.linalg.lu_factor/lu_solve) is ' + str(endtime -
starttime) + ' seconds')
```

```
Timing with scipy.linalg.lu_factor/lu_solve (N=1000)
Solve time (scipy.linalg.lu_factor/lu_solve) is 0.26235413551330566 second
s
```

np.linalg.solve performs a factorization (like LU) followed by substitution internally for each call. When used repeatedly with the same matrix but different right-hand sides, it recomputes the expensive factorization step every time. By using scipy.linalg.lu_factor once outside the loop, we avoid these repeated factorizations, performing only the much faster substitution steps inside the loop using scipy.linalg.lu_solve.