Lista 2

```
import numpy as np
from scipy import linalg


def eliminacja_gaussa_z_wyborem(A, b):
  n = len(b)
  x = np.zeros(n)

  # Eliminacja współczynników
  for k in range(n-1):
    # Wybór elementu podstawowego
    max_index = k + np.argmax(np.abs(A[k:, k]))
    if max_index != k:
      A[[k, max_index]] = A[[max_index, k]]
      b[[k, max_index]] = b[[max_index, k]]

    for i in range(k+1, n):
      factor = A[i, k] / A[k, k]
      A[i, k:] -= factor * A[k, k:]
      b[i] -= factor * b[k]

  # Podstawianie wstecz
  x[n-1] = b[n-1] / A[n-1, n-1]
  for i in range(n-2, -1, -1):
    x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]

  return x
```

zadanie 4

```
A = np.array([[0,0,2,1,2],
              [0,1,0,2,-1,],
              [1,2,0,-2,0],
              [0,0,0,-1,1],
              [0,1,-1,1,-1]], dtype=float)
b4 = np.array([[1], [1], [-4], [-2], [-1]], dtype=float) # do rozwiązania funkcją wbudowaną
b = np.array([1, 1, -4, -2, -1], dtype=float) # do rozwiązania zaimplementowaną funkcją
x4 = np.linalg.solve(A, b) # sprawdzamy wynik dla funkcji wbudowanej
x4
```

```
array([ 2., -2.,  1.,  1., -1.])
```

```
x = eliminacja_gaussa_z_wyborem(A,b) # porównujemy z funkcją zaimplementowaną
x
```

```
array([ 2., -2.,  1.,  1., -1.])
```

Jak widzimy, obie funkcje (wbudowana i zaimplentowana) dają takie same rozwiązanie.

zadanie 5

```
# Punkty, przez które przechodzi wielomian
punkty = np.array([[0, -1], [2, 2], [3, 3], [1, 4], [6, -2]], dtype=float)

# Macierz współczynników i wektor wyrazów wolnych
A = np.zeros((5, 5))
b = punkty[:, 1]
# wpisujemy punkty, przez które przechodzi wielomian jako macierz
for i in range(5):
  for j in range(5):
    A[i, j] = punkty[i, 0]**j

# Rozwiązanie układu równań
wspolczynniki = eliminacja_gaussa_z_wyborem(A.copy(), b.copy())

# Wypisanie wielomianu
print("Wielomian:")
print("y =", wspolczynniki[0], "+", wspolczynniki[1], "* x +", wspolczynniki[2], "* x^2 +", wspolczynniki[3], "* x^3 +", wspolczynniki[4
```

```
Wielomian:
y = -1.0 + 13.933333333333337 * x + -12.350000000000001 * x^2 + 3.7666666666666666 * x^3 + -0.35000000000000003 * x^4
```

```
# sprawdzamy, czy wielomian jest odpowiedni (+/- błędy numeryczne)
def y(x):
  return -1.0 + 13.933333333333337 * x + -12.350000000000001 * x**2 + 3.7666666666666666 * x**3 + -0.35000000000000003 * x**4
print(y(0), y(2), y(3), y(1), y(6))
```

```
-1.0 2.000000000000001 3.000000000000007 4.000000000000003 -2.0
```

Punkty podane w poleceniu pokrywają się z wyliczonymi zaimplementowanym wielomianem (oprócz błędów numerycznych).

```
x5 = np.linalg.solve(A.copy(), b.copy())
x5
```

```
array([ -1.        ,  13.93333333, -12.35      ,   3.76666667,
        -0.35      ])
```

Takie same (w przybliżeniu) wyniki uzyskujemy korzystając z wbudowanej funkcji.

zadanie 6

```
A = np.array([[3.50, 2.77, -0.76, 1.80],
             [-1.80, 2.68, 3.44, -0.09],
             [0.27, 5.07, 6.90, 1.61],
             [1.71, 5.45, 2.68, 1.71]], dtype = float)
b = np.array([7.31, 4.23, 13.85, 11.55], dtype = float)
x = eliminacja_gaussa_z_wyborem(A, b)
x6 = np.linalg.solve(A.copy(), b.copy())
det_A6 = np.linalg.det(A)
Ax6 = np.dot(A, x)

print('Rozwiązanie x:\n', x, '\nRozwiązanie funkcją wbudowaną:\n', x6)
print("Det(A):", det_A6)
print("Dokładność Ax ≈ b: \n", Ax6, "≈\n", b)
print(f"różnica: {Ax6 - b}")
```

```
Rozwiązanie x:
 [1. 1. 1. 1.]
Rozwiązanie funkcją wbudowaną:
 [1. 1. 1. 1.]
Det(A): 0.22579734000000876
Dokładność Ax ≈ b:
 [ 7.31000000e+00  1.32860857e+01 -3.24001100e+00 -4.69032638e-03] ≈
 [ 7.31000000e+00  1.32860857e+01 -3.24001100e+00 -4.69032638e-03]
różnica: [8.8817842e-16 0.0000000e+00 0.0000000e+00 0.0000000e+00]
```

Rozwiązanie jest bardzo dokładne - widzimy różnicę rzędu 10^{-16}.

zadanie 7

```
A = np.array([[10,-2,-1, 2, 3, 1,-4, 7],
              [5, 11, 3,10,-3, 3, 3,-4],
              [7, 12, 1, 5, 3,-12,2, 3],
              [8, 7, -2, 1, 3, 2, 2, 4],
              [2,-15,-1, 1, 4,-1, 8, 3],
              [4,  2, 9, 1,12,-1, 4, 1],
              [-1, 4, -7, -1, 1, 1, -1, -3],
              [-1, 3, 4, 1, 3, -4, 7, 6]], dtype = float)
x = np.array([[0], [12], [-5], [3], [-25], [-26], [9], [-7]], dtype = float)
x7 = np.array([0, 12, -5, 3, -25, -26, 9, -7], dtype = float)
eliminacja_gaussa_z_wyborem(A, x7)
```

```
array([-1.,  1., -1.,  1., -1.,  1., -1.,  1.])
```

Sprawdzając ręcznie widzimy, że rozwiązanie jest poprawne.

zadanie 8

```
def inverse_matrix(A, tol = 1e-10):
    det = np.linalg.det(A)
    if not np.isclose(det, 0, atol=tol):
        I = np.eye(A.shape[0])
        return np.column_stack([eliminacja_gaussa_z_wyborem(A.copy(), I[:, i]) for i in range(len(I[1]))])
    else:
        print("Macierz nie jest odwracalna (wyznacznik ≈ 0 w zadanej tolerancji).")
        return False
```

```
A = np.array([[2, -1, 0, 0, 0, 0],
              [-1, 2, -1, 0, 0, 0],
              [0, -1, 2, -1, 0, 0],
              [0, 0, -1, 2, -1, 0],
              [0, 0, 0, -1, 2, -1],
              [0, 0, 0, 0, -1, 5]], dtype = float)

# bierzemy pod uwagę układ równań A*x_i = i-ta kolumna macierzy I
# rozwiązujemy zadanie w etapach, obliczając każdą kolumnę osobno i potem łącząc kolumny w macierz
A_inv = inverse_matrix(A)
print(A_inv)
```

```
[[0.84 0.68 0.52 0.36 0.2  0.04]
 [0.68 1.36 1.04 0.72 0.4  0.08]
 [0.52 1.04 1.56 1.08 0.6  0.12]
 [0.36 0.72 1.08 1.44 0.8  0.16]
 [0.2  0.4  0.6  0.8  1.   0.2 ]
 [0.04 0.08 0.12 0.16 0.2  0.24]]
```

```
np.round(A @ A_inv, decimals=3)
```

```
array([[ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1., -0.,  0.,  0.,  0.],
       [-0., -0.,  1.,  0.,  0., -0.],
       [ 0.,  0., -0.,  1.,  0.,  0.],
       [-0., -0.,  0.,  0.,  1.,  0.],
       [-0., -0.,  0., -0., -0.,  1.]])
```

Jak widać, dostajemy macierz odwrotną i sprawdzając AA^{-1} ostrzymujemy I (bierzemy poprawkę na błędy numeryczne - w macierzy odwrotnej występują problematyczne ułamki).

Sprawdzamy trójdiagonalność - patrząc na macierz A^{-1} możemy stwierdzić, że nie jest trójdiagonalna. Drugim spodobem jest napisanie funkcji w pythonie.

```
def is_tridiagonal(A, tol=1e-10): # tolerancja błędu dla błędów numerycznych
    n = A.shape[0]
    for i in range(n):
        for j in range(n):
            # sprawdzamy tylko elementy poza główną przekątną i sąsiadującymi przekątnymi
            if abs(i - j) > 1 and not np.isclose(A[i, j], 0, atol=tol):
                return False
    return True
print("Czy macierz A^-1 jest trójdiagonalna?", is_tridiagonal(A_inv))
print("Czy macierz A jest trójdiagonalna?", is_tridiagonal(A))
```

```
Czy macierz A^-1 jest trójdiagonalna? False
Czy macierz A jest trójdiagonalna? True
```

Mimo, że macierz A jest trójdiagonalna, macierz odwrotna (A^{-1}) nie ma takich właściwości.

zadanie 9

```
A = np.array([
    [1, 3, -9, 6, 4],
    [2, -1, 6, 7, 1],
    [3, 2, -3, 15, 5],
    [8, -1, 1, 4, 2],
    [11, 1, -2, 18, 7]
], dtype=float)
I = np.eye(A.shape[0])
A_inv = inverse_matrix(A)
print(A_inv)
```

```
Macierz nie jest odwracalna (wyznacznik ≈ 0 w zadanej tolerancji).
False
```

Wyznacznik jest równy lub bliski zera → macierz jest nieodwracalna, lub źle uwarunkowana - wtedy nawet jeśli obliczymy macierz "odwrotną", $AA^{-1} \neq I$

```
A_inv = np.column_stack([eliminacja_gaussa_z_wyborem(A.copy(), I[:, i].copy()) for i in range(len(I[0]))])
A@A_inv
```

```
array([[ 0.  ,  1.  ,  1.  ,  0.  ,  0.  ],
       [ 0.  ,  0.25, -0.25,  0.  ,  0.  ],
       [-1.  ,  0.25,  2.25,  0.  ,  0.  ],
       [-1.  , -0.5 ,  0.  ,  1.  , -1.  ],
       [-1.  ,  0.75,  0.75,  0.  ,  0.  ]])
```

Jak widzimy, obliczona na siłę macierz "odwrotna", nie jest prawdziwą macierzą A^{-1}, jakość rozwiązania naszego problemu pozostawia wiele do życzenia.