Zadanie 4:

3. Dana jest macierz

$$\mathbf{A} = \begin{bmatrix} \frac{19}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & -\frac{17}{12} \\ \frac{13}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & -\frac{11}{12} & \frac{13}{12} \\ \frac{5}{6} & \frac{5}{6} & \frac{5}{6} & -\frac{1}{6} & \frac{5}{6} & \frac{5}{6} \\ \frac{5}{6} & \frac{5}{6} & -\frac{1}{6} & \frac{5}{6} & \frac{5}{6} & \frac{5}{6} \\ \frac{13}{12} & -\frac{11}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & \frac{13}{12} \\ -\frac{17}{12} & \frac{13}{12} & \frac{5}{6} & \frac{5}{6} & \frac{13}{12} & \frac{19}{12} \end{bmatrix}. \tag{4}$$

Przy użyciu metody potęgowej znajdź jej dwie największe na moduł wartości własne i odpowiadające im wektory własne.

4. Sprowadź macierz z zadania 3 do postaci trójdiagonalnej, a następnie znajdź jej wszystkie wartości własne.

Kod w języku Python:

```python
import numpy
import copy


def householder(A):
    I = numpy.diag([1 for k in range(0, len(A))])

    for k in range(0, len(A) - 2):
        x = numpy.zeros((len(A), 1))
        y = numpy.zeros((len(A), 1))

        # wypełnianie wektora x
        for i in range(0, len(A)):
            x[i] = A[i][k]

        # wypełnianie pierwszej części wektora y
        for i in range(0, k + 1):
            y[i] = A[i][k]

        # wypełnienie nastepnego elementu normą wektora y począwszy od k+1
        norm = 0.0
        for i in range(k + 1, len(A)):
            norm = norm + A[i][k] ** 2
        y[k + 1] = numpy.sqrt(norm)


        # obliczanie normy norm=||x-y|| i top=x-y
        norm = 0.0
        top = numpy.subtract(x, y)
        for i in top:
            norm = norm + i ** 2
        norm = numpy.sqrt(norm)

        # w=x-y/(||x-y||)
```

```python
        w = top / norm

        # P=I-2w*wT
        P = numpy.subtract(I, 2 * numpy.matmul(w, w.transpose()))

        # A=PAP
        A = PAPmultiply(P, A, k)

    return A


# lekko zoptymalizowane mnożenie układu z transformacją householdera PAP
def PAPmultiply(P, A, k):
    B = copy.copy(A)
    for i in range(k, len(A)):
        for j in range(k, len(A)):
            elem = 0.0
            for z in range(0, len(A)):
                elem = elem + P[i][z] * A[z][j]
            B[i][j] = elem
    C = copy.copy(B)
    for i in range(k, len(A)):
        for j in range(k, len(A)):
            elem = 0.0
            for z in range(0, len(A)):
                elem = elem + B[i][z] * P[z][j]
            C[j][i] = elem
    return C


def givens(A):
    Q = numpy.diag([1.0 for k in range(0, len(A))])
    for i in range(0, len(A) - 1):
        # stworzenie macierzy diagonalnej
        G = numpy.diag([1.0 for k in range(0, len(A))])

        # jak we wzorze z wykładow i+1=j
        norm = numpy.sqrt(A[i][i] ** 2 + A[i + 1][i] ** 2)
        c = A[i][i] / norm
        s = A[i + 1][i] / norm
        G[i][i] = c
        G[i][i + 1] = s
        G[i + 1][i] = -s
        G[i + 1][i + 1] = c
        # G jest macierzą givensa więc mnożenie ma byc O(1)*n
        # A=GA
        A = GAmultiply(G, A, i)
        # tutaj tez mnożenie O(n)
        # Q=Q*GT
        Q = QGmultiply(Q, G.transpose(), i)
    return A, Q


# mnoży macierz A przez macierz Givensa-stała liczba operacji O(1)*len(A)
def GAmultiply(G, A, i):
    GA = copy.copy(A)
    for k in range(i, i + 2):
```

```
        for x in range(0, len(A)):
            GA[k][x] = G[k][i] * A[i][x] + G[k][i + 1] * A[i + 1][x]
    return GA


def QGmultiply(Q, G, i):
    QG = copy.copy(Q)
    for x in range(0, len(A)):
        QG[x][i] = Q[x][i] * G[i][i] + Q[x][i + 1] * G[i + 1][i]
        QG[x][i + 1] = Q[x][i] * G[i][i + 1] + Q[x][i + 1] * G[i + 1][i + 1]
    return QG


def qr_algorithm(A):
    while True:
        temp = A[0][0]
        R, Q = givens(A)
        A = numpy.matmul(R, Q)
        # warunek stopu
        if abs(abs(temp) - abs(A[0][0])) < 1e-8:
            return A


if __name__ == "__main__":
    print("Diagonalizacja macierzy symetrycznej a nastepnie zastosowanie 
algorytmu QR")
    A = numpy.array([
        [19 / 12, 13 / 12, 5 / 6, 5 / 6, 13 / 12, -17 / 12],
        [13 / 12, 13 / 12, 5 / 6, 5 / 6, -11 / 12, 13 / 12],
        [5 / 6, 5 / 6, 5 / 6, -1 / 6, 5 / 6, 5 / 6],
        [5 / 6, 5 / 6, -1 / 6, 5 / 6, 5 / 6, 5 / 6],
        [13 / 12, -11 / 12, 5 / 6, 5 / 6, 13 / 12, 13 / 12],
        [-17 / 12, 13 / 12, 5 / 6, 5 / 6, 13 / 12, 19 / 12]
    ])
    A = qr_algorithm(householder(A))
    print(numpy.around(A, decimals=3))
```

Omówienie rozwiązania:

Do rozwiązania powyższego zadania użyłem transformacji Householdera i algorytmu QR. Powyższa macierz jest symetryczna, dlatego transformacja Householdera na naszej macierzy da w wyniku macierz trójdiagonalną. Algorytm transformacji Householdera obliczania macierzy trójdiagonalnej:

Dopóki k jest mniejsze od len(A):

-wypełnij wektor x k-tą kolumną macierzy A

-wypełnij wektor y do k-tego miejsca k-tą kolumną macierzy A. Następne miejsce w wektorze będzie normą wektora począwszy od punktu k+1. Pozostałe punkty będą równe 0

$$w = \frac{x - y}{||x - y||}$$

$$P = I - 2ww^T$$

$$A = PAP$$

Następnie aby obliczyć wartości własne skorzystałemz algorytmu QR, którego złożoność dla macierz trójdiagonalnej wynosi O(n) na jeden krok. Skorzystałem w algorytmie z obrotów Givensa do obliczenia macierzy Q i R potrzebnych do algorytmu. Obroty Givensa zwiększają złożoność algorytmu tak, że łącznie mamy już O(n^2). Pojedynczy krok ma złożoność O(1), lecz musimy go wykonać n razy

$$G_{n-1}...G_2G_1 = R$$

$$Q = G_1^T G_2^T ... G_{n-1}^T$$

$$A = QR$$

Mnożąc w każdym kroku macierz Q razy R otrzymujemy macierz A, która będzie zmierzać do postaci diagonalnej na której otrzymamy wartości własne naszej macierzy początkowej.

Wyniki i omówienie:

```
Diagonalizacja macierzy symetrycznej a nastepnie zastosowanie algorytmu QR
[[ 4.  0. -0.  0. -0.  0.]
 [ 0.  3.  0. -0. -0. -0.]
 [-0.  0. -2.  0.  0.  0.]
 [ 0. -0.  0. -1.  0.  0.]
 [-0. -0.  0.  0.  2. -0.]
 [ 0.  0.  0. -0. -0.  1.]]
```

Obliczone wartości własne z programu: [-2,-1,1,2,3,4]

Powyższy sposób obliczania wartości własnych jest lepszym i szybszym sposobem niż bezpośrednie obliczanie wartości własnych z równania

$$det(A - I\lambda) = 0$$

Obliczenie wyznacznika macierzy a następnie szukanie miejsc zerowych jest kosztowniejsze(O(n!) dla obliczenia wyznacznika metodą Laplace'a) od algorytmu QR. W podanej macierzy wystarczyło dokonać transformacji Householdera, która wymaga O(n^2) operacji, a następnie zastosować algorytm QR, który wymaga O(n^2) operacji co jest zdecydowanie lepszą złożonością niż O(n!).