METODY NUMERYCZNE

ZADANIE 4

Artur Guniewicz

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

Metoda: transformacja Householdera + algorytm QR + obroty Givensa

Transformacja Householdera na macierzy symetrycznej da w wyniku macierz trójdiagonalną, a macierz z zadania 3. taka właśnie jest.

Algorytm wygląda następująco:

1. Dopóki k < 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

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

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

$$A = PAP$$

Algorytm QR służy do obliczenia wartości własnych. W czasie O(n) na jeden krok można go wykonać dla macierzy trójdiagonalnej. Do obliczenia macierzy Q i R można użyć obrotów Givensa (które jednak zwiększają złożoność algorytmu do O(n^2)). Jeden krok ma złożoność O(1) ale należy go wykonać n razy.

$$G_{n-1} ... G_2 G_1 = R$$

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

$$A = QR$$

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.

Kod programu:

```
#
#*************************************************************
#*                                                           *
#*               Artur Guniewicz - Zadanie 4                 *
#*                                                           *
#*************************************************************
#

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 następnego elementu norma 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)
```

```python
        for i in top:
            norm=norm+i**2
        norm=numpy.sqrt(norm)

        # w=x-y/(||x-y||)
        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 transformacja 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 wykładów 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 być O(1)*n
        # A=GA
        A=GAmultiply(G,A,i)

        # tutaj też 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 następnie 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))
```

Kompilacja:

python3 Zadanie4.py

Wyniki:

```
Diagonalizacja macierzy symetrycznej, a następnie 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. ]]
```

Wartości własne: [-2,-1,1,2,3,4]

Komentarz:

Możliwe jest róœnież obliczenie wartości własnych z równania:

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

Jednak jest to mniej efektywny sposób.

Zastosowany algorytm jest algorytmem iteracyjnym, co oznacza, że dla dostatecznie dużej ilości kroków można otrzymać wyniki z dużą dokładnością.

Program został napisany w języku Python3.