HW1

금융공학 20173855
박완배

Thomas algorithm 을 이용하여 다음 두 연립방정식의 해를 구하였다.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 2 & 1 & 3 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 3 \\ 13 \\ 12 \\ 7 \end{pmatrix} \dots (\text{식 } 1)$$

$$\begin{pmatrix} 1 & 1.0000001 & 0 & 0 \\ 2.0000001 & 2 & 2.0000001 & 0 \\ 0 & 3.0000001 & 3 & 3.0000001 \\ 0 & 0 & 4.0000001 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 7.0000003 \\ 26.000001 \\ 48.000001 \\ 52.0000006 \end{pmatrix} \dots (\text{식 } 2)$$

(식 1)은 흔히 접하는 일반적인 연립방정식이나, (식 2)는 대각항과 대각항 양 옆의 항의 값 차이가 작은 것이 그 특징이다. (식 1)과 (식 2)를 직접 계산하였을 때 방정식의 해는 각각 $\mathbf{x_1} = (1, 2, 3, 4), \mathbf{x_2} = (4, 3, 6, 7)$이다. 즉, Thomas algorithm 을 이용하였을 경우, 이와 같은 결과가 도출되어야 한다. 프로그램을 이용하여 구한 결과는 다음과 같다.

(식 1)

```
------------------------------------------------------------------------
Solution of equation #1
x: [ 1.  2.  3.  4.]
Ax: [  3.  13.  12.   7.]
------------------------------------------------------------------------
```

(식 2)

```
------------------------------------------------------------------------
Solution of equation #2
x: [ 42.99999599 -35.99999999   6.00000205  45.99999795]
Ax: [  7.0000003   26.000001    48.00000103  52.0000006 ]
------------------------------------------------------------------------
```

(식 1)의 경우 직접 구한 해와 프로그램으로 구한 해가 일치하지만, (식 2)의 경우 직접 구한 해와 프로그램으로 구한 해가 다르다. 이는 (식 2)에서의 소거 과정에서 작은 수로 나누는 과정이 포함되는데, 이러한 경우 계산 과정에서의 절삭오차가 해를 구하는데 있어 큰 영향을 주기 때문이다. 이는 각 행의 원소 값이 상당히 비슷하기 때문에 발생하는 현상이다. 따라서 Thomas algorithm 을 이용하여 연립방정식의 해를 구하는 경우 이 점을 유의해야 한다.

<Python Code>[1]

```
import numpy as np
import copy

def tridiagonalSolver(a_, d_):
    #Deep copy
    a = copy.deepcopy(a_)
    d = copy.deepcopy(d_)

    #Tridiagonal Solver: Ax = d
    if len(a) != len(d):
        raise IndexError("Dimension of A and d should be equal")

    for i in range(len(a) - 1):
        d[i+1] += (-a[i+1][i] / a[i][i]) * d[i]
        a[i+1] += (-a[i+1][i] / a[i][i]) * a[i]

    x = np.zeros(len(d))
    x[-1] = d[-1] / a[-1][-1]
    for i in range(len(a) - 2, -1, -1):
        x[i] = (d[i] - a[i][i+1] * x[i+1]) / a[i][i]

    return x

if __name__ == "__main__":
    a1 = np.array([[1.0, 1.0, 0.0, 0.0],
                   [2.0, 1.0, 3.0, 0.0],
                   [0.0, 1.0, 2.0, 1.0],
                   [0.0, 0.0, 1.0, 1.0]])
    d1 = np.array([3.0, 13, 12, 7])

    #Diagonal term과 양 옆의 term 차이가 크지 않은 경우
    a2 = np.array([[1.0000001, 1.0, 0.0 ,0.0],
                   [2.0000001, 2.0, 2.0000001, 0.0],
                   [0.0, 3.0000001, 3.0, 3.0000001],
                   [0.0, 0.0, 1.0000001, 1.0]])
    d2 = np.array([7.0000003, 26.000001, 48.000001, 52.0000006])

    x1 = tridiagonalSolver(a1, d1)
    print('-'*70)
    print("Solution of equation #1")
    print("x:", x1)
    y1 = np.dot(a1, x1)
    print("Ax:", y1)
    print('-'*70)

    print("Solution of equation #2")
    x2 = tridiagonalSolver(a2, d2)
    print("x:", x2)
    y2 = np.dot(a2, x2)
    print("Ax:", y2)
    print('-'*70)
```

[1] 입력받는 행렬이 tridiagonal matrix 인지에 대한 검사가 없는 것이 본 알고리즘의 한계로 판단됨