Rachunek Macierzowy i Statystyka Wielowymiarowa
Sprawozdanie 2
Eliminacja Gaussa i LU Faktoryzacja

Adam Staniszewski
Przemysław Węglik

27 marca 2024

Spis treści

1 Eliminacja Gaussa

1.1 Algorytm eliminacji Gaussa bez pivotingu

1.1.1 Opis algorytmu

Algorytm eliminacji Gaussa polega na przekształceniu macierzy do postaci górnotrójkątnej poprzez iteracyjne zerowanie elementów pod przekątną w kolejnych kolumnach. W każdym wierszu jest wybierany element główny, w wariancie bez pivotingu jest to pierwszy element niezerowy. Na jego podstawie wykonywane są przekształcenia, które mają wyzerować pozostałe elementy w kolumnie.

Algorytm składa sięz dwóch części. Pierwsza zaczyna się od podzielenia wartości w pierwszym rzędzie macierzy **A** i pierwszego elementu wektor **b**, tak aby wartość pierwszego elementi na przekątnej **A** wyniosła 1. Kolejne wiersze są redukowane o wartość wyliczoną na podstawie pierwszych elementów poprzedniego i aktualnego wiersza. W drugiej fazie algorytm iteruje od ostatniego wiersza macierzy i odejmuje od kolejnych wierszy odpowiednio przeskalowane poprzednie wiersza, tak aby wyzerować wartości pod przekątną.

1.1.2 Pseudokod

```
function gauss_elimination(A, b):
    n = length(A)

    for i from 0 to n - 1:
        divisor = A[i, i]
        A[i] /= divisor
        b[i] /= divisor

        for j from i + 1 to n - 1:
            multiplier = A[j, i]
            A[j] -= multiplier * A[i]
            b[j] -= multiplier * b[i]

    for i from n - 1 to 0 step -1:
        for j from 0 to i - 1:
            multiplier = A[j, i]
            A[j] -= multiplier * A[i]
            b[j] -= multiplier * b[i]

    return b
```

1.1.3 Implementacja algorytmu w Pythonie

```python
def gauss_elimination(A, b):
    n = len(A)

    for i in range(n):
        divisor = A[i, i]
        A[i] /= divisor
        b[i] /= divisor

        for j in range(i + 1, n):
```

```
                multiplier = A[j, i]
                A[j] -= multiplier * A[i]
                b[j] -= multiplier * b[i]

    for i in range(n - 1, -1, -1):
        for j in range(i):
            multiplier = A[j, i]
            A[j] -= multiplier * A[i]
            b[j] -= multiplier * b[i]

    return b
```

1.1.4 Analiza złożoności

Algorytm wykonuje dzielenie dla wszytkich wierszy macierzy kwadratowej **A** $(n-i*n-i)$ i wszystkich elementów wektora **b** $(n-i)$. Do wszystkich elementów **A** i **b** musi zostać dodany współczynnik, do którego wyliczenia jest potrzeba operacja mnożenia. Mamy zatem $(n-1)^2$ operacji mnożenia i dodawania dla A oraz $n-1$ dla b. Sumarycznie mamy zatem:

$$\text{Operacje mnożenia} = \sum_{i=1}^{n-1}((n-i)^2 + 2(n-i)) \tag{1}$$

$$\text{Operacje dodawania} = \sum_{i=1}^{n-1}((n-i)^2 + (n-i)) \tag{2}$$

Wiedząc, że:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{3}$$

oraz:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{4}$$

mamy:

$$\text{Operacje mnożenia} = \frac{2n^3 + 3n^2 - 5n}{6} \tag{5}$$

$$\text{Operacje dodawania} = \frac{n^3 - n}{3} \tag{6}$$

sumarycznie mamy:

$$\text{Złożoność} = \frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6} \tag{7}$$

Rząd złożoności wynosi zatem $O(n^3)$

1.2 Algorytm eliminacji Gaussa z pivotingiem

1.2.1 Opis algorytmu

Zastosowanie **pivotingu** w algorytmie eleminacji Gaussa ma na celu zredukowanie powstających błędów numercznych. W tym wariancie elementem głównym dla każdego wiersza jest elementu o największej wartości bezwględnej. Jeśli ten element nie znajduje się na przekątnej, to zawierająca go kolumna zostanie przeniesiona w odpowiednie miejsce.

1.2.2 Pseudokod

```
function gauss_elimination_pivoting(A, b):
n = size(A, 1)
pivots = []

for i = 1 to n do:
    pivot_row = max(abs(A[i:, i])) + i - 1
    if pivot_row != i then:
        pivots.append((i, pivot_row))
        swap rows A[i] and A[pivot_row]
        swap elements b[i] and b[pivot_row]

    divisor = A[i, i]
    A[i] /= divisor
    b[i] /= divisor

    for j = i + 1 to n do:
        multiplier = A[j, i]
        A[j] -= multiplier * A[i]
        b[j] -= multiplier * b[i]

for i = n downto 1 do:
    for j = i - 1 downto 1 do:
        multiplier = A[j, i]
        A[j] -= multiplier * A[i]
        b[j] -= multiplier * b[i]

for (i, pivot_row) in reversed(pivots) do:
    swap rows A[i] and A[pivot_row]
    swap elements b[i] and b[pivot_row]

return b
```

1.2.3 Implementacja algorytmu w Pythonie

```python
def gauss_elimination_pivoting(A, b):
    n = len(A)
    pivots = []

    for i in range(n):
        pivot_row = np.argmax(np.abs(A[i:, i])) + i
        if pivot_row != i:
            pivots.append((i, pivot_row))
            A[[i, pivot_row]] = A[[pivot_row, i]]
            b[[i, pivot_row]] = b[[pivot_row, i]]

        divisor = A[i, i]
        A[i] /= divisor
        b[i] /= divisor
```

```
        for j in range(i + 1, n):
            multiplier = A[j, i]
            A[j] -= multiplier * A[i]
            b[j] -= multiplier * b[i]

    for i in range(n - 1, -1, -1):
        for j in range(i):
            multiplier = A[j, i]
            A[j] -= multiplier * A[i]
            b[j] -= multiplier * b[i]

    for i, pivot_row in pivots[::-1]:
        A[[i, pivot_row]] = A[[pivot_row, i]]
        b[[i, pivot_row]] = b[[pivot_row, i]]

    return b
```

1.2.4 Analiza złożoności

Wariant z pivotingiem posiada dwie modyfikacje dotyczące złożoności obliczeniowej. Po pierwsze, szukamy elementu głównego, na co potrzebujemy maksymalnie n operacji. Po drugie, kolejność rzędów musi zostać zamieniona, aby główny element zawsze był pierwszym elementem niezerowym. Zamiana jest operacją o stałej złożoności. Rząd złożoności obliczeniowej nadal wynosi więc $O(n^3)$.

1.2.5 Porównanie z Numpy

```
# gauss elimination
[-1.22364004  1.2599961   2.48610013 -0.76265502]
# gauss elimination with pivoting
[-1.22364004  1.2599961   2.48610013 -0.76265502]
# numpy.linalg.solve
[-1.22364004  1.2599961   2.48610013 -0.76265502]

# checking using numpy.allclose
Are the results close: True
Are the results close: True
```

Algorytm zwraca identyczne wyniki równania jak funkcji biblioteczna. Jedyne czego można by się dosuzkiwać to różna dokładność numeryczna, ale to róœnież zweryfikowano i wyniki są równe z dokładnością 10^{-8}

2 LU Faktoryzacja

2.1 Algorytm LU faktoryzacji bez pivotingu

2.1.1 Opis algorytmu

Faktoryzacja LU polega na dekompozycji macierzy kwadratowej na iloczyn dwóch macierzy trójkątnych - dolnej (L) i górnej (U). Macierz **L** jest inicjowana jako jednostokowa, a **U** jako kopia maciery **A**. Algorytm modyfikuje macierz **L**, poprzez ustawianie wartości L[j, i] na iloraz U[j, i] i U[i, i], a od **U[i][i:]** odejmuje iloczyn **L[j][i]** i **U[i][i:]**.

2.1.2 Pseudokod

```
function lu_decomposition(A):
    n = size(A)
    L = identity_matrix(n)
    U = A

    for i = 0 to n-1:
        for j = i+1 to n-1:
            L[j][i] = U[j][i] / U[i][i]
            for k = i to n-1:
                U[j][k] = U[j][k] - L[j][i] * U[i][k]

    return L, U
```

2.1.3 Implementacja w Pythonie

```python
def lu_decomposition(A):
    n = len(A)
    L = np.eye(n)
    U = A.copy()

    for i in range(n):
        for j in range(i + 1, n):
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]

    return L, U
```

2.1.4 Analiza złożoności

Eliminacja wartości z pierwszej kolumny wymaga n operacji dodawania i n operacji mnożenia dla $n-1$ elementów. Mamy zatem $2n(n-1)$ operacji dla pierwszej kolumny. Dla drugiej kolumny wykonujemy $2(n-i)(n-i+1)$ operacji. Łączna ilość operacji do wykonania wynosi zatem:

$$\sum_{i}^{n} 2(n-i)(n-i+1) \tag{8}$$

$$2\sum_{i}^{n}(n-i)(n-i) + (n-1) \tag{9}$$

Poprzez analogię do równania 4, możemy wywnioskować, że przybliżona złożoność będzie wynosić $\frac{2n^3}{3}$. Złożoność obliczeniowa jest więc rzędu $O(n^3)$.

2.2 Algorytm LU faktoryzacji z pivotingiem

2.2.1 Opis algorytmu

Faktoryzacja z pivotingiem stosuje inny sposób wyboru elementu głównego każdego wiersza. Analogicznie do Eliminacji Gaussa, wybieramy element o największej wartości bezwzględnej z każdego wiersza każdej macierzy, a następnie zamieniamy wiersze macierzy tak, aby główne elementy znalazły się na przekątnej.

2.2.2 Pseudokod

```
function lu_decomposition_pivoting(A):
n = length(A)
    L = zeros(n, n)
    U = copy(A)
    P = eye(n)

    for i from 1 to n do:
        pivot_row = argmax(abs(U[i:, i])) + i
        if pivot_row != i then:
            swap rows U[i] and U[pivot_row]
            swap rows L[i] and L[pivot_row]
            swap rows P[i] and P[pivot_row]

        for j from i + 1 to n do:
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]

    L = eye(n) + L
    return transpose(P), L, U
```

2.2.3 Implementacja w Pythonie

```
def lu_decomposition_pivoting(A):
    n = len(A)
    L = np.zeros((n, n))
    U = A.copy()
    P = np.eye(n)

    for i in range(n):
        pivot_row = np.argmax(np.abs(U[i:, i])) + i
        if pivot_row != i:
            U[[i, pivot_row]] = U[[pivot_row, i]]
            L[[i, pivot_row]] = L[[pivot_row, i]]
            P[[i, pivot_row]] = P[[pivot_row, i]]

        for j in range(i + 1, n):
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]

    L = np.eye(n) + L
    return P.T, L, U
```

2.2.4 Analiza złożoności

W przypadku faktoryzacji LU pivoting również nie wpływa znacznie na rząd złożoności obliczeniowej, przez co pozostaje on równy $O(n^3)$.

2.2.5 Porównanie z Scipy

```
L:
```

```
[[1.          0.          0.        ]
[2.51331659  1.          0.        ]
[1.12521846  0.47304452  1.        ]]
[[ 1.          0.          0.        ]
[ 0.39788064  1.          0.        ]
[ 0.44770263 -0.06369218  1.        ]]
U:
[[ 0.02773589  0.04715342  0.07249239]
[ 0.         -0.07082049 -0.08996962]
[ 0.          0.         -0.02421129]]
[[ 0.06970907  0.04769098  0.09222671]
[ 0.          0.0281781   0.03579717]
[ 0.          0.         -0.02421129]]
```

Widzimy wyniki L i U zwróce przez naszą funkcję (bez pivotingu!) i przez funkcję biblioteczną ‘scipy.linalg.lu‘. WIdzimy, że wyiki się od siebie różnią. Wynika to z tego, że funkcja biblioteczna stosuje pivoting. Jeśli użyejmy naszej funkcji z pivotingiem to uzyskamy identyczne wyniki. Warto zauważyć, że nawet w wersji bez pivotingu, jeśli trafi się nam "szczęśliwa" macierz, to macierz permutacji będzie równa macierzy identycznościowej i wtedy oba algorytmu: nasz bez pivotingu i biblioteczny zwrócą identyczny wynik. Dla n = 3 takie prawdopodobieństwo jest równe 1/6