Algorytmy macierzowe
Laboratorium 4 - Dodawanie i mnożenie macierzy hierarchicznych

Kacper Wąchała, Szymon Hołysz

1. Zastosowane algorytmy

1.1. Mnożenie macierzy przez wektor

Korzystając ze skompresowanej struktury SVD mnożymy rekurencyjnie macierz przez wektor dzieląc go na części a następnie sumując w pionie macierze kratowe. W ten sposób ostatecznie otrzymujemy wektor o pożądanym rozmiarze.

Algorithm 1: Mnożenie macierzy przez wektor

```
1:  function MATRIX_VECTOR_MULT(v, X)
2:      if v.children = ∅ then
3:          if v.rank = 0 then
4:              return 0
5:          end
6:          return v.U * v.D * (v.V * X)
7:      end
8:      X_1 ← X[0...m/2]
9:      X_2 ← X[m/2 + 1...m − 1]
10:     Y_1 ← MATRIX_VECTOR_MULT(v.children[0], X_1)
11:     Y_2 ← MATRIX_VECTOR_MULT(v.children[1], X_2)
12:     Y_3 ← MATRIX_VECTOR_MULT(v.children[2], X_1)
13:     Y_4 ← MATRIX_VECTOR_MULT(v.children[3], X_2)
14:     return [Y_1+Y_2  Y_3+Y_4]
15: end
```

1.2. Dodawanie macierzy skompresowanych

Rekurencyjne dodawanie macierzy sprowadza się do zsumowania odpowiadających sobie podmacierzy przechowywanych w liściach drzewa.

Algorithm 2: Dodawanie macierzy skompresowanych

1: **function** MATRIX_MATRIX_ADD(v, w, max_rank, ε)
2: **if** $v.children = [] \wedge w.children = []$ **then**
3: **if** $v.rank = 0 \wedge w.rank = 0$ **then**
4: **return** NODE(0, $v.size$)
5: **else if** $v.size = (1,1) \wedge w.size = (1,1)$ **then**
6: **if** $v.rank > 0$ **then**
7: $val\_v \leftarrow v.U[0,0] * v.D[0] * v.V[0,0]$
8: **else**
9: $val\_v \leftarrow 0$
10: **end**
11: **if** $w.rank > 0$ **then**
12: $val\_w \leftarrow v.U[0,0] * v.D[0] * v.V[0,0]$
13: **else**
14: $val\_w \leftarrow 0$
15: **end**
16: $val \leftarrow val\_v + val\_w$
17: **return** NODE(1, [1, 1], U=val, D=[1], V=[[1]])
18: **end**
19: **if** $v.rank \neq 0$ **then**
20: $$A \leftarrow v.U * \begin{bmatrix} v.D[0] & 0 & \dots & 0 \\ 0 & v.D[1] & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & v.D[r-1] \end{bmatrix} * v.V$$
21: $$B \leftarrow w.U * \begin{bmatrix} w.D[0] & 0 & \dots & 0 \\ 0 & w.D[1] & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & w.D[r-1] \end{bmatrix} * w.V$$
22: $M \leftarrow A + B$
23: $U, D, V \leftarrow$ TRUNCATED_SVD(M, $max\_rank + 1$, ε)
24: **return** COMPRESS_MATRIX(M, U, D, V, max_rank, ε)
25: **end**
26: **else if** $v.children \neq [] \wedge w.children \neq []$ **then**
27: $Y \leftarrow$ NODE(0, $v.size$)
28: **for** $vc, wc \in (v.children, w.children)$ **do**
29: $Y.children$.APPEND(MATRIX_MATRIX_ADD(vc, wc, max_rank, ε))
30: **end**

return Y

else if $v.children = [] \wedge w.children \neq []$ **then**

$\begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \leftarrow v.U$

$[V_1 \;\; V_2] \leftarrow v.U$

$Y \leftarrow \text{NODE}(0, v.size)$

$Y.children \leftarrow [$

$\text{MATRIX\_MATRIX\_ADD}(\text{Node}(v.rank, U_1.shape, U_1, v.D, V_1), w.children[0], max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(\text{Node}(v.rank, U_1.shape, U_1, v.D, V_2), w.children[1], max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(\text{Node}(v.rank, U_2.shape, U_2, v.D, V_1), w.children[2], max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(\text{Node}(v.rank, U_2.shape, U_2, v.D, V_2), w.children[3], max\_rank, \varepsilon)$

$]$

return Y

else

$\begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \leftarrow w.U$

$[V_1 \;\; V_2] \leftarrow w.U$

$Y \leftarrow \text{NODE}(0, v.size)$

$Y.children \leftarrow [$

$\text{MATRIX\_MATRIX\_ADD}(v.children[0], \text{Node}(v.rank, U_1.shape, U_1, v.D, V_1), max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(v.children[1], \text{Node}(v.rank, U_1.shape, U_1, v.D, V_2), max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(w.children[2], \text{Node}(v.rank, U_2.shape, U_2, v.D, V_1), max\_rank, \varepsilon),$

$\text{MATRIX\_MATRIX\_ADD}(w.children[3], \text{Node}(v.rank, U_2.shape, U_2, v.D, V_2), max\_rank, \varepsilon)$

$]$

return Y

end

end

1.3. Mnożenie macierzy skompresowanych

Mając zaimplementowane dodawanie macierzy skompresowanych możemy z niego skorzystać przy mnożeniu takich macierzy. Wymaga to przemnożenia i zsumowania odpowiadających sobie podmacierzy, a następnie rekonstrukcji macierzy wynikowej o odpowiednim rozmiarze.

Algorithm 3: Mnożenie macierzy skompresowanych

1: **function** MATRIX_MATRIX_MULT(v, w, max_rank, ε)
2: **if** $v.children = [] \land w.children = []$ **then**
3: **if** $v.rank = 0 \land w.rank = 0$ **then**
4: **return** NODE(0, $v.size$)
5: **else if** $v.size = (1,1) \land w.size = (1,1)$ **then**
6: **if** $v.rank > 0$ **then**
7: $val\_v \leftarrow v.U[0,0] * v.D[0] * v.V[0,0]$
8: **else**
9: $val\_v \leftarrow 0$
10: **end**
11: **if** $w.rank > 0$ **then**
12: $val\_w \leftarrow v.U[0,0] * v.D[0] * v.V[0,0]$
13: **else**
14: $val\_w \leftarrow 0$
15: **end**
16: $val \leftarrow val\_v * val\_w$
17: **return** NODE(1, $[1,1]$, U=val, D=$[1]$, V=$[[1]]$)
18: **end**
19: **else**
20: $A \leftarrow v.U * \begin{bmatrix} v.D[0] & 0 & \dots & 0 \\ 0 & v.D[1] & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & v.D[r-1] \end{bmatrix} * v.V$
21: $B \leftarrow w.U * \begin{bmatrix} w.D[0] & 0 & \dots & 0 \\ 0 & w.D[1] & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & w.D[r-1] \end{bmatrix} * w.V$
22: $M \leftarrow A * B$
23: $U, D, V \leftarrow$ TRUNCATED_SVD(M, $max\_rank + 1$, ε)
24: **return** COMPRESS_MATRIX(M, U, D, V, max_rank, ε)
25: **end**
26: **else if** $v.children \neq [] \land w.children \neq []$ **then**
27: $Y_{00} \leftarrow$ MATRIX_MATRIX_ADD(
MATRIX_MATRIX_MULT($v.children[0]$, $w.children[0]$, max_rank, ε),
MATRIX_MATRIX_MULT($v.children[1]$, $w.children[2]$, max_rank, ε),

$max\_rank, \varepsilon)$

28: $Y_{01} \leftarrow$ MATRIX_MATRIX_ADD(
MATRIX_MATRIX_MULT($v.children[0], w.children[1], max\_rank, \varepsilon$),
MATRIX_MATRIX_MULT($v.children[1], w.children[3], max\_rank, \varepsilon$),
max_rank, ε)

29: $Y_{10} \leftarrow$ MATRIX_MATRIX_ADD(
MATRIX_MATRIX_MULT($v.children[2], w.children[0], max\_rank, \varepsilon$),
MATRIX_MATRIX_MULT($v.children[3], w.children[2], max\_rank, \varepsilon$),
max_rank, ε)

30: $Y_{11} \leftarrow$ MATRIX_MATRIX_ADD(
MATRIX_MATRIX_MULT($v.children[2], w.children[1], max\_rank, \varepsilon$),
MATRIX_MATRIX_MULT($v.children[3], w.children[3], max\_rank, \varepsilon$),
max_rank, ε)

31: $root \leftarrow$ NODE($0, v.size$)

32: $root.children \leftarrow [Y_{00}, Y_{01}, Y_{10}, Y_{11}]$

33: **return** $root$

34: **else if** $v.children = [] \wedge w.children \neq []$ **then**

35: $\begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \leftarrow v.U$

36: $\begin{bmatrix} V_1 & V_2 \end{bmatrix} \leftarrow v.U$

37: $Y \leftarrow$ NODE($0, v.size$)

38: $Y.children \leftarrow$ [
MATRIX_MATRIX_MULT(Node($v.rank, U_1.shape, U_1, v.D, V_1$), $w.children[0]$, max_rank, ε),
MATRIX_MATRIX_MULT(Node($v.rank, U_1.shape, U_1, v.D, V_2$), $w.children[1]$, max_rank, ε),
MATRIX_MATRIX_MULT(Node($v.rank, U_2.shape, U_2, v.D, V_1$), $w.children[2]$, max_rank, ε),
MATRIX_MATRIX_MULT(Node($v.rank, U_2.shape, U_2, v.D, V_2$), $w.children[3]$, max_rank, ε)
]

39: **return** Y

40: **else**

41: $\begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \leftarrow w.U$

42: $\begin{bmatrix} V_1 & V_2 \end{bmatrix} \leftarrow w.U$

43: $Y \leftarrow$ NODE($0, v.size$)

44: $Y.children \leftarrow$ [

MATRIX_MATRIX_MULT($v.children[0]$, Node($v.rank$, $U_1.shape$, U_1, $v.D$, V_1), max_rank, ε),

MATRIX_MATRIX_MULT($v.children[1]$, Node($v.rank$, $U_1.shape$, U_1, $v.D$, V_2), max_rank, ε),

MATRIX_MATRIX_MULT($w.children[2]$, Node($v.rank$, $U_2.shape$, U_2, $v.D$, V_1), max_rank, ε),

MATRIX_MATRIX_MULT($w.children[3]$, Node($v.rank$, $U_2.shape$, U_2, $v.D$, V_2), max_rank, ε)

]

return Y

end

end

2. Implementacja

2.1. Mnożenie macierzy przez wektor

```
from tree import *

def matrix_vector_mult(node: Node, X: np.ndarray):
    if not node.children:
        if node.rank == 0:
            return np.zeros((X.size, X.size))
        result = node.U @ np.diag(node.D) @ (node.V @ X)
        return result

    Xs = np.split(X, 2)
    Y = [matrix_vector_mult(node.children[i], Xs[i%2]) for i in range(4)]
    return np.hstack((Y[0] + Y[1], Y[2] + Y[3]))
```

2.2. Dodawanie macierzy skompresowanych

```
def matrix_matrix_add(v, w, max_rank, eps):
    if not v.children and not w.children:
        if v.rank == 0 and w.rank == 0:
            return Node(0, v.size)
        elif v.size == (1,1) and w.size == (1,1):
            val_v = v.U[0,0]*v.D[0]*v.V[0,0] if v.rank > 0 else 0
            val_w = w.U[0,0]*w.D[0]*w.V[0,0] if w.rank > 0 else 0
            val = val_v + val_w
            U = np.array([[val]])
            D = np.array([1.0])
            V = np.array([[1.0]])
            return Node(1, (1,1), U, D, V)
        elif v.rank ≠ 0 and w.rank ≠ 0:
            A = v.U @ np.diag(v.D) @ v.V
            B = w.U @ np.diag(w.D) @ w.V
            M = A + B
            U, D, V = truncated_svd(M, max_rank + 1, eps)
            return compress_matrix(M, U, D, V, max_rank, eps)

    elif v.children and w.children:
        Y = Node(0, v.size)
        for vc, wc in zip(v.children, w.children):
            Y.children.append(
                matrix_matrix_add(vc, wc, max_rank, eps)
            )
        return Y

    elif not v.children and w.children:
        mid_row = w.children[0].size[0]
        mid_col = w.children[0].size[1]

        U1 = v.U[:mid_row, :]
        U2 = v.U[mid_row:, :]
        V1 = v.V[:, :mid_col]
        V2 = v.V[:, mid_col:]

        Y = Node(0, v.size)
        Y.children = [
            matrix_matrix_add(Node(v.rank, U1.shape, U1, v.D, V1), w.children[0],
max_rank, eps),
            matrix_matrix_add(Node(v.rank, U1.shape, U1, v.D, V2), w.children[1],
max_rank, eps),
            matrix_matrix_add(Node(v.rank, U2.shape, U2, v.D, V1), w.children[2],
max_rank, eps),
            matrix_matrix_add(Node(v.rank, U2.shape, U2, v.D, V2), w.children[3],
max_rank, eps)
```

```
        ]
        return Y

    else:
        mid_row = v.children[0].size[0]
        mid_col = v.children[0].size[1]

        U1 = w.U[:mid_row, :]
        U2 = w.U[mid_row:, :]
        V1 = w.V[:, :mid_col]
        V2 = w.V[:, mid_col:]

        Y = Node(0, v.size)
        Y.children = [
            matrix_matrix_add(v.children[0], Node(w.rank, U1.shape, U1, w.D, V1),
max_rank, eps),
            matrix_matrix_add(v.children[1], Node(w.rank, U1.shape, U1, w.D, V2),
max_rank, eps),
            matrix_matrix_add(v.children[2], Node(w.rank, U2.shape, U2, w.D, V1),
max_rank, eps),
            matrix_matrix_add(v.children[3], Node(w.rank, U2.shape, U2, w.D, V2),
max_rank, eps)
        ]
        return Y
```

2.3. Mnożenie macierzy skompresowanych

```
def matrix_matrix_mult(v, w, max_rank, eps):
    if not v.children and not w.children:
        if v.rank == 0 or w.rank == 0:
            return Node(0, v.size)
        elif v.size == (1,1) and w.size == (1,1):
            val_v = v.U[0,0]*v.D[0]*v.V[0,0] if v.rank>0 else 0
            val_w = w.U[0,0]*w.D[0]*w.V[0,0] if w.rank>0 else 0
            val = val_v * val_w
            U = np.array([[val]], dtype=np.float64)
            D = np.array([1.0], dtype=np.float64)
            V = np.array([[1.0]], dtype=np.float64)
            return Node(1, (1,1), U, D, V)
        else:
            A = v.U @ np.diag(v.D) @ v.V
            B = w.U @ np.diag(w.D) @ w.V
            M = A @ B
            U, D, V = truncated_svd(M, max_rank+1, eps)
            return compress_matrix(M, U, D, V, max_rank, eps)

    elif v.children and w.children:
        Y00 = matrix_matrix_add(
            matrix_matrix_mult(v.children[0], w.children[0], max_rank, eps),
            matrix_matrix_mult(v.children[1], w.children[2], max_rank, eps),
            max_rank, eps
        )
        Y01 = matrix_matrix_add(
            matrix_matrix_mult(v.children[0], w.children[1], max_rank, eps),
            matrix_matrix_mult(v.children[1], w.children[3], max_rank, eps),
            max_rank, eps
        )
        Y10 = matrix_matrix_add(
            matrix_matrix_mult(v.children[2], w.children[0], max_rank, eps),
            matrix_matrix_mult(v.children[3], w.children[2], max_rank, eps),
            max_rank, eps
        )
        Y11 = matrix_matrix_add(
            matrix_matrix_mult(v.children[2], w.children[1], max_rank, eps),
            matrix_matrix_mult(v.children[3], w.children[3], max_rank, eps),
```

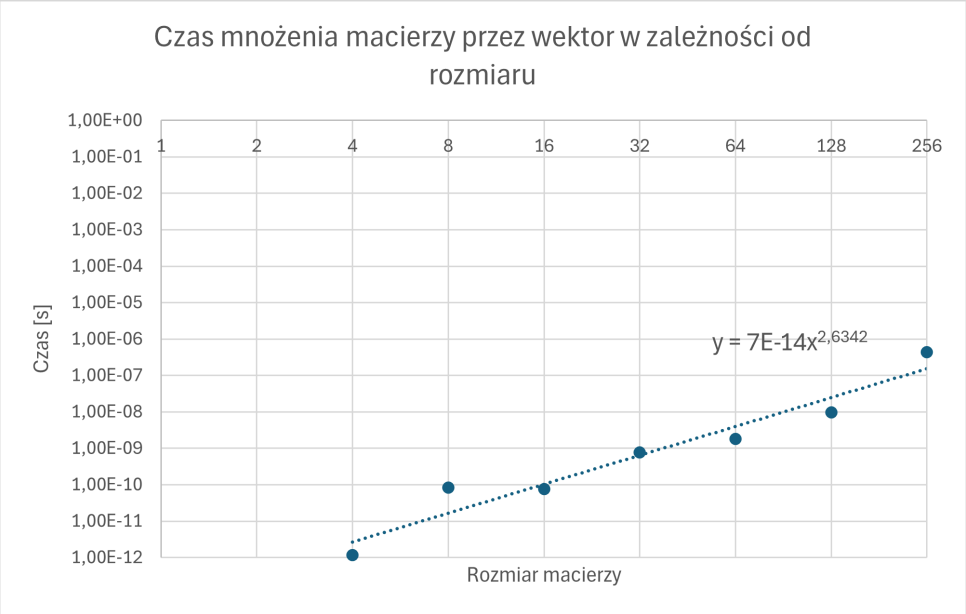
```
            max_rank, eps
        )
        root = Node(0, v.size)
        root.children = [Y00, Y01, Y10, Y11]
        return root

    elif not v.children and w.children:
        mid_row = w.children[0].size[0]
        mid_col = w.children[0].size[1]

        U1 = v.U[:mid_row, :]
        U2 = v.U[mid_row:, :]
        V1 = v.V[:, :mid_col]
        V2 = v.V[:, mid_col:]

        Y00 = matrix_matrix_mult(Node(v.rank, U1.shape, U1, v.D, V1), w.children[0],
max_rank, eps)
        Y01 = matrix_matrix_mult(Node(v.rank, U1.shape, U1, v.D, V2), w.children[1],
max_rank, eps)
        Y10 = matrix_matrix_mult(Node(v.rank, U2.shape, U2, v.D, V1), w.children[2],
max_rank, eps)
        Y11 = matrix_matrix_mult(Node(v.rank, U2.shape, U2, v.D, V2), w.children[3],
max_rank, eps)

        root = Node(0, v.size)
        root.children = [Y00, Y01, Y10, Y11]
        return root

    else:
        mid_row = v.children[0].size[0]
        mid_col = v.children[0].size[1]

        U1 = w.U[:mid_row, :]
        U2 = w.U[mid_row:, :]
        V1 = w.V[:, :mid_col]
        V2 = w.V[:, mid_col:]

        Y00 = matrix_matrix_mult(v.children[0], Node(w.rank, U1.shape, U1, w.D, V1),
max_rank, eps)
        Y01 = matrix_matrix_mult(v.children[1], Node(w.rank, U1.shape, U1, w.D, V2),
max_rank, eps)
        Y10 = matrix_matrix_mult(v.children[2], Node(w.rank, U2.shape, U2, w.D, V1),
max_rank, eps)
        Y11 = matrix_matrix_mult(v.children[3], Node(w.rank, U2.shape, U2, w.D, V2),
max_rank, eps)

        root = Node(0, v.size)
        root.children = [Y00, Y01, Y10, Y11]
        return root
```

3. Wyniki

Aby zbadać efektywność zaimplementowanych w ramach laboratorium algorytmów, zmierzono czasy ich wykonania dla losowych macierzy wielkości 2^k, $k = 1, 2, 3, ...$, a także błąd średniokwadratowy pomiędzy macierzą oryginalną a macierzą wynikową po dekompresji. Pomiarów dokonano z parametrami `max_rank=1` oraz $\varepsilon = 0.01$ Dodatkowo zwizualizowano wynikowe macierze kompresji.

3.1. Mnożenie macierzy przez wektor

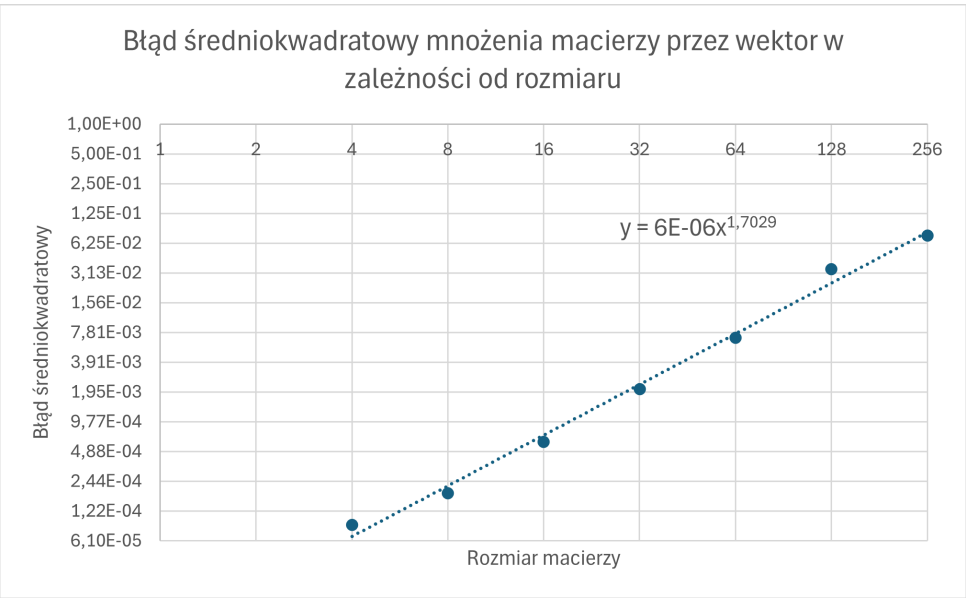
3.1.1. Czas wykonania



Rysunek 1: Wykres zależności czasu wykonania od rozmiaru macierzy dla algorytmu mnożenia macierzy przez wektor

Na podstawie otrzymanych wartości czasowych do wykresu dopasowano funkcję wykładniczą. Wyznaczono empirycznie wykładnik sugerujący złożoność czasową rzędu $O(n^{2.63})$.

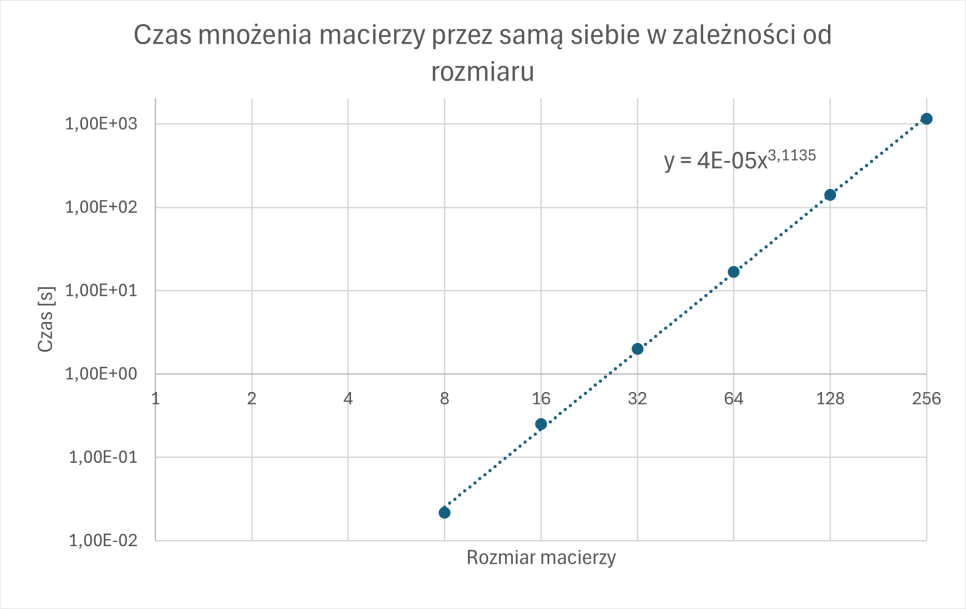
3.1.2. Błąd średniokwadratowy



Rysunek 2: Wykres zależności wartości błędu średniokwadratowego od rozmiaru macierzy dla algorytmu mnożenia macierzy przez wektor

3.2. Mnożenie macierzy przez samą siebie

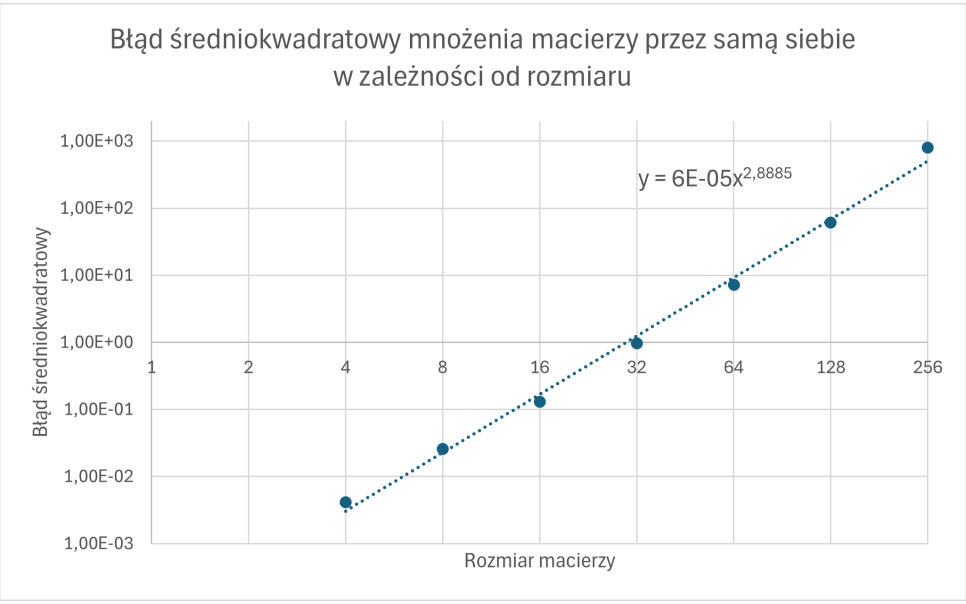
3.2.1. Czas wykonania



Rysunek 3: Wykres zależności czasu wykonania od rozmiaru macierzy dla algorytmu mnożenia macierzy przez wektor

Do uzyskanych wartości pomiaru czasu została dopasowana funkcja wykładnicza. Wyznaczono empirycznie wykładnik pozwalający oszacować rząd złożoności czasowej algorytmu na $O(n^{3.11})$.

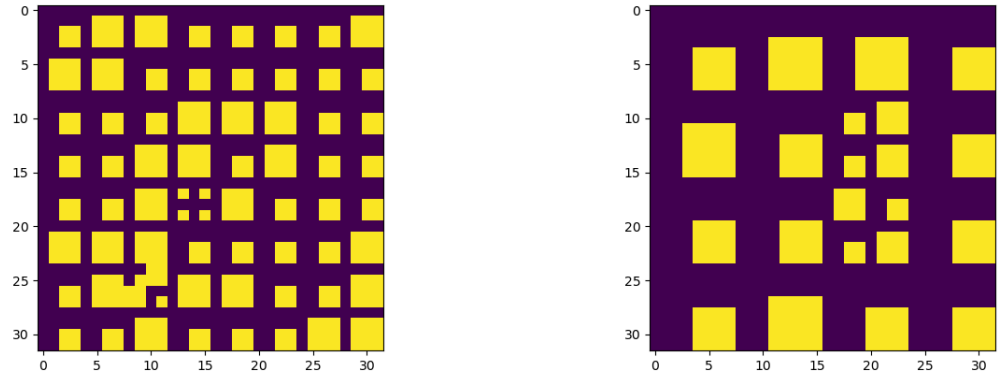
3.2.2. Błąd średniokwadratowy



Rysunek 4: Wykres zależności błędu średniokwadratowego od rozmiaru macierzy dla algorytmu mnożenia macierzy przez wektor

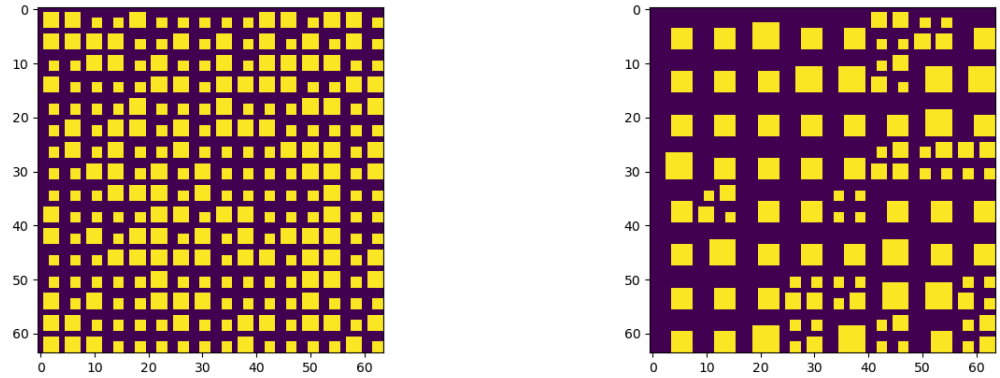
3.2.3. Macierze kompresji

3.2.3.1. $n = 2^5$



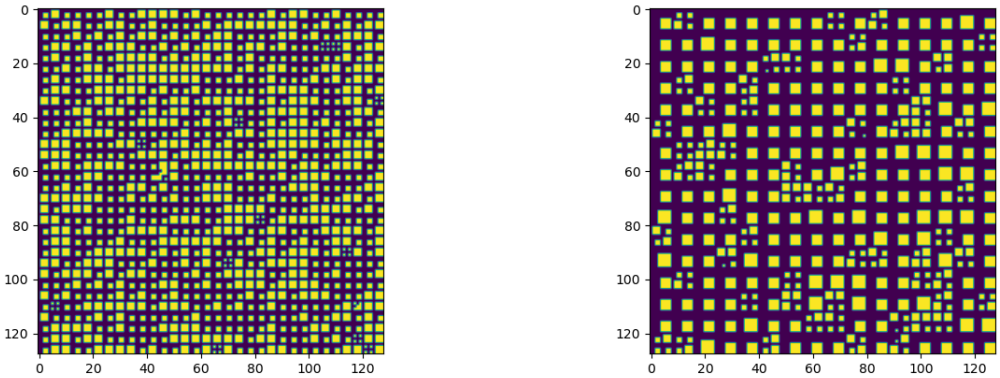
Rysunek 5: Macierze kompresji dla $n = 2^5$, $\varepsilon = 0.5$, z lewej `max_rank=2`, z prawej `max_rank=4`

3.2.3.2. $n = 2^6$



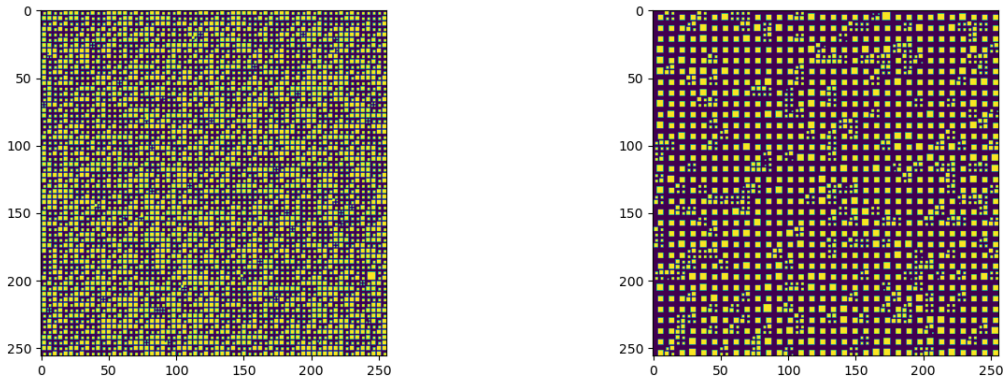
Rysunek 6: Macierze kompresji dla $n = 2^6$, $\varepsilon = 0.5$, z lewej `max_rank=2`, z prawej `max_rank=4`

3.2.3.3. $n = 2^7$



Rysunek 7: Macierze kompresji dla $n = 2^7$, $\varepsilon = 0.5$, z lewej `max_rank=2`, z prawej `max_rank=4`

3.2.3.4. $n = 2^8$



Rysunek 8: Macierze kompresji dla $n = 2^8$, $\varepsilon = 0.5$, z lewej `max_rank=2`, z prawej `max_rank=4`

4. Wnioski

W ramach laboratorium zaimplementowano algorytmy do wykonywania działań arytmetycznych na macierzach hierarchicznych. Sprawdzono poprawność implementacji oraz dokonano pomiarów złożoności czasowej.

Dla mnożenia macierzy skompresowanej przez wektor uzyskano empiryczną złożoność $\approx n^{2.63}$. Dla mnożenia dwóch macierzy skompresowanych hierarchicznie zmierzona złożoność wyniosła $\approx n^{3.11}$.

Błąd średniokwadratowy aproksymacji rośnie wraz z rozmiarem macierzy przy stałych wartościach `max_rank` i ε.