Algorytmy macierzowe

Laboratorium 5

Sprawozdanie

Łukasz Stępień, Szymon Urbański

1. Temat zadania

Laboratorium polegało na zaimplementowaniu algorytmu mnożenia skompresowanej macierzy przez wektor oraz przez samą siebie.

2. Rozwiązanie

Pseudokod:

```
matrix_vector_mult(v, X)
    if v.sons = ∅ then
        if v.rank = 0 then
            return zeros(size(A).rows);
        end if
        return v.U ∗ (v.V ∗ X);
    end if
    rows ← size(X).rows;
    X1 ← X(1 : rows/2, ∗);
    X2 ← X(rows/2 + 1 : rows, ∗);

    Y1_1 ← matrix_vector_mult(v.sons(1), X1);
    Y2_1 ← matrix_vector_mult(v.sons(2), X2);
    Y1_2 ← matrix_vector_mult(v.sons(3), X1);
    Y2_2 ← matrix_vector_mult(v.sons(4), X2);

    return concatenate(Y1_1 + Y2_1, Y1_2 + Y2_2);
```

Fragmenty kodu:

```python
def matrix_vector_mul(v, X):
    if v.children == []:
        if v.rank == 0:
            return np.zeros((v.size[1] - v.size[0], len(X[0])))
        return v.U @ (v.V @ X)
    rows = len(X)
    X1 = X[:rows//2, :]
    X2 = X[rows//2: rows, :]
    Y_11 = matrix_vector_mul(v.children[0], X1)
    Y_12 = matrix_vector_mul(v.children[1], X2)
    Y_21 = matrix_vector_mul(v.children[2], X1)
    Y_22 = matrix_vector_mul(v.children[3], X2)

    res = np.zeros((len(Y_11) + len(Y_12), len(X[0])))

    for i in range(len(Y_11)):
        for j in range(len(Y_11[i])):
            res[i][j] = Y_11[i][j] + Y_12[i][j]

    for i in range(len(Y_12)):
        for j in range(len(Y_12[i])):
            res[len(Y_11) + i][j] = Y_21[i][j] + Y_22[i][j]
    return res
```

Figure 1 Mnożenie skompresowanej macierzy przez wektor

```python
def generate_3d_grid_matrix(k):
    size = 2**(3*k)
    matrix = [[0] * size for _ in range(size)]
    eps = np.finfo(float).eps

    for i in range(size):
        x, y, z = np.unravel_index(i, (2**k, 2**k, 2**k))
        neighbors = [
            np.ravel_multi_index((x + dx, y + dy, z + dz),
                                 (2**k, 2**k, 2**k), mode='wrap')
            for dx in [-1, 0, 1]
            for dy in [-1, 0, 1]
            for dz in [-1, 0, 1]
            if (dx != 0 or dy != 0 or dz != 0) and 0 <= x + dx < 2**k and 0 <= y + dy < 2**k and 0 <= z + dz < 2**k
        ]
        for neighbor in neighbors:
            matrix[i][neighbor] = 1
        matrix[i][i] = 1
    return np.array(matrix)
```

Figure 2 Generacja macierzy o strukturze opisującej topologię trójwymiarowej

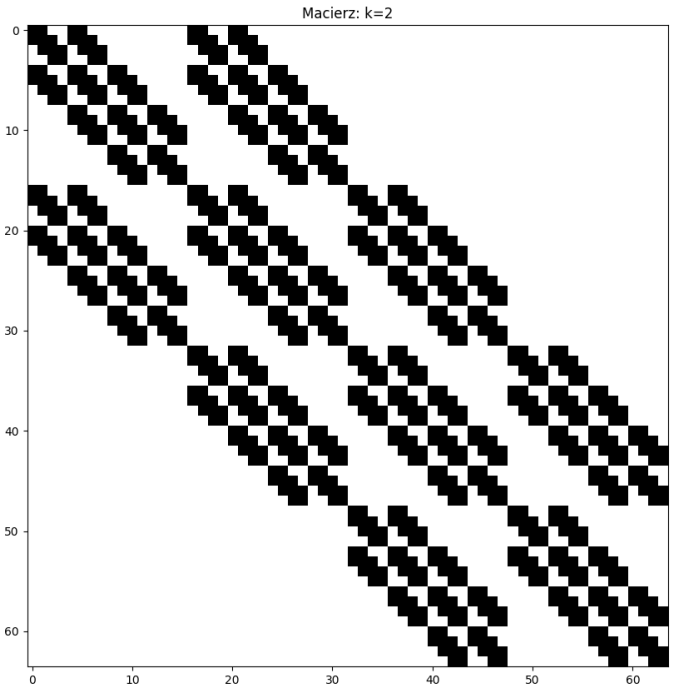
```python
class Node:
    def __init__(self, size, rank=0):
        self.children = []
        self.parent = None
        self.size = size
        self.rank = rank
        self.sv = []
        self.U = None
        self.V = None

    def append(self, node):
        self.children.append(node)
        node.parent = self


def create_tree(t_min, t_max, s_min, s_max, r, eps):
    global ax
    global A
    U, D, V = randomized_svd(
        A[t_min:t_max, s_min:s_max], n_components=r+1, random_state=0)
    if len(D) <= r or D[r] < eps:
        draw_black((t_min, s_min), (t_max, s_max), ax)
        v = compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r)

    else:
        draw_cross_with_square((t_min, s_min), (t_max, s_max), ax)
        v = Node((t_min, t_max, s_min, s_max))
        t_new_max = t_min + (t_max - t_min) // 2
        s_new_max = s_min + (s_max - s_min) // 2

        v.append(create_tree(t_min, t_new_max, s_min, s_new_max, r, eps))
        v.append(create_tree(t_min, t_new_max, s_new_max, s_max, r, eps))
        v.append(create_tree(t_new_max, t_max, s_min, s_new_max, r, eps))
        v.append(create_tree(t_new_max, t_max, s_new_max, s_max, r, eps))
    return v


def compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r):
    global A
    if np.all(A[t_min:t_max, s_min:s_max] == 0):
        v = Node((t_min, t_max, s_min, s_max))
        v.U = np.zeros((t_max - t_min, s_max - s_min))
        v.V = np.zeros((t_max - t_min, s_max - s_min))
        return v
    v = Node((t_min, t_max, s_min, s_max), r)
    v.sv = D[:r+1]
    v.U = U[:, :r + 1]
    v.V = np.diag(D[:r + 1]) @ V[:r + 1, :]
    return v
```

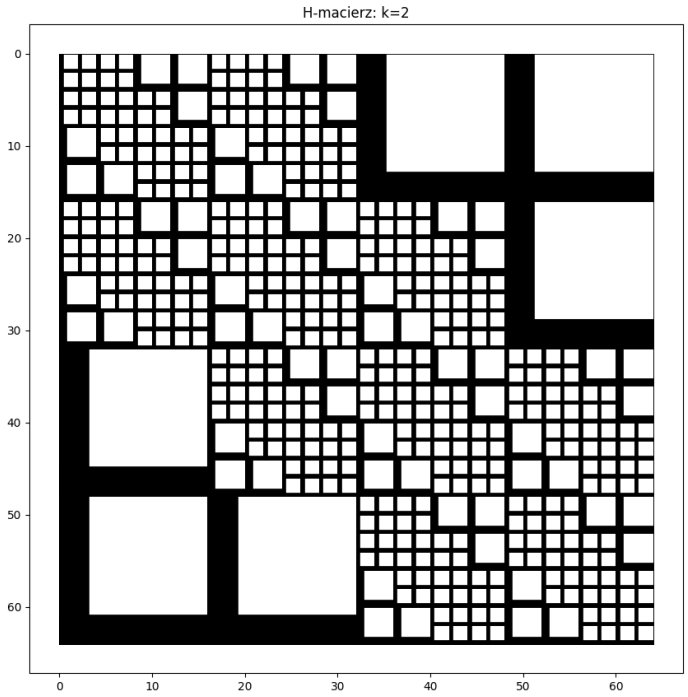
Figure 3 Kod odpowiadający za kompresję

3. Wyniki.

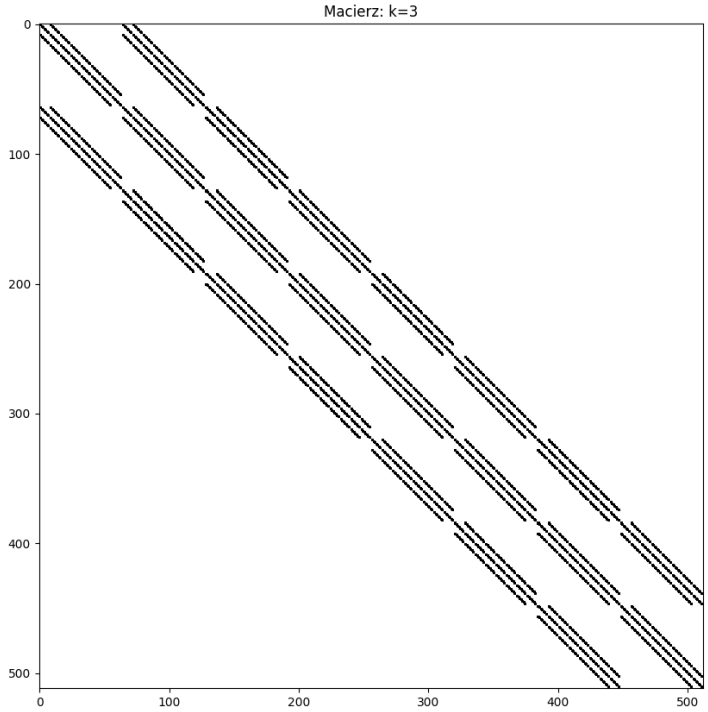
Poniżej przedstawiono wizualizacje macierzy oraz ich kompresji używanych do dalszych obliczeń.

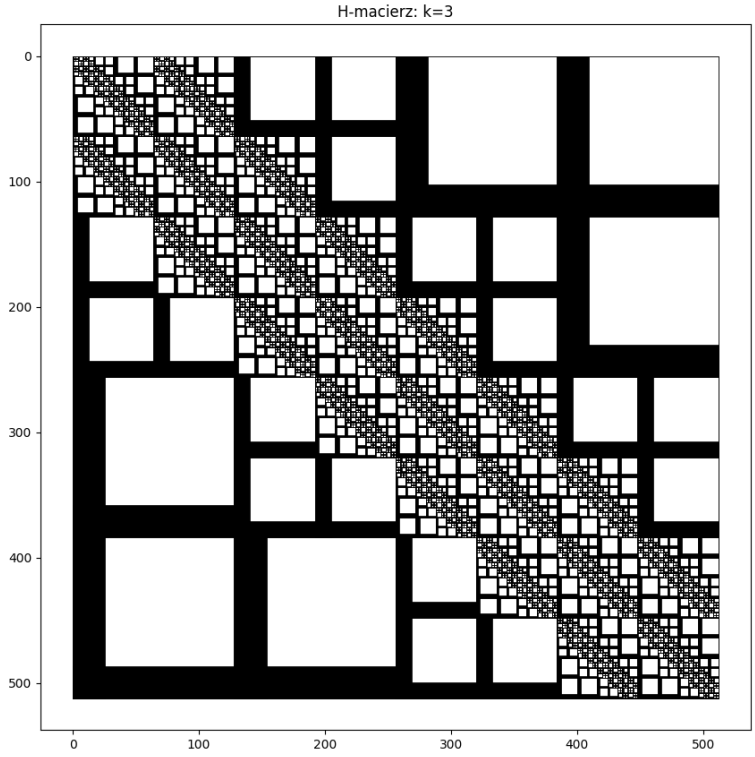
- Macierz o rozmiarze 2^{3*2}



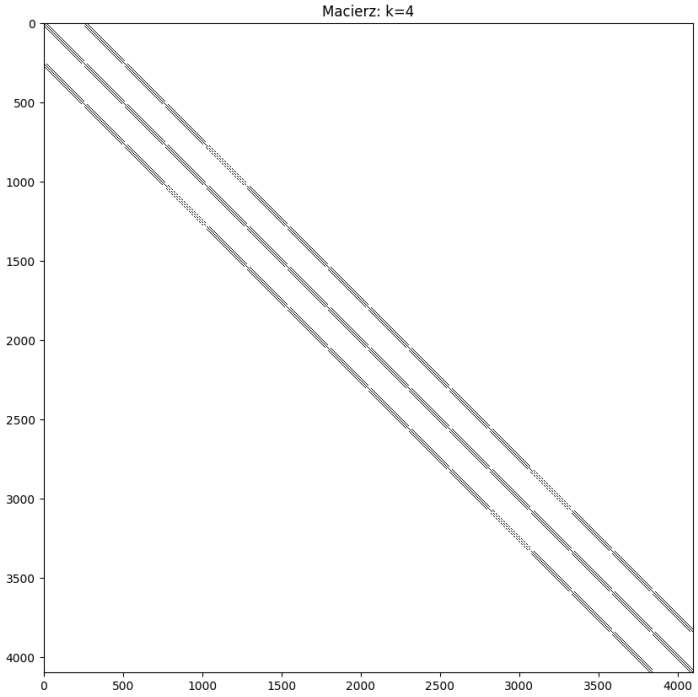


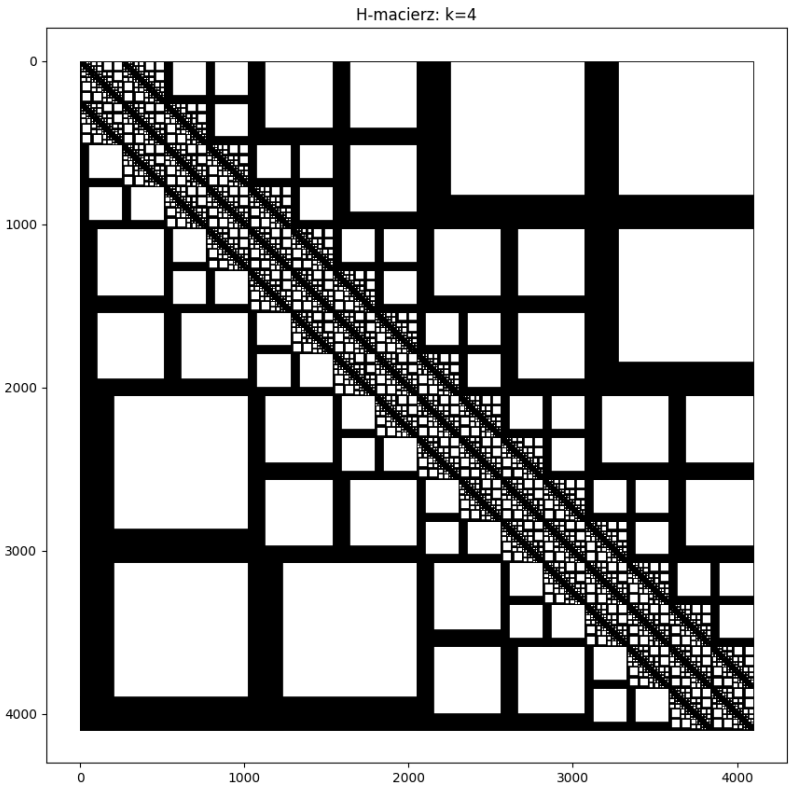
- Macierz o rozmiarze 2^{3*3}



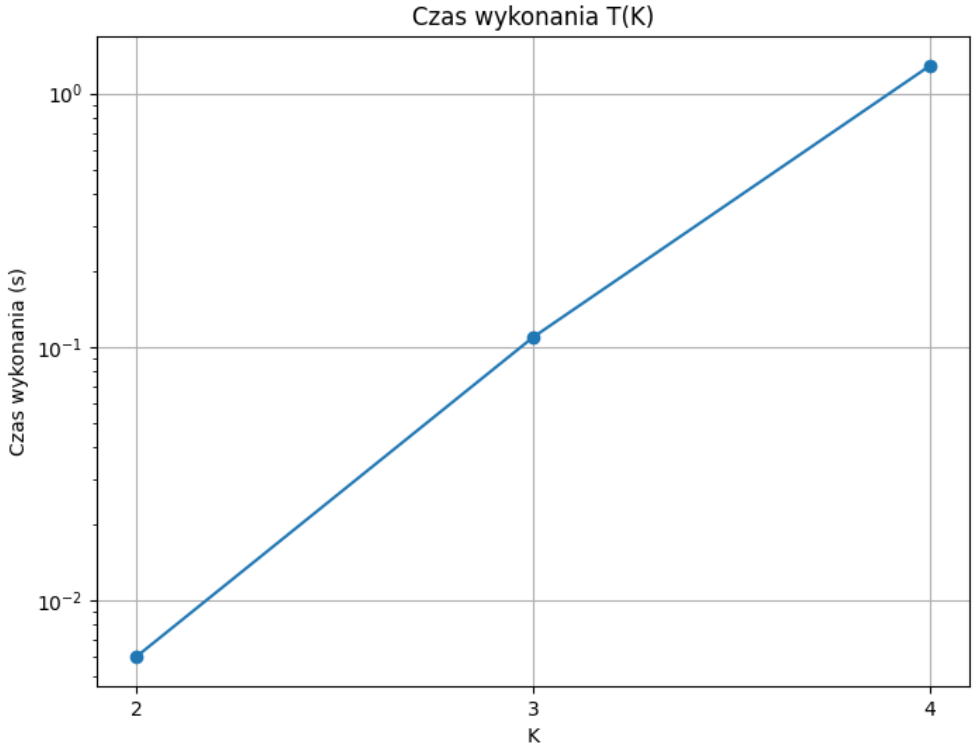


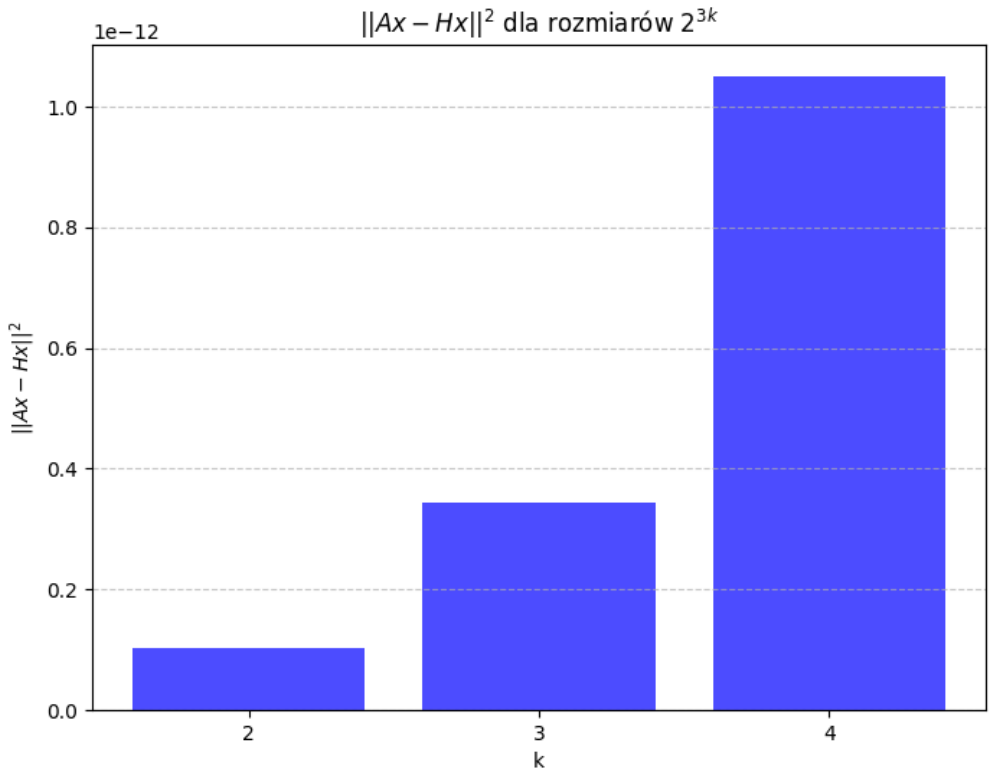
- Macierz o rozmiarze 2^{3*4}





Poniższe wykresy przedstawiają zależności czasu oraz błędu od rozmiaru macierzy:





4. Wnioski

Mnożenie skompresowanej macierzy przez wektor pozwala na redukcję ilości operacji, co przekłada się na skrócony czas wykonania algorytmu. Niskie błędy MSE oznaczają poprawną implementacje algorytmu. Można zauważyć, że im większy rozmiar macierzy, tym większy błąd oraz dłuższy czas wykonania. Kompresja macierzy zachodzi również poprawnie, o czym świadczą wygenerowane przez rysownik obrazy. Nie udało się zaimplementować algorytmu mnożenia H-macierzy przez samą siebie.