Przemysław Rola, Juliusz Wasieleski
Informatyka, III rok, grupa 6
styczeń 2024

Algorytmy macierzowe – algebra macierzy hierarchicznych – sprawozdanie

1. Opis ćwiczenia

Naszym zadaniem było, po wybraniu naszego ulubionego języka, wygenerowanie losowych macierzy.

Następnie mieliśmy porównać czasy wykonania i błędy dla mnożenia macierzy skompresowanych razy wektor oraz razy samą siebie.

2. Środowisko, biblioteki, założenia oraz użyte narzędzia

Ćwiczenie wykonaliśmy w języku Python przy użyciu Jupyer Notebooka. Do obliczeń, przechowywania danych użyliśmy bibliotek *numpy, pandas, scipy.*

Do rysowania wykresów użyliśmy biblioteki *matplotlib.*

Wszystkie obliczenia prowadziliśmy na komputerze Lenovo Y50-70 z systemem Windows 10 Pro w wersji 10.0.19045, procesor Intel Core i7-4720HQ 2.60GHz, 2601 MHz, rdzenie: 4, procesory logiczne: 8.

3. Implementacje

3.1 Mnożenie macierzy skompresowanej przez wektor

3.1.1 Pseudokod

matrix_vector_mult(v, X)*:*

if v.sons == Ø:

if v.rank == 0:

return zeros(size(A).rows)

*return v.U * (v.V * X)*

rows = size(X).rows

*X1 = X[1:rows/2, *]*

*X2 = X[rows/2 + 1 : rows, *]*

Y11 = matrix_vector_mult(v.sons(1), X1)

Y12 = matrix_vector_mult(v.sons(2), X2)

Y21 = matrix_vector_mult(v.sons(3), X1)

Y22 = matrix_vector_mult(v.sons(4), X2)

return $\begin{bmatrix} Y11 + Y12 \\ Y21 + Y22 \end{bmatrix}$

3.1.2 Istotne fragmenty implementacji

```
def _matxvec_rekur(node : CompressTree, vector : np.array):
    if node.leaf:
        return (node.u @ (node.v @ vector)) * node.s[0]
    else:
        n = len(vector)
        upper, lower = vector[:n//2], vector[n//2:]
        out_upper = _matxvec_rekur(node.childs[0][0], upper) + 
_matxvec_rekur(node.childs[0][1], lower)
        out_lower = _matxvec_rekur(node.childs[1][0], upper) + 
_matxvec_rekur(node.childs[1][1], lower)
        return np.append(out_upper, out_lower, axis=0)
```

3.2 Minimal degree permutation

3.2.1Pseudokod

mult(A,B) :

If v.sons == 0 and w.sons == 0:

If v.rank == 0 and w.rank == 0:

Return Zero matrix of proper dimentions

Else If v.rank != 0 and w.rank != 0:

Return v.U(v.V * w.U) * w.V

If v.sons > and w.sons>0:

A = $\begin{bmatrix} A1 & A2 \\ A3 & A4 \end{bmatrix}$

B = $\begin{bmatrix} B1 & B2 \\ B3 & B4 \end{bmatrix}$

Return $\begin{bmatrix} add(mult(A1,B1), mult(A2,B3)) & add(mult(A1,B2) + mult(A2,B4)) \\ add(mult(A3,B1), mult(A4,B3)) & add(mult(A3,B2), mult(A4,B4)) \end{bmatrix}$

If v.sons == 0 and w.sons > 0:

A = U1V1

U1 = U1’U1’’

V1 = V1’V1’’

A = $\begin{bmatrix} U1^{'} * V1' & U1^{'} * V1'' \\ U1^{''} * V1' & U1^{''} * V1'' \end{bmatrix}$

return $\begin{bmatrix} add(mult(A1,B1), mult(A2,B3)) & add(mult(A1,B2) + mult(A2,B4)) \\ add(mult(A3,B1), mult(A4,B3)) & add(mult(A3,B2), mult(A4,B4)) \end{bmatrix}$

If v.sons>0 and w = 0:

Analogicznie dla macierzy B

3.2.2 Istotne fragmenty implementacji

```
def _mul_rekur(node1 : CompressTree, node2 : CompressTree) ->  CompressTree:
    result_node = CompressTree(None, None, None, None, None)
    if node1.leaf and node2.leaf:
        r = node1.u.shape[1]
        U, V = _mul_compmatrix(node1.u @ np.diag(node1.s), node1.v, node2.u @
np.diag(node2.s), node2.v)
        result_node.make_leaf(U, np.ones((r,)), V)

    elif node1.leaf or node2.leaf:

        node = node1 if node1.leaf else node2

        n, r = node.u.shape

        u_part = [node.u[:n//2], node.u[n//2:]]
        v_part = [node.v[:,:n//2], node.v[:,n//2:]]

        fake_childs = [[None for _ in range(2)] for _ in range(2)]
        for i in range(2):
            for j in range(2):
                fake_childs[i][j] = CompressTree(None, None, None, None, None)
                fake_childs[i][j].make_leaf(u_part[i], np.ones((r,)), v_part[j])

        for i in range(2):
            for j in range(2):
                if node1.leaf:
                    first = _mul_rekur(fake_childs[i][0], node2.childs[0][j])
                    second = _mul_rekur(fake_childs[i][1], node2.childs[1][j])
                else:
                    first = _mul_rekur(node1.childs[i][0], fake_childs[0][j])
                    second = _mul_rekur(node1.childs[i][1], fake_childs[1][j])

                child = _add_rekur(first, second)
                result_node.childs[i][j] = child

    else:
        for i in range(2):
            for j in range(2):
                first = _mul_rekur(node1.childs[i][0], node2.childs[0][j])
                second = _mul_rekur(node1.childs[i][1], node2.childs[1][j])

                child = _add_rekur(first, second)
                result_node.childs[i][j] = child
    return result_node
```

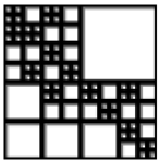
4. Analiza wykonanych pomiarów

4.1 Pomiary mnożenia razy wektor

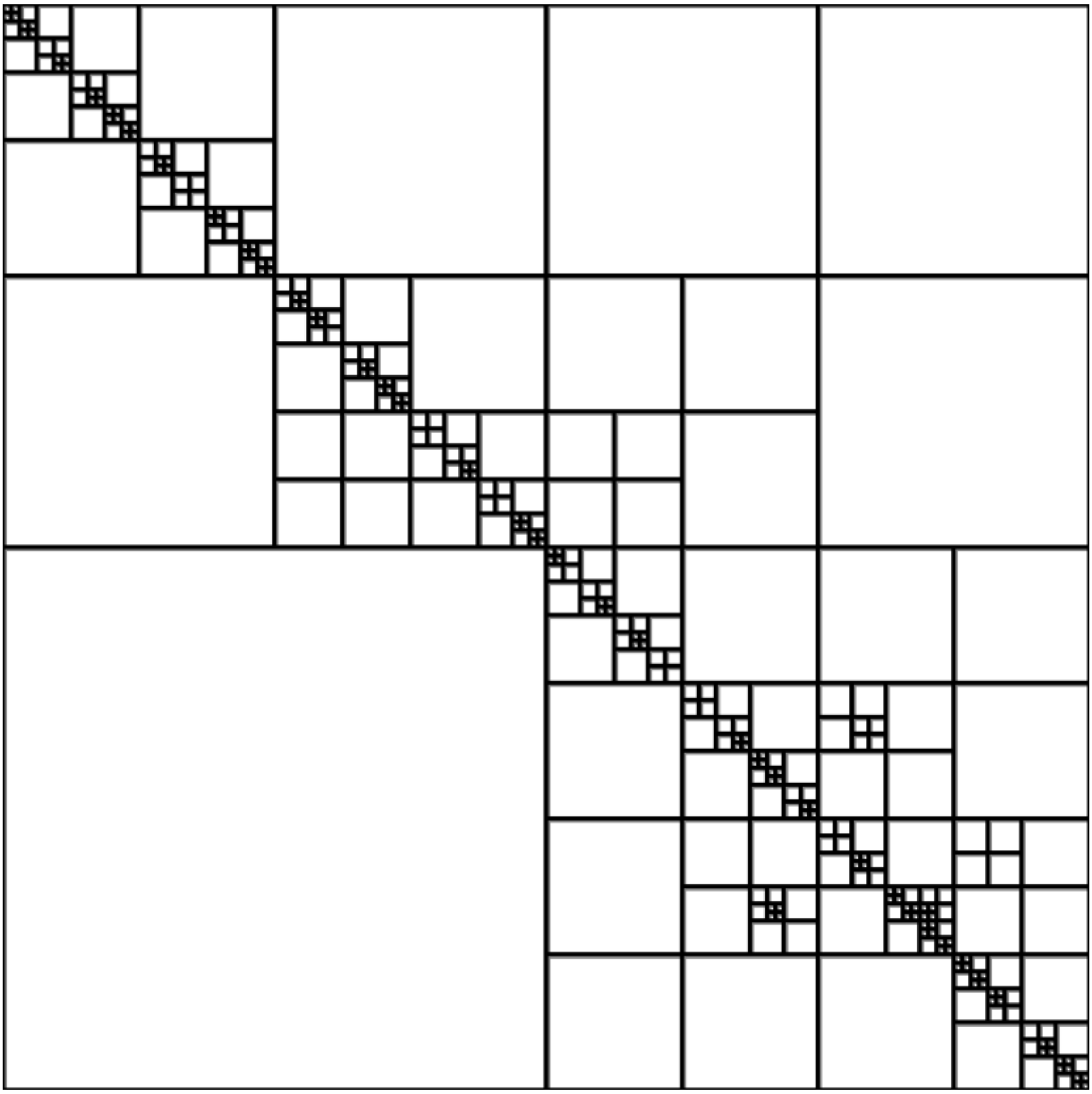
k	s	time	error
2	32	0.002167	9.893376
3	256	0.018169	287.397327
4	2048	0.101239	3557.652625

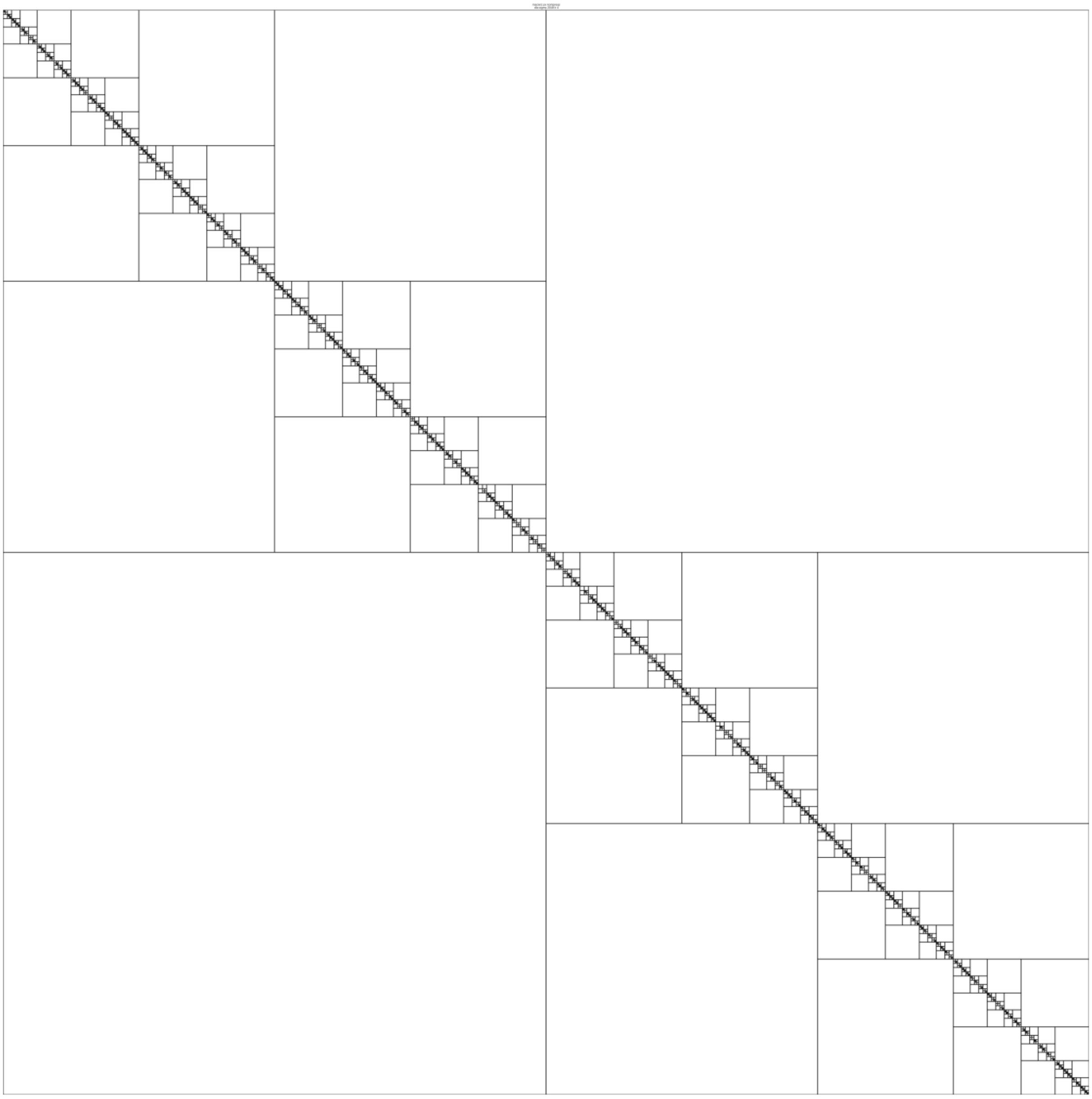
Tab.1 pomiary mnożenia razy wektor

macierz po kompresji
dla sigmy 32 k 2



macierz po kompresji
dla sigmy 256 k 3





Rys.

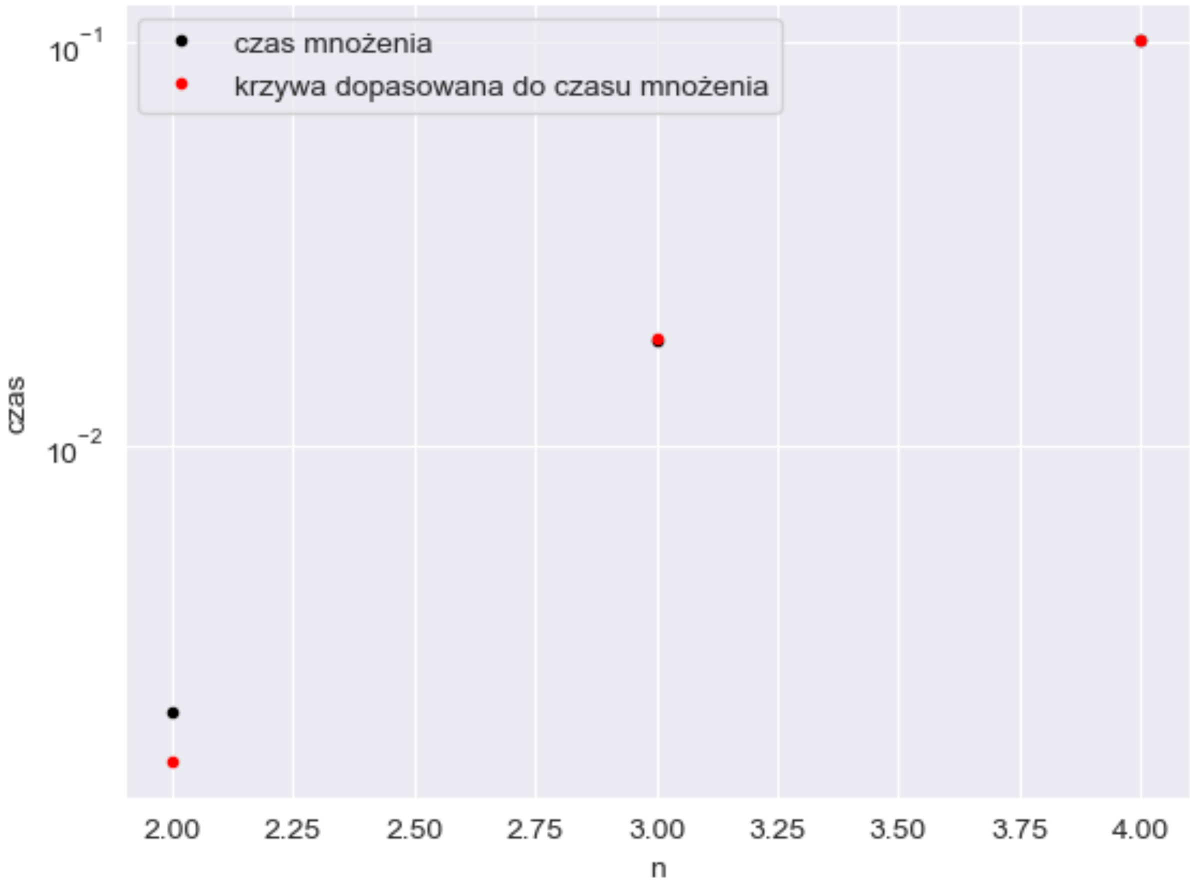
Złożoność obliczeniową szacowaliśmy empirycznie przy użyciu funkcji curve_fit z modułu scipy.optimize, która aproksymuje funkcję przy użyciu metody najmniejszych kwadratów. My próbowaliśmy aproksymować dane do funkcji postaci:

$$y = a \cdot x^k \tag{1}$$

Gdzie próbowaliśmy oszacować a oraz k.

Na podstawie czasu mnożenia macierzy metodą Bineta otrzymaliśmy:

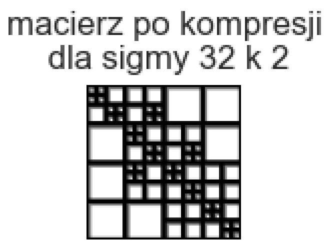
$$a = 2{,}65 \cdot 10^{-5}$$
$$k = 5{,}94 \tag{2}$$



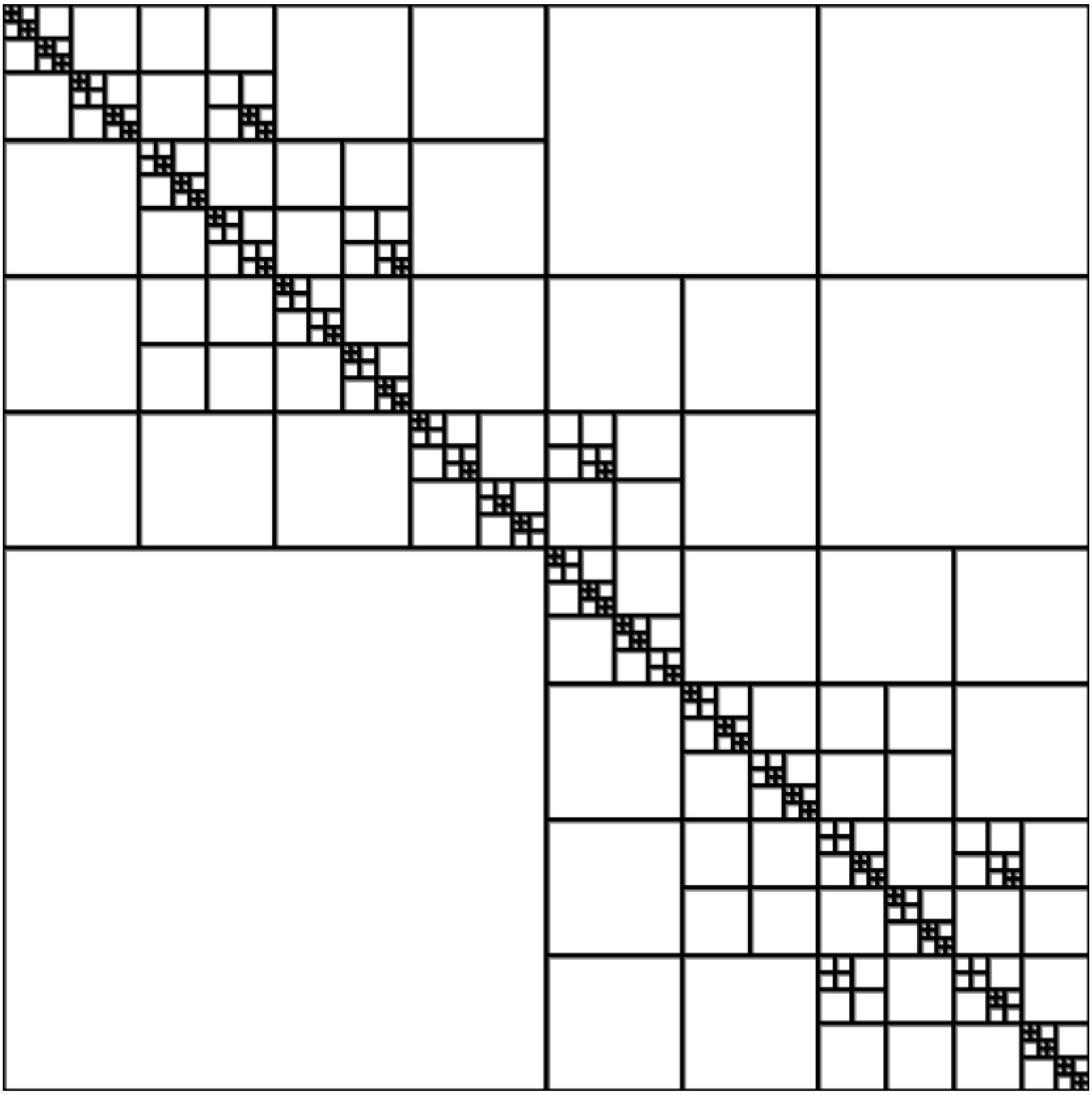
4.2 Pomiary mnożenia macierzy razy samą siebie

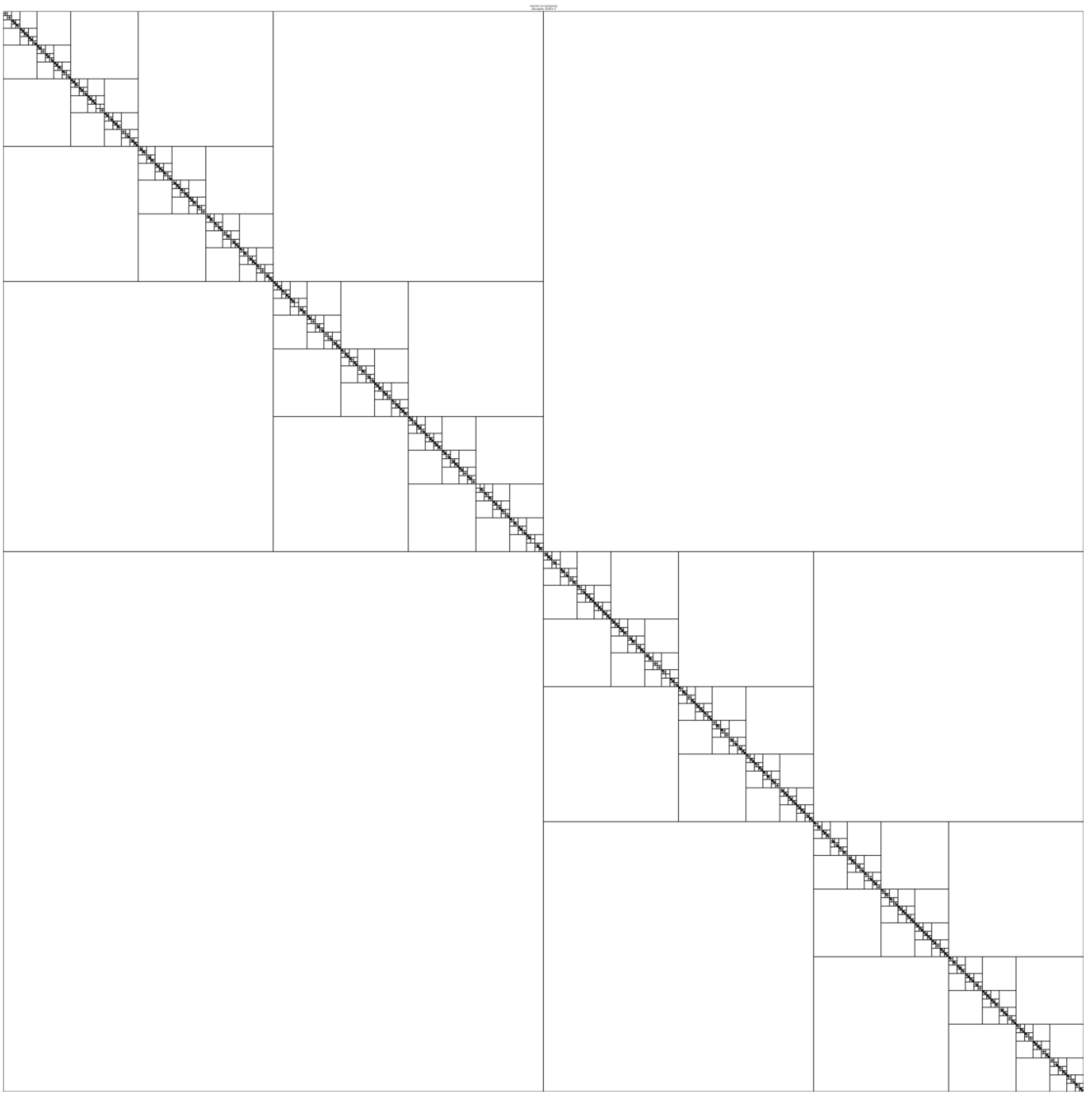
k	s	time	error
2	32	1.436627	168.266712
3	256	94.762856	2058.513895
4	2048	7836.314434	21588.409336

Tab.2 pomiary mnożenia macierzy skompresowanych



macierz po kompresji
dla sigmy 256 k 3





Złożoność obliczeniową szacowaliśmy empirycznie przy użyciu funkcji curve_fit z modułu scipy.optimize, która aproksymuje funkcję przy użyciu metody najmniejszych kwadratów. My próbowaliśmy aproksymować dane do funkcji postaci:

$$y = a \cdot x^k \tag{1}$$

Gdzie próbowaliśmy oszacować a oraz k.

Na podstawie czasu mnożenia macierzy metodą Bineta otrzymaliśmy:

$$a = 4{,}51 \cdot 10^{-6}$$
$$k = 15{,}35 \tag{2}$$

Rys.2

5. Wnioski

-