



AGH

Ćwiczenie 4

Algorytmy macierzowe

Autorzy:

Gabriel Kaźmierczak

Dariusz Piwowarski

Spis treści

1	Pseudokod	3
1.1	Procedura <i>matrix_vector_mult</i>	3
1.2	Procedura <i>split_compressed</i>	3
1.3	Procedura <i>matrix_matrix_add</i>	4
1.4	Procedura <i>matrix_matrix_mult</i>	5
1.5	Procedura <i>multiply_recursive</i>	6
2	Fragmenty kodu	7
2.1	Funkcje pomocnicze	7
2.2	Mnożenie macierzy przez wektor	8
2.3	Dodawanie macierzy	8
2.4	Mnożenie macierzy	10
3	Testy implementacji algorytmu mnożenia macierzy przez wektor	11
3.1	Sprawdzenie poprawności wyniku dla macierzy 256x256	11
3.2	Pomiar czasu wykonania	13
4	Testy implementacji algorytmu dodawania dwóch macierzy	16
4.1	Sprawdzenie poprawności wyniku dla macierzy 256x256	16
4.2	Pomiar czasu wykonania	19
5	Testy implementacji algorytmu dodawania mnożenia macierzy	22
5.1	Sprawdzenie poprawności wyniku dla macierzy 256x256	22
5.2	Pomiar czasu wykonania	24

1 Pseudokod

1.1 Procedura *matrix_vector_mult*

Require: $v : \text{Node}, X : \text{vectors}$

```

if  $\text{len}(v.\text{children}) = 0$  then
    if  $v.\text{rank} > 0$  then
        return  $v.U * v.Sigma * v.VT * X$ 
    else
        return  $\text{zeros}(X.\text{shape})$ 
    end if
end if
 $X1, X2 \leftarrow \text{split\_horizontal}(X)$ 
 $Y1 \leftarrow \text{matrix\_vector\_mult}(v.\text{children}[0], X1)$ 
 $Y2 \leftarrow \text{matrix\_vector\_mult}(v.\text{children}[1], X2)$ 
 $Y3 \leftarrow \text{matrix\_vector\_mult}(v.\text{children}[2], X1)$ 
 $Y4U, Sigma, VT, rmatrix\_vector\_mult(v.\text{children}[3], X2)$ 
return  $\begin{bmatrix} Y1 + Y2 \\ Y3 + Y4 \end{bmatrix}$ 

```

1.2 Procedura *split_compressed*

Require: $v : \text{Node}$

```

 $\text{nodes} \leftarrow [\text{Node}(\text{rank} = v.\text{rank}), \text{Node}(\text{rank} = v.\text{rank}),$ 
     $\text{Node}(\text{rank} = v.\text{rank}), \text{Node}(\text{rank} = v.\text{rank})]$ 
 $U1, U2 \leftarrow \text{split\_horizontal}(v.U)$ 
 $VT1, VT2 \leftarrow \text{split\_vertical}(v.VT)$ 
 $\text{nodes}[0].U \leftarrow U1$ 
 $\text{nodes}[0].Sigma \leftarrow v.Sigma$ 
 $\text{nodes}[0].VT \leftarrow VT1$ 
 $\text{nodes}[1].U \leftarrow U1$ 
 $\text{nodes}[1].Sigma \leftarrow v.Sigma$ 
 $\text{nodes}[1].VT \leftarrow VT2$ 
 $\text{nodes}[2].U \leftarrow U2$ 
 $\text{nodes}[2].Sigma \leftarrow v.Sigma$ 
 $\text{nodes}[2].VT \leftarrow VT1$ 
 $\text{nodes}[3].U \leftarrow U2$ 
 $\text{nodes}[3].Sigma \leftarrow v.Sigma$ 
 $\text{nodes}[3].VT \leftarrow VT2$ 

```

1.3 Procedura *matrix_matrix_add*

Require: $v : \text{Node}, w : \text{Node}$

```

if  $\text{len}(v.\text{children}) > 0$  then
  if  $\text{len}(v.\text{children}) = 0$  and  $\text{len}(w.\text{children}) = 0$  then
    return  $\text{Node}(\text{rank} = 0)$ 
  else if  $v.\text{rank} = 0$  then
    return  $w$ 
  else if  $w.\text{rank} = 0$  then
    return  $v$ 
  else
     $U, \text{Sigma}, VT, r \leftarrow \text{recompress}([v.U \ w.U], [v.\text{Sigma} \ w.\text{Sigma}], \begin{bmatrix} v.VT \\ w.VT \end{bmatrix})$ 
     $\text{node} \leftarrow \text{Node}(\text{rank} = r)$ 
     $\text{node}.U \leftarrow U$ 
     $\text{node}.\text{Sigma} \leftarrow \text{Sigma}$ 
     $\text{node}.VT \leftarrow VT$ 
    return  $\text{node}$ 
  end if
else if  $\text{len}(v.\text{children}) > 0$  and  $\text{len}(w.\text{children}) > 0$  then
   $\text{node} \leftarrow \text{Node}()$ 
   $\text{node}.\text{children} \leftarrow [$ 
     $\text{matrix\_matrix\_add}(v.\text{children}[0], w.\text{children}[0]),$ 
     $\text{matrix\_matrix\_add}(v.\text{children}[1], w.\text{children}[1]),$ 
     $\text{matrix\_matrix\_add}(v.\text{children}[2], w.\text{children}[2]),$ 
     $\text{matrix\_matrix\_add}(v.\text{children}[3], w.\text{children}[3])$ 
   $]$ 
  return  $\text{node}$ 
else if  $\text{len}(v.\text{children}) = 0$  and  $\text{len}(w.\text{children}) > 0$  then
  if  $v.\text{rank} = 0$  then
    return  $w$ 
  end if
   $\text{nodes} \leftarrow \text{split\_compressed}(v)$ 
   $\text{node} \leftarrow \text{Node}()$ 
   $\text{node}.\text{children} \leftarrow [$ 
     $\text{matrix\_matrix\_add}(v.\text{nodes}[0], w.\text{children}[0]),$ 
     $\text{matrix\_matrix\_add}(v.\text{nodes}[1], w.\text{children}[1]),$ 
     $\text{matrix\_matrix\_add}(v.\text{nodes}[2], w.\text{children}[2]),$ 
     $\text{matrix\_matrix\_add}(v.\text{nodes}[3], w.\text{children}[3])$ 
   $]$ 
  return  $\text{node}$ 
else
  if  $w.\text{rank} = 0$  then
    return  $v$ 
  end if

```

```

    nodes ← split_compressed(w)
    node ← Node()
    node.children ← [
        matrix_matrix_add(v.children[0], w.nodes[0]),
        matrix_matrix_add(v.children[1], w.nodes[1]),
        matrix_matrix_add(v.children[2], w.nodes[2]),
        matrix_matrix_add(v.children[3], w.nodes[3])
    ]
    return node
end if

```

1.4 Procedura *matrix_matrix_mult*

Require: $v : \text{Node}, w : \text{Node}$

```

if len(v.children) > 0 then
    if len(v.children) = 0 or len(w.children) = 0 then
        return Node(rank = 0)
    else
        node ← Node(rank = v.rank)
        node.U ← v.U
        node.Sigma ← v.Sigma
        node.VT ← (v.VT * w.U * w.Sigma) * w.VT
        return node
    end if
else if len(v.children) > 0 and len(w.children) > 0 then
    return multiply_recursive(v, w)
else if len(v.children) = 0 and len(w.children) > 0 then
    if v.rank = 0 then
        return w
    end if
    return multiply_recursive(v, w)
else
    if w.rank = 0 then
        return v
    end if
    return multiply_recursive(v, w)
end if

```

1.5 Procedura *multiply_recursive*

Require: $v : \text{Node}, w : \text{Node}$

$A \leftarrow v.children$

$B \leftarrow w.children$

if $len(A) = 0$ **then**

$A \leftarrow split_compressed(v)$

end if

if $len(B) = 0$ **then**

$B \leftarrow split_compressed(w)$

end if

$node = \text{Node}(v.n, v.m, \text{None})$

$node.children = [$

$\quad matrix_matrix_add(matrix_matrix_mult(A[0], B[0]), matrix_matrix_mult(A[1], B[2])),$

$\quad matrix_matrix_add(matrix_matrix_mult(A[0], B[1]), matrix_matrix_mult(A[1], B[3])),$

$\quad matrix_matrix_add(matrix_matrix_mult(A[2], B[0]), matrix_matrix_mult(A[3], B[2])),$

$\quad matrix_matrix_add(matrix_matrix_mult(A[2], B[1]), matrix_matrix_mult(A[3], B[3]))$

$]$

return $node$

2 Fragmenty kodu

2.1 Funkcje pomocnicze

```
1 def split(X: np.array):
2     n = X.shape[0] // 2
3     return X[:n], X[n:]
4
5
6 def split_horizontal(X: np.ndarray):
7     n = X.shape[0] // 2
8     return X[:n, :], X[n:, :]
9
10
11 def split_vertical(X: np.ndarray):
12     n = X.shape[1] // 2
13     return X[:, :n], X[:, n:]
14
15
16 def split_compressed(v: Node):
17     n, m = v.n // 2, v.m // 2
18     nodes = [Node(n, m, v.rank), Node(n, m, v.rank), Node(n, m, v.rank),
19             Node(n, m, v.rank)]
20     U1, U2 = split_horizontal(v.U)
21     VT1, VT2 = split_vertical(v.VT)
22     nodes[0].U = U1
23     nodes[0].Sigma = v.Sigma
24     nodes[0].VT = VT1
25     nodes[1].U = U1
26     nodes[1].Sigma = v.Sigma
27     nodes[1].VT = VT2
28     nodes[2].U = U2
29     nodes[2].Sigma = v.Sigma
30     nodes[2].VT = VT1
31     nodes[3].U = U2
32     nodes[3].Sigma = v.Sigma
33     nodes[3].VT = VT2
34     return nodes
```

2.2 Mnożenie macierzy przez wektor

```

1 def matrix_vector_mult(v: Node, X: np.ndarray):
2     if len(v.children) == 0:
3         if v.rank > 0:
4             # Przypadek, gdzie mamy skompresowaną macierz w postaci U,
             Sigma, VT
5             return v.U * v.Sigma @ v.VT @ X
6         else:
7             # Przypadek, gdy mamy skompresowaną macierz składającą się z
             samych zer
8             return np.zeros(X.shape)
9
10    # Przypadek, gdy macierz została podzielona na 4 pod-macierze
11    X1, X2 = split_horizontal(X)
12    Y1 = matrix_vector_mult(v.children[0], X1)
13    Y2 = matrix_vector_mult(v.children[1], X2)
14    Y3 = matrix_vector_mult(v.children[2], X1)
15    Y4 = matrix_vector_mult(v.children[3], X2)
16    return np.vstack((Y1 + Y2, Y3 + Y4))

```

2.3 Dodawanie macierzy

```

1 def recompress(A, B, epsilon):
2     Qa, Ra = np.linalg.qr(A, mode="reduced")
3     Qb, Rb = np.linalg.qr(B.T, mode="reduced")
4
5     U, Sigma, VT = np.linalg.svd(Ra @ Rb.T)
6
7     for r in range(0, Sigma.shape[0]):
8         if Sigma[r] < epsilon:
9             return Qa @ U[:, :r], Sigma[:r], (Qb @ VT.T[:, :r]).T, r
10
11    return Qa @ U, Sigma, (Qb @ VT.T).T, Sigma.shape[0]
12
13
14 def addition(v: Node, w: Node, epsilon):
15     U = np.hstack((v.U, w.U))
16     Sigma = np.hstack((v.Sigma, w.Sigma))
17     VT = np.vstack((v.VT, w.VT))
18     U, Sigma, VT, r = recompress(U * Sigma, VT, epsilon=epsilon)
19     node = Node(v.n, v.m, r)
20     node.U = U
21     node.Sigma = Sigma
22     node.VT = VT
23     return node
24
25

```



```
26
27 def matrix_matrix_add(v: Node, w: Node, epsilon):
28     if len(v.children) == 0 and len(w.children) == 0:
29         if v.rank == 0 and w.rank == 0:
30             return Node(v.n, v.m, 0)
31         elif v.rank == 0:
32             return w
33         elif w.rank == 0:
34             return v
35         else:
36             return addition(v, w, epsilon=epsilon)
37     elif len(v.children) > 0 and len(w.children) > 0:
38         node = Node(v.n, v.m, None)
39         node.children = [
40             matrix_matrix_add(v.children[0], w.children[0], epsilon=
epsilon),
41             matrix_matrix_add(v.children[1], w.children[1], epsilon=
epsilon),
42             matrix_matrix_add(v.children[2], w.children[2], epsilon=
epsilon),
43             matrix_matrix_add(v.children[3], w.children[3], epsilon=
epsilon)
44         ]
45         return node
46     elif len(v.children) == 0 and len(w.children) > 0:
47         if v.rank == 0:
48             return w
49         nodes = split_compressed(v)
50         node = Node(v.n, v.m, None)
51         node.children = [
52             matrix_matrix_add(nodes[0], w.children[0], epsilon=epsilon),
53             matrix_matrix_add(nodes[1], w.children[1], epsilon=epsilon),
54             matrix_matrix_add(nodes[2], w.children[2], epsilon=epsilon),
55             matrix_matrix_add(nodes[3], w.children[3], epsilon=epsilon)
56         ]
57         return node
58     else:
59         if w.rank == 0:
60             return v
61         nodes = split_compressed(w)
62         node = Node(v.n, v.m, None)
63         node.children = [
64             matrix_matrix_add(v.children[0], nodes[0], epsilon=epsilon),
65             matrix_matrix_add(v.children[1], nodes[1], epsilon=epsilon),
66             matrix_matrix_add(v.children[2], nodes[2], epsilon=epsilon),
67             matrix_matrix_add(v.children[3], nodes[3], epsilon=epsilon)
68         ]
69         return node
```

2.4 Mnożenie macierzy

```

1 def multiply_recursive(v: Node, w: Node, epsilon):
2     A = v.children
3     B = w.children
4     if len(A) == 0:
5         A = split_compressed(v)
6     if len(B) == 0:
7         B = split_compressed(w)
8
9     node = Node(v.n, v.m, None)
10    node.children = [
11        matrix_matrix_add(matrix_matrix_mult(A[0], B[0], epsilon),
12        matrix_matrix_mult(A[1], B[2], epsilon), epsilon),
13        matrix_matrix_add(matrix_matrix_mult(A[0], B[1], epsilon),
14        matrix_matrix_mult(A[1], B[3], epsilon), epsilon),
15        matrix_matrix_add(matrix_matrix_mult(A[2], B[0], epsilon),
16        matrix_matrix_mult(A[3], B[2], epsilon), epsilon),
17        matrix_matrix_add(matrix_matrix_mult(A[2], B[1], epsilon),
18        matrix_matrix_mult(A[3], B[3], epsilon), epsilon)
19    ]
20    return node
21
22 def matrix_matrix_mult(v: Node, w: Node, epsilon):
23     if len(v.children) == 0 and len(w.children) == 0:
24         if v.rank == 0 or w.rank == 0:
25             return Node(v.n, v.m, 0)
26         else:
27             node = Node(v.n, v.m, rank=v.rank)
28             node.U = v.U
29             node.Sigma = v.Sigma
30             node.VT = (v.VT @ w.U * w.Sigma) @ w.VT
31             return node
32     if len(v.children) > 0 and len(w.children) > 0:
33         return multiply_recursive(v, w, epsilon=epsilon)
34     if len(v.children) == 0 and len(w.children) > 0:
35         if v.rank == 0:
36             return Node(v.n, v.m, 0)
37         return multiply_recursive(v, w, epsilon=epsilon)
38     if len(v.children) > 0 and len(w.children) == 0:
39         if w.rank == 0:
40             return Node(w.n, w.m, 0)
41         return multiply_recursive(v, w, epsilon=epsilon)

```

3 Testy implementacji algorytmu mnożenia macierzy przez wektor

3.1 Sprawdzenie poprawności wyniku dla macierzy 256x256

```
169 def generate_matrix_compressed(n):
170     M = random_sparse(n, n, density=0.05).todense()
171     m = compress_matrix(M, r=2, epsilon=0.001)
172     return M, m
173
174
175 def test_matrix_vector_mult():
176     M1, m1 = generate_matrix_compressed(256)
177
178     m1.draw_matrix()
179     x = np.random.random((256, 1))
180     y = matrix_vector_mult(m1, x)
181
182     print(f"||y - M1 @ x||^2 = {np.sum(np.square(y - (M1 @ x)))}")
183
184
```

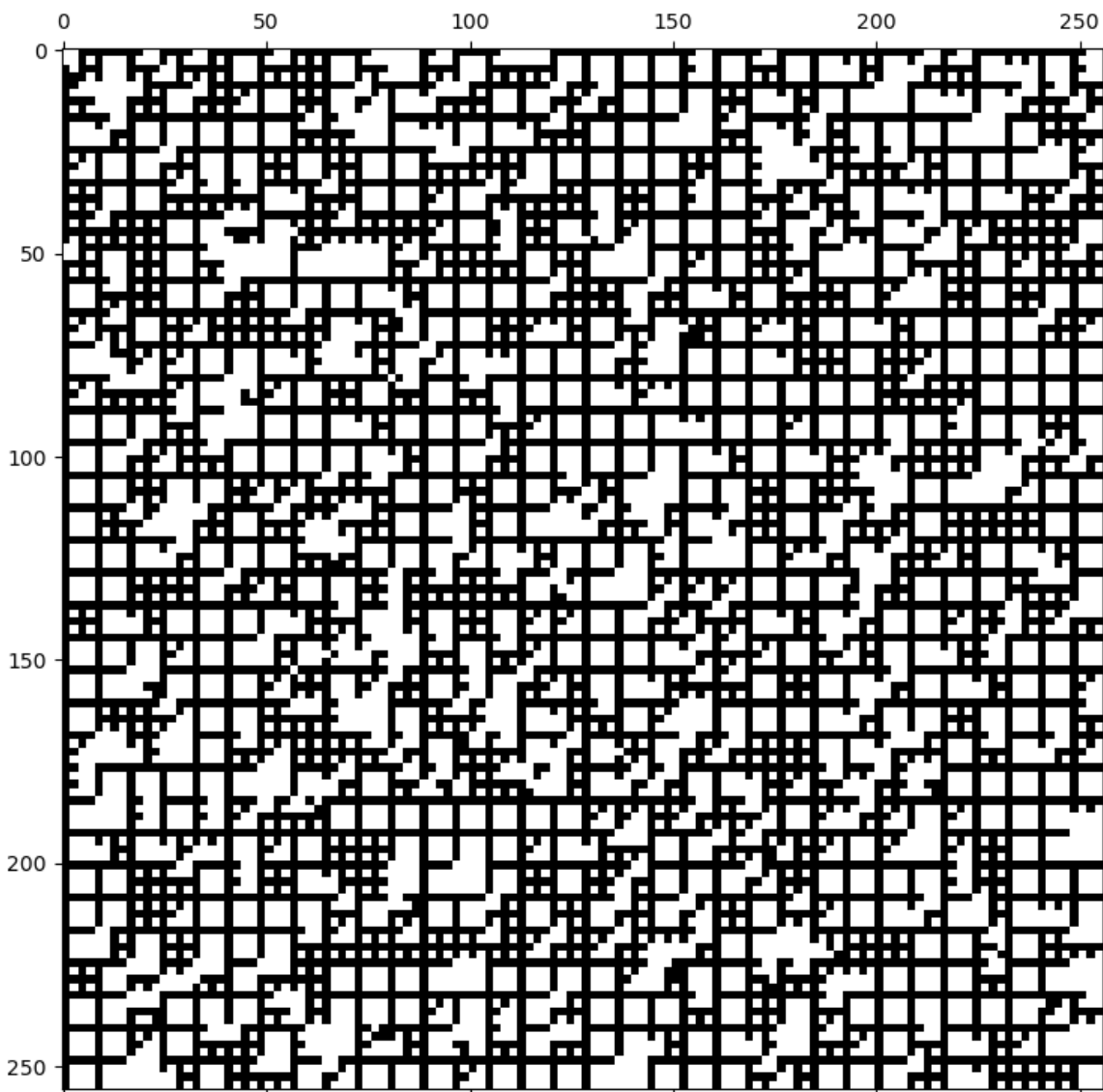
Run: multiplication ×

C:\ProgramData\Anaconda3\python.exe "C:\Users\Dariusz Piwowarski\Desktop\Studia\AM_WIET_2023\zad4\multiplication.py"

||y - M1 @ x||^2 = 2.399356445408462e-07

Rysunek 1: Suma różnicy kwadratów

Sprawdzamy poprawność mnożąc skompresowaną macierz rzadką (5% wartości niezerowych) przez wektor losowych wartości. Wynik porównujemy z standardowym mnożeniem macierzy przez wektor (w naszym przypadku wykonywane jest to za pomocą biblioteki NumPy w Pythonie). Jak widać, otrzymana suma różnicy kwadratów jest rzędu 10^{-7} , czyli możemy zakładać, że algorytm działa poprawnie.

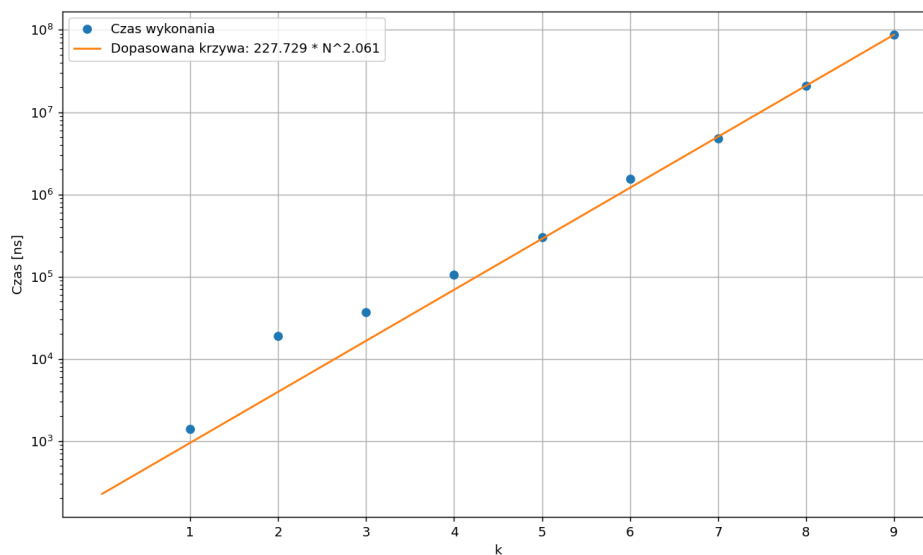


Rysunek 2: Macierz, którą mnożyliśmy przez wektor (skompresowana)

3.2 Pomiar czasu wykonania

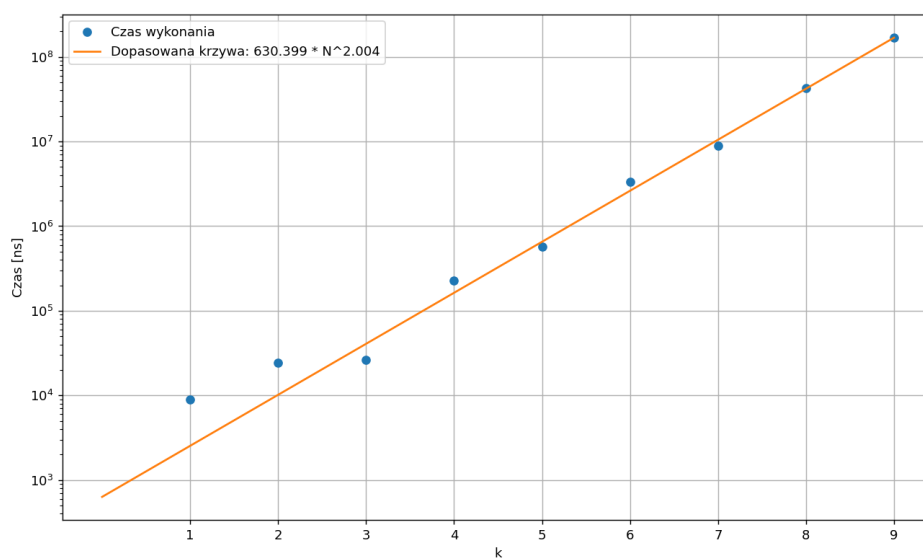
Oś Y na wykresach jest rysowana w skali logarytmicznej.

Macierze $2^k \times 2^k$, 10% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



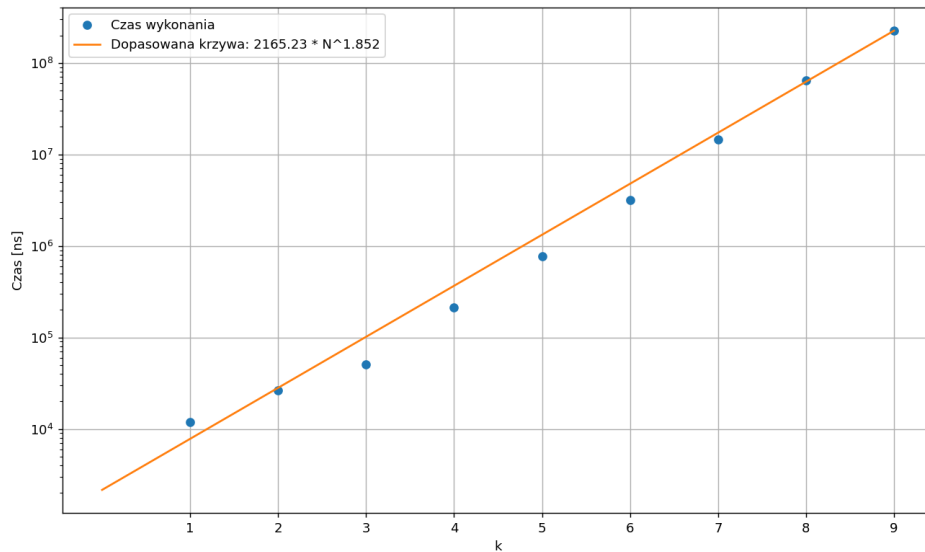
Rysunek 3: 10% wartości niezerowych

Macierze $2^k \times 2^k$, 20% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



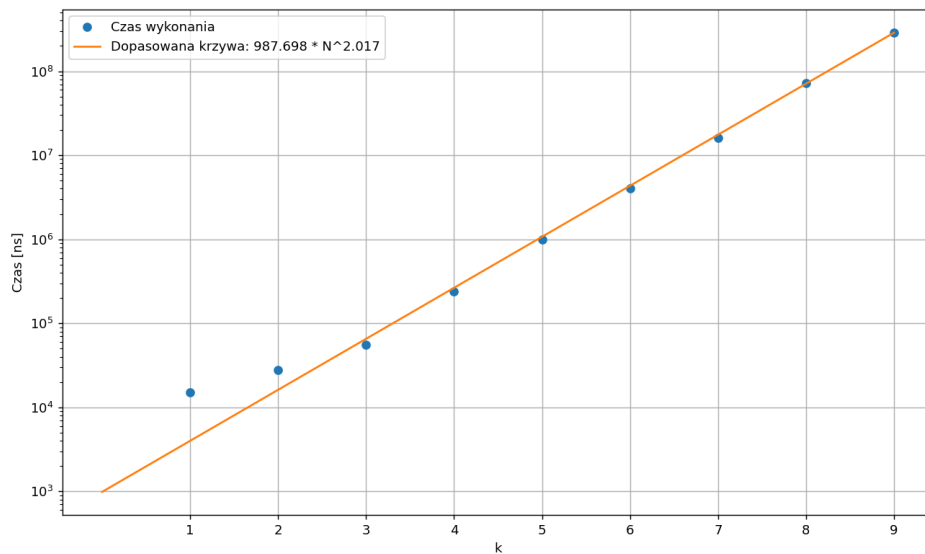
Rysunek 4: 20% wartości niezerowych

Macierze $2^k \times 2^k$, 30% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



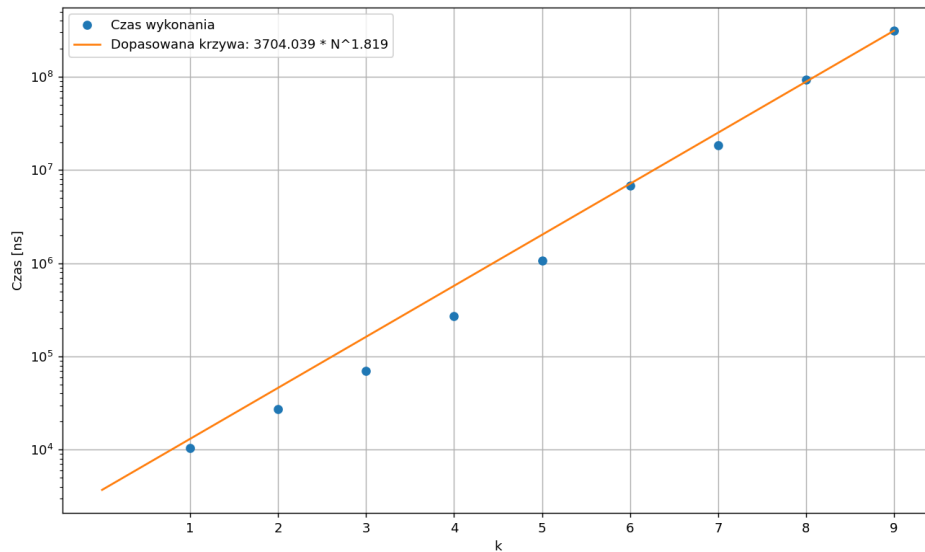
Rysunek 5: 30% wartości niezerowych

Macierze $2^k \times 2^k$, 40% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



Rysunek 6: 40% wartości niezerowych

Macierze $2^k \times 2^k$, 50% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



Rysunek 7: 50% wartości niezerowych

Zmierzona eksperymentalnie złożoność jest rzędu $O(n^2)$, czyli tyle samo co przy klasycznym mnożeniu nieskompresowanej macierzy przez wektor.

4 Testy implementacji algorytmu dodawania dwóch macierzy

4.1 Sprawdzenie poprawności wyniku dla macierzy 256x256

```
190 def generate_matrix_compressed(n):
191     M = random_sparse(n, n, density=0.01).todense()
192     m = compress_matrix(M, r=2, epsilon=0.001)
193     return M, m
194
195
196 def test_matrix_matrix_add():
197     M1, m1 = generate_matrix_compressed(256)
198     M2, m2 = generate_matrix_compressed(256)
199     m3 = matrix_matrix_add(m1, m2, epsilon=0.001)
200
201     m1.draw_matrix()
202     m2.draw_matrix()
203     m3.draw_matrix()
204
205     M3 = decompress(m3)
206     print(f"||M3 - (M1 + M2)||^2 = {np.sum(np.square(M3 - (M1 + M2)))}")
207
```

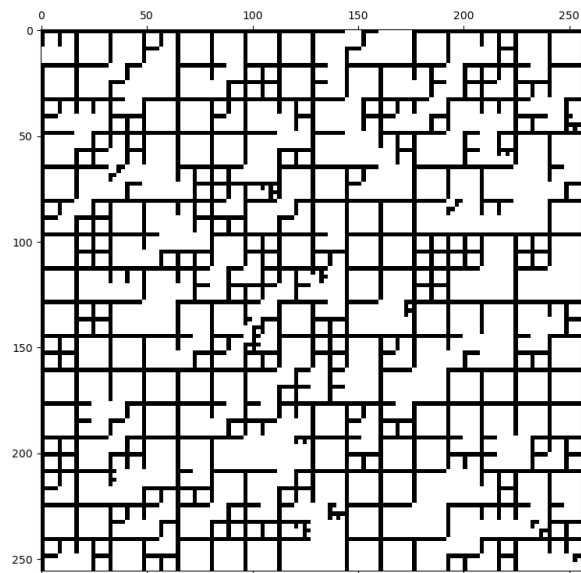
Run: multiplication ×

C:\ProgramData\Anaconda3\python.exe "C:\Users\Dariusz Piwowarski\Desktop\Studia\AM_WIET_2023\zad4\multiplication.py"

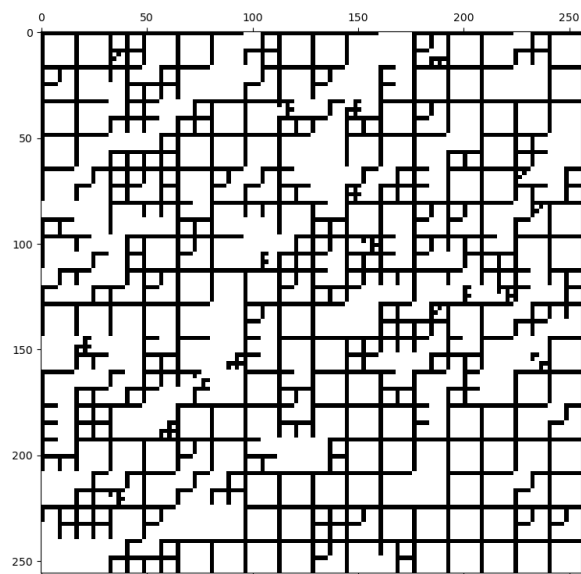
||M3 - (M1 + M2)||^2 = 4.737366906179064e-07

Rysunek 8: Suma różnicy kwadratów

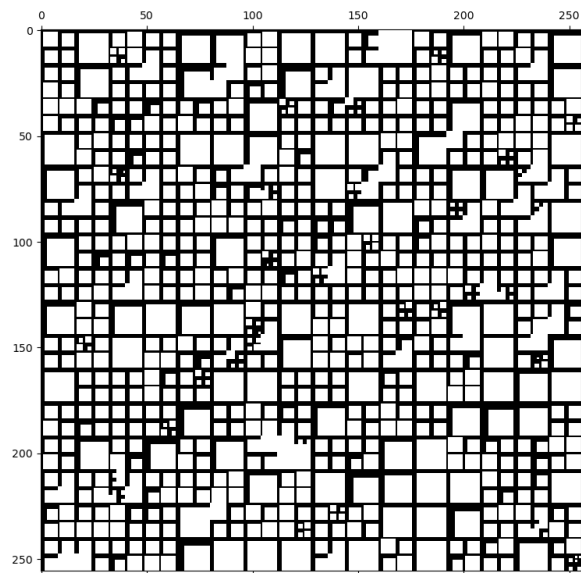
Sprawdzamy poprawność dodając do siebie 2 skompresowane macierze rzadkie (1% wartości niezerowych) o rozmiarze 256x256, a następnie dekompresujemy otrzymaną macierz. Wynik porównujemy z standardowym dodaniem macierzy (w naszym przypadku wykonywane jest to za pomocą biblioteki NumPy w Pythonie). Jak widać, otrzymana suma różnicy kwadratów jest rzędu 10^{-7} , czyli możemy zakładać, że algorytm działa poprawnie.



Rysunek 9: Macierz 1 po kompresji



Rysunek 10: Macierz 2 po kompresji

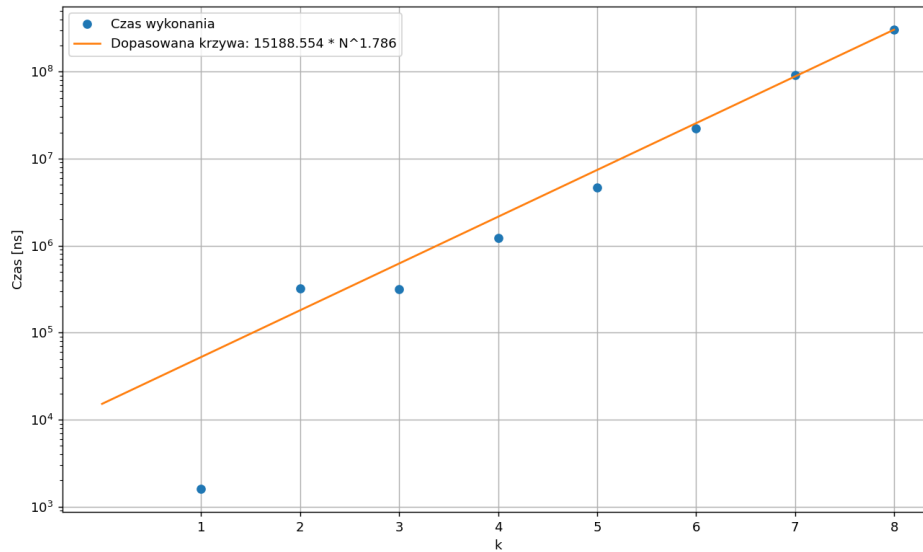


Rysunek 11: Suma macierzy

4.2 Pomiar czasu wykonania

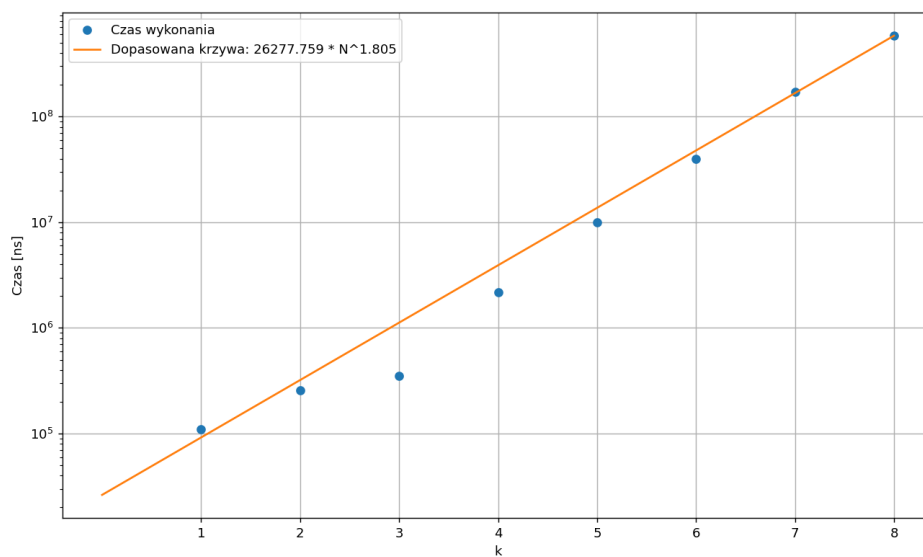
Oś Y na wykresach jest rysowana w skali logarytmicznej.

Macierze $2^k \times 2^k$, 10% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



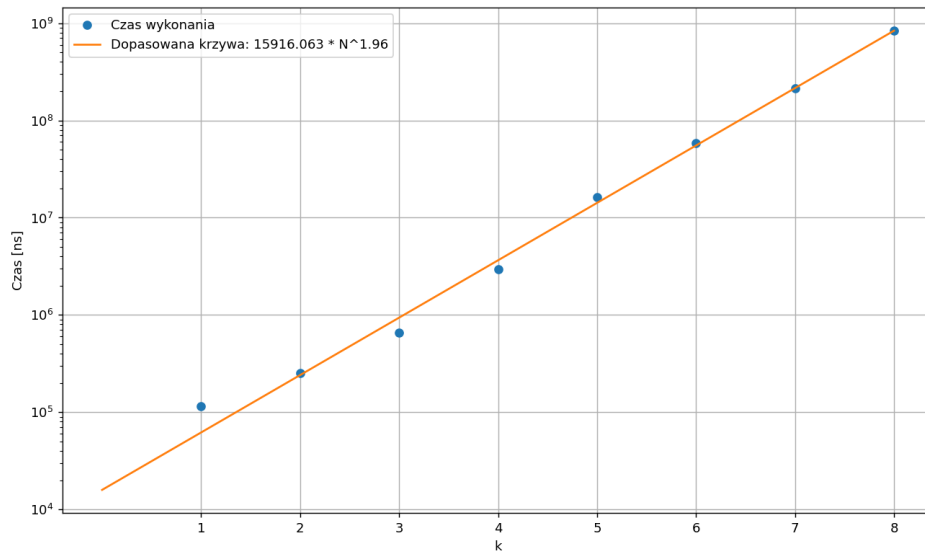
Rysunek 12: 10% wartości niezerowych

Macierze $2^k \times 2^k$, 20% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



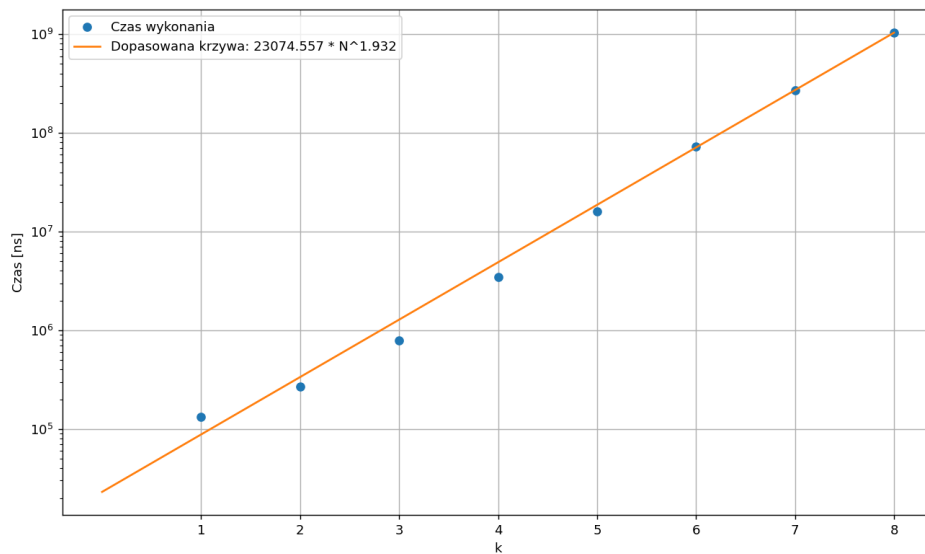
Rysunek 13: 20% wartości niezerowych

Macierze $2^k \times 2^k$, 30% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



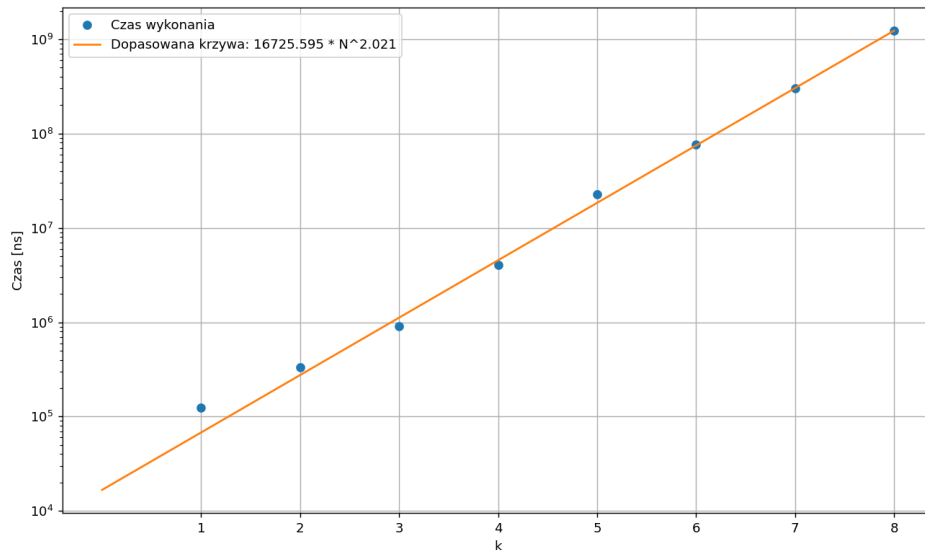
Rysunek 14: 30% wartości niezerowych

Macierze $2^k \times 2^k$, 40% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



Rysunek 15: 40% wartości niezerowych

Macierze $2^k \times 2^k$, 50% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



Rysunek 16: 50% wartości niezerowych

Zmierzona eksperymentalnie złożoność jest rzędu $O(n^2)$, czyli tyle samo co przy klasycznym dodawaniu nieskompresowanych macierzy.

5 Testy implementacji algorytmu dodawania mnożenia macierzy

5.1 Sprawdzenie poprawności wyniku dla macierzy 256x256



```
203 def test_matrix_matrix_mult():
204     M = random_sparse(256, 256, density=0.01).todense()
205     m1 = compress_matrix(M, r=2, epsilon=0.001)
206     m2 = compress_matrix(M, r=2, epsilon=0.001)
207     m3 = matrix_matrix_mult(m1, m2, epsilon=0.001)
208
209     m1.draw_matrix()
210     m3.draw_matrix()
211
212     M3 = decompress(m3)
213     print(f"||M3 - (M1 @ M1)||^2 = {np.sum(np.square(M3 - (M @ M)))}")
214
test_matrix_matrix_add()
```

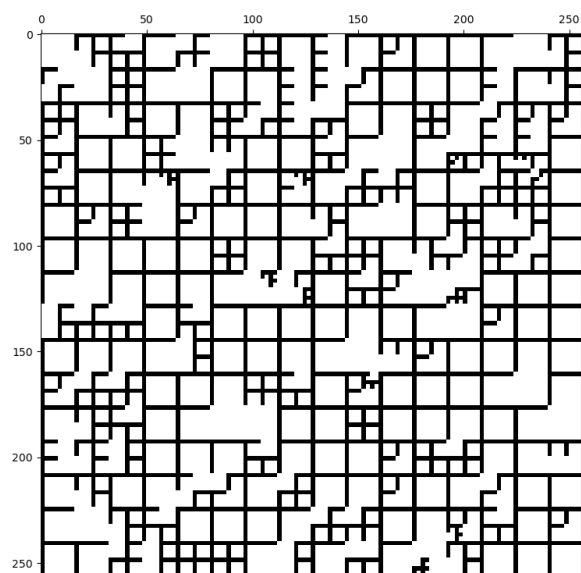
Run: multiplication ×

C:\ProgramData\Anaconda3\python.exe "C:\Users\Dariusz Piwowarski\Desktop\Studia\AM_WIET_2023\zad4\multiplication.py"

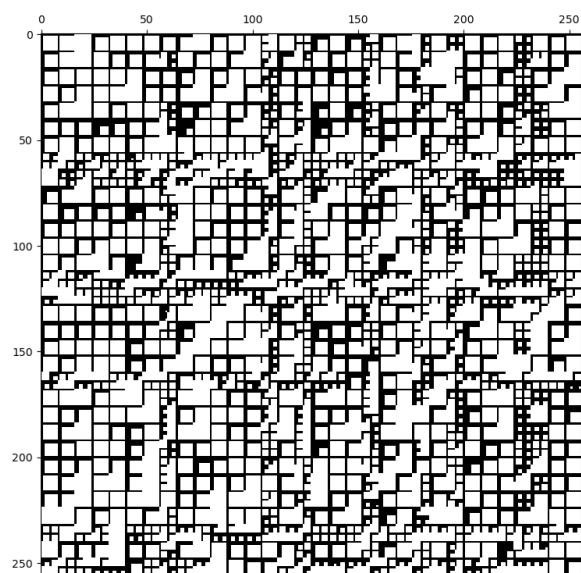
||M3 - (M1 @ M1)||^2 = 8.476135483480961e-07

Rysunek 17: Suma różnicy kwadratów

Sprawdzamy poprawność mnożąc macierz rzadką (1% wartości niezerowych, rozmiar 256x256) przez samą siebie, a następnie dekompresujemy otrzymaną macierz. Wynik porównujemy z standardowym mnożeniem macierzy (w naszym przypadku wykonywane jest to za pomocą biblioteki NumPy w Pythonie). Ponownie otrzymana suma różnicy kwadratów jest rzędu 10^{-7} , więc zakładamy, że algorytm działa poprawnie.



Rysunek 18: Macierz po kompresji

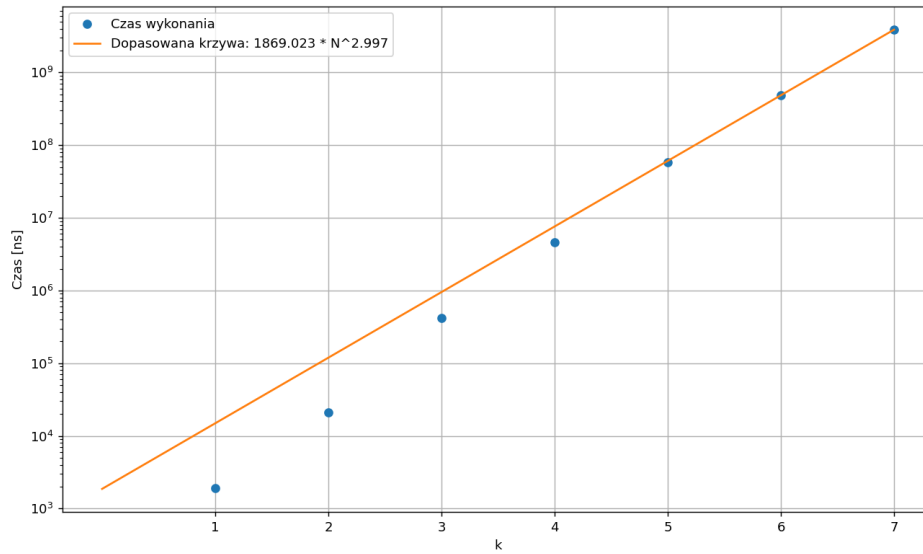


Rysunek 19: Macierz przemnożona przez samą siebie

5.2 Pomiar czasu wykonania

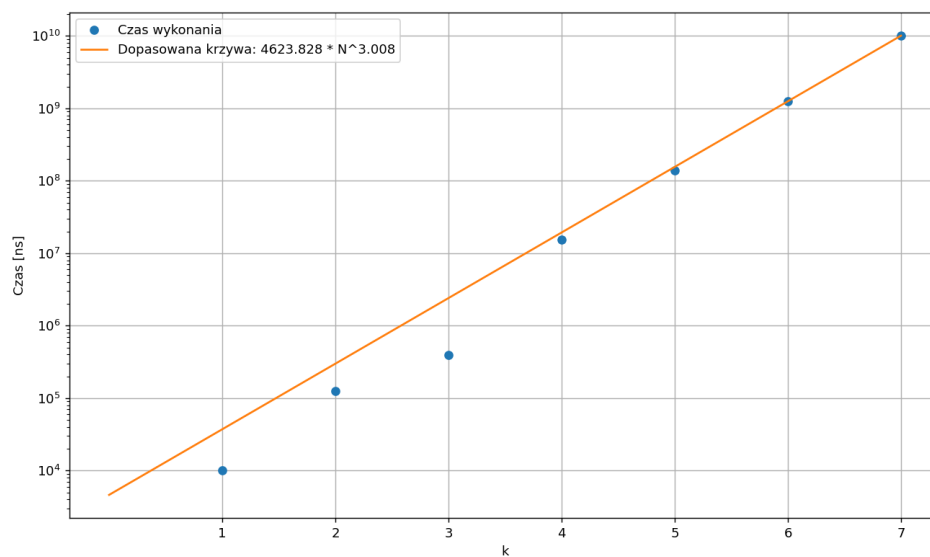
Oś Y na wykresach jest rysowana w skali logarytmicznej.

Macierze $2^k \times 2^k$, 10% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



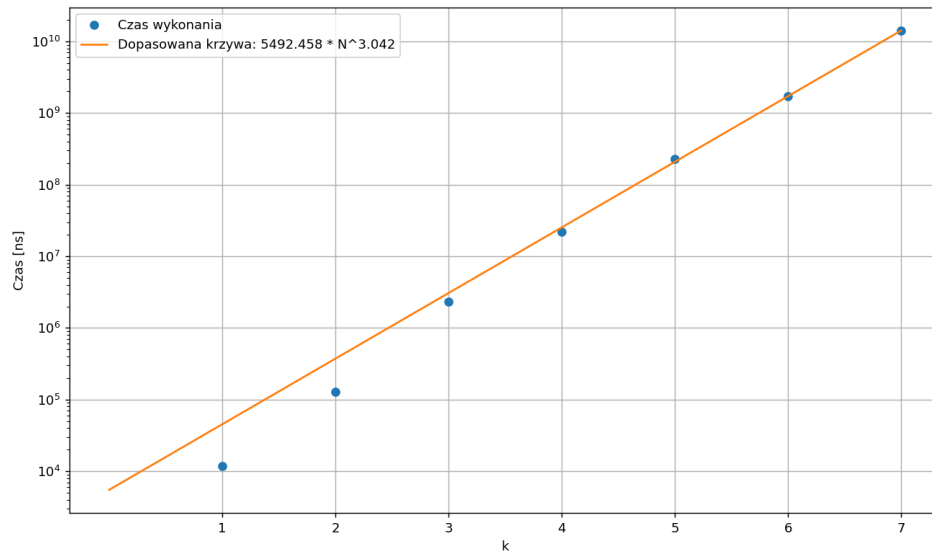
Rysunek 20: 10% wartości niezerowych

Macierze $2^k \times 2^k$, 20% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



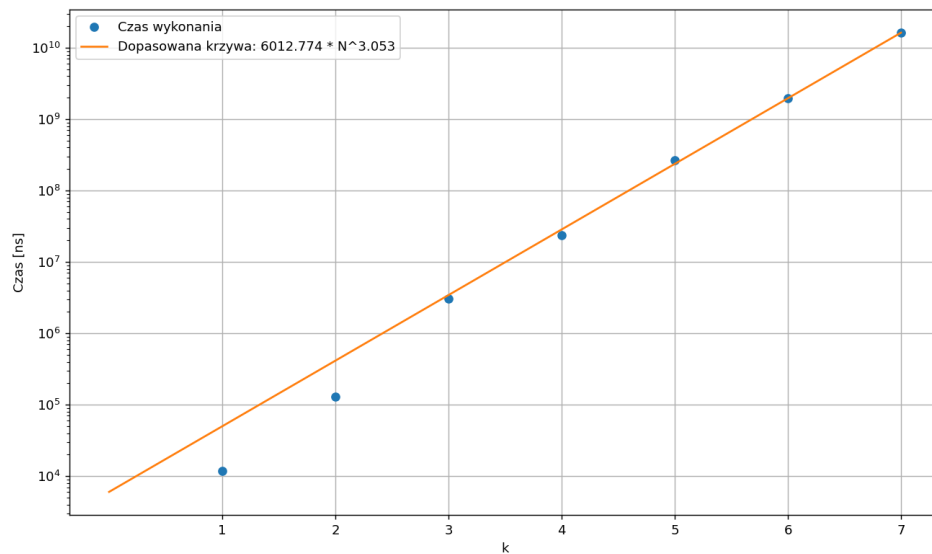
Rysunek 21: 20% wartości niezerowych

Macierze $2^k \times 2^k$, 30% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu

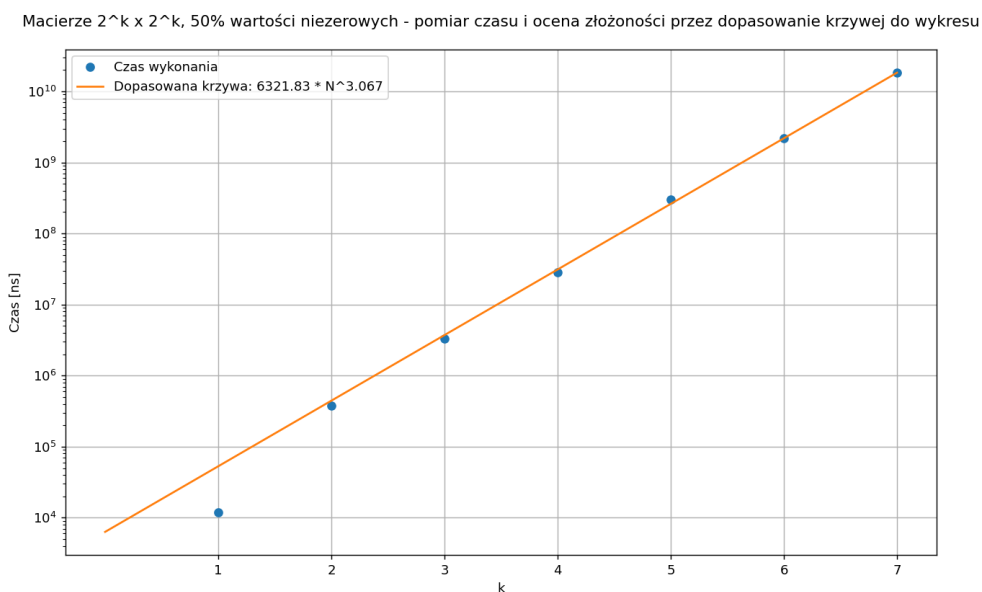


Rysunek 22: 30% wartości niezerowych

Macierze $2^k \times 2^k$, 40% wartości niezerowych - pomiar czasu i ocena złożoności przez dopasowanie krzywej do wykresu



Rysunek 23: 40% wartości niezerowych



Rysunek 24: 50% wartości niezerowych

Zmierzona eksperymentalnie złożoność jest rzędu $O(n^3)$, czyli tyle samo co przy klasycznym dodawaniu nieskompresowanych macierzy. W tym przypadku liczyliśmy na lepszą złożoność, którą powinna nam zapewnić kompresja, ale jak widać bez zastosowania algorytmów permutacji macierzy, nie udało nam się osiągnąć lepszej złożoności.