Ćwiczenie 4

Algorytmy macierzowe

Autorzy:
Gabriel Kaźmierczak
Dariusz Piwowarski

Spis treści

1 Pseudokod

1.1 Procedura *matrix_vector_mult*

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

1.2 Procedura *split_compressed*

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

1.3 Procedura *matrix_matrix_add*

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

$nodes \leftarrow split\_compressed(w)$
$node \leftarrow Node()$
$node.children \leftarrow [$
$matrix\_matrix\_add(v.children[0], w.nodes[0]),$
$matrix\_matrix\_add(v.children[1], w.nodes[1]),$
$matrix\_matrix\_add(v.children[2], w.nodes[2]),$
$matrix\_matrix\_add(v.children[3], w.nodes[3])$
$]$
return $node$
end if

1.4 Procedura *matrix_matrix_mult*

Require: $v : Node, w : Node$
if $len(v.children) > 0$ **then**
if $len(v.children) = 0$ **or** $len(w.children) = 0$ **then**
return $Node(rank = 0)$
else
$node \leftarrow Node(rank = v.rank)$
$node.U \leftarrow v.U$
$node.Sigma \leftarrow v.Sigma$
$node.VT \leftarrow (v.VT * w.U * w.Sigma) * w.VT$
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: $v : Node, w : Node$

$A \leftarrow v.children$
$B \leftarrow w.children$
if $len(A) = 0$ **then**
    $A \leftarrow split\_compressed(v)$
end if
if $len(B) = 0$ **then**
    $B \leftarrow split\_compressed(w)$
end if
$node = Node(v.n, v.m, None)$
$node.children = [$
    $matrix\_matrix\_add(matrix\_matrix\_mult(A[0], B[0]), matrix\_matrix\_mult(A[1], B[2])),$
    $matrix\_matrix\_add(matrix\_matrix\_mult(A[0], B[1]), matrix\_matrix\_mult(A[1], B[3])),$
    $matrix\_matrix\_add(matrix\_matrix\_mult(A[2], B[0]), matrix\_matrix\_mult(A[3], B[2])),$
    $matrix\_matrix\_add(matrix\_matrix\_mult(A[2], B[1]), matrix\_matrix\_mult(A[3], B[3]))$
$]$
return $node$

2 Fragmenty kodu

2.1 Funkcje pomocnicze

```
def split(X: np.array):
    n = X.shape[0] // 2
    return X[:n], X[n:]


def split_horizontal(X: np.ndarray):
    n = X.shape[0] // 2
    return X[:n, :], X[n:, :]


def split_vertical(X: np.ndarray):
    n = X.shape[1] // 2
    return X[:, :n], X[:, n:]


def split_compressed(v: Node):
    n, m = v.n // 2, v.m // 2
    nodes = [Node(n, m, v.rank), Node(n, m, v.rank), Node(n, m, v.rank),
   Node(n, m, v.rank)]
    U1, U2 = split_horizontal(v.U)
    VT1, VT2 = split_vertical(v.VT)
    nodes[0].U = U1
    nodes[0].Sigma = v.Sigma
    nodes[0].VT = VT1
    nodes[1].U = U1
    nodes[1].Sigma = v.Sigma
    nodes[1].VT = VT2
    nodes[2].U = U2
    nodes[2].Sigma = v.Sigma
    nodes[2].VT = VT1
    nodes[3].U = U2
    nodes[3].Sigma = v.Sigma
    nodes[3].VT = VT2
    return nodes
```

2.2 Mnożenie macierzy przez wektor

```
def matrix_vector_mult(v: Node, X: np.ndarray):
    if len(v.children) == 0:
        if v.rank > 0:
            # Przypadek, gdzie mamy skompresowaną macierz w postaci U, Sigma, VT
            return v.U * v.Sigma @ v.VT @ X
        else:
            # Przypadek, gdy mamy skompresowaną macierz składającą się z samych zer
            return np.zeros(X.shape)

    # Przypadek, gdy macierz została podzielona na 4 pod-macierze
    X1, X2 = split_horizontal(X)
    Y1 = matrix_vector_mult(v.children[0], X1)
    Y2 = matrix_vector_mult(v.children[1], X2)
    Y3 = matrix_vector_mult(v.children[2], X1)
    Y4 = matrix_vector_mult(v.children[3], X2)
    return np.vstack((Y1 + Y2, Y3 + Y4))
```

2.3 Dodawanie macierzy

```
def recompress(A, B, epsilon):
    Qa, Ra = np.linalg.qr(A, mode="reduced")
    Qb, Rb = np.linalg.qr(B.T, mode="reduced")

    U, Sigma, VT = np.linalg.svd(Ra @ Rb.T)

    for r in range(0, Sigma.shape[0]):
        if Sigma[r] < epsilon:
            return Qa @ U[:, :r], Sigma[:r], (Qb @ VT.T[:, :r]).T, r

    return Qa @ U, Sigma, (Qb @ VT.T).T, Sigma.shape[0]


def addition(v: Node, w: Node, epsilon):
    U = np.hstack((v.U, w.U))
    Sigma = np.hstack((v.Sigma, w.Sigma))
    VT = np.vstack((v.VT, w.VT))
    U, Sigma, VT, r = recompress(U * Sigma, VT, epsilon=epsilon)
    node = Node(v.n, v.m, r)
    node.U = U
    node.Sigma = Sigma
    node.VT = VT
    return node
```

```python

def matrix_matrix_add(v: Node, w: Node, epsilon):
    if len(v.children) == 0 and len(w.children) == 0:
        if v.rank == 0 and w.rank == 0:
            return Node(v.n, v.m, 0)
        elif v.rank == 0:
            return w
        elif w.rank == 0:
            return v
        else:
            return addition(v, w, epsilon=epsilon)
    elif len(v.children) > 0 and len(w.children) > 0:
        node = Node(v.n, v.m, None)
        node.children = [
            matrix_matrix_add(v.children[0], w.children[0], epsilon=epsilon),
            matrix_matrix_add(v.children[1], w.children[1], epsilon=epsilon),
            matrix_matrix_add(v.children[2], w.children[2], epsilon=epsilon),
            matrix_matrix_add(v.children[3], w.children[3], epsilon=epsilon)
        ]
        return node
    elif len(v.children) == 0 and len(w.children) > 0:
        if v.rank == 0:
            return w
        nodes = split_compressed(v)
        node = Node(v.n, v.m, None)
        node.children = [
            matrix_matrix_add(nodes[0], w.children[0], epsilon=epsilon),
            matrix_matrix_add(nodes[1], w.children[1], epsilon=epsilon),
            matrix_matrix_add(nodes[2], w.children[2], epsilon=epsilon),
            matrix_matrix_add(nodes[3], w.children[3], epsilon=epsilon)
        ]
        return node
    else:
        if w.rank == 0:
            return v
        nodes = split_compressed(w)
        node = Node(v.n, v.m, None)
        node.children = [
            matrix_matrix_add(v.children[0], nodes[0], epsilon=epsilon),
            matrix_matrix_add(v.children[1], nodes[1], epsilon=epsilon),
            matrix_matrix_add(v.children[2], nodes[2], epsilon=epsilon),
            matrix_matrix_add(v.children[3], nodes[3], epsilon=epsilon)
        ]
        return node
```

2.4 Mnożenie macierzy

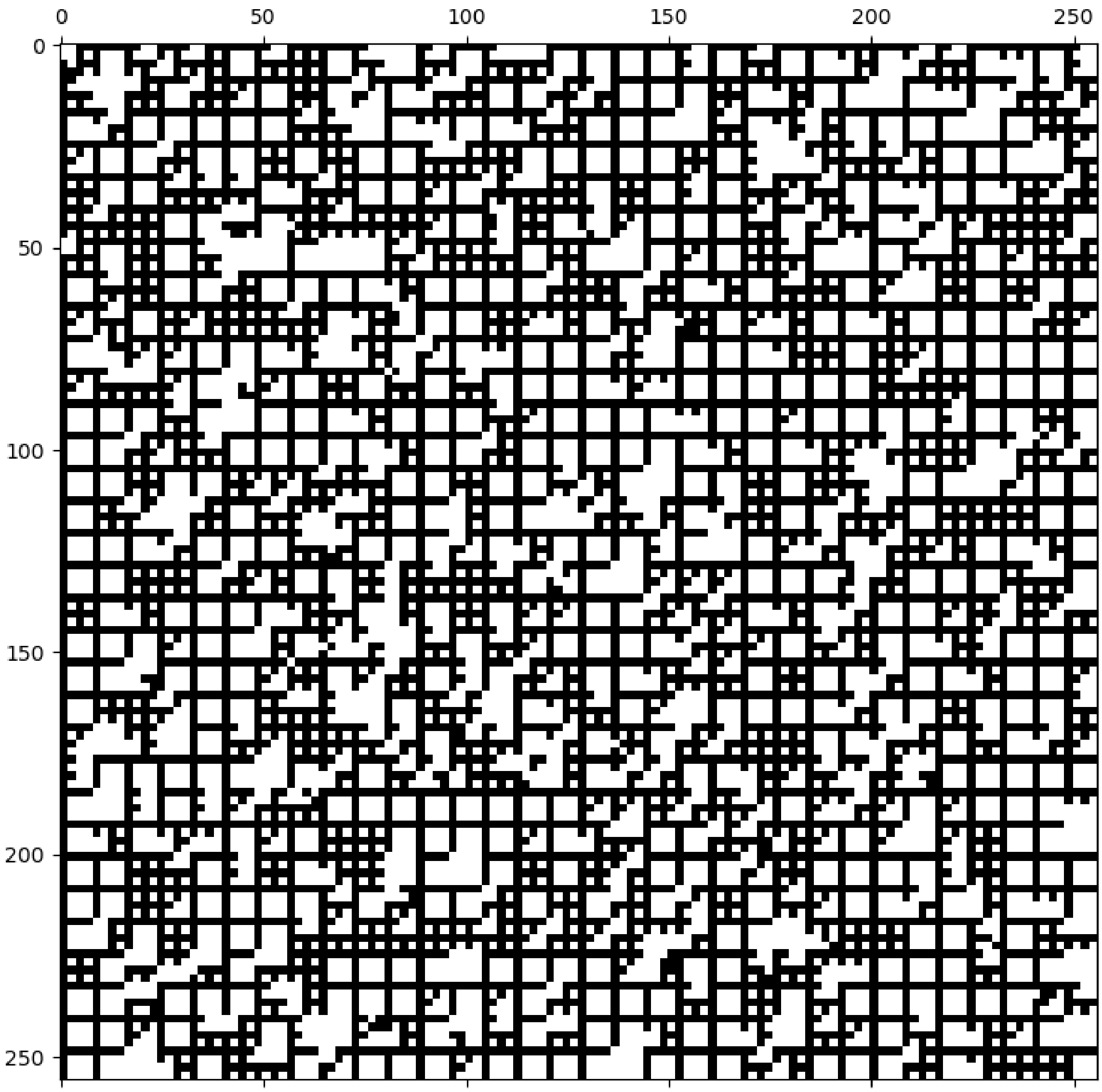
```
def multiply_recursive(v: Node, w: Node, epsilon):
    A = v.children
    B = w.children
    if len(A) == 0:
        A = split_compressed(v)
    if len(B) == 0:
        B = split_compressed(w)

    node = Node(v.n, v.m, None)
    node.children = [
        matrix_matrix_add(matrix_matrix_mult(A[0], B[0], epsilon),
   matrix_matrix_mult(A[1], B[2], epsilon), epsilon),
        matrix_matrix_add(matrix_matrix_mult(A[0], B[1], epsilon),
   matrix_matrix_mult(A[1], B[3], epsilon), epsilon),
        matrix_matrix_add(matrix_matrix_mult(A[2], B[0], epsilon),
   matrix_matrix_mult(A[3], B[2], epsilon), epsilon),
        matrix_matrix_add(matrix_matrix_mult(A[2], B[1], epsilon),
   matrix_matrix_mult(A[3], B[3], epsilon), epsilon)
    ]
    return node


def matrix_matrix_mult(v: Node, w: Node, epsilon):
    if len(v.children) == 0 and len(w.children) == 0:
        if v.rank == 0 or w.rank == 0:
            return Node(v.n, v.m, 0)
        else:
            node = Node(v.n, v.m, rank=v.rank)
            node.U = v.U
            node.Sigma = v.Sigma
            node.VT = (v.VT @ w.U * w.Sigma) @ w.VT
            return node
    if len(v.children) > 0 and len(w.children) > 0:
        return multiply_recursive(v, w, epsilon=epsilon)
    if len(v.children) == 0 and len(w.children) > 0:
        if v.rank == 0:
            return Node(v.n, v.m, 0)
        return multiply_recursive(v, w, epsilon=epsilon)
    if len(v.children) > 0 and len(w.children) == 0:
        if w.rank == 0:
            return Node(w.n, w.m, 0)
        return multiply_recursive(v, w, epsilon=epsilon)
```

3 Testy implementacji algorytmu mnożenia macierzy przez wektor

3.1 Sprawdzenie poprawności wyniku dla macierzy 256x256

Rysunek 1: Suma różnicy kwadratów

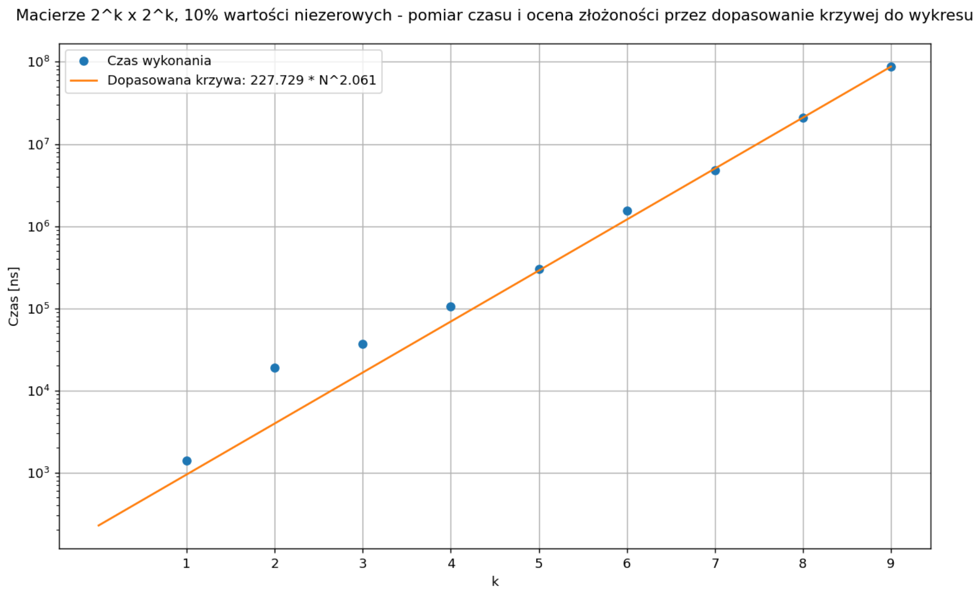
Sprawdzamy poprawność mnożąc skompresowaną macierz rzadką (5% wartości niezerowych) przez wektor losowych wartości. Wynik porównujemy z standardowym mnożeniem macierzy przez wektor (w naszym przypadku wykonywane jest to za pomocą biblioteki NumPy w Pythonie). Jak widać, otrzymana suma różnicy kwadratów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



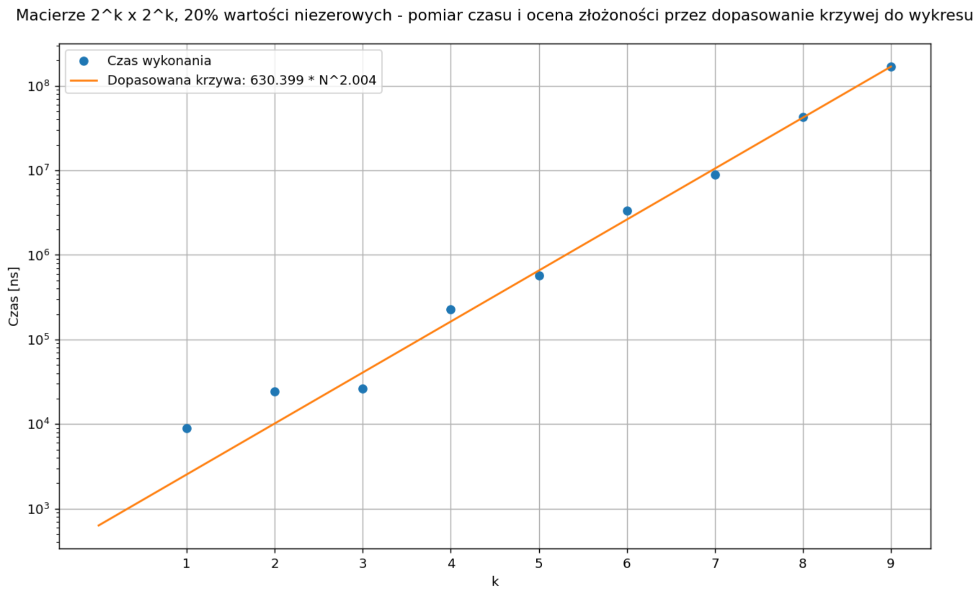
Rysunek 2: Macierz, którą mnożyliśmy przez wektor (skompresowana)

3.2 Pomiar czasu wykonania

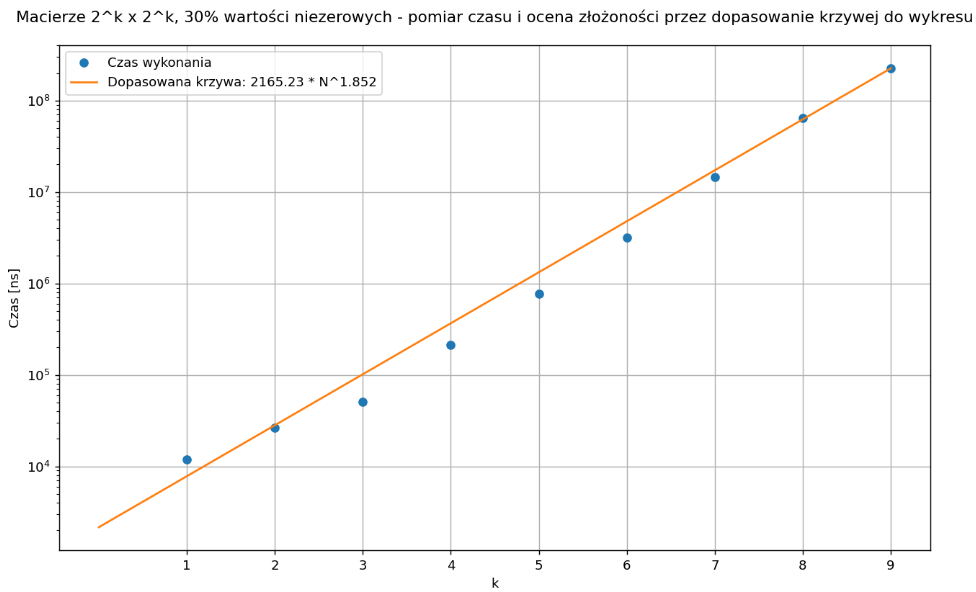
Oś Y na wykresach jest rysowana w skali logarytmicznej.



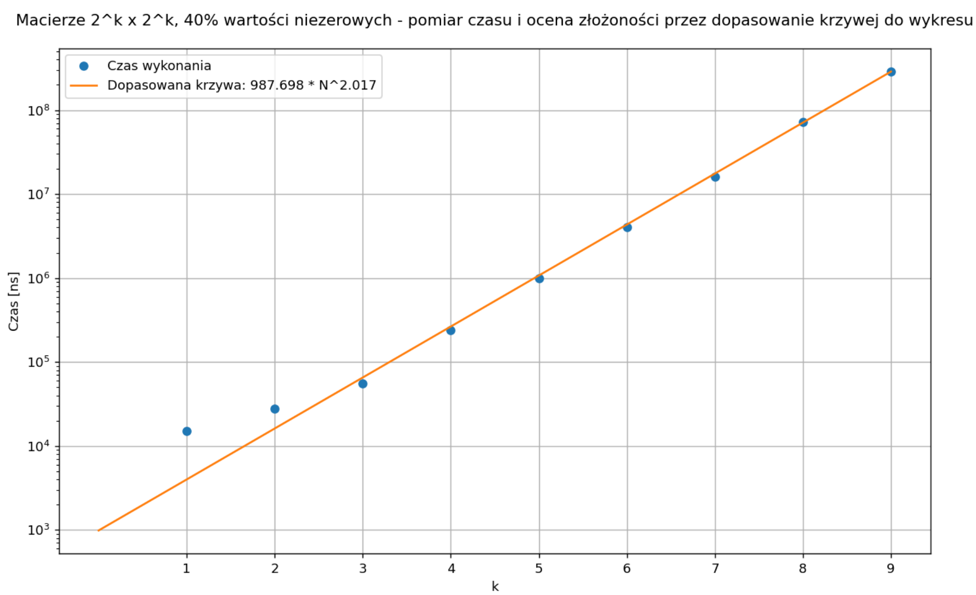
Rysunek 3: 10% wartości niezerowych



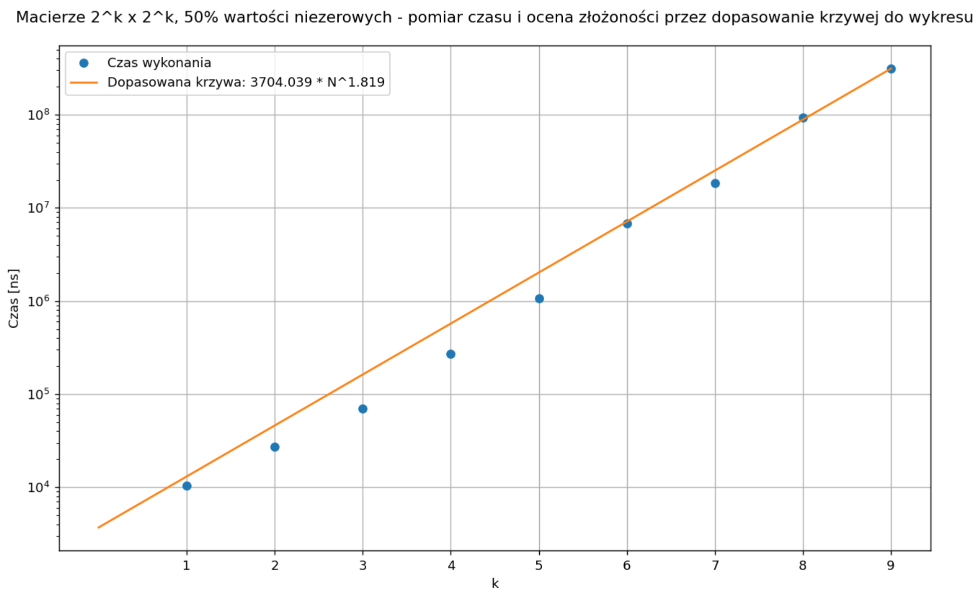
Rysunek 4: 20% wartości niezerowych



Rysunek 5: 30% wartości niezerowych



Rysunek 6: 40% wartości niezerowych

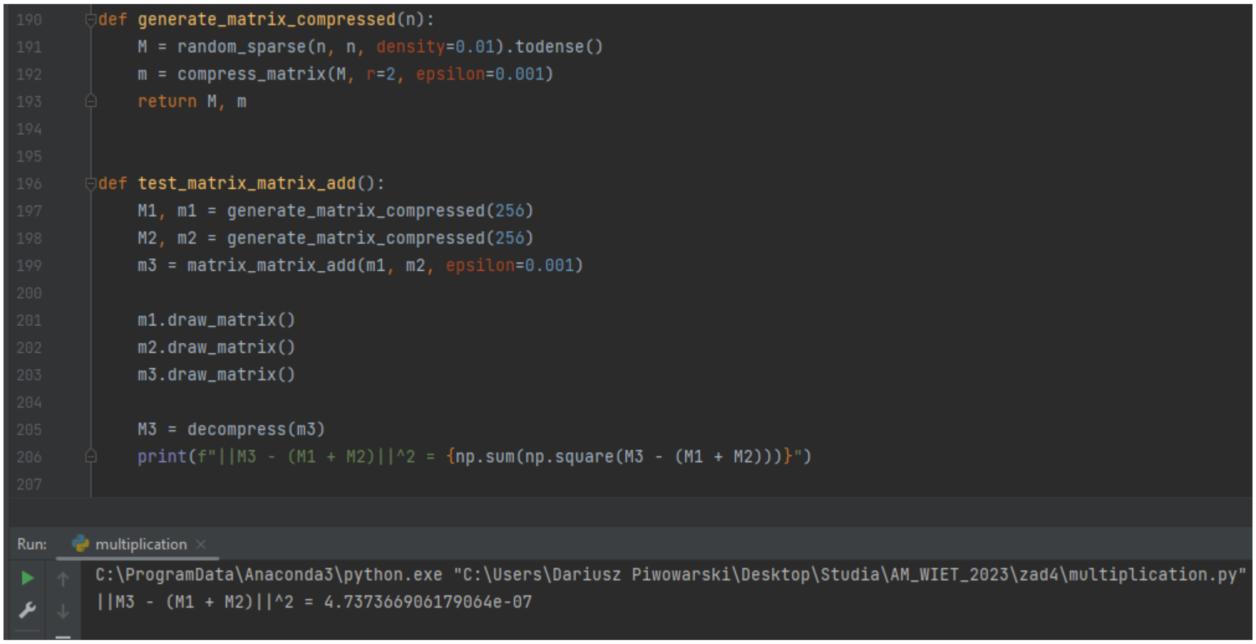


Rysunek 7: 50% wartości niezerowych

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

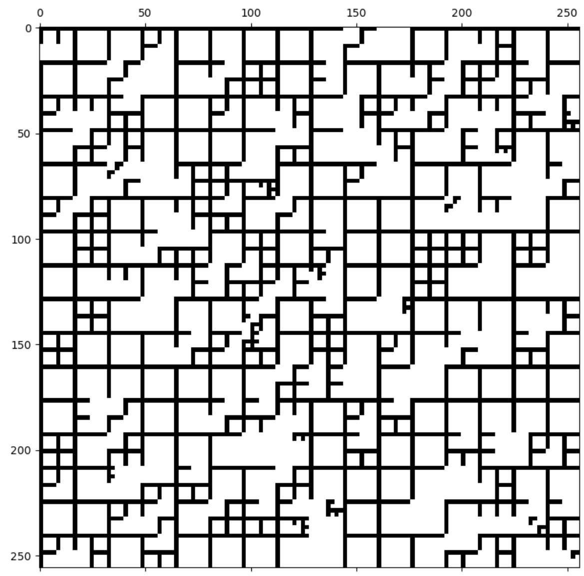
4 Testy implementacji algorytmu dodawania dwóch macierzy

4.1 Sprawdzenie poprawności wyniku dla macierzy 256x256

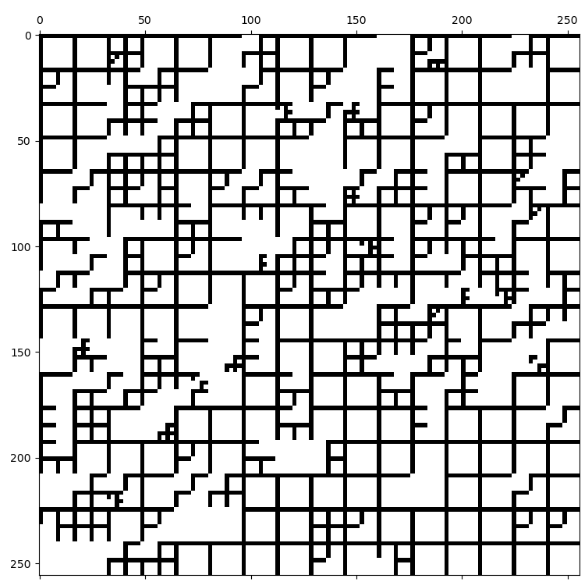


Rysunek 8: Suma różnicy kwadratów

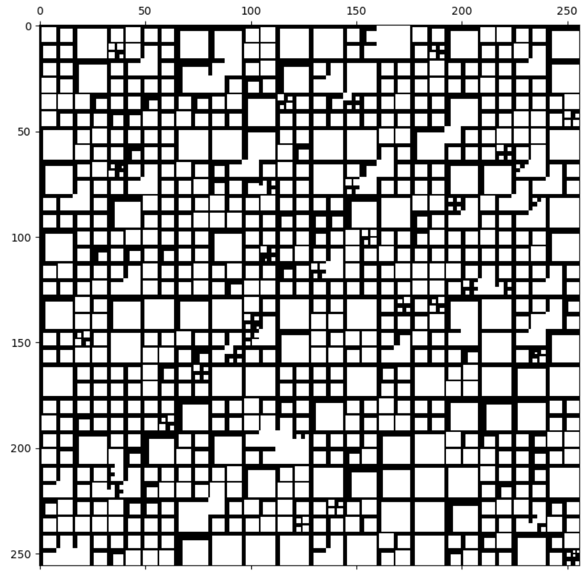
Sprawdzamy poprawność dodając do siebie 2 skompresowane macierze rzadkie (1% wartości niezerowych) o rozmiarze 256x256, a następnie dekompresujemy otrzymaną macierz. Wynik porównujemy z standardowym dodaniem macierzy (w naszym przypadku wykonywane jest to za pomocą biblioteki NumPy w Pythonie). Jak widać,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



Rysunek 9: Macierz 1 po kompresji



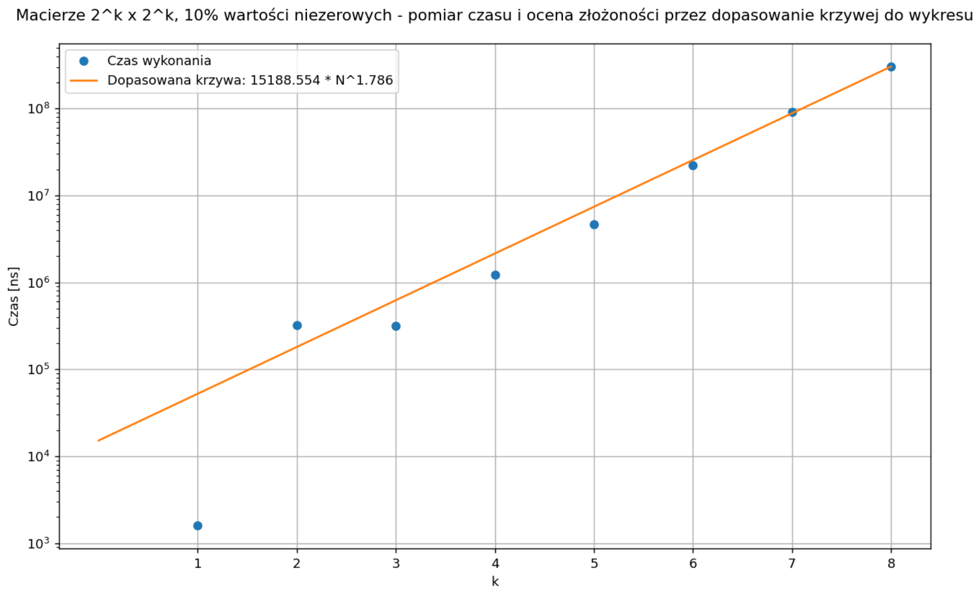
Rysunek 10: Macierz 2 po kompresji



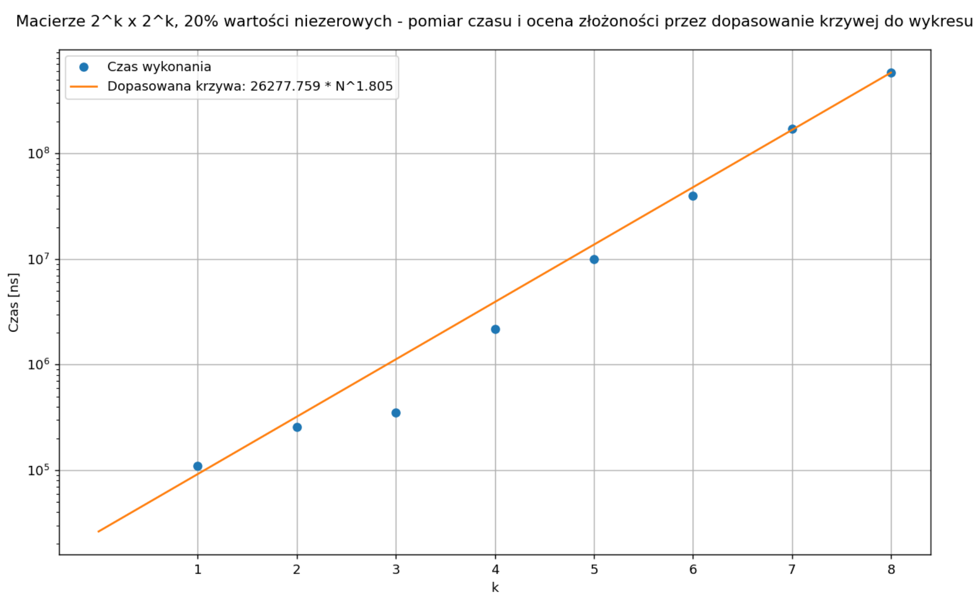
Rysunek 11: Suma macierzy

4.2 Pomiar czasu wykonania

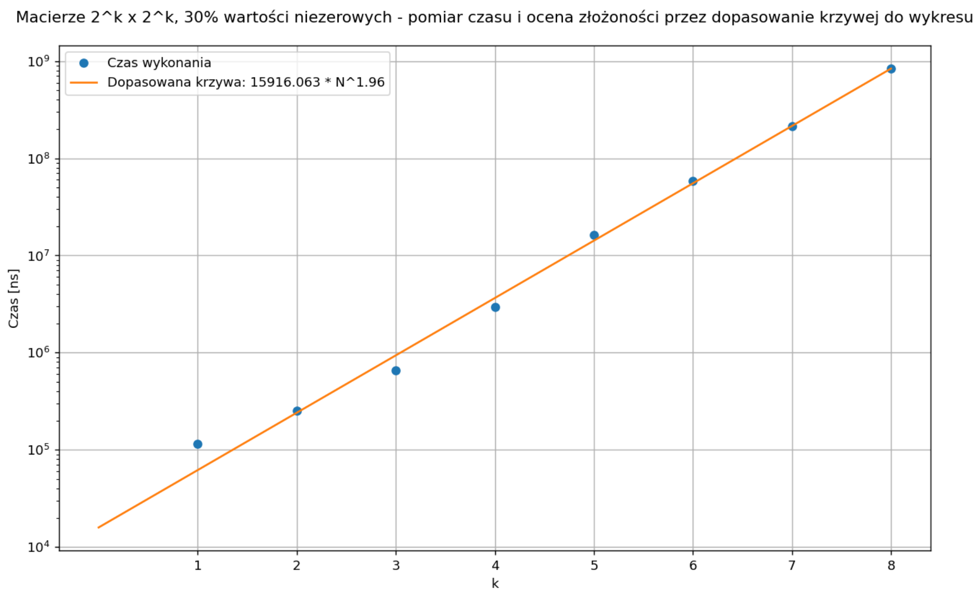
Oś Y na wykresach jest rysowana w skali logarytmicznej.



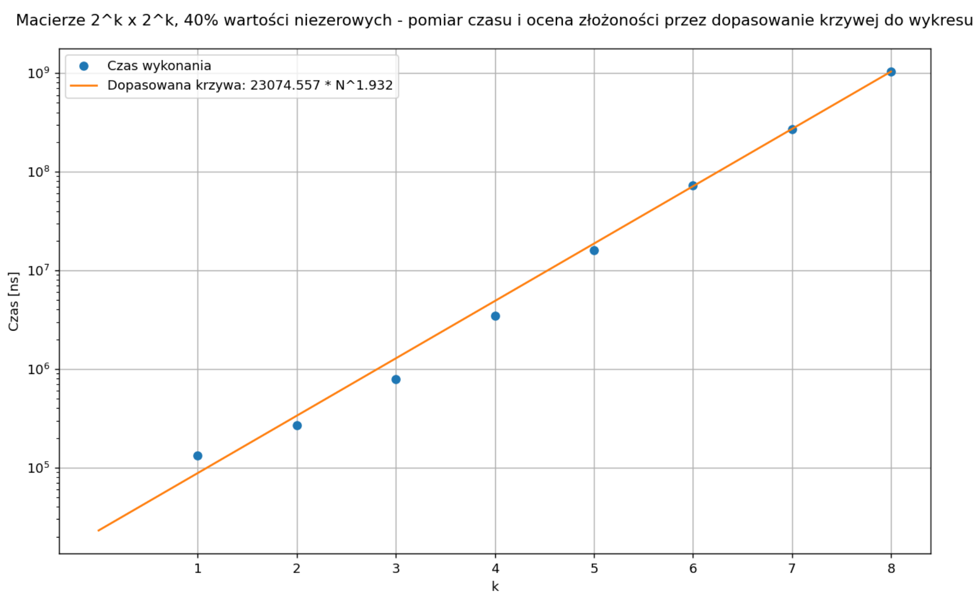
Rysunek 12: 10% wartości niezerowych



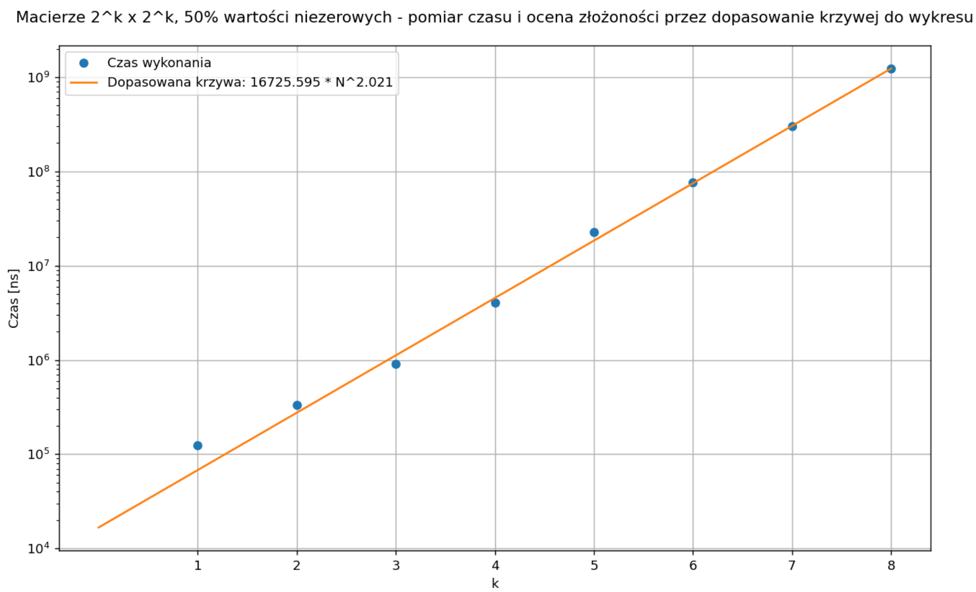
Rysunek 13: 20% wartości niezerowych



Rysunek 14: 30% wartości niezerowych



Rysunek 15: 40% wartości niezerowych



Rysunek 16: 50% wartości niezerowych

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

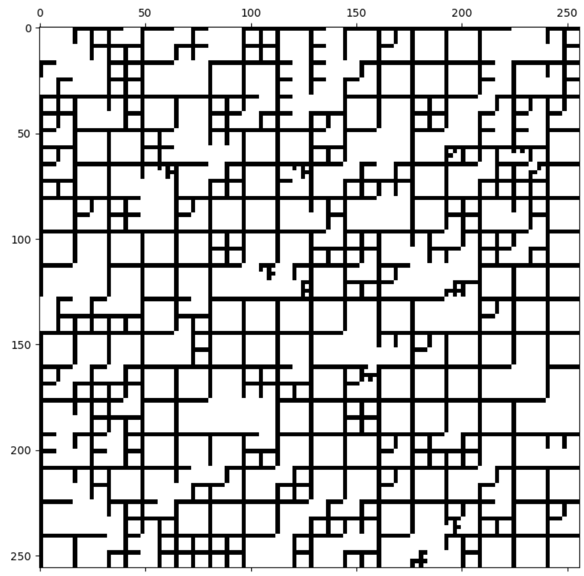
5 Testy implementacji algorytmu dodawania mnożenia macierzy

5.1 Sprawdzenie poprawności wyniku dla macierzy 256x256

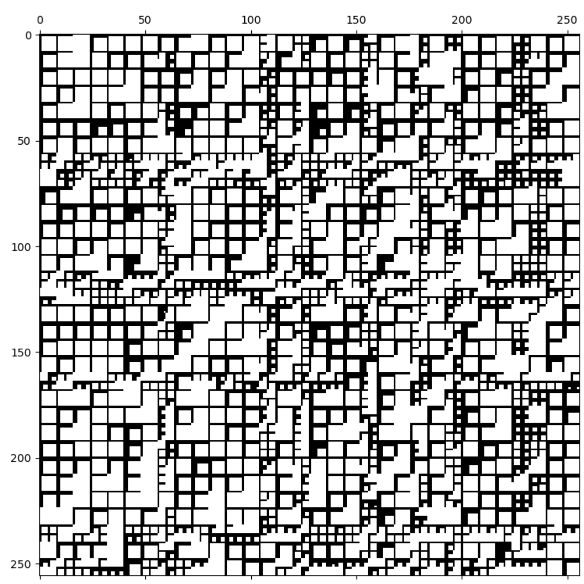
```
def test_matrix_matrix_mult():
    M = random_sparse(256, 256, density=0.01).todense()
    m1 = compress_matrix(M, r=2, epsilon=0.001)
    m2 = compress_matrix(M, r=2, epsilon=0.001)
    m3 = matrix_matrix_mult(m1, m2, epsilon=0.001)

    m1.draw_matrix()
    m3.draw_matrix()

    M3 = decompress(m3)
    print(f"||M3 - (M1 @ M1)||^2 = {np.sum(np.square(M3 - (M @ M)))}")

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.



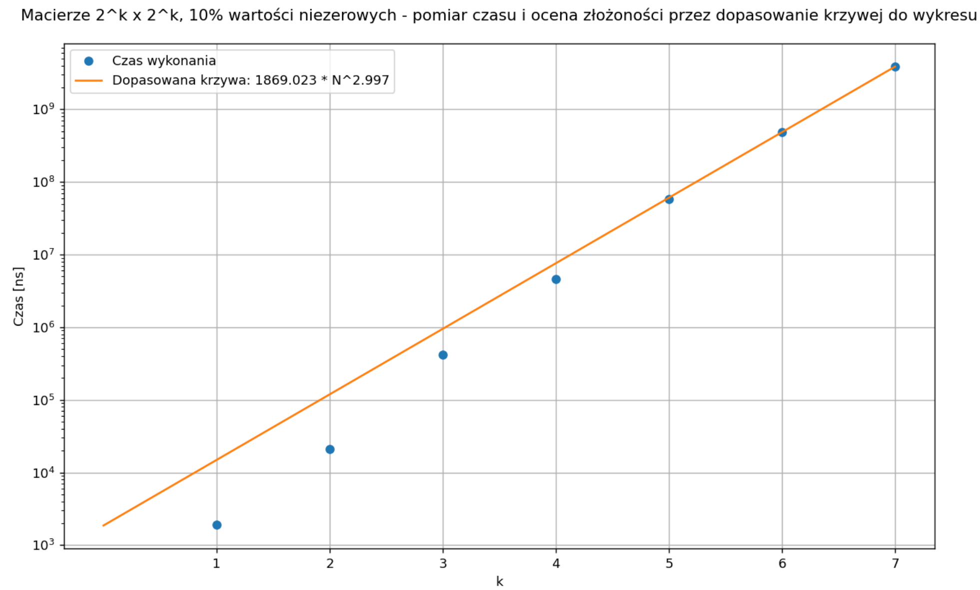
Rysunek 18: Macierz po kompresji



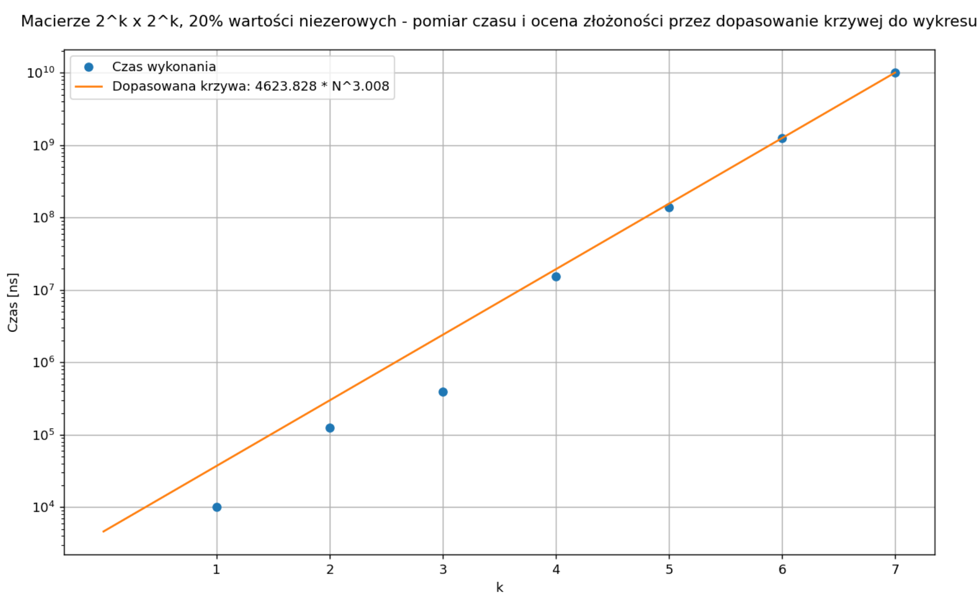
Rysunek 19: Macierz przemnożona przez samą siebie

5.2 Pomiar czasu wykonania

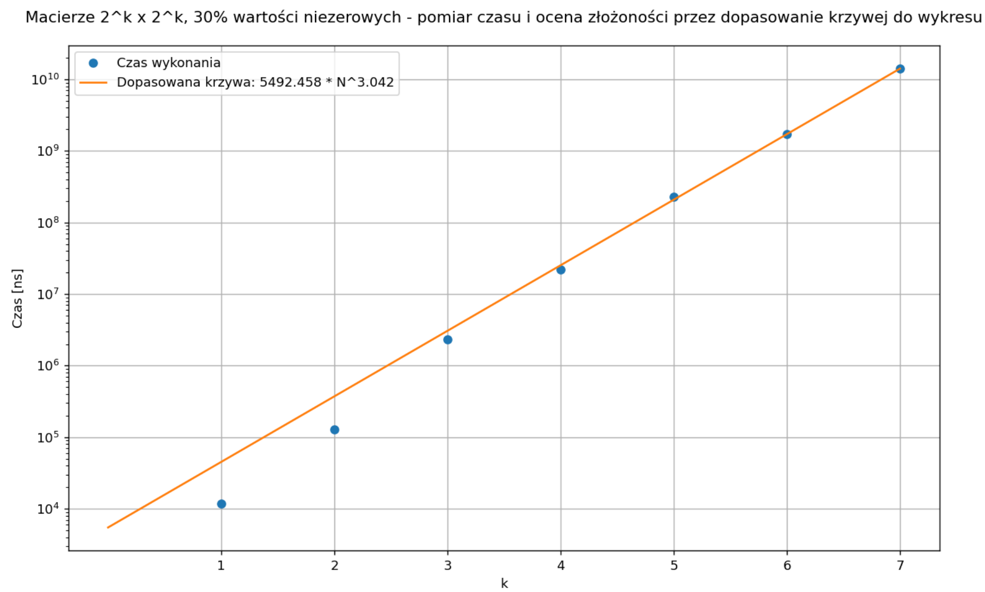
Oś Y na wykresach jest rysowana w skali logarytmicznej.



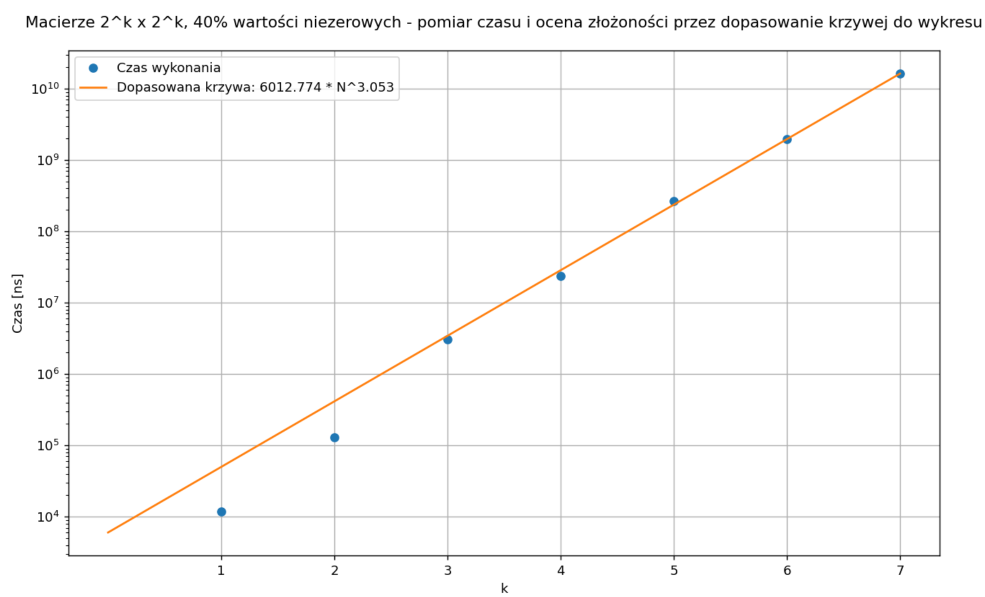
Rysunek 20: 10% wartości niezerowych



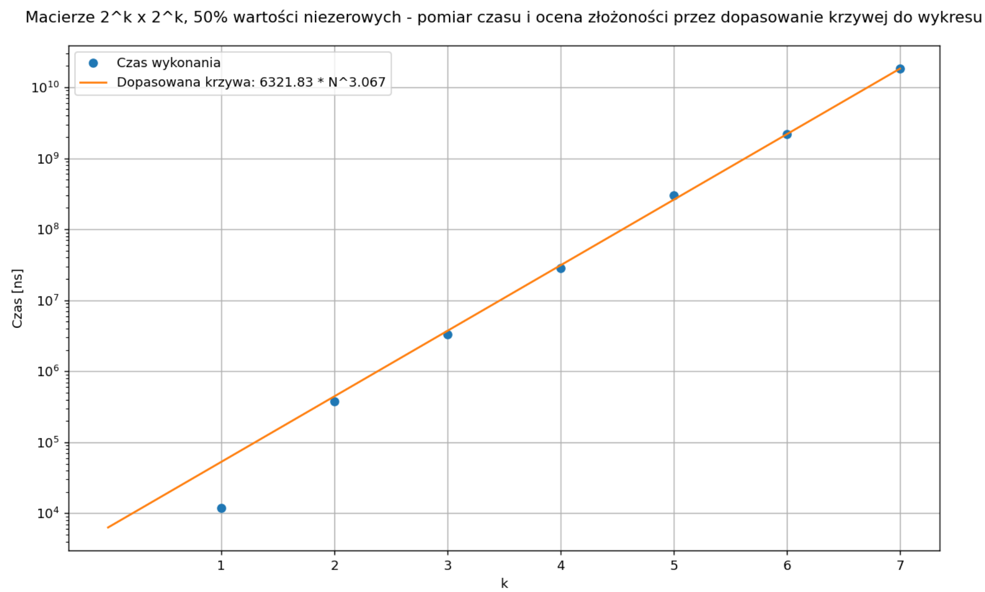
Rysunek 21: 20% wartości niezerowych



Rysunek 22: 30% wartości niezerowych



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.