Algorytmy Macierzowe

Sprawozdanie V

Grupa wtorek 13:00b

Michał Kuszewski i Michał Nożkiewicz

16 stycznia 2024

1 Opis zadania i użyte narzędzia

Naszym zadaniem było zaimplementowanie algorytmów operujących na macierzach hierarchicznych i przetestowanie ich na macierzach opisujących topologię siatki trójwymiarowej.

1. Mnożenie macierzy przez wektor
2. Mnożenie macierzy przez inną macierz

Do realizacji zadania użyliśmy języka Python. Korzystaliśmy z bibliotek numpy, matplotlib i scipy.

2 Pseudokody algorytmów

2.1 Mnożenie H-macierzy przez wektor

Algorytm 1: Multiply_by_vector

Data: Matrix H in hierarchical form, vector v
Result: $w = H \cdot v$

if *H is a leaf* **then**
  if *$H.rank = 0$* **then**
    return $zeros(v.length())$
  else
    return $H.U \cdot (H.V \cdot v)$
else
  $l := \frac{v.length()}{2}$
  $left := v[:l]$
  $right := v[l:]$
  $Y_1^{(1)} = Multiply\_by\_vector(H.sons(1), left)$
  $Y_1^{(2)} = Multiply\_by\_vector(H.sons(2), right)$
  $Y_2^{(1)} = Multiply\_by\_vector(H.sons(3), left)$
  $Y_2^{(2)} = Multiply\_by\_vector(H.sons(4), right)$
  return $\begin{bmatrix} Y_1^{(1)} + Y_1^{(2)} \\ Y_2^{(1)} + Y_2^{(2)} \end{bmatrix}$

2.2 Dodawanie macierzy hierarchicznych

Jako, że zaimplementowanie takiej funkcji było wymagane do implementacji mnożenia, postanowiliśmy zamieścić jej pseudokod.

Algorytm 2: Add

Data: Matrices A and B in hierarchical form
Result: $C = A + B$, C in hierarchical form

if *A is a leaf and B is a leaf* **then**
  if *$A.rank = 0$ or $B.rank = 0$* **then**
    return $Leaf(rank = 0)$
  else if *$A.rank$!= 0 and $B.rank = 0$* **then**
    return $copy(A)$
  else if *$A.rank = 0$ and $B.rank$!= 0* **then**
    return $copy(B)$
  else
    return $compress([A.U, B.U], [A.V, B.V])$
else if *A is a leaf and B is not a leaf* **then**
  $l1 := \frac{A.U.shape[0]}{2}$
  $U1 := A.U[: l1, :]$
  $U2 := A.U[l1 :, :]$
  $l2 := \frac{A.V.shape[1]}{2}$
  $V1 := A.V[:, : l2]$
  $V2 := A.V[:, l2 :]$
  return Node($\begin{bmatrix} Add(Leaf(U1, V1), B.sons(1)) \\ Add(Leaf(U1, V2), B.sons(2)) \\ Add(Leaf(U2, V1), B.sons(3)) \\ Add(Leaf(U2, V2), B.sons(4)) \end{bmatrix}$)
else if *A is not a leaf and B is a leaf* **then**
  Symetrically as in previous case
else
  return Node($\begin{bmatrix} Add(A.sons(1), B.sons(1)) \\ Add(A.sons(2), B.sons(2)) \\ Add(A.sons(3), B.sons(3)) \\ Add(A.sons(4), B.sons(4)) \end{bmatrix}$)

2.3 Mnożenie macierzy hierarchicznych

Algorytm 3: Mul

Data: Matrices A and B in hierarchical form
Result: $C = A \cdot B$, C in hierarchical form

if *A is a leaf and B is a leaf* **then**
  if *$A.rank = 0$ or $B.rank = 0$* **then**
    return $Leaf(rank = 0)$
  else
    return $Leaf(U = A.U \cdot (A.V \cdot B.V), V = B.V)$
else if *A is a leaf and B is not a leaf* **then**
  $l1 := \frac{A.U.shape[0]}{2}$
  $U1 := A.U[: l1, :]$
  $U2 := A.U[l1 :, :]$
  $l2 := \frac{A.V.shape[1]}{2}$
  $V1 := A.V[:, : l2]$
  $V2 := A.V[:, l2 :]$
  return Node($\begin{bmatrix} Add(Mul(Leaf(U1, V1), B.sons(1)), Mul(Leaf(U1, V2), B.sons(3))) \\ Add(Mul(Leaf(U1, V1), B.sons(2)), Mul(Leaf(U1, V2), B.sons(4))) \\ Add(Mul(Leaf(U2, V1), B.sons(1)), Mul(Leaf(U2, V2), B.sons(3))) \\ Add(Mul(Leaf(U2, V1), B.sons(2)), Mul(Leaf(U2, V2), B.sons(4))) \end{bmatrix}$)
else if *A is not a leaf and B is a leaf* **then**
  Symetrically as in previous case
else
  return Node($\begin{bmatrix} Add(Mul(A.sons(1), B.sons(1)), Mul(A.sons(2), B.sons(3))) \\ Add(Mul(A.sons(1), B.sons(2)), Mul(A.sons(2), B.sons(4))) \\ Add(Mul(A.sons(3), B.sons(1)), Mul(A.sons(4), B.sons(3))) \\ Add(Mul(A.sons(3), B.sons(2)), Mul(A.sons(4), B.sons(4))) \end{bmatrix}$)

3 Ważne fragmenty kodu

3.1 Mnożenie H-macierzy przez wektor

```
def multiply_by_vector(self, vector):
    half1, half2 = np.split(vector, 2)
    first = self.left_up.multiply_by_vector(half1) + self.right_up.multiply_by_vector(half2)
    second = self.left_low.multiply_by_vector(half1) + self.right_low.multiply_by_vector(half2)
    return np.vstack((first, second))
```

Rysunek 1: Mnożenie przez wektor dla węzła wewnętrznego

```
def multiply_by_vector(self, vector):
    if self.zeros:
        return np.zeros(vector.shape[0]).reshape(-1, 1)
    else:
        return self.U @ np.dot(self.V, vector).reshape(-1, 1)
```

Rysunek 2: Mnożenie przez wektor dla liścia

3.2 Dodawanie H-macierzy

```
def add(node1, node2, max_rank, length):
    if isinstance(node1, Leaf) and node1.zeros:
        return deepcopy(node2)
    elif isinstance(node2, Leaf) and node2.zeros:
        return deepcopy(node1)
    elif isinstance(node1, Leaf) and isinstance(node2, Leaf):
        if node1.U.shape[1] + node2.U.shape[1] <= max_rank:
            return Leaf(U=np.hstack((node1.U, node2.U)), V=np.vstack((node1.V, node2.V)))
        return compress(node1.U @ node1.V + node2.U @ node2.V, 0, max_rank, length)
    elif isinstance(node1, InternalNode) and isinstance(node2, InternalNode):
        return InternalNode(
            left_up=add(node1.left_up, node2.left_up, max_rank, length),
            right_up=add(node1.right_up, node2.right_up, max_rank, length),
            left_low=add(node1.left_low, node2.left_low, max_rank, length),
            right_low=add(node1.right_low, node2.right_low, max_rank, length)
        )
    else:
        length //= 2
        if isinstance(node1, InternalNode):
            node1, node2 = node2, node1

        U, V = node1.U, node1.V
        return InternalNode(
            left_up=add(Leaf(U=U[:length], V=V[:, :length]), node2.left_up, max_rank, length),
            right_up=add(Leaf(U=U[:length], V=V[:, length:]), node2.right_up, max_rank, length),
            left_low=add(Leaf(U=U[length:], V=V[:, :length]), node2.left_low, max_rank, length),
            right_low=add(Leaf(U=U[length:], V=V[:, length:]), node2.right_low, max_rank, length)
        )
```

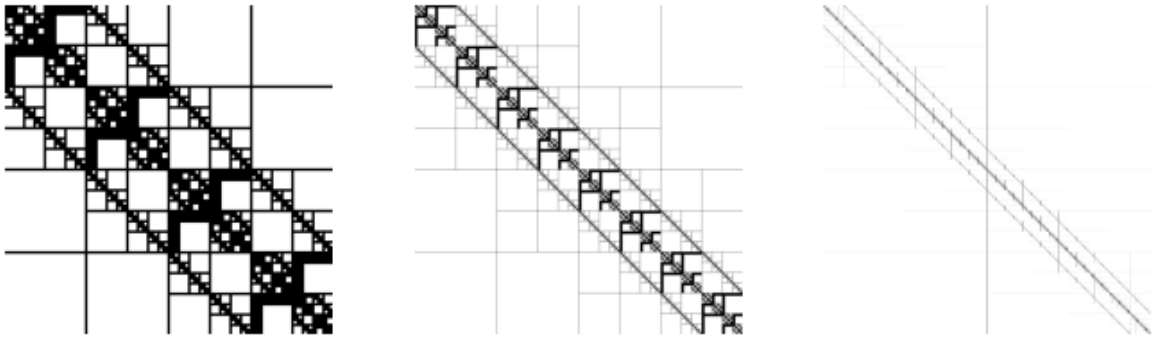
Rysunek 3: Dodawanie H-macierzy

3.3 Mnożenie H-macierzy

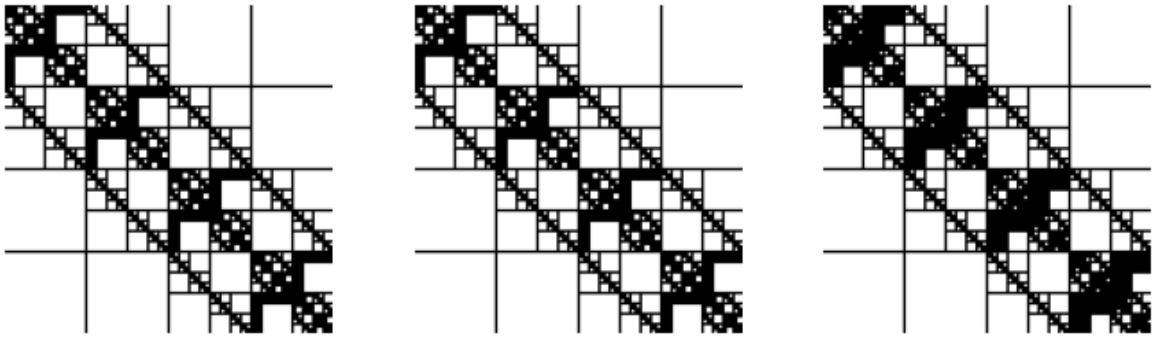
```
def mul(node1, node2, max_rank, length):
    if (isinstance(node1, Leaf) and node1.zeros) or (isinstance(node2, Leaf) and node2.zeros):
        return Leaf(zeros=True)
    elif isinstance(node1, Leaf) and isinstance(node2, Leaf):
        return Leaf(U=node1.U @ np.dot(node1.V, node2.U), V=node2.V)

    length //= 2
    if isinstance(node1, Leaf):
        U, V = node1.U, node1.V
        add_children(node1, U, V, length)

    elif isinstance(node2, Leaf):
        U, V = node2.U, node2.V
        add_children(node2, U, V, length)

    node = InternalNode(
        left_up=add(mul(node1.left_up, node2.left_up, max_rank, length),
                    mul(node1.right_up, node2.left_low, max_rank, length), max_rank, length),
        right_up=add(mul(node1.left_up, node2.right_up, max_rank, length),
                     mul(node1.right_up, node2.right_low, max_rank, length), max_rank, length),
        left_low=add(mul(node1.left_low, node2.left_up, max_rank, length),
                     mul(node1.right_low, node2.left_low, max_rank, length), max_rank, length),
        right_low=add(mul(node1.left_low, node2.right_up, max_rank, length),
                      mul(node1.right_low, node2.right_low, max_rank, length), max_rank, length)
    )
    if isinstance(node1, Leaf):
        remove_children(node1)

    elif isinstance(node2, Leaf):
        remove_children(node2)

    return node
```

Rysunek 4: Mnożenie H-macierzy

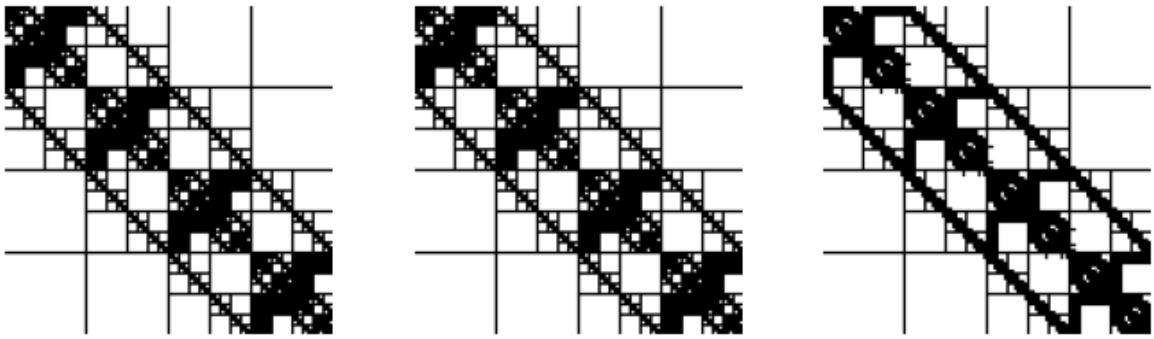
4 Wizualizacja działania algorytmów dodawania i mnożenia



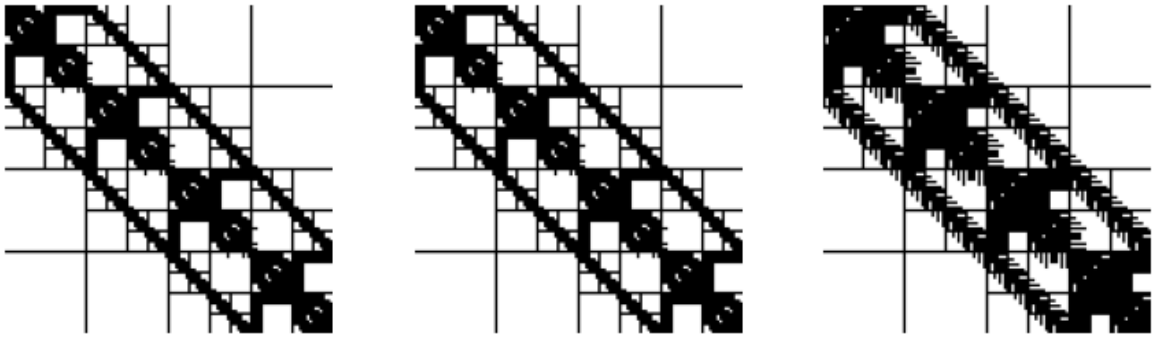
Rysunek 5: H-macierze reprezentujące siatkę 3d



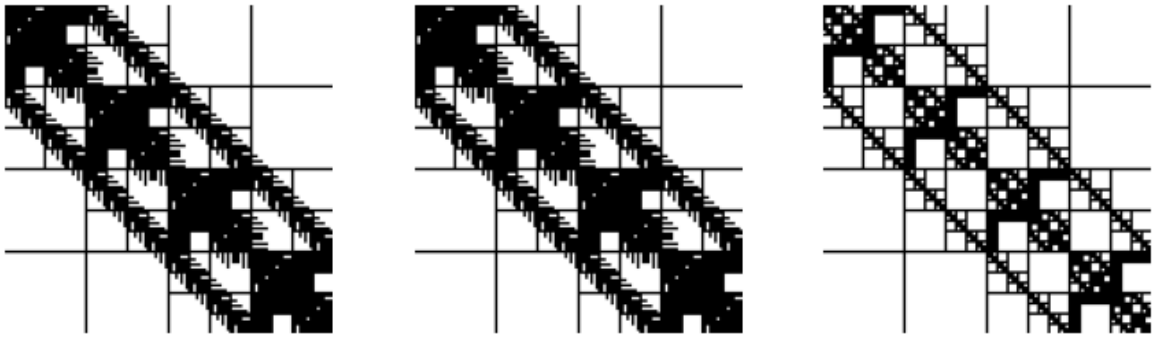
Rysunek 6: Suma tych samych macierzy



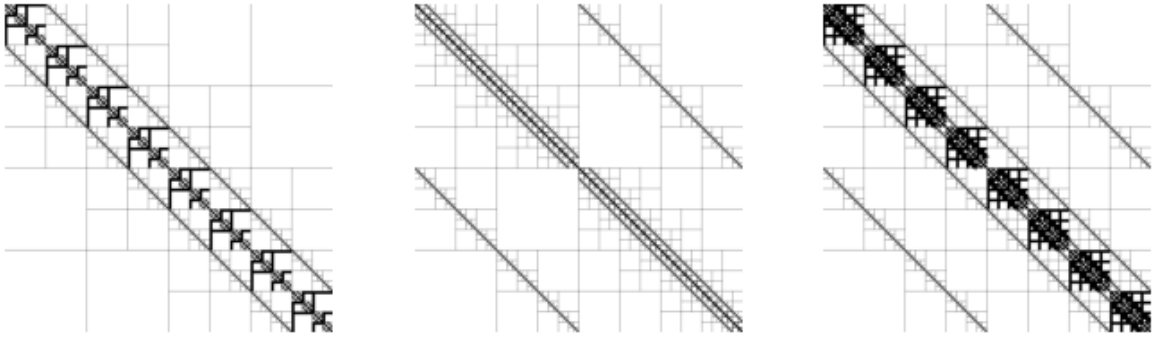
Rysunek 7: Suma tych samych macierzy



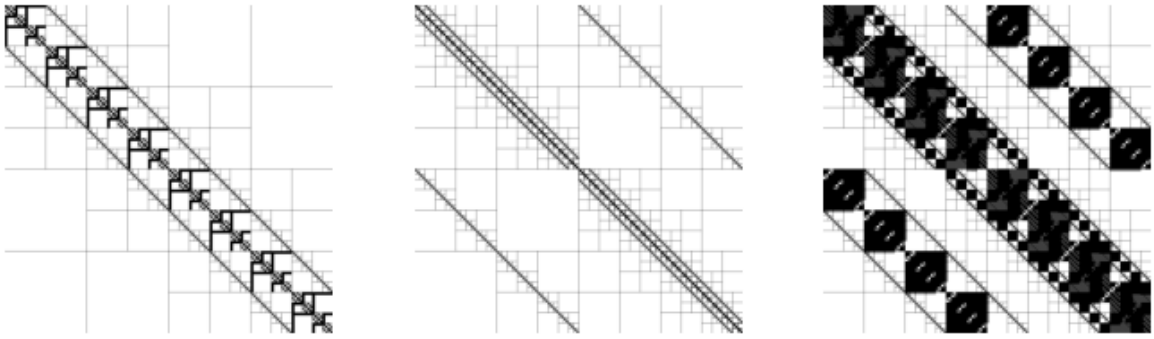
Rysunek 8: Suma tych samych macierzy



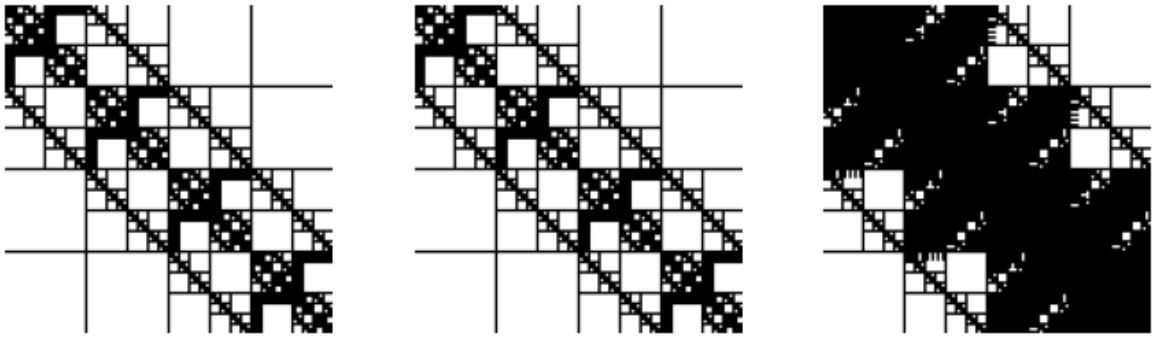
Rysunek 9: Suma tych samych macierzy



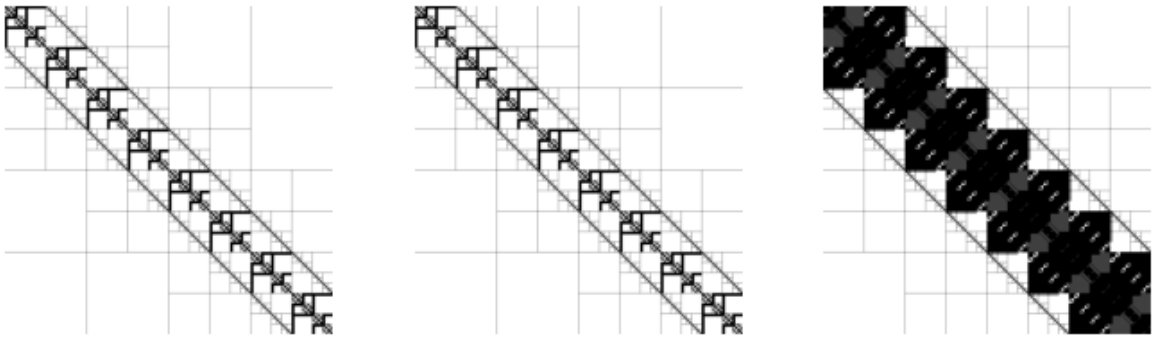
Rysunek 10: Suma różnych macierzy



Rysunek 11: Iloczyn macierzy



Rysunek 12: Iloczyn macierzy



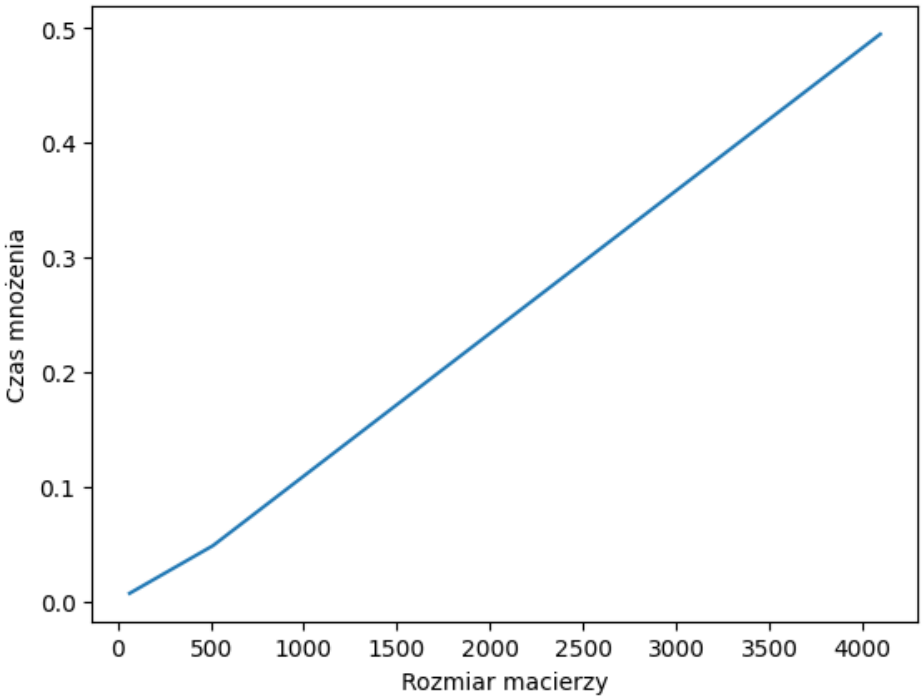
Rysunek 13: Iloczyn macierzy

5 Testy algorytmu mnożenia przez wektor

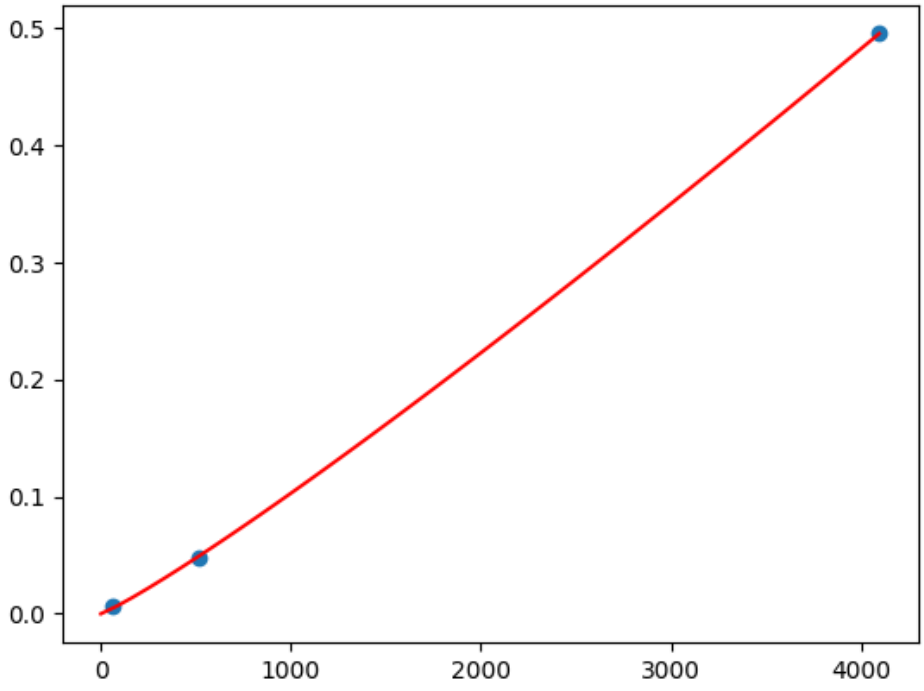
Dla każdej macierzy o rozmiarze $n \times n$ wylosowaliśmy gęsty wektor o długości n. Dla każdego mnożenia został zmierzony czas, a następnie wynik mnożenia przez macierz hierarchiczną porównaliśmy z wynikiem mnożenia tego wektora przez zwykłą macierz sprawdzając normę Frobeniusa z różnicy wynikowych wektorów. Następnie do pomiarów czasów dopasowaliśmy krzywą postaci $a \cdot n^b$.

Wymiar Macierzy	Czas mnożenia[s]	Odległość wektorów
64	0.006728	1.02305^{-30}
256	0.048369	1.18024^{-28}
4096	0.494660	1.50864^{-28}

Tabela 1: Tabela dla porównania stopnia kompresji



Rysunek 14: Wykres czasu mnożenia



Rysunek 15: Krzywa dopasowana do czasu mnożenia

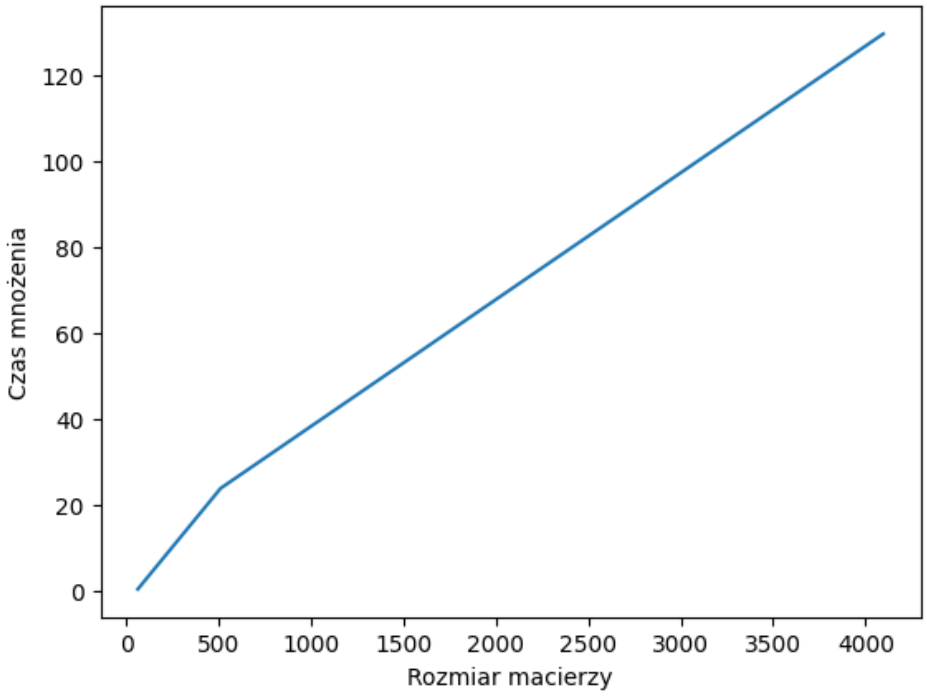
Parametry krzywej to $a = 4.66^{-5}$ oraz $b = 1.114$.

6 Testy algorytmu mnożenia macierzy

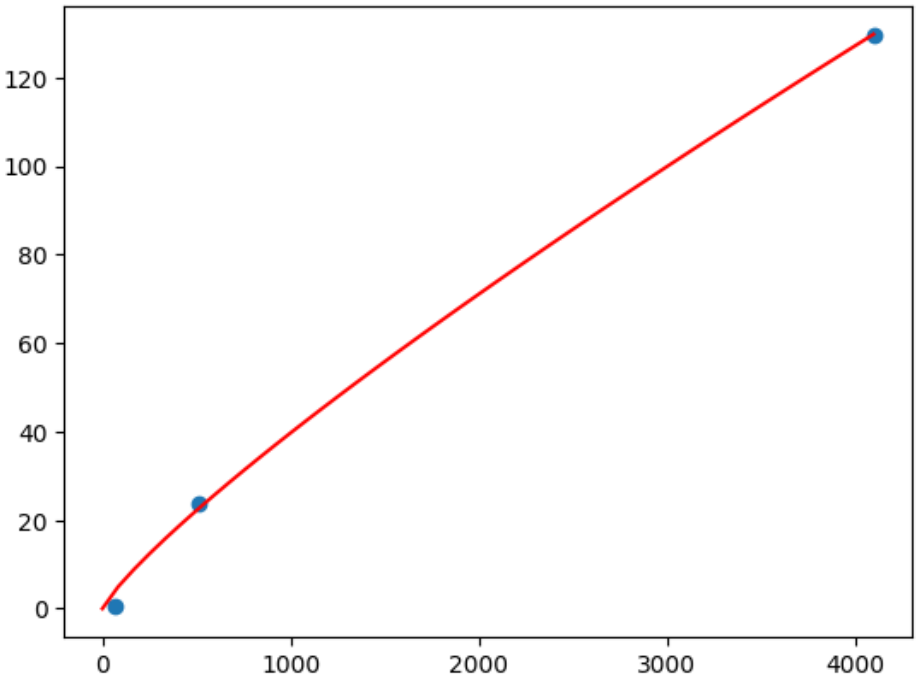
Analogiczne testy przeprowadziliśmy dla algorytmu mnożenia macierzy. W tym wypadku, aby porównać macierz wynikową ze zwykłą należało dokonać rekonstrukcji macierzy z reprezentacji drzewiastej do normalnej reprezentacji.

Wymiar Macierzy	Czas mnożenia[s]	Odległość macierzy
64	0.397846	3.250^{-30}
256	23.88238	1.0331^{-27}
4096	129.694831	2.16803^{-28}

Tabela 2: Tabela dla porównania stopnia kompresji



Rysunek 16: Wykres czasu mnożenia



Rysunek 17: Krzywa dopasowana do czasu mnożenia

Parametry krzywej to $a = 0.12019$ oraz $b = 0.83972$.

7 Wnioski

Odległości macierzy powstałych po zwykłym wymnożeniu macierzy a macierzy powstałej z mnożenia dwóch macierzy hierarchicznych były bardzo mało, więc można wnioskować, że algorytmy zostały zaimplementowane poprawnie. Ponadto pomiary czasowe są bardzo zadowalające. Dla macierzy rzadkich algorytmy miały złożoność podobną do liniowej, jednak dla większej pewności krzywą należałoby dopasować do większej ilości punktów pomiarów. W tym przypadku były to zaledwie trzy punkty.