Algorytmy macierzowe

Laboratorium 1
Sprawozdanie

Algorytmy rekurencyjnego mnożenia macierzy

Adam Naumiec

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki
Październik MMXXIII

Algorytmy macierzowe
Laboratorium 1

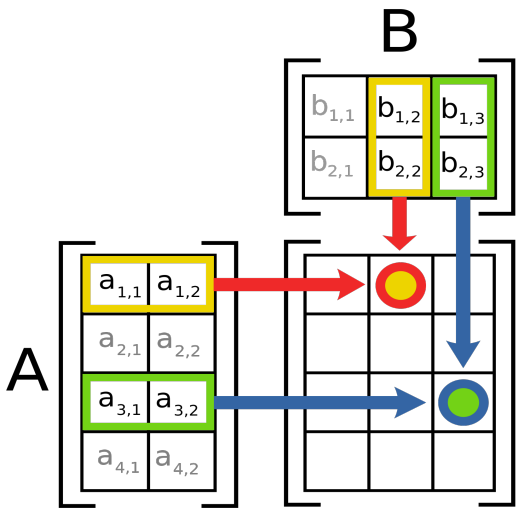
Adam Naumiec

Październik 2023

Spis treści

0 Abstrakt

Niniejszy dokument jest sprawozdaniem z wykonania Laboratorium 1 z przedmiotu Algorytmy macierzowe prowadzonego przez Pana prof. dr hab. Macieja Paszyńskiego w roku akademickim 2023/2024 na piątym semestrze studiów pierwszego stopnia na kierunku Informatyka prowadzonego na Akademii Górniczo-Hutniczej im. Stanisława Staszica w Krakowie na Wydziale Informatyki.



Rysunek 1: Mnożenie macierzy (2)

1 Streszczenie wykładu

Tematyką pierwszego wykładu i laboratorium było omówienie rekurencyjnych algorytmów mnożenia macierzy oraz ich złożoności obliczeniowej.

Zaprezentowane zostało także wykorzystanie i znaczenie tych algorytmów w praktyce oraz historia wykorzystania, postrzegania i zastosowania macierzy na przestrzeni wieków, a także matematyczne, inofmratyczne i algorytmiczne podstawy operacji na macierzach.

2 Zadania zrealizowane w ramach laboratorium

W ramach laboratorium zrealizowano następujące zadanie polegające na implementacji, wykonaniu testów wydajnościowych i przygotowaniu sprawozdania:

Proszę wybrać ulubiony język programownaia, wygenerować macierze losowe o wartościach z przedziału otwartego (0.00000001, 1.0) i zaimpelemntować wybrane algorytmy.

1. Rekurencyjne mnożenie macierzy metodą Binet'a (10 punktów)
2. Rekurencyjne mnożenie macierzy metodą Strassena (10 punktów)
3. Mnożenie macierzy metodą znalezioną przez sztuczną inteligencję (10 punktów)

Proszę zliczać liczbę operacji zmienno-przecinkowych (+-*/) podczas mnożenia macierzy.

3 Pseudokody algorytmów i fragmenty kodu

PRZYGOTOWANO pseudokody testowancyh algorytmów oraz przedstawiono wybrane najważniejsze fragmenty kodu zaimplementowanych algorytmów w języku programowania wysokiego poziomu.

3.1 Algorytm Binet'a

Załóżmy, że mamy dwie macierze A i B o wymiarach $n \times n$ i chcemy obliczyć ich iloczyn $C = A \cdot B$.

1. Podział macierzy na bloki

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Każda z macierzy A i B jest podzielona na mniejsze bloki A_{ij} i B_{ij}, gdzie $i, j = 1, 2, \ldots, n'$, a n' to rozmiar bloku.

2. Mnożenie bloków

   $C_{ij} = A_{ik} \cdot B_{kj}$ dla $i, j, k = 1, 2, \ldots, n'$

   Gdzie C_{ij} jest wynikowym blokiem macierzy C.

3. Łączenie wynikowych bloków

   Macierze C_{ij} są łączone w wynikową macierz C.

Implemetacja algorytmu w Pythonie:

```
def binet(A, B):
n = A.shape[0]
if n == 1:
    return A * B
else:
    A11, A12, A21, A22 = split_matrix(A)
    B11, B12, B21, B22 = split_matrix(B)

    C1 = binet(A11, B11) + binet(A12, B21)
    C2 = binet(A11, B12) + binet(A12, B22)
    C3 = binet(A21, B11) + binet(A22, B21)
    C4 = binet(A21, B12) + binet(A22, B22)

    return np.vstack((np.hstack((C1, C2)), np.hstack((C3, C4))))
```

Algorytm Binet'a to proces podziału macierzy na bloki, mnożenie bloków oraz ostateczne połączenie wynikowych bloków w wynikową macierz. To podejście pozwala na efektywne mnożenie dużych macierzy poprzez wykorzystanie operacji na mniejszych fragmentach danych.

3.2 Algorytm Strassena

1. Podział macierzy - Dzieli się macierze na mniejsze fragmenty.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22}. \end{bmatrix}$$

2. Równania rekurencyjne - Tworzy się pomocnicze macierze od M_1 do M_7 na podstawie odpowiednich operacji na mniejszych fragmentach

macierzy A i B:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}), \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11}, \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}), \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}), \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22}, \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}), \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}).
\end{aligned}
$$

3. Obliczenia rekurencyjne - Korzystając z przygotowanych macierzy pomocniczych oblicza sie wynikowe podmacierze C.

$$
C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},
$$

gdzie:

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7, \\
C_{12} &= M_3 + M_5, \\
C_{21} &= M_2 + M_4, \\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}
$$

Imlementacja algorytmu w Pythonie:

```
def strassen(A, B):
n = A.shape[0]
if n == 1:
    return A * B

A11, A12, A21, A22 = split_matrix(A)
B11, B12, B21, B22 = split_matrix(B)

M1 = strassen(A11 + A22, B11 + B22)
M2 = strassen(A21 + A22, B11)
M3 = strassen(A11, B12 - B22)
M4 = strassen(A22, B21 - B11)
M5 = strassen(A11 + A12, B22)
```

```
    M6 = strassen(A21 - A11, B11 + B12)
    M7 = strassen(A12 - A22, B21 + B22)

    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

    return C
```

3.3 Algorytm znaleziony przez sztuczną inteligencję

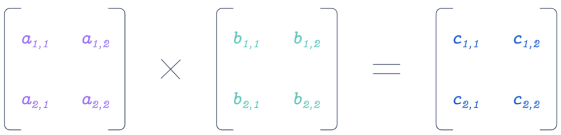
04 października 2022 ogłoszono, że sztuczna inteligencja odkryła szybszy algorytm rekurencyjnego mnożenia macierzy. Można zauważyć, że zasadniczo wzoruje się on na algorytmie Strassena, ale wprowadza szereg modyfikacji.

Cały artykuł dostępny jest tutaj: link.

DeepMind to firma zajmująca się badaniami nad sztuczną inteligencją, eksploruje różne metody wykorzystania AI do tworzenia nowych algorytmów, w tym w dziedzinie obliczeń tensorowych.

AlphaTensor, wspomniane w artykule, może być narzędziem, które wykorzystuje uczenie maszynowe do odkrywania efektywnych struktur danych i operacji matematycznych, które mogą poprawić wydajność obliczeniową, szczególnie w kontekście obliczeń tensorowych, które są wykorzystywane w głębokim uczeniu i sieciach neuronowych.

Materiały z artykułu:



Rysunek 2: Mnożenie macierzy - notacja jak przy algorytmie Binet'a i Strassena

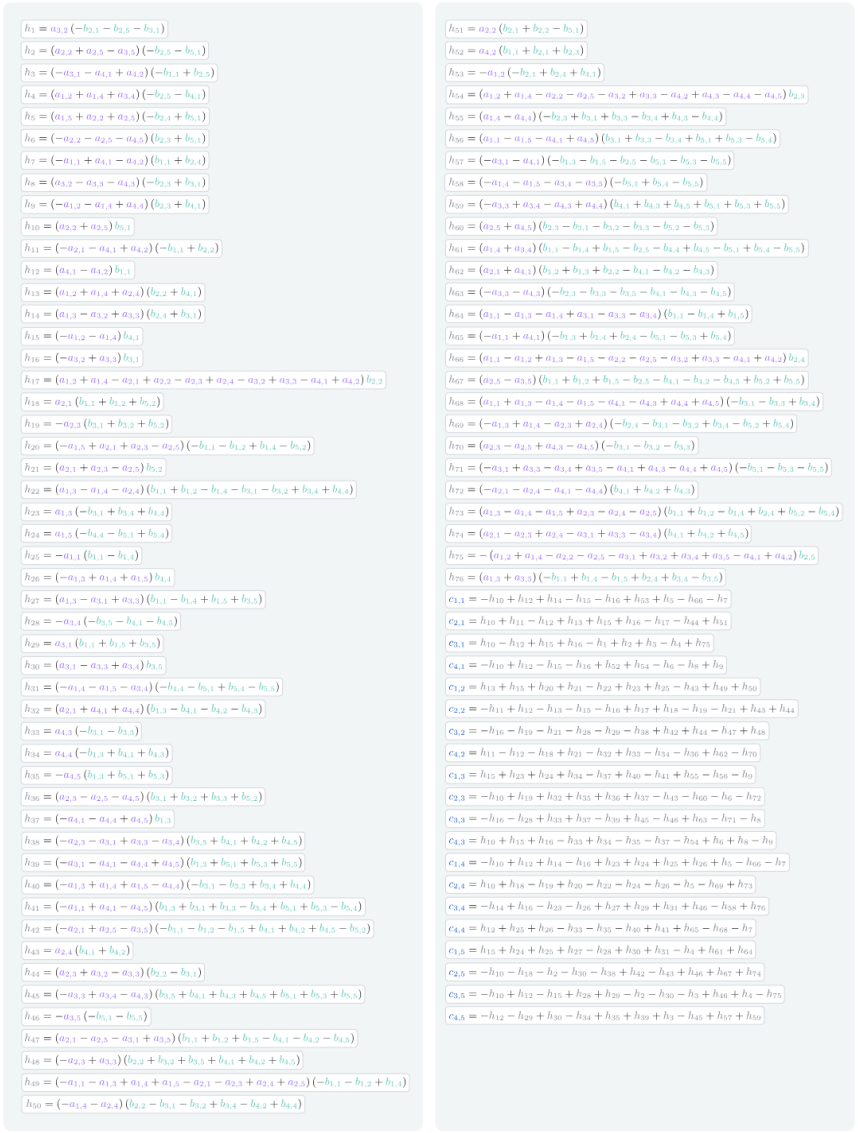
Rysunek 3: Algorytm Binet'a i Strassena

Na przykład dla macierzy 4x4 i 5x5 algorytm wykona tylko 76 mnożeń zamiast 80 jak w algorytmie Strassena.

Odkrycie algorytmu jest ważnym wydarzeniem w historii informatyki.

Implemetnacja algorytmu w Pythonie:

```
def ai(A, B):
m, n = A.shape
n, k = B.shape

if not (m % 4 == 0 and n % 5 == 0 and k % 5 == 0): return A @ B
```



Algorithm discovered by AlphaTensor using 76 multiplications, an improvement over state-of-the-art algorithms.

Rysunek 4: Algorytm znaleziony przez sztuczną inteligencję

```
height = m // 4
width = k // 5

A_prim = match_shape_matrix(A, 4, 5)
B_prim = match_shape_matrix(B, 5, 5)

h = [np.zeros() for _ in range(76)]

h[0] = ai(A_prim[2][1], -B_prim[1][0] - B_prim[1][4] - B_prim[2][0])
h[1] = ai(A_prim[1][1] + A_prim[1][4] - A_prim[2][4], -B_prim[1][4] - B_prim[4][0])
h[2] = ai(-A_prim[2][0] - A_prim[3][0] + A_prim[3][1], -B_prim[0][0] + B_prim[1][4])
h[3] = ai(A_prim[0][1] + A_prim[0][3] + A_prim[2][3], -B_prim[1][4] - B_prim[3][0])
h[4] = ai(A_prim[0][4] + A_prim[1][1] + A_prim[1][4], -B_prim[1][3] + B_prim[4][0])
h[5] = ai(-A_prim[1][1] - A_prim[1][4] - A_prim[3][4], B_prim[1][2] + B_prim[4][0])
h[6] = ai(-A_prim[0][0] + A_prim[3][0] - A_prim[3][1], B_prim[0][0] + B_prim[1][3])
h[7] = ai(A_prim[2][1] - A_prim[2][2] - A_prim[3][2], -B_prim[1][2] + B_prim[2][0])
h[8] = ai(-A_prim[0][1] - A_prim[0][3] + A_prim[3][3], B_prim[1][2] + B_prim[3][0])
h[9] = ai(A_prim[1][1] + A_prim[1][4], B_prim[4][0])
h[10] = ai(-A_prim[1][0] - A_prim[3][0] + A_prim[3][1], -B_prim[0][0] + B_prim[1][1])
h[11] = ai(A_prim[3][0] - A_prim[3][1], B_prim[0][0])
h[12] = ai(A_prim[0][1] + A_prim[0][3] + A_prim[1][3], B_prim[1][1] + B_prim[3][0])
h[13] = ai(A_prim[0][2] - A_prim[2][1] + A_prim[2][2], B_prim[1][3] + B_prim[2][0])
h[14] = ai(-A_prim[0][1] - A_prim[0][3], B_prim[3][0])
h[15] = ai(-A_prim[2][1] + A_prim[2][2], B_prim[2][0])
h[16] = ai(A_prim[0][1] + A_prim[0][3] - A_prim[1][0] + A_prim[1][1] - A_prim[1][2] + A_prim[1][3] - A_prim[2][1] + A_prim[2][2] - A_prim[3][0] + A_pri
        B_prim[1][1])
h[17] = ai(A_prim[1][0], B_prim[0][0] + B_prim[0][1] + B_prim[4][1])
h[18] = ai(-A_prim[1][2], B_prim[2][0] + B_prim[2][1] + B_prim[4][1])
h[19] = ai(-A_prim[0][4] + A_prim[1][0] + A_prim[1][2] - A_prim[1][4], -B_prim[0][0] - B_prim[0][1] + B_prim[0][3] - B_prim[4][1])
h[20] = ai(A_prim[1][0] + A_prim[1][2] - A_prim[1][4], B_prim[4][1])
h[21] = ai(A_prim[0][2] - A_prim[0][3] - A_prim[1][3], B_prim[0][0] + B_prim[0][1] - B_prim[0][3] - B_prim[2][0] - B_prim[2][1] + B_prim[2][3] + B_prim
h[22] = ai(A_prim[0][2], -B_prim[2][0] + B_prim[2][3] + B_prim[3][3])
h[23] = ai(A_prim[0][4], -B_prim[3][3] - B_prim[4][0] + B_prim[4][3])
h[24] = ai(-A_prim[0][0], B_prim[0][0] - B_prim[0][3])
h[25] = ai(-A_prim[0][2] + A_prim[0][3] + A_prim[0][4], B_prim[3][3])
h[26] = ai(A_prim[0][2] - A_prim[2][0] + A_prim[2][2], B_prim[0][0] - B_prim[0][3] + B_prim[0][4] + B_prim[2][4])
h[27] = ai(-A_prim[2][3], -B_prim[2][4] - B_prim[3][0] - B_prim[3][4])
h[28] = ai(A_prim[2][0], B_prim[0][0] + B_prim[0][4] + B_prim[2][4])
h[29] = ai(A_prim[2][0] - A_prim[2][2] + A_prim[2][3], B_prim[2][4])
h[30] = ai(-A_prim[0][3] - A_prim[0][4] - A_prim[2][3], -B_prim[3][3] - B_prim[4][0] + B_prim[4][3] - B_prim[4][4])
h[31] = ai(A_prim[1][0] + A_prim[3][0] + A_prim[3][3], B_prim[0][2] - B_prim[3][0] - B_prim[3][1] - B_prim[3][2])
h[32] = ai(A_prim[3][2], -B_prim[2][0] - B_prim[2][2])
h[33] = ai(A_prim[3][3], -B_prim[0][2] + B_prim[3][0] + B_prim[3][2])
h[34] = ai(-A_prim[3][4], B_prim[0][2] + B_prim[4][0] + B_prim[4][2])
h[35] = ai(A_prim[1][2] - A_prim[1][4] - A_prim[3][4], B_prim[2][0] + B_prim[2][1] + B_prim[2][2] + B_prim[4][1])
h[36] = ai(-A_prim[3][0] - A_prim[3][3] + A_prim[3][4], B_prim[0][2])
h[37] = ai(-A_prim[1][2] - A_prim[2][0] + A_prim[2][2] - A_prim[2][3], B_prim[2][4] + B_prim[3][0] + B_prim[3][1] + B_prim[3][4])
h[38] = ai(-A_prim[2][0] - A_prim[3][0] - A_prim[3][3] + A_prim[3][4], B_prim[0][2] + B_prim[4][0] + B_prim[4][2] + B_prim[4][4])
h[39] = ai(-A_prim[0][2] + A_prim[0][3] + A_prim[0][4] - A_prim[3][3], -B_prim[2][0] - B_prim[2][2] + B_prim[2][3] + B_prim[3][3])
h[40] = ai(-A_prim[0][0] + A_prim[3][0] - A_prim[3][4], B_prim[0][2] + B_prim[2][0] + B_prim[2][2] - B_prim[2][3] + B_prim[4][0] + B_prim[4][2] - B_pri
h[41] = ai(-A_prim[1][0] + A_prim[1][4] - A_prim[2][4], -B_prim[0][0] - B_prim[0][1] - B_prim[0][4] + B_prim[3][0] + B_prim[3][1] + B_prim[3][4] - B_pr
h[42] = ai(A_prim[1][3], B_prim[3][0] + B_prim[3][1])
h[43] = ai(A_prim[1][2] + A_prim[2][1] - A_prim[2][2], B_prim[1][1] - B_prim[2][0])
h[44] = ai(-A_prim[2][2] + A_prim[2][3] - A_prim[3][2], B_prim[2][4] + B_prim[3][0] + B_prim[3][2] + B_prim[3][4] + B_prim[4][0] + B_prim[4][2] + B_pri
h[45] = ai(-A_prim[2][4], -B_prim[4][0] - B_prim[4][4])
h[46] = ai(A_prim[1][0] - A_prim[1][4] - A_prim[2][0] + A_prim[2][4], B_prim[0][0] + B_prim[0][1] + B_prim[0][4] - B_prim[3][0] - B_prim[3][1] - B_prim
h[47] = ai(-A_prim[1][2] + A_prim[2][2], B_prim[1][1] + B_prim[2][1] + B_prim[2][4] + B_prim[3][0] + B_prim[3][1] + B_prim[3][4])
h[48] = ai(-A_prim[0][0] - A_prim[0][2] + A_prim[0][3] + A_prim[0][4] - A_prim[1][0] - A_prim[1][2] + A_prim[1][3] + A_prim[1][4],
        -B_prim[0][0] - B_prim[0][1] + B_prim[0][3])
h[49] = ai(-A_prim[0][3] - A_prim[1][3], B_prim[1][1] - B_prim[2][0] - B_prim[2][1] + B_prim[2][3] - B_prim[3][1] + B_prim[3][3])
h[50] = ai(A_prim[1][1], B_prim[1][0] + B_prim[1][1] - B_prim[4][0])
h[51] = ai(A_prim[3][1], B_prim[0][0] + B_prim[1][0] + B_prim[1][2])
h[52] = ai(-A_prim[0][1], -B_prim[1][0] + B_prim[1][3] + B_prim[3][0])
h[53] = ai(A_prim[0][1] + A_prim[0][3] - A_prim[1][1] - A_prim[1][4] - A_prim[2][1] + A_prim[2][2] - A_prim[3][1] + A_prim[3][2] - A_prim[3][3] - A_pri
        B_prim[1][2])
h[54] = ai(A_prim[0][3] - A_prim[3][3], -B_prim[1][2] + B_prim[2][0] + B_prim[2][2] - B_prim[2][3] + B_prim[3][2] - B_prim[3][3])
h[55] = ai(A_prim[0][0] - A_prim[0][4] - A_prim[3][0] + A_prim[3][4], B_prim[2][0] + B_prim[2][2] - B_prim[2][3] + B_prim[4][0] + B_prim[4][2] - B_prim
h[56] = ai(-A_prim[2][0] - A_prim[3][0], -B_prim[0][2] - B_prim[0][4] - B_prim[1][4] - B_prim[4][0] - B_prim[4][2] - B_prim[4][4])
h[57] = ai(-A_prim[0][3] - A_prim[0][4] - A_prim[2][3] - A_prim[2][4], -B_prim[4][0] + B_prim[4][3] - B_prim[4][4])
h[58] = ai(-A_prim[2][2] + A_prim[2][3] - A_prim[3][2] + A_prim[3][3], B_prim[3][0] + B_prim[3][2] + B_prim[3][4] + B_prim[4][0] + B_prim[4][2] + B_pri
h[59] = ai(A_prim[1][4] + A_prim[3][4], B_prim[1][2] - B_prim[2][0] - B_prim[2][1] - B_prim[2][2] - B_prim[4][1] - B_prim[4][2])
h[60] = ai(A_prim[0][3] + A_prim[2][3],
        B_prim[0][0] - B_prim[0][3] + B_prim[0][4] - B_prim[1][4] - B_prim[3][3] + B_prim[3][4] - B_prim[4][0] + B_prim[4][3] - B_prim[4][4])
h[61] = ai(A_prim[1][0] + A_prim[3][0], B_prim[0][1] + B_prim[0][2] + B_prim[1][1] - B_prim[3][0] - B_prim[3][1] - B_prim[3][2])
h[62] = ai(-A_prim[2][2] - A_prim[3][2], -B_prim[1][2] - B_prim[2][2] - B_prim[2][4] - B_prim[3][0] - B_prim[3][2] - B_prim[3][4])
h[63] = ai(A_prim[0][0] - A_prim[0][2] - A_prim[0][3] + A_prim[2][0] - A_prim[2][2] - A_prim[2][3], B_prim[0][0] - B_prim[0][3] + B_prim[0][4])
h[64] = ai(-A_prim[0][0] + A_prim[3][0], -B_prim[0][2] + B_prim[0][3] + B_prim[1][3] - B_prim[4][0] - B_prim[4][2] + B_prim[4][3])
h[65] = ai(A_prim[0][0] - A_prim[0][1] + A_prim[0][2] - A_prim[0][4] - A_prim[1][1] - A_prim[1][4] - A_prim[2][1] + A_prim[2][2] - A_prim[3][0] + A_pri
        B_prim[1][3])
h[66] = ai(A_prim[1][4] - A_prim[2][4],
        B_prim[0][0] + B_prim[0][1] + B_prim[0][4] - B_prim[1][4] - B_prim[3][0] - B_prim[3][1] - B_prim[3][4] + B_prim[4][1] + B_prim[4][4])
h[67] = ai(A_prim[0][0] + A_prim[0][2] - A_prim[0][3] - A_prim[0][4] - A_prim[3][0] - A_prim[3][2] + A_prim[3][3] + A_prim[3][4],
```

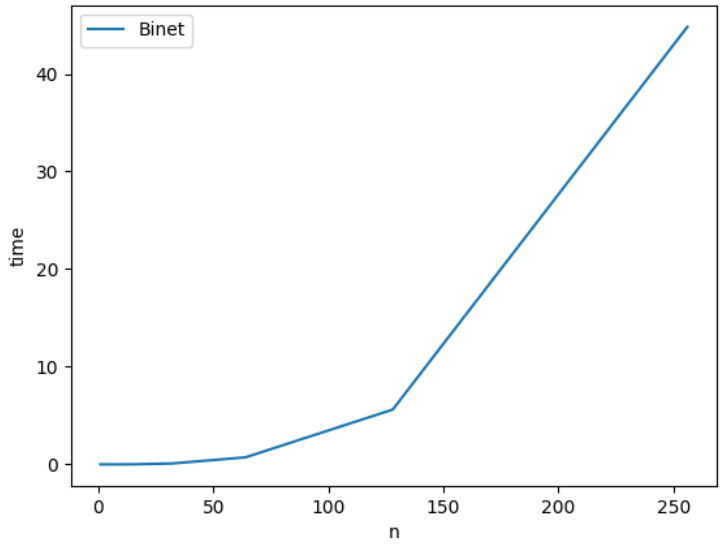
```
          -B_prim[2][0] - B_prim[2][2] + B_prim[2][3])
h[68] = ai(-A_prim[0][2] + A_prim[0][3] - A_prim[1][2] + A_prim[1][3], -B_prim[1][3] - B_prim[2][0] - B_prim[2][1] + B_prim[2][3] - B_prim[4][1] + B_pr
h[69] = ai(A_prim[1][2] - A_prim[1][4] + A_prim[3][2] - A_prim[3][4], -B_prim[2][0] - B_prim[2][1] - B_prim[2][2])
h[70] = ai(-A_prim[2][0] + A_prim[2][2] - A_prim[2][3] + A_prim[2][4] - A_prim[3][0] + A_prim[3][2] - A_prim[3][3] + A_prim[3][4],
          -B_prim[4][0] - B_prim[4][2] - B_prim[4][4])
h[71] = ai(-A_prim[1][0] - A_prim[1][3] - A_prim[3][0] - A_prim[3][3], B_prim[3][0] + B_prim[3][1] + B_prim[3][2])
h[72] = ai(A_prim[0][2] - A_prim[0][3] - A_prim[0][4] + A_prim[1][2] - A_prim[1][3] - A_prim[1][4],
           B_prim[0][0] + B_prim[0][1] - B_prim[0][3] + B_prim[1][3] + B_prim[4][1] - B_prim[4][3])
h[73] = ai(A_prim[1][0] - A_prim[1][2] + A_prim[1][3] - A_prim[2][0] + A_prim[2][2] - A_prim[2][3], B_prim[3][0] + B_prim[3][1] + B_prim[3][4])
h[74] = ai(-A_prim[0][1] - A_prim[0][3] + A_prim[1][1] + A_prim[1][4] + A_prim[2][0] - A_prim[2][1] - A_prim[2][3] - A_prim[2][4] + A_prim[3][0] - A_pr
           B_prim[1][4])
h[75] = ai(A_prim[0][2] + A_prim[2][2], -B_prim[0][0] + B_prim[0][3] - B_prim[0][4] + B_prim[1][3] + B_prim[2][3] - B_prim[2][4])

C = [[np.zeros() for _ in range(5)] for _ in range(4)]

for i in range(76):
    count_plus += h[i][1]
    count_times += h[i][2]

    h[i] = h[i][0]

C[0][0] = -h[9] + h[11] + h[13] - h[14] - h[15] + h[52] + h[4] - h[65] - h[6]
C[1][0] = h[9] + h[10] - h[11] + h[12] + h[14] + h[15] - h[16] - h[43] + h[50]
C[2][0] = h[9] - h[11] + h[14] + h[15] - h[0] + h[1] + h[2] - h[3] + h[74]
C[3][0] = -h[9] + h[11] - h[14] - h[15] + h[51] + h[53] - h[5] - h[7] + h[8]
C[0][1] = h[12] + h[14] + h[19] + h[20] - h[21] + h[22] + h[24] - h[42] + h[48] + h[49]
C[1][1] = -h[10] + h[11] - h[12] - h[14] - h[15] + h[16] + h[17] - h[18] - h[20] + h[42] + h[43]
C[2][1] = -h[15] - h[18] - h[20] - h[27] - h[28] - h[37] + h[41] + h[43] - h[46] + h[47]
C[3][1] = h[10] - h[11] - h[17] + h[20] - h[31] + h[32] - h[33] - h[35] + h[61] - h[69]
C[0][2] = h[14] + h[22] + h[23] + h[33] - h[36] + h[39] - h[40] + h[54] - h[55] - h[8]
C[1][2] = -h[9] + h[18] + h[31] + h[34] + h[35] + h[36] - h[42] - h[59] - h[5] - h[71]
C[2][2] = -h[15] - h[27] + h[32] + h[36] - h[38] + h[44] - h[45] + h[62] - h[70] - h[7]
C[3][2] = h[9] + h[14] + h[15] - h[32] + h[33] - h[34] - h[36] - h[53] + h[5] + h[7] - h[8]
C[0][3] = -h[9] + h[11] + h[13] - h[15] + h[22] + h[23] + h[24] + h[25] + h[4] - h[65] - h[6]
C[1][3] = h[9] + h[17] - h[18] + h[19] - h[21] - h[23] - h[25] - h[4] - h[68] + h[72]
C[2][3] = -h[13] + h[15] - h[22] - h[25] + h[26] + h[28] + h[30] + h[45] - h[57] + h[75]
C[3][3] = h[11] + h[24] + h[25] - h[32] - h[34] - h[39] + h[40] + h[64] - h[67] - h[6]
C[0][4] = h[14] + h[23] + h[24] + h[26] - h[27] + h[29] + h[30] - h[3] + h[60] + h[63]
C[1][4] = -h[9] - h[17] - h[1] - h[29] - h[37] + h[41] - h[42] + h[45] + h[66] + h[73]
C[2][4] = -h[9] + h[11] - h[14] + h[27] + h[28] - h[1] - h[29] - h[2] + h[45] + h[3] - h[74]
C[3][4] = -h[11] - h[28] + h[29] - h[33] + h[34] + h[38] + h[2] - h[44] + h[56] + h[58]

return C
```

4 Testy wydajnościowe

PRZYGOTOWANO testy wydajnościowe omawainych algorytmów w celu prezentacji czasu obliczeń oraz liczby wykonanych operacji zmiennoprzecinkowych. Czynność ta pozwoliła na praktyczne przetestowanie algorytmów oraz konformantacji ich teoretycznych złozoności obliczeniowych wraz z danymi empirycznymi.

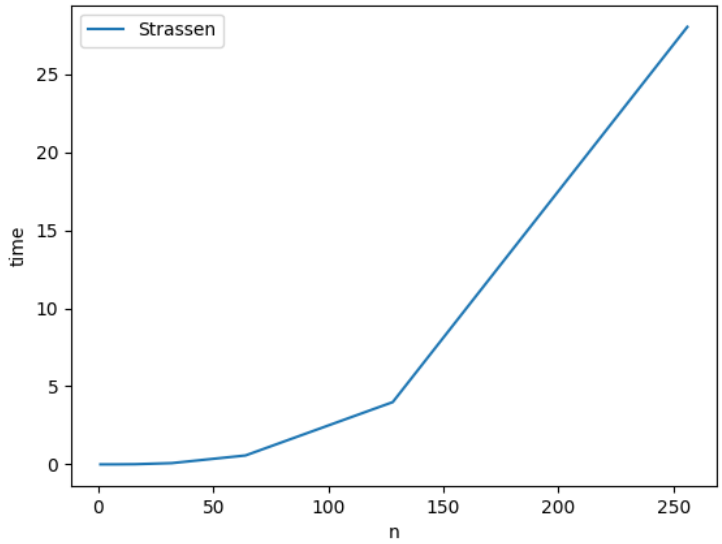
4.1 Czas działania

- **Algorytm Binet'a**



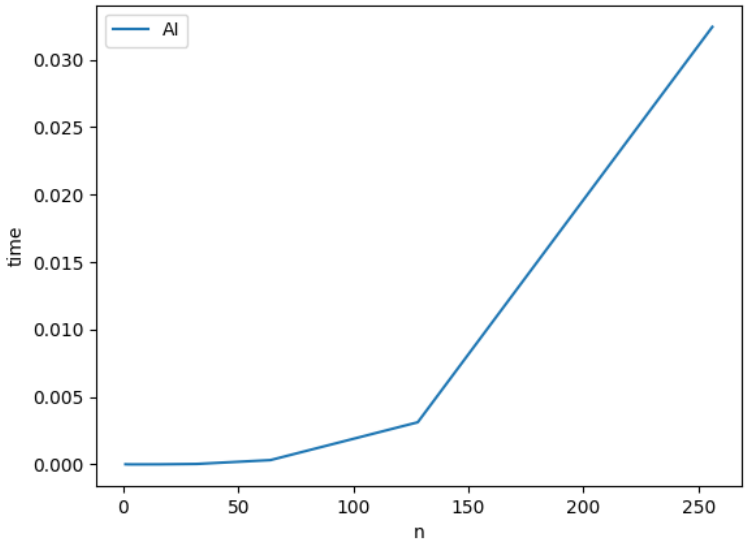
Rysunek 5: Wynik działania algorytmu Binet'a

- **Algorytm Strassena**



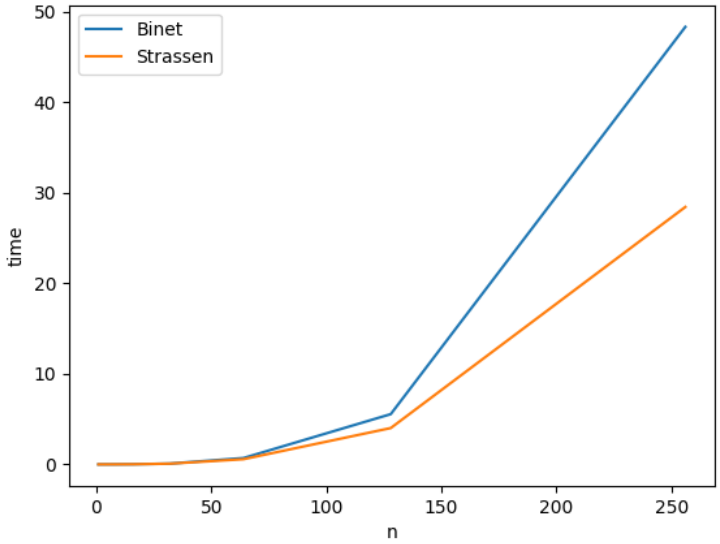
Rysunek 6: Wynik działania algorytmu Strassena

- **Algorytm AI**



Rysunek 7: Wynik działania algorytmu AI

- **Zestawienie algorytmu Binet'a i algorytmu Strassena**



Rysunek 8: Zestawienie wyniku działania algorytmów Binet'a i Strassena

4.2 Liczba operacji zmiennoprzecinkowych

Tabela 1. Tabela z porównaniem liczby operacji zmiennoprzecinkowych

Liczba operacji zmiennoprzecinkowych		
n	**Binet**	**Strassen**
0	18	20
1	234	340
2	1264	1926
3	12854	15414
4	101844	123678
5	7928304	9877738
6	16783224	22908150
7	54239124	63265206

5 Złożoność obliczeniowa

DOKONANO próby oszacowania złożoności obliczeniowej omawiancyh algorytmów podejściem eksperymentalnym i teoretycznym. Pozwoliło to na empiryczne oszacowanie poprawności teoretycznie wyzanczonej złożoności.

5.1 Oszacowanie eksperymentalne

Przeprowadzone testy wydajnościowe wykazały, że oszacowane teoretycznie złożoności mają odzwierciedlenie w praktyce, możliwe jest zastosowanie dodatkowych narzędzie, które dokładniej wyzaczą dokładną wartość empirycznie otrzymanych złożoności, ale zasadnym jest twierdzenie, że algorytmy zostały zaimplementowane poprawnie i prawidłowo oszacowano ich złożoności obliczeniowe (w szczególności złożoności czasowe).

5.2 Oszacowanie teoretyczne

1. **Algorytm Binet'a**
   Algorytm klasycznego mnożenia macierzy zakłada bezpośrednie mnożenie bloków macierzy A i B. Przykładowo, dla macierzy podzielonych na cztery bloki:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

   W przypadku takiego podejścia koszt składa się z:

- 8 mnożeń (względem ilości elementów wynikowej macierzy, każde pole to jedno mnożenie): $O((n/2)^3)$,
- 4 dodawania: $O((n/2)^2)$.

Uwzględniając koszt mnożenia bloków $A_{ij} \times B_{ij}$, ostatecznie, dla całego algorytmu klasycznego mnożenia macierzy o wymiarach $n \times n$, koszt to $8 \cdot O(n^3) + 4 \cdot O(n^2) = O(n^3)$.

2. **Algorytm Strassena**

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

gdzie:

- $\mathrm{C}_{11} = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) - (A_{12} + A_{22}) \cdot B_{11} + A_{11} \cdot (B_{12} - B_{22}) - A_{22} \cdot (B_{21} - B_{11})$,
- $\mathrm{C}_{12} = A_{11} \cdot (B_{12} + B_{22}) + (A_{12} - A_{22}) \cdot B_{22}$,
- $\mathrm{C}_{21} = (A_{21} + A_{22}) \cdot B_{11} + A_{22} \cdot (B_{21} + B_{22})$,
- $\mathrm{C}_{22} = (A_{11} + A_{12}) \cdot B_{22} - (A_{21} + A_{11}) \cdot B_{11} + A_{12} \cdot (B_{21} + B_{11}) - A_{22} \cdot (B_{12} + B_{22})$.

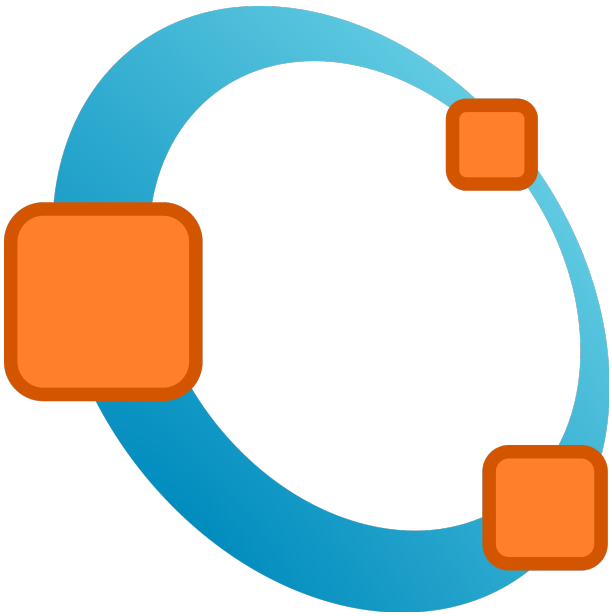
  Standardowe mnożenie macierzy wymaga $O(n^3)$ operacji, ponieważ potrzebne są 8 rekurencyjnych operacji mnożenia macierzy. Jednakże, w metodzie Strassena, dzięki innemu podejściu rekurencyjnemu, potrzebne są jedynie 7 rekurencyjnych mnożeń i $O(n^2)$ operacji dodawania lub odejmowania. To prowadzi do złożoności obliczeniowej $O(n^{\log_2 7})$, co jest w przybliżeniu równoważne $O(n^{2.81})$. Metoda Strassena zatem redukuje złożoność czasową mnożenia macierzy, co jest szczególnie korzystne dla dużych macierzy.

6 Porównanie wyników

Wykorzystano narzędzie Octave do przeliczenia przykładu dla wybranej małej macierzy i porównano wynik z tym, otrzymanym własną implementacją.

6.1 Octave

GNU Octave to darmowy otwartoźródłowy język programowania oraz środowisko do obliczeń numerycznych, które zostało zaprojektowane przede wszystkim do wykonywania zadań związanych z matematyką, inżynierią i naukami ścisłymi. Octave jest podobny do komercyjnego oprogramowania MATLAB, co oznacza, że ma wiele funkcji do manipulacji macierzami, obliczeń matematycznych, tworzenia wykresów i analizy danych.



Rysunek 9: Logo GNU Octave (3)

6.2 Wykorzystanie Octave

Program ten umożliwia wykonywanie zaawansowanych obliczeń matematycznych i numerycznych, obsługuje operacje na wektorach i macierzach, rozwiązywanie równań różniczkowych, algorytmy optymalizacji, analizę sygnałów, przetwarzanie obrazów i wiele innych zastosowań.

Jego interfejs użytkownika opiera się na wierszu poleceń, ale dostępne są również różne interfejsy graficzne, które ułatwiają pracę z programem. Octave ma duże wsparcie społeczności, co oznacza, że istnieje wiele dodatkowych pakietów i rozszerzeń dostępnych do pobrania, rozszerzających jego funkcjonalność.

Jest to potężne narzędzie do naukowców, inżynierów, studentów i każdego, kto zajmuje się analizą danych, matematyką czy innymi dziedzinami, które

wymagają zaawansowanych obliczeń numerycznych. Ponadto, jako projekt o otwartym kodzie źródłowym, Octave jest stale rozwijany i udoskonalany przez społeczność użytkowników i programistów.

6.3 Macierze w Octave

W GNU Octave macierze są fundamentalnym elementem, ponieważ wiele operacji w programie jest opartych na manipulacji nimi. Macierze w Octave są wielowymiarowymi tablicami liczb, które mogą zawierać liczby całkowite, zmiennoprzecinkowe, zespolone oraz inne typy danych.

Macierze w Octave pozwalają na wykonywanie różnorodnych operacji, takich jak dodawanie, odejmowanie, mnożenie, dzielenie, transponowanie czy obliczenia macierzowe. Są niezwykle elastyczne i przydatne w analizie danych, numerycznych obliczeniach oraz inżynierii. Operacje na macierzach w Octave są dość proste. Oto kilka przykładów operacji macierzowych:

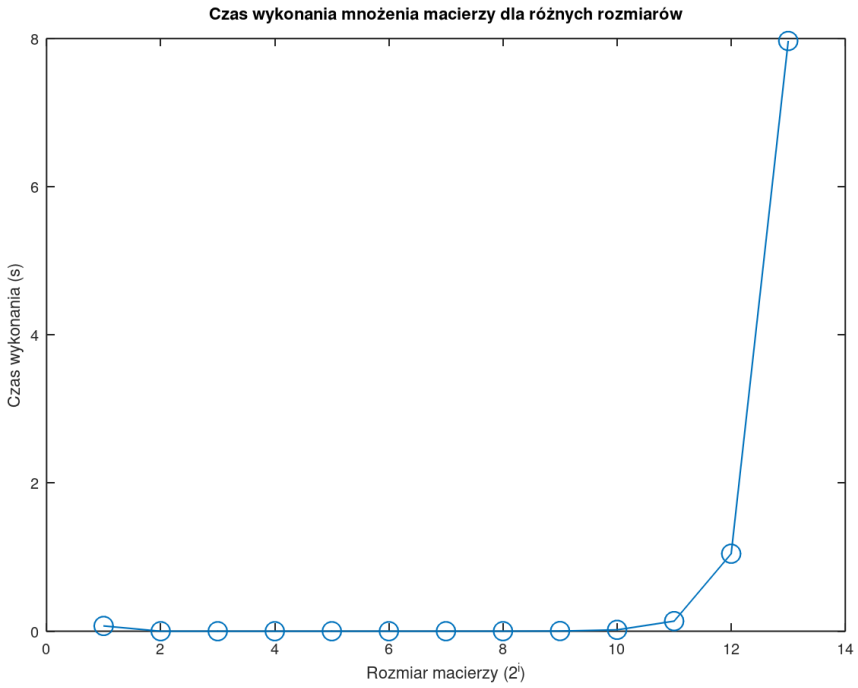
```
A = []  % Tworzy pustą macierz
B = [1 2 3; 4 5 6; 7 8 9]  % Tworzy macierz 3x3
C = ones(2, 3)  % Tworzy macierz 2x3 wypełnioną jedynkami
D = zeros(4, 2) % Tworzy macierz 4x2 wypełnioną zerami
E = eye(3)      % Tworzy macierz jednostkową 3x3

B(2, 3)  % Pobiera wartość z drugiego wiersza i trzeciej kolumny macierzy B
F = B + C   % Dodaje macierze B i C element po elemencie
G = B * D   % Mnoży macierze B i D
H = B'      % Transponuje macierz B
```

6.4 Porównanie wyników

Kod testujący:

```
sizes = [];
execution_times = [];

for i = 1:13
    size = 2^i;
    A = rand(size);
    B = rand(size);

    tic;
    C = A * B;
    execution_time = toc;
```

```
    sizes = [sizes, i];
    execution_times = [execution_times, execution_time];
end

plot(sizes, execution_times, '-o');
title('Czas wykonania mnożenia macierzy dla różnych rozmiarów', 'FontSize', 14);
xlabel('Rozmiar macierzy (2^i)');
ylabel('Czas wykonania (s)');
```



Rysunek 10: Wyniki w Octave

Udało się wykonać mnożenia dla $i = 1, 2, 3, ..., 13$.

7 Wnioski

LABORATORIUM wykazało istotność zagadnienia mnożenia macierzy oraz efektywność wykorzystania rekurencji do jego realizacji.

Refleksje płynące z laboratirum:

- **Złożoność obliczeniowa**
  Rekurencyjne podejście do mnożenia macierzy może zmniejszyć ilość potrzebnych operacji w porównaniu do klasycznego podejścia, szczególnie dla dużych macierzy. To właśnie złożoność obliczeniowa jest jednym z głównych argumentów przemawiających za zastosowaniem rekurencji w tym kontekście.

- **Podział na mniejsze problemy**
  Rekurencyjne podejście opiera się na podziale problemu mnożenia macierzy na mniejsze podproblemy, co może zwiększyć czytelność kodu i ułatwić jego implementację.

- **Koszt pamięciowy**
  Implementacja rekurencyjnego algorytmu mnożenia macierzy może wymagać dodatkowej pamięci ze względu na rekurencyjne wywołania. To może stanowić istotny czynnik, szczególnie dla dużych macierzy, gdzie zarządzanie pamięcią staje się kluczowe.

- **Efektywność w praktyce**
  Rekurencyjne podejście do mnożenia macierzy może wykazywać lepszą efektywność czasową dla pewnych rozmiarów macierzy, ale może być mniej wydajne w innych przypadkach, zwłaszcza ze względu na narzut związany z rekurencją.

- **Optymalizacje i algorytmy hybrydowe**
  Często stosowane są optymalizacje rekurencyjnych algorytmów mnożenia macierzy, takie jak algorytmy Strassena, które łączą rekurencję z klasycznymi metodami mnożenia. Te podejścia mogą wykorzystywać różne strategie w zależności od rozmiaru macierzy, by osiągnąć optymalne wyniki.

- **Liczba operacji na liczbach zmiennoprzecinkowych**
  Ważnym zagadnieniem jest liczba wykonanych operacji na liczbach zmiennoprzecinkowych podczas realizacji działania algorytmu. Należy pamiętać o jego wpływie na czas obliczeń, dokładnść na działaniu na liczbach zmiennoprzecinowych i złożoność pamięciowa. Konieczne jest dobre zrozumienie tematu działania i reprezentacji liczb zmiennoprzecinkowych w komputerze.

- **Wpływ platformy sprzętowej, wykorzystanych narzędzi i implementacji algorytmów**
  Wnioski z laboratorium mogą także uwzględniać wpływ konkretnych

platform sprzętowych na wydajność rekurencyjnych algorytmów mnożenia macierzy. Czasem wydajność takich algorytmów może różnić się w zależności od architektury sprzętowej. Wykorzystane narzędzią i implementacja algorytmów wpływa na rezultat działań.

- **Zastosowanie sztucznej inteligencji w algorytmice**
  Odkrycie przez sztuczną inteligencję innego efektywnego algorytmu rekurencyjnego mnożenia macierzy pokazuje trend, który może stać się w przyszłości popularny. Sztuczna inteligencja może być coraz chętniej wykorzystywana do rozwiązywania problemów algorytmicznych, ogólnie informatycznych i matematycznych oraz stosowana w innych dziedzinach nauki. Daje to nadzieję na rozwiązanie wielu problemów, z którymi nadal zmagają się naukowcy.

Temat ten jest szczególnie istotny ze względu na tak szerokie obecnie wykorzystanie mnożenia macierzy w informatyce.

Literatura

[1] Wykłady i laboratoria prowadzone przez Pana prof. dr hab. Macieja Paszyńskiego

[2] Wikipedia - mnożenie macierzy

[3] Wikipedia - GNU Octave

***