Adam Mytnik, Przemysław Rola, Juliusz Wasieleski
Informatyka, III rok, grupa 6
październik 2023

Algorytmy macierzowe – rekurencyjne mnożenie macierzy - sprawozdanie

1. Opis ćwiczenia

Naszym zadaniem było , po wybraniu naszego ulubionego języka, wygenerowanie losowych macierzy których elementy są z przedziału $(10^{-8}, 1)$ i zaimplementowanie algorytmów:

- Rekurencyjnego mnożenia macierzy metodą Binet'a
- Rekurencyjnego mnożenia macierzy metodą Strassena
- Rekurencyjnego mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję.

Następnie, mieliśmy sprawdzić działanie naszych implementacji na losowo wygenerowanych macierzach rozmiarów $2^k \times 2^k$ gdzie $k \in (2, 3, 4, \dots 16)$.

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. Implementacja algorytmów

3.1 Rekurencyjne mnożenie macierzy metodą Bineťa

3.1.1 Pseudokod

BMU*(A,B):* *# Binet Matrix Multiplication*

Jeżeli *A* oraz *B* mają rozmiar 1**:**

Zwróć *A*B*

W przeciwnym wypadku:

Podziel *A* i *B* na 4 równych rozmiarów mniejsze macierze $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

Zapisz do pomocniczych zmiennych *M*:

M0 = BMU(A_11, B_11)
M1 = BMU(A_12, B_21)
M2 = BMU(A_21, B_11)
M3 = BMU(A_22, B_21)
M4 = BMU(A_11, B_12)
M5 = BMU(A_12, B_22)
M6 = BMU(A_21, B_12)
M7 = BMU(A_22, B_22)

Zapisz macierz C jako:

$C_1 = M_0 + M_1$
$C_2 = M_4 + M_5$
$C_3 = M_2 + M_3$
$C_4 = M_6 + M_7$

Zwróć C

3.1.2 Istotne fragmenty implementacji

Implementacja jest dostarczona przez nas jako funkcja BMU(A, B). Jest to po prostu zapisane pseudokodu w języku Python i nie ma jakichkolwiek fragmentów, które wymagały od nas czegoś więcej niż przepisania pseudokodu.

3.2 Rekurencyjne mnożenie macierzy metodą Strassena

3.2.1 Pseudokod

SMU*(A,B):* *# Strassen Matrix Multiplication*

Jeżeli *A* oraz *B* mają rozmiar 1**:**

Zwróć *A*B*

W przeciwnym wypadku:

Podziel *A* i *B* na 4 równych rozmiarów mniejsze macierze $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

Zapisz do pomocniczych zmiennych *M*:

$M_1 = SMU(A_{11}+A_{22}, B_{11}+B_{22})$
$M_2 = SMU(A_{21}+ A_{22}, B_{11})$

M_3 = SMU(A_{11}, B_{12}- B_{22})

M_4 = SMU(A_{22}, B_{21}- B_{11})

M_5 = SMU(A_{11}+ A_{12}, B_{22})

M_6 = SMU(A_{21}- A_{11}, B_{11}+ B_{12})

M_7 = SMU(A_{12}- A_{22} B_{21}+B_{22})

Zapisz macierz C jako:

$C_1 = M_1 + M_4 - M_5 + M_7$

$C_2 = M_3 + M_5$

$C_3 = M_2 + M_4$

$C_4 = M_1 - M_2 + M_3 + M_6$

Zwróć C

3.2.2 Istotne fragmenty implementacji

Implementacja jest dostarczona przez nas jako funkcja SMU(A, B). Jest to po prostu zapisane pseudokodu w języku Python i nie ma jakichkolwiek fragmentów, które wymagały od nas czegoś więcej niż przepisania pseudokodu.

3.3 Rekurencyjne mnożenie macierzy metodą zaproponowaną przez sztuczną inteligencję

3.3.1 Pseudokod

Naszą implementację oparliśmy na artykule znajdującym się na stronie Deep Mind[i], który opisuje artykuł naukowy[ii] opowiadający o tym algorytmie.

AMU*(A,B):* *# Alpha tensor Matrix Multiplication*

Jeżeli *A* oraz *B* mają rozmiar 1 lub mają tylko jeden wiersz lub jedną kolumnę**:**

Zwróć *A*B*

W przeciwnym wypadku:

Podziel *A* i *B* na równych rozmiarów mniejsze macierze

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} & B_{15} \\ B_{21} & B_{22} & B_{23} & B_{24} & B_{25} \\ B_{31} & B_{32} & B_{33} & B_{34} & B_{35} \\ B_{41} & B_{42} & B_{43} & B_{44} & B_{45} \\ B_{51} & B_{52} & B_{53} & B_{54} & B_{55} \end{bmatrix}$$

Zapisz do pomocniczych zmiennych *h*:

h1 = AMU(A_{32} , -B_{21} - B_{25} - B_{31})

h2 = AMU(A_{22} + A_{25} - A_{35}, -B_{25} - B_{51})

h3 = AMU(-A_{31} - A_{41} + A_{42}, -B_{11} + B_{25})

h4 = AMU(A_{12} + A_{14} + A_{34}, -B_{25} - B_{41})

h5 = AMU(A_{15} + A_{22} + A_{25}, -B_{24} + B_{51})

…

h76 = AMU(A_{13} + A_{33}, -B_{11} + B_{14} - B_{15} + B_{24} + B_{34} - B_{35})

Zapisz macierz C jako:

$$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} \\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \end{bmatrix}$$

C_{11} = -h10 + h12 + h14 - h15 - h16 + h53 + h5 - h66 - h7

C_{12} = h10 + h11 - h12 + h13 + h15 + h16 - h17 - h44 + h51

…

C_{45} = -h12 - h29 + h30 - h34 + h35 + h39 + h3 - h45 + h57 + h59

Zwróć C

Warto zwrócić uwagę, że algorytm zaproponowany przez Deep Mind mnoży macierze rozmiarów $4^n \times 5^m$ *i* $5^m \times 5^k$ co daje nam macierz rozmiaru $4^n \times 5^k$. Jest to znacząca różnica ponieważ algorytmy Strassena i Bineta mnożą macierze rozmiarów $2^l \times 2^l$, gdzie $k, l, n, m \in N$

3.3.2 Istotne fragmenty implementacji

Implementacja jest dostarczona przez nas jako funkcja AMU(A, B). Jest to po prostu zapisane pseudokodu w języku Python i nie ma jakichkolwiek fragmentów, które wymagały od nas czegoś więcej niż przepisania pseudokodu.

4. Analiza wykonanych pomiarów

4.1 Pomiary mnożenia macierzy metodą Bineťa

4.1.1 Pomiary

Liczba elementów macierzy	operacje addytywne	operacje multiplikatywne	Wszystkie operacje zmiennoprzecinkowe	czas wykonania [s]
16	48	64	112	0.000698
64	448	512	960	0.003138
256	3840	4096	7936	0.026383
1024	31744	32768	64512	0.185999
4096	258048	262144	520192	1.915873
16384	2080768	2097152	4177920	15.815352
65536	16711680	16777216	33488896	122.837212
262144	133955584	134217728	268173312	960.272768

Tab. 1 Pomiary mnożenia macierzy metodą Bineta

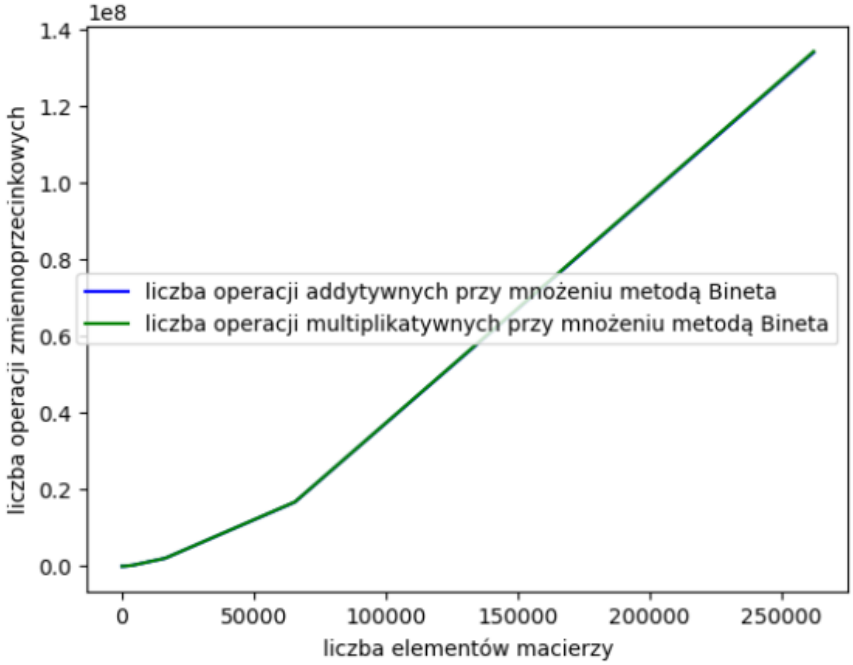
Gdzie rozmiar macierzy to ilość elementów pojedynczej macierzy którą wymnażamy.

4.1.2 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 1. Można zauważyć, że operacji addytywnych jak i multipikatywnych jest prawie tyle samo, czyli liczba mnożeń i dodawań jest zbilansowana i chcąc przyśpieszać niskopoziomowo pojedyncze operacje

zmiennoprzecinkowe musimy zajmować się i dodawaniem, i mnożeniem ponieważ oba mają podobne znaczenie.



Wykres 1

Szacunek złożoności obliczeniowej

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.

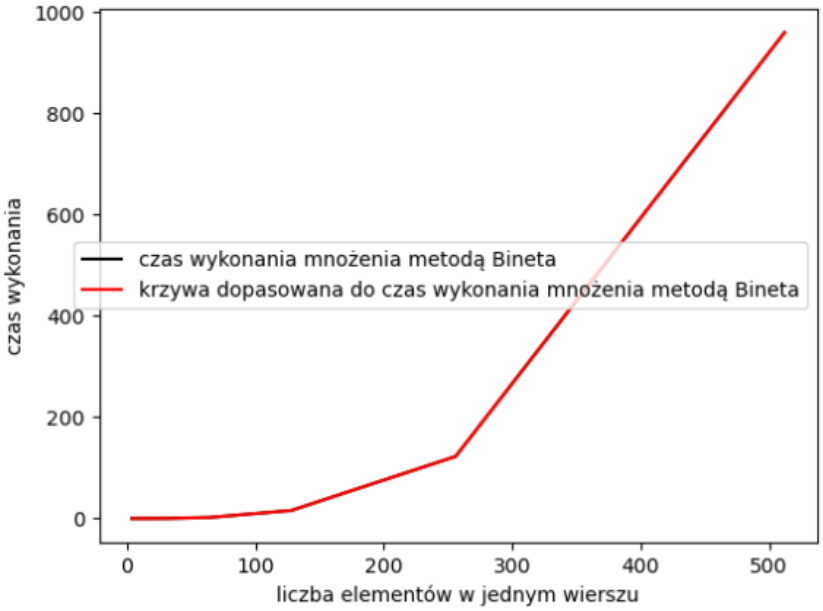
Na podstawie czasu mnożenia macierzy metodą Bineta otrzymaliśmy:

$$a = 8{,}697 \cdot 10^{-6}$$
$$k = 2{,}968 \tag{2}$$

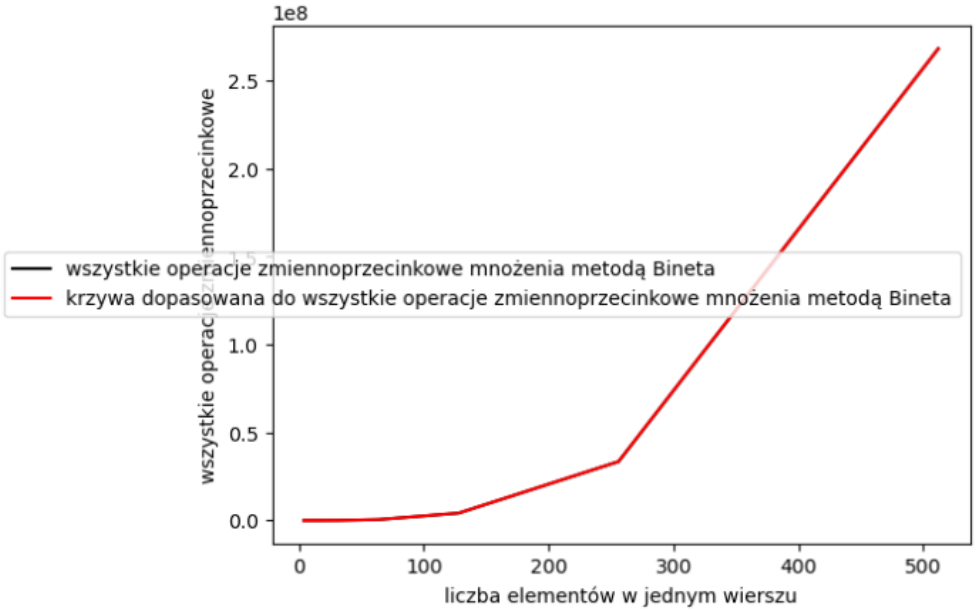
A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą Bineta otrzymaliśmy:

$$a = 1{,}979$$
$$k = 3{,}001 \tag{3}$$

Nasze dopasowane funkcje przedstawiliśmy graficznie razem z oryginalnymi danymi na wykresach numer 2 i 3. Na podstawie tych wykresów możemy stwierdzić, że udało nam się dość dobrze oszacować prawdziwą złożoność. Dodatkowo szacunki te pokrywają się z teoretyczną złożonością, która wynosi $O(n^3)$.



Wykres nr. 2



Wykres nr. 3

Zależność liczby operacji zmiennoprzecinkowych od czasu wykonania

Patrząc na fakt, że szacunki złożoności obliczeniowej dały bardzo zbliżone wyniki jeśli chodzi o wykładnik k zarówno dla czasu jak i dla liczby operacji zmiennoprzecinkowych są zbliżone do siebie. Z tego możemy też wysnuć wniosek że większość czau idzie na samo mnożenie macierzy i nie tracimy zbyt dużo na zarządzaniu pamięcią.

4.2 Pomiary mnożenia macierzy metodą Strassena

4.2.1 Pomiary

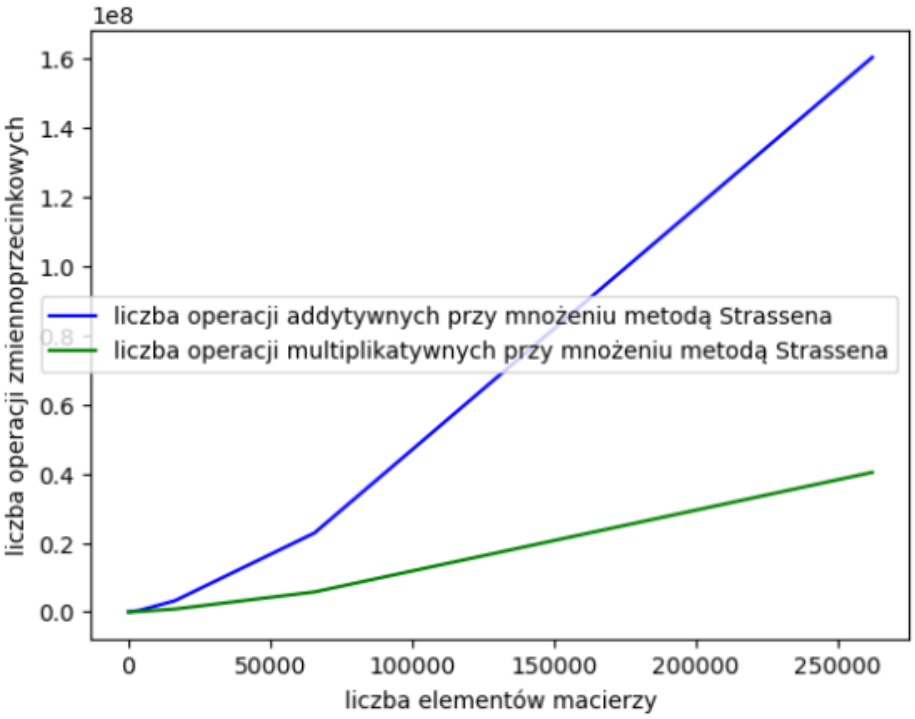
Liczba elementów macierzy	operacje addytywne	operacje multiplikatywne	Wszystkie operacje zmiennoprzecinkowe	czas wykonania [s]
16	132	49	181	0.001424
64	1116	343	1459	0.005570
256	8580	2401	10981	0.036255
1024	63132	16807	79939	0.249674
4096	454212	117649	571861	1.720127
16384	3228636	823543	4052179	15.953701
65536	22797060	5764801	28561861	112.551242
262144	160365852	40353607	200719459	790.659609

Tab. 1 Pomiary mnożenia macierzy metodą Bineta

4.2.2 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 4. Można zauważyć, że operacji multiplikatywnych jest znacznie mniej niż addytywnych. Jest to istotna cecha tego algorytmu ponieważ przyspieszając dodawanie niskopoziomowo możemy uzyskać dużo lepsze czasy działania tego algorytmu.



Wykres 4

Szacunek złożoności obliczeniowej

Złożoność obliczeniową szacowaliśmy empirycznie zgodnie ze wzorem nr 1, tak jak w punkcie 4.1.2.

Na podstawie czasu mnożenia macierzy metodą Strassena otrzymaliśmy:

$$a = 1{,}931 \cdot 10^{-5}$$
$$k = 2{,}809$$

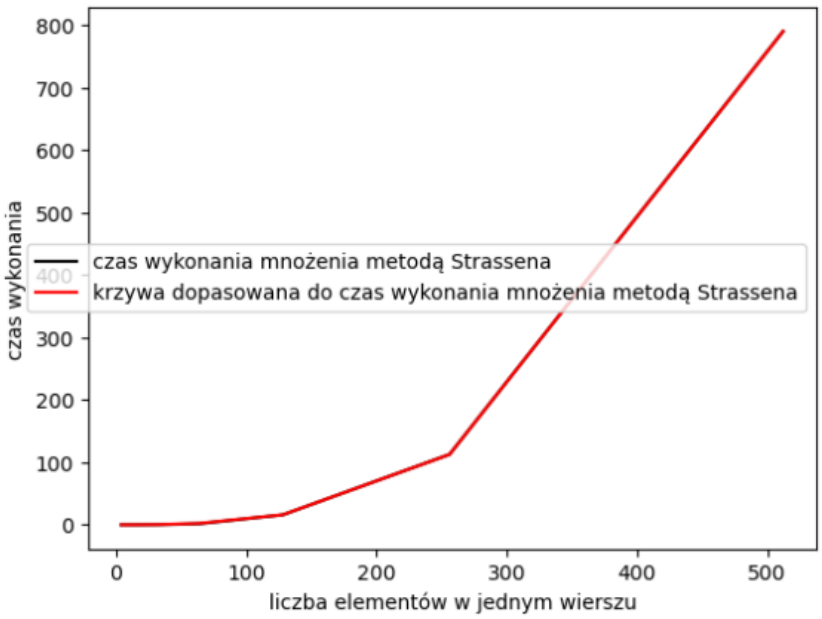
(4)

A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą Bineta otrzymaliśmy:

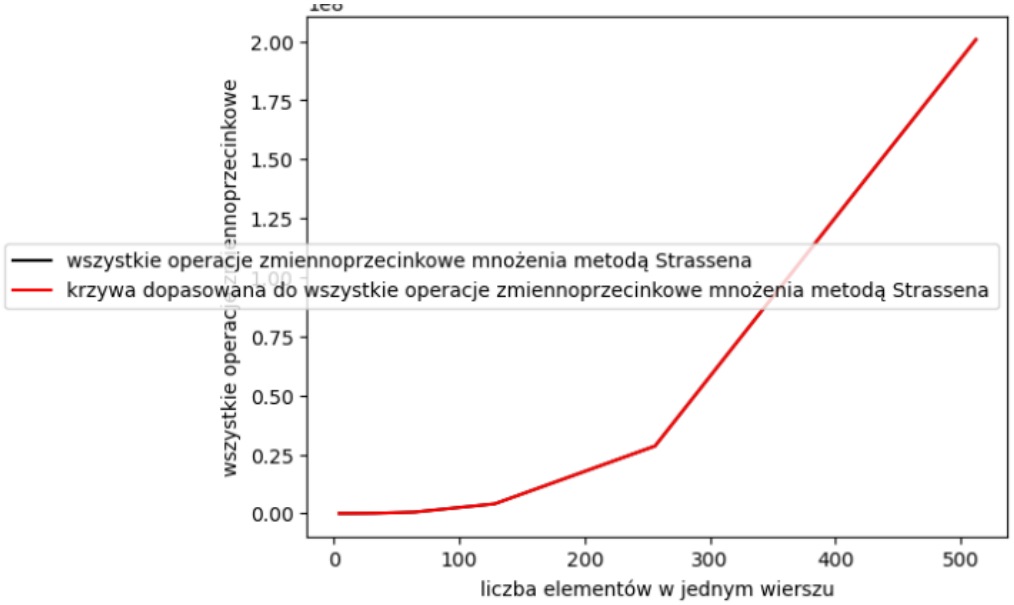
$$a = 4{,}796$$
$$k = 2{,}813$$

(5)

Nasze dopasowane funkcje przedstawiliśmy graficznie razem z oryginalnymi danymi na wykresach numer 5 i 6. Na podstawie tych wykresów możemy stwierdzić, że udało nam się dość dobrze oszacować prawdziwą złożoność, ponieważ oba wykresy nakładają się na siebie. Dodatkowo szacunki te pokrywają się z teoretyczną złożonością, która wynosi $O(n^{2,807})$.



Wykres nr. 5



Wykres nr. 6

Zależność liczby operacji zmiennoprzecinkowych od czasu wykonania

Patrząc na fakt, że szacunki złożoności obliczeniowej dały bardzo zbliżone wyniki jeśli chodzi o wykładnik k zarówno dla czasu jak i dla liczby operacji zmiennoprzecinkowych są zbliżone do siebie. Z tego możemy też wysnuć wniosek że większość czasu idzie na samo mnożenie macierzy i nie tracimy zbyt dużo na zarządzaniu pamięcią.

4.3 Pomiary mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję

4.3.1 Pomiary

Ze względu na fakt że metoda zaproponowana przez sztuczną inteligencję mnoży macierze o wymiarach $4^n \times 5^m$ *i* $5^m \times 5^k$ gdzie $k, n, m \in N$ mieliśmy problem z przeprowadzeniem eksperymentów na tym kodzie, ponieważ za n, m i k mogliśmy przyjmować różne wartości. W celu pewnego uproszczenia ograniczyliśmy się do sprawdzania różnych wartości n i m, i zrobiliśmy założenie m=k. Ze względu na fakt, że tutaj nie w każdej kolumnie porządek rosnący wygląda tak samo, zdecydowaliśmy się posortować nasze wyniki po czasie działania algorytmu.

Liczba elementów macierzy A	Liczba elementów macierzy B	Liczba elementów macierzy A i B	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
20	25	45	539	76	615	0.001860
80	25	105	1586	304	1890	0.004232
320	25	345	5774	1216	6990	0.011366

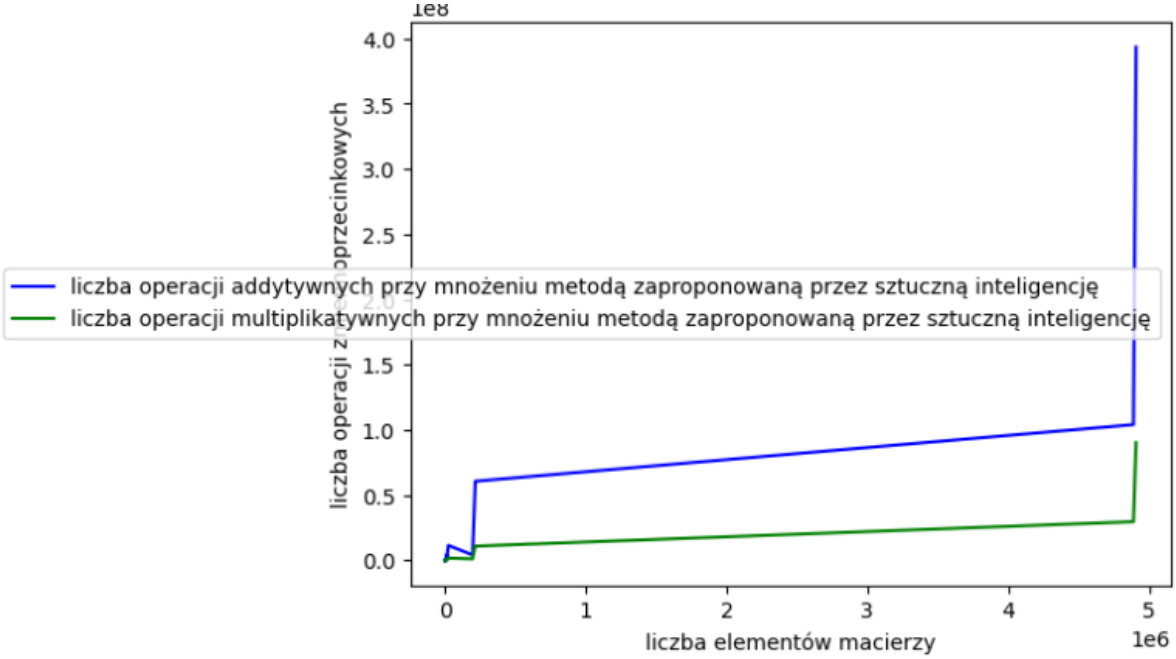
Liczba elementów macierzy A	Liczba elementów macierzy B	Liczba elementów macierzy A i B	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
100	625	725	8015	1900	9915	0.012434
1280	25	1305	22526	4864	27390	0.039976
400	625	1025	51505	5776	57281	0.166569
5120	25	5145	89534	19456	108990	0.157974
1600	625	2225	148450	23104	171554	0.350789
500	15625	16125	173075	47500	220575	0.268171
20480	25	20505	357566	77824	435390	0.818255
6400	625	7025	536230	92416	628646	1.007297
2000	15625	17625	756845	144400	901245	1.357135
25600	625	26225	2087350	369664	2457014	4.183449
8000	15625	23625	4074951	438976	4513927	12.865574
2500	390625	393125	4190375	1187500	5377875	10.508418
32000	15625	47625	11568234	1755904	13324138	28.387908
10000	390625	400625	16267225	3610000	19877225	31.374192
40000	390625	430625	60698075	10974400	71672475	116.267460
12500	9765625	9778125	104076875	29687500	133764375	225.861067
50000	9765625	9815625	393411125	90250000	483661125	820.507655

Tab. 3 Wyniki pomiarów dla algorytmu zaproponowanego przez sztuczną inteligencję

4.3.2 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 4. Można zauważyć, że podobnie do algorytmu Strassena, operacji multiplikatywnych jest znacznie mniej niż addytywnych. Niestety nie jesteśmy w stanie dostrzec żadnego bardziej skomplikowanego trendu poza tym, że liczba operacji wzrasta. Możliwe że niezbyt klarowny kształt wykresu wynika ze źle dobranych rozmiarów macierzy na których testowaliśmy nasz algorytm



Wykres 7

Szacunek złożoności obliczeniowej

Złożoność obliczeniową szacowaliśmy empirycznie zgodnie ze wzorem nr 1, tak jak w punkcie 4.1.2.

Na podstawie czasu mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję otrzymaliśmy:

$$a = 8{,}094 \cdot 10^{-3}$$
$$k = 1{,}438$$

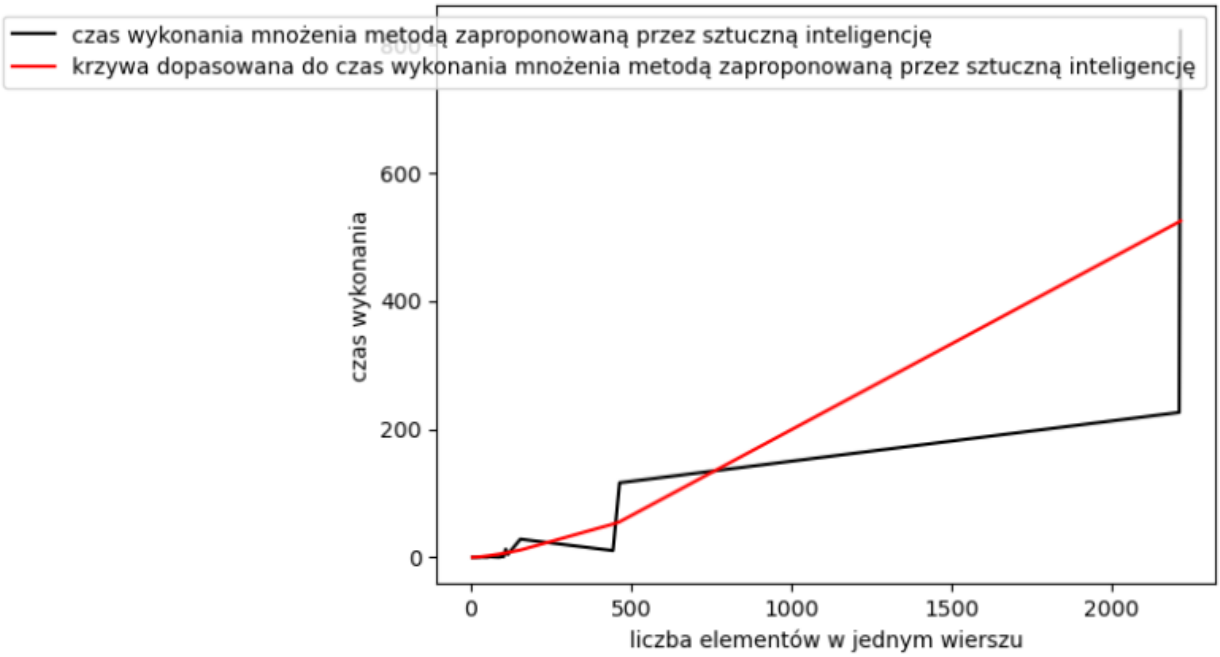
(6)

A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję otrzymaliśmy:

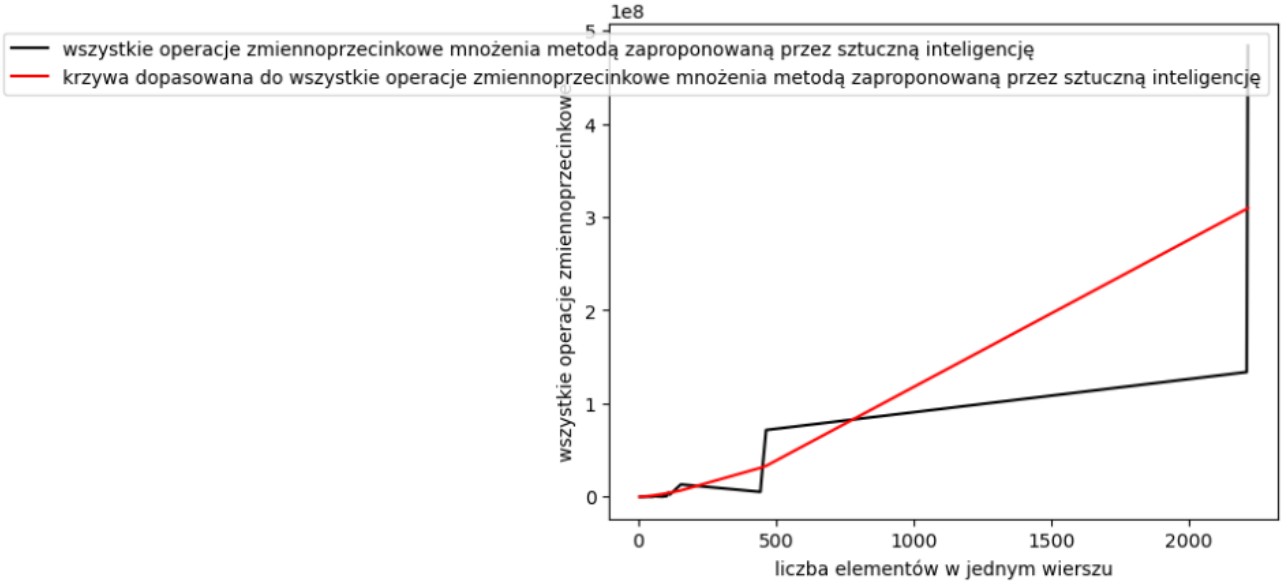
$$a = 5155{,}6$$
$$k = 1,438$$

(7)

Nasze dopasowane funkcje przedstawiliśmy graficznie razem z oryginalnymi danymi na wykresach numer 8 i 9. Na podstawie tych wykresów możemy stwierdzić, że nie udało nam się dość dobrze oszacować prawdziwej złożoności. Wykładnik okazał się zdecydowanie za mały. Prawdopodobnie wynika to z faktu złego dobierania rozmiarów macierzy do testów. Dodatkowo niemożliwa jest złożoność mniejsza niż $O(n^2)$.



Wykres nr. 8

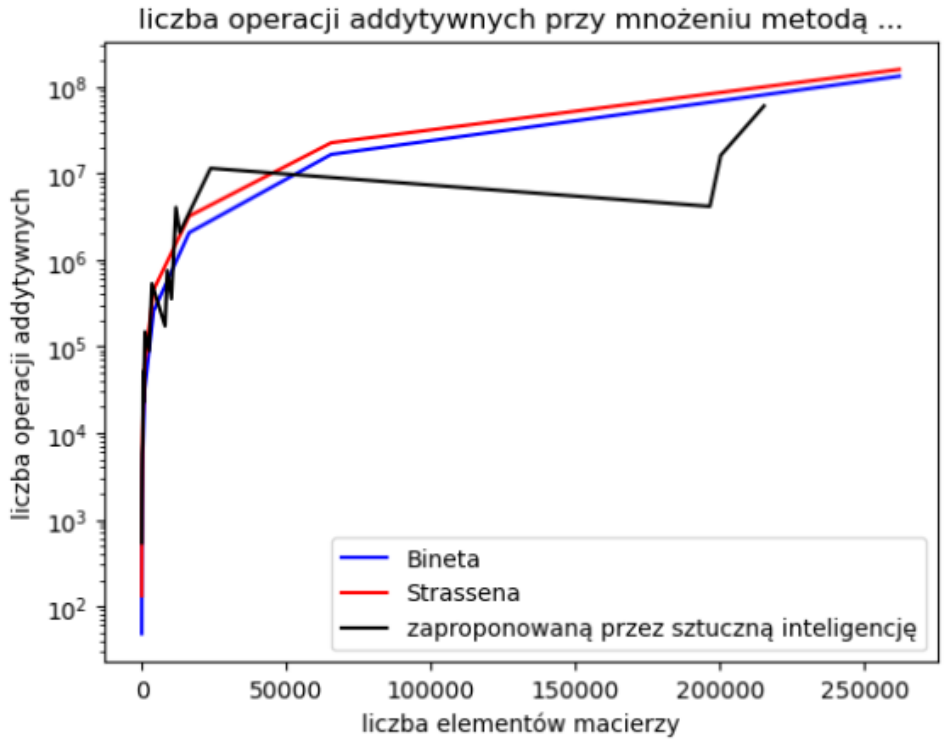


Wykres nr. 9

4.4 Porównanie wyników trzech powyższych algorytmów

4.4.1 Operacje addytywne

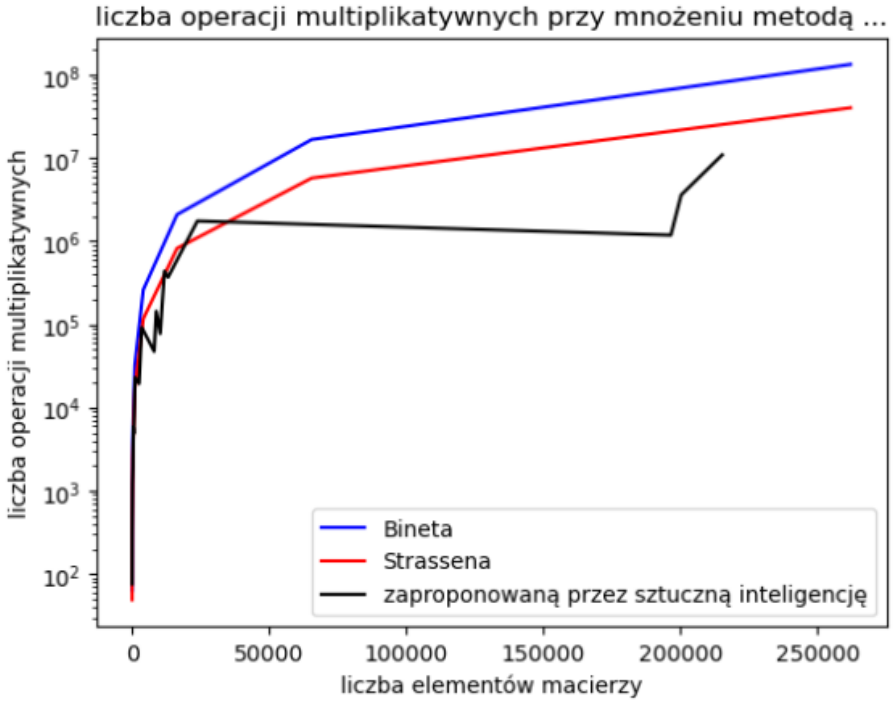
Wykres numer 10 przedstawia graficzne porównanie liczby operacji addytywnych dla wszystkich metod mnożenia macierzy. Możemy na nim dostrzec, że metoda Strassena potrzebuje więcej operacji addytywnych, ale relatywnie nie dużo więcej.



Wykres nr. 10

4.4.2 Operacje multiplikatywne

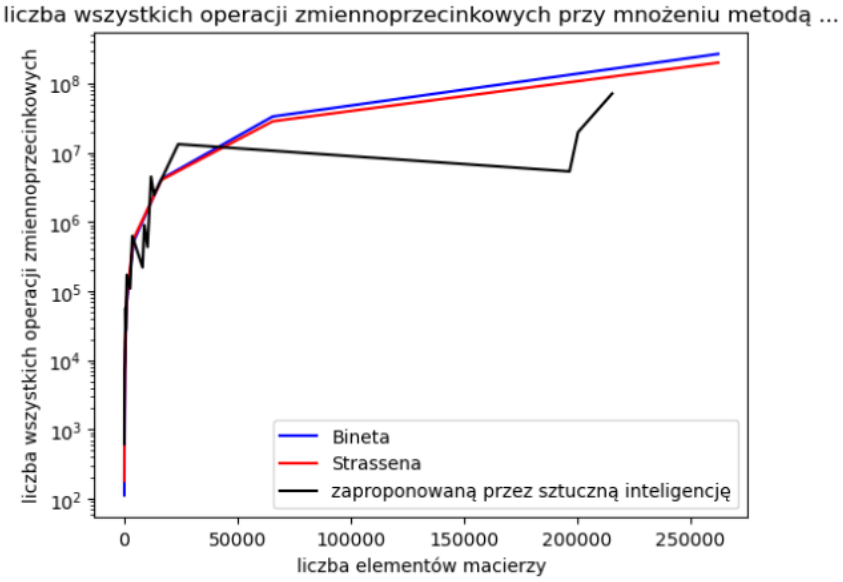
Wykres numer 11 przedstawia graficzne porównanie liczby operacji multiplikatywnych dla wszystkich metod mnożenia macierzy. Na nim widzimy już, że w metodzie Strassena udało się zniwelować liczbę operacji multiplikatywnych i to znacząco.



Wykres nr.11

4.4.3 Wszystkie operacje zmiennoprzecinkowe

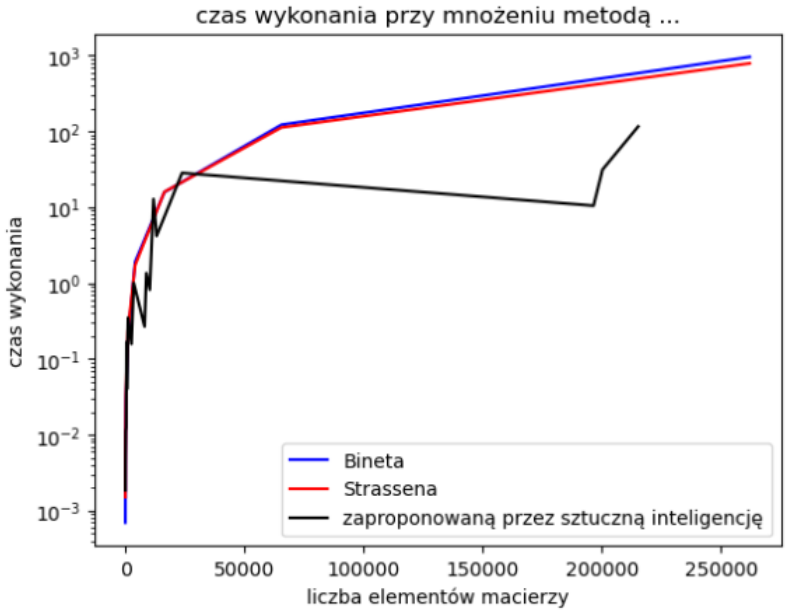
Wykres numer 12 przedstawia graficzne porównanie liczby wszystkich operacji zmiennoprzecinkowych dla obu metod mnożenia macierzy. Możemy na nim dostrzec, że metoda Strassena potrzebuje mniej operacji zmiennoprzecinkowych. Jest to uzyskane faktem, że trochę zwiększając liczbę operacji addytywnych względem metody Bineta w metodzie Strassena zmniesza się znacząco liczbę operacji multiplikatywnych. Dodatkowo widzimy, że dla dużych rozmiarów macierzy metoda proponowana przez sztuczną inteligencję radzi sobie najlepiej.



Wykres nr. 12

4.4.4 Czas działania

Wykres numer 13 przedstawia graficzne porównanie czasu działania dla obu metod mnożenia macierzy. Na nim możemy również dostrzec, że metoda Strassena jest szybsza. Pokrywa się to z szacunkami złożoności obliczeniowej oraz naszymi oczekiwaniami. Owe przyspieszenie zostało uzyskane dzięki zmiejszeniu liczby operacji zmiennoprzecinkowych, które opisaliśmy w punkcie 4.4.3.



Wykres nr. 13

4.5 Porównanie z Octave

4.5.1 Porównanie Octave w naszą implementacją mnożenia macierzy

Żeby różnica była możliwie jak największa zdecydowaliśmy się na porównanie czasau mnożenia dwóch macierzy rozmiarów $2^9 \times 2^9$. Z Tabel nr 1 i 2 możemy odczytać, że naszym implementacją metody Bineta i Strassena zajęło to odpowiednio 960 i 790 sekund. Z kolei w Octave zajęło wymnożenie dwóch macierzy takiej wielkości 411 sekund.

Porównując te wyniki widać od razu supremację języków przeznaczonych do mnożenia macierzy nad np. Pythonem, który jest językiem wolnym (chyba, że impementacja danej funkcji jest zrobiona przy użyciu C++)

4.5.2 Porównanie Octave z wbudowanym mnożeniem

Uznaliśmy, że ciekawym może być porównanie mnożenia macierzy w językach przeznaczonych do operacji na macierzach z mnożeniem macierzy wbudowanym do biblioteki numpy. W ramach tego eksperymentu wygenerowaliśmy dwie macierze rozmiaru $2^{13} \times 2^{13}$i 10 razy wymnożyliśmy je ze sobą zarówno w Pythonie jak i w Octave. Uśredniony czas mnożenia tych macierzy w Pythonie wyniósł 10,16 sekund. Z kolei w Octave 11,23 sekund. Jak można dostrzec wyniki są porównywalne, a co za tym idzie można wysnuć wniosek że mnożenie macierzy z biblioteki numpy może zastąpić mnożenie z języka Octave.

Mogłoby wydawać się, że nastąpił jakiś błąd, bo nie powinno być tak, że mnożenie mniejzej macierzy w podpunkcie 4.5.1 trwało znacąco dłużej. Wynika to jednak z różnic w przechowywaniu macierzy. W podpunkcie 4.5.1 za każdym razem generowaliśmy nową, losową macierz. Z kolei w podpunkcie 4.5.2 wygenerowaliśmy dwie losowe macierze, zapisaliśmy je do plików i potem tylko odczytywaliśmy.

Pokazuje to, że bardzo duży wpływ na czas wykonywania całych programów mają nie tylko algorytmy, ale także sposoby zajmowania się tak dużymi danymi (bo jeden z takich plików potrafił zajmować ponad 1GB miejsca i jak się można domyślić wczytanie tak dużego pliku zajmowało dużo dłużej niż potem samo wymnożenie macierzy)

5. Wnioski

- Metoda zaproponowana przez Strasena jest rzeczywiście szybsza od metody Bineta, a zmniejszenie wykładnika o 0,2 robi różnicę. Mniejsze (lepsze) wyniki dawała metoda Strassena już od macierzy rozmiarów $2^8 \times 2^8$, a dla macierzy rozmiarów $2^7 \times 2^7$ wyniki obu metod były porównywalne.
- Złożoność wynikająca z czasu działania jak i liczby operacji zmiennoprzecinkowych dla obu metod była podobna z czego wynika fakt, że nie ma dużych strat na przenoszeniu macierzy w pamięci, tylko głównie są to same w sobie algorytmy wymnażania tych macierzy.
- Metoda zaproponowana przez sztuczną inteligencję rzeczywiście może być szybsza, ale ze względu na to że mnoży macierze o innych wymiarach nie bylimyw stanie rzetelnie jej porównać. Jej niewątpliwym minusem jest długość, ponieważ w implementacji jest około 100 różnych podstawień (w porównaniu do 7 czy 8 w metodach Bineta i Strassena)
- Mnożenie macierzy z biblioteki numpy może zastąpić mnożenie z Octave, gdyż czasy wykonywania tych działań są do siebie zbliżone

[i] https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor

[ii] Fawzi A., Balog M., Huang A., Hubert T., Romera-Paredes B., Barekatain M., Novikov A., J. R. Ruiz F., Schrittwieser J., Swirszcz G., Silver D., Hassabis D. & Kohli P. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* **610** (2022), str. 47-53