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

Algorytmy macierzowe – rekurencyjne algorytmy macierzowe - sprawozdanie

# Opis ćwiczenia

Naszym zadaniem było , po wybraniu naszego ulubionego języka, wygenerowanie losowych macierzy których elementy są z przedziału i zaimplementowanie algorytmów:

* Rekurencyjnego odwracania macierzy
* Rekurencyjnej LU faktoryzacji macierzy
* Rekurencyjnego obliczania wyznacznika.

Następnie, mieliśmy sprawdzić działanie naszych implementacji na losowo wygenerowanych macierzach rozmiarów gdzie .

# 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.

# Implementacja algorytmów

## Rekurencyjnego odwracania macierzy

### Pseudokod



### 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.

## Rekurencyjnej LU faktoryzacji macierzy

### 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

Zapisz do pomocniczych zmiennych *M*:

M1 = SMU(A11+A22, B11+B22)

M2 = SMU(A21+ A22, B11)

M3 = SMU(A11, B12- B22)

M4 = SMU(A22, B21- B11)

M5 = SMU(A11+ A12, B22)

M6 = SMU(A21- A11, B11+ B12)

M7 = SMU(A12- A22 B21+B22)

Zapisz macierz C jako:

C1 = M1 + M4 - M5 + M7

C2 = M3 + M5

C3 = M2 + M4

C4 = M1- M2 + M3 + M6

**Zwróć**C

### 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.

## Rekurencyjnego obliczania wyznacznika

### Pseudokod

Naszą implementację oparliśmy na artykule znajdującym się na stronie Deep Mind[[1]](#endnote-1), który opisuje artykuł naukowy[[2]](#endnote-2) 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

Zapisz do pomocniczych zmiennych *h*:

h1 = AMU(A32 , -B21 - B25 - B31)  
h2 = AMU(A22 + A25 - A35, -B25 - B51)  
h3 = AMU(-A31 - A41 + A42, -B11 + B25)  
h4 = AMU(A12 + A14 + A34, -B25 - B41)  
h5 = AMU(A15 + A22 + A25, -B24 + B51)

…  
h76 = AMU(A13 + A33, -B11 + B14 - B15 + B24 + B34 - B35)

Zapisz macierz C jako:

C11 = -h10 + h12 + h14 - h15 - h16 + h53 + h5 - h66 - h7

C12 = h10 + h11 - h12 + h13 + h15 + h16 - h17 - h44 + h51

…

C45 = -h12 - h29 + h30 - h34 + h35 + h39 + h3 - h45 + h57 + h59

**Zwróć**C

Warto zwrócić uwagę, że algorytm zaproponowany przez Deep Mind mnoży macierze rozmiarów co daje nam macierz rozmiaru . Jest to znacząca różnica ponieważ algorytmy Strassena i Bineta mnożą macierze rozmiarów , gdzie

### 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.

# Analiza wykonanych pomiarów

## Pomiary rekurencyjnego odwracania macierzy

### 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 |

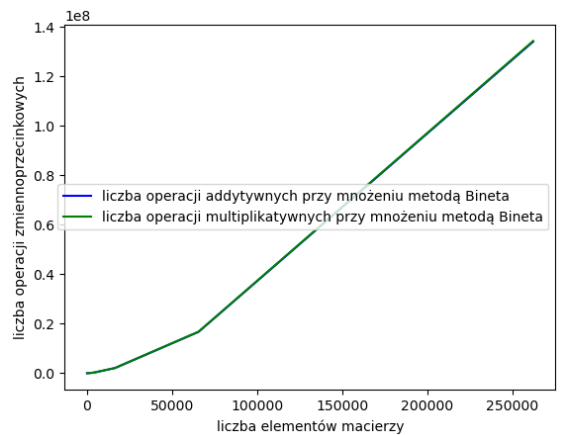
Tab. 1 Pomiary mnożenia macierzy metodą Bineta

Gdzie rozmiar macierzy to ilość elementów pojedynczej macierzy którą wymnażamy.

### 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:

**(1)**

Gdzie próbowaliśmy oszacować a oraz k.

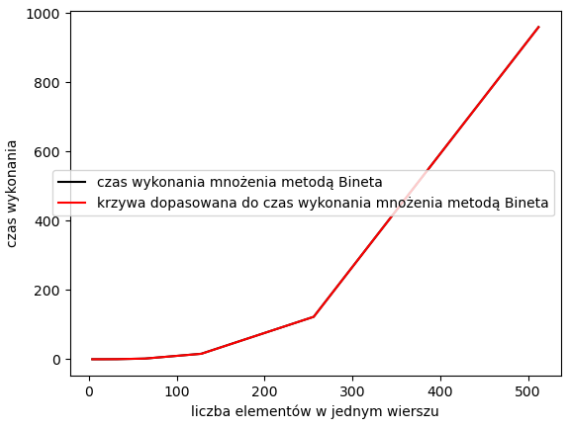
Na podstawie czasu mnożenia macierzy metodą Bineta otrzymaliśmy:

**(2)**

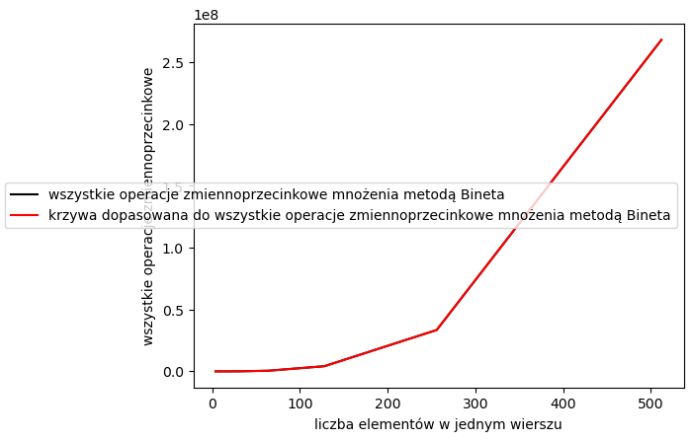
A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą Bineta otrzymaliśmy:

**(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(n3).



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.

## Pomiary LU faktoryzacji macierzy

### Pomiary

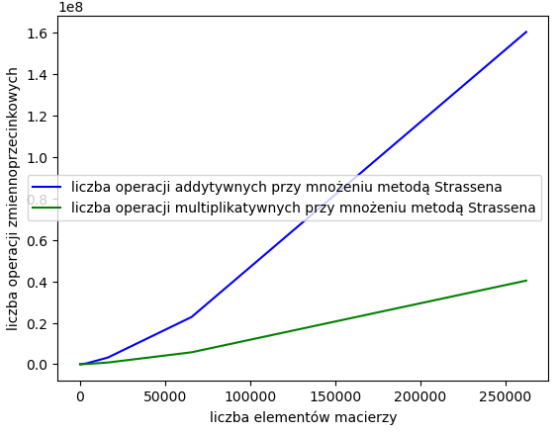
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liczba elementów macierzy | operacje addytywne | operacje multiplikatywne | Wszystkie operacje zmiennoprzecinkowe | czas wykonania [s] |
| 16 | 132 | 49 | 181 | 0.001424 |
| 64 | 1116 | 343 | 1459 | 0.005570 |
| 256 | 8580 | 2401 | 10981 | 0.036255 |
| 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

### 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.

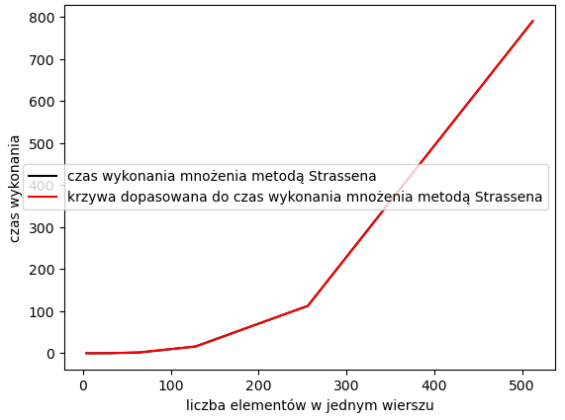
Na podstawie czasu mnożenia macierzy metodą Strassena otrzymaliśmy:

**(4)**

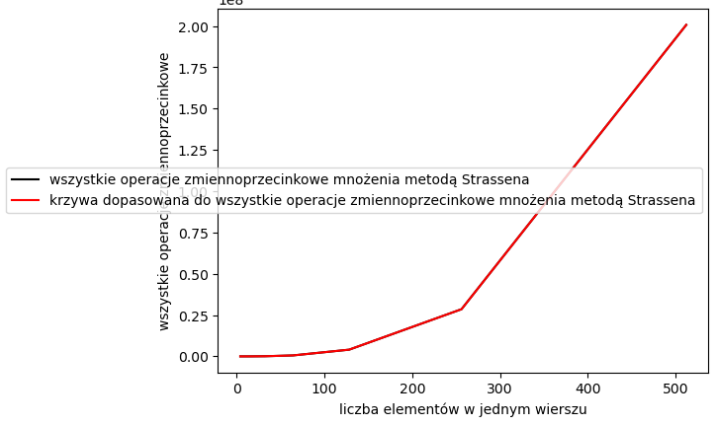
A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą Bineta otrzymaliśmy:

**(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(n2,807).



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.

## Rekurencyjnego obliczania wyznacznika

### Pomiary

Ze względu na fakt że metoda zaproponowana przez sztuczną inteligencję mnoży macierze o wymiarach gdzie 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.

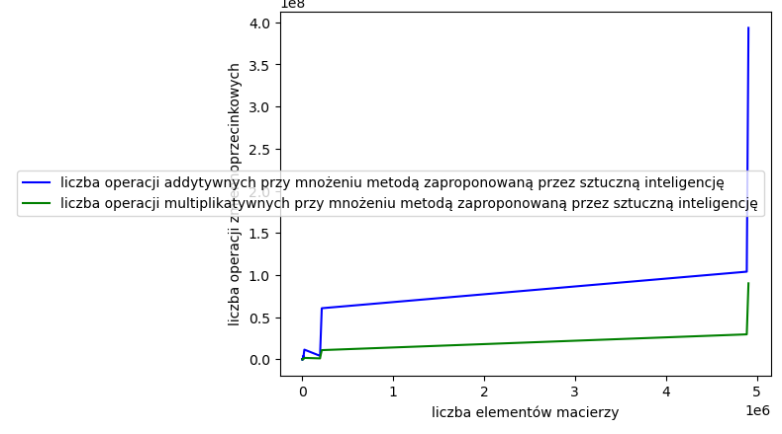
| **Liczba elementów macierzy A** | **Liczba elementów macierzy B** | **Liczba elementów macierzy A i B** | **operacje addytywne** | **operacje multiplikatywne** | **wszystkie operacje zmiennoprzecinkowe** | **czas wykonania** |
| --- | --- | --- | --- | --- | --- | --- |
| 20 | 25 | 45 | 539 | 76 |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

### 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.

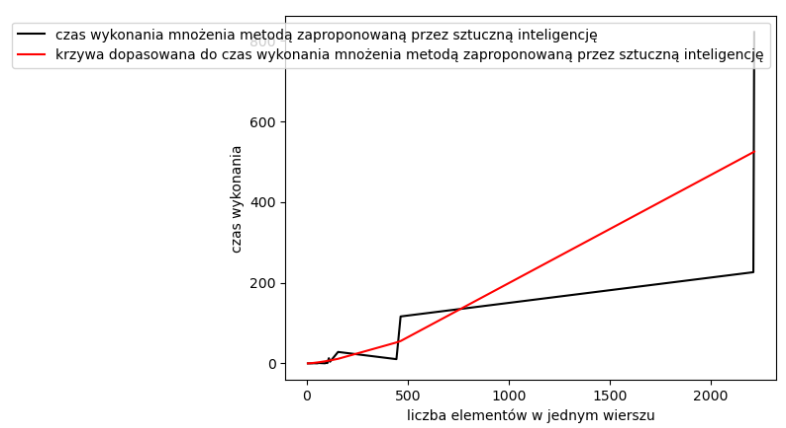
Na podstawie czasu mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję otrzymaliśmy:

**(6)**

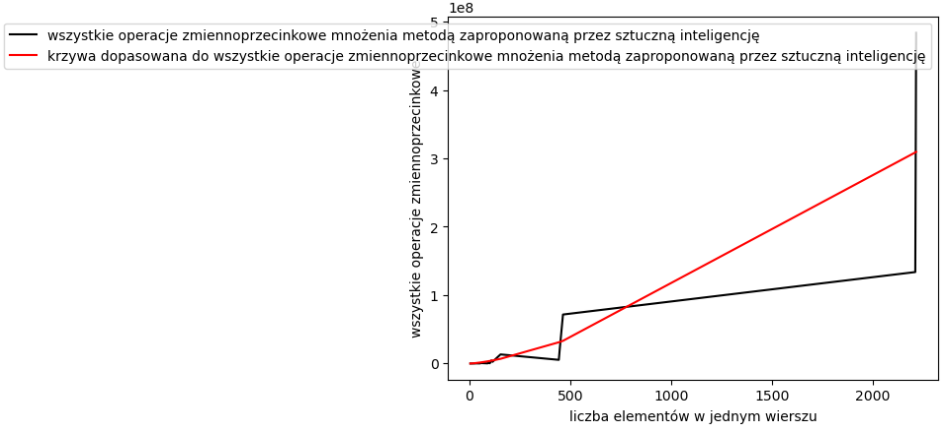
A na podstawie liczby operacji zmiennoprzecinkowych mnożenia macierzy metodą zaproponowaną przez sztuczną inteligencję otrzymaliśmy:

**(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(n2).



Wykres nr. 8

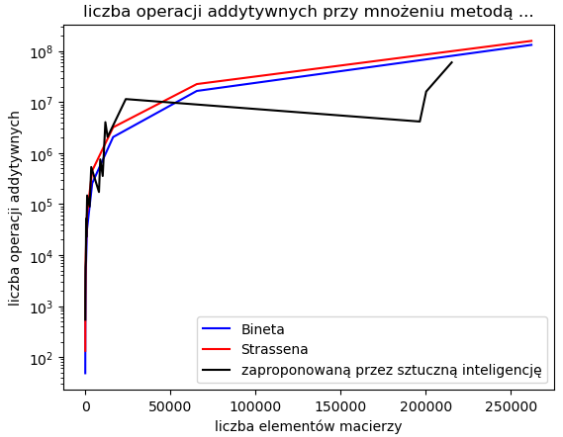


Wykres nr. 9

## Porównanie wyników trzech powyższych algorytmów

### Operacje addytywne

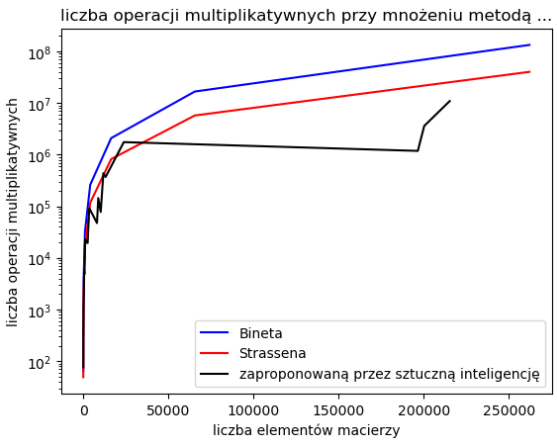
Wykres numer 10 przedstawia graficzne porównanie liczby operacji addytywnych dla wszystkich metod mnożenia macierzy. Możemy na nim dostrzec, że metoda Strassena potrzebuje więcej operacji addytywnych, ale relatywnie nie dużo więcej.



Wykres nr. 10

### Operacje multiplikatywne

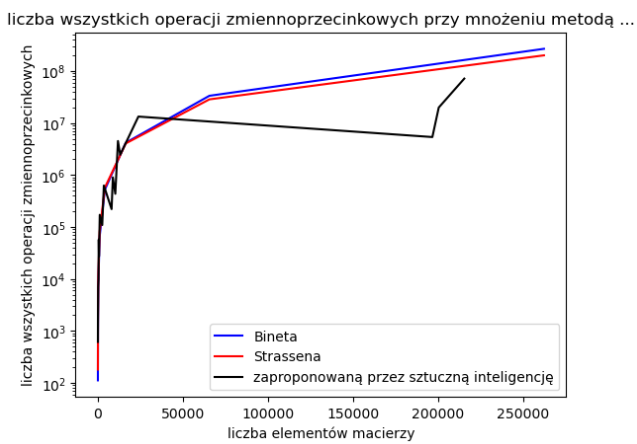
Wykres numer 11 przedstawia graficzne porównanie liczby operacji multiplikatywnych dla wszystkich metod mnożenia macierzy. Na nim widzimy już, że w metodzie Strassena udało się zniwelować liczbę operacji multiplikatywnych i to znacząco.



Wykres nr.11

### Wszystkie operacje zmiennoprzecinkowe

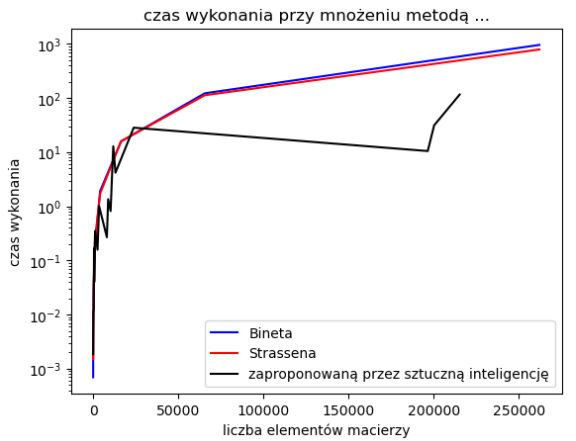
Wykres numer 12 przedstawia graficzne porównanie liczby wszystkich operacji zmiennoprzecinkowych dla obu metod mnożenia macierzy. Możemy na nim dostrzec, że metoda Strassena potrzebuje mniej operacji zmiennoprzecinkowych. Jest to uzyskane faktem, że trochę zwiększając liczbę operacji addytywnych względem 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

### 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

## Porównanie z numpy

### Porównanie Octave w naszą implementacją mnożenia macierzy

Żeby różnica była możliwie jak największa zdecydowaliśmy się na porównanie czasau mnożenia dwóch macierzy rozmiarów . 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++)

### 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 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)

# 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 , a dla macierzy rozmiarów 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 zbliożone

1. https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor [↑](#endnote-ref-1)
2. Fawzi A., [Balog](https://www.nature.com/articles/s41586-022-05172-4#auth-Matej-Balog-Aff1) M., [Huang](https://www.nature.com/articles/s41586-022-05172-4#auth-Aja-Huang-Aff1) A., [Hubert](https://www.nature.com/articles/s41586-022-05172-4#auth-Thomas-Hubert-Aff1) T., [Romera-Paredes](https://www.nature.com/articles/s41586-022-05172-4#auth-Bernardino-Romera_Paredes-Aff1) B., [Barekatain](https://www.nature.com/articles/s41586-022-05172-4#auth-Mohammadamin-Barekatain-Aff1) M., [Novikov](https://www.nature.com/articles/s41586-022-05172-4#auth-Alexander-Novikov-Aff1) A., [J. R. Ruiz](https://www.nature.com/articles/s41586-022-05172-4#auth-Francisco_J_-R__Ruiz-Aff1) F., [Schrittwieser](https://www.nature.com/articles/s41586-022-05172-4#auth-Julian-Schrittwieser-Aff1) J., [Swirszcz](https://www.nature.com/articles/s41586-022-05172-4#auth-Grzegorz-Swirszcz-Aff1) G., [Silver](https://www.nature.com/articles/s41586-022-05172-4#auth-David-Silver-Aff1) D., [Hassabis](https://www.nature.com/articles/s41586-022-05172-4#auth-Demis-Hassabis-Aff1) D. & [Kohli](https://www.nature.com/articles/s41586-022-05172-4#auth-Pushmeet-Kohli-Aff1) P. [Discovering faster matrix multiplication algorithms with reinforcement learning](https://www.nature.com/articles/s41586-022-05172-4). Nature **610** (2022), str. 47-53 [↑](#endnote-ref-2)