# Algorytmy Macierzowe laboratorium 1.

## Mnożenie macierzy

**Antoni Kucharski, Joachim Grys**

22.10.2024

# 1 Wprowadzenie

Celem ćwiczenia było zaimplementowanie trzech algorytmów rekurencyjnego mnożenia macierzy:

- Metoda Binet'a
- Metoda Strassena
- Metoda AI

Dla każdej z metod należało wykonać wykresy zależności

- ilości operacji addytywnych
- ilości operacji multiplikatywnych
- wszystkich operacji zmiennoprzecinkowych
- czasu trwania mnożenia macierzy

od wielkości macierzy wejściowej.

# 2 Algorytm Binet'a

## 2.1 Idea

### 2.1.1 Macierze blokowe

W algorytmie Binet'a należy podzielić wejściowe macierze kwadratowe na cztery bloki zgodnie ze wzorem:

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

Dla macierzy kwadratowych o nieparzystej wielkości, przyjmując $n$ jako liczbę wierszy i kolumn oraz $k = \frac{n-1}{2}$ i $m = \frac{n+1}{2}$ wymiary poszczególnych bloków to:

- $A_{11}$: $k \times k$
- $A_{12}$: $k \times m$
- $A_{21}$: $m \times k$
- $A_{22}$: $m \times m$

### 2.1.2 Algorytm

Dla macierzy kwadratowych $A$, $B$ tej samej wielkości algorytm Binet'a wygląda następująco:

Jeżeli $A = [a]$ i $B = [b]$ to $AB = [ab]$, w przeciwnym wypadku

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

8 mnożeń bloków w macierzy wynikowej to rekurencyjne wywołania algorytmu.

### 2.1.3 Pseudokod

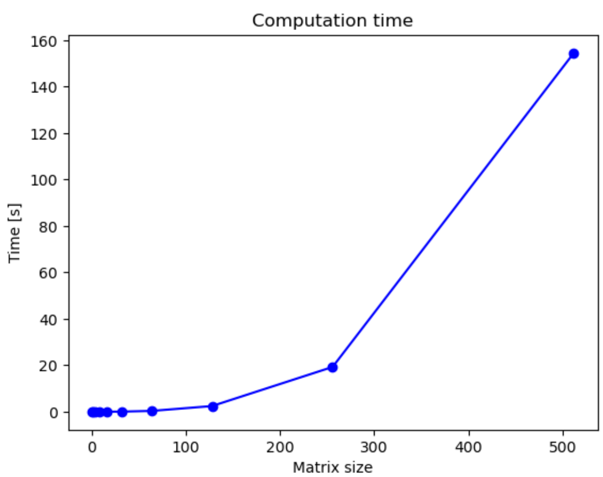
*def* ***partition****(M) do*
*(n, m) = shape(M)*
*n = n / 2, m = m / 2*
***return M[:n, :m], M[:n, m:], M[n:, :m], M[n:, m:]***
*end*

*def* ***binet****([a], [b]), do: [a * b]*
*def* ***binet****(A, B) do*
$A_{11}, A_{12}, A_{21}, A_{22}$ *= partition(A)*
$B_{11}, B_{12}, B_{21}, B_{22}$ *= partition(B)*
***return*** $\begin{bmatrix} \mathbf{binet(A_{11}, B_{11}) + binet(A_{12}, B_{21})} & \mathbf{binet(A_{11}, B_{12}) + binet(A_{12}, B_{22})} \\ \mathbf{binet(A_{21}, B_{11}) + binet(A_{22}, B_{21})} & \mathbf{binet(A_{21}, B_{12}) + binet(A_{22}, B_{22})} \end{bmatrix}$
*end*

## 2.2 Analiza algorytmu

| | A.size | B.size | Time [s] | Additions | Subtractions | Multiplications | Divisions |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0.000023 | 0 | 0 | 1 | 0 |
| 1 | 2 | 2 | 0.000020 | 4 | 0 | 8 | 0 |
| 3 | 4 | 4 | 0.000094 | 48 | 0 | 64 | 0 |
| 7 | 8 | 8 | 0.000657 | 448 | 0 | 512 | 0 |
| 15 | 16 | 16 | 0.005040 | 3840 | 0 | 4096 | 0 |
| 31 | 32 | 32 | 0.036079 | 31744 | 0 | 32768 | 0 |
| 63 | 64 | 64 | 0.346451 | 258048 | 0 | 262144 | 0 |
| 127 | 128 | 128 | 2.434857 | 2080768 | 0 | 2097152 | 0 |
| 255 | 256 | 256 | 19.277508 | 16711680 | 0 | 16777216 | 0 |
| 511 | 512 | 512 | 154.519365 | 133955584 | 0 | 134217728 | 0 |

Tabela 1: Szczegółowe wyniki dla algorytmu Binet'a



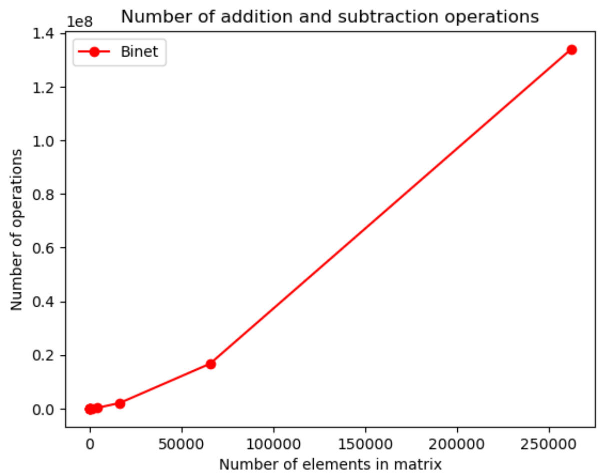

Rysunek 1: Wykres czasu dla macierzy $n \times n$

## 2.3 Złożoność obliczeniowa

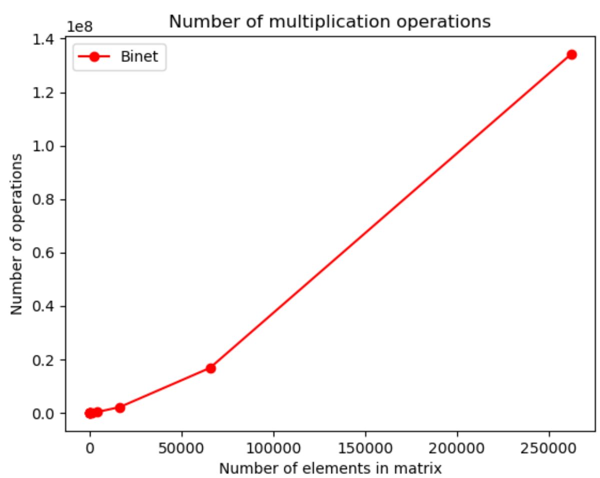
W metodzie Binet'a mamy 8 rekurencyjnych mnożeń i 4 dodawania w każdym kroku. Oznaczając jako $T(n)$ złożoność mnożenia macierzy $n \times n$ otrzymujemy równanie rekurencyjne:

$$T(n) = 8T(\frac{n}{2}) + 4n^2$$

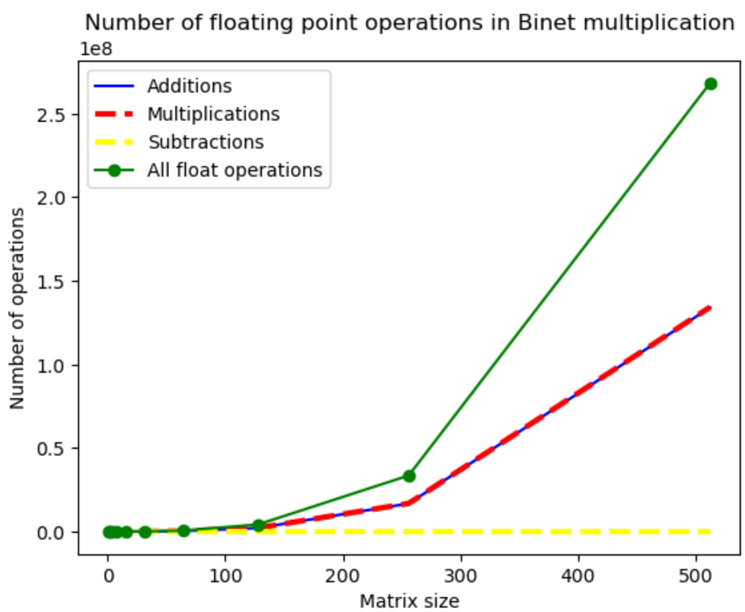
$$T(n) = O(n^{\log_2 8}) = O(n^3)$$




Rysunek 2: Wykres ilości dodawań i odejmowań dla końcowej $n$ elementowej macierzy




Rysunek 3: Wykres ilości mnożeń dla końcowej $n$ elementowej macierzy




Rysunek 4: Wykres ilości operacji zmiennoprzecinkowych dla macierzy $n \times n$

# 3 Algorytm Strassena

## 3.1 Idea

Algorytm Strassena jest efektywną metodą mnożenia macierzy, która redukuje liczbę mnożeń potrzebnych do obliczenia iloczynu dwóch macierzy. Tradycyjna metoda mnożenia dwóch macierzy $n \times n$ wymaga $n^3$ operacji mnożenia. Algorytm Strassena zmniejsza tę liczbę, co prowadzi do szybszego działania przy dużych macierzach

Zamiast wykonywać pełne mnożenie blokowe dwóch macierzy za pomocą 8 operacji mnożenia, algorytm Strassena redukuje tę liczbę do 7 mnożeń dzięki sprytnemu rozkładaniu macierzy na bloki i odpowiedniej manipulacji tymi blokami. Taki algorytm działa tylko dla macierzy $2^n \times 2^n$. Żeby działał dla każdego

**Algorithm 1** Algorytm Strassena dla mnożenia macierzy

**procedure** STRASSEN($A$, $B$)
  **if** $n == 1$ **then**
    **return** $A \times B$
  **end if**
  Podziel $A$ i $B$ na cztery macierze o wymiarach $\frac{n}{2} \times \frac{n}{2}$:
  $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
  Oblicz pomocnicze macierze:
  $M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
  $M_2 = (A_{21} + A_{22}) \times B_{11}$
  $M_3 = A_{11} \times (B_{12} - B_{22})$
  $M_4 = A_{22} \times (B_{21} - B_{11})$
  $M_5 = (A_{11} + A_{12}) \times B_{22}$
  $M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
  $M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$
  Oblicz końcowe bloki macierzy wyniku $C$:
  $C_{11} = M_1 + M_4 - M_5 + M_7$
  $C_{12} = M_3 + M_5$
  $C_{21} = M_2 + M_4$
  $C_{22} = M_1 - M_2 + M_3 + M_6$
  Złóż macierz $C$:
  $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$
  **return** $C$
**end procedure**

rodzaju macierzy kwadratowej trzeba go jednak *"uodpornić"*, uzupełniając na początku macierz zerami. Poniżej jest implementacja tego algorytmu.

```
def Strassen(A: np.ndarray, B: np.ndarray, iter: int = 0) -> np.ndarray:

    shape_start = A.shape[0]
```

```
    if A.shape[0] == 1:
        return A * B

    if iter == 0:
        m = 1
        n = A.shape[0]
        while m < n:
            m *= 2
        A = np.pad(A, ((0, m - A.shape[0]), (0, m - A.shape[1])),
                   mode='constant', constant_values=(0, 0))
        B = np.pad(B, ((0, m - B.shape[0]), (0, m - B.shape[1])),
                   mode='constant', constant_values=(0, 0))

    n = A.shape[0] // 2

    A_11, A_12, A_21, A_22 = A[:n, :n], A[:n, n:], A[n:, :n], A[n:, n:]
    B_11, B_12, B_21, B_22 = B[:n, :n], B[:n, n:], B[n:, :n], B[n:, n:]

    iter_temp = iter + 1

    M1 = Strassen(A_11 + A_22, B_11 + B_22, iter_temp)
    M2 = Strassen(A_21 + A_22, B_11, iter_temp)
    M3 = Strassen(A_11, B_12 - B_22, iter_temp)
    M4 = Strassen(A_22, B_21 - B_11, iter_temp)
    M5 = Strassen(A_11 + A_12, B_22, iter_temp)
    M6 = Strassen(A_21 - A_11, B_11 + B_12, iter_temp)
    M7 = Strassen(A_12 - A_22, B_21 + B_22, iter_temp)

    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[:shape_start, :shape_start]
```

Taki kod działa, natomiast na potrzeby analizy dobierzemy rozmiary macierzy tak, aby nie trzeba było macierzy uzupełniać zerami. Pozwoli to nam na lepszą analize danych.

## 3.2 Analiza algorytmu

Dla macierzy o wymiarach n×nn×n, złożoność można wyrazić jako:

$$T(n) = 7T((\frac{n}{2})^2) + O((\frac{n}{2})^2)$$

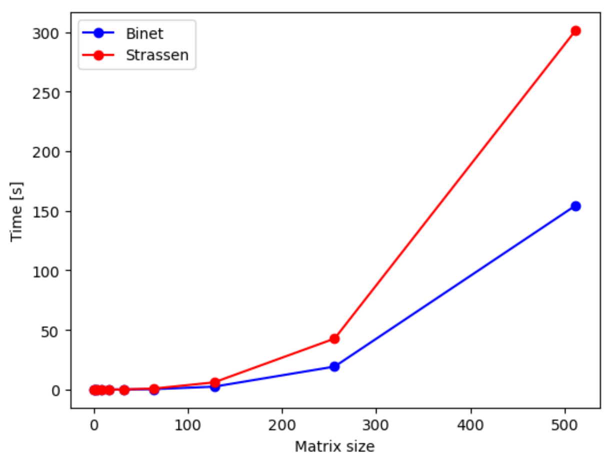
Co daje nam:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

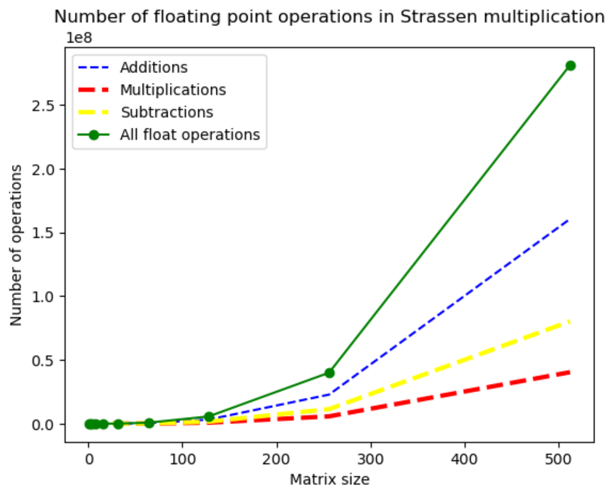
### 3.3 Wyniki

| | A.size | B.size | Time [s] | Additions | Subtractions | Multiplications | Divisions |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0.000013 | 0 | 0 | 1 | 0 |
| 1 | 2 | 2 | 0.000074 | 12 | 6 | 7 | 0 |
| 3 | 4 | 4 | 0.000453 | 132 | 66 | 49 | 0 |
| 7 | 8 | 8 | 0.002539 | 1116 | 558 | 343 | 0 |
| 15 | 16 | 16 | 0.017107 | 8580 | 4290 | 2401 | 0 |
| 31 | 32 | 32 | 0.116922 | 63132 | 31566 | 16807 | 0 |
| 63 | 64 | 64 | 0.822051 | 454212 | 227106 | 117649 | 0 |
| 127 | 128 | 128 | 6.026194 | 3228636 | 1614318 | 823543 | 0 |
| 255 | 256 | 256 | 42.766295 | 22797060 | 11398530 | 5764801 | 0 |
| 511 | 512 | 512 | 301.524905 | 160365852 | 80182926 | 40353607 | 0 |

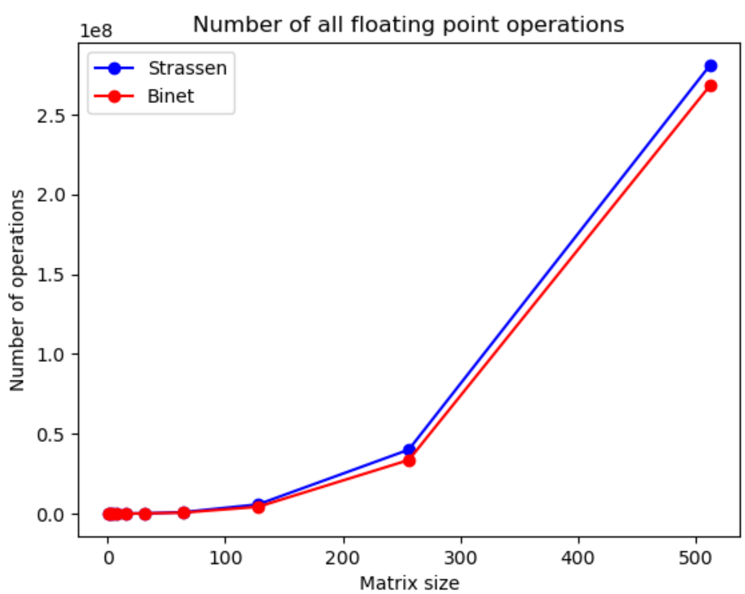
Tabela 2: Szczegółowe wyniki dla algorytmu Strassena



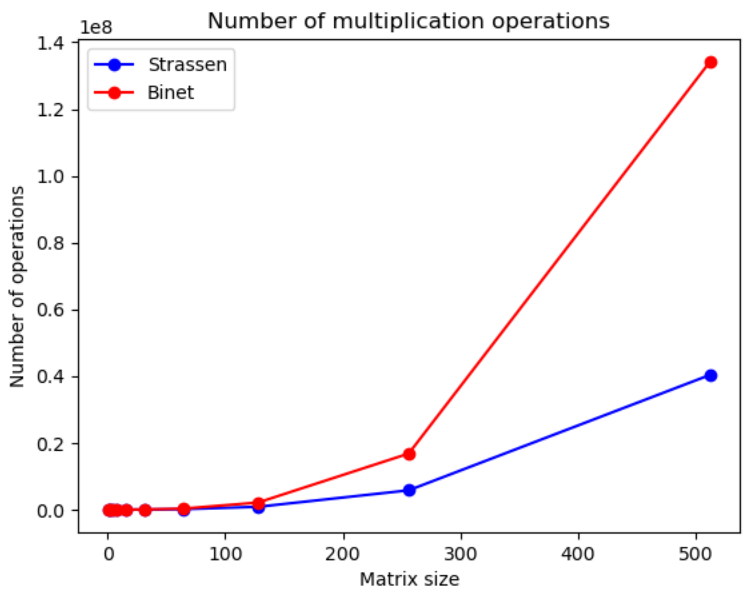

Rysunek 5: Wykres czasu dla macierzy $n \times n$




Rysunek 6: Wykres ilości mnożeń dla końcowej $n$ elementowej macierzy




Rysunek 7: Wykres ilości operacji zmiennoprzecinkowych dla macierzy $n \times n$




Rysunek 8: Wykres ilości mnożeń dla macierzy $n \times n$
Powyższe wykresy pokazują zaletę algorytmu Strassena - jest duża różnica w ilości operacji multiplikatywnych, które w arytmetyce zmiennoprzecinkowej potrafią całkowicie wypaczeć wynik działania. Również

# 4 Algorytm AI

## 4.1 Idea

Algorytm mnożenia macierzy opracowany przez sztuczną inteligencję wykorzystuje uczenie ze wzmocnieniem do odkrywania bardziej efektywnych metod mnożenia. Traktując problem jako grę tensorową, AI eksploruje przestrzeń możliwych algorytmów w poszukiwaniu takich, które minimalizują liczbę operacji mnożenia. Algorytm ten jest jednak ograniczony - działa tylko dla mnożenia macierzy $A - 4^n \times 5^m$ przez $B - 5^m \times 5^k$.

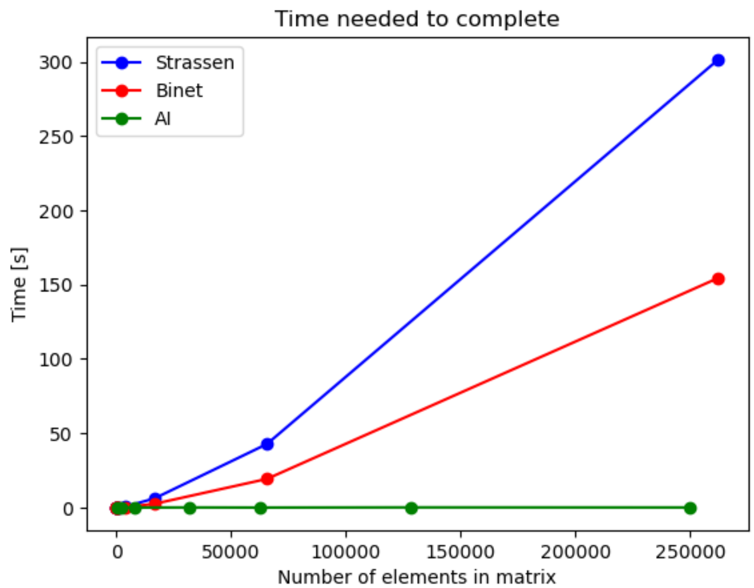
## 4.2 Analiza algorytmu

| | N | M | Time [s] | Additions | Subtractions | Multiplications | Divisions | Size of result |
|---|---|---|---|---|---|---|---|---|
| 564 | 20 | 25 | 0.001031 | 281 | 258 | 76 | 0 | 500 |
| 565 | 80 | 25 | 0.002115 | 803 | 783 | 304 | 0 | 2000 |
| 567 | 320 | 25 | 0.003899 | 2891 | 2883 | 1216 | 0 | 8000 |
| 570 | 1280 | 25 | 0.079934 | 11243 | 11283 | 4864 | 0 | 32000 |
| 566 | 100 | 625 | 0.005077 | 5065 | 2950 | 1900 | 0 | 62500 |
| 574 | 5120 | 25 | 0.113159 | 44651 | 44883 | 19456 | 0 | 128000 |
| 568 | 400 | 625 | 0.066352 | 26957 | 24548 | 5776 | 0 | 250000 |
| 571 | 1600 | 625 | 0.118409 | 75407 | 73043 | 23104 | 0 | 1000000 |
| 575 | 6400 | 625 | 0.378391 | 269207 | 267023 | 92416 | 0 | 4000000 |
| 569 | 500 | 15625 | 0.087882 | 116825 | 56250 | 47500 | 0 | 7812500 |
| 572 | 2000 | 15625 | 0.404592 | 466445 | 290400 | 144400 | 0 | 31250000 |
| 576 | 8000 | 15625 | 4.823394 | 2140345 | 1934606 | 438976 | 0 | 125000000 |
| 573 | 2500 | 390625 | 4.343946 | 2871625 | 1318750 | 1187500 | 0 | 976562500 |
| 577 | 10000 | 390625 | 10.803828 | 10623725 | 5643500 | 3610000 | 0 | 3906250000 |

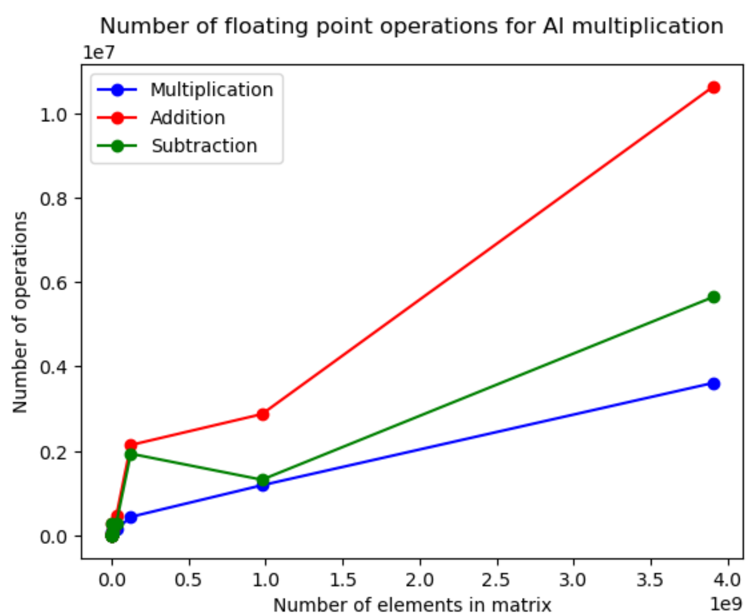
Tabela 3: Szczegółowe wyniki dla algorytmu AI

Poniższe wykresy zawierają pierwsze 7 danych zebranych podczas pomiarów metody AI, ze względu na skale, jak bardzo algorytm AI jest lepszy od "konwencjonalnych metod".

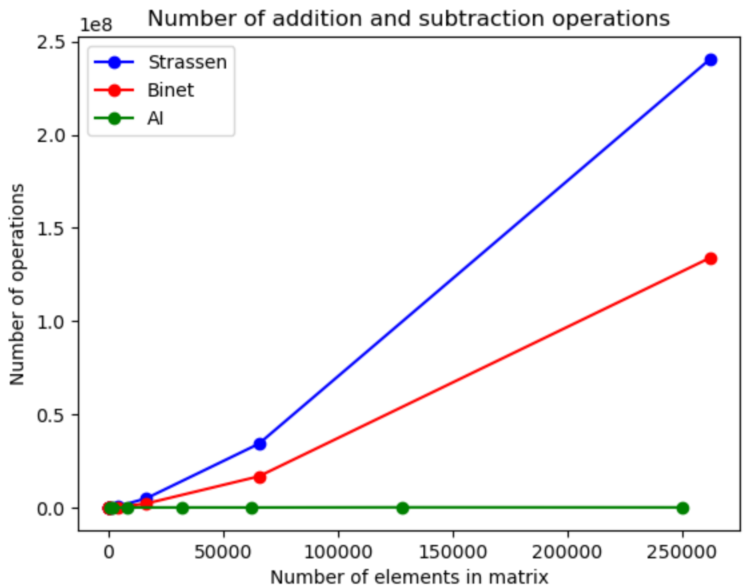
Powyższe wyniki są **za dobre**, mogą sugerować złożoność liniową tego algorytmu. Przyczyną jest prawdopodobnie zbyt słaba jakość testów - większych nie udało się wygenerować przez ograniczenia sprzętowe.



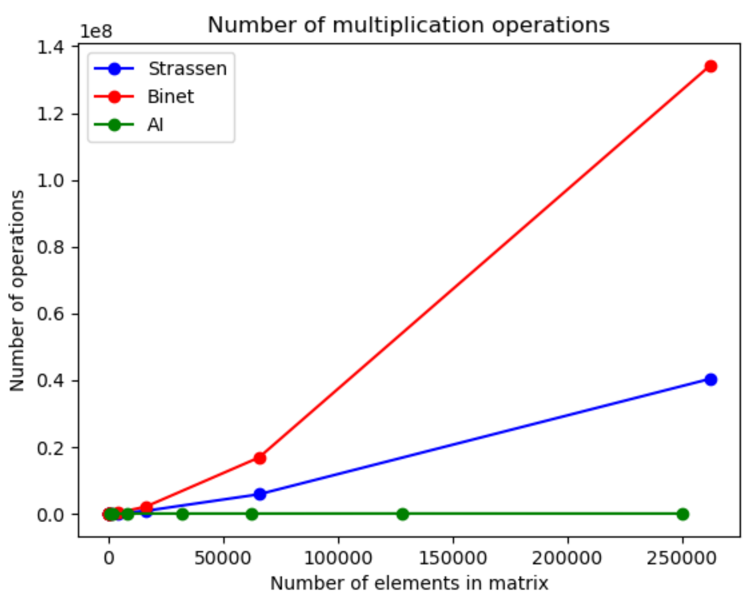

Rysunek 9: Wykres czasu dla macierzy $n \times n$




Rysunek 10: Wykres ilości operacji zmiennoprzecinkowych dla macierzy $n \times n$




Rysunek 11: Wykres ilości dodawań i odejmowań dla końcowej $n$ elementowej macierzy




Rysunek 12: Wykres ilości mnożeń dla końcowej $n$ elementowej macierzy

## 5 Porównanie z MATLABem

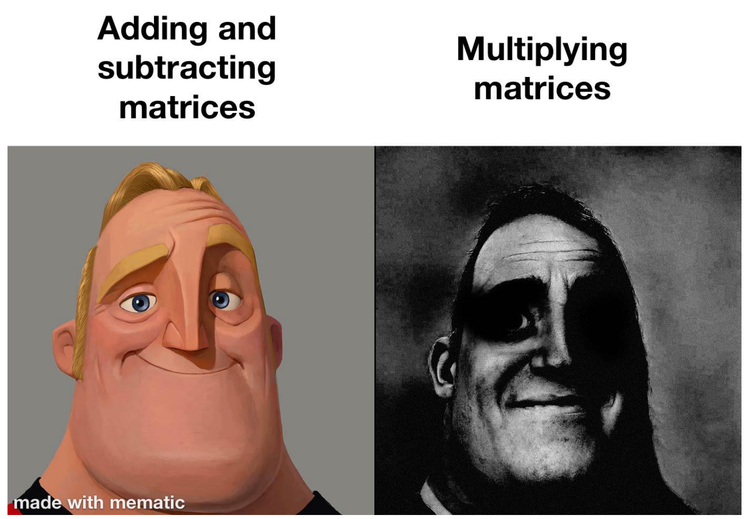
```
MATLAB
64x64: 0.000192s
128x128: 0.000319s
256x256: 0.000354s
512x512: 0.00132s
1024x1024: 0.00799s
2048x2048: 0.058941s

PYTHON
64x64: 0.30934834480285645s
128x128: 2.35215425491333s
256x256: 18.8021457195282s
```

Rysunek 13: Czas mnożnenia macierzy w MATLAB vs Python

Porównaliśmy również czas mnożenia poszczególnych macierzy w programie MATLAB do podstawowego mnożenia za pomocą operatora @ w NumPy. Wyniki mówią same za siebie — MATLAB zdecydowanie lepiej radzi sobie z mnożeniem macierzy.

## 6 Wnioski

- Algorytm Strassena w naszej implementacji nie działa szybciej niż algorytm Binet'a - lecz daje dużą przestrzeń do poprawy.
- Metoda zaproponowana przez sztuczną inteligencje jest dużo szybsza niż konwencjonale metody używane do tej pory. Jednak ograniczenie wielkości macierzy oraz duża ilość podstawień powoduje, że w "codziennym użyciu"jest ona gorsza niż algorytm Strassena czy Binet'a (odpowiednio 7 i 8 podstawień)




Rysunek 14: Mem na polepszenie humoru