

Raport 1

Rekurencyjne mnożenie macierzy

1. Opis pseudokodu

Algorytm Binet'a:

- Jeśli rozmiar macierzy wynosi 2×2 :
 - Pomnóż macierz standardowym mnożeniem macierzy
 - Zwróć wynik
- W przeciwnym przypadku:
 - Podziel macierze na 4 mniejsze części (A_{11} , A_{12} , A_{21} , A_{22})
 - Rekurencyjnie oblicz cztery iloczyny macierzy
 C_{11} = wynik mnożenia A_{11} i B_{11} , a także wynik mnożenia A_{12} i B_{21} , dodane do siebie.
 C_{12} = wynik mnożenia A_{11} i B_{12} , a także wynik mnożenia A_{12} i B_{22} , dodane do siebie.
 C_{21} = wynik mnożenia A_{21} i B_{11} , a także wynik mnożenia A_{22} i B_{21} , dodane do siebie.
 C_{22} = wynik mnożenia A_{21} i B_{12} , a także wynik mnożenia A_{22} i B_{22} , dodane do siebie.
 - Zwróć obliczoną macierz C jako wynik

Algorytm Strassena:

- Jeśli rozmiar macierzy wynosi 2×2 :
 - Pomnóż macierz standardowym mnożeniem macierzy
 - Zwróć wynik
- W przeciwnym przypadku:
 - Podziel macierze na 4 mniejsze części (A_{11} , A_{12} , A_{21} , A_{22})
 - Wykonaj siedem rekurencyjnych wywołań, obliczając siedem częściowych wyników:
 $P1 = \text{StrassenMlt}(A_{11} + A_{22}, B_{11} + B_{22})$
 $P2 = \text{StrassenMlt}(A_{21} + A_{22}, B_{11})$
 $P3 = \text{StrassenMlt}(A_{11}, B_{12} - B_{22})$
 $P4 = \text{StrassenMlt}(A_{22}, B_{21} - B_{11})$
 $P5 = \text{StrassenMlt}(A_{11} + A_{12}, B_{22})$
 $P6 = \text{StrassenMlt}(A_{21} - A_{11}, B_{11} + B_{12})$
 $P7 = \text{StrassenMlt}(A_{12} - A_{22}, B_{21} + B_{22})$
 - Oblicz cztery podmacierze wynikowe C_{11} , C_{12} , C_{21} i C_{22} za pomocą wyrażeń:
 $C_{11} = P1 + P4 - P5 + P7$

$C_{12} = P_3 + P_5$
 $C_{21} = P_2 + P_4$
 $C_{22} = P_1 - P_2 + P_3 + P_6$

- Zwróć obliczoną macierz C jako wynik

2. Kluczowe fragmenty kodu

Generowanie macierzy

```
def random_matrix(matrix_size, min_val, max_val):
    """Return matrix with random floats from [min_val, max_val]"""
    return (max_val - min_val) * np.random.random(matrix_size) + min_val

exp = 4
matrix_size = (exp, exp)

min_val = 0.00000001
max_val = 1

A = random_matrix(matrix_size, min_val, max_val)
B = random_matrix(matrix_size, min_val, max_val)
```

Rysunek 1

Do generowania macierzy została wykorzystana funkcja `random.random()` z modułu `numpy`

Funkcja `random_matrix()` zwraca macierz wypełnioną wartościami z przedziału `[min_val, max_val]`

```
else:
    matrix_size = len(A)

    A11 = A[:matrix_size//2, :matrix_size//2]
    A12 = A[:matrix_size//2, matrix_size//2:]
    A21 = A[matrix_size//2:, :matrix_size//2]
    A22 = A[matrix_size//2:, matrix_size//2:]

    B11 = B[:matrix_size//2, :matrix_size//2]
    B12 = B[:matrix_size//2, matrix_size//2:]
    B21 = B[matrix_size//2:, :matrix_size//2]
    B22 = B[matrix_size//2:, matrix_size//2:]

    C11 = binet_mlt(A11, B11) + binet_mlt(A12, B21)
    C12 = binet_mlt(A11, B12) + binet_mlt(A12, B22)
    C21 = binet_mlt(A21, B11) + binet_mlt(A22, B21)
    C22 = binet_mlt(A21, B12) + binet_mlt(A22, B22)
    operation_count += matrix_size ** 2

    return np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
```

Rysunek 2

Implementacja algorytmu Binet'a

Macierze z modułu numpy dzielone są na mniejsze bloki

Następnie rekurencyjnie obliczana jest macierz C

Funkcja zwraca wynik „składając” macierz C wykorzystując np.hstack() oraz np.vstack()

```
A11 = A[:n//2, :n//2]
B11 = B[:n//2, :n//2]

A12 = A[:n//2, (n//2):]
B12 = B[:n//2, (n//2):]

A21 = A[(n//2):, :n//2]
B21 = B[(n//2):, :n//2]

A22 = A[(n//2):, (n//2):]
B22 = B[(n//2):, (n//2):]

P1 = strassen_mlt((A11 + A22), (B11 + B22))
P2 = strassen_mlt((A21 + A22), B11)
P3 = strassen_mlt(A11, (B12 - B22))
P4 = strassen_mlt(A22, (B21 - B11))
P5 = strassen_mlt((A11 + A12), B22)
P6 = strassen_mlt((A21 - A11), (B11 + B12))
P7 = strassen_mlt((A12 - A22), (B21 + B22))

C11 = (P1 + P4 - P5 + P7)
C12 = (P3 + P5)
C21 = (P2 + P4)
C22 = (P1 - P2 + P3 + P6)

result = np.empty((n, n))
result[:n // 2, :n // 2] = C11
result[:n // 2, n // 2:] = C12
result[n // 2:, :n // 2] = C21
result[n // 2:, n // 2:] = C22

return result
```

Rysunek 3

Implementacja algorytmu Strassena

Analogicznie jak w poprzednim punkcie, macierz dzielona jest na mniejsze bloki

Następnie rekurencyjnie liczone są bloki pomocnicze P według podanych wzorów

Z obliczonych mniejszych bloków składana i zwracana jest macierz wynikowa C.

```

elif len(A) == 2:
    a00 = A[0,0]*B[0,0] + A[0,1]*B[1,0]
    a01 = A[0,0]*B[0,1] + A[0,1]*B[1,1]

    a10 = A[1,0]*B[0,0] + A[1,1]*B[1,0]
    a11 = A[1,0]*B[0,1] + A[1,1]*B[1,1]

    operation_count += 12
    return np.array([[a00,a01],
                     [a10,a11]])

```

Rysunek 4

Obie implementacje w warunku końcowym rekurencji korzystają ze standardowego mnożenia macierzy 2x2

```

for exp in exponents:
    A = random_matrix((exp, exp), min_val, max_val)
    B = random_matrix((exp, exp), min_val, max_val)

    t_start = time()
    std_op = standard_mlt(A, B)[1]
    std_time = time() - t_start

    operation_count = 0
    t_start = time()
    binet_mlt(A, B)
    bin_time = time() - t_start
    bin_op = operation_count

    operation_count = 0
    t_start = time()
    strassen_mlt(A, B)
    str_time = time() - t_start
    str_op = operation_count

    res_times.append((std_time, bin_time, str_time))
    res_operations.append((std_op, bin_op, str_op))

```

Rysunek 5

W celu przetestowania szybkości działania implementacji zostały obliczone szybkość wykonania oraz ilość operacji dla poszczególnych wykładników

AI

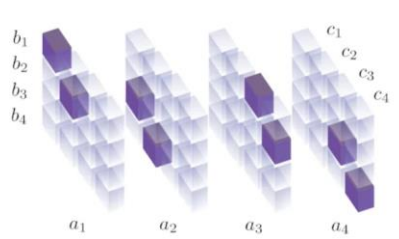
```

filename = "factorizations_r.npz"
with open(filename, 'rb') as f:
    factorizations = dict(np.load(f, allow_pickle=True))

for key in factorizations:
    u, v, w = factorizations[key]
    rank = u.shape[-1]
    assert rank == v.shape[-1] and rank == w.shape[-1]
    print(f'{key}: rank={u.shape[-1]}')

```

Rysunek 6

$$\begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \cdot \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}$$


$$\begin{aligned}
 m_1 &= (a_1 + a_4)(b_1 + b_4) \\
 m_2 &= (a_3 + a_4)b_1 \\
 m_3 &= a_1(b_2 - b_4) \\
 m_4 &= a_4(b_3 - b_1) \\
 m_5 &= (a_1 + a_2)b_4 \\
 m_6 &= (a_3 - a_1)(b_1 + b_2) \\
 m_7 &= (a_2 - a_4)(b_3 + b_4) \\
 c_1 &= m_1 + m_4 - m_5 + m_7 \\
 c_2 &= m_3 + m_5 \\
 c_3 &= m_2 + m_4 \\
 c_4 &= m_1 - m_2 + m_3 + m_6
 \end{aligned}$$

$$\mathbf{u} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & -1 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$\mathbf{w} = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Rysunek 7

```

for i in range(mul_len):
    A_elements = t1[:, i]
    B_elements = t2[:, i]

    A_factor = np.zeros((n,m), dtype = float)
    for j in range(len(A_elements)):
        if A_elements[j] != 0:
            x,y = np.unravel_index(j, (4,5))
            A_factor += A[x*n:(x+1)*n, y*m: (y+1)*m]*A_elements[j]

            # plus_count += len(A_factor[0])*len(A_factor)
            operation_count += len(A_factor)*len(A_factor[0])

    B_factor = np.zeros((k,k), dtype = float)
    for j in range(len(B_elements)):
        if B_elements[j] != 0:
            x,y = np.unravel_index(j, (5,5))
            B_factor += B[x*k:(x+1)*k, y*k: (y+1)*k]*B_elements[j]

            # plus_count += len(B_factor)**2
            operation_count += len(B_factor)**2

    m_res = ai_mlt(A_factor, B_factor, factor)
    #operation_count += 1
    mtab[i] = m_res

```

Rysunek 8

```

res_matrix = np.zeros((n*4,m*5), dtype = float)

xind = 0
yind = 0
for i in range(t3.shape[0]):
    x,y = np.unravel_index(i, (4,5))
    res_fragment = np.zeros((n,m), dtype = float)
    C_elements = t3[i, :]
    for j in range(len(C_elements)):
        if C_elements[j] != 0:
            res_fragment += mtab[j]*C_elements[j]

            # plus_count += n*m

    if xind >= 4:
        yind+=1
        xind=0

    res_matrix[xind*n:(xind+1)*n, yind*m: (yind+1)*m] = res_fragment

    xind+=1

return res_matrix

```

Rysunek 9

```

for x in range(4):
    for y in range(5):
        mb = r2[x*n:(x+1)*n, y*m: (y+1)*m]
        ma = r1[x*n:(x+1)*n, y*m: (y+1)*m]
        # print(ma)
        # print("-----")
        # print(mb)
        print(np.allclose(ma, mb), x, y)
        if not np.allclose(ma, mb):
            print(exists(r1, r2, ma))
        # print("\n\n")

```

✓ 0.0s

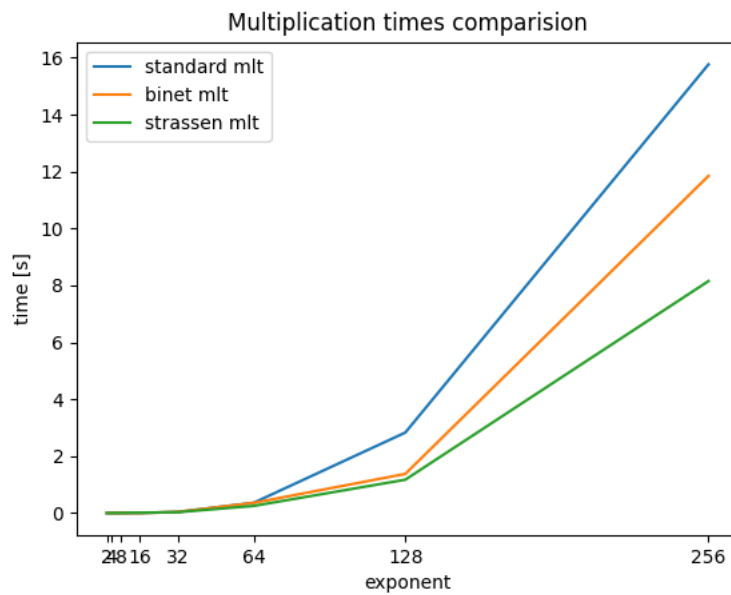
```

True 0 0
True 0 1
True 0 2
True 0 3
True 0 4
True 1 0
True 1 1
True 1 2
True 1 3
True 1 4
True 2 0
True 2 1
True 2 2
True 2 3
True 2 4
True 3 0
True 3 1
True 3 2
- - -

```

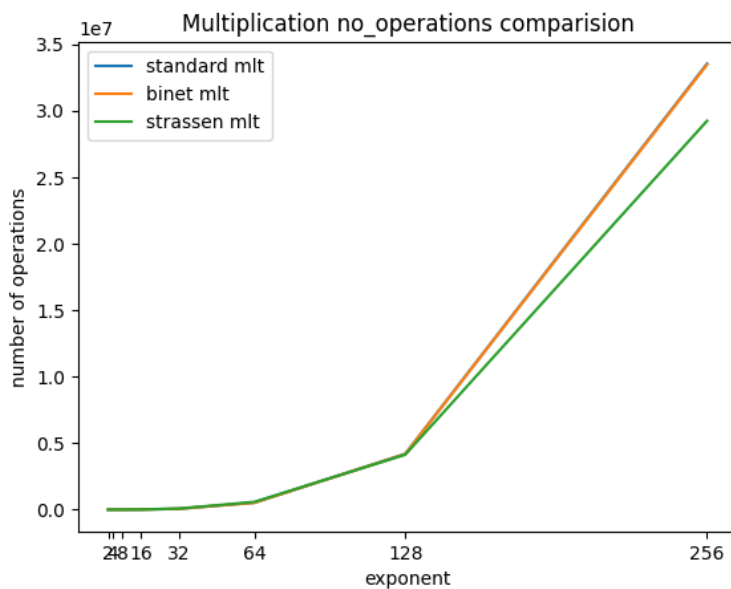
Rysunek 10

3. Wykresy



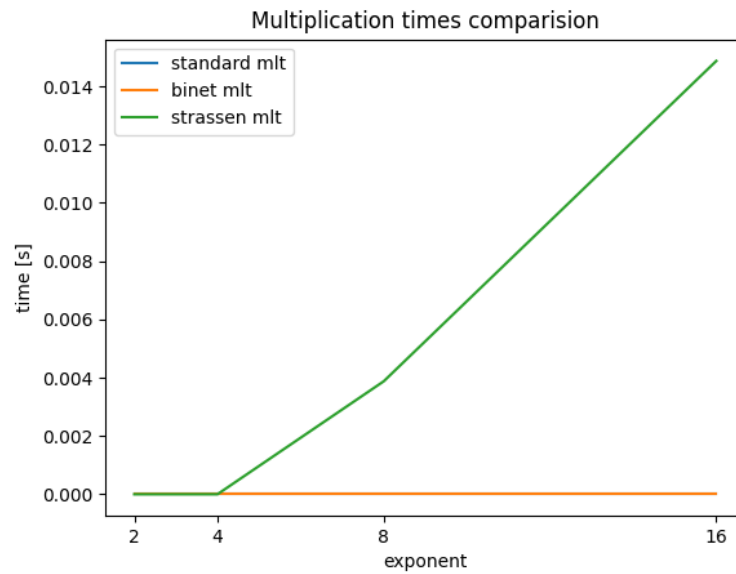
Wykres 1

Wykres ilustrujący porównanie szybkości wykonania poszczególnych algorytmów.
(Linie na wykresie zostały dodane w celu poprawy czytelności obrazu)

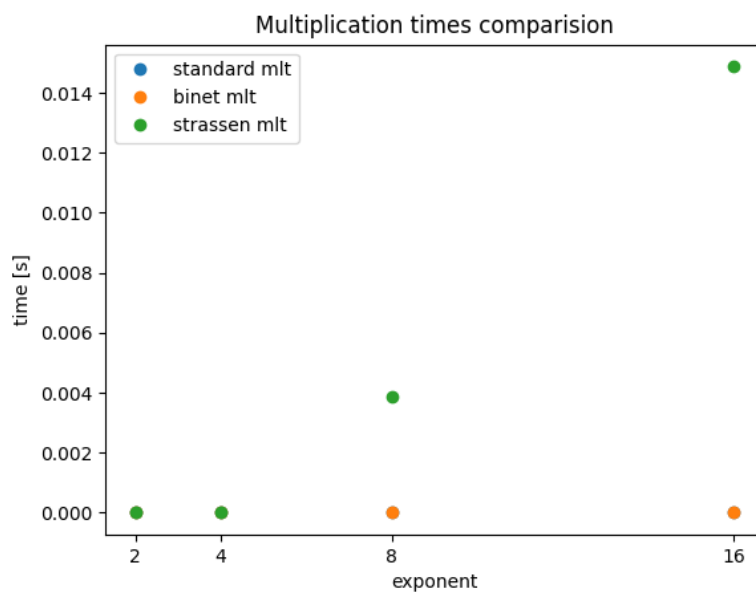


Wykres 2

Wykres ilustrujący ilość wykonanych operacji zmiennoprzecinkowych dla poszczególnych algorytmów. Wyniki dla standardowego mnożenia macierzy i mnożenia metodą Binte'a są bardzo podobne i nie widać różnicy liczbie wykonanych operacji dla takiej skali na wykresie.
(Linie na wykresie zostały dodane w celu poprawy czytelności obrazu)

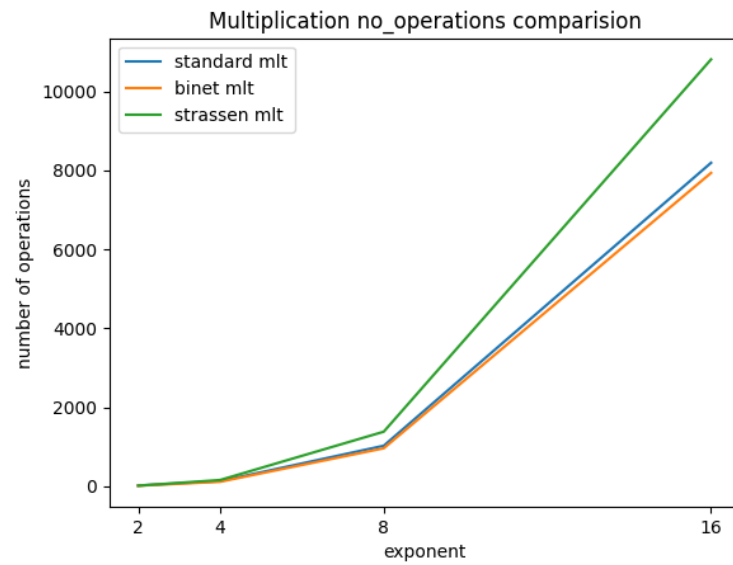


Wykres 3

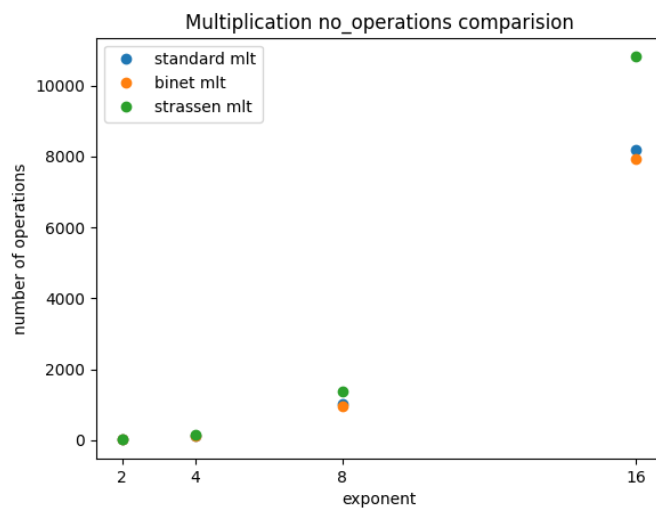


Wykres 4

Wykresy ilustrujące szybkość działania poszczególnych algorytmów przy niskich wartościach wykładnika
(Na drugim wykresie poglądowo zostały dodane linie łączące punkty)

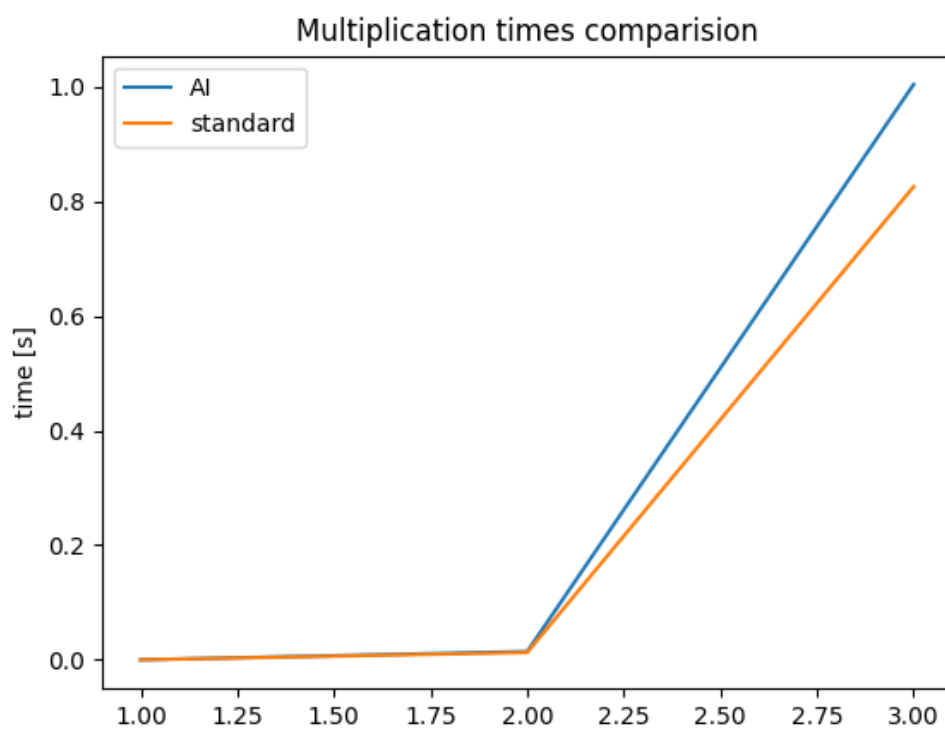


Wykres 5

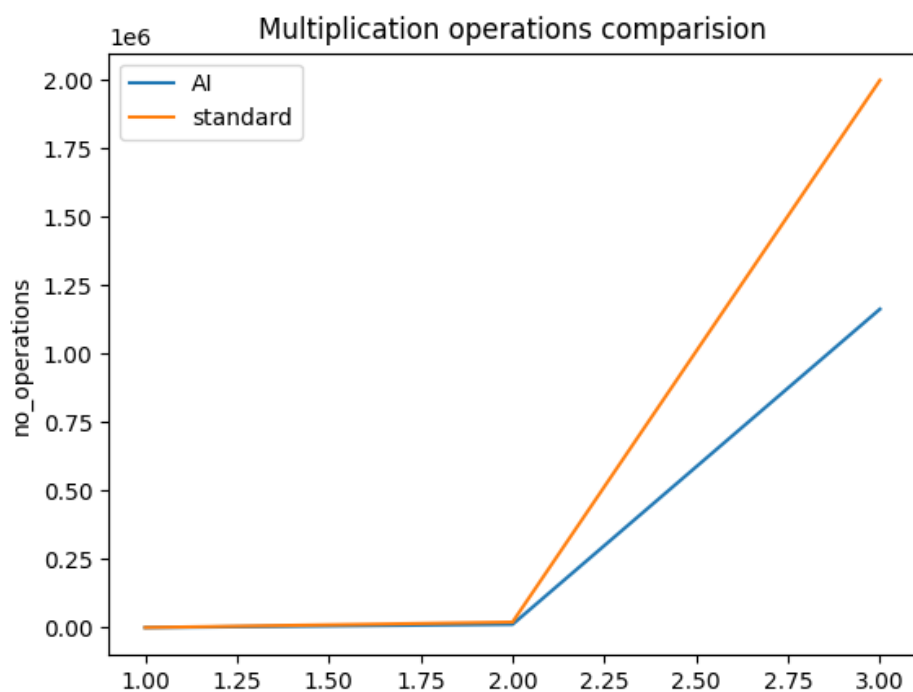


Wykres 6

Wykresy ilustrujące ilość wykonanych operacji zmiennoprzecinkowych poszczególnych algorytmów przy niskich wartościach wykładnika
(Na drugim wykresie poglądowo zostały dodane linie łączące punkty)



Wykres 7



Wykres 8

Wykresy ilustrujące porównanie czasów działania oraz ilości wykonanych operacji algorytmu AI ze standardowym mnożeniem macierzy.

4. Ocena złożoności obliczeniowej

5. Porównanie z MATLAB