Akademia Górniczo-Hutnicza

Wydział Informatyki

Algorytmy macierzowe

Ćwiczenie 1

Mnożenie macierzy metodami rekurencyjnymi

Jakub Szewczyk i Albert Arnautov

Kraków, 21 października 2025

Spis treści

1 Wstęp

Celem ćwiczenia było zaimplementowanie i porównanie metod mnożenia macierzy metodami rekurencyjnymi: metodą Binet'a, metodą Strassena oraz metoda "AI"

2 Pseudokod

Poniżej zostały przedstawione pseudokody poszczególnych metod

2.1 Metoda Bineta

Algorytm 1 Metoda Bineta - pseudokod (cz. 1)

```
1:  procedure BINET(A, B)
2:      n ← rows(A)
3:      m ← cols(A)
4:      p ← cols(B)

5:      if n = 1 then
6:          C ← zero matrix of size 1 × p
7:          for col = 0 to p − 1 do
8:              for i = 0 to m − 1 do
9:                  C[0][col] ← C[0][col] + A[0][i] × B[i][col]
10:             end for
11:         end for
12:         return C
13:     end if

14:     if p = 1 then
15:         C ← zero matrix of size n × 1
16:         for row = 0 to n − 1 do
17:             for i = 0 to m − 1 do
18:                 C[row][0] ← C[row][0] + A[row][i] × B[i][0]
19:             end for
20:         end for
21:         return C
22:     end if

23:     mid_n ← ⌊n/2⌋
24:     mid_p ← ⌊p/2⌋
25:     mid_m ← ⌊m/2⌋
26:     A11 ← A[0 : mid_n − 1, 0 : mid_m − 1]
27:     A12 ← A[0 : mid_n − 1, mid_m : m − 1]
28:     A21 ← A[mid_n : n − 1, 0 : mid_m − 1]
29:     A22 ← A[mid_n : n − 1, mid_m : m − 1]

30:     B11 ← B[0 : mid_m − 1, 0 : mid_p − 1]
31:     B12 ← B[0 : mid_m − 1, mid_p : p − 1]
32:     B21 ← B[mid_m : m − 1, 0 : mid_p − 1]
33:     B22 ← B[mid_m : m − 1, mid_p : p − 1]
34:     C11 ← MatAdd(Binet(A11, B11), Binet(A12, B21))
35:     C12 ← MatAdd(Binet(A11, B12), Binet(A12, B22))
36:     C21 ← MatAdd(Binet(A21, B11), Binet(A22, B21))
37:     C22 ← MatAdd(Binet(A21, B12), Binet(A22, B22))
```

Algorytm 2 Metoda Bineta - pseudokod (cz. 2)

```
    C ← zero matrix of size n × p
    C[0 : mid_n − 1, 0 : mid_p − 1] ← C11
    C[0 : mid_n − 1, mid_p : p − 1] ← C12
    C[mid_n : n − 1, 0 : mid_p − 1] ← C21
    C[mid_n : n − 1, mid_p : p − 1] ← C22
    return C
end procedure
```

2.2 Metoda Strassena

Algorytm 3 Metoda Strassena - pseudokod (cz. 1)

```
procedure STRASSEN(A, B)
    n ← length(A)

    if n = 1 then
        return [[A[0][0] × B[0][0]]]
    end if

    if n = 2 then
        return standard_matrix_multiply(A, B)
    end if

    C ← create empty matrix of size n × n
    k ← ⌊n/2⌋

    if n mod 2 = 0 then
        A11 ← A[0 : k, 0 : k]
        A12 ← A[0 : k, k : n]
        A21 ← A[k : n, 0 : k]
        A22 ← A[k : n, k : n]

        B11 ← B[0 : k, 0 : k]
        B12 ← B[0 : k, k : n]
        B21 ← B[k : n, 0 : k]
        B22 ← B[k : n, k : n]

        M1 ← Strassen(Add(A11, A22), Add(B11, B22))
        M2 ← Strassen(Add(A21, A22), B11)
        M3 ← Strassen(A11, Subtract(B12, B22))
        M4 ← Strassen(A22, Subtract(B21, B11))
        M5 ← Strassen(Add(A11, A12), B22)
        M6 ← Strassen(Subtract(A21, A11), Add(B11, B12))
        M7 ← Strassen(Subtract(A12, A22), Add(B21, B22))

        C11 ← Add(Subtract(Add(M1, M4), M5), M7)
        C12 ← Add(M3, M5)
        C21 ← Add(M2, M4)
        C22 ← Add(Subtract(Add(M1, M3), M2), M6)
```

Algorytm 4 Metoda Strassena - pseudokod (cz. 2)

```
        for i = 0 to k − 1 do
            for j = 0 to k − 1 do
                C[i][j] ← C11[i][j]
                C[i][j + k] ← C12[i][j]
                C[i + k][j] ← C21[i][j]
                C[i + k][j + k] ← C22[i][j]
            end for
        end for
    else
        right_vector ← A[0 : n − 1, :] × B[:, n − 1]
        down_vector ← A[n − 1, :] × B[:, 0 : n − 1]
        corner_element ← A[n − 1, :] × B[:, n − 1]

        C_sub ← Strassen(A[0 : n − 1, 0 : n − 1], B[0 : n − 1, 0 : n − 1])

        last_col ← A[:, n − 1]
        last_row ← B[n − 1, :]

        for i = 0 to n − 2 do
            for j = 0 to n − 2 do
                C_sub[i][j] ← C_sub[i][j] + last_col[i] × last_row[j]
                C[i][j] ← C_sub[i][j]
            end for
        end for

        for i = 0 to n − 2 do
            C[i][n − 1] ← right_vector[i]
        end for

        for j = 0 to n − 2 do
            C[n − 1][j] ← down_vector[j]
        end for

        C[n − 1][n − 1] ← corner_element
    end if

    return C
end procedure
```

3 Ważne fragmenty kodu

3.1 Generowanie losowej macierzy rozmiaru n

```python
from random import uniform

def randomize_matrix(n):
    return [[uniform(1e-8, 1) for _ in range(n)] for _ in range(n)]
```

3.2 Implementacja metody Bineta

```python
def binet(A, B):
    global counter_add, counter_mul

    n = A.shape[0]
    m = A.shape[1]
    p = B.shape[1]

    if n == 1:
        C = np.zeros((n, p))
        for col in range(p):
            for i in range(m):
                counter_add += 1
                counter_mul += 1
                C[0][col] += A[0][i] * B[i][col]
        return C
    if p == 1:
        C = np.zeros((n, p))
        for row in range(n):
            for i in range(m):
                counter_add += 1
                counter_mul += 1
                C[row][0] += A[row][i] * B[i][0]
        return C

    mid_n = n // 2
    mid_p = p // 2
    mid_m = m // 2
    A11 = A[:mid_n, :mid_m]
    A12 = A[:mid_n, mid_m:]
    A21 = A[mid_n:, :mid_m]
    A22 = A[mid_n:, mid_m:]

    B11 = B[:mid_m, :mid_p]
    B12 = B[:mid_m, mid_p:]
    B21 = B[mid_m:, :mid_p]
    B22 = B[mid_m:, mid_p:]

    C11 = mat_add(binet(A11, B11), binet(A12, B21))
    C12 = mat_add(binet(A11, B12), binet(A12, B22))
    C21 = mat_add(binet(A21, B11), binet(A22, B21))
    C22 = mat_add(binet(A21, B12), binet(A22, B22))

    C = np.zeros((n, p))
    C[:mid_n, :mid_p] = C11
    C[:mid_n, mid_p:] = C12
    C[mid_n:, :mid_p] = C21
    C[mid_n:, mid_p:] = C22

    return C
```

3.3 Implementacja metody Strassena

```python
def strassen(A, B):
    global counter_add, counter_mul, counter_sub

    n = len(A)

    if n == 1:
        counter_mul += 1
        return [[A[0][0] * B[0][0]]]

    if n == 2:
        counter_add += 4
        counter_mul += 8
        return (np.array(A) @ np.array(B)).tolist()

    C = [[None for _ in range(n)] for _ in range(n)]
    k = n // 2

    if n % 2 == 0:
        A11 = [row[:k] for row in A[:k]]
        A12 = [row[k:] for row in A[:k]]
        A21 = [row[:k] for row in A[k:]]
        A22 = [row[k:] for row in A[k:]]

        B11 = [row[:k] for row in B[:k]]
        B12 = [row[k:] for row in B[:k]]
        B21 = [row[:k] for row in B[k:]]
        B22 = [row[k:] for row in B[k:]]

        M1 = strassen(add(A11, A22), add(B11, B22))
        M2 = strassen(add(A21, A22), B11)
        M3 = strassen(A11, sub(B12, B22))
        M4 = strassen(A22, sub(B21, B11))
        M5 = strassen(add(A11, A12), B22)
        M6 = strassen(sub(A21, A11), add(B11, B12))
        M7 = strassen(sub(A12, A22), add(B21, B22))

        C11 = add(sub(add(M1, M4), M5), M7)
        C12 = add(M3, M5)
        C21 = add(M2, M4)
        C22 = add(sub(add(M1, M3), M2), M6)

        for i in range(k):
            for j in range(k):
                C[i][j] = C11[i][j]
                C[i][j + k] = C12[i][j]
                C[i + k][j] = C21[i][j]
                C[i + k][j + k] = C22[i][j]

    else:
        A = np.asarray(A)
        B = np.asarray(B)

        right_vector = (A[:-1, :] @ B[:, -1]).tolist()
        counter_mul += (n - 1) * n
        counter_add += (n - 1) * (n - 1)

        down_vector = (A[-1, :] @ B[:, :-1]).tolist()
        counter_mul += (n - 1) * n
        counter_add += (n - 1) * (n - 1)
```

```
        corner_element = (A[-1, :] @ B[:, -1]).item()
        counter_mul += n
        counter_add += (n - 1)

        C_sub = strassen(A[:-1, :-1].tolist(), B[:-1, :-1].tolist())

        last_col = A[:, -1]
        last_row = B[-1, :]

        for i in range(n - 1):
            for j in range(n - 1):
                C_sub[i][j] += last_col[i] * last_row[j]
                counter_add += 1
                counter_mul += 1
                C[i][j] = C_sub[i][j]

        for i in range(n - 1):
            C[i][n - 1] = right_vector[i]

        for j in range(n - 1):
            C[n - 1][j] = down_vector[j]

        C[-1][-1] = corner_element

    return C
```

3.4 Implementacja "metody AI"

```
def strassen_5x5(A, B):
    a11, a12, a13, a14, a15 = A[0, :]
    a21, a22, a23, a24, a25 = A[1, :]
    a31, a32, a33, a34, a35 = A[2, :]
    a41, a42, a43, a44, a45 = A[3, :]

    b11, b12, b13, b14, b15 = B[0, :]
    b21, b22, b23, b24, b25 = B[1, :]
    b31, b32, b33, b34, b35 = B[2, :]
    b41, b42, b43, b44, b45 = B[3, :]
    b51, b52, b53, b54, b55 = B[4, :]

    h1 = a32 * (-b21 - b25 - b31)
    h2 = (a22 + a25 - a35) * (-b25 - b51)
    h3 = (-a31 - a41 + a42) * (-b11 + b25)
    h4 = (a12 + a14 + a34) * (-b25 - b41)
    h5 = (a15 + a22 + a25) * (-b24 + b51)
    h6 = (-a22 - a25 - a45) * (b23 + b51)
    h7 = (-a11 + a41 - a42) * (b11 + b24)
    h8 = (a32 - a33 - a43) * (-b23 + b31)
    h9 = (-a12 - a14 + a44) * (b23 + b41)
    h10 = (a22 + a25) * b51
    h11 = (-a21 - a41 + a42) * (-b11 + b22)
    h12 = (a41 - a42) * b11
    h13 = (a12 + a14 + a24) * (b22 + b41)
    h14 = (a13 - a32 + a33) * (b24 + b31)
    h15 = (-a12 - a14) * b41
    h16 = (-a32 + a33) * b31
    h17 = (a12 + a14 - a21 + a22 - a23 + a24 - a32 + a33 - a41 + a42) * b22
    h18 = a21 * (b11 + b12 + b52)
    h19 = -a23 * (b31 + b32 + b52)
```

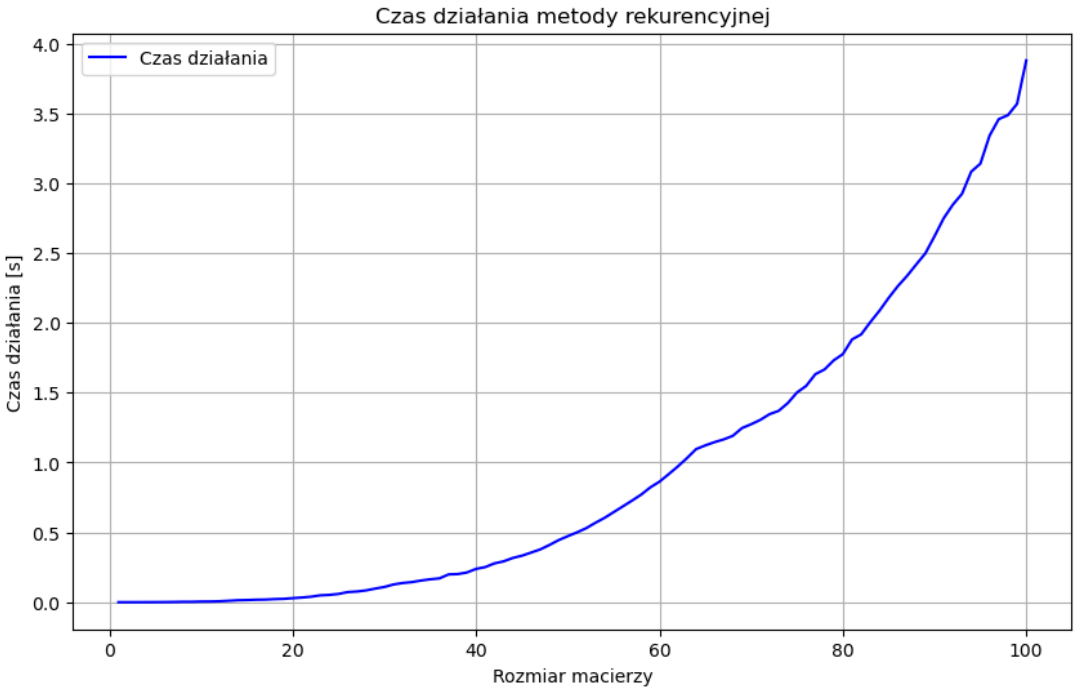
```
    h20 = (-a15 + a21 + a23 - a25) * (-b11 - b12 + b14 - b52)
    h21 = (a21 + a23 - a25) * b52
    h22 = (a13 - a14 - a24) * (b11 + b12 - b14 - b31 - b32 + b34 + b44)
    h23 = a13 * (-b31 + b34 + b44)
    h24 = a15 * (-b44 - b51 + b54)
    h25 = -a11 * (b11 - b14)
    h26 = (-a13 + a14 + a15) * b44
    h27 = (a13 - a31 + a33) * (b11 - b14 + b15 + b35)
    h28 = -a34 * (-b35 - b41 - b45)
    h29 = a31 * (b11 + b15 + b35)
    h30 = (a31 - a33 + a34) * b35
    h31 = (-a14 - a15 - a34) * (-b44 - b51 + b54 - b55)
    h32 = (a21 + a41 + a44) * (b13 - b41 - b42 - b43)
    h33 = a43 * (-b31 - b33)
    h34 = a44 * (-b13 + b41 + b43)
    h35 = -a45 * (b13 + b51 + b53)
    h36 = (a23 - a25 - a45) * (b31 + b32 + b33 + b52)
    h37 = (-a41 - a44 + a45) * b13
    h38 = (-a23 - a31 + a33 - a34) * (b35 + b41 + b42 + b45)
    h39 = (-a31 - a41 - a44 + a45) * (b13 + b51 + b53 + b55)
    h40 = (-a13 + a14 + a15 - a44) * (-b31 - b33 + b34 + b44)
    h41 = (-a11 + a41 - a45) * (b13 + b31 + b33 - b34 + b51 + b53 - b54)
    h42 = (-a21 + a25 - a35) * (-b11 - b12 - b15 + b41 + b42 + b45 - b52)
    h43 = a24 * (b41 + b42)
    h44 = (a23 + a32 - a33) * (b22 - b31)
    h45 = (-a33 + a34 - a43) * (b35 + b41 + b43 + b45 + b51 + b53 + b55)
    h46 = -a35 * (-b51 - b55)
    h47 = (a21 - a25 - a31 + a35) * (b11 + b12 + b15 - b41 - b42 - b45)
    h48 = (-a23 + a33) * (b22 + b32 + b35 + b41 + b42 + b45)
    h49 = (-a11 - a13 + a14 + a15 - a21 - a23 + a24 + a25) * (-b11 - b12 + b14)
    h50 = (-a14 - a24) * (b22 - b31 - b32 + b34 - b42 + b44)
    h51 = a22 * (b21 + b22 - b51)
    h52 = a42 * (b11 + b21 + b23)
    h53 = -a12 * (-b21 + b24 + b41)
    h54 = (a12 + a14 - a22 - a25 - a32 + a33 - a42 + a43 - a44 - a45) * b23
    h55 = (a14 - a44) * (-b23 + b31 + b33 - b34 + b43 - b44)
    h56 = (a11 - a15 - a41 + a45) * (b31 + b33 - b34 + b51 + b53 - b54)
    h57 = (-a31 - a41) * (-b13 - b15 - b25 - b51 - b53 - b55)
    h58 = (-a14 - a15 - a34 - a35) * (-b51 + b54 - b55)
    h59 = (-a33 + a34 - a43 + a44) * (b41 + b43 + b45 + b51 + b53 + b55)
    h60 = (a25 + a45) * (b23 - b31 - b32 - b33 - b52 - b53)
    h61 = (a14 + a34) * (b11 - b14 + b15 - b25 - b44 + b45 - b51 + b54 - b55)
    h62 = (a21 + a41) * (b12 + b13 + b22 - b41 - b42 - b43)
    h63 = (-a33 - a43) * (-b23 - b33 - b35 - b41 - b43 - b45)
    h64 = (a11 - a13 - a14 + a31 - a33 - a34) * (b11 - b14 + b15)
    h65 = (-a11 + a41) * (-b13 + b14 + b24 - b51 - b53 + b54)
    h66 = (a11 - a12 + a13 - a15 - a22 - a25 - a32 + a33 - a41 + a42) * b24
    h67 = (a25 - a35) * (b11 + b12 + b15 - b25 - b41 - b42 - b45 + b52 + b55)
    h68 = (a11 + a13 - a14 - a15 - a41 - a43 + a44 + a45) * (-b31 - b33 + b34)
    h69 = (-a13 + a14 - a23 + a24) * (-b24 - b31 - b32 + b34 - b52 + b54)
    h70 = (a23 - a25 + a43 - a45) * (-b31 - b32 - b33)
    h71 = (-a31 + a33 - a34 + a35 - a41 + a43 - a44 + a45) * (-b51 - b53 - b55)
    h72 = (-a21 - a24 - a41 - a44) * (b41 + b42 + b43)
    h73 = (a13 - a14 - a15 + a23 - a24 - a25) * (b11 + b12 - b14 + b24 + b52 - b54)
    h74 = (a21 - a23 + a24 - a31 + a33 - a34) * (b41 + b42 + b45)
    h75 = -(a12 + a14 - a22 - a25 - a31 + a32 + a34 + a35 - a41 + a42) * b25
    h76 = (a13 + a33) * (-b11 + b14 - b15 + b24 + b34 - b35)

    c11 = -h10 + h12 + h14 - h15 - h16 + h53 + h5 - h66 - h7
    c21 = h10 + h11 - h12 + h13 + h15 + h16 - h17 - h44 + h51
    c31 = h10 - h12 + h15 + h16 - h1 + h2 + h3 - h4 + h75
    c41 = -h10 + h12 - h15 - h16 + h52 + h54 - h6 - h8 + h9
```

```
94
95      c12 = h13 + h15 + h20 + h21 - h22 + h23 + h25 - h43 + h49 + h50
96      c22 = -h11 + h12 - h13 - h15 - h16 + h17 + h18 - h19 - h21 + h43 + h44
97      c32 = -h16 - h19 - h21 - h28 - h29 - h38 + h42 + h44 - h47 + h48
98      c42 = h11 - h12 - h18 + h21 - h32 + h33 - h34 - h36 + h62 - h70
99
100     c13 = h15 + h23 + h24 + h34 - h37 + h40 - h41 + h55 - h56 - h9
101     c23 = -h10 + h19 + h32 + h35 + h36 + h37 - h43 - h60 - h6 - h72
102     c33 = -h16 - h28 + h33 + h37 - h39 + h45 - h46 + h63 - h71 - h8
103     c43 = h10 + h15 + h16 - h33 + h34 - h35 - h37 - h54 + h6 + h8 - h9
104
105     c14 = -h10 + h12 + h14 - h16 + h23 + h24 + h25 + h26 + h5 - h66 - h7
106     c24 = h10 + h18 - h19 + h20 - h22 - h24 - h26 - h5 - h69 + h73
107     c34 = -h14 + h16 - h23 - h26 + h27 + h29 + h31 + h46 - h58 + h76
108     c44 = h12 + h25 + h26 - h33 - h35 - h40 + h41 + h65 - h68 - h7
109
110     c15 = h15 + h24 + h25 + h27 - h28 + h30 + h31 - h4 + h61 + h64
111     c25 = -h10 - h18 - h2 - h30 - h38 + h42 - h43 + h46 + h67 + h74
112     c35 = -h10 + h12 - h15 + h28 + h29 - h2 - h30 - h3 + h46 + h4 - h75
113     c45 = -h12 - h29 + h30 - h34 + h35 + h39 + h3 - h45 + h57 + h59
114
115     C = np.array([
116     [c11, c12, c13, c14, c15],
117     [c21, c22, c23, c24, c25],
118     [c31, c32, c33, c34, c35],
119     [c41, c42, c43, c44, c45]
120     ])
121
122     return C
```

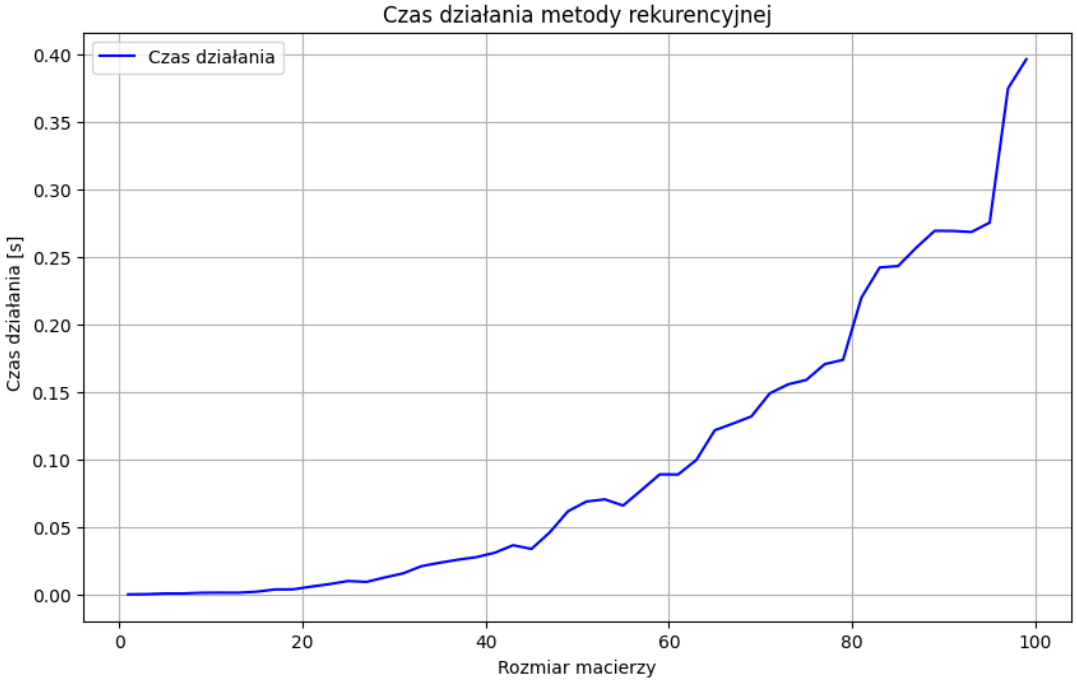
4 Wykresy i porównanie metod

W tym rozdziale przedstawiono wykresy czasów działania zaimplementowanych metod rekurencyjnych, liczbę operacji zmiennoprzecinkowych oraz użycie pamięci w zależności od rozmiaru macierzy.

4.1 Analiza czasów działania algorytmów rekurencyjnych



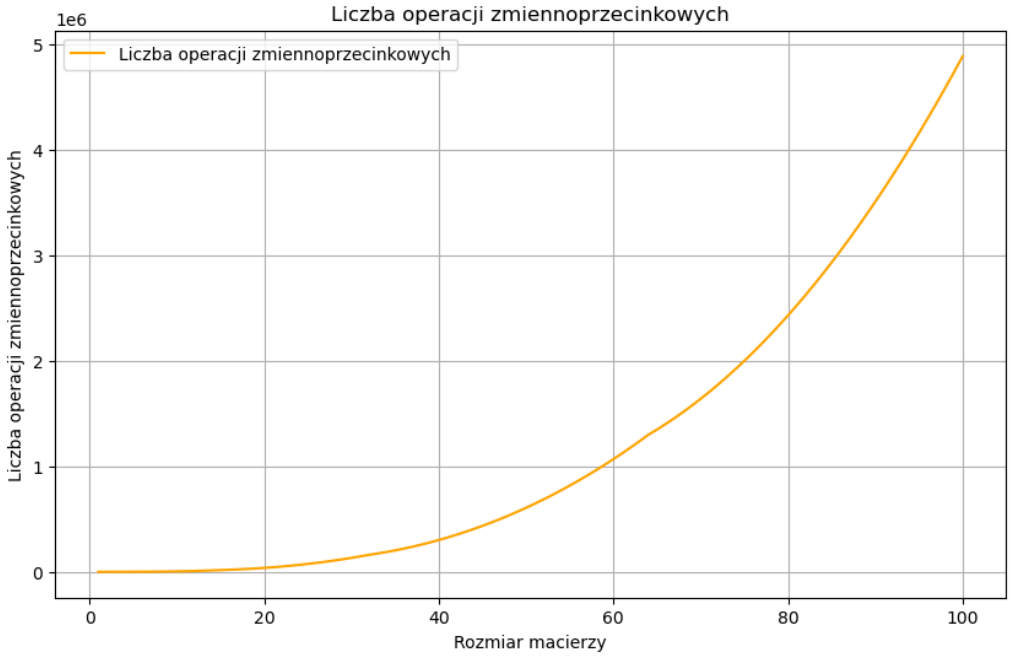
Rysunek 1: Wykres czasu działania w zależności od rozmiaru macierzy - metoda Bineta



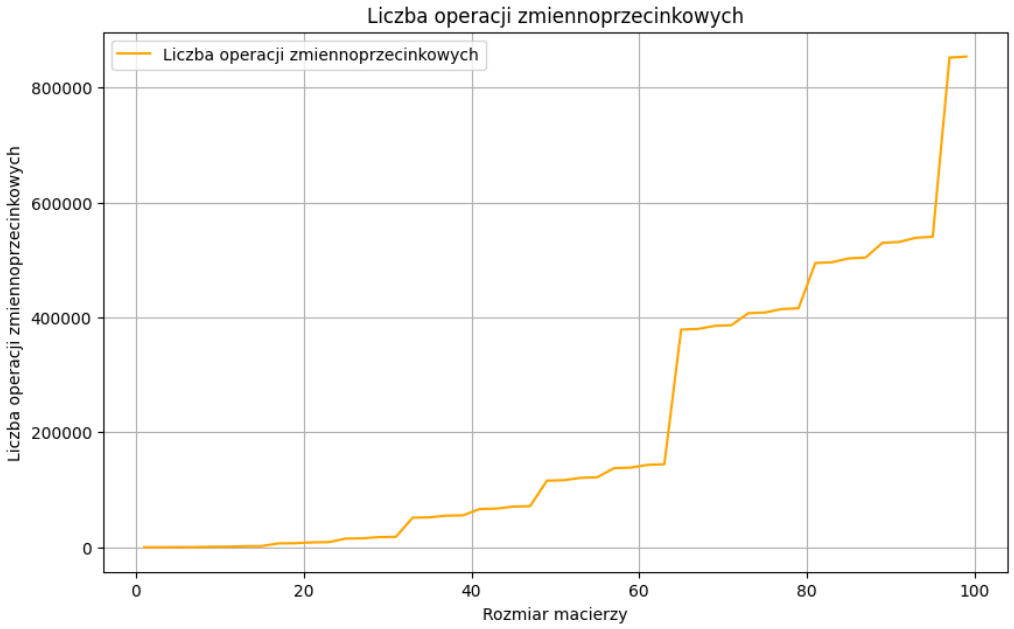
Rysunek 2: Wykres czasu działania w zależności od rozmiaru macierzy - metoda Strassena

Na powyższych wykresach widać, że metoda Strassena działa znacznie szybciej niż metoda Bineta. Widać jednak również, że są to algorytmy o podobnej złożoności obliczeniowej - metoda Strassena teoretycznie osiąga złożoność $O(n^{2.8074})$, a metoda Bineta $O(n^3)$

4.2 Analiza liczby operacji zmiennoprzecinkowych



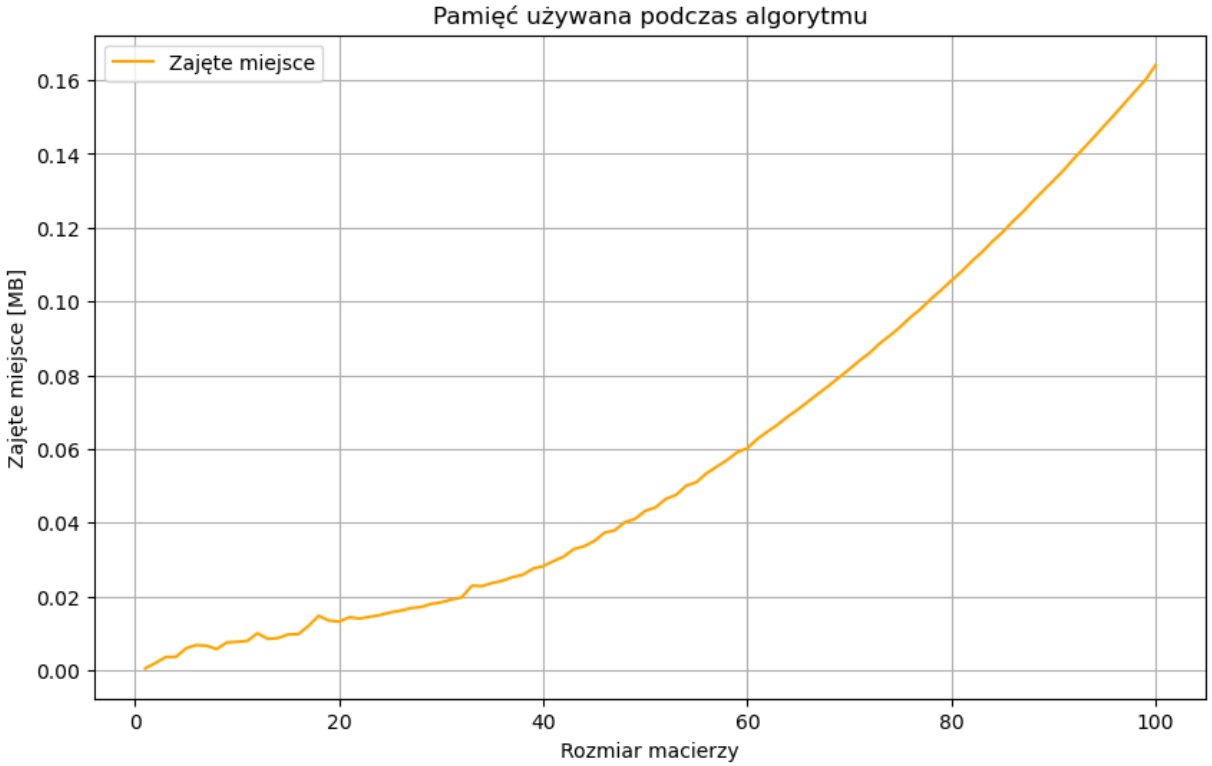
Rysunek 3: Wykres liczby operacji zmiennoprzecinkowych w zależności od rozmiaru macierzy - metoda Bineta



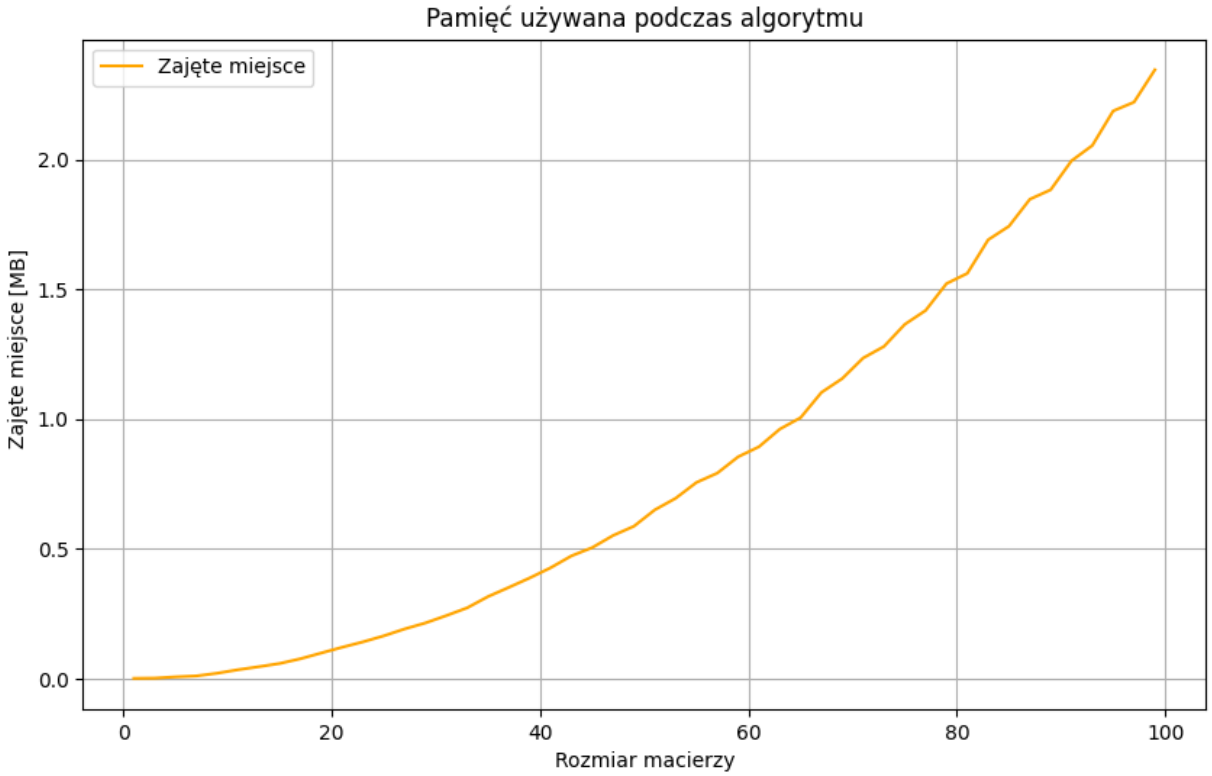
Rysunek 4: Wykres liczby operacji zmiennoprzecinkowych w zależności od rozmiaru macierzy - metoda Strassena

Porównując oba powyższe wykresy można zauważyć, że na pierwszym wykresie (metoda Bineta) krzywa ma bardziej "gładki" kształt od tej widocznej na drugim wykresie (metoda Strassena). Metoda Bineta dla tych samych rozmiarów macierzy wykonuje także dużo więcej operacji zmiennoprzecinkowych w porównaniu z metodą Strassena.

4.3 Analiza użycia pamięci



Rysunek 5: Wykres użycia pamięci w zależności od rozmiaru macierzy - metoda Bineta



Rysunek 6: Wykres użycia pamięci w zależności od rozmiaru macierzy - metoda Strassena

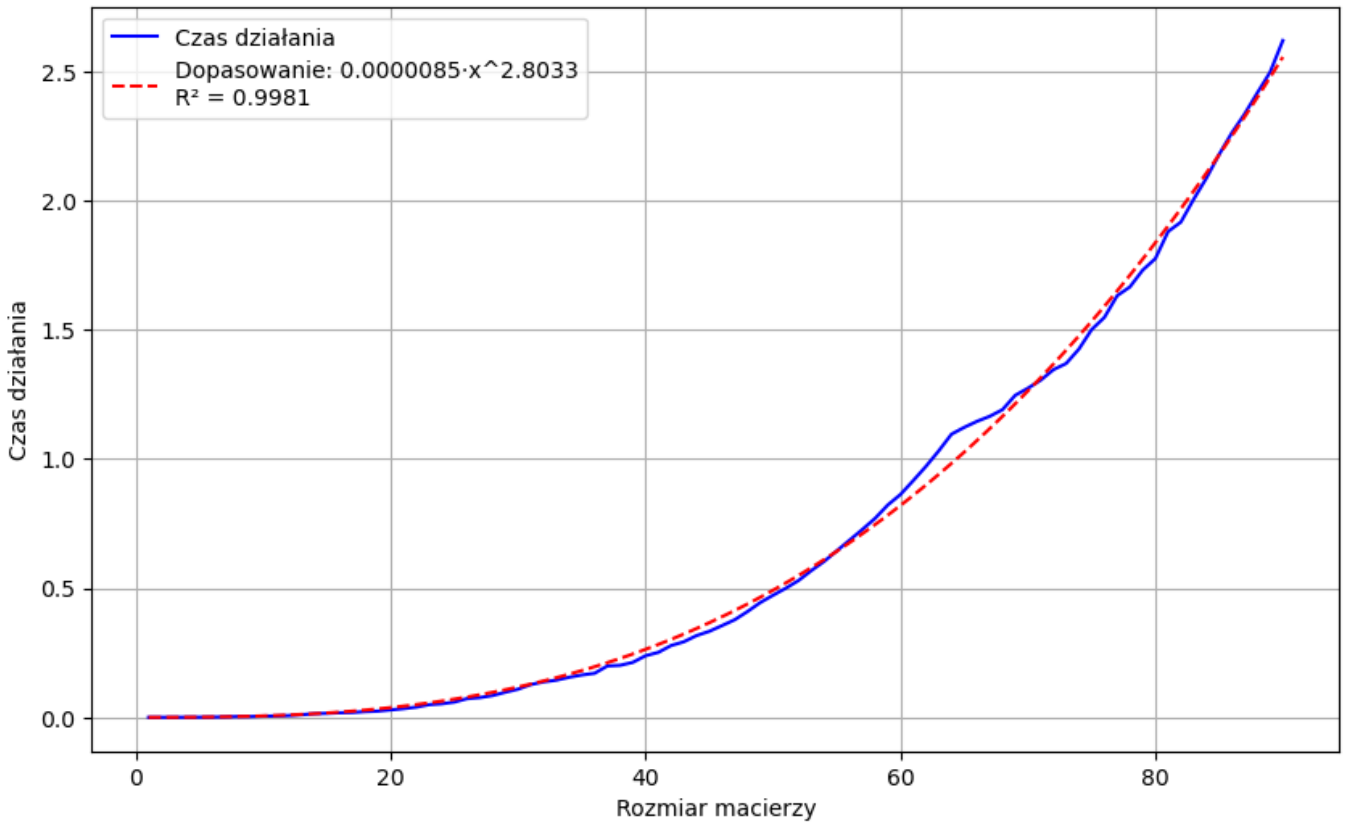
Metoda Strassena używała więcej pamięci niż metoda Bineta dla tych samych rozmiarów macierzy.

5 Analiza złożoności algorytmów rekurencyjnych

Zdecydowaliśmy się na empiryczne zbadanie złożoności algorytmów, do analizy złożoności wykorzystaliśmy więc dopasowywanie krzywej do czasów działania algorytmów.

5.1 Metoda Bineta

Spodziewamy się, że metoda Bineta osiąga złożoność asymptotyczną $O(n^3)$. Dopasowujemy więc funkcję postaci $f(x) = a \cdot x^b$ do czasów działania algorytmu, korzystając z funkcji *curve_fit* z biblioteki *scipy.optimize*:

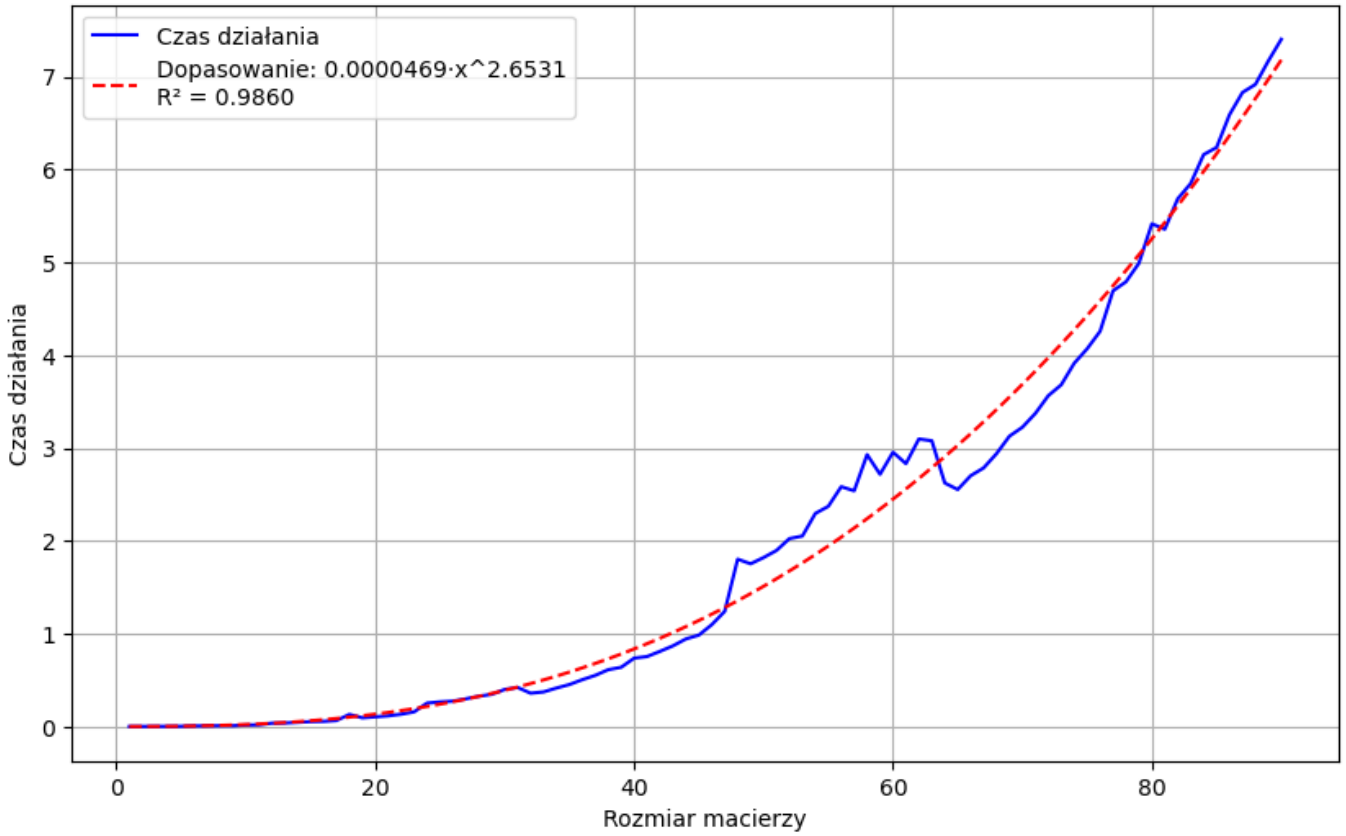


Rysunek 7: Dopasowanie krzywej do wykresu czasów działania algorytmów dla metody Bineta

Otrzymaliśmy bardzo dobre dopasowanie do danych dla parametrów $a = 0.00000085$ i $b = 2.8033$, przy których współczynnik determinacji R^2 wyniósł aż 0.9976. Jest to jednak trochę lepsza złożoność niż teoretyczna złożoność metody Bineta, która wynosi $O(n^3)$. Do otrzymania empirycznej złożoności $O(n^3)$ mogłoby być potrzebne przeprowadzenie testów dla macierzy o większych rozmiarach.

5.2 Metoda Strassena

W tym przypadku również spróbujemy dopasować do danych funkcję postaci $f(x) = b \cdot x^a$. Otrzymujemy następujące dopasowanie dla parametrów $b = 0.0000469$ i $a = 2.6531$:



Rysunek 8: Dopasowanie krzywej do wykresu czasów działania algorytmów dla metody Bineta (usunięte zostały pomiary dla macierzy o rozmiarach od $n = 81$ do $n = 100$, w celu lepszego dopasowania krzywej)

Złożoność empiryczną wyznaczyliśmy na $O(n^{2.6531})$, co także jest lepszą złożonością niż ta przewidywana ($O(n^{2.8074})$), jednak tu również problem mogła stanowić zbyt mała liczba testów oraz zauważalna anomalia dla rozmiarów macierzy od $n = 45$ do $n = 65$.