

# Algorytmy macierzowe - Drugi zestaw zadań

Jakub Frączek      Kacper Garus

30 października 2024

## Spis treści

<b>1</b>	<b>Wstęp</b>	<b>3</b>
<b>2</b>	<b>Dane techniczne</b>	<b>3</b>
2.1	Hardware . . . . .	3
2.2	Software . . . . .	3
<b>3</b>	<b>Rekurencyjne odwracanie macierzy</b>	<b>3</b>
3.1	Opis teoretyczny . . . . .	3
3.2	Pseudokod . . . . .	4
3.3	Implementacja . . . . .	4
<b>4</b>	<b>Rekurencyjna LU faktoryzacja</b>	<b>5</b>
4.1	Opis teoretyczny . . . . .	5
4.2	Pseudokod . . . . .	6
4.3	Implementacja . . . . .	7
<b>5</b>	<b>Rekurencyjna eliminacja Gaussa</b>	<b>7</b>
5.1	Opis teoretyczny . . . . .	7
5.2	Pseudokod . . . . .	8
5.3	Implementacja . . . . .	9
<b>6</b>	<b>Rekurencyjne liczenie wyznacznika macierzy</b>	<b>10</b>
<b>7</b>	<b>Opis teoretyczny</b>	<b>10</b>
7.1	Pseudokod . . . . .	10
7.2	Implementacja . . . . .	10
<b>8</b>	<b>Pomiar liczby operacji zmiennoprzecinkowych i czasów wykonania</b>	<b>11</b>
8.1	Rekurencyjne odwracanie macierzy . . . . .	11
8.2	Rekurencyjna LU faktoryzacja . . . . .	13
8.3	Rekurencyjna eliminacja Gaussa . . . . .	15
8.4	Rekurencyjne liczenie wyznacznika macierzy . . . . .	17

<b>9</b>	<b>Porównanie liczby operacji zmiennoprzecinkowych i czasów wykonania</b>	<b>18</b>
<b>10</b>	<b>Oszacowanie złożoności obliczeniowej</b>	<b>20</b>
<b>11</b>	<b>Porównanie wyników z Octave</b>	<b>20</b>
11.1	Rekurencyjne odwracanie macierzy . . . . .	20
11.2	Rekurencyjna LU faktoryzacja . . . . .	21
11.3	Rekurencyjna eliminacja Gaussa . . . . .	21
11.4	Rekurencyjne liczenie wyznacznika . . . . .	22
<b>12</b>	<b>Wnioski</b>	<b>23</b>
<b>13</b>	<b>Źródła</b>	<b>23</b>

# 1 Wstęp

Tematem zadania było wygenerowanie losowych macierzy o wartościach z przedziału otwartego  $(0.00000001, 1.0)$ , a następnie zaimplementowanie algorytmów:

1. Rekurencyjnego odwracania macierzy
2. Rekurencyjnej LU faktoryzacji
3. Rekurencyjnej eliminacji Gaussa
4. Rekurencyjnego liczenia wyznacznika

Następnie zliczyć liczbę operacji zmienna-przecinkowych dokonaną podczas mnożenia macierzy. Algorytmy miały zostać zaprojektowane tak, aby przyjmować macierze o dowolnych wymiarach.

## 2 Dane techniczne

### 2.1 Hardware

Testy algorytmów zostały wykonane na komputerze z zainstalowaną 64 bitową wersją windowsa 11. Wykorzystany procesor to Intel Core i5-9300H.

### 2.2 Software

Algorytmy zostały zaimplementowane w języku Python i przetestowane na wersji 3.11.9. Wykorzystane biblioteki to:

1. sys
2. os
3. numpy
4. matplotlib
5. time

## 3 Rekurencyjne odwracanie macierzy

### 3.1 Opis teoretyczny

Algorytm odwracania macierzy polega na podzieleniu macierzy na 4 podmacierze. Następnie zgodnie ze wzorem:

$$\begin{bmatrix} A_{11}^{-1} + A_{11}^{-1} A_{12} S_{22}^{-1} A_{21} A_{11}^{-1} - A_{11}^{-1} A_{12} S_{22}^{-1} \\ -S_{22}^{-1} A_{21} A_{11}^{-1} & S_{22}^{-1} \end{bmatrix}$$

Wywołujemy rekurencyjnie funkcję odwracania, a w miejscach, gdzie występuje mnożenie macierzy korzystamy z algorytmu Strassena zaimplementowanego na poprzednim laboratorium.

### 3.2 Pseudokod

```
Funkcja inverse(A)
    Jeżeli rozmiar(A) == 1
        Jeżeli A == 0
            Zwroć 0
        W przeciwnym wypadku
            Zwróć 1/A

S = kształt macierzy A

A = macierz A uzupełniona do parzystego kształtu zerami

środek = dzielenie_całkowite(rozmiar(A[0]), 2)

a11 = Wiersze od 0 do środek, Kolumny od 0 do środek z macierzy A
a12 = Wiersze od 0 do środek, Kolumny od środek do n z macierzy A
a21 = Wiersze od środek do n, Kolumny od 0 do środek z macierzy A
a22 = Wiersze od środek do n, Kolumny od środek do n z macierzy A

a11_inv = inverse(a11)
s22 = a22 - a21*a11_inv*a12
s22_inv = inverse(s22)

c11 = a11_inv + a11_inv * a12 * s22_inv * a21 * a11_inv
c12 = -a11_inv * a12 * s22_inv
c21 = -s22_inv * a21 * a11_inv
c22 = s22_inv

C = złącz macierz z c11, c12, c21, c22

C = macierz C przycięta do kształtu S

Zwróć macierz C
```

### 3.3 Implementacja

Algorytm postanowiliśmy zaimplementować w języku Python:

```

def recursive_inverse(a):
    if np.size(a[0]) == 1:

        return a if a[0, 0] == 0 else np.array([[1 / a[0, 0]]])
    original_shape = a.shape
    a = pad_matrix_even(a)

    n = np.size(a[0])
    mid = n // 2

    a11 = a[:mid, :mid]
    a12 = a[:mid, mid:]
    a21 = a[mid:, :mid]
    a22 = a[mid:, mid:]

    a11inv = recursive_inverse(a11)
    s22 = a22 - strassen(strassen(a21, a11inv), a12)
    s22inv = recursive_inverse(s22)

    b11 = a11inv + strassen(
        strassen(strassen(strassen(a11inv, a12), s22inv), a21), a11inv
    )
    b12 = -strassen(strassen(a11inv, a12), s22inv)
    b21 = -strassen(strassen(s22inv, a21), a11inv)
    b22 = s22inv

    return unpad_matrix(
        np.vstack((np.hstack((b11, b12)), np.hstack((b21, b22)))), original_shape
    )

```

## 4 Rekurencyjna LU faktoryzacja

### 4.1 Opis teoretyczny

Rekurencyjna eliminacja Gaussa polega na podzieleniu macierzy na 4 podmacierze, a następnie obliczenie ich zawartości według wzoru:

$$LU = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ A_{21}U_{11}^{-1} & L_s \end{bmatrix} \begin{bmatrix} U_{11} & L_{11}^{-1}A_{12} \\ 0 & U_s \end{bmatrix} \quad (1)$$

1. Obliczenie rekurencyjnie  $[L_{11}, U_{11}] = LU(A_{11})$
2. Obliczenie rekurencyjnie  $U_{11}^{-1} = \text{inverse}(U_{11})$

3. Obliczenie  $L_{21} = A_{21}U_{11}^{-1}$
4. Obliczenie rekurencyjnie  $L_{11}^{-1} = \text{inverse}(L_{11})$
5. Obliczenie  $U_{12} = L_{11}^{-1}A_{12}$
6. Obliczenie  $L_{22} = S = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12}$
7. Obliczenie rekurencyjnie  $[L_s, U_s] = LU(S)$
8.  $U_{22} = U_s$
9.  $L_{22} = L_s$

## 4.2 Pseudokod

Funkcja LU(A)

Jeżeli rozmiar(A) == 1  
Zwróć 1, a

S = kształt macierzy A

A = macierz A uzupełniona do parzystego kształtu zerami

środek = dzielenie\_całkowite(rozmiar(A[0]), 2)

a11 = Wiersze od 0 do środek, Kolumny od 0 do środek z macierzy A  
a12 = Wiersze od 0 do środek, Kolumny od środek do n z macierzy A  
a21 = Wiersze od środek do n, Kolumny od 0 do środek z macierzy A  
a22 = Wiersze od środek do n, Kolumny od środek do n z macierzy A

l11, u11 = LU(a11)  
u11\_inv = inverse(u11)  
l21 = a21 \* u11\_inv  
l11\_inv = inverse(l11)  
u12 = l11\_inv \* a12  
s = a22 - a21 \* u11\_inv \* l11\_inv \* a12  
l22, u22 = LU(s)  
l12 = macierz zer w kształcie macierzy a11  
u21 = macierz zer w kształcie macierzy a11

L = złoż macierz z l11, l12, l21, l22 i przytnij ją do kształtu S  
U = złoż macierz z u11, u12, u21, u22 i przytnij ją do kształtu S

Zwróć L, U

## 4.3 Implementacja

Algorytm LU faktoryzacji również został zaimplementowany w języku Python:

```
def recursive_LU(a):
    if np.size(a[0]) == 1:

        return np.array([[1]]), a
    original_shape = a.shape
    a = pad_matrix_even(a)

    n = np.size(a[0])
    mid = n // 2

    a11 = a[:mid, :mid]
    a12 = a[:mid, mid:]
    a21 = a[mid:, :mid]
    a22 = a[mid:, mid:]

    l11, u11 = recursive_LU(a11)
    u11inv = recursive_inverse(u11)
    l21 = strassen(a21, u11inv)
    l11inv = recursive_inverse(l11)
    u12 = strassen(l11inv, a12)
    s = a22 - strassen(strassen(strassen(a21, u11inv), l11inv), a12)
    l22, u22 = recursive_LU(s)
    l = unpad_matrix(
        np.vstack((np.hstack((l11, np.zeros(l11.shape))), np.hstack((l21, l22)))),
        original_shape,
    )
    u = unpad_matrix(
        np.vstack((np.hstack((u11, u12)), np.hstack((np.zeros(l11.shape), u22)))),
        original_shape,
    )
    return l, u
```

## 5 Rekurencyjna eliminacja Gaussa

### 5.1 Opis teoretyczny

Rekurencyjna eliminacja Gaussa jest opisana następującym wzorem

$$\begin{bmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{bmatrix} = \begin{bmatrix} U_{11} & L_{11}^{-1}A_{12} \\ 0 & U_s \end{bmatrix} = \begin{bmatrix} RHS_1 \\ RHS_2 \end{bmatrix} = \begin{bmatrix} L_{11}^{-1}b_1 \\ L_s^{-1}b_2 - L_s^{-1}A_{21}U_{11}^{-1}L_{11}^{-1}b_1 \end{bmatrix}$$

1. Oblicz rekurencyjnie  $[L_{11}, U_{11}] = LU(A_{11})$ .
2. Oblicz rekurencyjnie  $L_{11}^{-1} = \text{inverse}(L_{11})$ .
3. Oblicz rekurencyjnie  $U_{11}^{-1} = \text{inverse}(U_{11})$ .
4. Oblicz  $S = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12}$ .
5. Oblicz rekurencyjnie  $[L_S, U_S] = LU(S)$ .
6. Ustal  $C_{11} = U_{11}$ ,  $C_{12} = L_{11}^{-1}A_{12}$ ,  $C_{22} = U_S$ .
7. Oblicz  $RHS_1 = L_{11}^{-1}b_1$ .
8. Oblicz  $RHS_2 = L_S^{-1}b_2 - L_S^{-1}A_{21}U_{11}^{-1}L_{11}^{-1}b_1$ .

## 5.2 Pseudokod

Funkcja Gauss(A, b)

SA = kształt macierzy A

A = macierz A uzupełniona do parzystego kształtu zerami

środek = dzielenie\_całkowite(rozmiar(A[0]), 2)

a11 = Wiersze od 0 do środek, Kolumny od 0 do środek z macierzy A

a12 = Wiersze od 0 do środek, Kolumny od środek do n z macierzy A

a21 = Wiersze od środek do n, Kolumny od 0 do środek z macierzy A

a22 = Wiersze od środek do n, Kolumny od środek do n z macierzy A

Sb = kształt wektora b

b = wektor b uzupełniony do parzystej długości zerami

b1 = wektor b od początku do środka

b2 = wektor b od środka do końca

l11, u11 = LU(a11)

l11\_inv = inverse(l11)

u11\_inv = inverse(u11)

s = a22 - a21 \* u11\_inv \* l11\_inv \* a12

ls, us = LU(s)

ls\_inv = inverse(ls)

c11 = u11

c12 = l11\_inv \* a12

c21 = macierz zer w kształcie macierzy a11



```

c22 = us

LHS = złoŹ macierz z c11, c12, c21, c22 i przytnij ję do kształtu SA

RHS1 = l11_inv * b1
RHS2 = ls_inv * b2 - ls_inv * a21 * u11_inv * l11_inv * b1

RHS = złoŹ wektor z wektorów RHS1 i RHS2, przytnij go do kształtu Sb

Zwróć LHS, RHS

```

### 5.3 Implementacja

Algorytm eliminacji Gaussa również został zaimplementowany w języku Python:

```

def recursive_Gauss(a, b):

    original_shape_a = a.shape
    a = pad_matrix_even(a)
    n = np.size(a[0])
    mid = n // 2

    a11 = a[:mid, :mid]
    a12 = a[:mid, mid:]
    a21 = a[mid:, :mid]
    a22 = a[mid:, mid:]

    original_shape_b = b.shape

    b = pad_vector_even(b)

    b1 = b[:mid]
    b2 = b[mid:]

    l11, u11 = recursive_LU(a11)
    l11inv = recursive_inverse(l11)
    u11inv = recursive_inverse(u11)
    s = a22 - strassen(strassen(strassen(a21, u11inv), l11inv), a12)
    ls, us = recursive_LU(s)
    lsinv = recursive_inverse(ls)

    c11 = u11
    c12 = strassen(l11inv, a12)
    c21 = np.zeros(c12.shape)
    c22 = us

```

```

lhs = unpad_matrix(
    np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22)))), original_shape_a
)

rhs1 = matrice_vector_mult(l11inv, b1)
rhs2 = matrice_vector_mult(lsinv, b2) - matrice_vector_mult(
    strassen(strassen(strassen(lsinv, a21), u11inv), l11inv), b1
)

rhs = unpad_vector(np.hstack((rhs1, rhs2)), original_shape_b)

return lhs, rhs

```

## 6 Rekurencyjne liczenie wyznacznika macierzy

## 7 Opis teoretyczny

Wyznacznik macierzy obliczany jest zgodnie z podanym wzorem:

$$\det(A) = l_{11} \cdot \dots \cdot l_{nn} \cdot u_{11} \cdot \dots \cdot u_{nn} = (l_{ii} = 1) = u_{11} \cdot \dots \cdot u_{nn}$$

Gdzie: -  $l_{ii}$  to przekątna macierzy  $L$  -  $u_{ii}$  to przekątna macierzy  $U$

### 7.1 Pseudokod

```

Funkcja determinant(A)
  L, U = LU(A)
  det = 1
  Dla u = element przekątnej macierzy U
    det = det * u
  Zwróć det

```

### 7.2 Implementacja

Algorytm eliminacji Gaussa również został zaimplementowany w języku Python:

```

def recursive_determinant(a):
    l,u=recursive_LU(a)
    det=1
    for i in range(u.shape[0]):
        det*=u[i,i]

```

```
return det
```

## 8 Pomiar liczby operacji zmiennoprzecinkowych i czasów wykonania

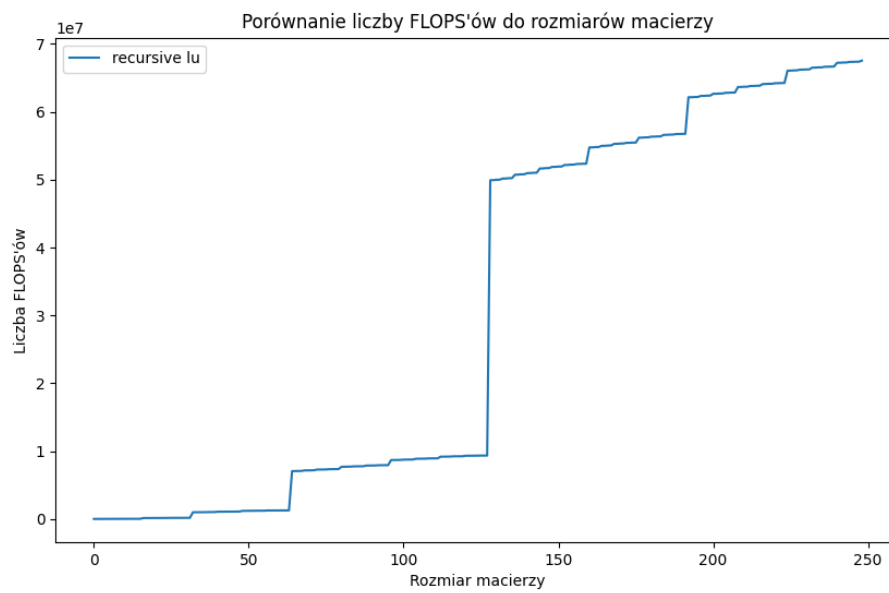
Podobnie jak w laboratorium pierwszym zliczanie operacji zmiennoprzecinkowych osiągnęliśmy poprzez napisanie klasy dziedziczącej po `float`'cie oraz przeciążenie metod realizujących operacje zmiennoprzecinkowe, tak aby oprócz wykonania operacji aktualizowały one licznik.

### 8.1 Rekurencyjne odwracanie macierzy

Najpierw przeprowadziliśmy pomiary dla algorytmu rekurencyjnego odwracania macierzy. W tabeli poniżej znajdują się przykładowe wyniki dla wybranych rozmiarów macierzy.

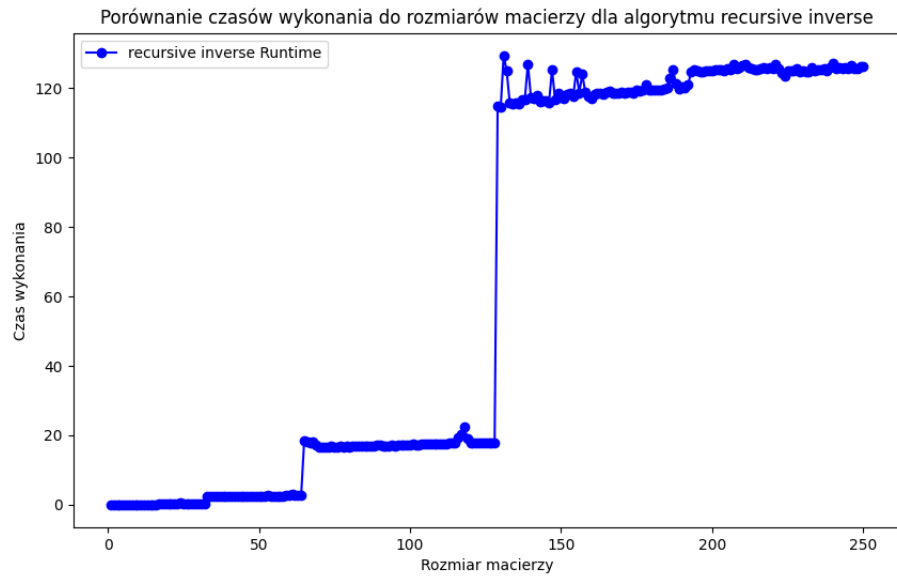
Rozmiar macierzy	Mnożenie	Dodawanie	Odejmowanie
2	10	1	1
50	233464	797025	385105
100	1634636	5749410	2782226
150	10984987	33548648	15498120
200	11444476	40928120	19831800
250	11528902	44626863	22236885

Poniżej znajduje się wykres prezentujący sumę operacji zmiennoprzecinkowych dla rozmiarów macierzy od 1 do 250.



Rysunek 1: Wykres sumarycznej liczby FLOPS'ów dla rekurencyjnego odwracania macierzy

Finalnie sporządziliśmy wykres czasu działania od rozmiaru macierzy.



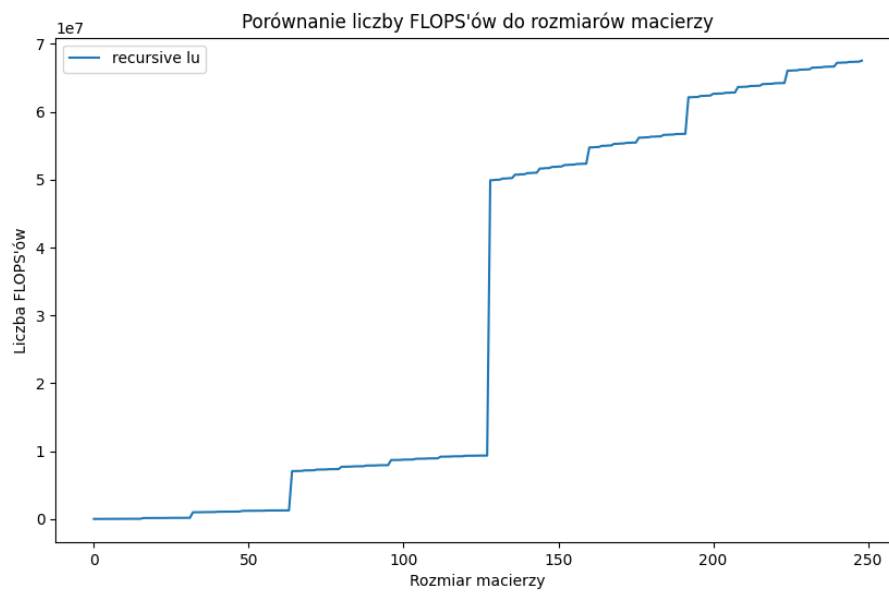
Rysunek 2: Wykres czasu wykonania w zależności od rozmiaru macierzy dla rekurencyjnego odwracania macierzy

## 8.2 Rekurencyjna LU faktoryzacja

Po przeprowadzeniu pomiarów sporządziliśmy tabelę zawierającą liczby FLOPS'ów dla wybranych rozmiarów macierzy.

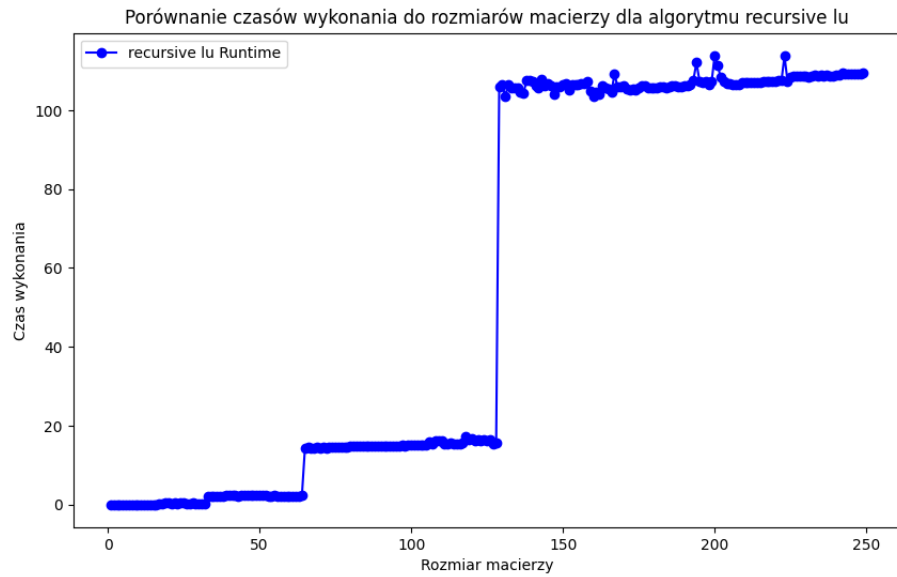
Rozmiar macierzy	Mnożenie	Dodawanie	Odejmowanie
2	5	0	1
50	207393	663721	319537
100	1455057	4890885	2351520
150	9653187	28923656	13298213
200	10198893	35230231	16940640
250	10359973	38275181	18897262

Poniżej znajduje się wykres prezentujący sumę operacji zmiennoprecinkowych dla rozmiarów macierzy od 1 do 250.



Rysunek 3: Wykres sumarycznej liczba FLOPS'ów dla rekurencyjnej LU faktoryzacji

Na końcu sporządziliśmy wykres czasu działania od rozmiaru macierzy.



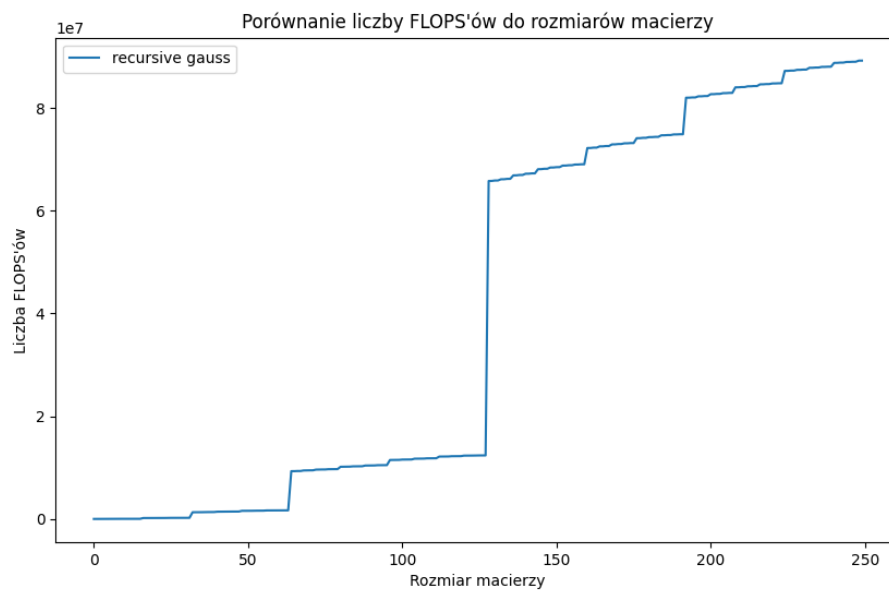
Rysunek 4: Wykres czasu wykonania w zależności od rozmiaru macierzy dla rekurencyjnej LU faktoryzacji

### 8.3 Rekurencyjna eliminacja Gaussa

Pnownie po przeprowadzeniu pomiarów sporządziliśmy tabelę zawierającą liczby FLOPS'ów dla wybranych rozmiarów macierzy.

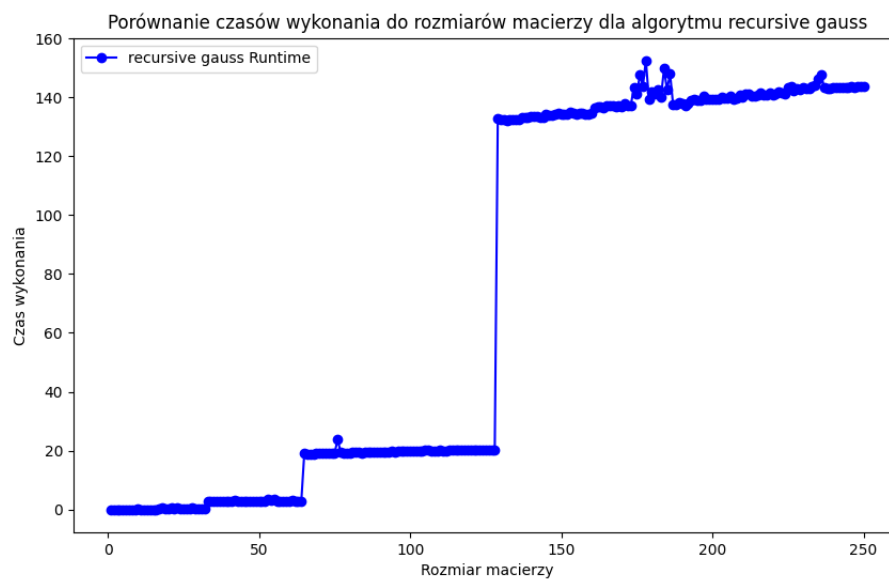
Rozmiar macierzy	Mnożenie	Dodawanie	Odejmowanie
2	8	0	1
50	275250	880253	423244
100	1924281	6464124	3105401
150	12725180	38129527	17525345
200	13464465	46480441	22338954
250	13698407	50537636	24936894

Poniżej znajduje się wykres prezentujący sumę operacji zmiennoprecinkowych dla rozmiarów macierzy od 1 do 250.



Rysunek 5: Wykres sumarycznej liczba FLOPS'ów dla rekurencyjnej eliminacji Gaussa

Finalnie sporządziliśmy wykres czasu działania od rozmiaru macierzy.



Rysunek 6: Wykres czasu wykonania w zależności od rozmiaru macierzy dla rekurencyjnej eliminacji Gaussa

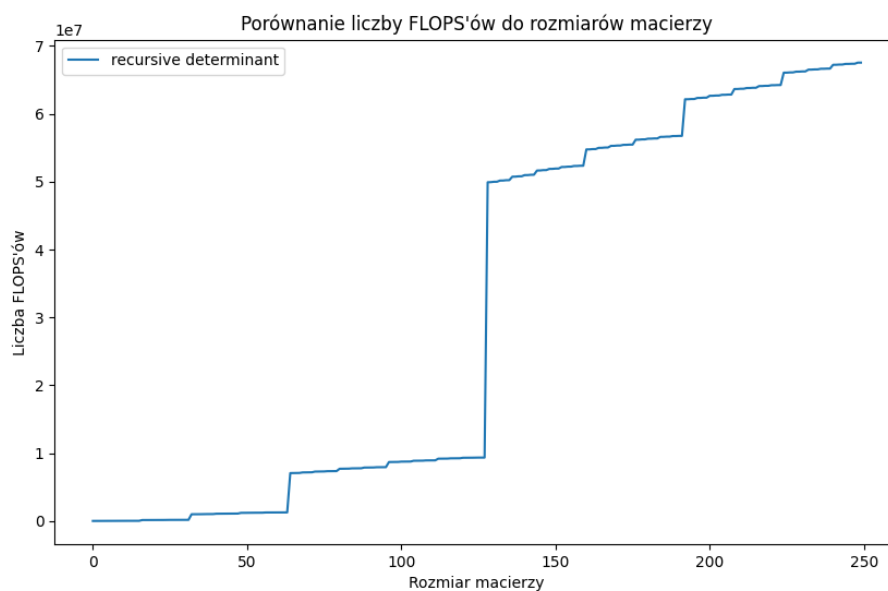


## 8.4 Rekurencyjne liczenie wyznacznika macierzy

Tak jak poprzednio sporządziliśmy tabelę zawierającą liczby FLOPS'ów dla wybranych rozmiarów macierzy.

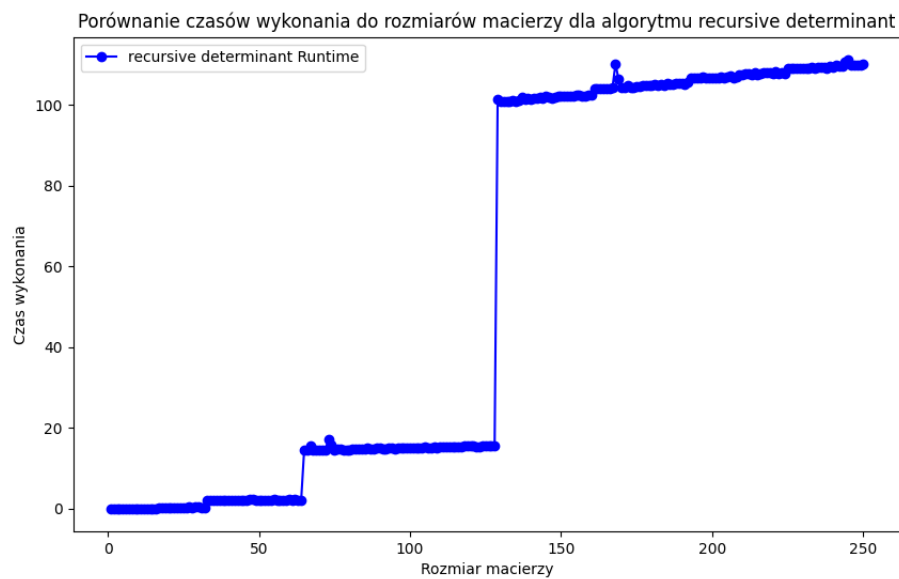
Rozmiar macierzy	Mnożenie	Dodawanie	Odejmowanie
2	7	0	1
50	207443	663721	319537
100	1455157	4890885	2351520
150	9653337	28923656	13298213
200	10199093	35230231	16940640
250	10360223	38275181	18897262

Poniżej znajduje się wykres prezentujący sumę operacji zmiennoprecinkowych dla rozmiarów macierzy od 1 do 250.



Rysunek 7: Wykres sumarycznej liczby FLOPS'ów dla rekurencyjnego liczenia wyznacznika

Finalnie sporządziliśmy wykres czasu działania od rozmiaru macierzy.

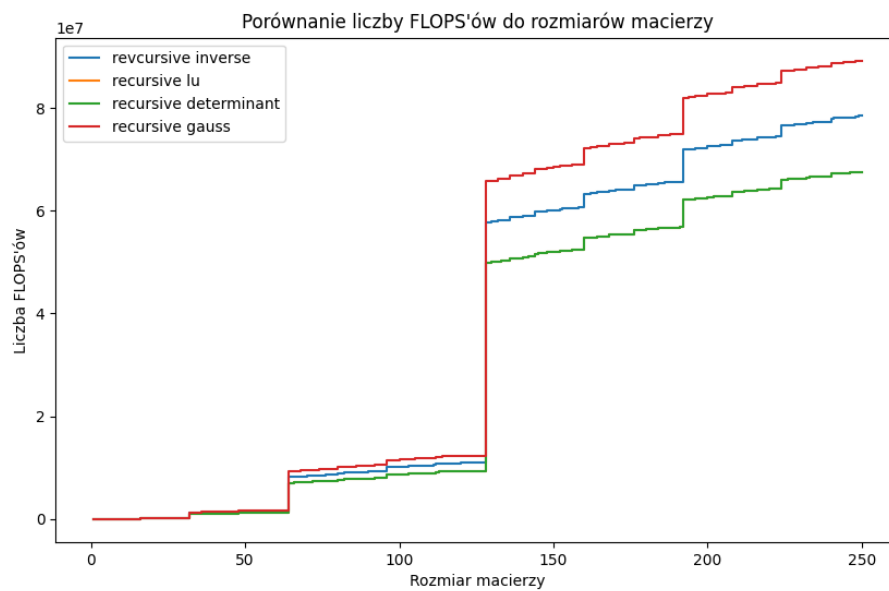


Rysunek 8: Wykres czasu wykonania w zależności od rozmiaru macierzy dla rekurencyjnego liczenia wyznacznika

## 9 Porównanie liczby operacji zmiennoprzecinkowych i czasów wykonania

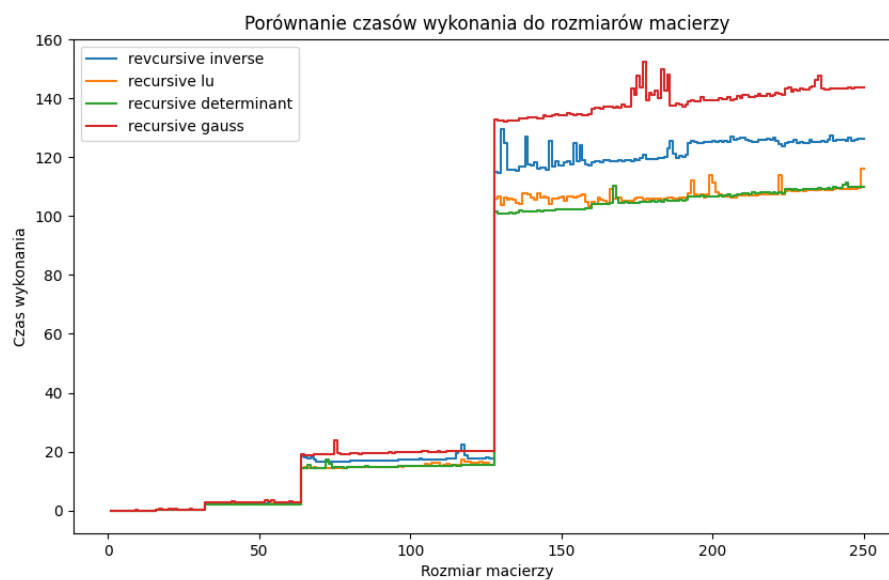
Zdecydowaliśmy się również sporządzić wykresy obrazujące jak ma się liczba operacji zmiennoprzecinkowych między sobą dla rozważanych algorytmów i to samo zrobić dla czasów wykonania.

Porównanie liczby operacji zmiennoprzecinkowych:



Rysunek 9: Porównanie sumarycznej liczby FLOPS'ów dla wszystkich rozważanych algorytmów

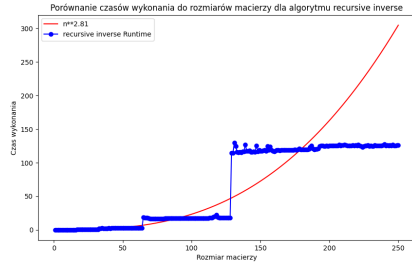
Porównanie czasów działania:



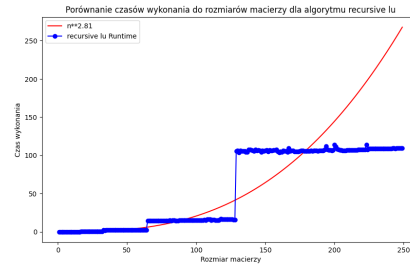
Rysunek 10: Porównanie czasów działania dla wszystkich rozważanych algorytmów

## 10 Oszacowanie złożoności obliczeniowej

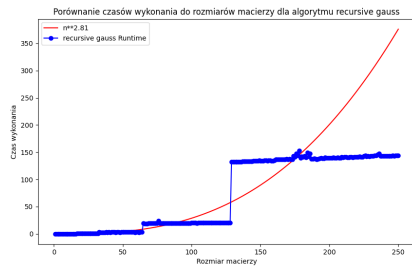
Z powodu wykorzystania algorytmu Strassena w algorytmach rekurencyjnych spodziewaliśmy się otrzymać złożoność obliczeniową w okolicach  $n^{2.81}$ . Aby potwierdzić nasze przypuszczenia postanowiliśmy dopasować krzywą tej złożoności do otrzymanych wykresów czasów wykonania.



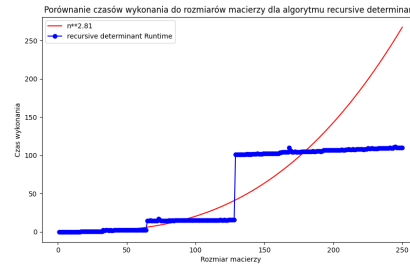
Rysunek 11: Dopasowanie krzywej złożoności do wykresu czasu działania rekurencyjnego odwracania macierzy



Rysunek 12: Dopasowanie krzywej złożoności do wykresu czasu działania rekurencyjnej LU faktoryzacji



Rysunek 13: Dopasowanie krzywej złożoności do wykresu czasu działania rekurencyjnej eliminacji Gaussa



Rysunek 14: Dopasowanie krzywej złożoności do wykresu czasu działania rekurencyjnego liczenia wyznacznika macierzy

Jak widać na powyższych wykresach nasze przypuszczenia się potwierdziły.

## 11 Porównanie wyników z Octave

### 11.1 Rekurencyjne odwracanie macierzy

Przeprowadziliśmy testy dla macierzy:

$$A = \begin{bmatrix} 1.12 & 6.31 & 3.52 & 5.31 & 3.23 \\ 2.43 & 5.23 & 7.43 & 6.54 & 2.22 \\ 0.76 & 4.98 & 7.86 & 4.00 & 7.43 \\ 9.99 & 4.33 & 5.44 & 3.45 & 2.45 \\ 6.43 & 5.44 & 3.23 & 1.23 & 2.67 \end{bmatrix}$$

Otrzymane wyniki to:

```
A_inv =
 2.8284e-02  -8.4689e-02  -1.7002e-02   1.5894e-01  -6.2336e-02
 2.4815e-03   1.0608e-01  -7.5664e-02  -2.3734e-01   3.3718e-01
 -3.0839e-01   2.7350e-01   3.7129e-02  -1.5153e-01   1.8139e-01
 2.7983e-01  -1.1230e-01  -3.1221e-02   2.4472e-01  -3.8282e-01
 1.7106e-01  -2.9131e-01   1.6457e-01   1.7137e-01  -2.0541e-01
```

```
[[ 0.02828355 -0.08468873 -0.01700191  0.15894487 -0.0623362 ]
 [ 0.00245146  0.10607761 -0.07566418 -0.23734449  0.33717916]
 [-0.30839429  0.2735043  0.03712922 -0.151527  0.18138766]
 [ 0.27982703 -0.11229532 -0.03122072  0.24472092 -0.3828247 ]
 [ 0.17105885 -0.29131435  0.16457285  0.17137227 -0.20540847]]
```

Rysunek 15: Wyniki dla odwracania macierzy otrzymane z wykorzystaniem Octave

Rysunek 16: Wyniki dla odwracania macierzy otrzymane z wykorzystaniem własnej implementacji

Jak widać wyniki są takie same.

## 11.2 Rekurencyjna LU faktoryzacja

Ponownie wykorzystaliśmy macierz z poprzedniego podpunktu. Otrzymane wyniki:

```
L =
 1.0000    0    0    0    0
 0.1121  1.0000    0    0    0
 0.0761  0.7984  1.0000    0    0
 0.6436  0.4555 -0.3117  1.0000    0
 0.2432  0.7171  0.7847 -0.7051  1.0000

U =
 9.9900  4.3300  5.4400  3.4500  2.4500
 0  5.8246  2.9101  4.9232  2.9553
 0  0  5.1226 -0.1934  4.8839
 0  0  0 -3.2933  1.2695
 0  0  0  0 -3.4327
```

```
L =
[[ 1. 0. 0. 0. 0. ]
 [ 2.16964286 1. 0. 0. 0. ]
 [ 0.67857143 -0.08252688 1. 0. 0. ]
 [ 0.91964286 0.14068618 -4.52578534 1. 0. ]
 [ 5.74107143 3.63883407 -2.97466451 0.83429996 1. ]]

U =
[[ 1.12000000e+00 6.31000000e+00 3.52000000e+00 5.31000000e+00
 3.23000000e+00]
 [ 0.00000000e+00 -8.46844643e+00 -2.07142857e-01 -4.98880357e+00
 -4.78794643e+00]
 [ 0.00000000e+00 0.00000000e+00 5.45433372e+00 -1.42644871e-02
 4.84307998e+00]
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 -1.33923099e+01
 2.49595704e+01]
 [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 -4.86834850e+00]]
```

Rysunek 17: Wyniki dla lu faktoryzacji macierzy otrzymane z wykorzystaniem Octave

Rysunek 18: Wyniki dla lu faktoryzacji macierzy otrzymane z wykorzystaniem własnej implementacji

Wyniki się zgadzają.

## 11.3 Rekurencyjna eliminacja Gaussa

Testy zostały przeprowadzone dla tej samej macierzy A oraz wektora b:

$$\mathbf{b} = \begin{bmatrix} 1.23 \\ 2.12 \\ 7.54 \\ 8.55 \\ 3.45 \end{bmatrix}$$

Oto otrzymane wyniki:

Upper Triangular Matrix:  
Columns 1 through 5:

```

1.1200    6.3100    3.5200    5.3100    3.2300
      0   -8.4604   -0.2071   -4.9808   -4.7879
      0      0    5.4543   -0.0143    4.8431
      0      0      0   -13.3923   24.9596
      0      0      0      0   -4.8683

```

Column 6:

```

1.2300
-0.5487
 6.6601
31.0901
-7.7420

```

```

lhs =
[1.12 6.31 3.52 5.31 3.23]
[ 0.   -8.46044643 -0.20714286 -4.98888357 -4.78794643]
[ 0.      0.   5.45433372 -0.01426449  4.84307998]
[ 0.      0.      0.   -13.39238994  24.95957041]
[ 0.      0.      0.      0.   -4.8683485]
rhs =
[ 1.23   -0.54866071  6.66007788 31.0900754 -7.74198382]

```

Rysunek 19: Wyniki dla eliminacji gaussa otrzymane z wykorzystaniem Octave

Rysunek 20: Wyniki dla eliminacji gaussa otrzymane z wykorzystaniem własnej implementacji

Jak widać na powyższych rysunkach wyniki się zgadzają

## 11.4 Rekurencyjne liczenie wyznacznika

Test został przeprowadzony na macierzy A opisanej powyżej.  
Oto otrzymane wyniki:

```

>> determinant
-3369.7
>>

```

```

-3369.6916398120434

```

Rysunek 21: Wyniki dla rekurencyjnego liczenia wyznacznika otrzymane z wykorzystaniem Octave

Rysunek 22: Wyniki dla rekurencyjnego liczenia wyznacznika z wykorzystaniem własnej implementacji

Wynik uzyskany z Octave został zaokrąglony, ale i tak można stwierdzić, że nasza metoda jest poprawna.

## 12 Wnioski

- Algorytm Strassena okazuje się bardzo przydatny przy implementacji opisanych algorytmów
- Octave jest dość intuicyjne i posiada sporo wbudowanych funkcji, które pozwalają wykonywać podstawowe operacje macierzowe.
- Podejście rekurencyjne sprawia, że implementacja rozważanych algorytmów jest dość intuicyjna i prosta.
- Mimo złożoności rzędu  $n^{2.81}$  algorytmy wykonują się BARDZO długo dla macierzy o rozmiarach liczonych w setkach.
- Po porównaniu wyników naszych algorytmów z Octave, przekonaliśmy się, że nasza implementacja jest poprawna.

## 13 Źródła

1. Wykład z kursu "Algorytmy Macierzowe"
2. Dokumentacja Octave - <https://docs.octave.org/latest/>