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

Algorytmy macierzowe – rekurencyjne algorytmy macierzowe - sprawozdanie

1. Opis ćwiczenia

Naszym zadaniem było , po wybraniu naszego ulubionego języka, wygenerowanie losowych macierzy których elementy są z przedziału $(10^{-8}, 1)$ i zaimplementowanie algorytmów:

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

Następnie, mieliśmy sprawdzić działanie naszych implementacji na losowo wygenerowanych macierzach rozmiarów $2^k \times 2^k$ gdzie $k \in (2, 3, 4, \dots 16)$.

2. 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.

3. Implementacja algorytmów

3.1 Rekurencyjnego odwracania macierzy

3.1.1 Pseudokod

reverse_matrix*(A, czy_macierz_trójkątna) :*

Jeżeli *A* ma rozmiar 1**:**

Zwróć *odwrotność elementu A*

W przeciwnym wypadku:

Podziel *A* na 4 równych rozmiarów mniejsze macierze $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

Oblicz ${A_{11}}^{-1}$

Jeżeli macierz jest trójkątna (górna lub dolna):

Oblicz pomocniczą macierz $S = A_{22}$

W przeciwnym wypadku (macierz nie jest trójkątna):

Oblicz pomocniczą macierz $S = A_{22} - A_{21} * {A_{11}}^{-1} * A_{12}$

Wygeneruj macierz C jako wypełnioną zerami

Jeżeli macierz nie jest trójkątna:

$C_{11} = {A_{11}}^{-1} + {A_{11}}^{-1} * A_{12} * S^{-1} * A_{21} * {A_{11}}^{-1}$

$C_{12} = -{A_{11}}^{-1} * A_{12} * S^{-1}$

$C_{21} = -S^{-1} * A_{21} * {A_{11}}^{-1}$

W przeciwnym wypadku:

$C_{11} = {A_{11}}^{-1}$

Jeżeli jest trójkątna dolna:

$C_{21} = -S^{-1} * A_{21} * {A_{11}}^{-1}$

Jeżeli jest trójkątna górna:

$C_{12} = -{A_{11}}^{-1} * A_{12} * S^{-1}$

$C_{22} = S^{-1}$

Zwróć C

3.1.2 Istotne fragmenty implementacji

```
def reverse_matrix(A, traigonal = None) -> np.array:
    n = len(A)
    if n == 1:
        a = A[0][0]
        if a != 0:
            return np.array([[1/a]], dtype=Number)
        else:
            raise ValueError("Matrix is not invertilbe")
    n//=2
    A_11,A_12,A_21,  A_22 = A[:n,:n],A[:n,n:],A[n:,:n], A[n:,n:]

    invA_11 = reverse_matrix(A_11)

    if traigonal != None:
        S = A_22
    else:
        S = A_22 - mul(mul(A_21, invA_11),A_12)

    invS = reverse_matrix(S)
    C = np.zeros((2*n,2*n), dtype=Number)
    if traigonal == None:
        C[:n,:n] = invA_11 + mul(mul(mul(invA_11,A_12), invS),mul(A_21, invA_11))
        C[:n,n:] = -1 * mul(mul(invA_11,A_12), invS)
        C[n:,:n] = -1 * mul(mul(invS,A_21), invA_11)
    else:
        C[:n,:n] = invA_11
        if traigonal == "L":
            C[n:,:n] = -1 * mul(mul(invS,A_21), invA_11)
        else:
            C[:n,n:] = -1 * mul(mul(invA_11,A_12), invS)


    C[n:,n:] = invS
    return C
```

3.2 Rekurencyjnej LU faktoryzacji macierzy

3.2.1 Pseudokod

LU_factorise*(A,B):*

Jeżeli *A* ma rozmiar 1**:**

Zwróć *macierz jednostkową o rozmiarze 1 oraz A*

W przeciwnym wypadku:

Podziel *A na* równych rozmiarów mniejsze macierze $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

Oblicz $L_{11}\ U_{11}$ poprzez rekurencyjne wywołanie LU_factorise(A_{11})

Oblicz ${U_{11}}^{-1}$ pamiętając, że jest to macierz trójkątna górna

$L_{21} = A_{21} * {U_{11}}^{-1}$

Oblicz ${L_{11}}^{-1}$ pamiętając, że jest to macierz trójkątna dolna

$U_{12} = {L_{11}}^{-1} * A_{12}$

Oblicz pomocniczą macierz $S = A_{21} * {U_{11}}^{-1} * {L_{11}}^{-1} * A_{12}$

Oblicz $L_{22}\ U_{22}$ poprzez rekurencyjne wywołanie LU_factorise(A_{22})

Zwróć $L \quad oraz\ U$

3.2.2 Istotne fragmenty implementacji

```
def LU_factorise(A):
    if len(A) == 1:
        return np.array([[Number(1)]],dtype=Number), A
    n = len(A)//2
    A_11, A_12, A_21, A_22 = A[:n,:n], A[:n,n:], A[n:,:n],A[n:,n:]
    L_11, U_11 = LU_factorise(A_11)
    invU_11 = reverse_matrix(U_11, "U")
    L_21 = mul(A_21, invU_11)
    invL_11 = reverse_matrix(L_11, "L")
    U_12 = mul(invL_11,A_12)
    S = A_22 - mul(mul(A_21, invU_11), mul(invL_11, A_12))
    L_22, U_22 = LU_factorise(S)
    L, U = np.zeros((2*n,2*n), dtype=Number), np.zeros((2*n,2*n), dtype=Number)
    L[:n,:n], L[n:,:n], L[n:,n:] = L_11, L_21, L_22
    U[:n,:n], U[:n,n:], U[n:,n:] = U_11, U_12, U_22
    return L, U
```

3.3 Rekurencyjnego obliczania wyznacznika

3.3.1 Pseudokod

recursive_det(A):

L, U = LU_factiorise(A)

Zwróć iloczyn elementów na przekątnej U

3.3.2 Istotne fragmenty implementacji

Brak znaczących elementów do opisu, kod jest przepisaniem z pseudokodu na kod Pythona.

4. Analiza wykonanych pomiarów

4.1 Pomiary rekurencyjnego odwracania macierzy

rozmiar	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
4	126	102	228	0.001025
8	1588	726	2314	0.011059
16	14400	5010	19410	0.080217
32	114856	34542	149398	0.530120
64	862056	239202	1101258	5.603233
128	6270328	1663086	7933414	33.094046
256	44843400	11594370	56437770	236.112997
512	317722936	80967822	398690758	1749.358699

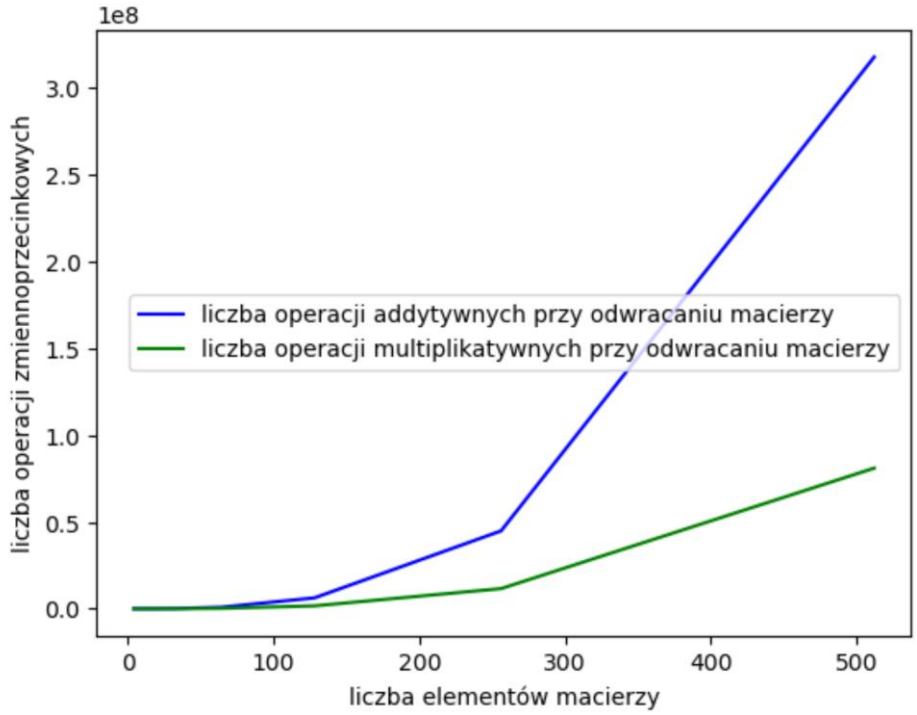
Tab. 1 Pomiary rekurencyjnego odwracania macierzy

Gdzie rozmiar macierzy to ilość elementów w pojedynczym wierszu.

4.1.1 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 1. Można zauważyć, że operacji addytywnych jest znacznie więcej niż multiplikatywnych, z czego można wysnuć wniosek, że chcąc przyspieszyć działanie odwracania macierzy lepiej niskopoziomowo przyspieszać dodawanie.



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:

$$y = a \cdot x^k \tag{1}$$

Gdzie próbowaliśmy oszacować a oraz k.

Na podstawie czasu rekurencyjnego odwracania macierzy otrzymaliśmy:

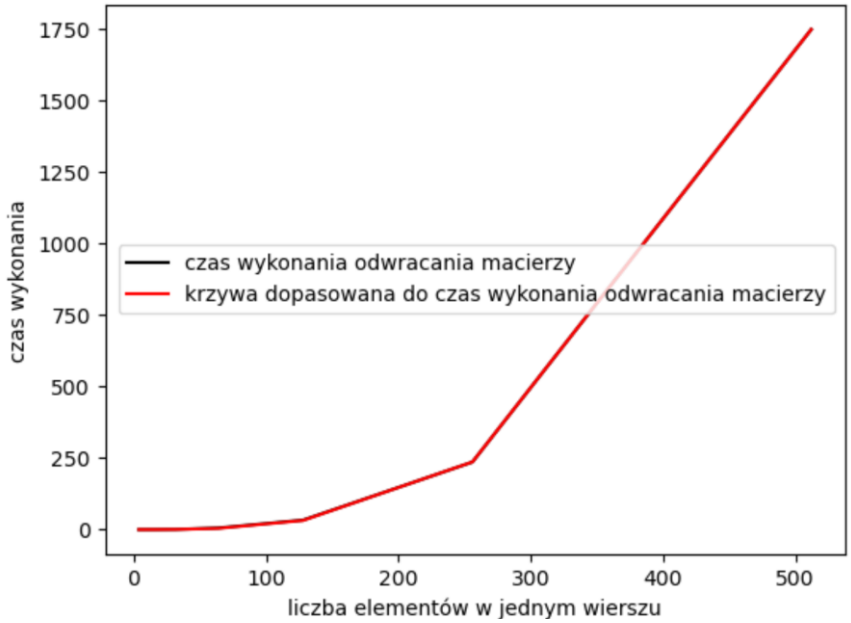
$$a = 2{,}638 \cdot 10^{-5}$$
$$k = 2{,}887 \tag{2}$$

A na podstawie liczby operacji zmiennoprzecinkowych rekurencyjnego odwracania macierzy otrzymaliśmy:

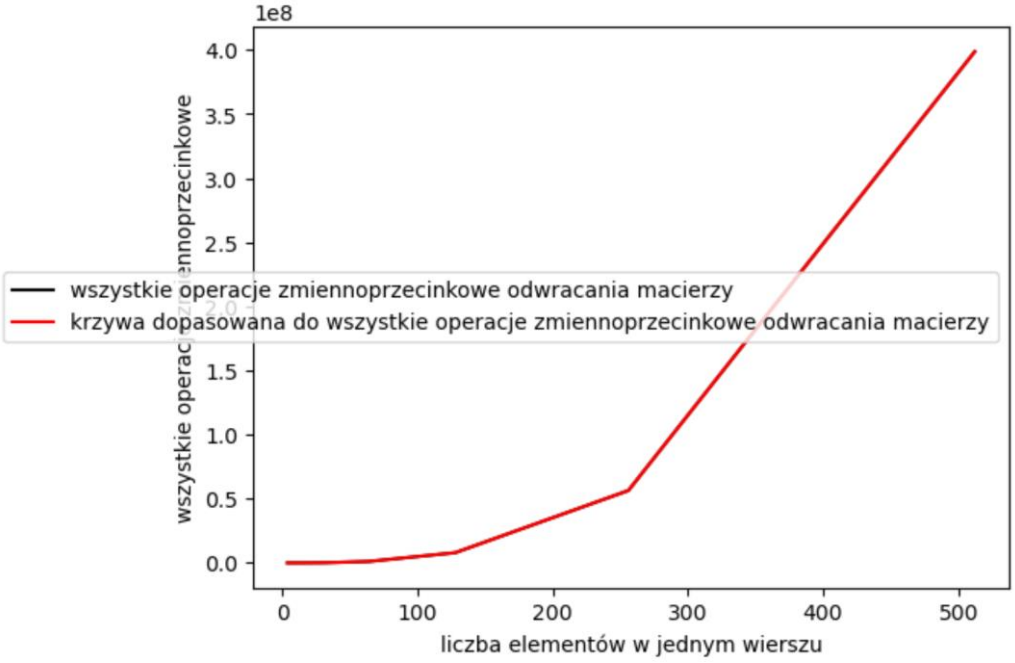
$$a = 9{,}076$$
$$k = 2{,}821 \tag{3}$$

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, ponieważ oba wykresy pokrywają

się ze sobą. Dodatkowo szacunki te pokrywają się z teoretyczną złożonością, która jest ograniczona mnożeniem macierzy. Ponieważ do tego celu korzystaliśmy z algorytmu Strassena, który ma złożoność $O(n^{2,807})$ to widzimy, że wyszła nam trochę gorsza złożoność Dlaczego? Domniemamy, że trochę większa złożoność na podstawie liczby operacji zmiennoprzecinkowych wynika z błędów pomiarowych. W przeciwieństwie do niej, większa złożoność na podstawie czasu działania, wynika ze strat podczas kopiowania macierzy w pamięci i przechowywaniem ich .



Wykres nr. 2



Wykres nr. 3

Zależność liczby operacji zmiennoprzecinkowych od czasu wykonania

Patrząc na fakt, który już wspomnieliśmy, można dostrzec, że szacunek złożoności na podstawie czasu wykonania jest gorszy niż na podstawie liczby operacji zmiennoprzecinkowych ponieważ wyniki są przekłamane ze wsklędu na przechowywanie i przesyłanie macierzy w pamięci.

4.2 Pomiary LU faktoryzacji macierzy

4.2.1 Pomiary

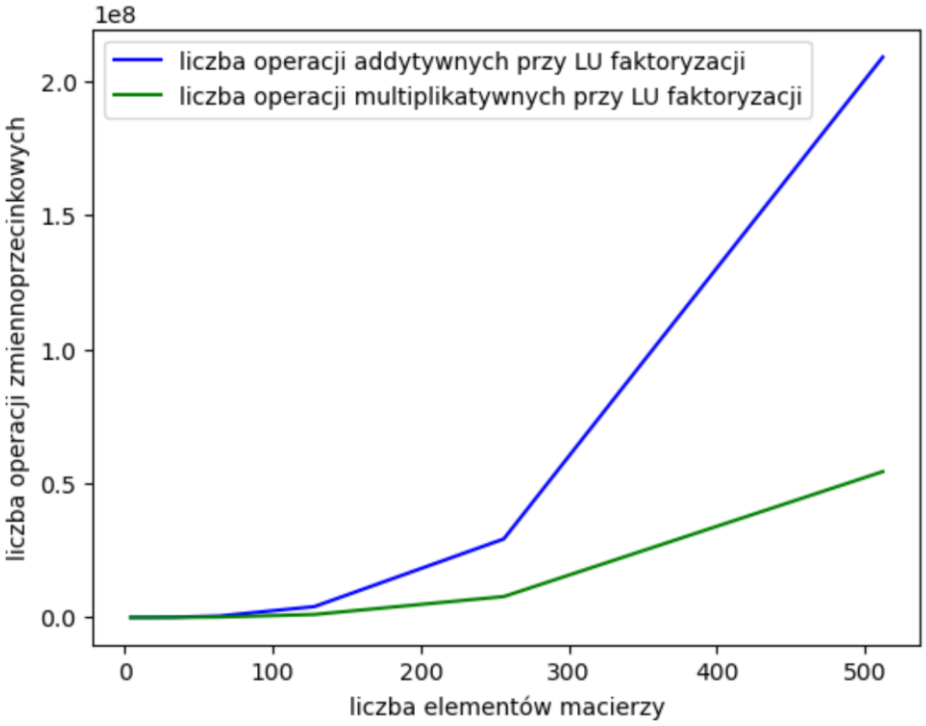
rozmiar	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
4	60	51	111	0.001200
8	832	431	1263	0.007110
16	8240	3213	11453	0.054933
32	69764	22835	92599	0.423226
64	543328	159861	703189	3.556670
128	4040148	1115411	5155559	22.379935
256	29269632	7784133	37053765	158.050221
512	208938868	54371555	263310423	1093.819095

Tab. 2 Pomiary LU faktoryzacji

4.2.2 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 4. Można zauważyć, że operacji addytywnych jest znacznie więcej niż multiplikatywnych, z czego można wysnuć wniosek, że chcąc przyspieszyć działanie LU faktoryzacji lepiej niskopoziomowo przyspieszać dodawanie.



Wykres 4

Szacunek złożoności obliczeniowej

Złożoność obliczeniową szacowaliśmy empirycznie zgodnie ze wzorem nr 1, tak jak w punkcie 4.1.2.

Na podstawie czasu LU faktoryzacji otrzymaliśmy:

$$a = 2{,}984 \cdot 10^{-5}$$
$$k = 2{,}791$$

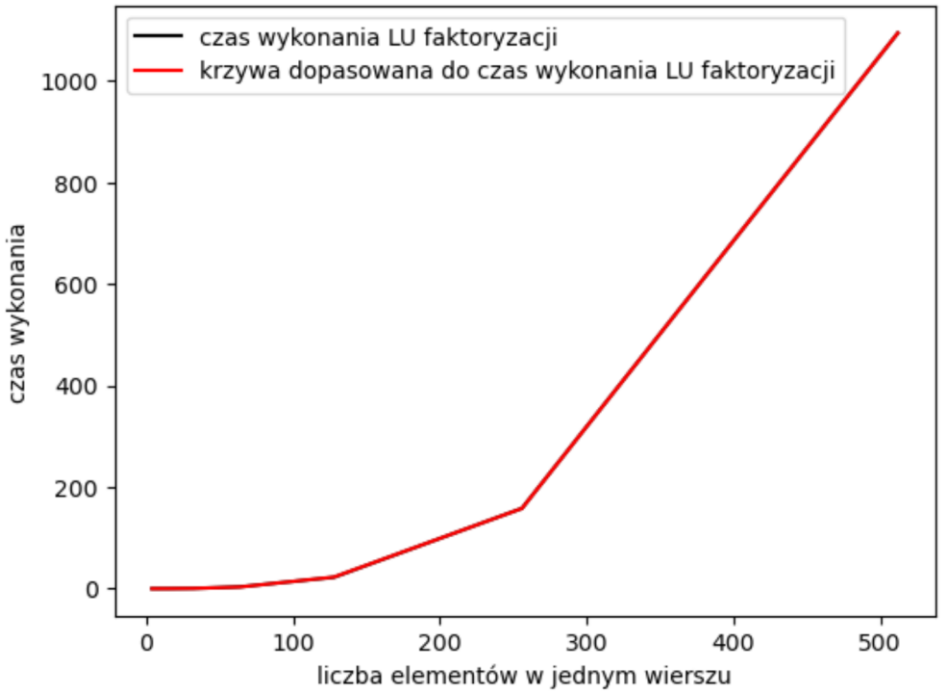
(4)

A na podstawie liczby operacji zmiennoprzecinkowych LU faktoryzacji otrzymaliśmy:

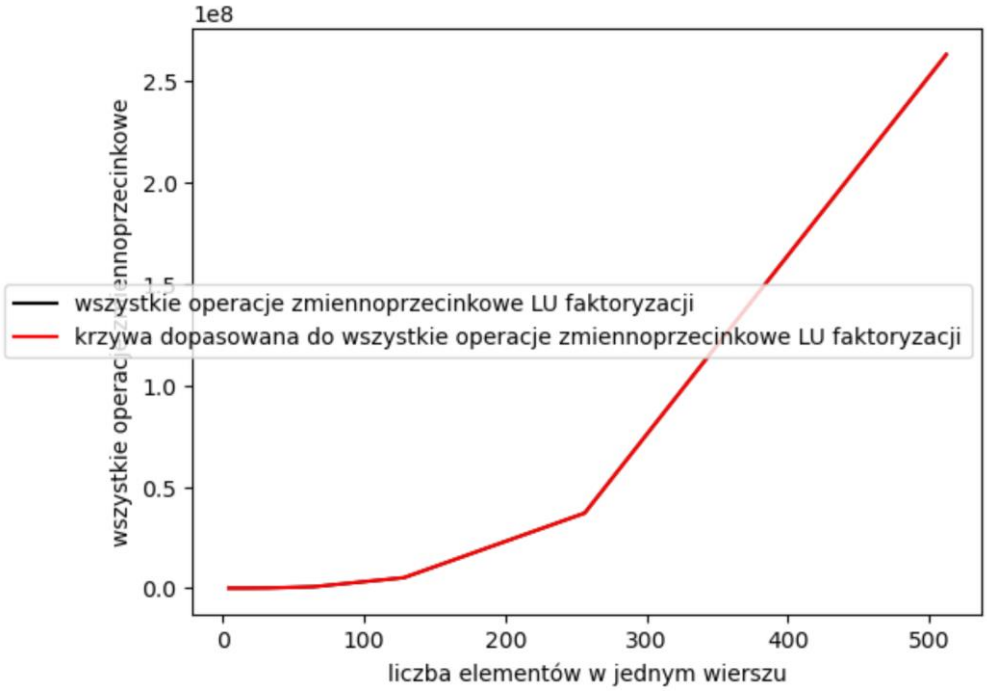
$$a = 5{,}675$$
$$k = 2{,}830$$

(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 pokrywają się ze sobą. Dodatkowo szacunki te pokrywają się z teoretyczną złożonością, która jest ograniczona mnożeniem macierzy. Ponieważ oparliśmy naszą LU faktoryzację na algorytmie mnożenia macierzy metodą Strassena, który ma złożoność $O(n^{2{,}807})$ to możemy zobaczyć że empiryczne szacunki złożoności pokrywają się z teoretycznymi.



Wykres nr. 5



Wykres nr. 6

Zależność liczby operacji zmiennoprzecinkowych od czasu wykonania

Patrząc na fakt, że złożoność wynikająca z czasu wykonania wyszła mniejsza niż ta z liczby operacji zmiennoprzecinkowych możemy wywnioskować, że tak jak dla algorytmu z analizowanego w podpunkcie 4.1.2 przechowywanie macierzy pogarszało złożoność tak tutaj jego czas skalował się lepiej niż czas wykonania operacji zmiennoprzecinkowych i dzięki temu otrzymaliśmy lepszy wynik.

4.3 Rekurencyjnego obliczania wyznacznika

4.3.1 Pomiary

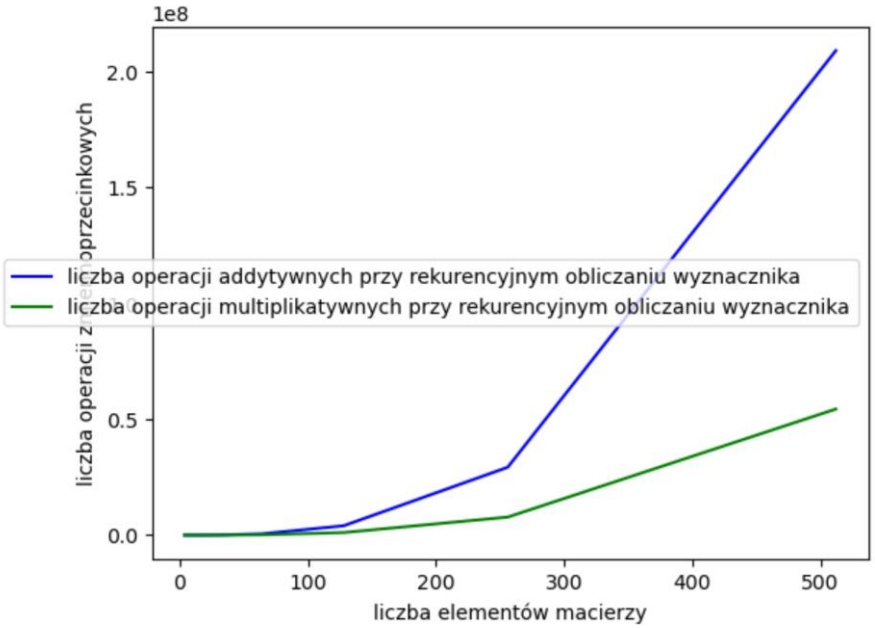
rozmiar	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
4	60	54	114	0.001177
8	832	438	1270	0.006409
16	8240	3228	11468	0.043480
32	69764	22866	92630	0.310992
64	543328	159924	703252	3.178278
128	4040148	1115538	5155686	21.205492
256	29269632	7784388	37054020	162.948553
512	208938868	54372066	263310934	1115.711659

Tab. 3 Wyniki pomiarów dla rekurencyjnego obliczania wyznacznika macierzy

4.3.2 Analiza wyników

Zależność operacji addytywnych od multiplikatywnych

Powyższą zależność przedstawia wykres numer 1. Można zauważyć, że operacji addytywnych jest znacznie więcej niż multiplikatywnych, z czego można wysnuć wniosek, że chcąc przyspieszyć działanie rekurencyjnego obliczania wyznacznika lepiej niskopoziomowo przyspieszać dodawanie.



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:

$$a = 3{,}241 \cdot 10^{-5}$$
$$k = 2{,}782$$

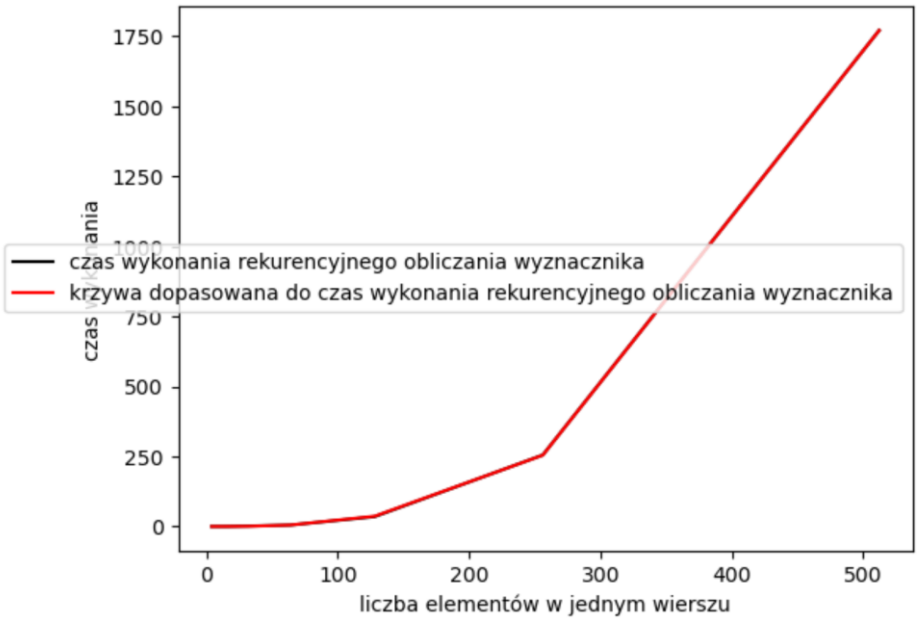
(6)

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

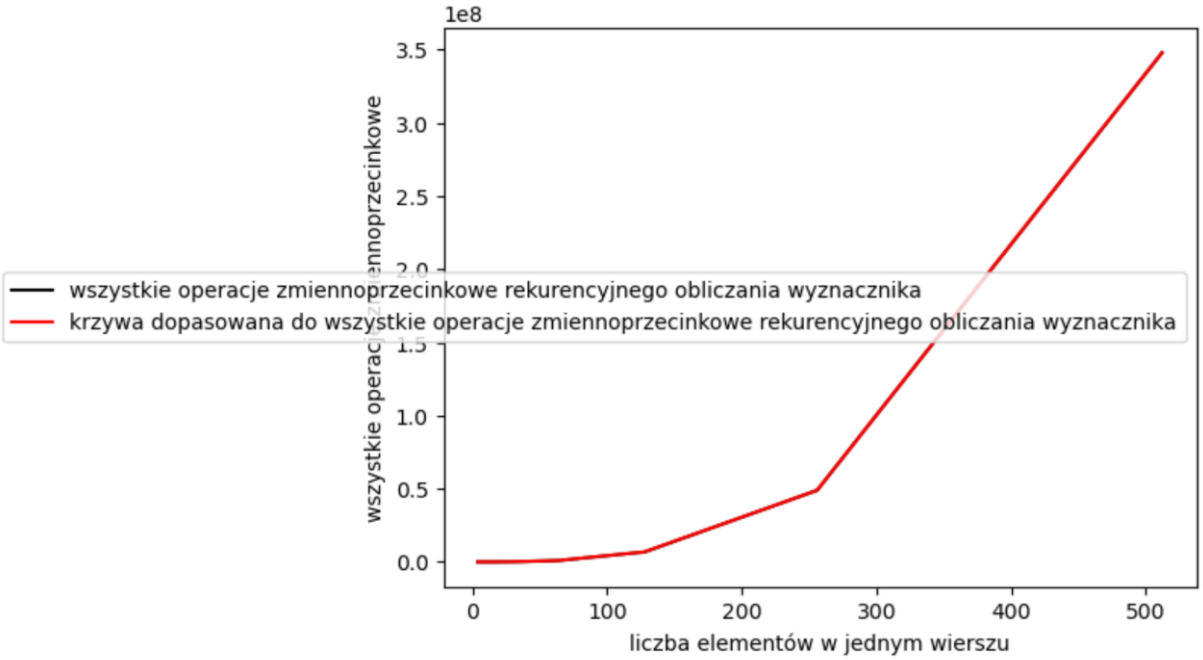
$$a = 5{,}675$$
$$k = 2{,}830$$

(7)

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 pokrywają się ze sobą. Dodatkowo szacunki te pokrywają się z teoretyczną złożonością, która jest ograniczona mnożeniem macierzy. Ponieważ oparliśmy rekurencyjne obliczanie wyznacznika na naszej implementacji LU faktoryzacji (która to z kolei jest oparta na algorytmie mnożenia macierzy metodą Strassena, który ma złożoność $O(n^{2,807})$) to możemy zobaczyć że empiryczne szacunki złożoności pokrywają się z teoretycznymi.



Wykres nr. 8

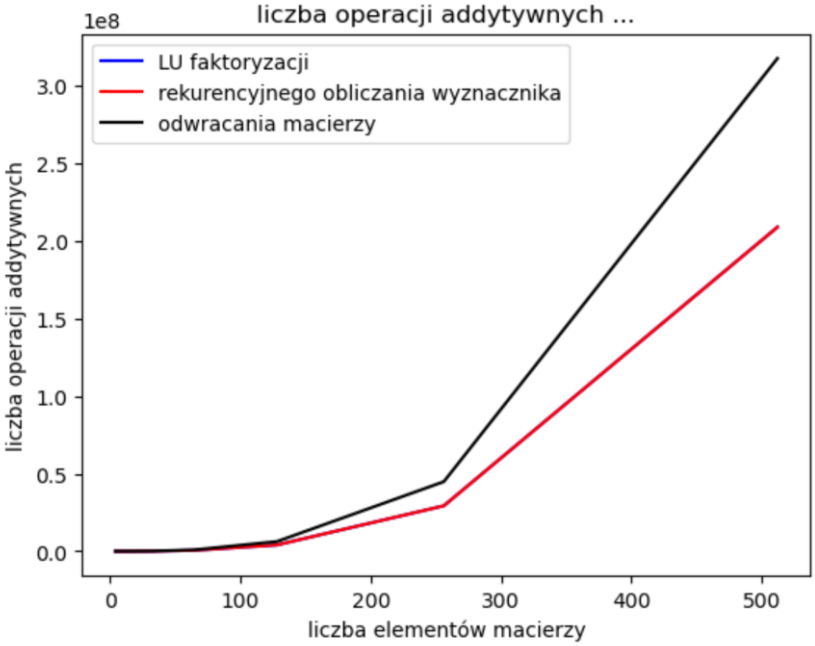


Wykres nr. 9

4.4 Porównanie wyników trzech powyższych algorytmów

4.4.1 Operacje addytywne

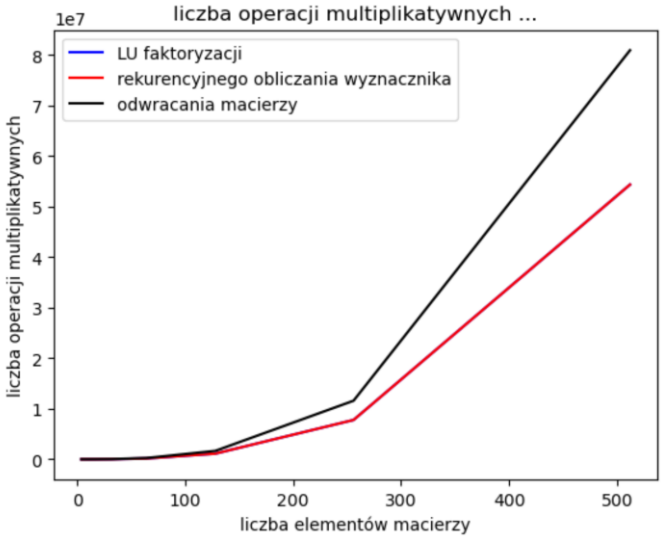
Wykres numer 10 przedstawia graficzne porównanie liczby operacji addytywnych dla wszystkich algorytmów zaimplementowanych przez nas w tym zadaniu. Możemy na nim dostrzec, że LU faktoryzacja wykonuje tyle samo operacji co rekurencyjne obliczanie wyznacznika (co nie jest zaskoczeniem ponieważ ten drugi algorytm mocno bazuje na LU faktoryzacji). Dodatkowo widzimy, że odwracanie macierzy potrzebuje więcej operacji addytywnych niż LU faktoryzacja.



Wykres nr. 10

4.4.2 Operacje multiplikatywne

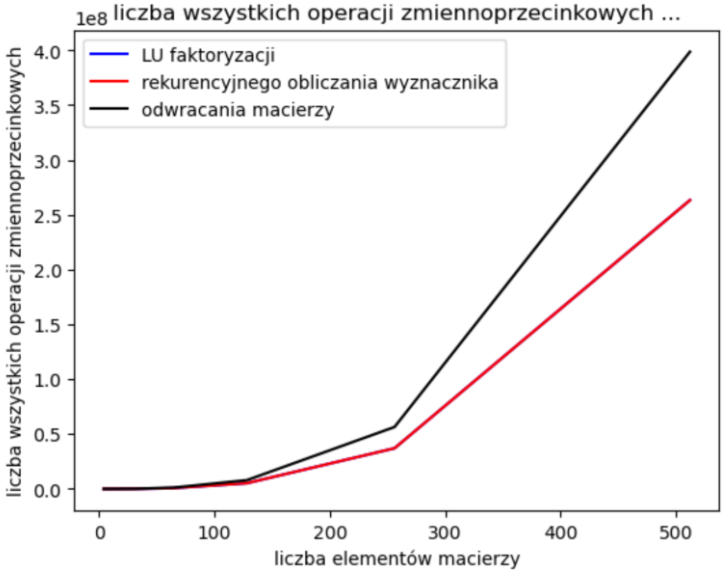
Wykres numer 11 przedstawia graficzne porównanie liczby operacji multiplikatywnych dla wszystkich algorytmów zaimplementowanych przez nas w tym zadaniu. Możemy na nim dostrzec, że podobnie jak w podpunkcie 4.4.1, LU faktoryzacja wykonuje tyle samo operacji co rekurencyjne obliczanie wyznacznika (co nie jest zaskoczeniem ponieważ ten drugi algorytm mocno bazuje na LU faktoryzacji). Dodatkowo widzimy, że odwracanie macierzy potrzebuje więcej operacji addytywnych niż LU faktoryzacja.



Wykres nr.11

4.4.3 Wszystkie operacje zmiennoprzecinkowe

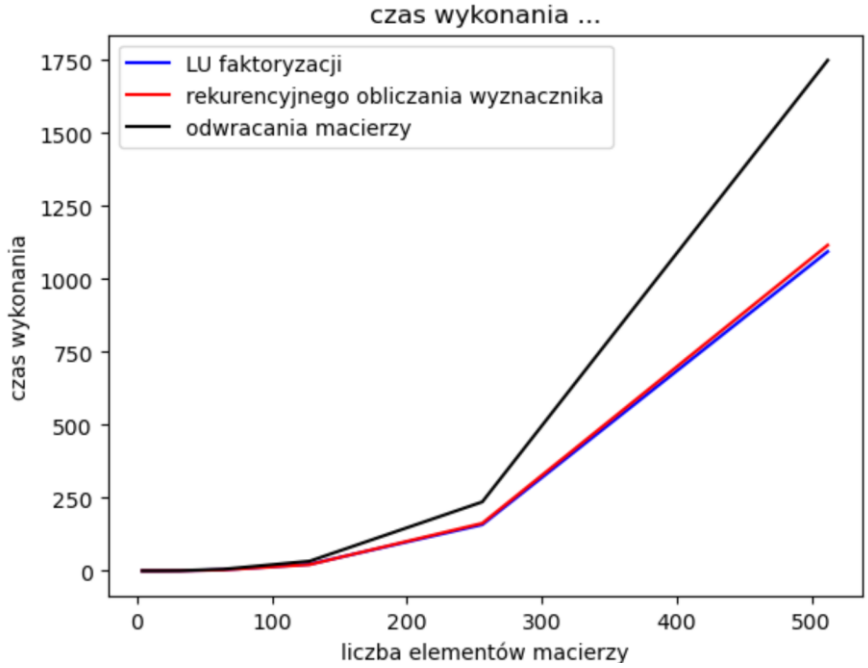
Wykres numer 11 przedstawia graficzne porównanie liczby operacji multiplikatywnych dla wszystkich algorytmów zaimplementowanych przez nas w tym zadaniu. Możemy na nim dostrzec, że podobnie jak w podpunkcie 4.4.1, LU faktoryzacja wykonuje tyle samo operacji co rekurencyjne obliczanie wyznacznika (co nie jest zaskoczeniem ponieważ ten drugi algorytm mocno bazuje na LU faktoryzacji). Dodatkowo widzimy, że odwracanie macierzy potrzebuje więcej operacji addytywnych niż LU faktoryzacja.



Wykres nr. 12

4.4.4 Czas działania

Wykres numer 13 przedstawia graficzne porównanie czasu działania dla wszystkich algorytmów z tego zadania. Na nim możemy dostrzec, że najlepiej wypadają po kolei algorytm: LU faktoryzacji, rekurencyjnego obliczania wyznacznika i odwracania macierzy. Takie wyniki nie są zaskoczeniem ponieważ wyznacznik obliczamy na podstawie LU faktoryzacji. W naszej implementacji, uzyskawszy macierz U obliczamy wyznacznik na podstawie jej przekątnej. Te operacje dają jak widać narzut na czas wykonania (na liczbie operacji zmiennoprzecinkowych nie widzieliśmy zbytnio tego efektu)



Wykres nr. 13

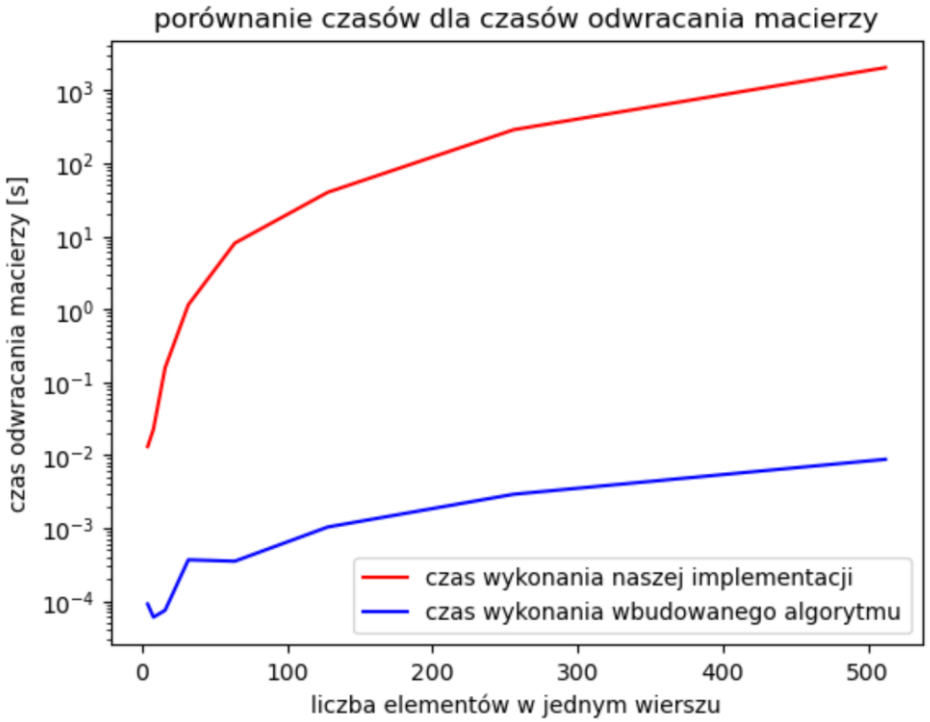
4.5 Porównanie z numpy

4.5.1 Porównanie czasu wykonania

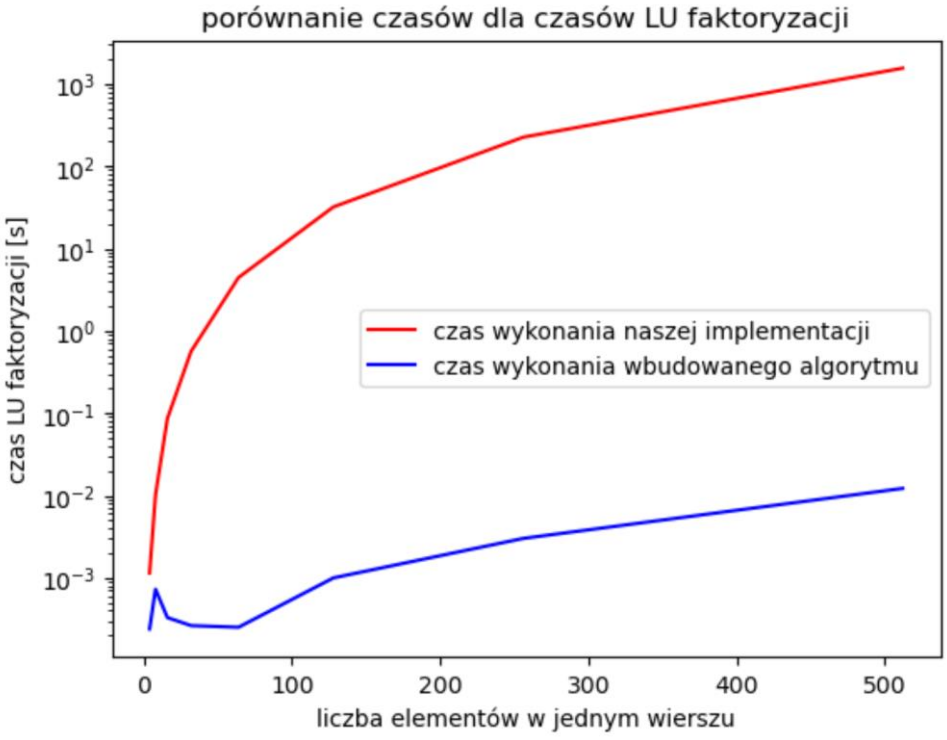
Na poniższych wykresach (14-16) przedstawiliśmy porównanie naszych implementacji z istniejącymi już w bibliotekach do pythona implementacjami, odpowiednio np.linalg.inv, scipy.linalg.lu, np.linalg.det. Implementacje te w rzeczywistości są napisane w językach C i C++ co czyni je porównywalne do języków przeznaczonych do mnożenia macierzy takich jak Matlab czy Octave.

Trzy poniższe wykresy mają na osi Y skalę logarytmiczną, ponieważ w innym wypadku nie bylibyśmy w stanie wyczytać z nich niczego poza tym, że się sporo różnią.

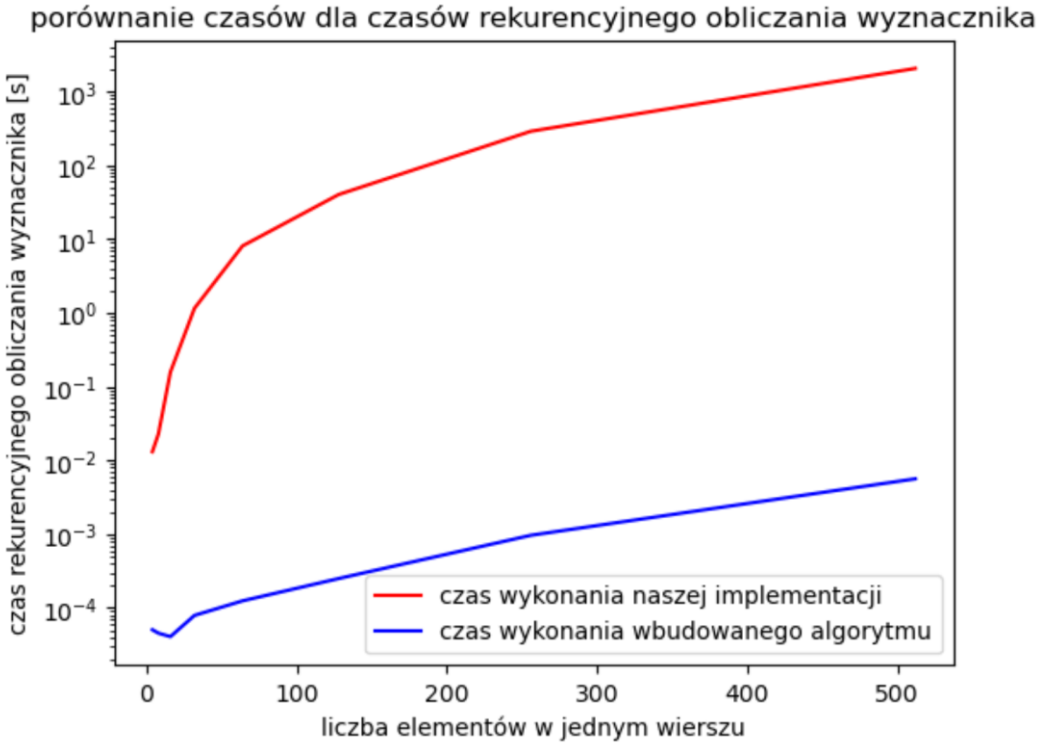
Zauważmy, dla każdego z algorytmów, jak bardzo się różnią od siebie te wykresy. Czasy wykonania algorytmów wbudowanych to ułamki sekund, w momencie kiedy nasze implementacje sięgają nawet czasów 30- kilku minut.



Wykres 14



Wykres 15



Wykres 16

4.5.2 Sprawdzenie poprawności

Ze względu na trudności z porównywaniem większych macierzy, sprawdzaliśmy poprawności algorytmu rekurencyjnego odwracania macierzy na macierzach rozmiaru 4x4. Jak widać na poniższym zrzucie ekranu z wyników porównania, nasze odwracanie macierzy działa poprawnie.

```
wynik numpy
[[-2.66627686 -2.78886994  2.45530699  3.27617722]
 [ 0.78643566 -1.89182589  2.18335355 -0.37777623]
 [ 4.1973028   4.62412877 -5.25306066 -2.58262953]
 [-3.745935   -1.30027207  3.01083834  1.73736948]]
wynik naszego odwracania
[[-2.6662768630404567 -2.788869944446736 2.4553069858462777
  3.2761772179235784]
 [0.786435657389644 -1.8918258892515931 2.1833535482731534
  -0.37777623342050504]
 [4.1973028022140895 4.62412877115851 -5.253060659923591
  -2.582629528893925]
 [-3.7459350018396975 -1.3002720672553085 3.010838339384048
  1.7373694809623903]]
```

Rys. 1 poprawność odwracania macierzy

Sprawdzenie poprawności LU faktoryzacji było dużo trudniejsze ponieważ wiele bibliotek zwraca te dwie macierze w różnych konwencjach. Zatem zdecydowaliśmy, że najlepszym sposobem na sprawdzenie LU faktoryzacji będzie porównanie wyznacznika obliczonego na podstawie LU faktoryzacji naszej implementacji. Jak widać na poniższym rysunku nr 2. Nasza implementacja zachowuje się poprawnie.

```
wynik numpy
0.04697805916786064
wynik wyzancznika na podstawie naszej LU faktoryzacji
0.04697805916786069

--------------------
wynik numpy
-0.002908248179225858
wynik wyzancznika na podstawie naszej LU faktoryzacji
-0.0029082481792261617

--------------------
wynik numpy
0.10446398663137366
wynik wyzancznika na podstawie naszej LU faktoryzacji
0.10446398663137162

--------------------
wynik numpy
-12.154752125197827
wynik wyzancznika na podstawie naszej LU faktoryzacji
-12.154752125030502
```

Rys.2 sprawdzenie poprawności LU faktoryzacji naszej implementacji. Kolejne logi to kolejne rozmiary macierzy, od 4x4 do 32x32

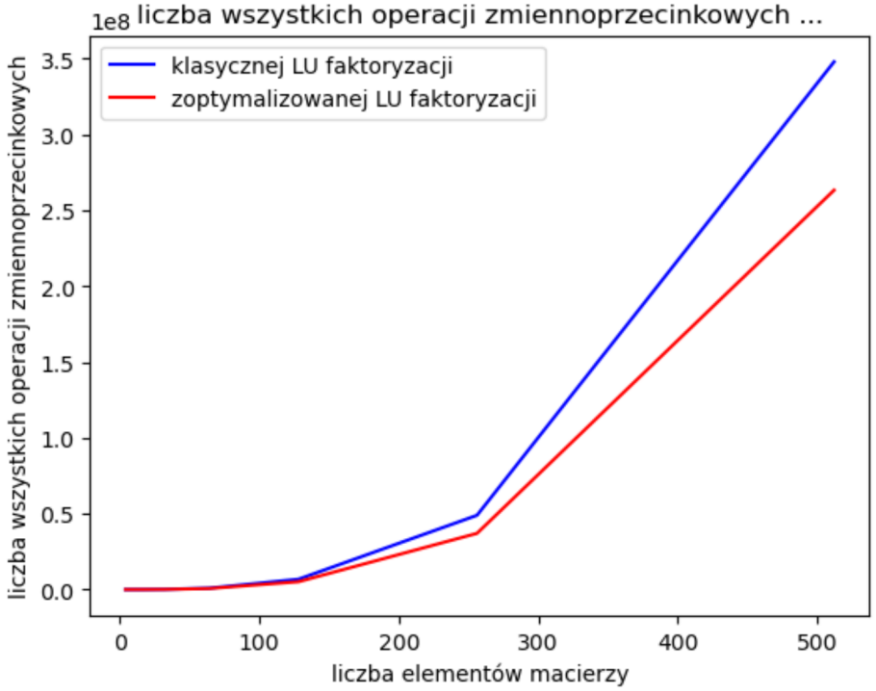
4.6 Optymalizacje LU faktoryzacji

Dzięki zastosowaniu optymalizacji polegających na sprawniejszym obliczaniu macierzy odwrotnej do trójkątnej górnej lub dolnej, uzyskaliśmy lepsze wyniki, które przedstawione są w tabeli nr 2. Dla porównania poniżej zamieszczamy pomiary LU faktoryzacji bez stosowania tej optymalizacji.

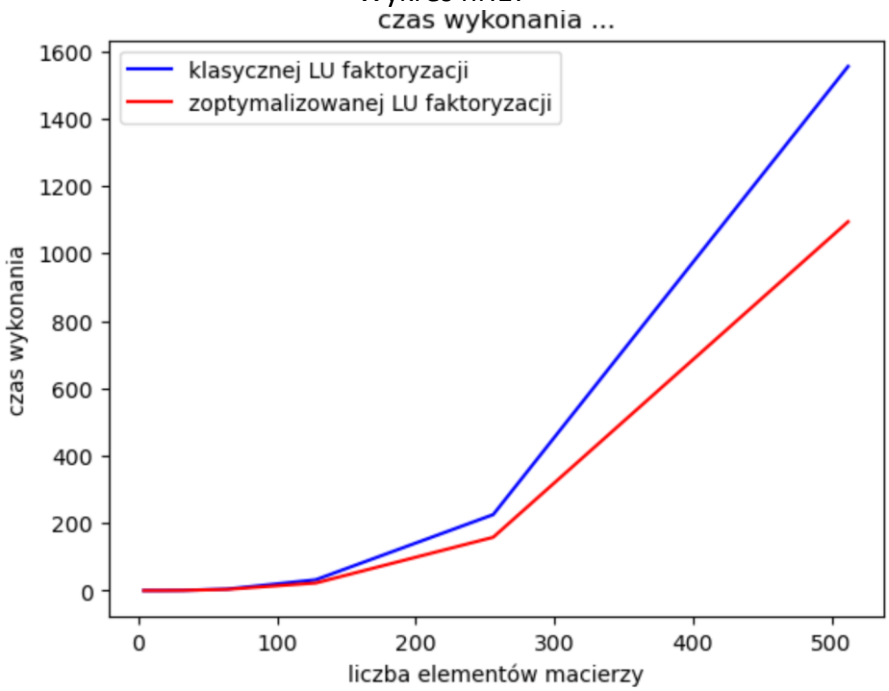
rozmiar	operacje addytywne	operacje multiplikatywne	wszystkie operacje zmiennoprzecinkowe	czas wykonania
4	62	69	131	0.001145
8	1018	587	1605	0.010178
16	10576	4341	14917	0.086105
32	90962	30707	121669	0.565409
64	712536	214533	927069	4.449985
128	5310786	1495715	6806501	32.035126
256	38514632	10435317	48949949	225.179765
512	275067634	72883379	347951013	1554.568073

Tab 4. Pomiary LU faktoryzacji bez stosowania optymalizacji

Najłatwiej będzie zobaczyć poprawę powodowaną przez tę optymalizację na graficznych przedstawieniach danych.



Wykres nr.17



Wykres nr.18

Na obu powyższych wykresach widać, że optymalizacja odwracania macierzy rzeczywiście poprawia zdolności LU faktoryzacji, zarówno jeśli chodzi o czas wykonania jak i liczbę operacji zmiennoprzecinkowych

5. Wnioski

- Każdy z implementowanych przez nas algorytmów był zależny od algorytmu mnożenia macierzy na którym go oparliśmy.
- Ze względu na fakt, że wybraliśmy do każdego metodę Strassena, każdy z algorytmów zaimplementowanych przez nas cechował się większą liczbą operacji addytywnych od multiplikatywnych, a złożoność obliczeniowa policzona przez nas

empirycznie dla każdego oscylowała niedaleko ($O^{2,807}$) co jest złożonością metody Strassena.

- Czasy działania naszych implementacji są nieporównanie dłuższe od czasów wykonania tych samych algorytmów wbudowanych w pythona. Z tego względu samodzielne implementowanie tych algorytmów w pythonie nie ma innego sensu oprócz dydaktycznego.
- Optymalizacje odwracania macierzy polegające na nieobliczaniu fragmentów macierzy trójkątnych które i tak zawsze są zerami rzeczywiście przyspieszają wykonanie LU faktoryzacji (ponieważ ta zawsze korzysta z macierzy trójkątnych)