Algorytmy macierzowe
Sprawozdanie z laboratorium nr. 2

Szymon Nowak-Trzos, Dominik Adamczyk

29 listopada 2023

1 Treść zadania

Zadanie polegało na zaimplementowaniu u porównaniu trzech algorytmów:

- Rekurencyjne odwracanie macierzy
- Rekurencyjna LU faktoryzacja
- Rekurencyjne obliczanie wyznacznika

2 Wstęp

Zgodnie z treścią, zaimplementowane zostały wszystkie wskazane algorytmy. Całe zadanie zostało napisane w języku Python 3.10. Testy przeprowadzono na urządzeniu z cpu Intel Core i5-1135g7 o bazowej częstotliwości 2.4GHz.

3 Użyte rozwiązania

Aby zliczyć ilość operacji addytywnych i multiplikatywnych stworzona została nowa klasa Num() dziedzicząca po Python'owej klasie float. Umożliwiło nam to wykonywanie zliczanie ilości operacji w liczniku zawartym w tej klasie.

```
class Num(float):
    counts = Counter(mul=0, add=0)
    def __add__(self, other):
        self.counts["add"] += 1
        return Num(super(Num, self).__add__(other))

    def __radd__(self, other):
        self.counts["add"] += 1
        return Num(super(Num, self).__radd__(other))

    def __mul__(self, other):
        self.counts["mul"] += 1
        return Num(super(Num, self).__mul__(other))

    ### Other overloaded methods

    def reset_counter(self):
        self.counts["mul"] = 0
        self.counts["add"] = 0
```

Dodatkowo zaimplementowane zostały funkcje pomocnicze do dzielenia macierzy na części oraz sklejania macierzy z tych części.

```
def matrix_partition(A):
    n = A.shape[0] // 2
    m = A.shape[1] // 2
    A11 = A[:n, :m]
    A12 = A[:n, m:]
    A21 = A[n:, :m]
    A22 = A[n:, m:]
    return A11, A12, A21, A22

def matrix_repartition(A11, A12, A21, A22):
    C = np.block([[A11, A12],
                  [A21, A22]])
    return C
```

Algorytmy zostały zaimplementowane w osobnych funkcjach. Do mnożenia macierzy użyto mnożenia z biblioteki numpy w celach przyśpieszenia obliczeń.

4 Algorytmy

Poniżej prezentujemy opisy i kody implementowanych algorytmów. Kod Python'a służy również za pseudokod. Wszystkie algorytmy działają dla dowolnych macierzy kwadratowych.

4.1 Rekurencyjne odwracanie macierzy

Jako pierwszy został zaimplementowany algorytm rekurencyjnego odwracania macierzy. Wykorzystywany jest on też w następnych algorytmach.

Algorytm ten jest wyprowadzony wprost z odwracania macierzy eliminacją Gaussa.

```
    def inverse(A):
        if A.shape[0] == 1:
            if A[0][0] != 0:
                return np.array([[Num(1)/A[0][0]]], dtype=Num)
            else:
                raise ValueError("Matrix not Invertible")
        A11, A12, A21, A22 = matrix_partition(A)
        A11_i = inverse(A11)
        A2111_i = A21 @ A11_i
        S22 = A22 - A2111_i @ A12
        S22_i = inverse(S22)
        B11 = A11_i @ (np.eye(A11_i.shape[0], dtype=Num) + A12 @ S22_i @ A2111_i)
        B12 = -A11_i @ A12 @ S22_i
        B21 = -S22_i @ A2111_i
        B22 = S22_i
        return matrix_repartition(B11, B12, B21, B22)
```

4.2 Rekurencyjna LU faktoryzacja

```
    def LU(A):
        if A.shape[0] == 1:
            return np.array(A, dtype=Num), np.array([[1]], dtype=Num)
        A11, A12, A21, A22 = matrix_partition(A)
        L11, U11 = LU(A11)
        U11_i = inverse(U11)
        L21 = A21 @ U11_i
        L11_i = inverse(L11)
        U12 = L11_i @ A12
        S = A22 - L21 @ U12
        Ls, Us = LU(S)
        U22 = Us
        L22 = Ls
        return [matrix_repartition(L11, np.zeros((L11.shape[0], L22.shape[1]), dtype=
                                                 Num), L21, L22),
                matrix_repartition(U11, U12, np.zeros((U22.shape[0], U11.shape[1]),
                                                      dtype=Num), U22)]
```

4.3 Rekurencyjne obliczanie wyznacznika

Obliczanie wyznacznika macierzy sprowadza się do przemnożenia komórek na głównej przekątnej macierzy L po dekompozycji LU macierzy A.

```
def determinant(A):
    L, U = LU(A)
    return np.prod(np.diag(L))
```

5 Sprawdzanie poprawności zaimplementowanych algorytmów

Ze względu na wykorzystanie operacji zmiennoprzecinkowych, do porównywania macierzy wynikowej skorzystaliśmy z funkcji allclose z biblioteki numpy.

```
def compare(A, B):
    return np.allclose(np.array(A, dtype=float), np.array(B, dtype=float),
                                rtol=1e-05, atol=1e-08, equal_nan=False)
```

5.1 Sprawdzanie odwracania macierzy

Wynik naszego algorytmu odwracania macierzy jest porównywany z wynikiem odwracania macierzy algorytmem z biblioteki numpy (np.linalg.inv).

```
def check_inverse(A):
    return compare(inverse(A), np.linalg.inv(np.array(A, dtype=float)))
```

5.2 Sprawdzanie LU faktoryzacji

LU faktoryzacja sprawdzana jest przez porównanie czy A = L @ U, oraz czy L to macierz dolna trójkątna a U to macierz górna trójkątna. Do obu tych rzeczy wykorzystana została biblioteka numpy.

```
def check_LU(A):
    L, U = LU(A)
    return compare(L, np.tril(L)) and compare(U, np.triu(U)) and compare(A, L @ U)
```

5.3 Sprawdzanie obliczania wyznacznika

Wynik obliczania wyznacznika naszym algorytmem porównywany jest z wynikiem funkcji z biblioteki numpy (np.linalg.det).

```
def check_determinant(A):
    return compare(determinant(A), np.linalg.det(np.array(A, dtype=float)))
```

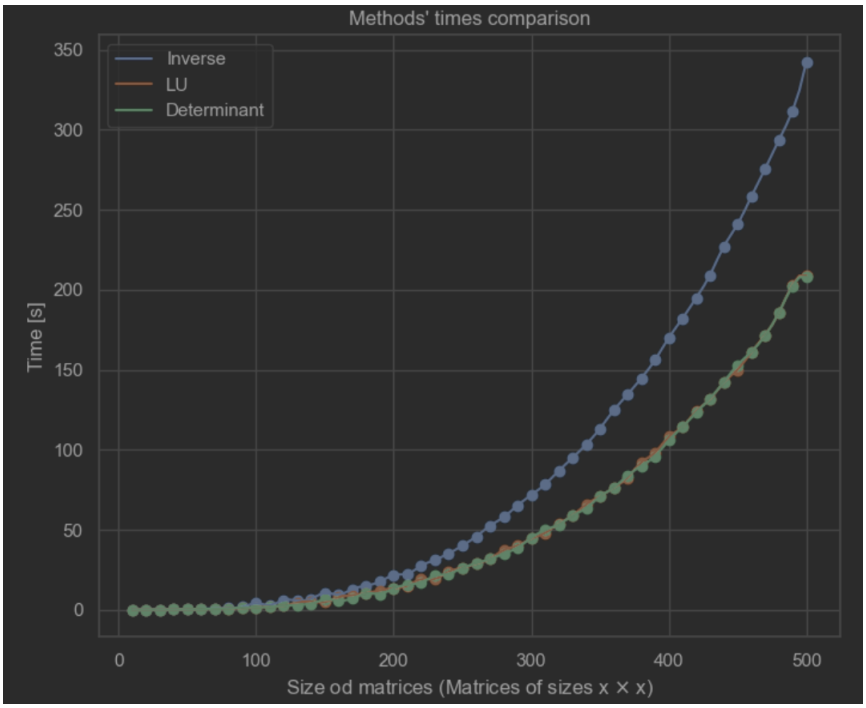
5.4 Poprawność na różnych macierzach

Sprawdzona została poprawność naszych algorytmów dla różnych rozmiarów macierzy. Fragment wyników znajduje się w tabeli poniżej.

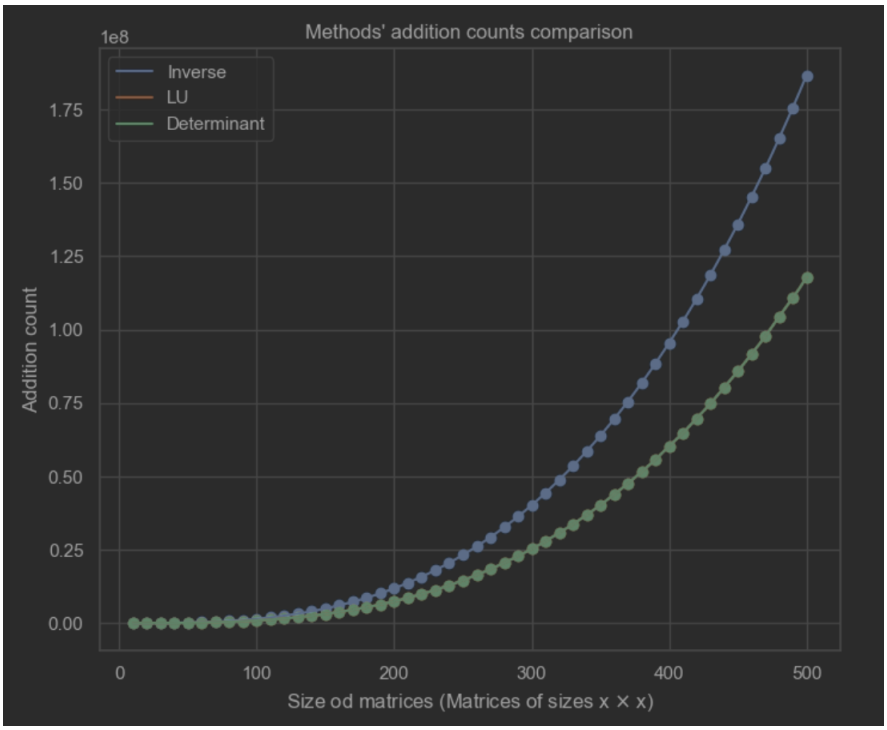
	Inverse	LU	Determinant
8	True	True	True
9	True	True	True
10	True	True	True
11	True	True	True
12	True	True	True
13	True	True	True
14	True	True	True
15	True	True	True

Tabela 1: Tabela poprawności algorytmów

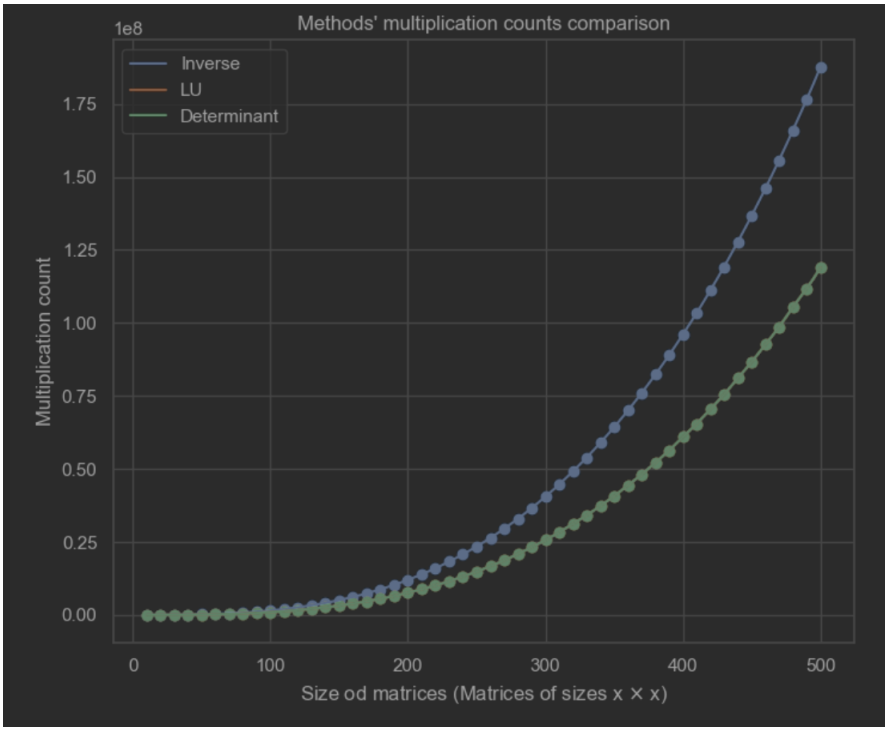
6 Wykresy



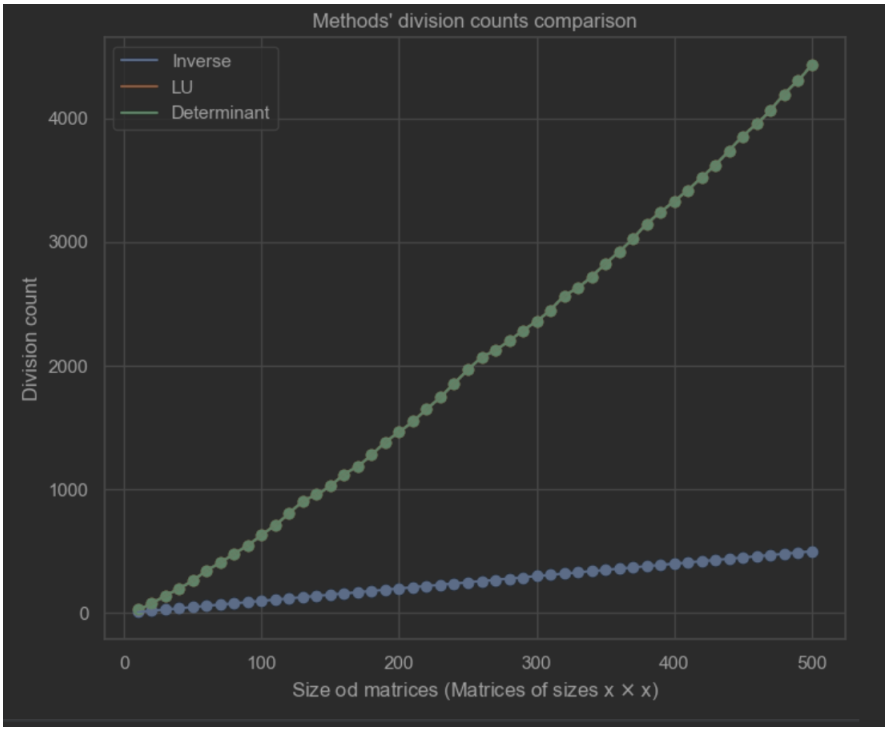
Wykres 2: Czas wykonywania algorytmu



Wykres 3: Ilość operacji addytywnych



Wykres 4: Ilość mnożeń



Wykres 5: Ilość dzieleń

7 Obliczanie złożoności obliczeniowej algorytmów

Złożoność obliczeniową dla naszych algorytmów wyliczyliśmy empirycznie korzystając z funkcji ***curve_fit*** z scipy.optimize. Dopasowywaliśmy współczynniki funkcji $f(x) = a * x^b$

```
from scipy.optimize import curve_fit
from scipy.stats import linregress

def func_exp(x, a, b):
    return a * np.power(x, b)

def fit_to_df(df, line=False):
    coeffs = {}
    for method in df.columns:
        x = df.index
        y = df[method]
        y = y.dropna(how='all')
        x = x.dropna(how='all')[:len(y)]


        if line:
            slope, intercept, r, p, se = linregress(x, y)
            popt = [slope, intercept]
        else:
            popt, _ = curve_fit(func_exp, x, y)
        coeffs[method] = popt
    return coeffs
```

7.1 Rekurencyjne odwracanie macierzy

7.1.1 Złożoność obliczeniowa

$$O(n) \approx (1.496) * n^{(3.000)}$$

7.1.2 Złożoność czasowa

$$O(n) \approx (2.033\text{e}{-6}) * n^{(3.044)}$$

7.2 Rekurencyjna LU faktoryzacja

7.2.1 Złożoność obliczeniowa

$$O(n) \approx (0.935) * n^{(3.000)}$$

7.2.2 Złożoność czasowa

$$O(n) \approx (1.300\text{e}{-6}) * n^{(3.040)}$$

7.3 Rekurencyjne obliczanie wyznacznika

7.3.1 Złożoność obliczeniowa

$$O(n) \approx (0.935) * n^{(3.000)}$$

7.3.2 Złożoność czasowa

$$O(n) \approx (1.113\text{e}{-6}) * n^{(3.066)}$$

7.4 Dopasowanie krzywej do ilości dzieleń

Dla ilości dzieleń próbowaliśmy oszacować złożoność dwoma metodami - korzystając z scipy.curve_fit oraz z regresji liniowej. Dla regresji liniowej dostaliśmy następujące wyniki:

- Odwracanie macierzy:
$$O(n) = n$$
- LU faktoryzacja oraz obliczanie wyznacznika:
$$O(n) \approx 9n - 258$$

Korzystając z scipy.curve_fit dostaliśmy następujące wyniki:

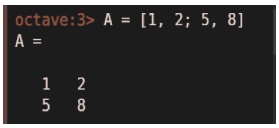
- Odwracanie macierzy:
$$O(n) = n$$
- LU faktoryzacja oraz obliczanie wyznacznika:
$$O(n) \approx (2.602) * n^{(1.195)}$$

Używając regresji liniowej mamy o rząd wielkości większy RSS niż przy scipy.curve_fit.

8 Porównanie wyników z Octave

Przez brak unikalności dekompozycji LU nie porównujemy wyników z wynikami z Octave'a.

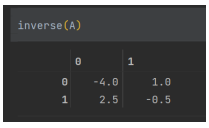
(a) Macierz w Python'ie



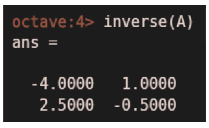
(b) Macierz w Octave

Wykres 6

8.1 Odwracanie macierzy



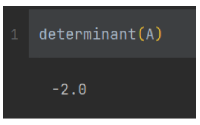
(a) Macierz odwrotna w Python'ie



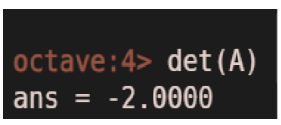
(b) Macierz odwrotna w Octave

Wykres 7

8.2 Wyznacznik macierzy



(a) Wyznacznik macierzy w Python'ie



(b) Wyznacznik macierzy w Octave

Wykres 8

9 Wnioski

Zaimplementowane algorytmy mają w przybliżeniu złożoność obliczeniową i czasową $O(n^3)$, co jest oczekiwanym rezultatem, ze względu na użytą metodę mnożenia macierzy. Testy poprawności nie wykazały błędów w implementacji algorytmów. Czasy wykonania algorytmów LU faktoryzacji i obliczania wyznacznika są praktycznie takie same. Wynika to z faktu, że drugi algorytm w polega w pełni na działaniu pierwszego.