Algorytmy Macierzowe
Sprawozdanie 2
Rekurencyjne mnożenie macierzy

Przemek Węglik
Szymon Paszkiewicz

23 listopada 2022

Spis treści

1 Rekurencyjne odwracanie macierzy

1.1 Opis algorytmu

Algorytm dzieli macierz na 4 podmacierze, następnie wykonuje na każdej macierzy odpowiednie operacje:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \tag{1}$$

$$A_{1,1}^{-1} = inverse(A_{1,1}) \tag{2}$$

$$S^{-1} = inverse(A_{2,2} - A_{2,1} * A_{1,1}^{-1} * A_{1,2}) \tag{3}$$

$$A^{-1} = \begin{bmatrix} A_{1,1}^{-1}(I + A_{1,2}S^{-1} * A_{2,1} * A_{1,1}^{-1}) & -A_{1,1}^{-1} * A_{1,2} * S \\ -S * A_{2,1} * A_{1,1}^{-1} & S \end{bmatrix} \tag{4}$$

gdzie:

A - macierz, którą chcemy odwrócić,

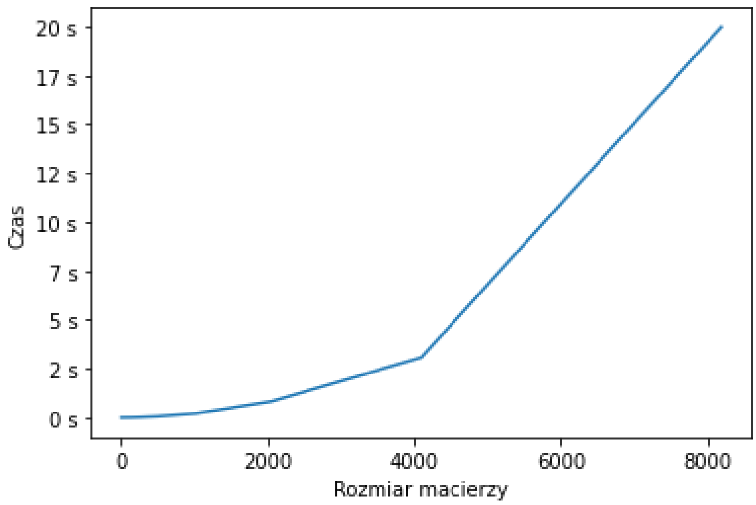
inverse - funkcja rekurencyjna odwracająca macierz,

S - macierz pomocnicza.

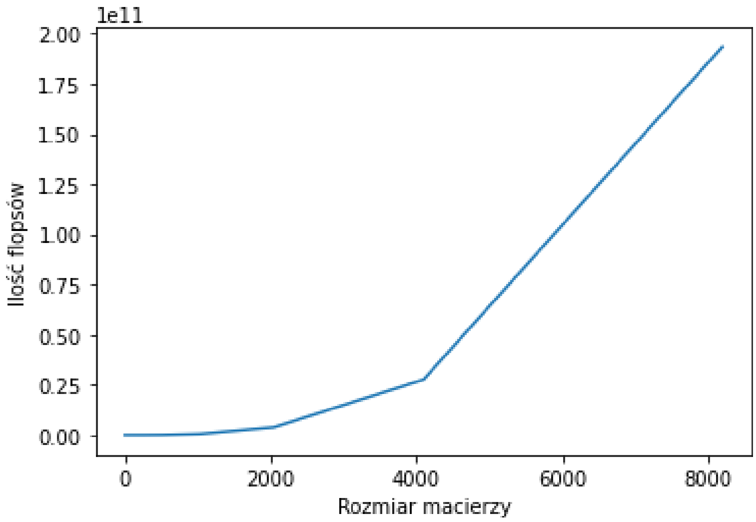
1.2 Pseudo-kod

```
def inverse(A):
    if A.size > 1:
        split_at = A.shape[0] // 2
        A11, A12, A21, A22 = split(A, split_at)

        A11_inv = inverse(A11)
        S22 = A22 + A21 * A11_inv * A12
        S22_inv = inverse(S22)

        B11 = A11_inv * (I + A12 * S22_inv * A21 * A11_inv)
        B12 = -A11_inv * A12 * S22_inv
        B21 = -S22_inv * A21 * A11_inv
        B22 = S22_inv

        return (
            [[B11, B12]
            [B21, B22]]
        )
    else:
        return 1/A
```

1.3 Benchmarki



Rysunek 1: Wykres czasu od rozmiaru macierzy.



Rysunek 2: Wykres libczby operacji od rozmiaru macierzy.

1.4 Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu zależy od algorytmu mnożenia macierzy jaki został wykorzytany do obliczeń. W naszym przypadku był to algorytm Strassen'a, którego złożoność obliczeniowa wynosi: $O(n^{2.807})$.

1.5 Porównanie rozwiązania

Wyniki naszego algorytmu:

```
%% INPUT
A = np.array([[1, 8, 5], [2, 4, 6], [3, 5, 7]])
print(recursive_matrix_inverse(A, counter))

%% OUTPUT

[[-0.1   -1.55   1.4 ]
 [ 0.2   -0.4    0.2 ]
 [-0.1    0.95  -0.6 ]]
```

Wyniki uzyskane w numpy'u:

```
%% INPUT
A = np.array([[1, 8, 5], [2, 4, 6], [3, 5, 7]])
print(np.linalg.inv(A))

%% OUTPUT

[[-0.1   -1.55   1.4  ]
 [ 0.2   -0.4    0.2  ]
 [-0.1    0.95  -0.6  ]]
```

2 Rekurencyjna faktoryzacja LU

2.1 Opis algorytmu

Algorytm dzieli macierz na 4 podmacierze, a następnie przy ich użyciu wykonuje poniższe operacje.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \tag{5}$$

$$[L_{1,1}, U_{1,1}] = LU(A_{1,1}) \tag{6}$$

$$[L_s, U_s] = LU(A_{2,2} - A_{2,1}U_{1,1}^{-1}L_{1,1}^{-1}A_{1,2}) \tag{7}$$

$$L = \begin{pmatrix} L_{1,1} & 0 \\ A_{2,1}U_{1,1}^{-1} & L_s \end{pmatrix} \tag{8}$$

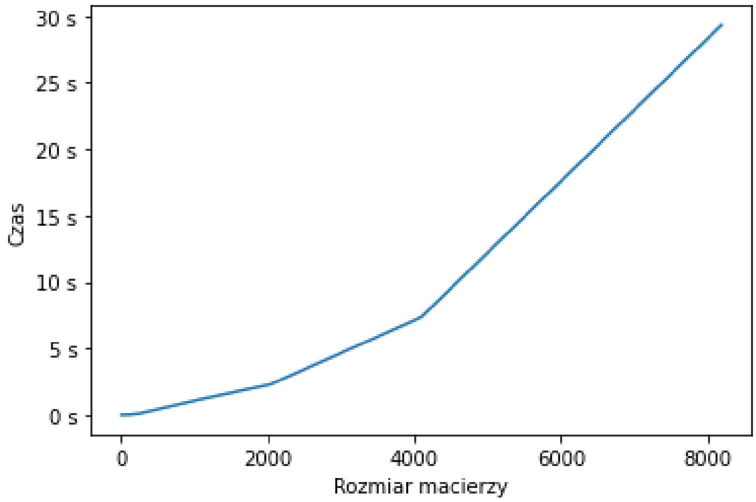
$$U = \begin{pmatrix} U_{1,1} & L_{1,1}^{-1}A_{1,2} \\ 0 & U_s \end{pmatrix} \tag{9}$$

2.2 Pseudo-kod

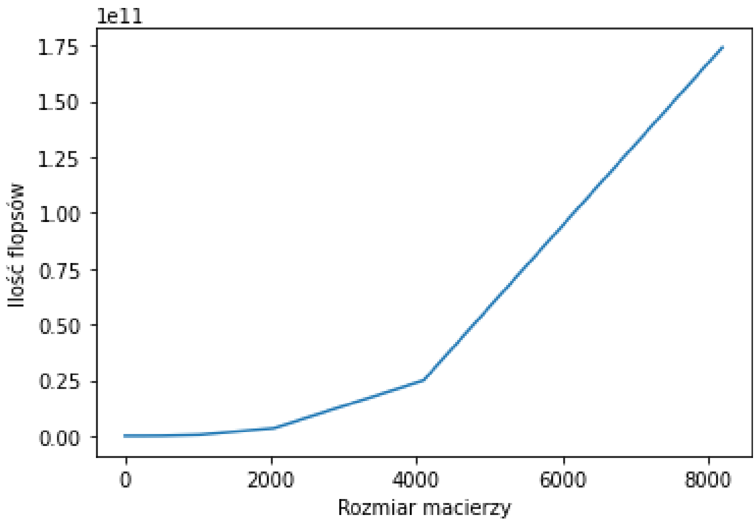
```
def LU(A):
    if A.size > 1:
        split_at = A.shape[0] // 2
        A11, A12, A21, A22 = split(A, split_at)

        L11, U11 = LU(A11)
        U11_inv = inverse(U11)

        L21 = A21 *U11_inv
        L11_inv = inverse(L11)

        U12 = L11_inv * A12
        L22 = A22 + -A21 * U11_inv * L11_inv * A12

        Ls, Us = LU(L22)
        U22 = Us
        L22 = Ls

        return(
            [[L11, 0]
            [L21, L22]],
```

```
            [[U11, U12]
            [0, U22]]
            )
    else:
        return (1, A[0, 0])
```

2.3 Benchmarki

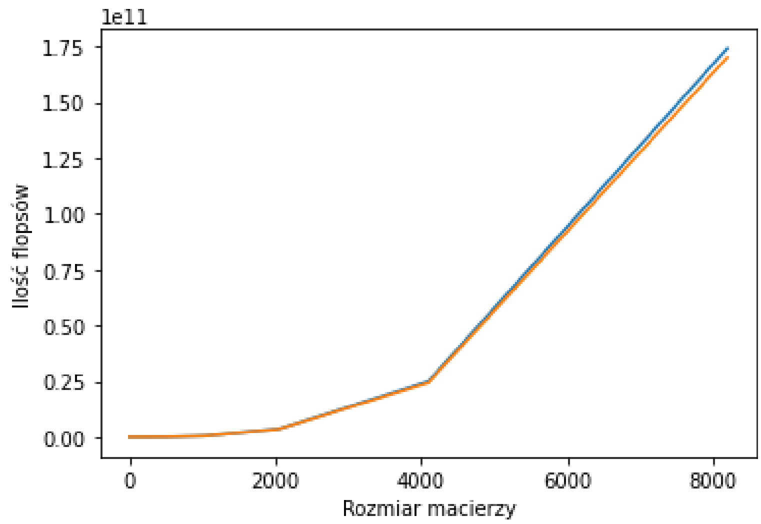


Rysunek 3: Wykres czasu od rozmiaru macierzy.



Rysunek 4: Wykres libczby operacji od rozmiaru macierzy.

2.4 Złożoność obliczeniowa



Rysunek 5: Wykres liczby operacji od rozmiaru macierzy z naniesioną dodatkowo krzywą y = 1.8 $x^{2.807}$

Złożoność obliczeniowa faktoryzacji LU to złożoność zastosowanego algorytmu mnożenia macierzy $O(n^{2.807})$.

2.5 Porównanie rozwiązania

```
Wyniki uzyskane przez nasze implementacje:
%% INPUT
A = np.array([[1, 8, 5], [2, 4, 6], [3, 5, 7]])
L, U = recursive_lu(A, counter)
print(L)
print(U)
%% OUTPUT
[[1.          , 0.          , 0.          ],
 [2.          , 1.          , 0.          ],
 [3.          , 1.58333333, 1.          ]]

[[  1.          ,   8.          ,   5.          ],
 [  0.          , -12.          ,  -4.          ],
 [  0.          ,   0.          ,  -1.66666667]]
```

Wynik produkowany przez SciPy odibegał nieco od naszego (tzn. L i U były różne), ale nasze L i U mnożą się do A, więc wszystko jest w porządku

3 Rekurencyjne obliczanie wyznacznika

3.1 Opis algorytmu

Algorytm wykorzystuje faktoryzacja LU do obliczenia wyznacznika macierzy. Najpierw wykonujemy faktoryzacje, a następnie bierzemy wszystkie liczby z głównej przekątniej macierzy U i L, po czym mnożymi je wszystkie ze sobą.

$$[L_{n,n}, U_{n,n}] = LU(A_{n,n}) \tag{10}$$

$$U_{n,n} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix} \tag{11}$$

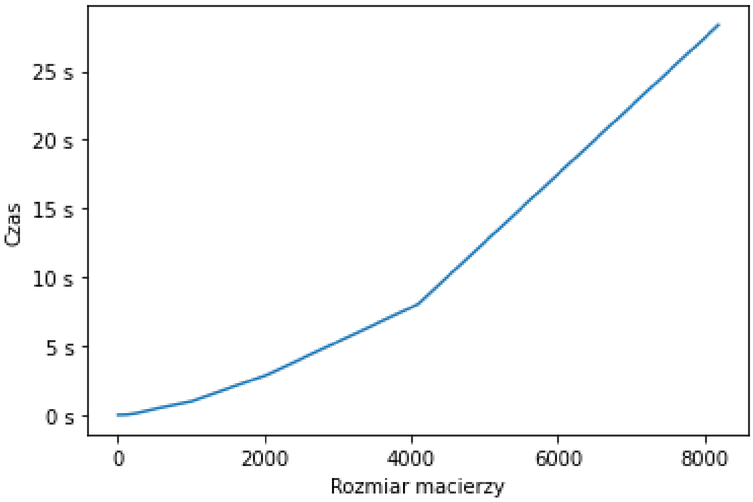
$$L_{n,n} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{pmatrix} \tag{12}$$

$$det(A) = \prod_{i=0}^{n} u_{i,i} * l_{i,i} \tag{13}$$

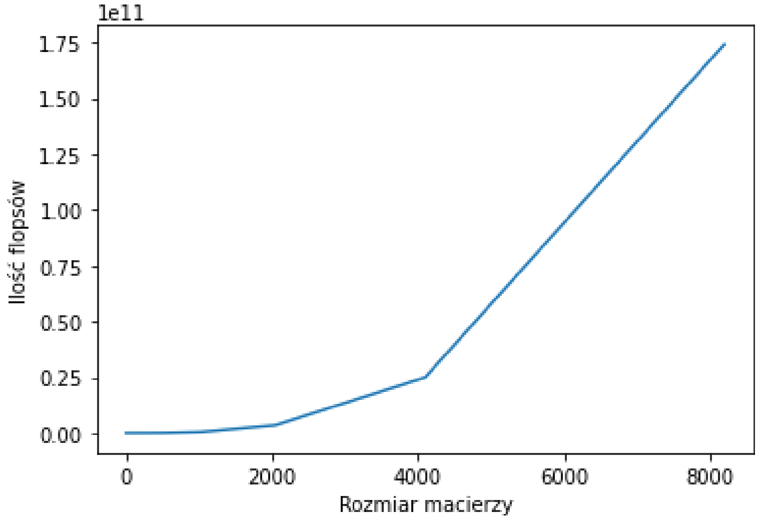
3.2 Pseudo-kod

```
def determinant(A):
    L, U = LU(A)

    U_diag = diagonal(U)

    det = 1
    for u in U_diag:
        det *= u

    return det
```

3.3 Benchmarki



Rysunek 6: Wykres czasu od rozmiaru macierzy.



Rysunek 7: Wykres libczby operacji od rozmiaru macierzy.

3.4 Złożoność obliczeniowa

Złożoność obliczeniowa algorytmu zależy od funkcji wyznaczającej LU oraz od rozmiaru macierzy. Ostatecznie więc uzyskujemy: $O(n^{2.807})$

3.5 Porównanie rozwiązania

Wyniki naszego algorytmu:

```
%% INPUT
A = np.array([[1, 8, 5], [2, 4, 6], [3, 5, 7]])
print(recursive_determinant(A, counter))

%% OUTPUT

20.000000000000004
```

Wyniki uzyskane w numpy'u:

```
%% INPUT
A = np.array([[1, 8, 5], [2, 4, 6], [3, 5, 7]])
print(np.linalg.det(A))

%% OUTPUT

19.999999999999996
```

Natomiast według Wolframa wyznacznik wynosi równe 20.