Labolatorium 2

Autorzy:

Patryk Klatka

Wojciech Łoboda

Import bibliotek oraz ich konfiguracja

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import time

# Matplotlib settings
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
plt.style.use('ggplot')
```

Rekurencyjne odwracanie macierzy

Odwracanie macierzy jest kluczowym elementem w wielu obszarach matematyki i inżynierii, szczególnie w analizie danych, statystycznych i obliczeniach związanych z algorytmami. Przykładowo, jest pomocne przy rozwiązywaniu układów równań liniowych, przy transformacjach geometrycznych w grafice komputerowej i wielu innych.

Psuedokod

Badana macierz:

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

Kroki:

1. Zawołanie reukrencyjne dla macierzy $A_{11}^{-1} = inverse(A_{11})$
2. Obliczenie dopełnienia Schura $S_{22} = A_{22} - A_{21} * A_{11}^{-1} * A_{12}$
3. Zawołanie reukrencyjne dla macierzy $S_{22}^{-1} = inverse(S_{22})$
4. Obilczenie odpowiednio:
    - $B_{11} = A_{11}^{-1} + A_{11}^{-1} * A_{12} * S_{22}^{-1} * A_{21} * A_{11}^{-1}$
    - $B_{12} = -A_{11}^{-1} * A_{12} * S_{22}^{-1}$
    - $B_{21} = -S_{22}^{-1} * A_{21} * A_{11}^{-1}$
    - $B_{22} = S_{22}^{-1}$

Najważniejszymi fragmentami powyższego pseudokodu są obliczenia rekurencyjne: w szczególności obliczenie dopełnienia Schura. Pozwala nam to na obliczenie macierzy odwrotnej blokowo, nie cierpiąc bardzo na złożoności obliczeniowej, czego dowiemy się w następnym punkcie.

Analiza złożoności obliczeniowej

W celu obliczenia złożoności obliczeniowej algorytmu rekurencyjnego odwracania macierzy, należy rozwiązać następujące równanie rekurencyjne:

$$T(n) = 2T(\frac{n}{2}) + 10 * C + d$$

gdzie C to złożoność obliczeniowa algorytmu mnożenia macierzy, a d to liczba pozostałych operacji o stałej złożoności, którą pominiemy.

Dla każdego kroku, wykonujemy 2 zawołania rekurencyjne funkcji `inverse` oraz wykonujemy 10 mnożeń o złożoności danego algorytmu mnożenia macierzy. Załóżmy, że korzystamy z algorytmu Strassena o złożoności $O(n^{\log_2 7})$.

Korzystając z twierdzenia o rekurencji uniwersalnej (znane również pod nazwą Master Theorem):

$$T(n) = aT(\frac{n}{b}) + f(n)$$

wnioskujemy, że $a = 2$, $b = 2$ oraz $f(n) = 10 * (\frac{n}{2})^{2.81}$.

Ponieważ $f(n)$ jest wielomianowo większe niż $n^{log_b a}$, to złożoność obliczeniowa algorytmu rekurencyjnego odwracania macierzy wynosi $\Theta(f(n)) = \Theta(n^{\log_2 7})$, a zatem wnioskujemy, że złożoność rekruencyjnego odwracania macierzy zależy od złożoności algorytmu mnożenia macierzy (wytłumaczenie wzorów tutaj).

```python
def number_of_flops_in_matrix_multiplication(n):
    """
    Compute the number of flops in a matrix multiplication.

    Note that this is not the most efficient way to compute the product.
    It's just an approximation of the number of flops.
    """
    return 2 * n ** 3 - n ** 2

def inverse(A, l=0):
    """
    Inverse of a matrix using recursion.
    """
    # Get the shape of the matrix
    m, n = A.shape

    # Check if the matrix is square
    if m != n:
        raise ValueError('Matrix must be square.')

    # Check if the matrix is invertible
    if np.linalg.det(A) == 0:
        raise ValueError('Matrix is not invertible.')

    # Check if the matrix is 2x2
    if m == 2:
        # Compute the inverse of the matrix
        A_inv = np.zeros_like(A)
        A_inv[0, 0] = A[1, 1]
        A_inv[0, 1] = -A[0, 1]
        A_inv[1, 0] = -A[1, 0]
        A_inv[1, 1] = A[0, 0]
        A_inv /= np.linalg.det(A)
```

```python
        return A_inv, 4

    # Otherwise, recursively compute the inverse
    else:
        # Split the matrix into blocks
        A11 = A[:m//2, :m//2]
        A12 = A[:m//2, m//2:]
        A21 = A[m//2:, :m//2]
        A22 = A[m//2:, m//2:]

        # Recursively compute the inverse of the block A11
        A11_inv, c1 = inverse(A11)

        # Compute the Schur complement
        S22 = A22 - A21 @ A11_inv @ A12

        # Compute the inverse of the Schur complement
        S22_inv, c2 = inverse(S22)

        # Compute the inverse of the matrix
        A_inv = np.zeros_like(A)
        A_inv[:m//2, :m//2] = A11_inv + A11_inv @ A12 @ S22_inv @ A21 @ A11_inv
        A_inv[:m//2, m//2:] = -A11_inv @ A12 @ S22_inv
        A_inv[m//2:, :m//2] = -S22_inv @ A21 @ A11_inv
        A_inv[m//2:, m//2:] = S22_inv

        return A_inv, l + c1 + c2 + number_of_flops_in_matrix_multiplication(n/2) * 10 + n
```

Test poprawności algorytmu

```python
test_matrix = np.array([
    [5,1,3,4],
    [0,1,8,5],
    [9,3,6,1],
    [7,3,9,2]
]).astype(np.float64)

m1 = inverse(test_matrix)
m2 = np.linalg.inv(test_matrix)
print("Correct" if np.allclose(m1[0], m2) else "ERROR")
print("Correct" if np.allclose(test_matrix, inverse(inverse(test_matrix)[0])[0]) else "ERROR")
```

```
Correct
Correct
```

Analiza algorytmu

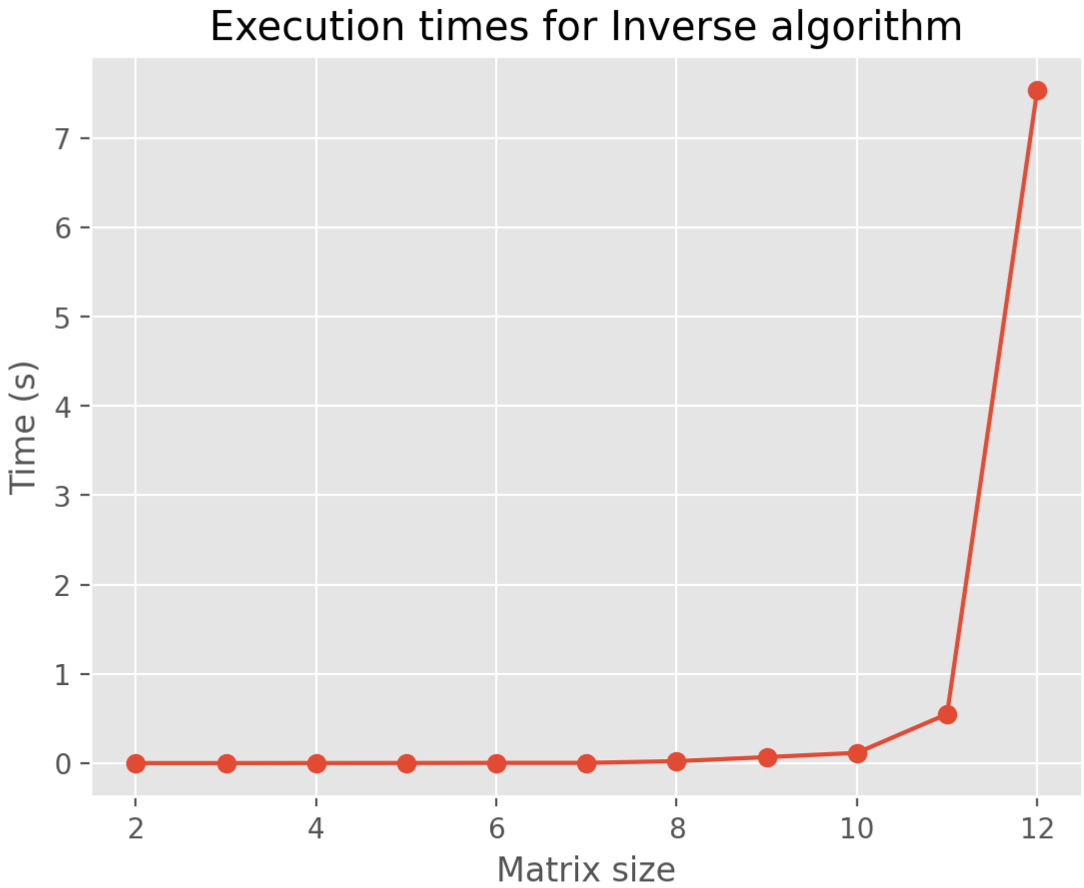
```python
# Generate matrices
def generate_matrix(n):
    return np.random.uniform(0.00000001, 1, (n, n))

k = 12
test_matrices = [generate_matrix(2**i) for i in range(2, k+1)]
```

Czas wykonywania algorytmu

```python
def plot_execution_times(algorithm, test_matrices):
    execution_times = []
```

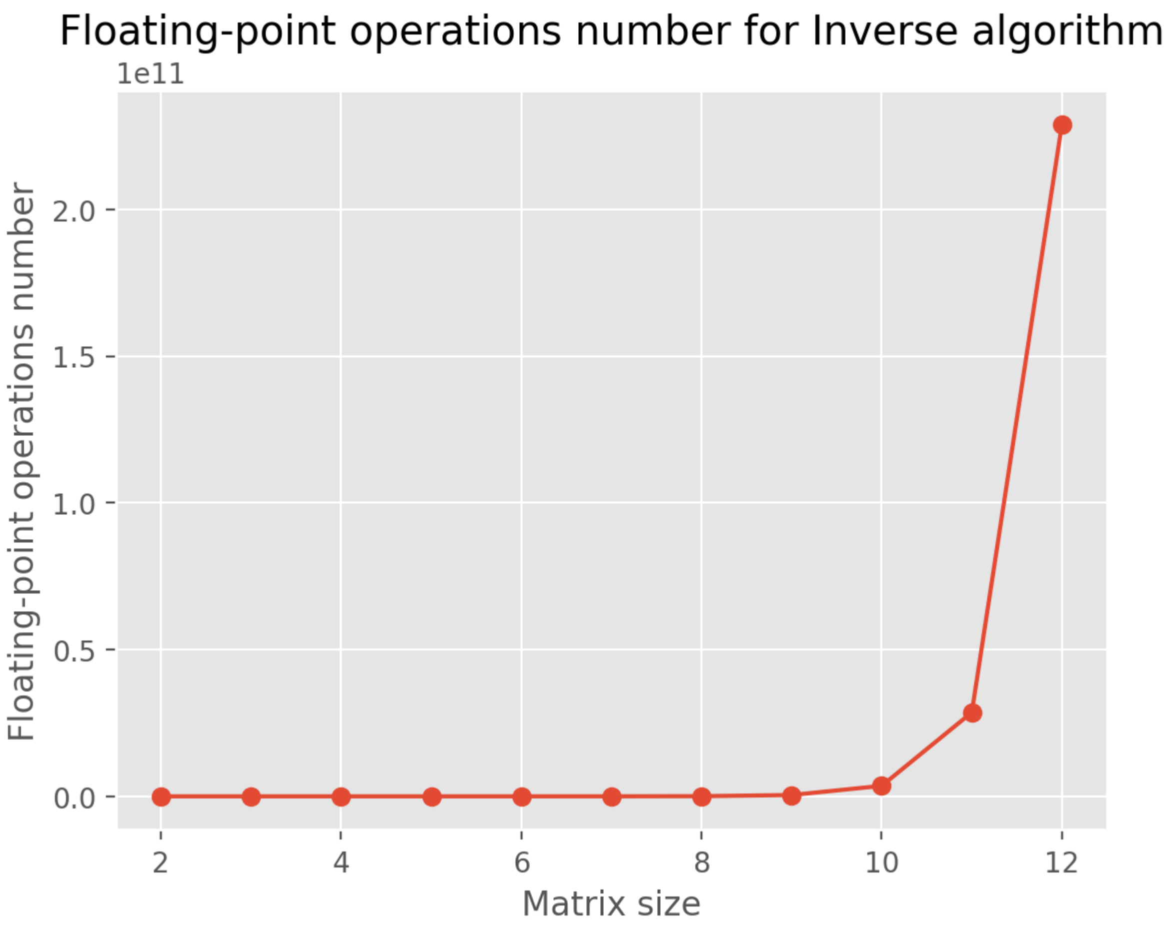
```
    for A in test_matrices:
        start_time = time.time()
        algorithm(A)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    plt.title("Execution times for " + algorithm.__name__.title() + " algorithm")
    plt.xlabel("Matrix size")
    plt.ylabel("Time (s)")
    plt.plot([i for i in range(2,k+1)] ,execution_times, 'o-')
    plt.show()

plot_execution_times(inverse, test_matrices)
```

```
/opt/homebrew/lib/python3.11/site-packages/numpy/linalg/linalg.py:2180: RuntimeWarning: overfl
ow encountered in det
  r = _umath_linalg.det(a, signature=signature)
```



Liczba operacji zmiennoprzecinkowych

```
def plot_flops(algorithm, test_matrices):
    flops = []
    for A in test_matrices:
        res = algorithm(A)
        flops.append(res[-1])
    plt.title("Floating-point operations number for " + algorithm.__name__.title() + " algorith
    plt.xlabel("Matrix size")
    plt.ylabel("Floating-point operations number")
    plt.plot([i for i in range(2,k+1)] ,flops, 'o-')
    plt.show()

plot_flops(inverse, test_matrices)
```



Rekurencyjna faktoryzacja LU

Faktoryzacja LU to metoda rozkładu macierzy na dwie macierze - L i U. Macierz L jest macierzą dolnotrójkątną, a U górną. Ta technika ma wiele zastosowań, wśród których najważniejsze to rozwiązywanie układów równań liniowych, obliczanie wyznacznika macierzy oraz odwracanie macierzy.

Psuedokod

$$LU = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

Kroki do obliczenia rekurencyjnie podmacierzy L oraz U:

1. Wyznaczenie macierzy L_{11} oraz U_{11} poprzez wykonanie rekurencyjne faktoryzacji LU dla macierzy A_{11}
2. Obliczenie rekurencyjnie $U_{11}^{-1} = inverse(U_{11})$
3. Obliczenie $L_{21} = A_{21} * U_{11}^{-1}$
4. Obliczenie rekurencyjne $L_{11}^{-1} = inverse(L_{11})$
5. Obliczenie $U_{12} = L_{11}^{-1} * A_{12}$
6. Obliczenie dopelnienia Schura $S = A_{22} - A_{21} * U_{11}^{-1} * L_{11}^{-1} * A_{12}$
7. Obliczenie rekurencyjne macierzy L_s oraz U_s dla macierzy S
8. Podstawienie $L_{22} = L_s$ oraz $U_{22} = U_s$

Najważniejszymi fragmentami powyższego pseudokodu są obliczenia rekurencyjne: w szczególności obliczenie dopełnienia Schura. Głównie te fragmenty pozwalają nam na wykonanie faktoryzacji LU macierzy blokowo.

Analiza złożoności obliczeniowej

W celu obliczenia złożoności obliczeniowej algorytmu rekurencyjnego odwracania macierzy, należy rozwiązać następujące równanie rekurencyjne:

$$T(n) = 4T(\frac{n}{2}) + 5 * C + d$$

gdzie C to złożoność obliczeniowa algorytmu mnożenia macierzy, a d to liczba pozostałych operacji o stałej złożoności, którą pominiemy.

Tak jak dla poprzedniego algorytmu, z twierdzenia o rekurencji uniwersalnej wnioskujemy, że $a = 4$, $b = 2$ oraz $f(n) = 5 * (\frac{n}{2})^{2.81}$.

Ponieważ $f(n)$ jest wielomianowo większe niż $n^{log_b a}$, to złożoność obliczeniowa algorytmu rekurencyjnego odwracania macierzy wynosi $\Theta(f(n)) = \Theta(n^{\log_2 7})$, a zatem wnioskujemy, że złożoność rekruencyjnego odwracania macierzy zależy od złożoności algorytmu mnożenia macierzy.

```python
def LU_factorization_recursive(A, l=0):
    """
    LU factorization of a matrix using recursion.
    """
    # Get the shape of the matrix
    m, n = A.shape

    # Check if the matrix is square
    if m != n:
        raise ValueError('Matrix must be square.')

    # Check if the matrix is 2x2
    if m == 2:
        # Compute the LU decomposition of the matrix
        L = np.eye(2)
        L[1, 0] = A[1, 0] / A[0, 0]

        U = np.zeros_like(A)
        U[0, 0] = A[0, 0]
        U[0, 1] = A[0, 1]
        U[1, 1] = A[1, 1] - L[1, 0] * A[0, 1]

        return L, U, 3
    else:
        # Split the matrix into blocks
        A11 = A[:m//2, :m//2]
        A12 = A[:m//2, m//2:]
        A21 = A[m//2:, :m//2]
        A22 = A[m//2:, m//2:]

        # Recursively compute the LU decomposition of the block A11
        L11, U11, c1 = LU_factorization_recursive(A11)

        U11_inv, c2 = inverse(U11)

        L21 = A21 @ U11_inv

        L11_inv, c3 = inverse(L11)

        U12 = L11_inv @ A12

        S = A22 - A21 @ U11_inv @ L11_inv @ A12
```

```
        Ls, Us, c4 = LU_factorization_recursive(S)

        U22 = Us
        L22 = Ls

        # Compute the LU decomposition of the matrix
        L = np.zeros_like(A)
        L[:m//2, :m//2] = L11
        L[m//2:, :m//2] = L21
        L[m//2:, m//2:] = L22

        U = np.zeros_like(A)
        U[:m//2, :m//2] = U11
        U[:m//2, m//2:] = U12
        U[m//2:, m//2:] = U22

        return L, U, l + c1 + c2 + c3 + c4 + number_of_flops_in_matrix_multiplication(n/2) * 5
```

Test poprawności algorytmu

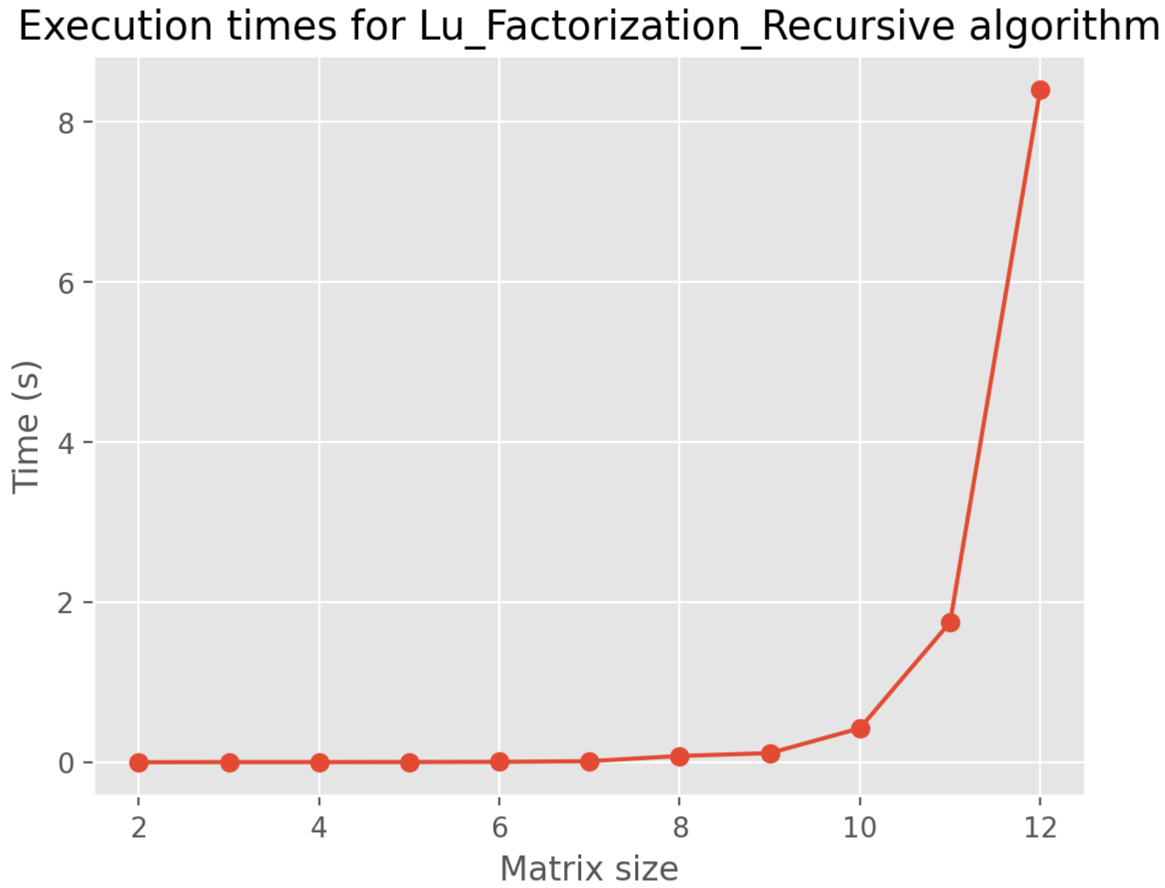
```
test_matrix = np.array([
 [5, 2, 3 ,9],
 [2 ,4 ,6 ,8],
 [7, 4, 8 ,4],
 [3 ,4, 5, 5],

]).astype(np.float64)

m1 = LU_factorization_recursive(test_matrix)
print("Correct" if np.allclose(m1[0] @ m1[1], test_matrix) else "ERROR")
```

```
Correct
```

Analiza algorytmu

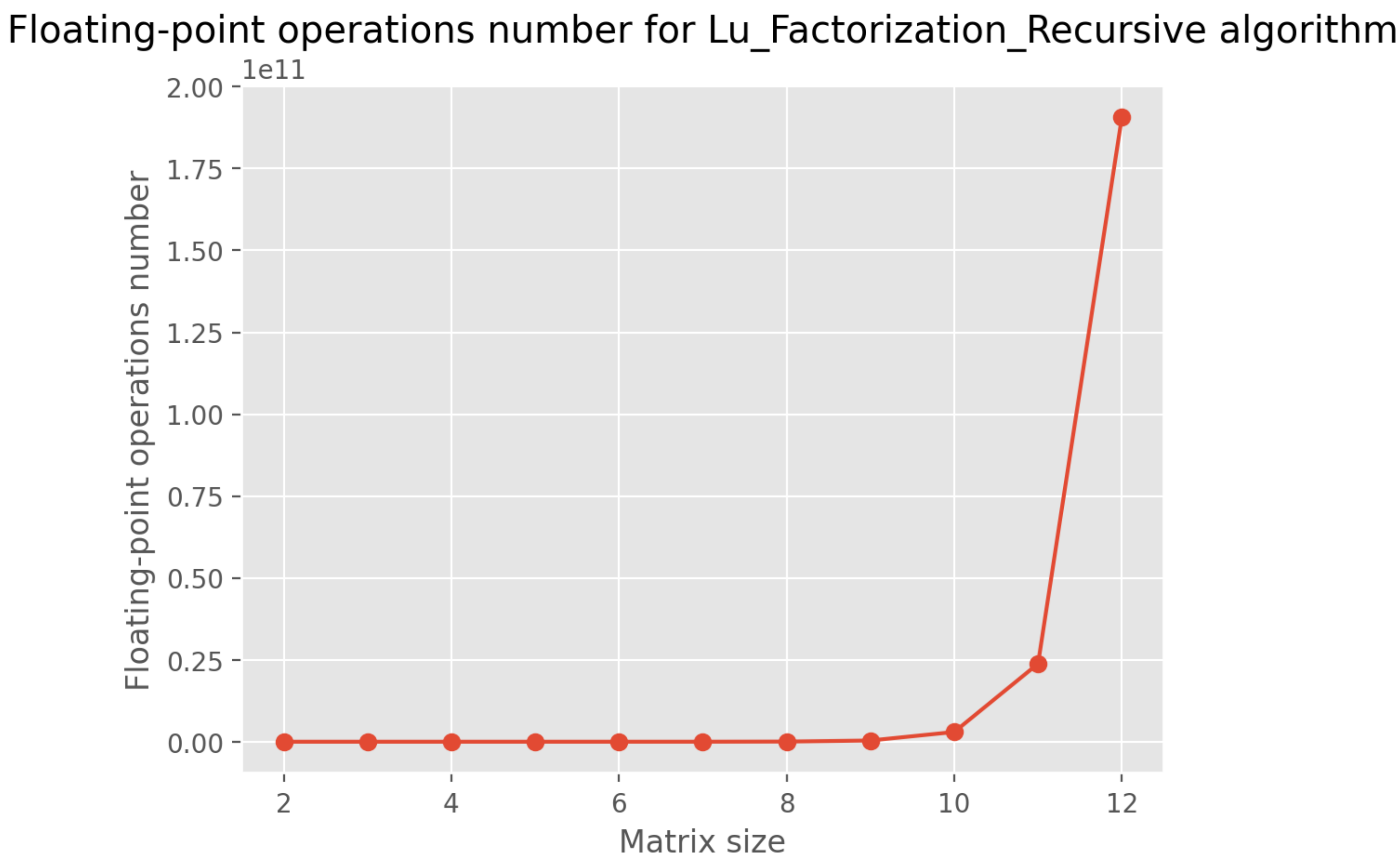
Czas wykonywania algorytmu

```
plot_execution_times(LU_factorization_recursive, test_matrices)
```



Liczba operacji zmiennoprzecinkowych

```
plot_flops(LU_factorization_recursive, test_matrices)
```



Rekurencyjne obliczanie wyznacznika

Obliczanie wyznacznika to jedna z podstawowych operacji macierzowych. Wyznacznik to pewna wartość która charakteryzuje macierz kwadratową np. wyznacznik jest niezerowy wtedy i tylko wtedy kiedy macierz jest odwracalna. Wyznacznik może być wykorzystany między innymi do obliczania układów równań liniowych.

Wzór na wyznacznik macierzy A o rozmiarze $n \times n$:

$det(A) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} a_{\sigma(i),i}$

gdzie S_n to grupa permutacji n-elementowych, sgn to funkcja znaku permutacji, $a_{i,j}$ to element w i-tym wierszu i j-tej kolumnie.

Wykorzystując twierdzenie Cauchy'ego: $det(AB) = det(A) * det(B)$ oraz własność wyznacznika mówiącą że wyznacznik macierzy dolnejtrójkątnej i górnejtrójkątnej to iloczyn elementów znajdujących się na przekątnej macierzy, możemy wyprowadzić wzór na wyznacznik znając faktoryzacje LU macierzy:

$det(A) = l_{11} * ... * l_{nn} * u_{11} * ... * u_{nn} = u_{11} * ... * u_{nn}$ (jeżeli zakładamy faktoryzacje z $l_{ii} = 1$)

gdzie l_{ii} to elementu na przekątnej L i u_{ii} to elementy na przekątnej U.

Pseudokod algorytmu obliczenia wyznacznika dla macierzy A:

1. Wyznaczenie rekurencyjnie faktoryzacji LU dla macierzy A.
2. Wymnożenie elementów na przekątnych macierzy L i U.

Analiza złożoności obliczeniowej

Algorytm rekurencyjnego obliczenia wyznacznika macierzy A o rozmiarze $n \times n$ polega na znalezieniu rekurencyjnie faktoryzaji LU co ma złożóność $O(n^{log_2 7})$ a następnie na wykonaniu $O(n)$ mnożeń elementów na przekątnych. A zatem złożóność całego algorytmu wynosci $O(n^{log_2 7} + n) = O(n^{log_2 7})$.

```
def recursive_determinant(A):
    #computing LU matrices recursivly
    L, U, flops = LU_factorization_recursive(A)

    #determinant is a product of elements on diagonal of LU matrices (L has only ones).
    return np.prod(np.diag(U)), flops + U.shape[0] - 1
```

Test poprawności algorytmu

```
test_matrix = np.array([
 [5, 2, 3 ,9],
 [2 ,4 ,6 ,8],
 [7, 4, 8 ,4],
 [3 ,4, 5, 5],

]).astype(np.float64)

m, _ = recursive_determinant(test_matrix)
print("Correct" if np.allclose(m, np.linalg.det(test_matrix)) else "ERROR")
```

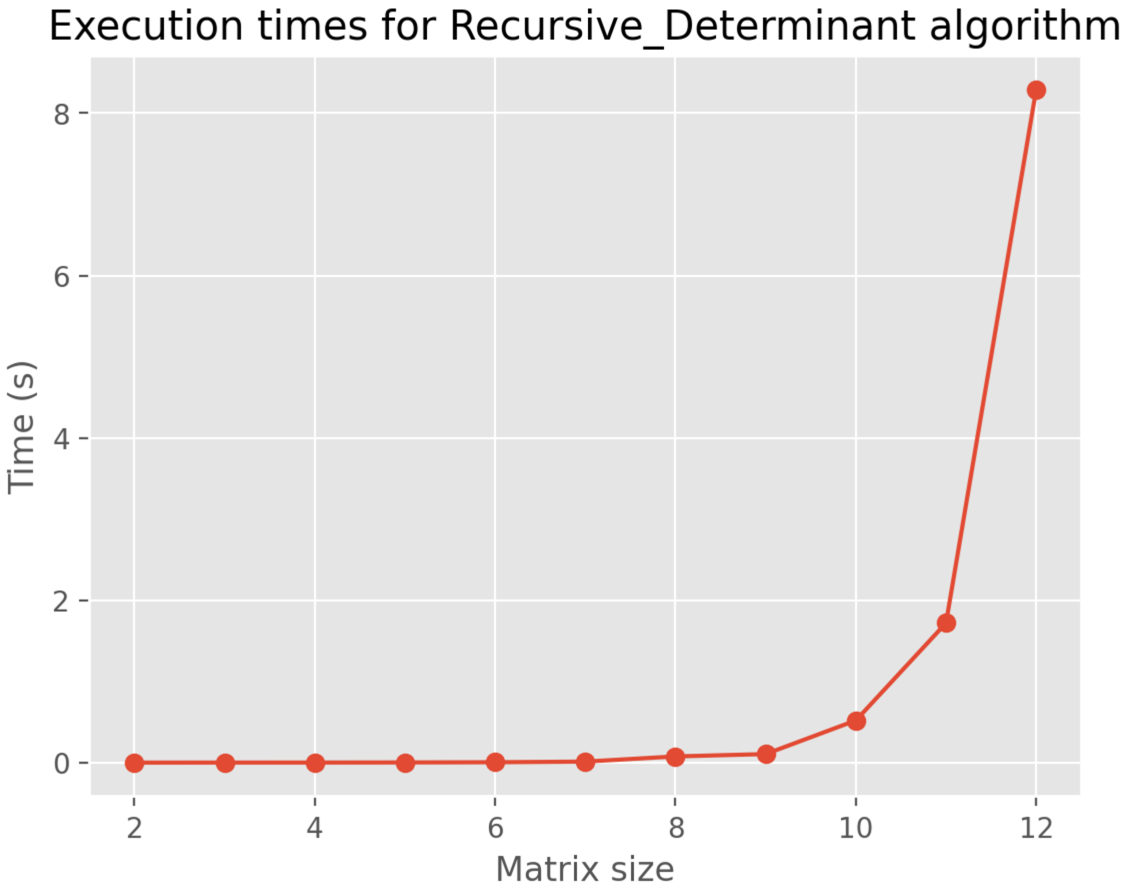
```
Correct
```

Analiza algorytmu

Czas wykonywania algorytmu

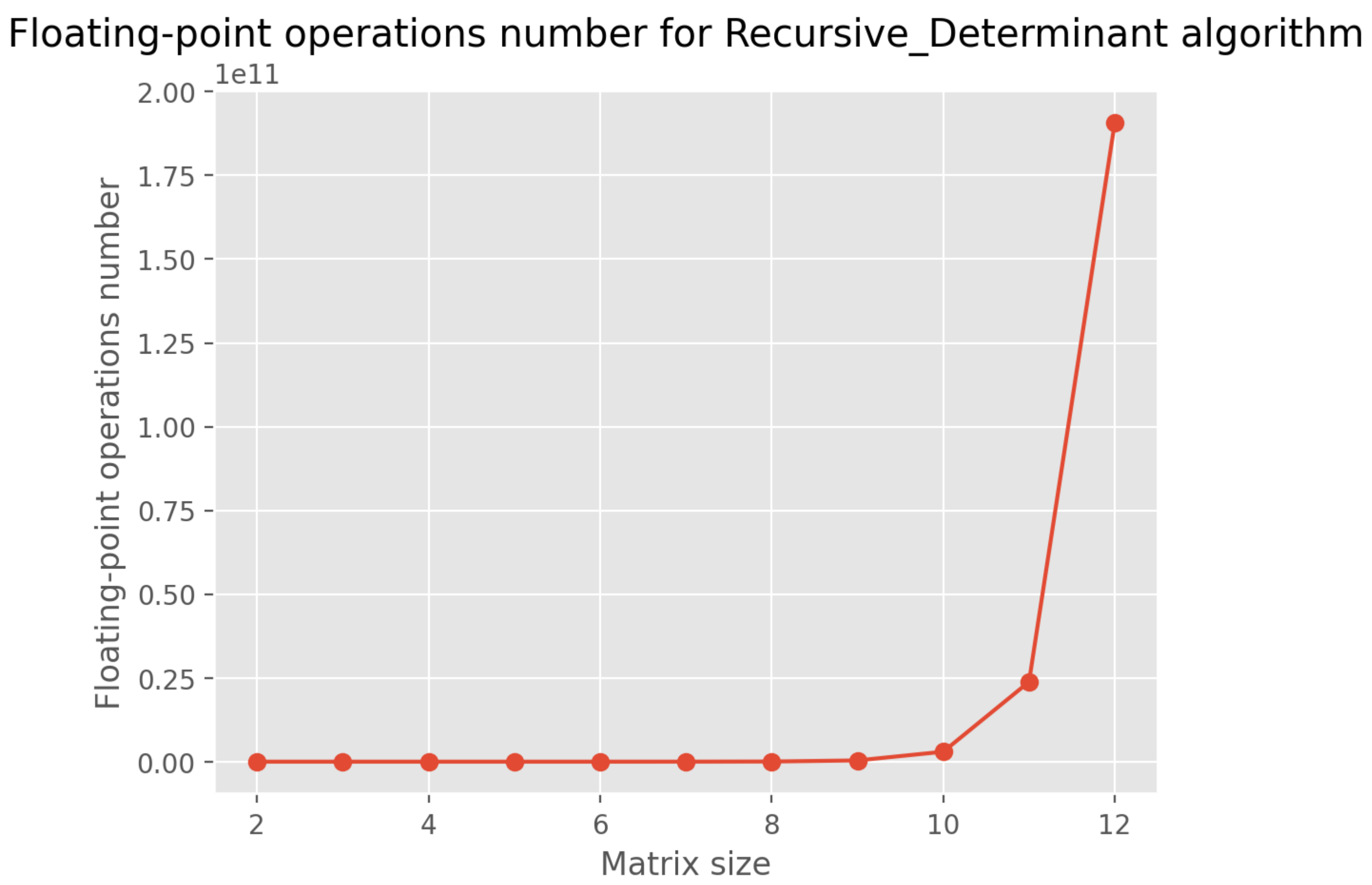
```
plot_execution_times(recursive_determinant, test_matrices)
```

```
/opt/homebrew/lib/python3.11/site-packages/numpy/core/fromnumeric.py:88: RuntimeWarning: overf
low encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```



Liczba operacji zmiennoprzecinkowych

```
plot_flops(recursive_determinant, test_matrices)
```



Porównanie wyników zaimplementowanych algorytmów do ich odpowiedników w środowisku MATLAB

Wygenerowana macierz:

```
0.7577    0.1712    0.0462    0.3171
    0.7431    0.7060    0.0971    0.9502
    0.3922    0.0318    0.8235    0.0344
    0.6555    0.2769    0.6948    0.4387
```

Wyniki ze środowiska MATLAB:

```
>> inv(mat)

ans =

    2.0990    0.3865    1.9524   -2.5074
    0.8852    8.1778   15.4969  -19.5678
   -0.9419   -0.2786    0.2319    1.2659
   -2.2030   -5.2978  -13.0653   16.3710

>> det(mat)

ans =

    0.0201

>> [L,U] = lu(mat)

L =
```

```
    1.0000         0         0         0
    0.9807    1.0000         0         0
    0.5176   -0.1055    1.0000         0
    0.8650    0.2394    0.7981    1.0000


U =

    0.7577    0.1712    0.0462    0.3171
         0    0.5382    0.0519    0.6392
         0         0    0.8050   -0.0623
         0         0         0    0.0611
```

```python
test_matrix = np.array([
    [0.7577, 0.1712, 0.0462, 0.3171],
    [0.7431, 0.7060, 0.0971, 0.9502],
    [0.3922, 0.0318, 0.8235, 0.0344],
    [0.6555, 0.2769, 0.6948, 0.4387],
])

test_matrix_inverse = inverse(test_matrix)[0]
test_matrix_determinant = recursive_determinant(test_matrix)[0]
test_matrix_LU = LU_factorization_recursive(test_matrix)

print(f"Matrix inverse:\n{test_matrix_inverse}\n")
print(f"Matrix determinant: {test_matrix_determinant}\n")
print(f"Matrix LU decomposition:\nL = {test_matrix_LU[0]}\n\nU = {test_matrix_LU[1]}\n")
```

```
Matrix inverse:
[[  2.09969678   0.38666056   1.95291882  -2.50831804]
 [  0.88882236   8.1818861   15.50517293 -19.57978503]
 [ -0.94216534  -0.27867616   0.23151591   1.26645674]
 [ -2.20617661  -5.30065207 -13.07129684  16.38001097]]

Matrix determinant: 0.020038737824377883

Matrix LU decomposition:
L = [[ 1.          0.          0.          0.        ]
 [ 0.98073116  1.          0.          0.        ]
 [ 0.51761911 -0.10558728  1.          0.        ]
 [ 0.86511812  0.23934596  0.79800294  1.        ]]

U = [[ 0.7577      0.1712      0.0462      0.3171    ]
 [ 0.          0.53809883  0.05179022  0.63921015]
 [ 0.          0.          0.80505439 -0.06224456]
 [ 0.          0.          0.          0.06105002]]
```

Porównując wyniki ze środowiska MATLAB możemy zauważyć, że wyniki porównane z naszymi implementacjami algorytmów są identyczne.

Wnioski

Na podstawie przeprowadzonych testów, stwierdzamy, że zaimplementowane przez nas algorytmy dziająją poprawnie. Rekurencyjny algorytm odwracania macierzy ma lepszą złożóność obliczeniową niż odwracanie macierzy wykorzystując zwykłą eliminacje Gaussa, czyli algorytm o złożóności $O(n^3)$. Podobnie w przypadku algorytmu rekurencyjnej faktoryzacji LU, gdzie również jesteśmy w stanie osiągnąć lepszą złożoność niż algorytm wykorzystujący eliminacje Gaussa. Złożoność algorytmu obliczania wyznacznika zależy od implmenetacji faktoryzacji LU, w naszym przypadku zastosowaliśmy reukrencyjną faktoryzacja osiągając złożóność $O(n^{log_2 7})$, a zatem lepszą niż inne metody na przykład: rozwinięcie Laplace'a - $O(n!)$, faktoryzacje LU z eliminacją Gaussa - $O(n^3)$ czy algorytm Bareissa $O(n^3)$. Rekurencyjna faktoryzacja LU i obliczenie wyznacznika dziają niemal w identycznym czasie, również liczba operacji zmiennoprzecinkowych jest zbliżona.