Algorytmy Macierzowe

Laboratorium nr. 2

Artur Gęsiarz, Błażej Kapkowski

1. **Cel Zadania:**
   Celem zadania było zaimplementowanie rekurencyjnych algorytmów odwracania macierzy oraz obliczania wyznaczników. W zadaniu zaimplementowane zostały cztery algorytmy:
   - **Rekurencyjne odwracanie macierzy** – metoda umożliwiająca odwrócenie macierzy poprzez rozbijanie jej na mniejsze podmacierze i iteracyjne znajdowanie odwrotności.
   - **Rekurencyjna eliminacja Gaussa** – rekurencyjna metoda do obliczania wyznacznika, redukująca macierz do postaci trójkątnej poprzez operacje elementarne.
   - **Rekurencyjna LU faktoryzacja** – metoda rozkładu macierzy na iloczyn macierzy dolnotrójkątnej i górnotrójkątnej, z wykorzystaniem podziału na mniejsze podmacierze.
   - **Rekurencyjne liczenie wyznacznika** – technika obliczania wyznacznika macierzy poprzez rozbijanie jej na mniejsze podmacierze i iteracyjne wyznaczanie mniejszych wyznaczników.

2. **Rekurencyjne odwracanie macierzy**
   Odwracanie macierzy metodą rekurencyjną polega na wykorzystaniu dekompozycji macierzy na mniejsze podmacierze, a następnie obliczaniu odwrotności tych podmacierzy w sposób rekurencyjny. Algorytm ten opiera się na rozkładzie macierzy na mniejsze podmacierze, z których każda może być odwrócona osobno, a następnie odpowiednie wyniki są scalane, by uzyskać odwrotność całej macierzy.

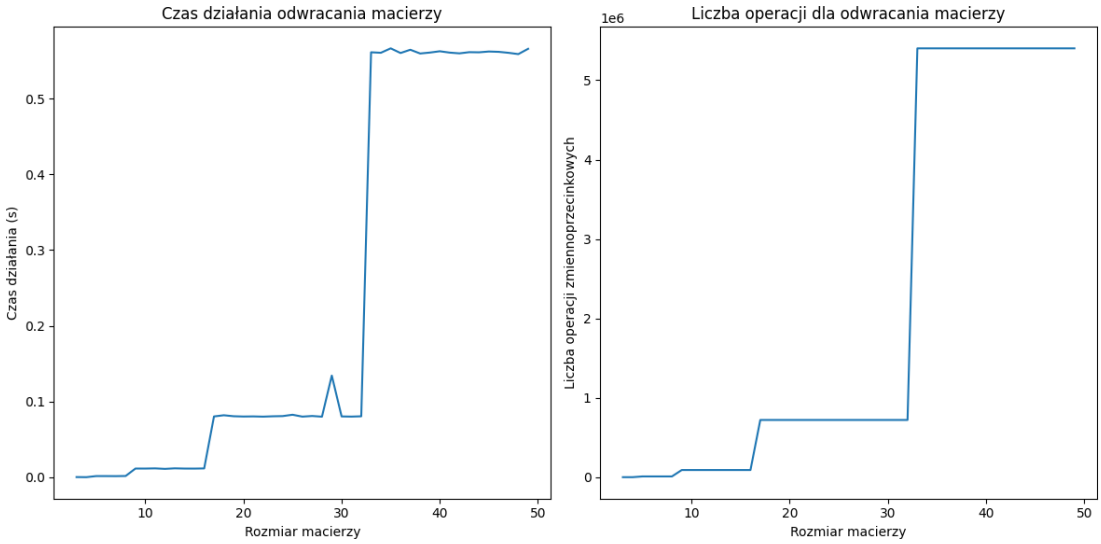
```python
def invert(A):
    n = A.shape[0]
    mid = n // 2

    if n == 1:
        A[0, 0] += 1e-17
        return np.array([[1 / A[0, 0]]]), 1

    A11, A12, A21, A22 = split_matrix(A)
```

```python
    count = [0] * 16

    # A11_inv = inverse(A11)
    A11_inv, count[0] = invert(A11)

    # S22 = A22 - A21 @ A11_inv @ A12
    S22, count[1] = strassen(A21, A11_inv)
    S22, count[2] = strassen(S22, A12)
    S22, count[3] = A22 - S22, mid**2

    # S22_inv = inverse(S22)
    S22_inv, count[4] = invert(S22)
```

```python
    # B11 = A11_inv + A11_inv @ A12 @ S22_inv @ A21 @ A11_inv
    B11, count[5] = strassen(A11_inv, A12)
    B11, count[6] = strassen(B11, S22_inv)
    B11, count[7] = strassen(B11, A21)
    B11, count[8] = strassen(B11, A11_inv)
    B11, count[9] = A11_inv + B11, mid

    # B12 = -A11_inv @ A12 @ S22_inv
    B12, count[10] = strassen(A11_inv, A12)
    B12, count[11] = strassen(B12, S22_inv)
    B12, count[12] = -1 * B12, mid**2
```

```
# B21 = -S22_inv @ A21 @ A11_inv
B21, count[13] = strassen(S22_inv, A21)
B21, count[14] = strassen(B21, A11_inv)
B21, count[15] = -1 * B21, mid**2

# B22 = S22_inv
B22 = S22_inv

return np.vstack((np.hstack((B11, B12)), np.hstack((B21, B22)))), sum(count)
```

- **Wyniki:**



- **Oszacowanie złożoności obliczeniowej:**

Algortym dzieli macierz na cztery podmacierze A11, A12, A21, A22, a nastepnie rekurencyjne oblicza odwrotnosci tych macierzy.

Dla kazdego wywołania funkcji invert, zloznosc obliczniowa jest zwiazana z:

1. Wywołaniem funkcji rekurencyjnych dla podmacierzy, ktore maja rozmar $\frac{n}{2} x \frac{n}{2}$.

2. Wykonaniem operacji na podmacierzach, takich jak mnozenie macierzy $O(n^{2.81})$,

$$T(n) = 2 * T\left(\frac{n}{2}\right) + 4 * \left(\frac{n}{2}\right)^2 + 10 \cdot \left(\frac{n}{2}\right)^{2.807}$$
$$T(n) = O(n^{2.807})$$

3. Rekurencyjna eliminacja Gaussa

Eliminacja Gaussa służy do obliczania wyznacznika macierzy kwadratowej poprzez przekształcenie jej w macierz trójkątną górną. Algorytm bazuje na modyfikacji elementów macierzy w taki sposób, aby uzyskać wszystkie wartości poniżej przekątnej równe zero. Finalnie wyznacznik macierzy można obliczyć jako iloczyn elementów znajdujących się na przekątnej tej trójkątnej macierzy.

Poniżej opis poszczególnych fragmentów kodu:

Funkcja gauss_det służy do obliczenia wyznacznika macierzy przy użyciu eliminacji Gaussa. Na początku sprawdza wymiar macierzy. Jeśli macierz ma wymiar 1, zwraca jedyny element jako wyznacznik.

```
def gauss_det(matrix):
    global flops

    n = len(matrix)

    if n == 1:
        return matrix[0][0]
```

Poniższy fragment kodu sprawdza, czy element na przekątnej głównej wiersza i jest równy zero. Jeśli tak, algorytm próbuje zamienić ten wiersz z innym wierszem poniżej, który ma element niezerowy w tej samej kolumnie. W przeciwnym wypadku wyznacznik wynosi zero i zwracana jest wartość 0

```
for i in range(n):
        if matrix[i][i] == 0:
            for k in range(i + 1, n):
                if matrix[k][i] != 0:
                    matrix[i], matrix[k] = matrix[k], matrix[i]
                    flops += n
                    break
            else:
                return 0
```

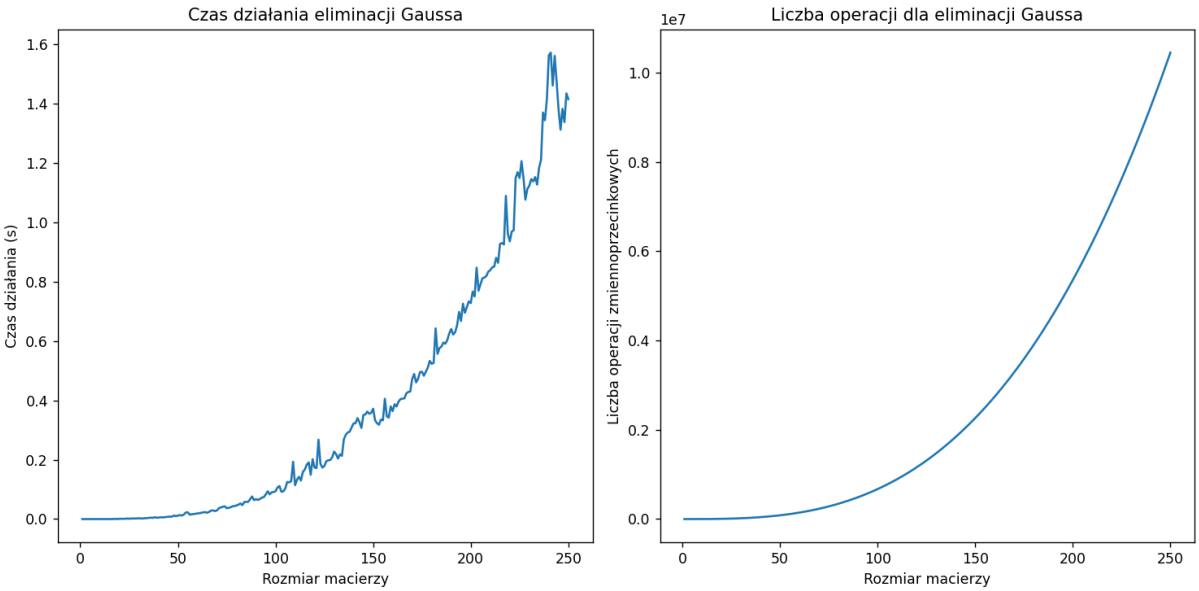
Dla każdego wiersza poniżej aktualnie przetwarzanego, algorytm oblicza współczynnik ratio, który jest ilorazem elementu w danym wierszu i kolumnie przez element na przekątnej głównej bieżącego wiersza. Następnie aktualizuje każdy element wiersza poprzez odjęcie iloczynu ratio i wartości elementu z aktualnie przetwarzanego wiersza. To tworzy zerowe elementy poniżej przekątnej.

```
        for j in range(i + 1, n):
            ratio = matrix[j][i] / matrix[i][i]
            flops += 1

            for k in range(i, n):
                matrix[j][k] -= ratio * matrix[i][k]
                flops += 2
```

Po uzyskaniu macierzy trójkątnej górnej, wyznacznik jest obliczany jako iloczyn elementów na przekątnej głównej.

```
det = 1
    for i in range(n):
        det *= matrix[i][i]
        flops += 1

    return det
```

- **Wyniki:**



- **Oszacowanie złożoności obliczeniowej:**

  W pierwszym kroku algorytm sprawdza, czy element na przekątnej jest równy zero. Jeśli tak, zamienia odpowiednie wiersze, co w najgorszym przypadku może wymagać do n operacji dla każdej kolumny. Zatem ta część ma złożoność O(n^2). Eliminacja dolnego trójkąta: Dla każdego elementu poniżej przekątnej, algorytm wykonuje operacje arytmetyczne, aby uzyskać zera w odpowiednich miejscach.

Na pierwszym etapie, dla pierwszej kolumny, trzeba zredukować n - 1 elementów, co wymaga operacji O(n) na każdym elemencie, w tym odejmowania i mnożenia.
Na drugim etapie (dla drugiej kolumny), redukcja dotyczy n – 2 elementów, co znowu wymaga O(n) operacji na każdym elemencie, i tak dalej.

Suma operacji w tej części to:

$O(n) + O(n-1) + O(n-2) + \ldots + O(1) = O(\frac{n(n-1)}{2}) = O(n^2)$

Operacje te są powtarzane dla każdego elementu poniżej przekątnej, co daje łącznie złożoność **O(n^3)** dla całego kroku eliminacji.

Po zakończeniu eliminacji, wyznacznik jest obliczany jako iloczyn elementów na przekątnej, co wymaga O(n) operacji, więc nie zmienia on złożoności.

4. **Rekurencyjna LU faktoryzacja**

LU faktoryzacja to metoda rozkładu macierzy na iloczyn dwóch macierzy: L - dolnotrójkątnej oraz U - górnotrójkątnej. Rekurencyjna metoda LU faktoryzacji polega na rozbijaniu macierzy A na mniejsze podmacierze i znajdowaniu dla nich odpowiednich macierzy

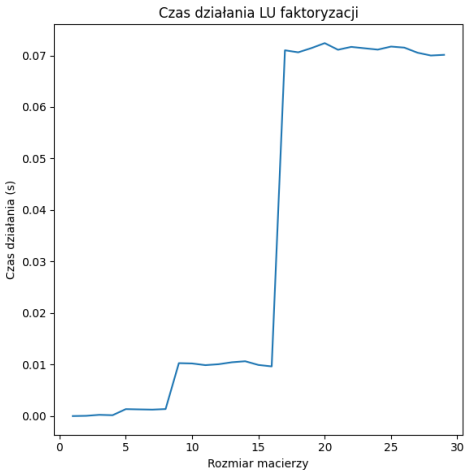
```
def lu_factorization(A):
  n = A.shape[0]
  mid = n // 2

  if n == 1:
    return np.array([[1]]), A.copy(), 0

  flops = [0] * 10
  A11, A12, A21, A22 = split_matrix(A)
```

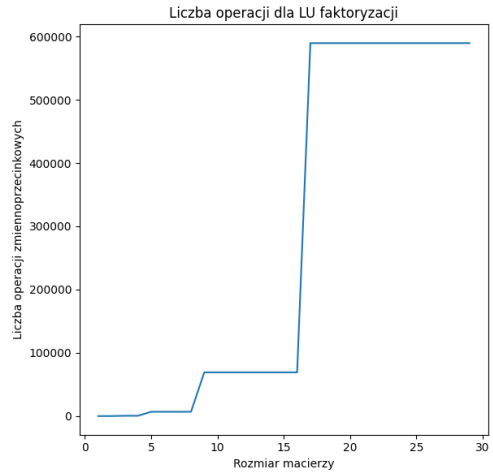
```
L11, U11, flops[0] = lu_factorization(A11)
U11_inv, flops[1] = invert(U11)
L21, flops[2] = strassen(A21,U11_inv)
L11_inv, flops[3] = invert(L11)
U12, flops[4] = strassen(L11_inv, A12)
```

```
# S = A22 - A21 @ U11_inv @ L11_inv @ A12
S, flops[5] = strassen(A21, U11_inv)
S, flops[6] = strassen(S, L11_inv)
S, flops[7] = strassen(S, A12)
S, flops[8] = A22 - S, mid**2

L22, U22, flops[9] = lu_factorization(S)
```

```
L = np.block([
    [L11, np.zeros((mid, n - mid))],
    [L21, L22]
])

U = np.block([
    [U11, U12],
    [np.zeros((n - mid, mid)), U22]
])

return L, U, sum(flops)
```

- **Wyniki:**





- **Oszacowanie złożonosci obliczeniowej:**
  Dzielimy macierz A na cztery podmacierze.

Dla każdej z tych podmacierzy (poza operacjami Strassena i inwersjami) wywoływana jest rekurencja. Tak więc, mamy 4 wywołania funkcji LU dla podmacierzy o rozmiarze n/2.

Mnożenie macierzy Strassena wykonuje się na podmacierzach o rozmiarze $\frac{n}{2}$. Stressen wykonuje 7 mnozen macierzy.

Dla każdej z macierzy wykonujemy inwersję, co wymaga rozwiązania układu równań, którego złożoność jest rowna $O(n^3)$ w najgorszym przypadku.

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{3}{2} * \left(\frac{n}{2}\right)^3 + 5 * \left(\frac{n}{2}\right)^{2.807}$$

$$T(n) = \frac{3}{2^{3.807}} n^{2.807} + n^2$$

5. **Rekurencyjne liczenie wyznacznika**

Algorytm rekurencyjny obliczania wyznacznika macierzy, przedstawiony w poniższym kodzie, opiera się na **rozwinięciu Laplace'a**. Rozwój ten polega na wyznaczeniu wyznacznika macierzy przez rozwinięcie po pierwszym wierszu i zastosowanie tzw. **minora** i **kofaktora** dla każdego elementu tego wiersza.

Funkcja det_rec oblicza wyznacznik macierzy metodą rekurencyjną:

Jeśli macierz jest 1x1, zwraca jej jedyny element jako wyznacznik.
Jeśli macierz jest 2x2, wyznacznik obliczany jest bezpośrednio ze wzoru det = ad-bc

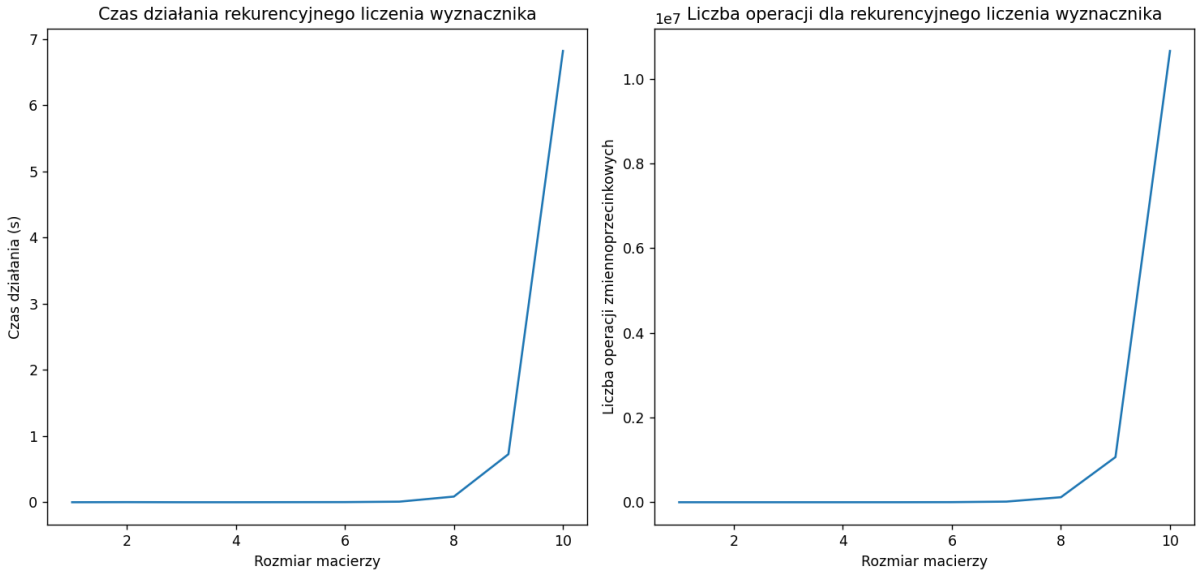
```
def det_rec(matrix):
    global flops

    if len(matrix) == 1:
        return matrix[0][0]

    if len(matrix) == 2:
        flops += 3
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]
```

Dla macierzy o wymiarze większym niż 2x2, funkcja wykorzystuje rozwinięcie Laplace'a po pierwszym wierszu:

Obliczanie kofaktora: Dla każdego pierwszego elementu kolumny, algorytm oblicza kofaktor – mnożnik, który zależy od pozycji elementu i wartości samego elementu.
Tworzenie podmacierzy: Algorytm tworzy podmacierz, która jest macierzą (n-1)x(n-1) powstałą przez usunięcie wiersza 0 i kolumny col z oryginalnej macierzy.
Rekurencyjne wywołanie: Wywołuje funkcję det_rec na podmacierzy, aby obliczyć wyznacznik podmacierzy.
Sumowanie kofaktorów: Obliczony wyznacznik jest dodawany do głównego wyznacznika det po przemnożeniu przez odpowiedni kofaktor.

```
    det = 0
    for col in range(len(matrix)):
        submatrix = [row[:col] + row[col + 1:] for row in matrix[1:]]
        cofactor = (-1) ** col * matrix[0][col]

        flops += 1

        det += cofactor * det_rec(submatrix)

        flops += 1
```

- **Wyniki:**



- **Oszacowanie złożoności obliczeniowej:**

Rozważmy macierz n x n. Algorytm rozwinięcia Laplace'a oblicza wyznacznik przez rozwinięcie wzdłuż pierwszego wiersza, co wymaga wykonania n wywołań funkcji det_rec dla podmacierzy o rozmiarze (n−1)x(n−1). Dla każdego z tych wywołań tworzona jest podmacierz i wywoływana jest funkcja rekurencyjna.

Oznaczmy złożoność obliczeniową algorytmu dla macierzy n x n jako T(n).
Wówczas rekurencyjnie możemy zapisać:

$$T(n) = nT(n-1)$$

nT(n-1) to Koszt rekursji, ponieważ algorytm wykonuje n wywołań dla podmacierzy o rozmiarze (n-1)x(n-1).

Rozwinięcie: $T(n) = n \cdot T(n-1) = n \cdot ((n-1) \cdot T(n-2) = n \cdot (n-1) \cdot (n-2) \cdot T(n-3) = itd.$

Patrząc na rozwinięcie, widzimy, że tworzy się ciąg zależny od $n(n-1)(n-2)(n-3)$… = n!
co oznacza, że: $T(n) = O(n!)$.