Sprawozdanie

1. Rekurencyjne odwracanie macierzy

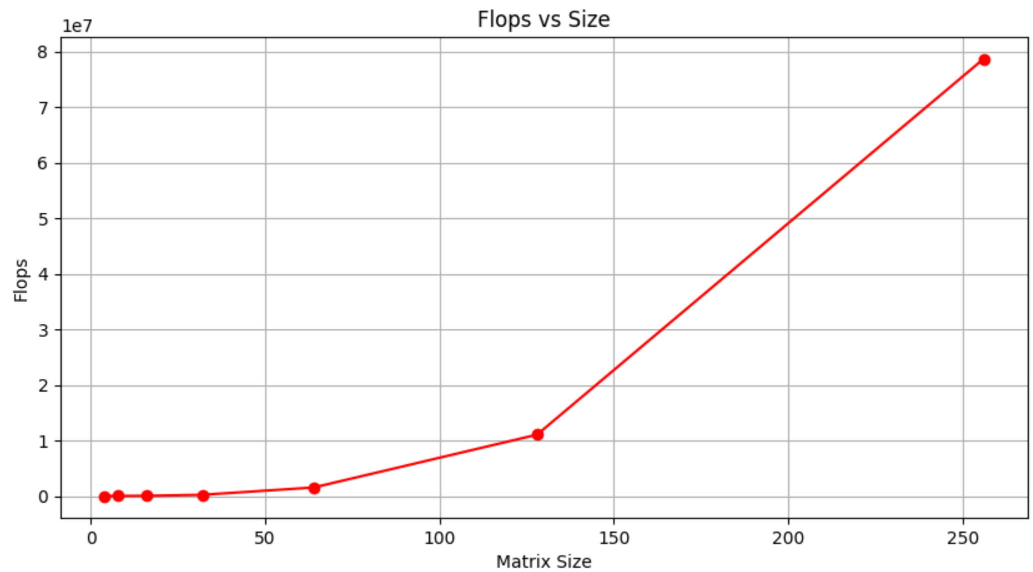
Pseudo kod:

💡
```
inverse(A):
    if A jest w wymiarach 2x2:
        return odwrócone A (według definicji)
    Podziel macierz A na macierze A11, A12, A21, A22S
    A11_inv = inverse(A11)
    S22 = A22 - A21 * A11_inv * A12
    S22_inv = inverse(S22)
    C1 = A11_inv * A12
    C2 = S22_inv * A21 * A11_inv
    B11 = A11_inv + C1 * C2
    B12 = - C1 * S22_inv
    B21 = - C2
    B22 = S22_inv
    Połącz B11, B12, B21, B22
    Return B
```

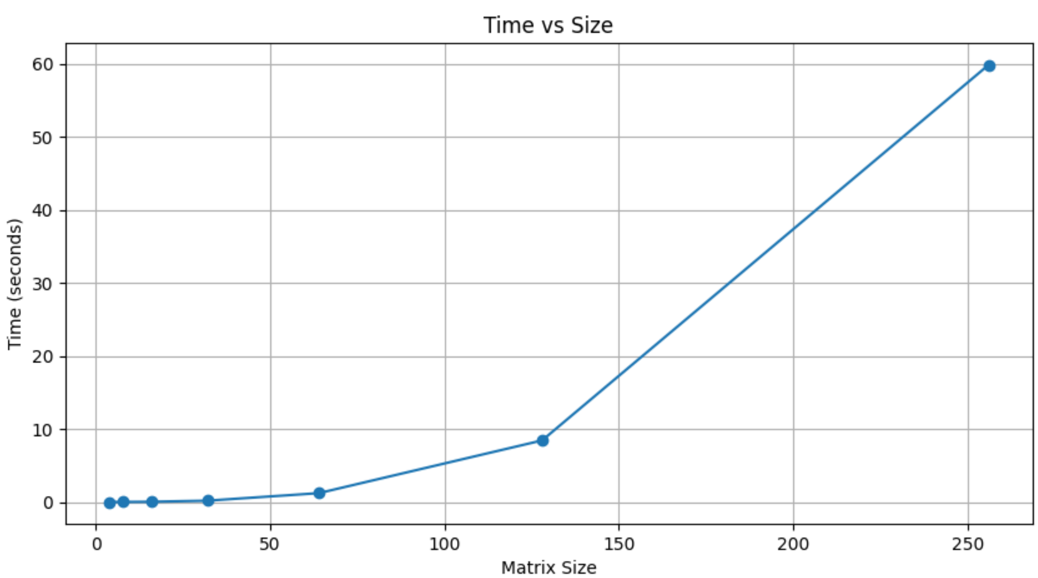
Wykresy:

Zrezygnowaliśmy z testów dla macierzy większych od 256x256 z powodu czasu wykonania algorytmu.

Wykres zależności działań zmiennoprzecinkowych od rozmiaru mnożonych macierzy:



Wykres zależności czasu od rozmiaru mnożonych macierzy:



Szacowanie złożoności:

Oszacowana złożonośc: `log2(n) * n^3`

Przyjęta złożonośc algorytmu mnożenia macierzy metoda strassena: `n^3`

> `log2(n)` ← wynika z faktu, że w każdym rekurencyjnym wykonaniu funkcji dzielimy macierz na 4 części z kótrych każda jest wymiarów S/2 x S/2 gdzie S x S jest

początkowym rozmiarem macierzy

`n^3` ← bierze się ze złożoności algorytmu mnożenia macierzy metoda strassena, której użyliśmy w implementacji. Jest ona wykonywana w każdym wywołaniu rekurencyjnym głównej funkcji.

Testowanie poprawności porównując wynik z matlabem:

Funkcje odwracającą macierz testowaliśmy korzystając z zależności:

`initial_matrix @ inverted_matrix = eye`

gdzie:

- `initial_matrix`- pierwotna macierz
- `inverted_matrix` - odwrócona macierz
- `eye` - macierz jednostkowa

Implementacja rekurencyjnego odwracania macierzy wraz z zliczaniem operacji zmiennoprzecinkowych i dostosowywaniem rozmiaru macierzy

Do implementacji wykorzystano metodę Strassena z poprzedniego laboratorium.

```
def __inverseInner(self, A: np.ndarray):
        if A.shape == (1, 1):
            x = 1 / A[0, 0]
            self.__flops += 1
            return np.array([[x]])

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

        # A11_inv = inverse(A11)
        A11_inv= self.__inverseInner(A11)

        # S22 = A22 - A21 @ A11_inv @ A12
        S22 = self.strassen.multiplyMatrices(A21, A11_inv)
        S22 = self.strassen.multiplyMatrices(S22, A12)
        S22 = A22 - S22
        self.__flops += math.prod(A.shape)

        # S22_inv = inverse(S22)
```

```
        S22_inv = self.__inverseInner(S22)

        # B11 = A11_inv @ (I + A12 @ S22_inv @ A21 @ A11_inv)
        I = np.eye(A11.shape[0])
        B11 = self.strassen.multiplyMatrices(A12, S22_inv)
        B11 = self.strassen.multiplyMatrices(B11, A21)
        B11 = self.strassen.multiplyMatrices(B11, A11_inv)
        B11 = I + B11
        self.__flops += A11.shape[0]
        B11 = self.strassen.multiplyMatrices(A11_inv, B11)

        # B12 = -1 * (A11_inv @ A12 @ S22_inv)
        B12 = self.strassen.multiplyMatrices(A11_inv, A12)
        B12 = self.strassen.multiplyMatrices(B12, S22_inv)
        B12 = -1 * B12
        self.__flops = math.prod(B12.shape)

        # B21 = -1 * (S22_inv @ A21 @ A11_inv)
        B21 = self.strassen.multiplyMatrices(S22_inv, A21)
        B21 = self.strassen.multiplyMatrices(B21, A11_inv)
        B21 = -1 * B21
        self.__flops = math.prod(B21.shape)

        return np.vstack((np.hstack((B11, B12)), np.hstack((B21, S22_inv))))
```

2. Rekurencyjna LU Faktoryzacja

Pseudo kod:

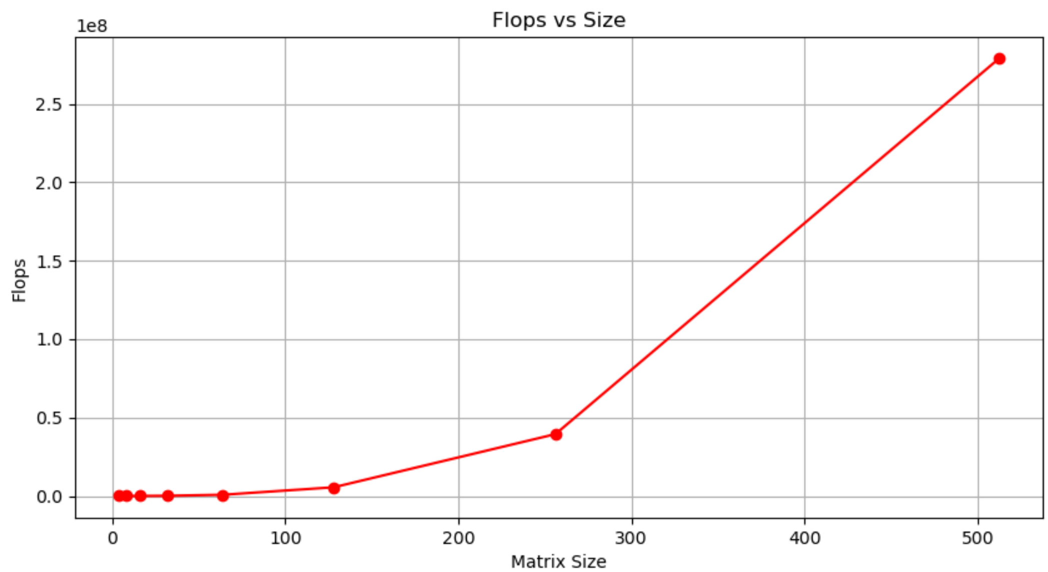
💡
```
LU(A):
    if A jest w wymiarach 2x2:
        return sfaktoryzowane A
    Podziel macierz A na macierze A11, A12, A21, A22
    L11, U11 = LU(A11)
    U11_inv = inverse(U11)
    L21 = A21 @ U11_inv
    L11_inv = inverse(L11)
    U12 = L11_inv @ A12
    S = A22 - A21 @ U11_inv @ L11_inv @ A12
    L22, U22 = LU(S)
    Połącz L11, L21, L22
    Połącz U11, U12, U22
```

Return L, U

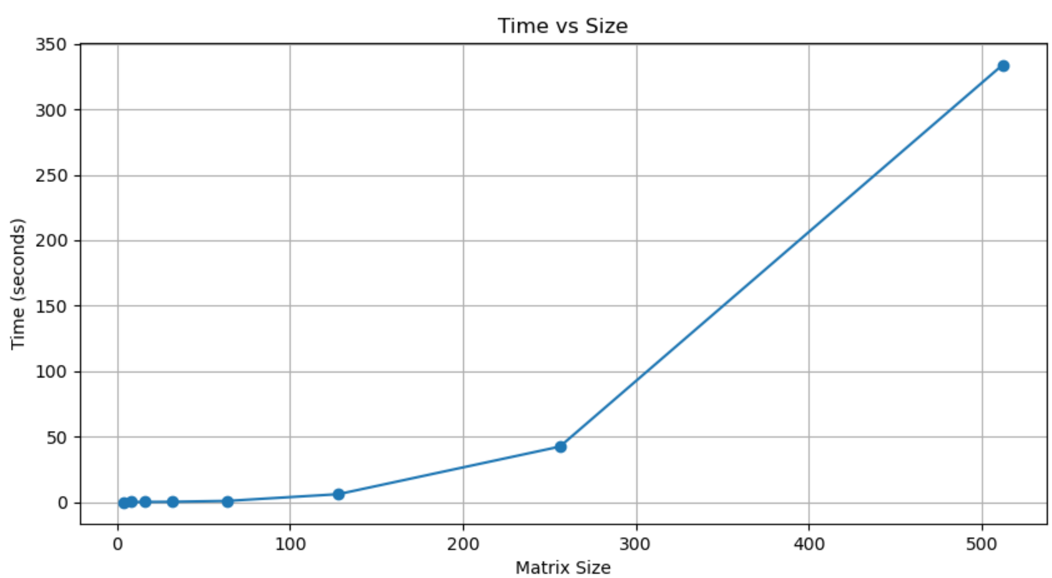
Wykresy:

Przeprowadzone analogicznie jak wyżej.

Wykres zależności działań zmiennoprzecinkowych od rozmiaru mnożonych macierzy:



Wykres zależności czasu od rozmiaru mnożonych macierzy:



Szacowanie złożoności:

Dwa rekurencyjne wywołania funkcji `O(n^2)`

Pięć mnożeń macierzy metodą Strassena `O(n^ln(7))`

Odpowiedź to `O(n^ln(7))`

Testowanie poprawności porównując wynik z matlabem:

Funkcję faktoryzującą macierz testowano mnożąc przez siebie macierze L i U wcześniej upewniwszy się, że mają one oczekiwaną postać:

> `L @ U = initial_matrix`

gdzie `initial_matrix` to pierwotna macierz, a `L` i `U` to macierze wynikowe algorytmu.

Implementacja algorytmu LU faktoryzacji w pythonie.

Do implementacji wykorzystano wcześniej zaimplementowaną metodę odwracającą macierz oraz metodę Strassena mnożenia macierzy z poprzedniego laboratorium.

```
def __LUInner(self, A: np.ndarray):
      if A.shape == (1, 1):
          return np.array([[1]]), A

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

      L11, U11 = self.__LUInner(A11)
      L11_inv = self.inverse.inverse(L11)
      U11_inv = self.inverse.inverse(U11)

      L21 = self.strassen.multiplyMatrices(A21, U11_inv)
      U12 = self.strassen.multiplyMatrices(L11_inv, A12)

      S = self.strassen.multiplyMatrices(A21, U11_inv)
      S = self.strassen.multiplyMatrices(S, L11_inv)
      S = self.strassen.multiplyMatrices(S, A12)
      S = A22 - S

      self.__flops += S.shape[0] ** 2 # one operation for every element on square matrix

      L22, U22 = self.__LUInner(S)

      return (
          np.vstack((np.hstack((L11, np.zeros_like(A12))), np.hstack((L21, L22)))),
```

```
        np.vstack((np.hstack((U11, U12)), np.hstack((np.zeros_like(A21), U22))))
    )
```

2. Rekurencyjne obliczanie wyznacznika

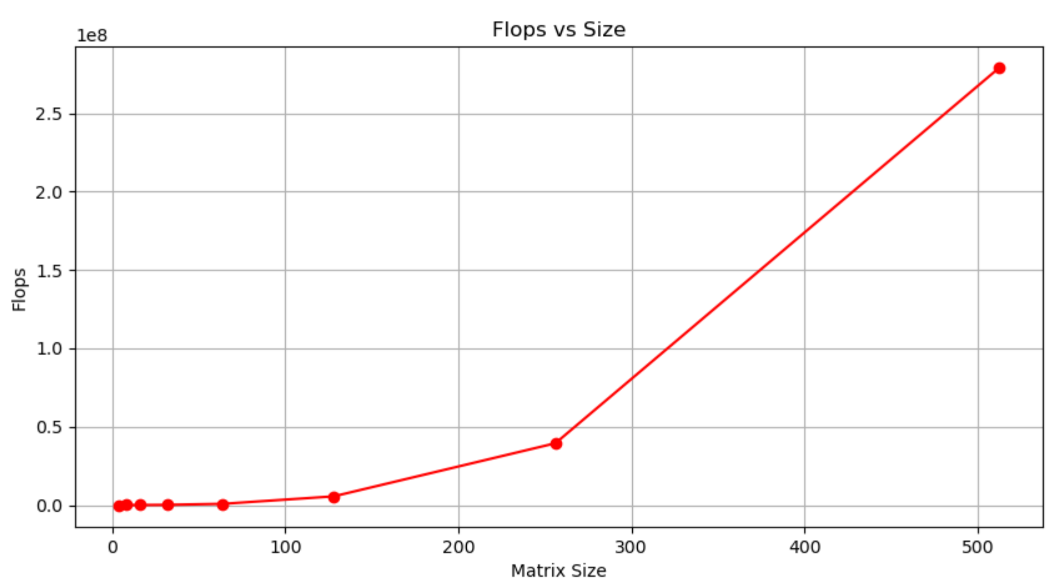
Pseudo kod:

> 💡 L, U = LU_faktoryzacja(A)
> Det = iloczyn_elementow_na_przekatnej(U)

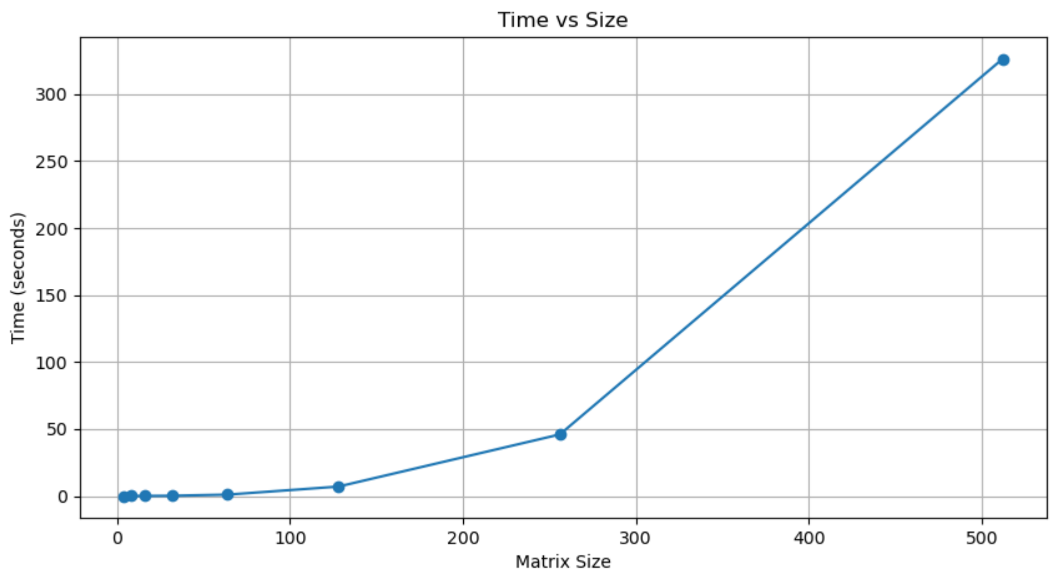
Wykresy:

Przeprowadzone analogicznie jak wyżej.

Wykres zależności działań zmiennoprzecinkowych od rozmiaru mnożonych macierzy:



Wykres zależności czasu od rozmiaru mnożonych macierzy:



Szacowanie złożoności:

Koszt faktoryzacji kodu to `O(n^ln(7))`

Koszt sumowania iloczynów to `O(n)`

Co daje złożoność `O(n^ln(7))`

Testowanie poprawności porównując wynik z matlabem:

Funkcję liczącą wyznacznik porównaliśmy licząc zwyczajnie wyznacznik odpowiadającą funkcję w matlabie.

Implementacja algorytmu liczącego wyznacznik w pythonie.

Do implementacji wykorzystano wcześniej zaimplementowaną metodę do LU faktoryzacji.

```
class DetCalculationEngine:
    __flops = 0
    lu = LUCalculationEngine()

    def resetCounter(self):
        self.__flops = 0
```

```
    def getFlops(self):
        return self.__flops

    def det(self, A: np.ndarray):
        self.resetCounter()
        self.lu.resetCounter()

        L, U = self.lu.LU(A)
        res = np.prod(np.diagonal(U))

        self.__flops += self.lu.getFlops() + A.shape[0] - 1

        return res
```