Algorytmy macierzowe

Laboratorium 2 - Rekurencyjne odwracanie macierzy, eliminacja Gaussa, LU faktoryzacja oraz liczenie wyznacznika

Kacper Wąchała, Szymon Hołysz

1. Zastosowane algorytmy

1.1. Rekurencyjne odwracanie macierzy

Algorithm 1: Odwracanie macierzy

1: **function** INVERSE(A)
2: $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$
3: $A_{11}^{-1} \leftarrow \text{INVERSE}(A_{11})$
4: $S_{22} \leftarrow A_{22} - A_{21}A_{11}^{-1}A_{12}$
5: $S_{22}^{-1} \leftarrow \text{INVERSE}(S_{22})$
6: $B_{11} \leftarrow A_{11}^{-1} + A_{11}^{-1}A_{12}S_{22}^{-1}A_{21}A_{11}^{-1}$
7: $B_{12} \leftarrow -A_{11}^{-1}A_{12}S_{22}^{-1}$
8: $B_{21} \leftarrow -S_{22}^{-1}A_{21}A_{11}^{-1}$
9: $B_{22} \leftarrow S_{22}^{-1}$
10: $B \leftarrow \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$
11: **return** B
12: **end**

1.2. Rekurencyjna eliminacja Gaussa

Algorithm 2: Eliminacja Gaussa

function INVERSE(A)
    $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$
    $\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \leftarrow b$
    $L_{11}, U_{11} \leftarrow \text{LU}(A_{11})$
    $L_{11}^{-1} \leftarrow$ INVERSE(L_{11})
    $U_{11}^{-1} \leftarrow$ INVERSE(U_{11})
    $S \leftarrow A_{22} - A_{21} U_{11}^{-1} L_{11}^{-1} A_{12}$
    $L_S, U_S \leftarrow \text{LU}(S)$
    $C_{11} \leftarrow U_{11}$
    $C_{12} \leftarrow L_{11} b_1$
    $C_{22} \leftarrow U_S$
    $L_S^{-1} \leftarrow$ INVERSE(L_S)
    $R_1 \leftarrow L_{11}^{-1} b_1$
    $L_S^{-1} \leftarrow$ INVERSE(L_S)
    $R_2 \leftarrow L_S^{-1} b_2 - L_S^{-1} A_{21} U_{11}^{-1} L_{11}^{-1} b_1$
    $C \leftarrow \begin{pmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{pmatrix}$
    $R \leftarrow \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$
    $m, n \leftarrow$ SHAPE(C)
    $o, p \leftarrow$ SHAPE(R)
    for $i < m$ **do**
        $f \leftarrow C[i][i]$
        for $j < n$ **do**
            $C[i][j] \leftarrow (C[i][j]) * \frac{1}{f}$
        end
        for $j < p$ **do**
            $R[i][j] \leftarrow (R[i][j]) * \frac{1}{f}$
        end
    end
    return C, R
end

1.3. Rekurencyjna faktoryzacja LU

Algorithm 3: Faktoryzacja LU

1: **function** LU(A)
2: $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$
3: $L_{11}, U_{11} \leftarrow \mathrm{LU}(A_{11})$
4: $U_{11}^{-1} \leftarrow \textsc{Inverse}(U_{11})$
5: $L_{21} \leftarrow A_{21} U_{11}^{-1}$
6: $L_{11}^{-1} \leftarrow \textsc{Inverse}(L_{11})$
7: $U_{12} \leftarrow L_{11}^{-1} A_{12}$
8: $L_{22} \leftarrow A_{22} - A_{21} U_{11}^{-1} L_{11}^{-1} A_{12}$
9: $L_{22}, U_{22} \leftarrow \mathrm{LU}(L_{22})$
10: $L \leftarrow \begin{pmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{pmatrix}$
11: $U \leftarrow \begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix}$
12: **return** L, U
13: **end**

1.4. Obliczanie wyznacznika

Algorithm 4: Obliczanie wyznacznika

```
1: function DET(A)
2:     L, U ← LU(A)
3:     result ← 0
4:     s ← SIZE(A)
5:     for i < s do
6:         result ← result * U_ii
7:     end
8:     return result
9: end
```

2. Implementacja

W celu realizacji obliczeń na macierzach została zaimplementowana klasa `Matrix` zawierająca metody elementarnych działań i porównywania macierzy, oraz metody mnożeń przedstawione poniżej.

2.1. Rekurencyjne odwracanie macierzy

```python
def inverse(self) -> Matrix:
    assert self.shape.cols == self.shape.rows
    def closest_power(n: int) -> int:
        n -= 1
        n |= n >> 1
        n |= n >> 2
        n |= n >> 4
        n |= n >> 8
        n |= n >> 16
        n += 1
        return n

    n = self.shape.rows
    if n == 1:
        assert self[0][0] != 0, "Can't inverse"
        OperationCounter.multiplications += 1
        return Matrix([[1/self[0][0]]])

    if n == 2:
        a, b = self[0]
        c, d = self[1]
        determ = a * d - b * c
        assert determ != 0, "Can't inverse"

        OperationCounter.multiplications += 2
        OperationCounter.additions += 1

        inv = Matrix([[d / determ, -b / determ],
                      [-c / determ, a / determ]])
        return inv

    m = closest_power(n)
    if m != n:
        padded = Matrix([row[:] + [0]*(m-n) for row in self])
        for _ in range(m-n):
            padded.append([0]*m)
        for i in range(n, m):
            padded[i][i] = 1
        padded_inv = padded.inverse()
        trimmed = Matrix([row[:n] for row in padded_inv[:n]])
        return trimmed

    p = n // 2

    a11 = Matrix([r[:p] for r in self[:p]])
    a12 = Matrix([r[p:] for r in self[:p]])
```

```
    a21 = Matrix([r[:p] for r in self[p:]])
    a22 = Matrix([r[p:] for r in self[p:]])

    a11_inv = a11.inverse()
    s22 = a22 - a21.binet(a11_inv).binet(a12)
    s22_inv = s22.inverse()

    b11 = a11_inv + a11_inv.binet(a12).binet(s22_inv).binet(a21).binet(a11_inv)
    b12 = -a11_inv.binet(a12).binet(s22_inv)
    b21 = -s22_inv.binet(a21).binet(a11_inv)
    b22 = s22_inv

    return Matrix.block([[b11, b12], [b21, b22]])
```

2.2. Rekurencyjna eliminacja Gaussa

```
def gauss(self, b: Matrix) -> tuple[Matrix, Matrix]:
    assert self.shape.cols == self.shape.rows
    assert self.shape.rows == b.shape.rows
    n = self.shape.rows
    p = n // 2

    a11 = Matrix([r[:p] for r in self[:p]])
    a12 = Matrix([r[p:] for r in self[:p]])
    a21 = Matrix([r[:p] for r in self[p:]])
    a22 = Matrix([r[p:] for r in self[p:]])
    b1 = Matrix(r for r in b[:p])
    b2 = Matrix(r for r in b[p:])

    l11, u11 = a11.lufac()
    l11_inv = l11.inverse()
    u11_inv = u11.inverse()

    s = a22 - a21.binet(u11_inv).binet(l11_inv).binet(a12)
    ls, us = s.lufac()

    c11 = u11
    c12 = l11_inv.binet(a12)
    c22 = us

    rhs1 = l11_inv.binet(b1)
    ls_inv = ls.inverse()
    rhs22 = ls_inv.binet(b2) - ls_inv.binet(a21).binet(u11_inv).binet(l11_inv).binet(b1)

    c = Matrix.block([[c11, c12], [zeros(c22.shape.rows), c22]])
    rhs = Matrix.block([[rhs1], [rhs22]])

    for i in range(LU.shape.rows):
        factor = c[i][i]
        for j in range(LU.shape.cols):
            c[i][j] /= factor
        for j in range(rhs.shape.cols):
            rhs[i][j] /= factor

    return c, rhs
```

2.3. Rekurencyjna LU faktoryzacja

```
def lufac(self) -> tuple[Matrix, Matrix]:
    rows, cols = self.shape
    if rows ≠ cols: raise "Macierz musi być kwadratowa"

    return self.recursive_lufac()

def recursive_lufac(self):
    n = self.shape.rows
    if n == 1:
        return Matrix([[0]]), self

    if n == 2:
        a11 = self[0][0]
        a12 = self[0][1]
        a21 = self[1][0]
        a22 = self[1][1]
        L = id(2)
        L[1][0] = a21 / a11

        u11 = a11
        u12 = a12
        u22 = a22 - a21 * a12 / a11

        return L, Matrix([[u11, u12], [0, u22]])

    p = n // 2

    a11 = Matrix([r[:p] for r in self[:p]])
    a12 = Matrix([r[p:] for r in self[:p]])
    a21 = Matrix([r[:p] for r in self[p:]])
    a22 = Matrix([r[p:] for r in self[p:]])

    l11, u11 = a11.recursive_lufac()
    u11i = u11.inverse()
    l21 = a21.binet(u11i)
    l11i = l11.inverse()
    u12 = l11i.binet(a12)
    l22 = a22 - a21.binet(u11i).binet(l11i).binet(a12)
    l22, u22  = l22.recursive_lufac()

    L = Matrix.block([[l11, zeros(p)], [l21, l22]])
    U = Matrix.block([[u11, u12], [zeros(p), u22]])

    return L, U
```

2.4. Obliczanie wyznacznika

```
def det(self) -> N:
    _, U = self.lufac()
    result = 1
    for i in range(U.shape.rows):
        result *= U[i][i]
    return result
```

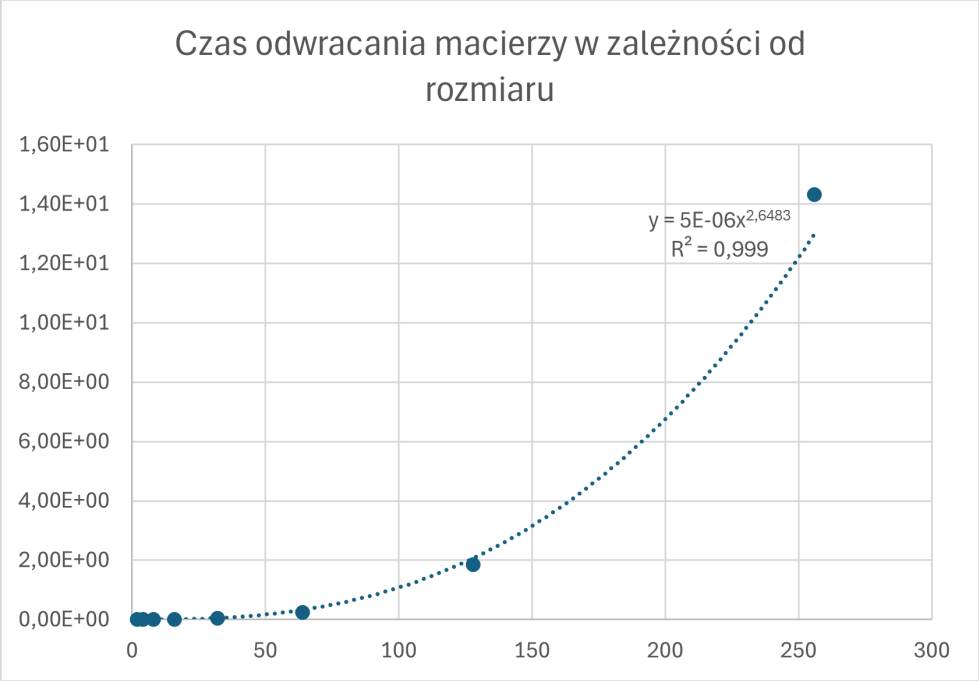
3. Wyniki

Aby zbadać efektywność zaimplementowanych metod rekurencyjnych, zmierzono czasy ich wykonania, a także liczbę wykonanych operacji zmiennoprzecinkowych dla rozmiarów macierzy 2^n, $n = 1, 2, 3, \ldots$

3.1. Rekurencyjne odwracanie macierzy

3.1.1. Czas wykonania

Poniżej przedstawiono wykres czasu wykonania algorytmu w zależności od rozmiaru macierzy:

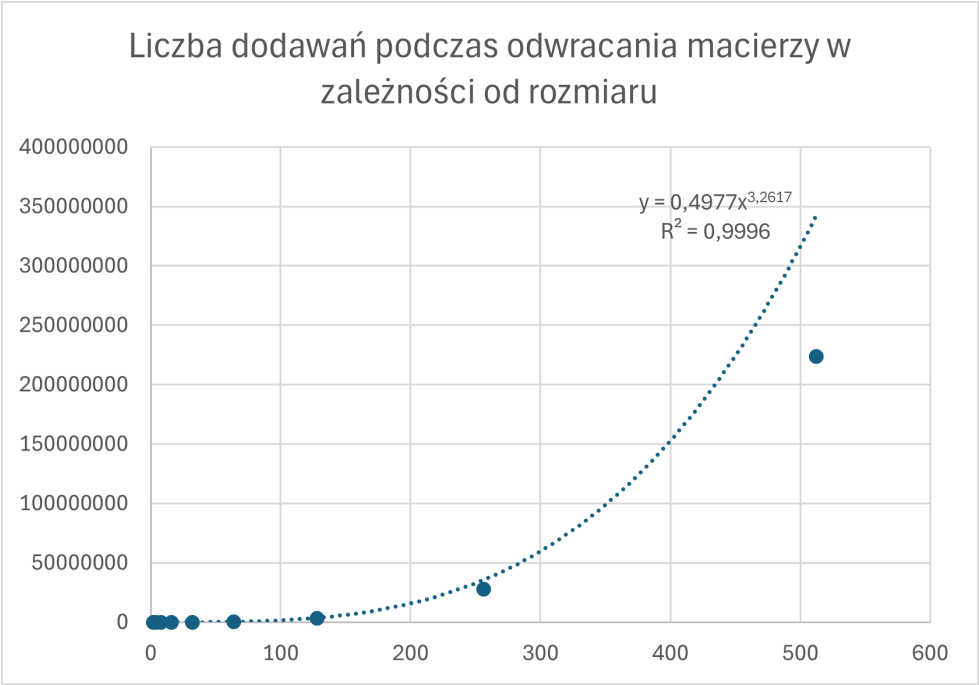


Rysunek 1: Wykres zależności czasu wykonania algorytmu rekurencyjnego odwracania względem rozmiaru macierzy

Na wykresie można zauważyć wykładniczy charakter przebiegu wykresu. Na podstawie otrzymanych wartości czasowych do wykresu dopasowano funkcję wykładniczą. Wyznaczono empirycznie wykładnik sugerujący złożoność czasową rzędu $O(n^{2.65})$, natomiast można ten wynik uznać za zbyt optymistyczny, ponieważ w implementacji algorytmu wykorzystano mnożenie macierzy w złożoności $O(n^3)$.

3.1.2. Liczba dodawań

Wykres poniżej prezentuje liczbę wykonanych operacji dodawania w trakcie wykonania algorytmu w zależności od rozmiaru macierzy.

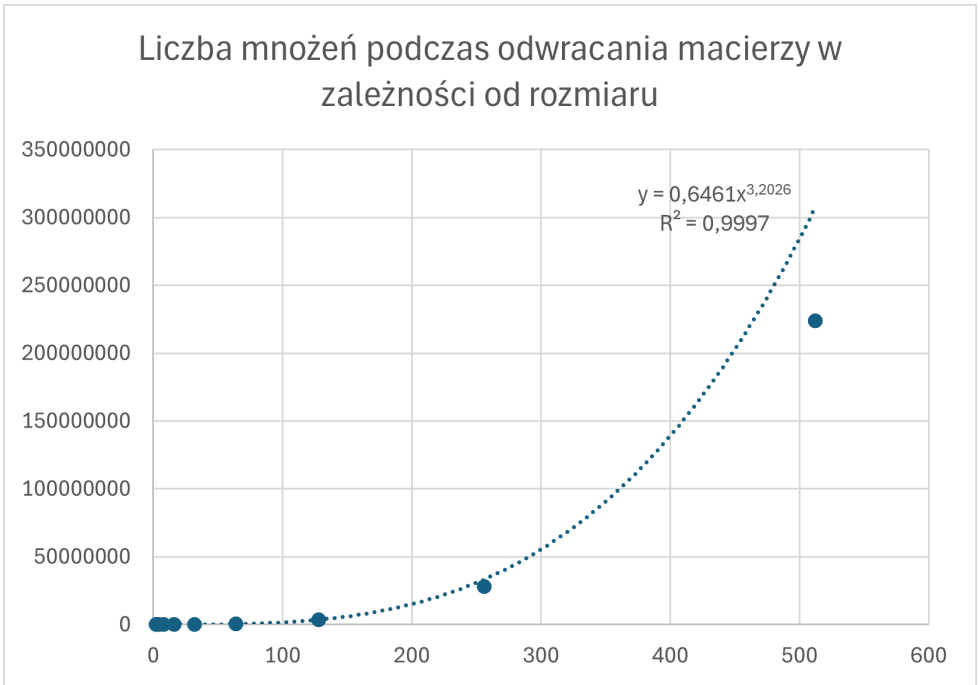


Rysunek 2: Wykres zależności liczby dodawań względem rozmiaru macierzy dla algorytmu rekurencyjnego odwracania

Do otrzymanych wartości również dopasowano funkcję wykładniczą, oszacowany wykładnik – 3.26.

3.1.3. Liczba mnożeń

Na poniższym wykresie zobrazowano zależność liczby wykonanych operacji mnożenia od rozmiaru macierzy.



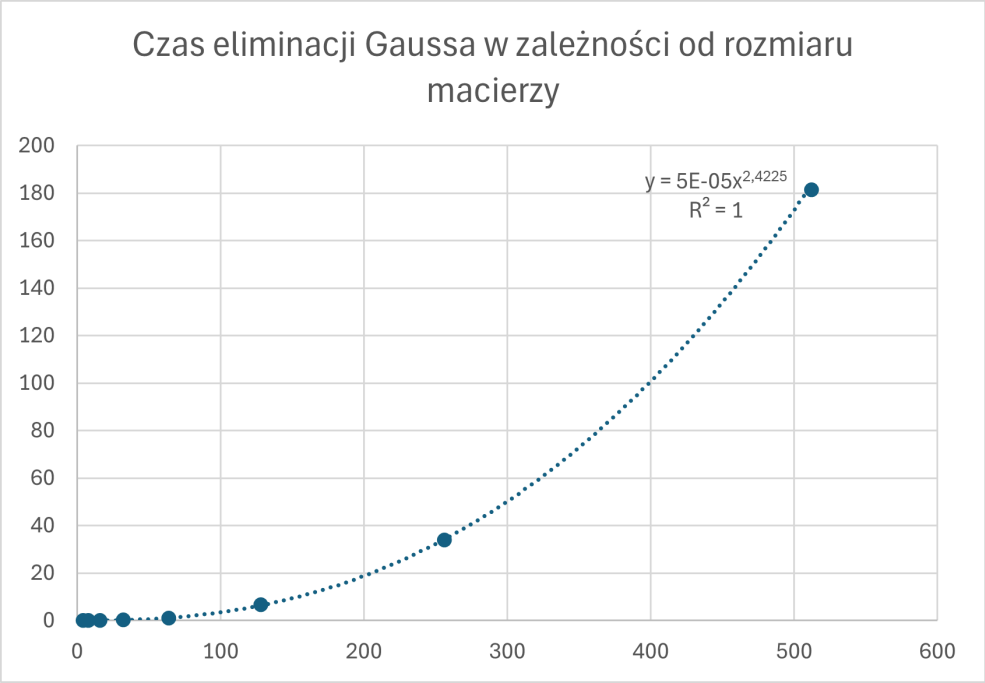
Rysunek 3: Wykres zależności liczby mnożeń względem rozmiaru macierzy dla algorytmu rekurencyjnego odwracania

Dopasowana funkcja wykładnicza sugeruje złożoność rzędu $O(n^{3.20})$.

3.2. Rekurencyjna eliminacja Gaussa

3.2.1. Czas wykonania

Zamieszczony wykres ukazuje zależność czasu wykonania algorytmu rekurencyjnej eliminacji Gaussa od rozmiaru macierzy ze współczynnikami:

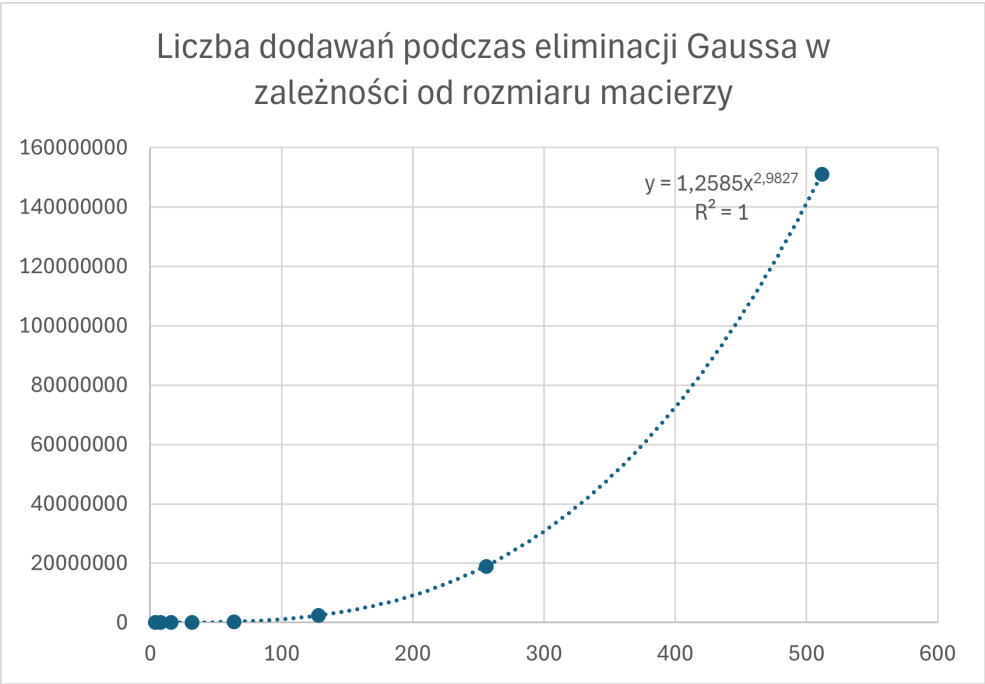


Rysunek 4: Wykres zależności czasu wykonania względem rozmiaru macierzy dla algorytmu eliminacji Gaussa

Do otrzymanych wartości dopasowano funkcję wykładniczą. Znaleziony wykładnik sugeruje złożoność czasową rzędu $O(n^{2.42})$, natomiast ponownie wynik ten może być zaniżony, ponieważ w implementacji algorytmu wykorzystano tradycyjne mnożenie $O(n^3)$.

3.2.2. Liczba dodawań

Poniżej przedstawiono wykres liczby wykonanych operacji dodawania w trakcie algorytmu rekurencyjnej eliminacji Gaussa w zależności od rozmiaru macierzy ze współczynnikami:

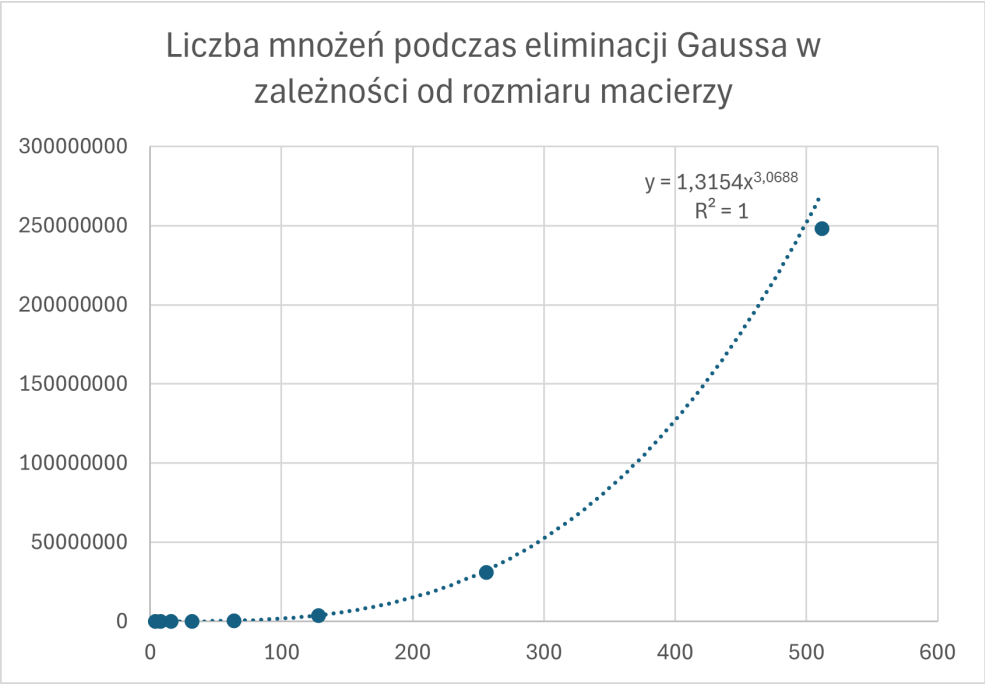


Rysunek 5: Wykres zależności liczby dodawań względem rozmiaru macierzy dla algorytmu eliminacji Gaussa

Wyznaczony empirycznie wykładnik funkcji wykładniczej dopasowanej do wykresu pozwala oszacować złożoność na $O(n^{2.98})$, co jest rezultatem bardzo zbliżonym do oczekiwanej złożoności $O(n^3)$.

3.2.3. Liczba mnożeń

Na poniższym wykresie pokazano zależność liczby mnożeń względem rozmiaru macierzy ze współczynnikami:



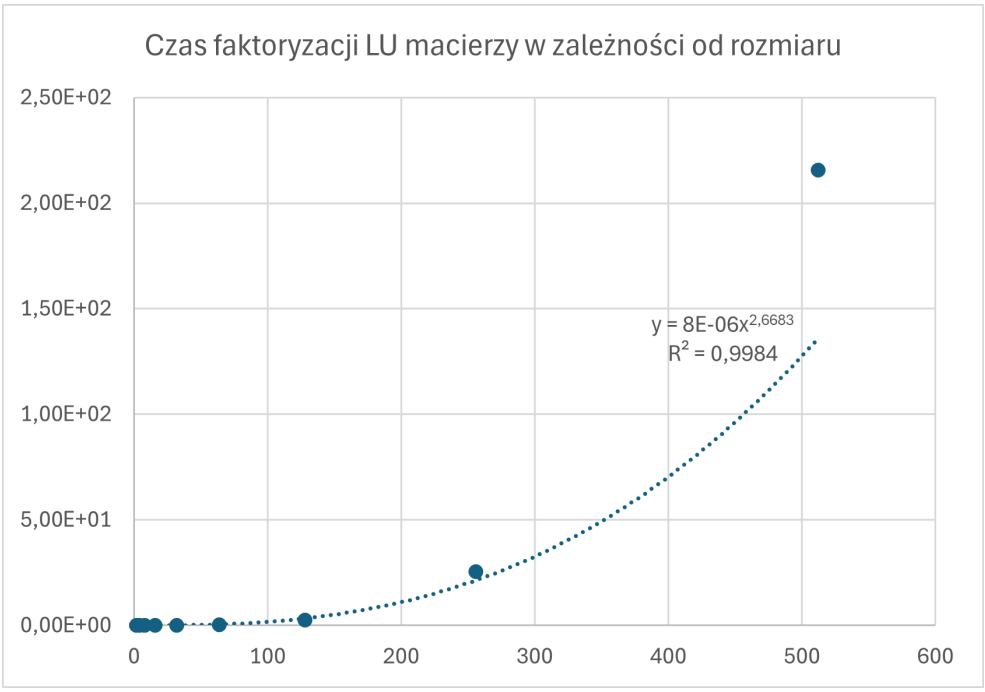
Rysunek 6: Wykres zależności liczby mnożeń względem rozmiaru macierzy dla algorytmu eliminacji Gaussa

Podobnie jak w przypadku liczby dodawań, wyznaczony wykładnik może sugerować złożoność obliczeniową rzędu $O(n^{3.06})$ – wynik również bliski oczekiwanemu $O(n^3)$.

3.3. Rekurencyjna LU faktoryzacja

3.3.1. Czas wykonania

Poniżej ukazano wykres zależności czasu wykonania algorytmu rekurencyjnej LU faktoryzacji od rozmiaru macierzy:

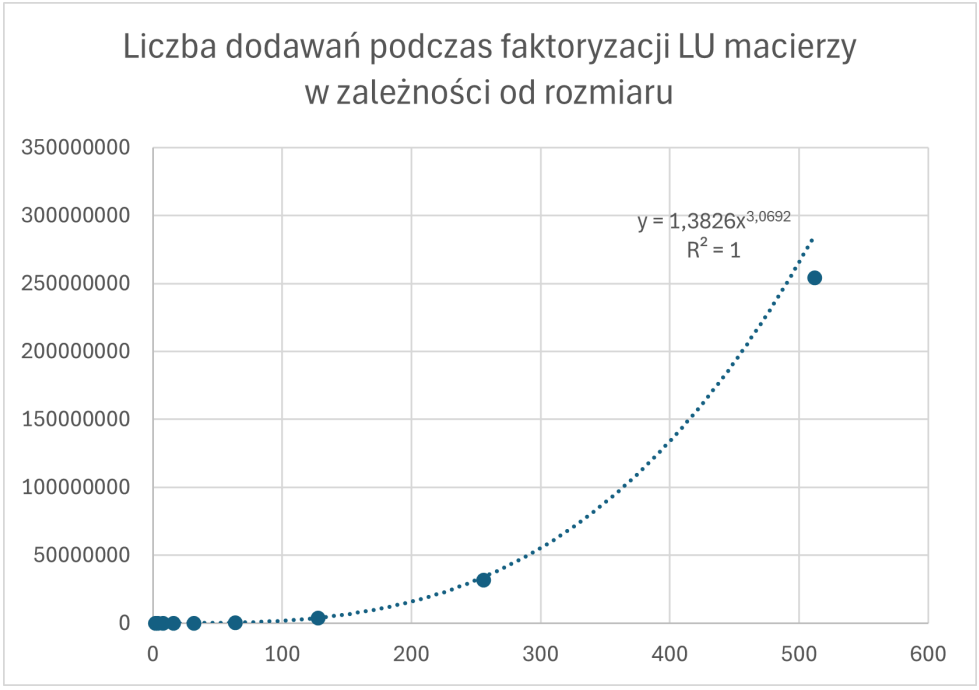


Rysunek 7: Wykres zależności czasu wykonania względem rozmiaru macierzy dla algorytmu rekurencyjnej LU faktoryzacji

Do uzyskanych wartości pomiaru czasu została dopasowana funkcja wykładnicza. Wyznaczono empirycznie wykładnik pozwalający oszacować rząd złożoności czasowej algorytmu na $O(n^{2.67})$. Tak mała wartość wykładnika zapewne ponownie wynika z niewystarczająco dobrego dopasowania krzywej wykładniczej.

3.3.2. Liczba dodawań

Na poniższym wykresie zobrazowano zależność liczby wykonanych operacji dodawania od rozmiaru macierzy:

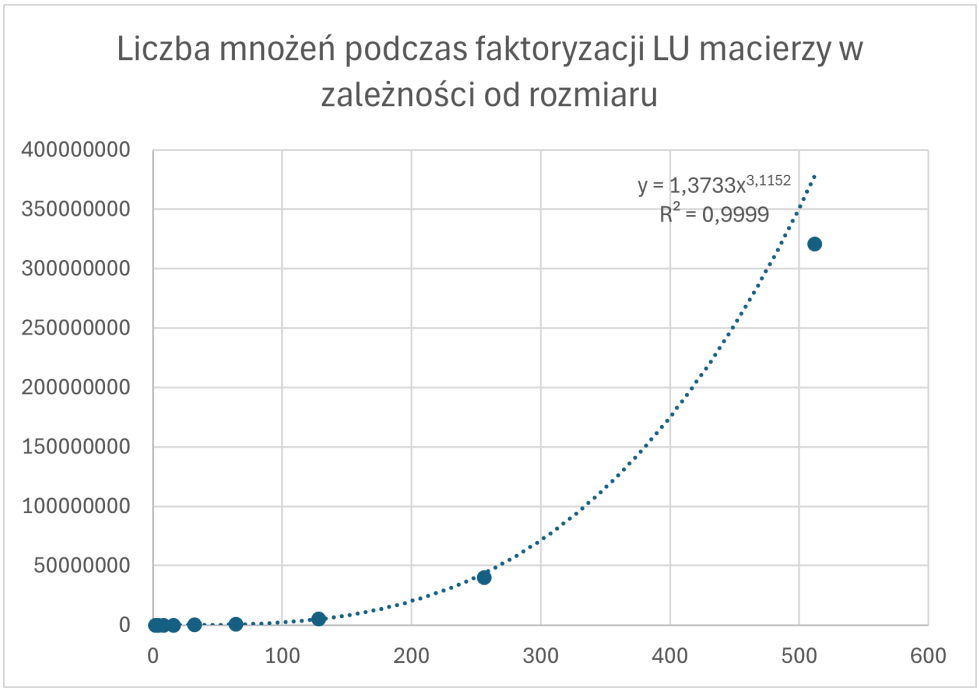


Rysunek 8: Wykres zależności liczby dodawań względem rozmiaru macierzy dla algorytmu rekurencyjnej LU faktoryzacji

W tym przypadku wartość wykładnika została oszacowana na ok. 3.07.

3.3.3. Liczba mnożeń

Poniżej przedstawiono wykres zależności liczby wykonanych mnożeń względem rozmiaru macierzy:



Rysunek 9: Wykres zależności liczby mnożeń względem rozmiaru macierzy dla algorytmu rekurencyjnej LU faktoryzacji

Wyznaczona empirycznie wartość wykładnika funkcji wykładniczej sugeruje złożoność rzędu $O(n^{3.12})$

4. Wnioski

Wykonane zadania pozwoliły zapoznać się z algorytmami odwracania macierzy, eliminacji Gaussa, a także LU faktoryzacji w ich rekurencyjnych wariantach. Oszacowano także złożoności obliczeniowe powyższych algorytmów. Jest możliwe polepszenie złożoności głównie poprzez wykorzystanie innych algorytmów mnożenia, innych niż wykorzystane w tych implementacjach klasyczne mnożenie $O(n^3)$ (np. algorytm Strassena, metody wyprowadzone przez AI AlphaTensor).