Odwracanie macierzy, Dekompozycja LU, Eliminacja Gaussa

Maciej Borowiec

23.11.2025

1 Odwracanie macierzy

1.1 Przebieg algorytmu

- Jeśli macierz wejściowa A jest o wymiarze 1x1 - zwracamy macierz 1x1 z odwrotną wartością
- W przeciwnym wypadku:
  - Dzielimy macierz na podmacierze:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

  - Obliczamy rekurencyjnie A_{11}^{-1}
  - Obliczamy $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ używając rekurencyjnego mnożenia macierzy
  - Obliczamy rekurencyjnie S^{-1}
  - Obliczamy podmacierze macierzy wynikowej (mnożenie macierzy wykonujemy rekurencyjnie):

$$B_{11} = A_{11}^{-1}(I + A_{12}S^{-1}A_{21}A_{11}^{-1})$$

$$B_{12} = -A_{11}^{-1}A_{12}S^{-1}$$

$$B_{21} = -S^{-1}A_{21}A_{11}^{-1}$$

$$B_{22} = S^{-1}$$

  - Zwracamy macierz wynikową:

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

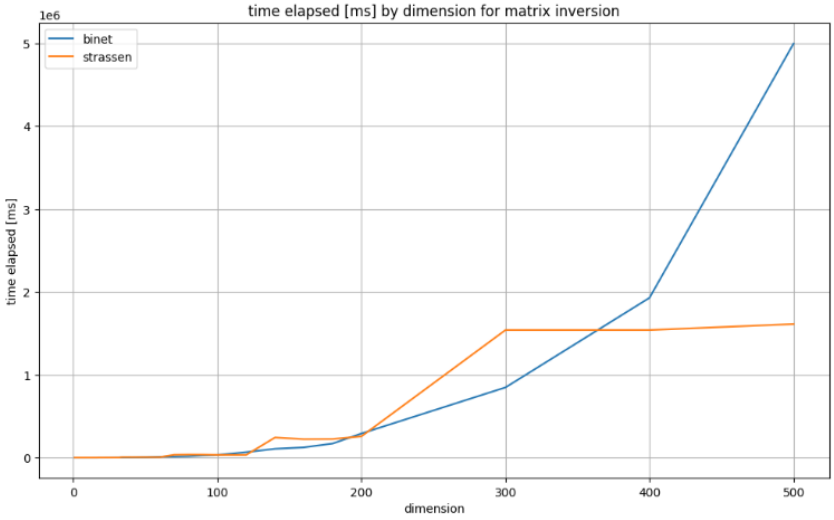
1.2 Kod

```
# invert matrix A
def inverse(A, mul_fun):
    flop_count = 0

    def f(A):
        nonlocal flop_count
        # initialize result matrix as zeros
        B = np.zeros(A.shape)
        # if matrix can be split further, split it
        if A.shape[0] > 1:
            # split matrix
            A11, A12, A21, A22 = split_matrix(A)
            # calculate inv(A11)
            A11_inv = f(A11)
            # calculate S (aka S22)
            M1, f1 = mul_fun(A21, A11_inv)
            M2, f2 = mul_fun(M1, A12)
            S = A22 - M2
            flop_count += f1 + f2 + el_count(S)
            # calculate inv(S)
            S_inv = f(S)
            # calculate B11
            M1, f1 = mul_fun(A12, S_inv)
            M2, f2 = mul_fun(M1, A21)
            M3, f3 = mul_fun(M2, A11_inv)
            B11, f4 = mul_fun(A11_inv, np.eye(M3.shape[0]) + M3)
            flop_count += f1 + f2 + f3 + f4 + M3.shape[0]
            # calculate B12
            M1, f1 = mul_fun(A11_inv, A12)
            M2, f2 = mul_fun(M1, S_inv)
            B12 = -M2
            flop_count += f1 + f2
            # calculate B21
            M1, f1 = mul_fun(S_inv, A21)
            M2, f2 = mul_fun(M1, A11_inv)
            B21 = -M2
            flop_count += f1 + f2
            # "calculate" B22
            B22 = S_inv
            # write the result into new matrix
            half_r, half_c = B11.shape
            B[:half_r, :half_c] = B11
            B[:half_r, half_c:] = B12
            B[half_r:, :half_c] = B21
            B[half_r:, half_c:] = B22
        # if you can no longer split matrix - scalar inversion
        else:
            B[0,0] = 1 / A[0,0]
            flop_count += 1
        # return resulting matrix
        return B

    return f(A), flop_count
```

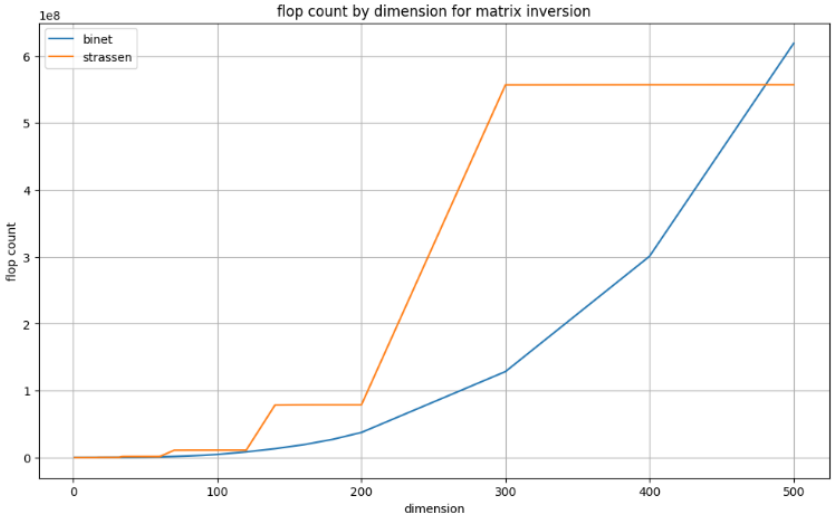
Listing 1: Algorytm odwracania macierzy

1.3 Wykresy przedstawiające wydajność algorytmu

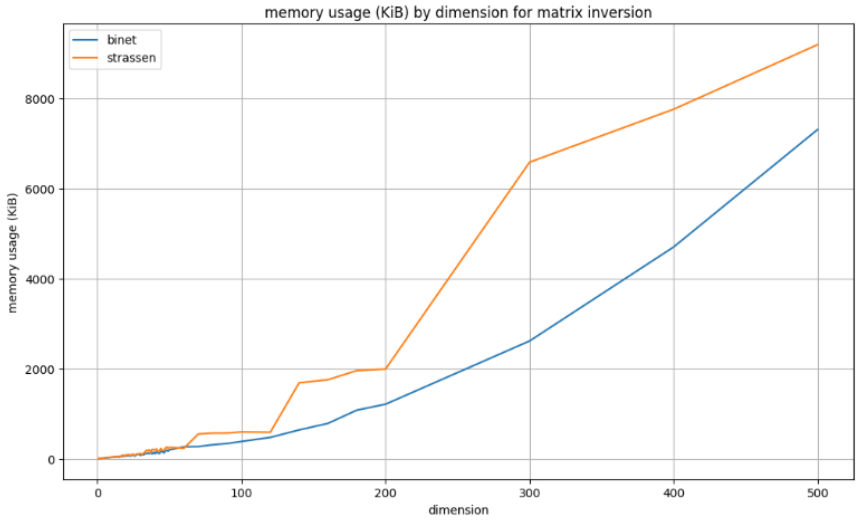
Poniżej zostały przedstawione wykresy pokazujące czas trwania, liczbę operacji zmiennoprzecinkowych, zużycie pamięci i liczbę operacji zmiennoprzecinkowych na sekundę w zależności od rozmiaru macierzy.



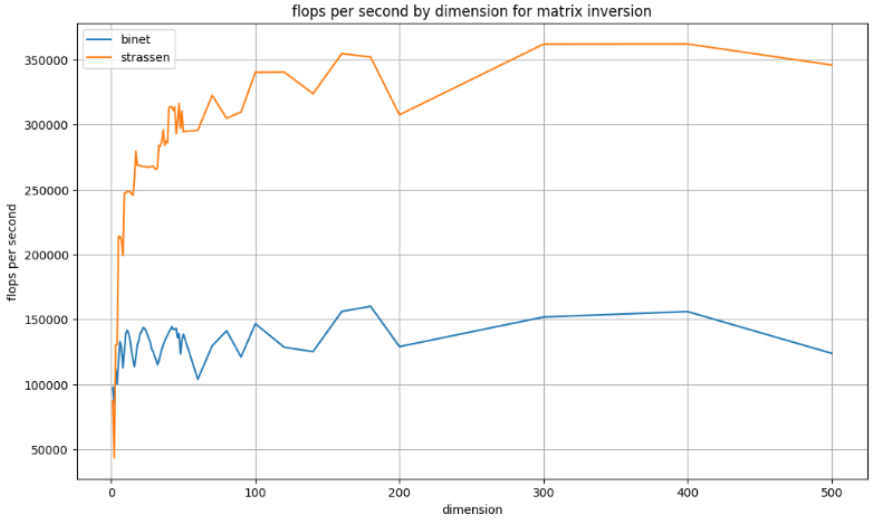
Rysunek 1: Czas wykonania algorytmu w zależności od rozmiaru macierzy



Rysunek 2: Liczba operacji zmiennoprzecinkowych algorytmu w zależności od rozmiaru macierzy



Rysunek 3: Zużycie pamięci algorytmu w zależności od rozmiaru macierzy



Rysunek 4: Liczba operacji zmiennoprzecinkowych na sekundę algorytmu w zależności od rozmiaru macierzy

2 Dekompozycja LU i wyznacznik macierzy

2.1 Przebieg algorytmu

- Jeśli macierz wejściowa A jest o wymiarze 1x1 - zwracamy $L = [1]$ i $U = A$
- W przeciwnym wypadku:
  - Dzielimy macierz na podmacierze:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

  - Obliczamy rekurencyjnie $L_{11}, U_{11} = LU(A_{11})$
  - Obliczamy L_{11}^{-1} i U_{11}^{-1} używając rekurencyjnego odwracania macierzy
  - Obliczamy L_{21} i U_{12} (mnożenie macierzy wykonujemy rekurencyjnie):

$$L_{21} = A_{21} U_{11}^{-1}$$

$$U_{12} = L_{11}^{-1} A_{12}$$

  - Obliczamy $S = A_{22} - A_{21} U_{11}^{-1} L_{11}^{-1} A_{12}$ używając rekurencyjnego mnożenia macierzy
  - Obliczamy rekurencyjnie $L_{22}, U_{22} = LU(S)$
  - Zwracamy macierze wynikowe:

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

- Możemy również obliczyć wyznacznik jako iloczyn wszystkich wartości na przekątnych macierzy L i U

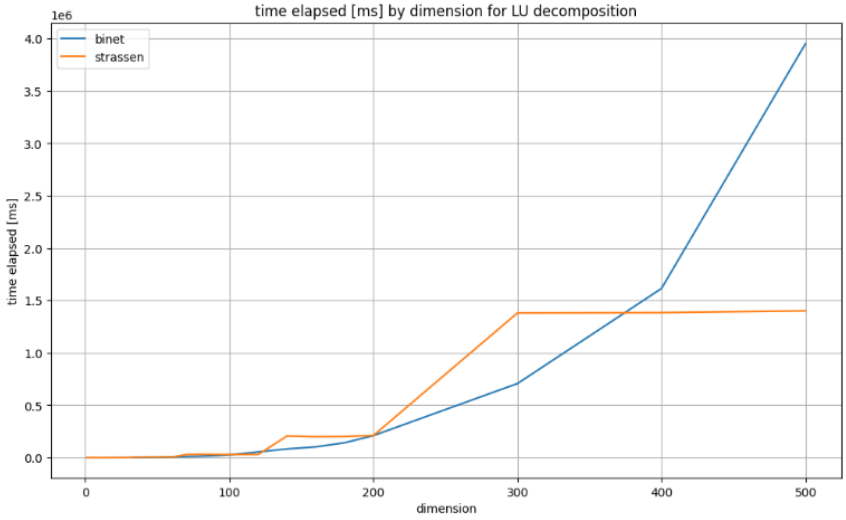
2.2 Kod

```
# decompose A to LU and calculate determinant
def LU(A, mul_fun):
    flop_count = 0

    def f(A):
        nonlocal flop_count
        # initialize result matrices as zeros
        L = np.zeros(A.shape)
        U = np.zeros(A.shape)
        # if matrix can be split further, split it
        if A.shape[0] > 1:
            # split matrix
            A11, A12, A21, A22 = split_matrix(A)
            # calculate L11 and U11
            L11, U11 = f(A11)
            # calculate inv(U11)
            U11_inv, f1 = inverse(U11, mul_fun)
            flop_count += f1
            # calculate L21
            L21, f1 = mul_fun(A21, U11_inv)
            flop_count += f1
            # calculate inv(L11)
            L11_inv, f1 = inverse(L11, mul_fun)
            flop_count += f1
            # calculate U12
            U12, f1 = mul_fun(L11_inv, A12)
            flop_count += f1
            # calculate S
            M1, f1 = mul_fun(A21, U11_inv)
            M2, f2 = mul_fun(M1, L11_inv)
            M3, f3 = mul_fun(M2, A12)
            S = A22 - M3
            flop_count += f1 + f2 + f3 + el_count(S)
            # calculate L22 and U22
            L22, U22 = f(S)
            # write the results into new matrices
            half_r, half_c = A11.shape
            L[:half_r, :half_c] = L11
            L[half_r:, :half_c] = L21
            L[half_r:, half_c:] = L22
            U[:half_r, :half_c] = U11
            U[:half_r, half_c:] = U12
            U[half_r:, half_c:] = U22
        # if you can no longer split matrix - trivial
        else:
            L[0,0] = 1
            U[0,0] = A[0,0]
        # return resulting matrix
        return L, U

    L, U = f(A)
    det = 1
    for i in range(A.shape[0]):
        det *= L[i,i] * U[i,i]
    return L, U, det, flop_count
```

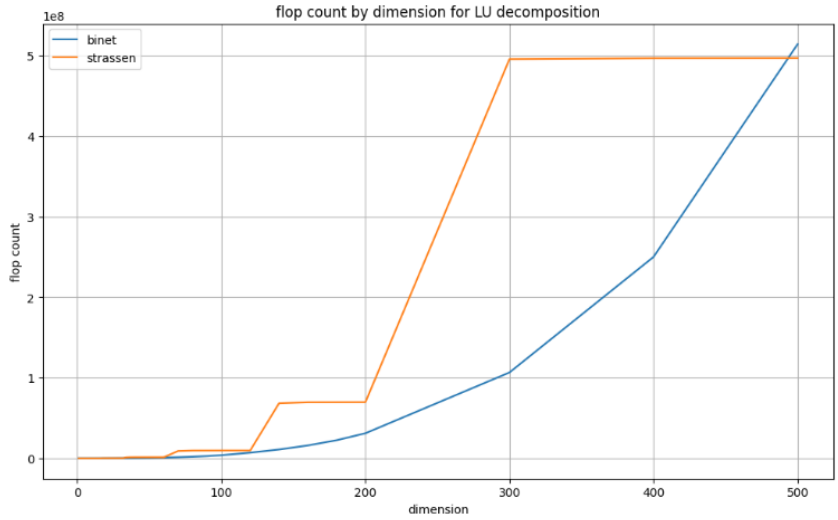
Listing 2: Algorytm dekompozycji LU

2.3 Wykresy przedstawiające wydajność algorytmu

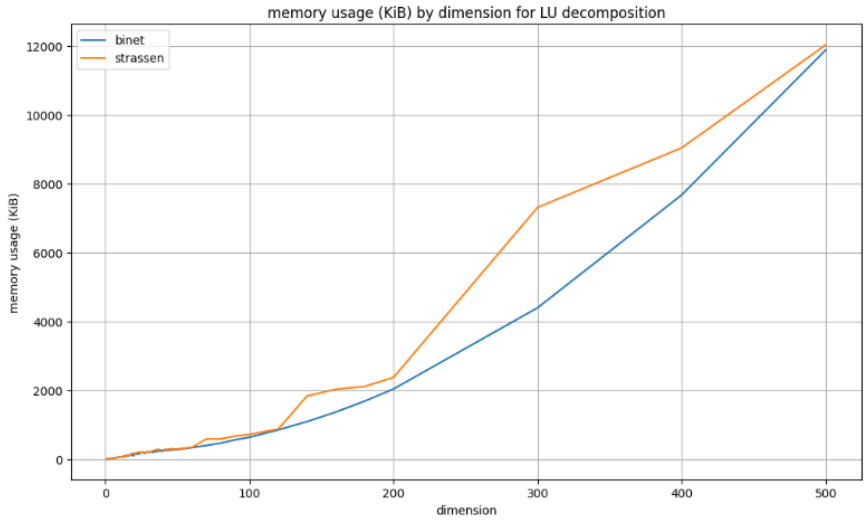
Poniżej zostały przedstawione wykresy pokazujące czas trwania, liczbę operacji zmiennoprzecinkowych, zużycie pamięci i liczbę operacji zmiennoprzecinkowych na sekundę w zależności od rozmiaru macierzy.



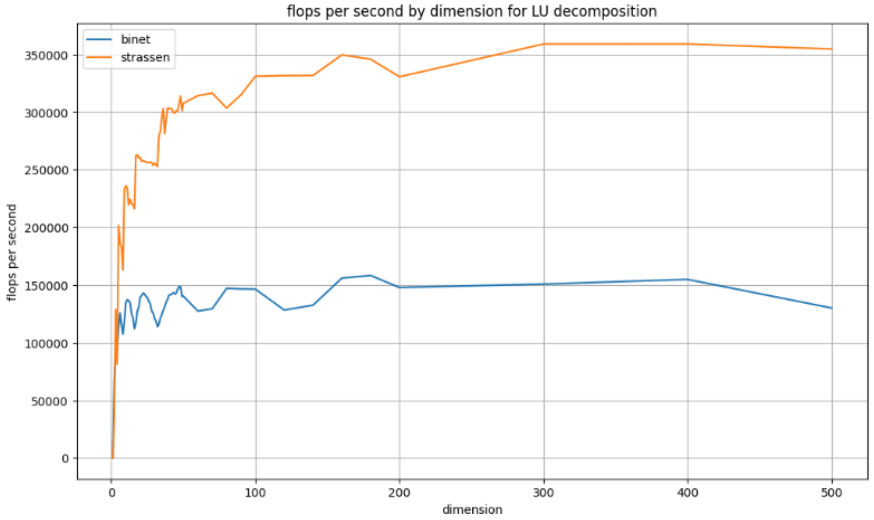
Rysunek 5: Czas wykonania algorytmu w zależności od rozmiaru macierzy



Rysunek 6: Liczba operacji zmiennoprzecinkowych algorytmu w zależności od rozmiaru macierzy



Rysunek 7: Zużycie pamięci algorytmu w zależności od rozmiaru macierzy



Rysunek 8: Liczba operacji zmiennoprzecinkowych na sekundę algorytmu w zależności od rozmiaru macierzy

3 Eliminacja Gaussa

3.1 Przebieg algorytmu

- Przyjmujemy na wejściu macierz A i wektor b oznaczające układ równań $Ax = b$, gdzie x to wektor niewiadomych, których wartości chcemy znaleźć
- Jeśli A i b są o wymiarach 1x1, $A = [a_{11}]$ i $b = [b_1]$ - zwracamy $x = [\frac{b_1}{a_{11}}]$
- W przeciwnym wypadku:
  - Dzielimy komponenty układu:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

  - Obliczamy $L_{11}, U_{11} = LU(A_{11})$ używając rekurencyjnej dekompozycji LU
  - Obliczamy L_{11}^{-1} i U_{11}^{-1} używając rekurencyjnego odwracania macierzy
  - Obliczamy $S = A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12}$ używając rekurencyjnego mnożenia macierzy
  - Obliczamy rekurencyjnie $L_S, U_S = LU(S)$
  - Obliczamy L_S^{-1} używając rekurencyjnego odwracania macierzy
  - Obliczamy komponenty układu po eliminacji Gaussa (mnożenie macierzy wykonujemy rekurencyjnie):

$$C_{11} = U_{11},\ C_{12} = L_{11}^{-1}A_{12},\ C_{22} = U_S$$

$$RHS_1 = L_{11}^{-1}b_1,\ RHS_2 = L_S^{-1}b_2 - L_S^{-1}A_{21}U_{11}^{-1}L_{11}^{-1}b_1$$

  - Rozwiązujemy rekurencyjnie nowy układ:

$$\begin{bmatrix} C_{11} & C_{12} \\ 0 & C_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} RHS_1 \\ RHS_2 \end{bmatrix}$$

    czyli:

$$x_2 = \textbf{Gauss}(C_{22}, RHS_2)$$

$$x_1 = \textbf{Gauss}(C_{11}, RHS_1 - C_{12}x_2)$$

  - Zwracamy wektor wynikowy:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

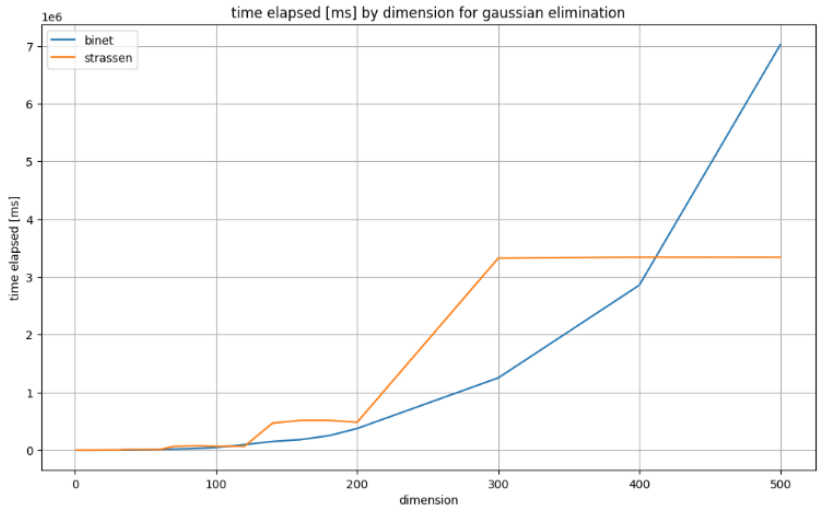
3.2 Kod

```
# solve linear equation Ax=b using gaussian elimination
def gauss(A, b, mul_fun):
    flop_count = 0

    def f(A, b):
        nonlocal flop_count
        # initialize result vector as zeros
        x = np.zeros(b.shape)
        # if matrix and vector can be split further, split them
        if A.shape[0] > 1:
            # split matrix and vector
            half_n = A.shape[0]//2
            A11, A12, A21, A22 = split_matrix(A)
            b1, b2 = b[:half_n], b[half_n:]
            # calculate L11 and U11
            L11, U11, _, f1 = LU(A11, mul_fun)
            flop_count += f1
            # calculate inv(L11) and inv(U11)
            L11_inv, f1 = inverse(L11, mul_fun)
            U11_inv, f2 = inverse(U11, mul_fun)
            flop_count += f1 + f2
            # calculate S
            M1, f1 = mul_fun(A21, U11_inv)
            M2, f2 = mul_fun(M1, L11_inv)
            M3, f3 = mul_fun(M2, A12)
            S = A22 - M3
            flop_count += f1 + f2 + f3 + el_count(S)
            # calculate LS and US
            LS, US, _, f1 = LU(S, mul_fun)
            flop_count += f1
            # calculate C
            C11 = U11
            C12, f1 = mul_fun(L11_inv, A12)
            C22 = US
            flop_count += f1
            # calculate RHS1
            RHS1, f1 = mul_fun(L11_inv, b1)
            flop_count += f1
            # calculate RHS2
            LS_inv, f1 = inverse(LS, mul_fun)
            M1, f2 = mul_fun(LS_inv, b2)
            M2, f3 = mul_fun(LS_inv, A21)
            M3, f4 = mul_fun(M2, U11_inv)
            M4, f5 = mul_fun(M3, L11_inv)
            M5, f6 = mul_fun(M4, b1)
            RHS2 = M1 - M5
            flop_count += f1 + f2 + f3 + f4 + f5 + f6 + el_count(RHS2)
            # calculate xs
            x2 = f(C22, RHS2)
            M1, f1 = mul_fun(C12, x2)
            x1 = f(C11, RHS1 - M1)
            flop_count += f1 + el_count(M1)
            # assign to x
            x[:half_n] = x1
            x[half_n:] = x2
        # if you can no longer split matrix - trivial
        else:
            x[0] = b[0] / A[0,0]
        # return resulting matrix
        return x

    return f(A, b), flop_count
```

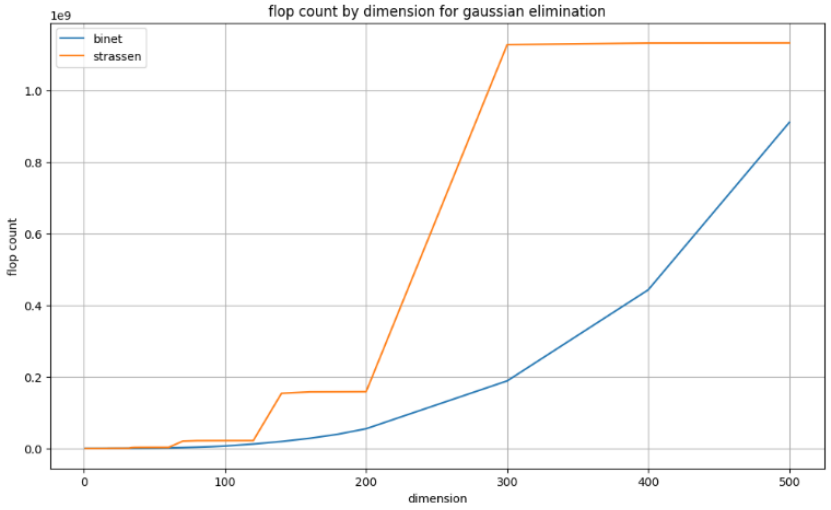
Listing 3: Algorytm eliminacji Gaussa

3.3 Wykresy przedstawiające wydajność algorytmu

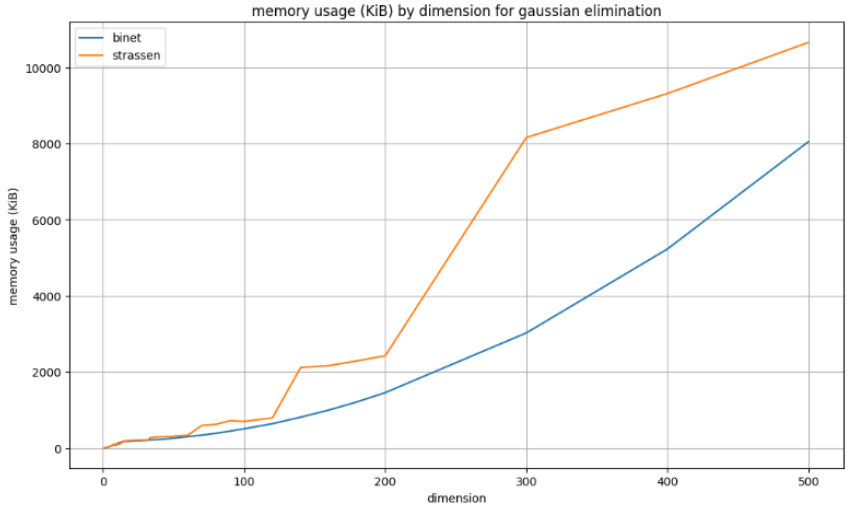
Poniżej zostały przedstawione wykresy pokazujące czas trwania, liczbę operacji zmiennoprzecinkowych, zużycie pamięci i liczbę operacji zmiennoprzecinkowych na sekundą w zależności od rozmiaru macierzy i wektora.



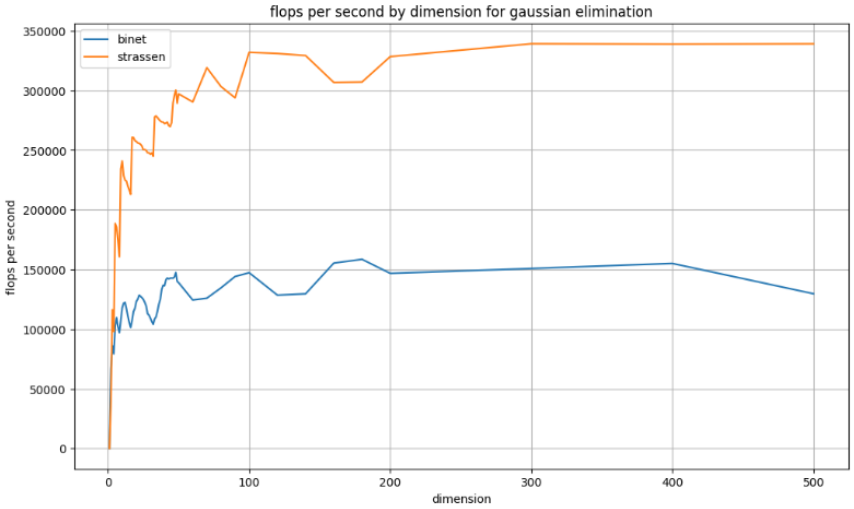
Rysunek 9: Czas wykonania algorytmu w zależności od rozmiaru macierzy i wektora



Rysunek 10: Liczba operacji zmiennoprzecinkowych algorytmu w zależności od rozmiaru macierzy i wektora



Rysunek 11: Zużycie pamięci algorytmu w zależności od rozmiaru macierzy i wektora



Rysunek 12: Liczba operacji zmiennoprzecinkowych na sekundę algorytmu w zależności od rozmiaru macierzy i wektora