

Lab 2. Odwracanie macierzy

1. Rekurencyjne odwracanie macierzy

1.1. Pseudokod

```
func rec_inv(M: Matrix(n*n)):
    if n == 1:
        if |M[0,0]| < epsilon: raise "Macierz nieodwracalna"
        return [1/M[0,0]]

    # M = [[A, B],
    #       [C, D]]
    A, B, C, D = podziel_macierz_na_4_bloki(M)

    A_inv = rec_inv(A)
    B_inv = rec_inv(B)
    C_inv = rec_inv(C)
    D_inv = rec_inv(D)

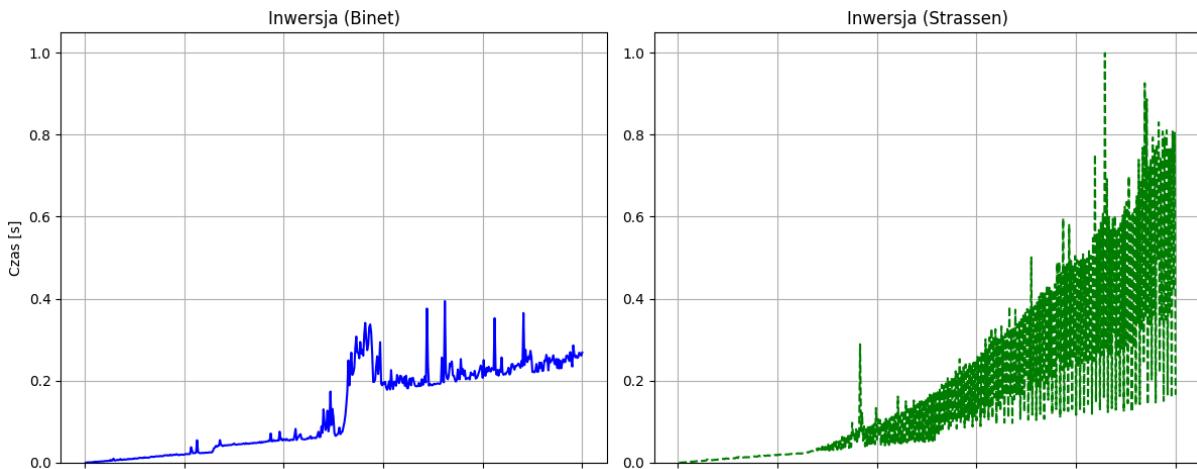
    S = D - C*A_inv*B

    S_inv = rec_inv(S)

    M1 = A_inv + A_inv*B*S_inv*C*A_inv
    M2 = -A_inv*B*S_inv
    M3 = -S_inv*C*A_inv
    M4 = S_inv

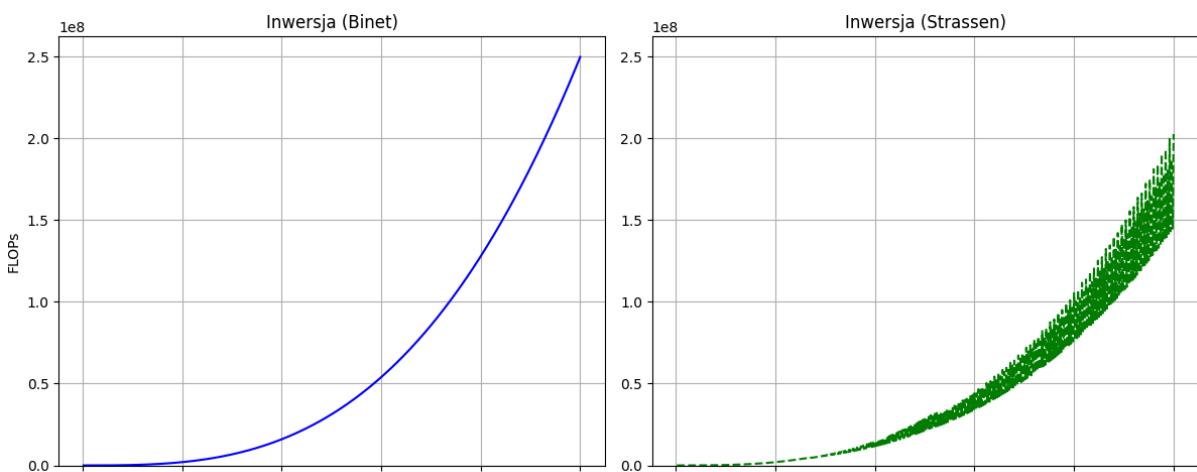
    return [[M1, M2],
            [M3, M4]]
```

1.2. Czas działania

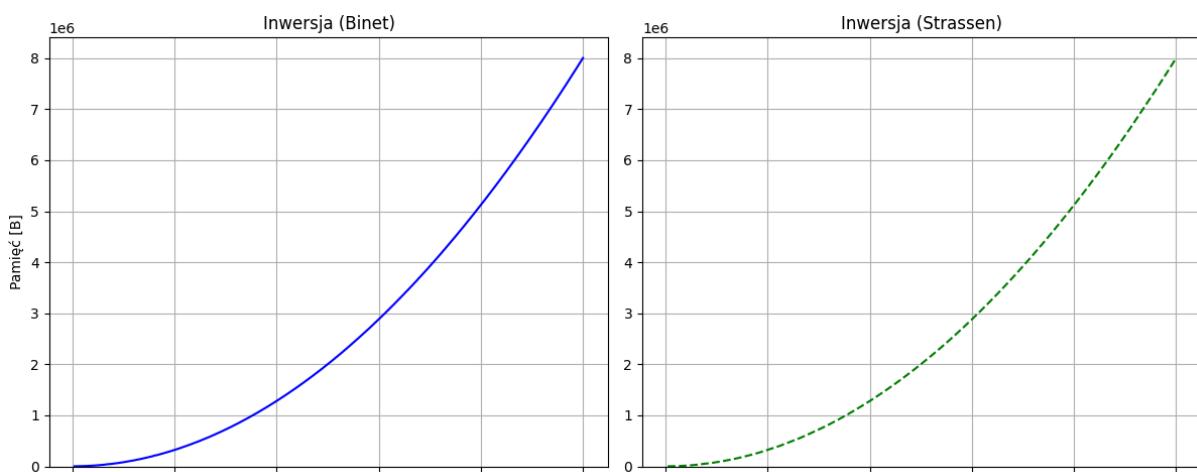


Rozmiar $n \in [500]$

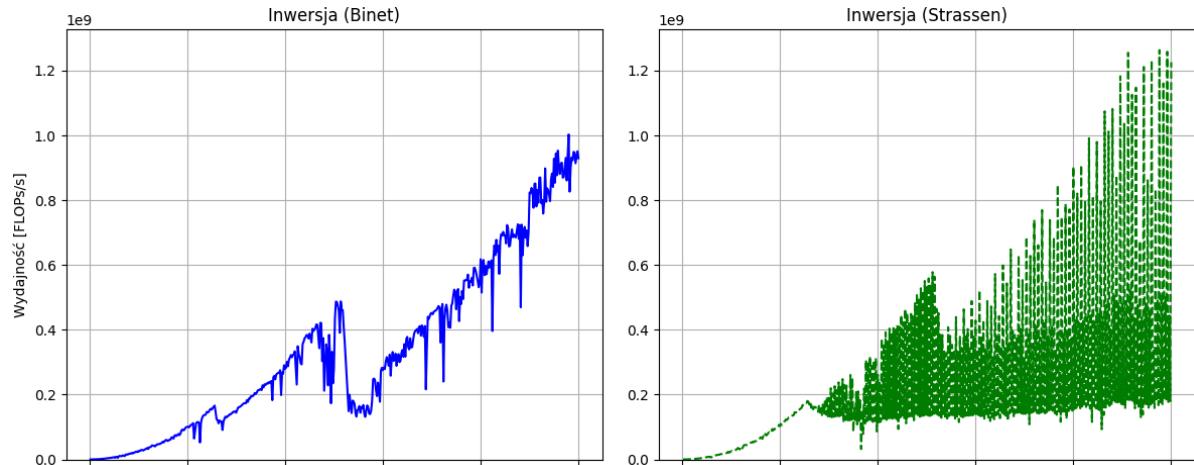
1.3. Liczba flopsów



1.4. Zużycie pamięci



1.5. Wydajność (flops/s)



2. Rekurencyjna faktoryzacja LU

2.1. Pseudokod

```
func solve_lower(L: Matrix(n*n), b: Vector(n*1), fun: Strassen/Binet):
    # Rozwiązuje układ  $Lx = b$ , gdzie  $L$  jest dolnotrójkątna
    # używając do mnożenia fun
    if n == 1:
        if |M[0,0]| < epsilon: raise "Macierz osobliwa"
        return b / M[0,0]

    #  $L_{12}$  byłaby macierzą zerową
    L_11, _, L_21, L_22 = podziel_macierz_na_4_bloki(M)

    b1, b2 = podziel_wektor_na_pol(b)

    x1 = solve_lower(L11_inv, b1)
    x2 = solve_lower(L22_inv, (b2 - L21 * x1))

    return [x1,
            x2]
```

```
func solve_upper(U: Matrix(n*n), b: Vector(n*1), fun: Strassen/Binet):
    # Rozwiązuje układ  $Ux = b$ , gdzie  $U$  jest górnottrójkątna
    # używając do mnożenia fun
    if n == 1:
        if |M[0,0]| < epsilon: raise "Macierz osobliwa"
        return b / M[0,0]

    #  $U_{21}$  byłaby macierzą zerową
    U_11, U_12, _, U_22 = podziel_macierz_na_4_bloki(M)

    b1, b2 = podziel_wektor_na_pol(b)

    x1 = solve_lower(U11_inv, (b1 - U12 * x2))
    x2 = solve_lower(U22_inv, b2)

    return [x1,
            x2]
```

```

func rec_lu(A: Matrix(n*n), fun: Strassen/Binet):
    # Oblicza faktoryzację LU i wyznacznik
    # przy użyciu rekurencji blokowej. Używa fun do mnożenia macierzy.

    if n == 1:
        if |M[0,0]| < epsilon: raise "Macierz osobliwa"
        L = [[1.0]]
        U = A
        det = A[0,0]
        return L, U, det

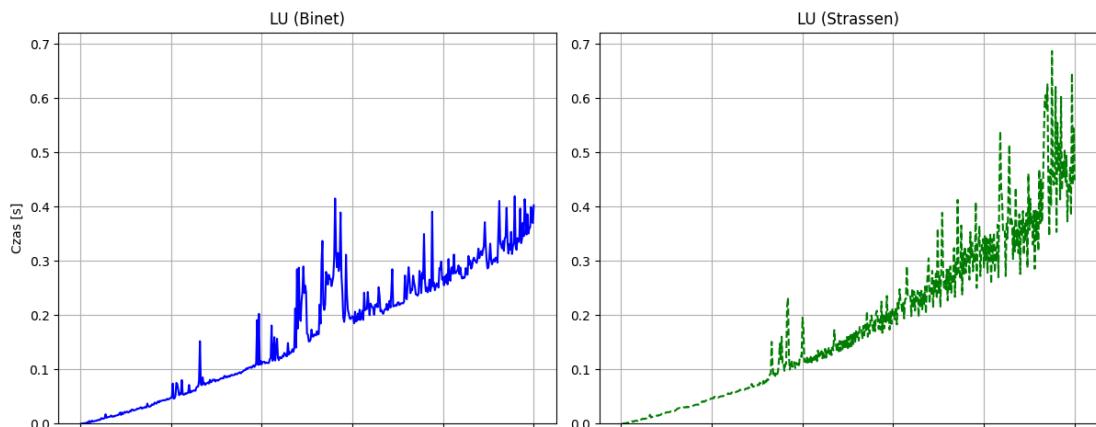
    A_11, A_12, A_21, A_22 = podziel_macierz_na_4_bloki(A)

    L_11, U_11, det_1 = rec_lu(A11)
    U_12 = solve_lower(L_11, A_12)
    L_21 = solve_upper(U_11.T, A_21.T).T # A.T - transpozycja A
    S = A_22 - L_21*U_21
    L22, U22, det_2 = rec_lu(S)
    det = det_1 * det_2

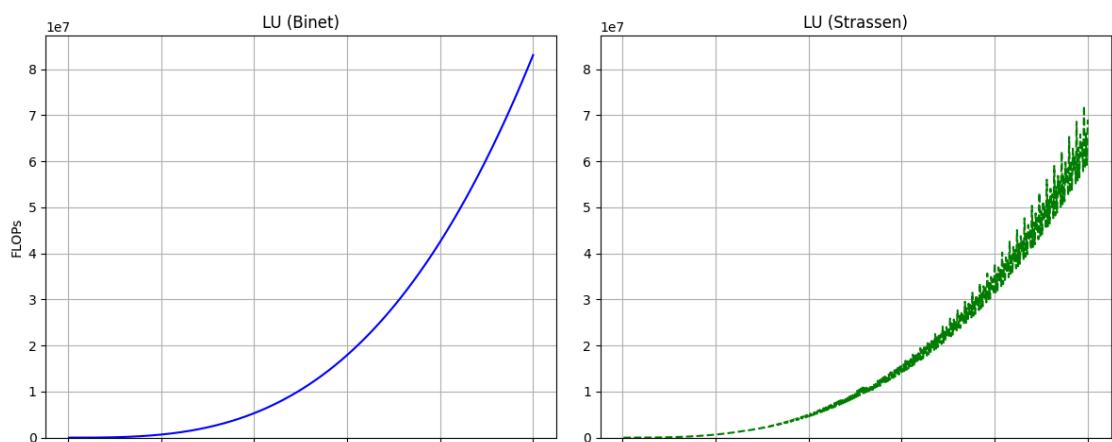
    return L := [[L_11, 0], [L_21, L_22]],
           U := [[U_11, U_12], [U_21, 0]],
           det

```

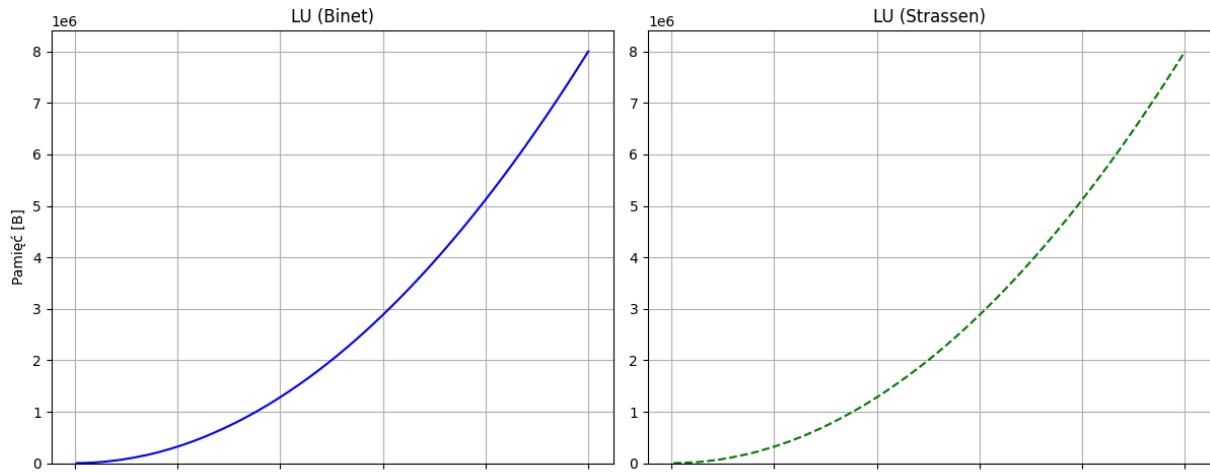
2.2. Czas działania



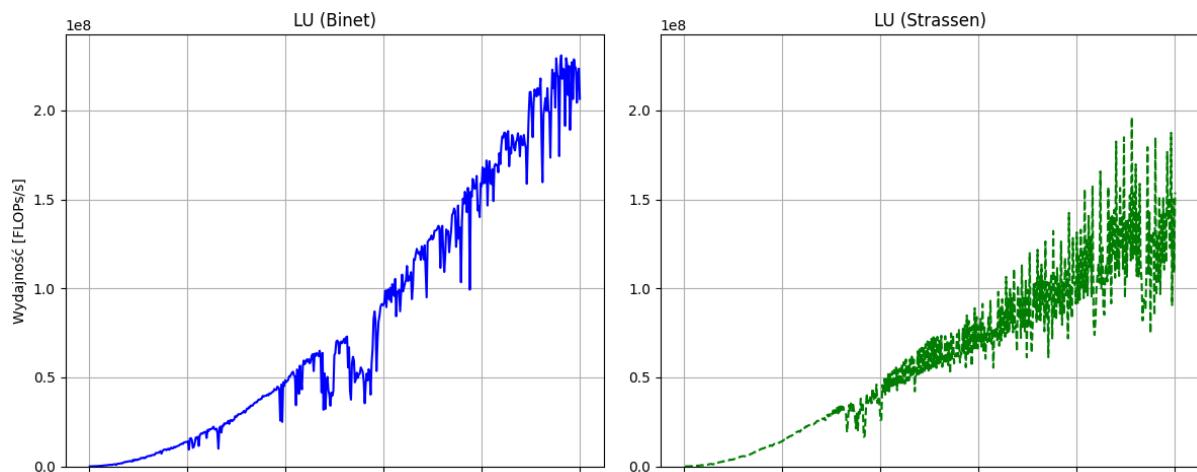
2.3. Liczba flopsów



2.4. Zużycie pamięci



2.5. Wydajność (flops/s)



3. Rekurencyjna eliminacja Gaussa

3.1. Pseudokod

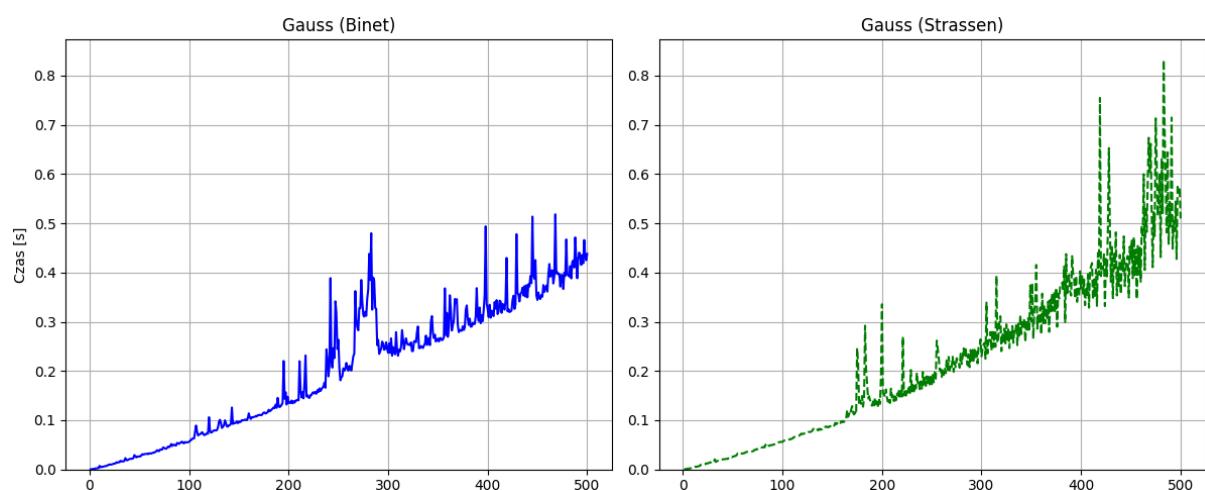
```
func gauss(A: Matrix(n*n), b: Vector(n*1), fun: Strassen/Binet):
    # Rozwiązuje układ  $A \cdot x = b$ 

    L, U = rec_lu(A)

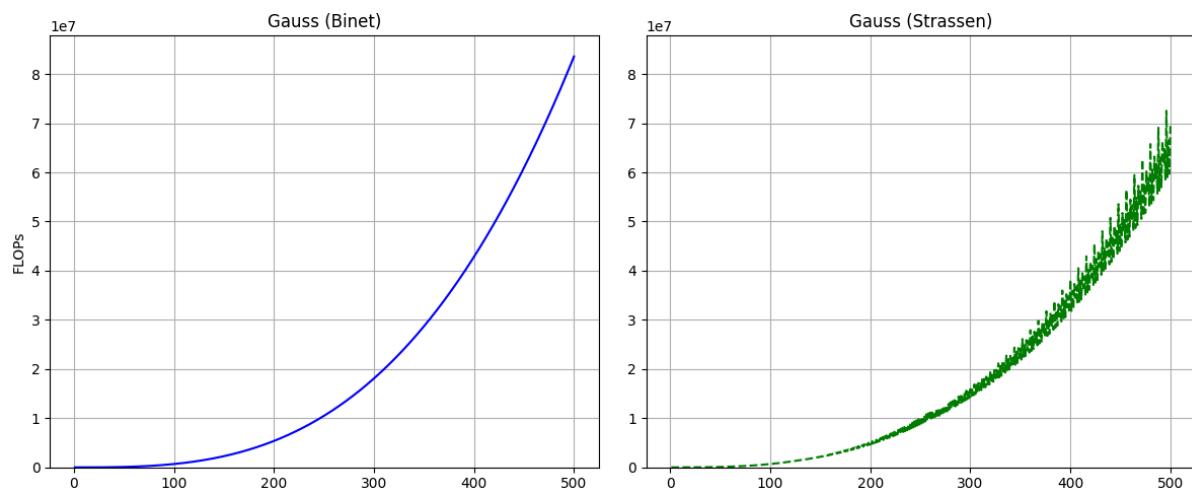
    c = solve_lower(L, b)
    x = solve_upper(U, c)

    return c
```

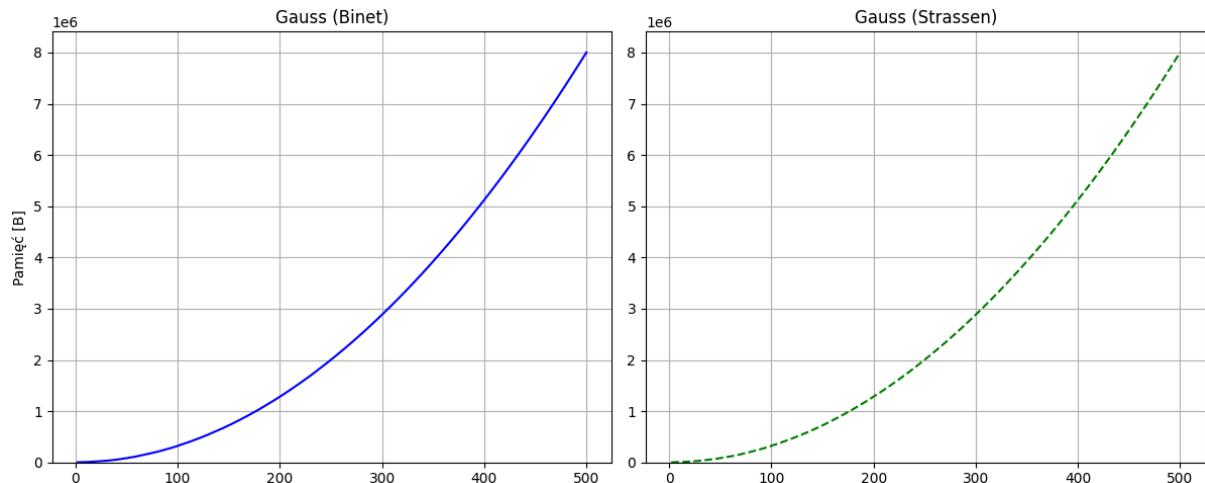
3.2. Czas działania



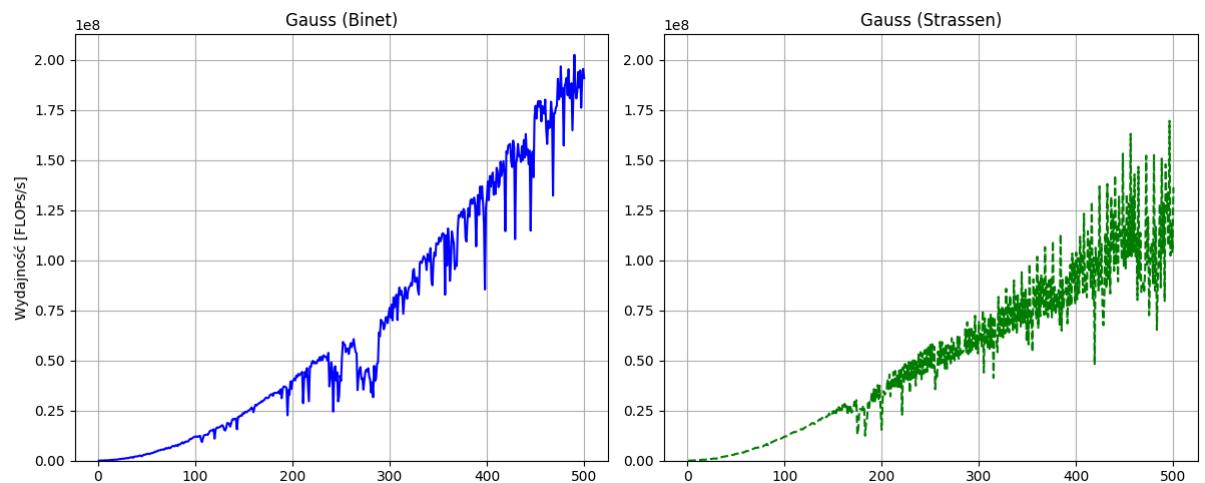
3.3. Liczba flopsów



3.4. Zużycie pamięci



3.5. Wydajność (flops/s)



4. Kod programu

4.1. Odwrocanie

```
def recursive_invert(M, multiply_func, count=None):
    if count is None:
        count = {'add': 0, 'mul': 0}

    n = M.shape[0]
    if M.shape[1] != n:
        raise ValueError("Macierz musi być kwadratowa.")

    if n == 1:
        val = M[0, 0]
        epsilon = 1e-8

        if np.abs(val) < epsilon:
            raise np.linalg.LinAlgError("Macierz numerycznie osobliwa.")

        count['mul'] += 1
        return np.array([[1.0 / val]]), count

    # Podział na bloki
    mid = n // 2
    A = M[:mid, :mid]
    B = M[:mid, mid:]
    C = M[mid:, :mid]
    D = M[mid:, mid:]

    # A_inv = A^-1
    A_inv, count = recursive_invert(A, multiply_func, count)

    # CA_inv = C * A^-1
    CA_inv, count = multiply_func(C, A_inv, count)

    # S = D - C * A^-1 * B
    S_part, count = multiply_func(CA_inv, B, count)
    S = np.subtract(D, S_part)
    count['add'] += S.size

    # S_inv = S^-1
    S_inv, count = recursive_invert(S, multiply_func, count)

    # H = S^-1
    H = S_inv

    # G = -S^-1 * (C * A^-1)
    G_part, count = multiply_func(S_inv, CA_inv, count)
    G = np.negative(G_part)
    count['mul'] += G.size

    # F = -(A^-1 * B) * S^-1
    A_inv_B, count = multiply_func(A_inv, B, count)
    F_part, count = multiply_func(A_inv_B, S_inv, count)
    F = np.negative(F_part)
```

```

count['mul'] += F.size

# E = A^-1 - (A^-1 * B * S^-1) * (C * A^-1)
# E = A_inv - F_part * CA_inv
E_part, count = multiply_func(F_part, CA_inv, count)
E = np.subtract(A_inv, E_part)
count['add'] += E.size

res = np.empty_like(M)
res[:mid, :mid] = E
res[:mid, mid:] = F
res[mid:, :mid] = G
res[mid:, mid:] = H

return res, count

```

4.2. Gauss i LU

```

def solve_lower_recursive(L, b, multiply_func, count=None):
    if count is None:
        count = {'add': 0, 'mul': 0}

    n = L.shape[0]
    if n == 1:
        val = L[0, 0]
        if np.abs(val) < 1e-8:
            raise np.linalg.LinAlgError("Macierz dolnotrójkątna osobliwa.")
        count['mul'] += 1
        return b / val, count

    mid = n // 2
    L11 = L[:mid, :mid]
    L21 = L[mid:, :mid]
    L22 = L[mid:, mid:]

    b1 = b[:mid]
    b2 = b[mid:]

    # x1 = L11_inv * b1
    x1, count = solve_lower_recursive(L11, b1, multiply_func, count)

    # x2 = L22_inv * (b2 - L21 * x1)
    b2_mod_part, count = multiply_func(L21, x1, count)
    b2_mod = np.subtract(b2, b2_mod_part)
    count['add'] += b2_mod.size

    x2, count = solve_lower_recursive(L22, b2_mod, multiply_func, count)

    x = np.vstack((x1, x2))
    return x, count

def solve_upper_recursive(U, b, multiply_func, count=None):
    if count is None:
        count = {'add': 0, 'mul': 0}

    n = U.shape[0]

```

```

if n == 1:
    val = U[0, 0]
    if np.abs(val) < 1e-8:
        raise np.linalg.LinAlgError("Macierz górnoprójkątna osobiwa.")
    count['mul'] += 1
    return b / val, count

mid = n // 2
U11 = U[:mid, :mid]
U12 = U[:mid, mid:]
U22 = U[mid:, mid:]

b1 = b[:mid]
b2 = b[mid:]

# x2 = U22_inv * b2
x2, count = solve_upper_recursive(U22, b2, multiply_func, count)

# x1 = U11_inv * (b1 - U12 * x2)
b1_mod_part, count = multiply_func(U12, x2, count)
b1_mod = np.subtract(b1, b1_mod_part)
count['add'] += b1_mod.size

x1, count = solve_upper_recursive(U11, b1_mod, multiply_func, count)

x = np.vstack((x1, x2))
return x, count


def recursive_lu(A, multiply_func, count=None):
    if count is None:
        count = {'add': 0, 'mul': 0}

    n = A.shape[0]
    if A.shape[1] != n:
        raise ValueError("Macierz musi być kwadratowa.")

    if n == 1:
        val = A[0, 0]
        if np.abs(val) < 1e-8:
            raise np.linalg.LinAlgError("Element na przekątnej zbyt bliski zera.")
        L = np.array([[1.0]])
        U = A
        det = val
        return L, U, det, count

    mid = n // 2
    A11 = A[:mid, :mid]
    A12 = A[:mid, mid:]
    A21 = A[mid:, :mid]
    A22 = A[mid:, mid:]

    # L11, U11, det1 = lu(A11)
    L11, U11, det1, count = recursive_lu(A11, multiply_func, count)

    # U12 = L11_inv * A12

```

```

U12, count = solve_lower_recursive(L11, A12, multiply_func, count)

# L21 = A21 * U11_inv
L21, count = solve_upper_recursive(U11.T, A21.T, multiply_func, count)
L21 = L21.T

# S = A22 - L21 * U12 (Dopełnienie Schura)
S_part, count = multiply_func(L21, U12, count)
S = np.subtract(A22, S_part)
count['add'] += S.size

# L22, U22, det2 = lu(S)
L22, U22, det2, count = recursive_lu(S, multiply_func, count)

det = det1 * det2
count['mul'] += 1

L = np.zeros_like(A)
U = np.zeros_like(A)

L[:mid, :mid] = L11
L[mid:, :mid] = L21
L[mid:, mid:] = L22

U[:mid, :mid] = U11
U[:mid, mid:] = U12
U[mid:, mid:] = U22

return L, U, det, count

def recursive_gauss_solve(A, b, multiply_func, count=None):
    if count is None:
        count = {'add': 0, 'mul': 0}

    # Faktoryzacja A = LU
    L, U, det, count = recursive_lu(A, multiply_func, count)

    # Rozwiąż L*c = b (Forward substitution)
    c, count = solve_lower_recursive(L, b, multiply_func, count)

    # Rozwiąż U*x = c (Backward substitution)
    x, count = solve_upper_recursive(U, c, multiply_func, count)

    return x, count

```