Raport 2

Rekurencyjne wyznaczanie odwrotności, LU i wyznacznika macierzy

Michał Kobiera, Maciej Pięta

1. Opis ćwiczenia

Celem ćwiczenia było zaimplementowanie trzech algorytmów:

- Rekurencyjnego odwracania macierzy
- Rekurencyjnej faktoryzacji LU macierzy
- Rekurencyjnego obliczania wyznacznika

Oraz następnie zmierzenie czasu działania i ilości operacji zmiennoprzecinkowych dla macierzy o rozmiarach $2^n x\ 2^n$ wypełnionych liczbami z przedziału (0,1) i oszacowanie złożoności obliczeniowej zaimplementowanych algorytmów

2. Opis algorytmów

1. Rekurencyjne odwracanie macierzy

Inverse(A)

Jeśli macierz A ma tylko 1 element:

Zwróć 1/A

W przeciwnym wypadku:

Podziel A na 4 macierze A11, A12, A21, A22:

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

Oblicz rekurencyjnie $A11^{-1} = inverse(A11)$

Oblicz $S22 = A22 - A21 * A11^{-1} * A12$

Oblicz rekurencyjnie $S22^{-1} = inverse(S22)$

Oblicz:

$$B11 = A11^{-1} * (I + A12 * S22^{-1} * A21 * A11^{-1})$$

$$B12 = -A11^{-1} * A12 * S22^{-1}$$

$$B21 = -S22^{-1} * A21 * A11^{-1}$$

$$B22 = S22^{-1}$$

Zwróć

$$A^{-1} = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

2. Rekurencyjna faktoryzacja LU

LU(A):

Jeśli macierz A ma tylko 1 element:

$L = 1$

$U = A$

Zwróć L, U

W przeciwnym wypadku:

Podziel A na 4 macierze A11, A12, A21, A22:

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

Oblicz rekurencyjnie $L11, U11 = LU(A11)$

Oblicz:

$U11^{-1}, L11^{-1}$

$L21 = A21 * U11^{-1}$

$U12 = L11^{-1} * A12$

$S = A22 - A21 * U11^{-1} * L11^{-1} * A12$

$LS, US = LU(S)$

$$L = \begin{bmatrix} L11 & \mathbf{0} \\ L21 & LS \end{bmatrix} \qquad S = \begin{bmatrix} U11 & U12 \\ \mathbf{0} & US \end{bmatrix}$$

Zwróć L, S

3. Rekurencyjne obliczanie wyznacznika

Det(A):

Oblicz $L, U = LU(A)$

$\det = 1$

$dla\ i = 1 \ldots len(A)$:

$det = det * L[i][i] * U[i][i]$

Zwróć det

3. Implementacje algorytmów

Rekurencyjne odwracanie macierzy

```python
def inverse(A):
    if A.shape[0] != A.shape[1]:
        print("ERROR: Wrong matrix size!")
        return None

    if len(A) == 2:
        return np.array(
            [[A[1, 1], -A[0, 1]],
             [-A[1, 0], A[0, 0]]], dtype = Number) / (A[0, 0] * A[1, 1] - A[0, 1] * A[1, 0])

    matrix_size = len(A)
    A11 = A[:matrix_size // 2, :matrix_size // 2]
    A12 = A[:matrix_size // 2, matrix_size // 2:]
    A21 = A[matrix_size // 2:, :matrix_size // 2]
    A22 = A[matrix_size // 2:, matrix_size // 2:]



    A11_inv = inverse(A11)
    S22 = A22 - strassen_mlt(A21, strassen_mlt(A11_inv, A12))
    S22_inv = inverse(S22)

    B11_2 = np.identity(len(A11), dtype = Number) + strassen_mlt(strassen_mlt(A12, S22_inv), strassen_mlt(A21, A11_inv))
    B11 = strassen_mlt(A11_inv, B11_2)

    B12 = strassen_mlt(-A11_inv, strassen_mlt(A12, S22_inv))
    B21 = strassen_mlt(-S22_inv, strassen_mlt(A21, A11_inv))
    B22 = S22_inv

    res = np.empty(A.shape, dtype = Number)
    res[:matrix_size // 2, :matrix_size // 2] = B11
    res[:matrix_size // 2, matrix_size // 2:] = B12
    res[matrix_size // 2:, :matrix_size // 2] = B21
    res[matrix_size // 2:, matrix_size // 2:] = B22
    return res
```

Rekurencyjna LU faktoryzacja ¶

```
def LU_factorization(A):
    if A.shape[0] != A.shape[1]:
        print("ERROR: Wrong matrix size!")
        return None

    if len(A) == 1:
        L = np.array([[Number(1.0)]], dtype=Number)
        U = A.copy()
        return L, U

    if len(A) == 2:
        a00 = A[0, 0]
        a01 = A[0, 1]
        a10 = A[1, 0]
        a11 = A[1, 1]

        L = np.array([[1.0, 0.0],
                      [a10 / a00, 1.0]], dtype=Number)

        U = np.array([[a00, a01],
                      [0.0, a11 - (a10 / a00) * a01]], dtype=Number)
        return L, U

    matrix_size = len(A)
    A11 = A[:matrix_size // 2, :matrix_size // 2]
    A12 = A[:matrix_size // 2, matrix_size // 2:]
    A21 = A[matrix_size // 2:, :matrix_size // 2]
    A22 = A[matrix_size // 2:, matrix_size // 2:]
    L11, U11 = LU_factorization(A11)
    U11_inv = inverse(U11)
    L21 = strassen_mlt(A21, U11_inv)
    L11_inv = inverse(L11)
    U12 = strassen_mlt(L11_inv, A12)
    S = A22 - strassen_mlt(strassen_mlt(A21, U11_inv), strassen_mlt(L11_inv, A12))

    LS, US = LU_factorization(S)

    L = np.zeros(A.shape, dtype = Number)
    U = np.zeros(A.shape, dtype = Number)

    L[:matrix_size // 2, :matrix_size // 2] = L11
    L[matrix_size // 2:, :matrix_size // 2] = L21
    L[matrix_size // 2:, matrix_size // 2:] = LS

    U[:matrix_size // 2, :matrix_size // 2] = U11
    U[:matrix_size // 2, matrix_size // 2:] = U12
    U[matrix_size // 2:, matrix_size // 2:] = US


    return L, U
```

Rekurencyjne obliczanie wyznacznika

```
def det(A):
    L, U = LU_factorization(A)
    result = np.prod(np.diagonal(L)*np.diagonal(U))
    return result
```

Do mnożenia macierzy wykorzystaliśmy zaimplementowaną przez nas funkcję *strassen_mlt(A, B):*

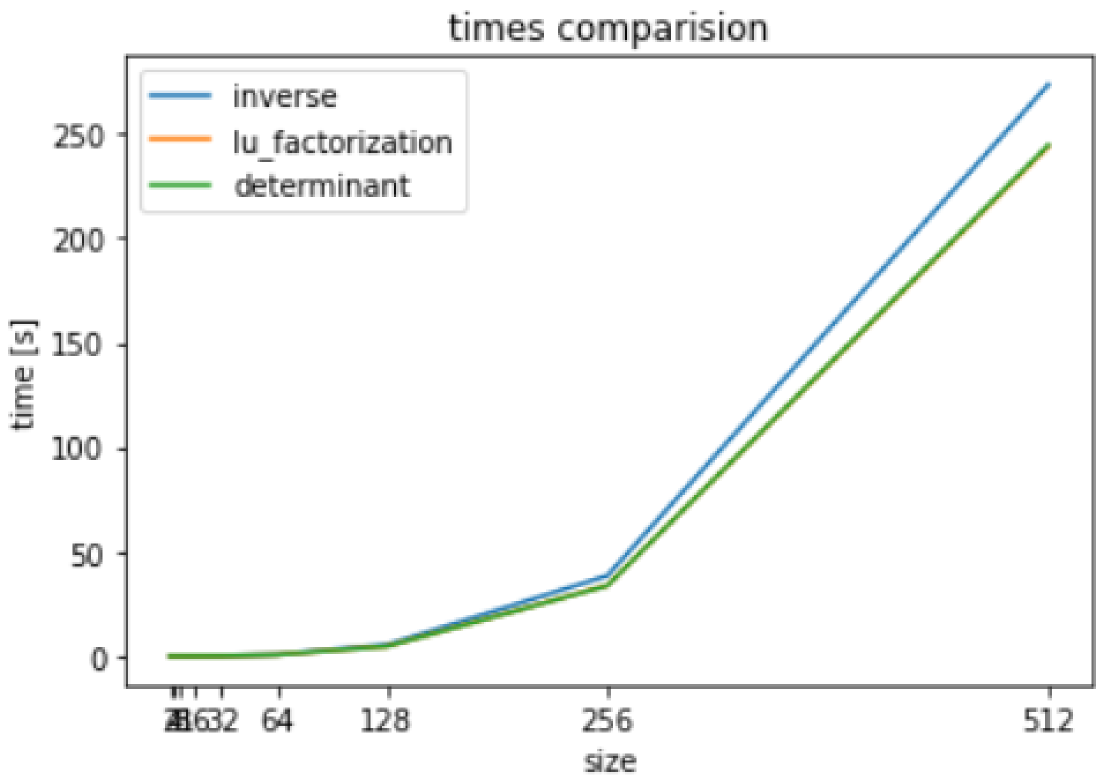
```
def strassen_mlt(A,B):
    n = len(A)
    if(n==2):
        a00 = A[0,0]*B[0,0] + A[0,1]*B[1,0]
        a01 = A[0,0]*B[0,1] + A[0,1]*B[1,1]

        a10 = A[1,0]*B[0,0] + A[1,1]*B[1,0]
        a11 = A[1,0]*B[0,1] + A[1,1]*B[1,1]

        return np.array([[a00,a01],
                         [a10,a11]], dtype = Number)

    A11 = A[:n//2, :n//2]
    B11 = B[:n//2, :n//2]

    A12 = A[:n//2, (n//2):]
    B12 = B[:n//2, (n//2):]

    A21 = A[(n//2): , :n//2]
    B21 = B[(n//2): , :n//2]

    A22 = A[(n//2): , (n//2):]
    B22 = B[(n//2): , (n//2):]

    P1 = strassen_mlt((A11 + A22), (B11 + B22))
    P2 = strassen_mlt((A21 + A22), B11)
    P3 = strassen_mlt(A11, (B12 - B22))
    P4 = strassen_mlt(A22, (B21 - B11))
    P5 = strassen_mlt((A11 + A12), B22)
    P6 = strassen_mlt((A21 - A11), (B11 + B12))
    P7 = strassen_mlt((A12 - A22), (B21 + B22))

    C11 = (P1 + P4 - P5 + P7)
    C12 = (P3 + P5)
    C21 = (P2 + P4)
    C22 = (P1 - P2 + P3 + P6)

    result = np.empty((n, n), dtype = Number)
    result[:n // 2, :n // 2] = C11
    result[:n // 2, n // 2:] = C12
    result[n // 2:, :n // 2] = C21
    result[n // 2:, n // 2:] = C22

    return result
```

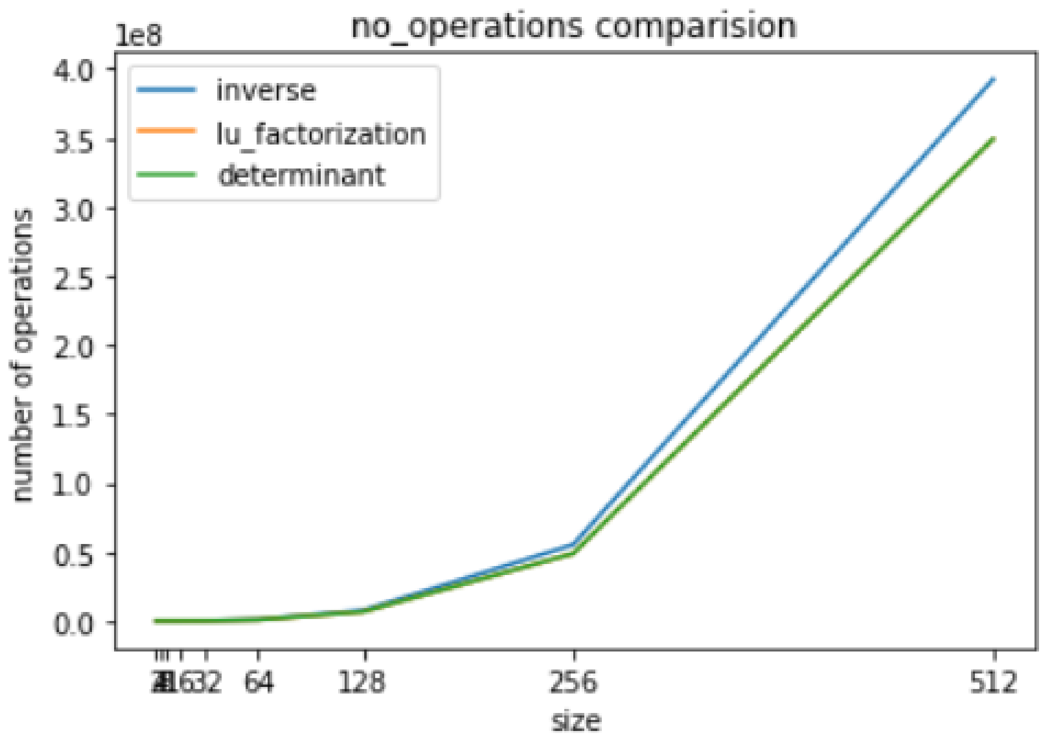
4. Wykresy czasu działania i ilości operacji zmiennoprzecinkowych

W celu zliczania ilości operacji zmiennoprzecinkowych zaimplementowaliśmy własny typ Number, który dziedziczy po domyślnym typie float i nadpisuje operatory +, -, *, / by po każdym wykonaniu danej operacji zwiększać zmienną statyczną *operation_counter* :

```python
class Number(float):
    operation_counter = 0

    def __repr__(self) -> str:
        return f"{self:.8f}"

    def __radd__(self, other):
        Number.operation_counter += 1
        return Number(super().__radd__(other))

    def __add__(self, other):
        Number.operation_counter += 1
        return Number(super().__add__(other))

    def __rsub__(self, other):
        Number.operation_counter += 1
        return Number(super().__rsub__(other))

    def __sub__(self, other):
        Number.operation_counter += 1
        return Number(super().__sub__(other))

    def __mul__ (self, other):
        Number.operation_counter += 1
        return Number(super().__mul__(other))

    def __rmul__ (self, other):
        Number.operation_counter += 1
        return Number(super().__rmul__(other))

    def __truediv__(self, other):
        Number.operation_counter += 1
        return Number(super().__truediv__(other))

    def __rtruediv__(self, other):
        Number.operation_counter += 1
        return Number(super().__rtruediv__(other))

    def counter_reset():
        Number.operation_counter = 0
```

Porównanie czasów wykonania poszczególnych algorytmów:





Dane z wykresów:

-----------------------------------Times-----------------------------------

	2	4	8	16	32	64	128	256	512
Inverse	0.0	0.0	0.002001	0.012003	0.091403	0.724075	5.391992	38.501112	273.52911
LU	0.0	0.0	0.001	0.009002	0.078017	0.642651	4.683525	33.829473	243.781058
det	0.0	0.0	0.0	0.009002	0.078019	0.602293	4.61769	33.686094	244.55308

-------------------------Floating point operations-------------------------

	2	4	8	16	32	64	128	256	512
Inverse	5	138	1828	17144	139200	1054768	7712048	55313424	392539120
LU	4	79	1199	12847	112787	893767	6705035	48805999	349304051
det	8	87	1215	12879	112851	893895	6705291	48806511	349305075

5. Szacowanie złożoności

Do oszacowania złożoności wykorzystaliśmy dane z wykresów oraz funkcję *curve_fit* z biblioteki *scipy* która aproksymuje parametry funkcji korzystając z metody najmniejszych kwadratów.

Jako model przyjęliśmy funkcję typu $f(x) = ax^b$ i otrzymaliśmy następujące wyniki:

1. Odwrotność macierzy:

   Bazując na otrzymanej ilości operacji zmiennoprzecinkowych otrzymaliśmy:

   $$f(x) \sim 8.564 * x^{2.828}$$

   Na podstawie czasu:

   $$f(x) \sim 5.919 * 10^{-6} * x^{2.829}$$

2. Faktoryzacja LU:

   Na podstawie ilości operacji zmiennoprzecinkowych:

   $$f(x) \sim 7.046 * x^{2.840}$$

   Na podstawie czasu:

   $$f(x) \sim 4.648 * 10^{-6} * x^{2.849}$$

3. Wyznacznik macierzy:

Na podstawie ilości operacji zmiennoprzecinkowych:

$$f(x) \sim 7.047 * x^{2.840}$$

Na podstawie czasu:

$$f(x) \sim 4.648 * 10^{-6} * x^{2.849}$$

```
from scipy.optimize import curve_fit
```

```
def func(x, a, b):
    return a * np.power(x,b)
```

```
x_data = sizes
y_det_op = res_operations[:, 2]
y_lu_op = res_operations[:, 1]
y_inv_op = res_operations[:, 0]

y_det_time = res_times[:, 2]
y_lu_time = res_times[:, 1]
y_inv_time = res_times[:, 0]
```

Inverse ¶

```
params_inv_op, cov_inv_op = curve_fit(func, x_data, y_inv_op)
print(params_inv_op, "\n",  cov_inv_op)
```

```
[3.07741239 3.01450861]
 [[ 1.80399444e-04 -1.21046245e-05]
 [-1.21046245e-05  8.12471307e-07]]
```

```
params_inv_time, cov_inv_time = curve_fit(func, x_data, y_inv_time)
print(params_inv_time, "\n",  cov_inv_time)
```

```
[1.31023574e-06 3.01349882e+00]
 [[ 1.44793968e-16 -2.28193894e-11]
 [-2.28193894e-11  3.59747975e-06]]
```

LU

```python
params_lu_op, cov_lu_op = curve_fit(func, x_data, y_lu_op)
print(params_lu_op, "\n",  cov_lu_op)
```

```
[2.28086866 3.02972878]
 [[ 4.80811955e-04 -4.35270603e-05]
 [-4.35270603e-05  3.94167638e-06]]
```

```python
params_lu_time, cov_lu_time = curve_fit(func, x_data, y_lu_time)
print(params_lu_time, "\n",  cov_lu_time)
```

```
[1.81601510e-06 2.89025392e+00]
 [[ 6.31604280e-15 -7.18430949e-10]
 [-7.18430949e-10  8.17510111e-05]]
```

det

```python
params_det_op, cov_det_op = curve_fit(func, x_data, y_det_op)
print(params_det_op, "\n", cov_det_op)
```

```
[2.2834843  3.02950209]
 [[ 4.61316591e-04 -4.17143681e-05]
 [-4.17143681e-05  3.77320028e-06]]
```

```python
params_det_time, cov_det_time = curve_fit(func, x_data, y_det_time)
print(params_det_time, "\n",  cov_det_time)
```

```
[4.47765654e-06 2.70898018e+00]
 [[ 4.60195563e-13 -2.12440762e-08]
 [-2.12440762e-08  9.81187649e-04]]
```

6. Sprawdzenie poprawności zaimplementowanych algorytmów

Do sprawdzenia poprawności implementacji funkcji *inverse(A)* oraz *det(A)* wykorzystaliśmy istniejące funkcję biblioteki *numpy: numpy.linalg.inv(A), numpy.linalg.det(A)*

Poprawność implementacji LU_factorization(A) była sprawdzana poprzez sprawdzenie bliskości wyniku mnożenia obliczonych macierzy *L, U* z wejściową macierzą *A*

Odwrotność:

```
Number.counter_reset()
res = inverse(A_test)

print(f"Matrix size: {matrix_size}")
print("Number of operations:", Number.operation_counter)
print("Is correct?:", np.allclose(transform_to_float(res), np.linalg.inv(transform_to_float(A_test))))
```

```
Matrix size: (16, 16)
Number of operations: 17144
Is correct?: True
```

Faktoryzacja LU:

```
Number.counter_reset()
L, U = LU_factorization(A_test)

print(f"Matrix size: {matrix_size}")
print("Number of operations:", Number.operation_counter)
print("Is correct?:", np.allclose(transform_to_float(A_test), transform_to_float(L @ U)))
```

```
Matrix size: (16, 16)
Number of operations: 12847
Is correct?: True
```

Wyznacznik:

```
Number.counter_reset()
A_det = det(A_test)

print(f"Matrix size: {matrix_size}")
print("Number of operations:", Number.operation_counter)
print("Is correct?:", abs(A_det - np.linalg.det(transform_to_float(A_test))) < 1e-8)
```

```
Matrix size: (16, 16)
Number of operations: 12878
Is correct?: True
```