Algorytmy Macierzowe

Laboratorium nr. 3

Artur Gęsiarz, Błażej Kapkowski

Cel zadania

Celem zadania było zaprojektowanie algorytmu kompresji obrazu z wykorzystaniem dekompozycji SVD (Singular Value Decomposition) oraz wizualizacja efektów kompresji dla różnych parametrów. Kompresję wykonano na poszczególnych kanałach RGB obrazu, a efekty działania algorytmu zostały przedstawione w formie rekonstrukcji obrazu oraz analizy wartości osobliwych.

Opis rozwiązania

1. Deklaracja funkcji dekompozycji SVD

Jednym z kluczowych elementów implementacji była funkcja **svd_decomposition**, która realizuje dekompozycję macierzy A na trzy składowe*:*

$$A \approx USV^T$$

gdzie:

- U — macierz ortogonalna zawierająca wektory własne AA^T,
- S — wartości osobliwe, uporządkowane malejąco,
- V — macierz ortogonalna zawierająca wektory własne A^TA.

Dla optymalizacji obliczeń zastosowano metodę iteracyjną **power_iteration** do znajdowania wektorów własnych i odpowiadających im wartości własnych macierzy A^TA.

```
def power_iteration(A, num_simulations=100):
    b = np.random.rand(A.shape[1])
```

```
    for _ in range(num_simulations):
        b_next = np.dot(A, b)
        b = b_next / np.linalg.norm(b_next)

    eigenvalue = np.dot(b, np.dot(A, b)) / np.dot(b, b)

    return b, eigenvalue


def svd_decomposition(A, k, epsilon=1e-10):
    m, n = A.shape
    U = np.zeros((m, k))
    S = []
    V = np.zeros((k, n))

    B = np.dot(A.T, A)
    num_components = 0

    for _ in range(k):
        v, sigma_squared = power_iteration(B)

        if sigma_squared < epsilon ** 2:
            break

        sigma = np.sqrt(sigma_squared)
        S.append(sigma)

        V[num_components, :] = v

        u = np.dot(A, v) / sigma
        U[:, num_components] = u

        B -= sigma_squared * np.outer(v, v)

        num_components += 1

        if np.allclose(B, 0, atol=epsilon):
            break

    S = np.array(S)
    U = U[:, :num_components]
    V = V[:num_components, :]

    return U, S, V
```

2. Kompresja macierzy

Funkcja **svd_compress** została zaimplementowana w celu kompresji macierzy do zadanego rzędu (max_rank). Kompresja polega na zachowaniu tylko tych wartości osobliwych S, które są większe od zadanego progu ϵ.

```
def svd_compress(A, max_rank, epsilon=1e-10):
    U, S, Vt = svd_decomposition(A, max_rank)

    rank = min(max_rank, np.sum(S > epsilon))
    U_reduced = U[:, :rank]
    S_reduced = S[:rank]
    V_reduced = Vt[:rank, :]

    return Node(rank=rank, size=A.shape, singular_values=S_reduced,
U=U_reduced, V=V_reduced)
```

3. Struktura danych do przechowywania wyników kompresji

Aby umożliwić reprezentację skompresowanych danych oraz hierarchiczną kompresję obrazu, zaimplementowaliśmy klasę Node. Każdy węzeł drzewa kwadrantów przechowuje stopień kompresji (rank), rozmiar macierzy oraz ewentualne składowe macierzy po dekompozycji SVD.

```
class Node:
    def __init__(self, rank, size, singular_values=None, U=None, V=None):
        self.rank = rank
        self.size = size
        self.singular_values = singular_values
        self.U = U
        self.V = V
        self.children = []

    def draw_compression(self, matrix, x_bounds=(0, None), y_bounds=(0,
None)):
        if x_bounds[1] is None: x_bounds = (x_bounds[0], self.size[0])
        if y_bounds[1] is None: y_bounds = (y_bounds[0], self.size[1])

        if not self.children:
            x_start, x_end = x_bounds
            y_start, y_end = y_bounds
            matrix[x_start:x_end, y_start:y_start + self.rank] = 0
            matrix[x_start:x_start + self.rank, y_start:y_end] = 0
            return

        x_mid = (x_bounds[0] + x_bounds[1]) // 2
        y_mid = (y_bounds[0] + y_bounds[1]) // 2
```

```
        self.children[0].draw_compression(matrix, (x_bounds[0], x_mid),
(y_bounds[0], y_mid))
        self.children[1].draw_compression(matrix, (x_bounds[0], x_mid),
(y_mid, y_bounds[1]))
        self.children[2].draw_compression(matrix, (x_mid, x_bounds[1]),
(y_bounds[0], y_mid))
        self.children[3].draw_compression(matrix, (x_mid, x_bounds[1]),
(y_mid, y_bounds[1]))
```

4. Kompresja obrazu w postaci drzewa kwadrantów

Aby umożliwić hierarchiczną kompresję, zaimplementowaliśmy **drzewo kwadrantów**, w którym obraz dzielony jest na mniejsze podmacierze, jeśli kompresja globalna nie spełnia zadanych kryteriów błędu.

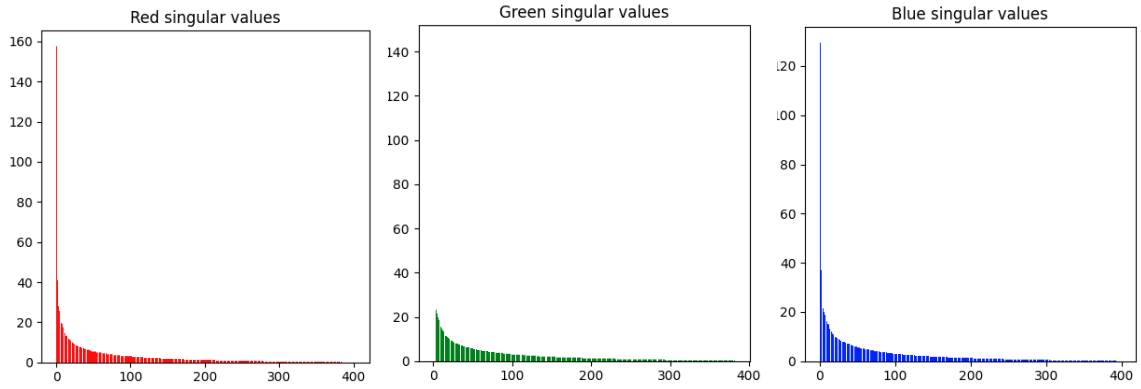
Funkcja **compress_matrix_tree** tworzy takie drzewo rekursywnie:

```
def compress_matrix_tree(A, max_rank, epsilon=1e-10, min_size=2):
    if A.shape[0] <= min_size or A.shape[1] <= min_size:
        return svd_compress(A, max_rank, epsilon)

    compressed = svd_compress(A, max_rank, epsilon)

    if compute_compression_error(A, compressed) <= epsilon:
        return compressed

    root = Node(rank=0, size=A.shape)
    mid_row, mid_col = A.shape[0] // 2, A.shape[1] // 2

    submatrices = [
        A[:mid_row, :mid_col],
        A[:mid_row, mid_col:],
        A[mid_row:, :mid_col],
        A[mid_row:, mid_col:]
    ]
    for submatrix in submatrices:
        if submatrix.size > 0:
            root.children.append(compress_matrix_tree(submatrix, max_rank,
epsilon, min_size))

    return root
```

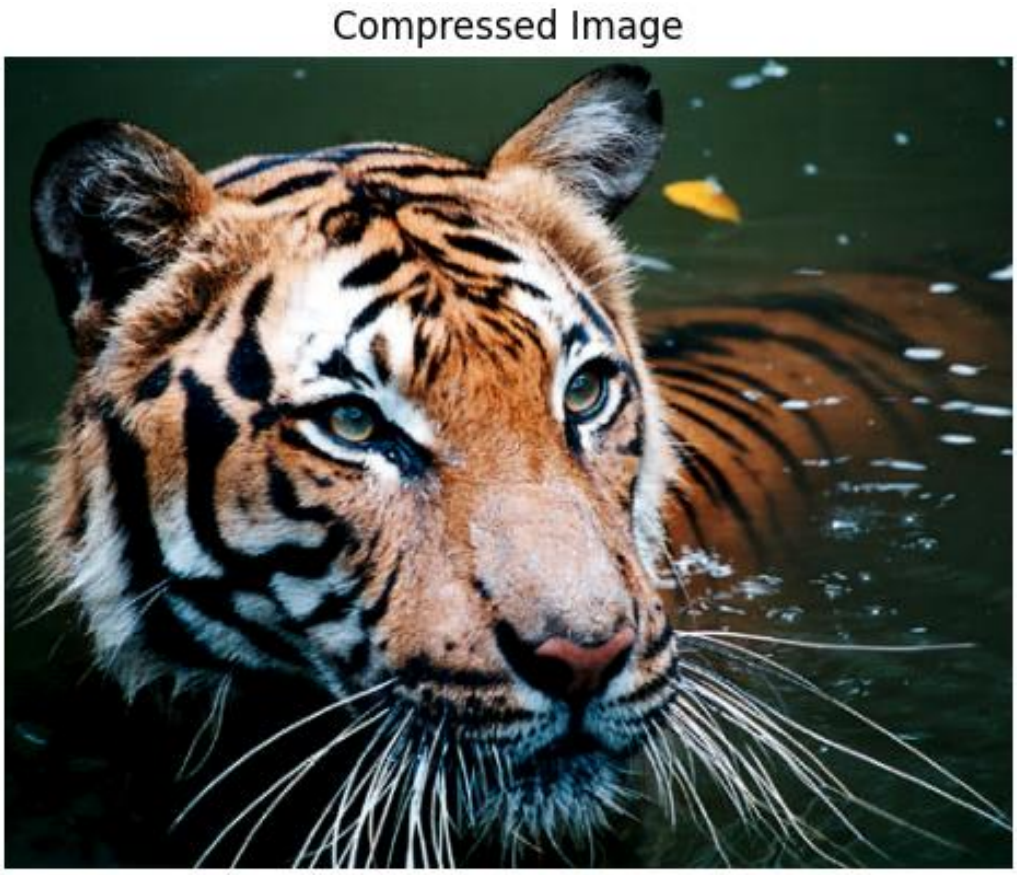
4. Analiza wartości osobliwych dla wybranego obrazu

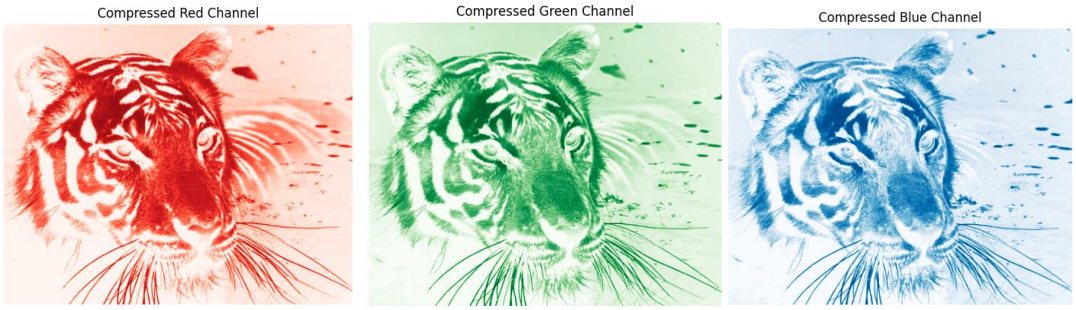
Obraz 1. Tygrys

Wartości osobliwe kanałów obrazu zostały zwizualizowane jako wykresy słupkowe, umożliwiając ocenę ilości istotnych składowych.

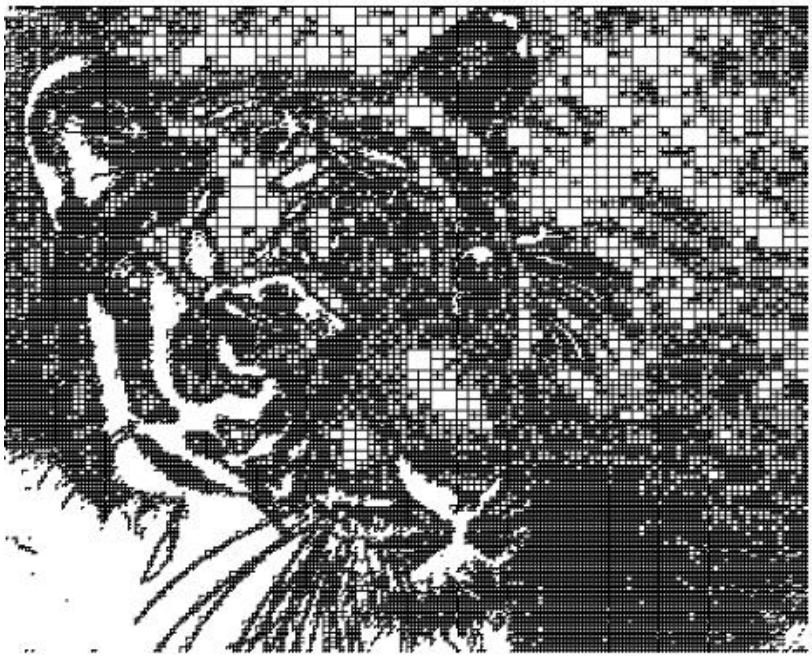


1. **Max_rank = 1, epsilon = 0.052, last_singular**

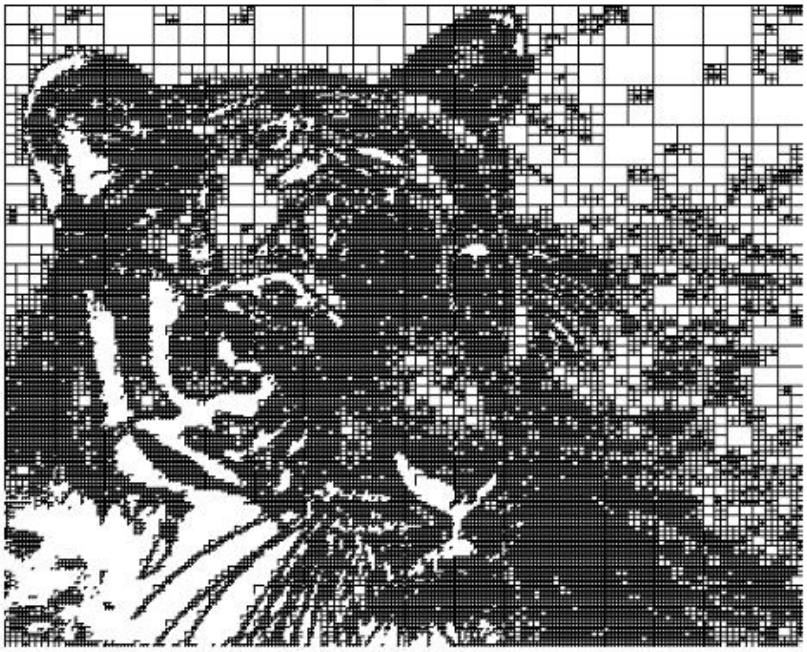




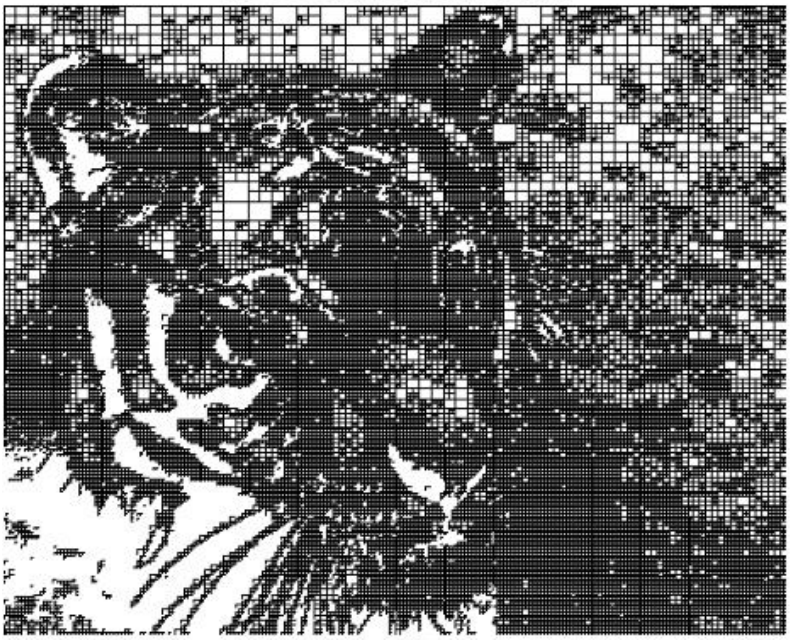
Red Compression Matrix



Green Compression Matrix

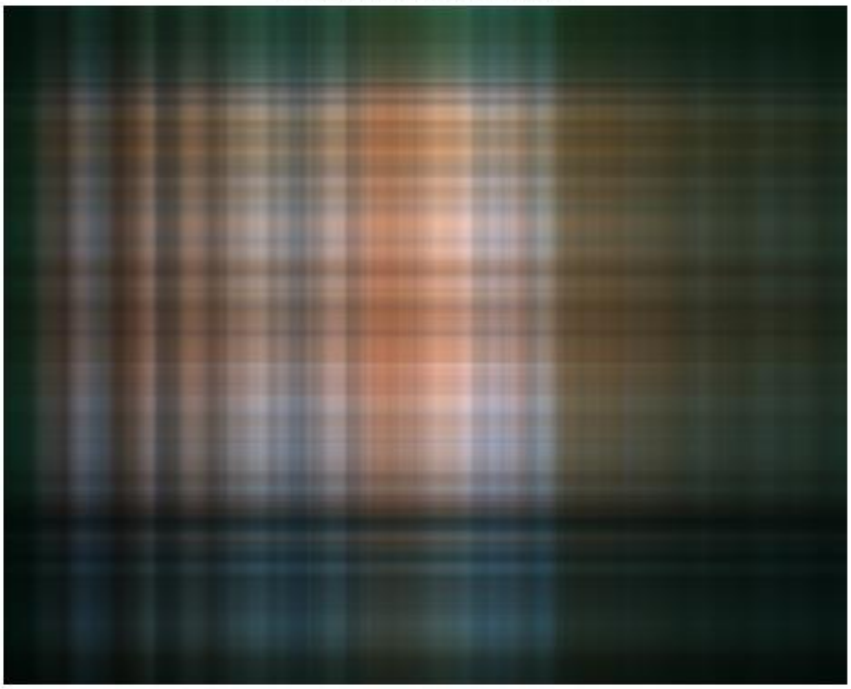


Blue Compression Matrix

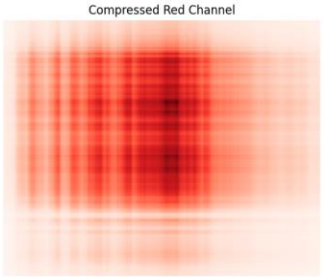


2. **Max_rank = 1, epsilon = 1.55, middle_singular**

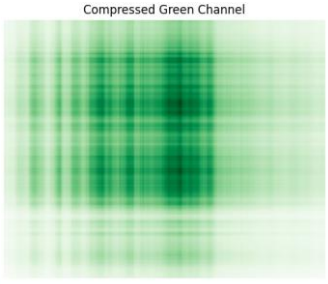
Compressed Image



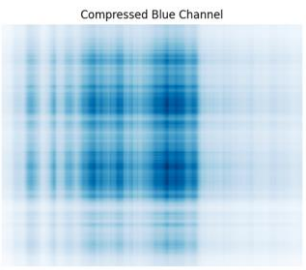
Compressed Red Channel



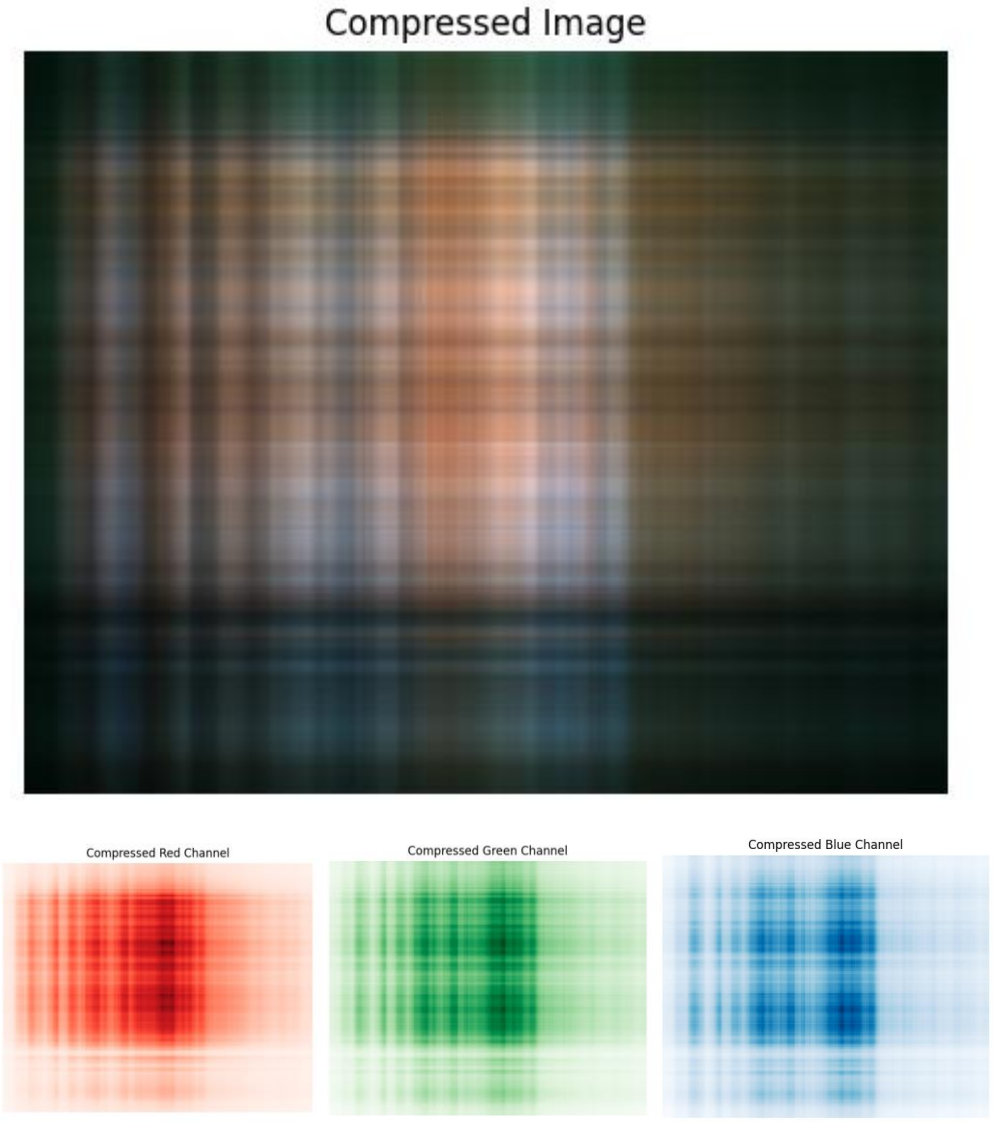
Compressed Green Channel



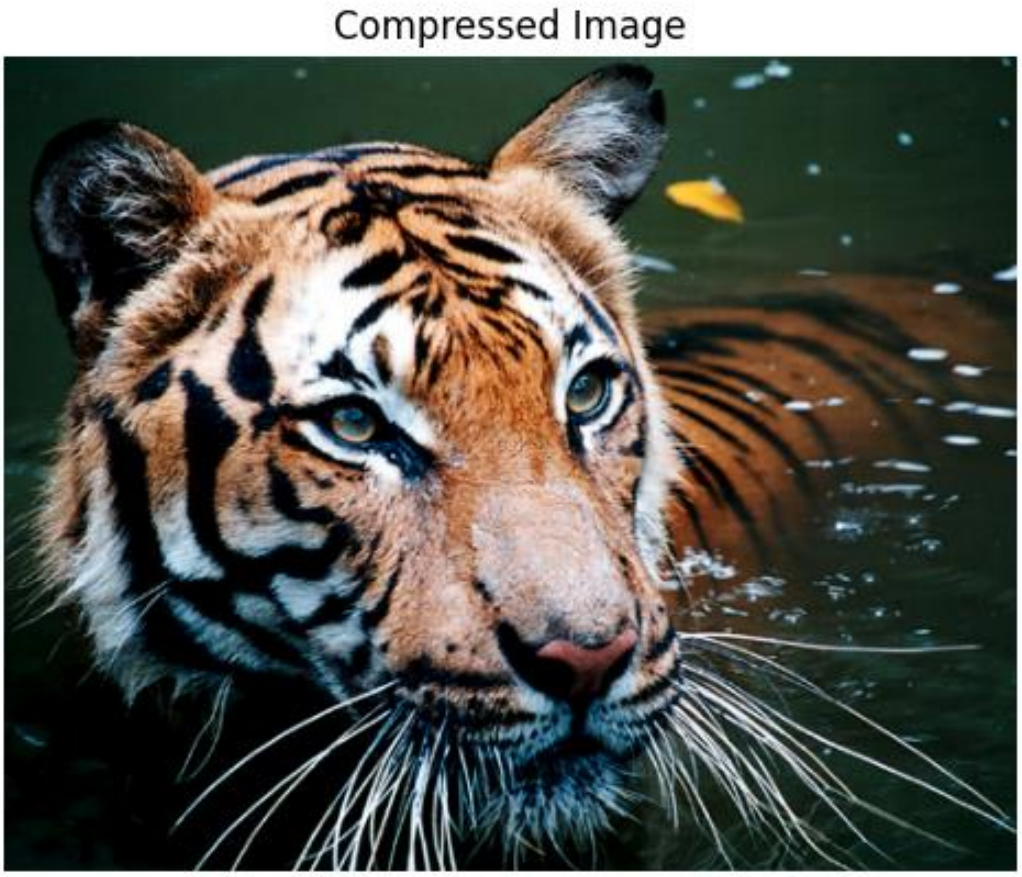
Compressed Blue Channel



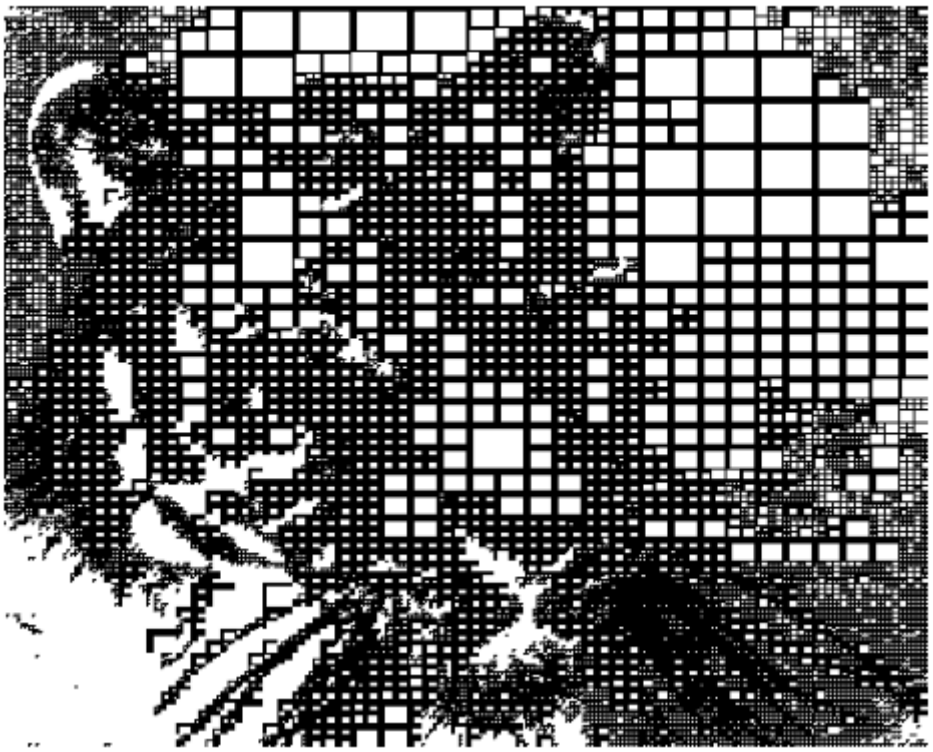
3. **Max_rank = 1, epsilon = 41.101, first_singular**



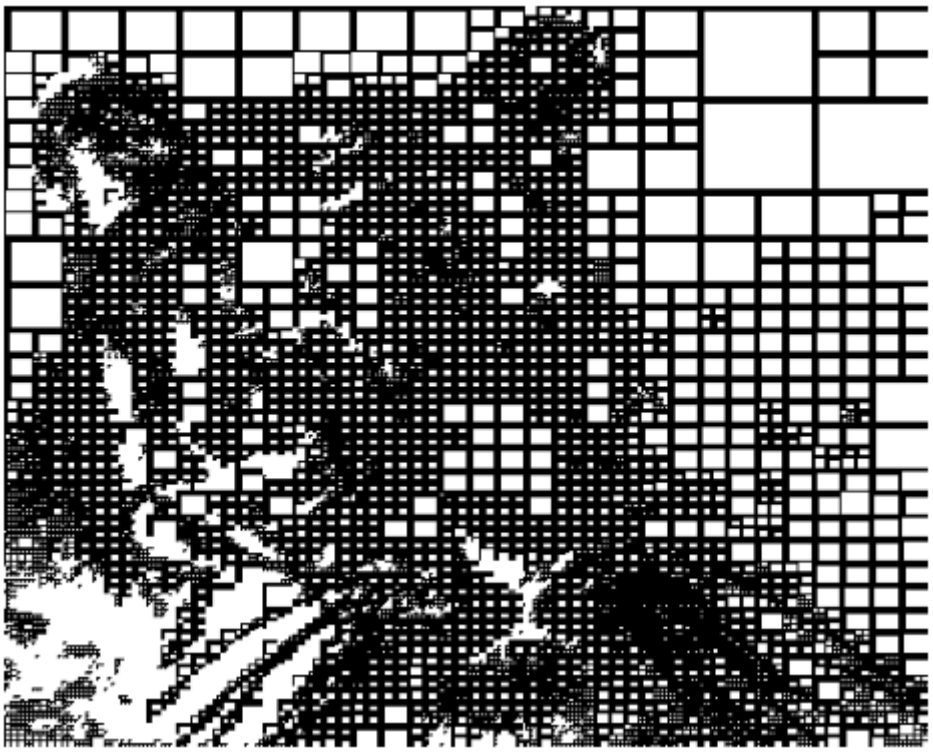
4. **Max_rank = 4, epsilon = 0.52, last_singular**



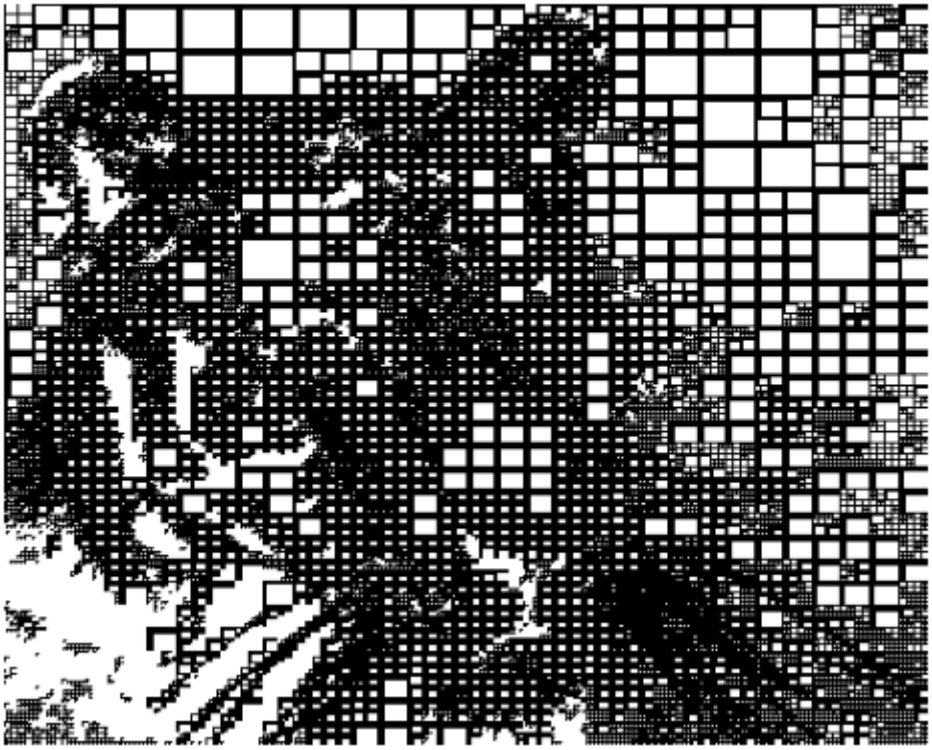
Red Compression Matrix



Green Compression Matrix

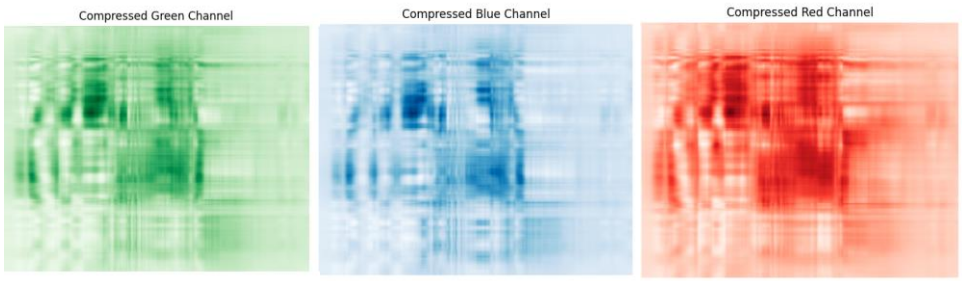


Blue Compression Matrix

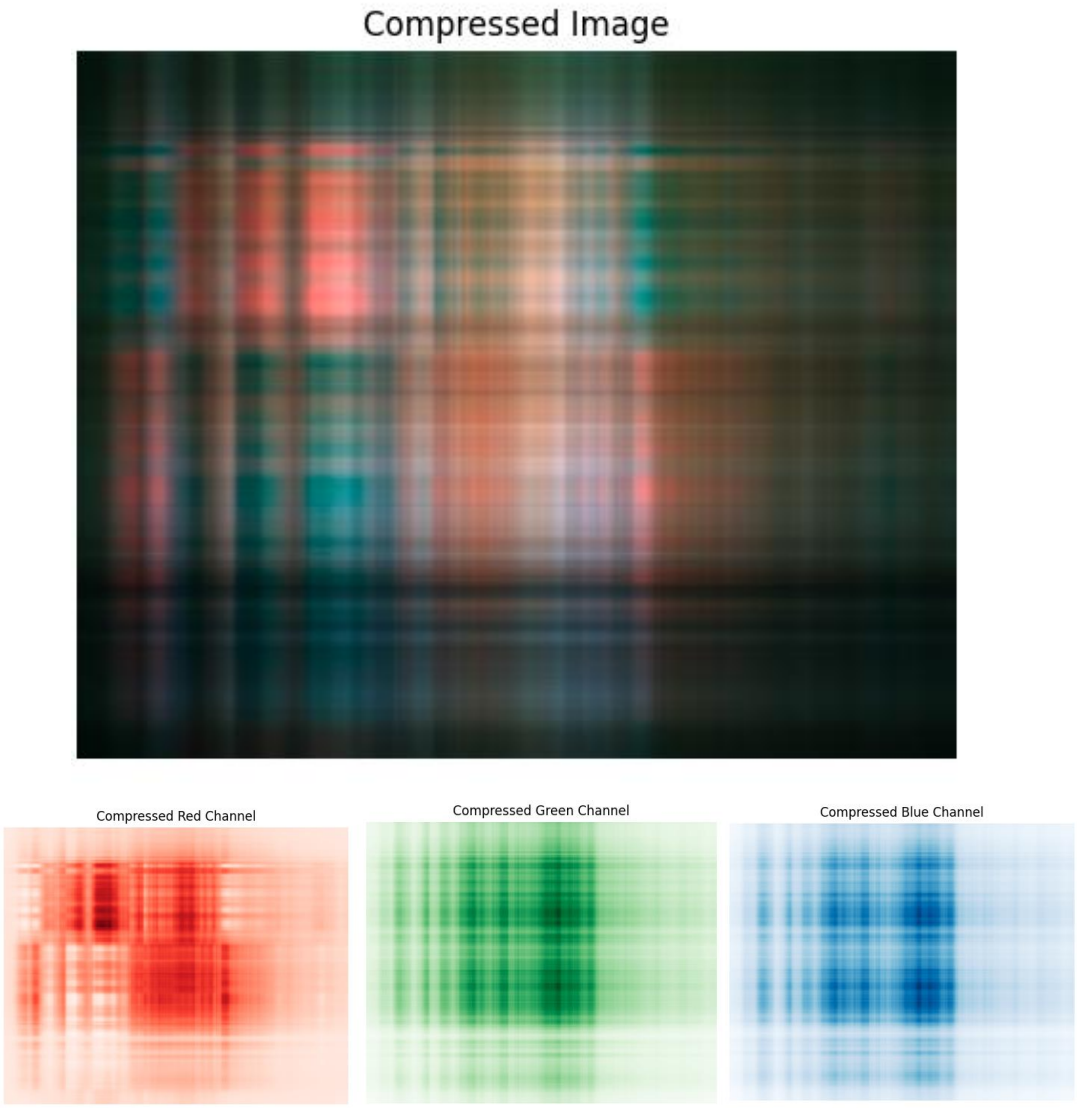


5. **Max_rank = 4, epsilon = 1.55, middle_singular**

Compressed Image



6. **Max_rank = 4, epsilon = 41.101, first_singular**



Wyniki i obserwacja

1. **Zależność błędu rekonstrukcji od wartości osobliwych:**
   - Zachowanie większej liczby wartości osobliwych poprawia jakość rekonstrukcji.
   - Kompresja z odrzuceniem najmniejszych wartości osobliwych umożliwia znaczne zmniejszenie rozmiaru danych przy akceptowalnym błędzie.

2. **Hierarchiczna kompresja:**
    - Dzieląc obraz na mniejsze regiony, można skuteczniej dopasować jakość kompresji do lokalnych właściwości obrazu (np. bardziej szczegółowe regiony wymagają więcej danych).
3. **Wizualne różnice:**
    - Obrazy skompresowane przy niskich wartościach *max_rank* tracą szczegóły w miejscach o wysokiej częstotliwości (np. krawędzie).
4. **Zależność od wartości parametru epsilon:**
    - Przy wyższych wartościach epsilon na wczesnych etapach analizy drzewo kwadrantów nie tworzy dzieci, co powoduje słabą kompresję macierzy. Wynika to z faktu, że błędy rekonstrukcji mieszczą się w granicach tolerancji, przez co algorytm nie dzieli obrazu na podmacierze.
      Natomiast przy niższych wartościach epsilon, szczególnie dla późniejszych wartości osobliwych, algorytm bardziej precyzyjnie dopasowuje podział obrazu na mniejsze regiony, co skutkuje widocznie lepszą kompresją obrazu, uwzględniającą lokalne szczegóły.