

# Inżynieria obrazów

Algorytm JPEG i steganografia

Marcin Lasak 272886

## Spis treści

1. Wstęp teoretyczny .....	3
1.1 JPEG.....	3
1.2 Steganografia.....	4
2. Zadania .....	6
2.1 Pełny algorytm JPEG .....	6
2.1.1 Krok 1 – Konwersja z modelu RGB do YCbCr.....	6
2.1.2 Krok 2 – Podpróbkiwanie kanałów chrominancji .....	6
2.1.3 Krok 3 – Podział na bloki 8x8.....	10
2.1.4 Krok 4 – Dyskretna transformacja kosinusowa .....	11
2.1.5 Krok 5 – Dzielenie przez macierz kwantyzacji.....	12
2.1.6 Krok 6 – Zaokrąglanie.....	12
2.1.7 Krok 7 – Zygzakowanie .....	15
2.1.8 Krok 8 – RLE i kodowanie Huffmana .....	16
2.1.9 Dekodowanie.....	19
2.1.10 Badanie wpływu współczynnika jakości na jakość i rozmiar obrazu .....	21
2.1.11 Wywołanie w programie main.exe.....	22
2.2 Ukrycie wiadomości w obrazie .....	24
2.3 Badanie wpływu liczby użytych najmniej znaczących bitów na jakość obrazka .....	26
2.4 Zapis wiadomości od zadanej pozycji.....	29
2.5 Ukrywanie obrazu w obrazie.....	31
2.6 Odkrywanie obrazu bez przekazywania długości wiadomości .....	33
Źródła .....	36

# 1. Wstęp teoretyczny

## 1.1 JPEG

JPEG (Joint Photographic Experts Group) to format grafiki rastrowej i algorytm stratnej jej kompresji. Pozwala on na zmniejszenie rozmiaru pliku graficznego, kosztem jego jakości. Wykorzystuje on jednak słabości ludzkiej percepcji wzrokowej, aby teoretycznie mniej dokładnie zapisany obraz, wydawał się tak dokładny jak oryginał. Poszczególne kroki algorytmu JPEG to:

1. Konwersja modelu barw: RGB  $\rightarrow$  YCbCr.
2. Przeskalowanie w dół (stratne) macierzy składowych Cb i Cr.
3. Podział obrazu na bloki o rozmiarze 8x8.
4. Wykonanie dyskretnej transformacji cosinusowej na każdym bloku obrazu.
5. Podzielenie każdego bloku obrazu przez macierz kwantyzacji.
6. Zaokrąglenie wartości w każdym bloku do liczb całkowitych.
7. Zwinięcie każdego bloku 8x8 do wiersza 1x64 – algorytm ZigZag.
8. Zakodowanie danych obrazu – m.in. algorytmem Huffmana.

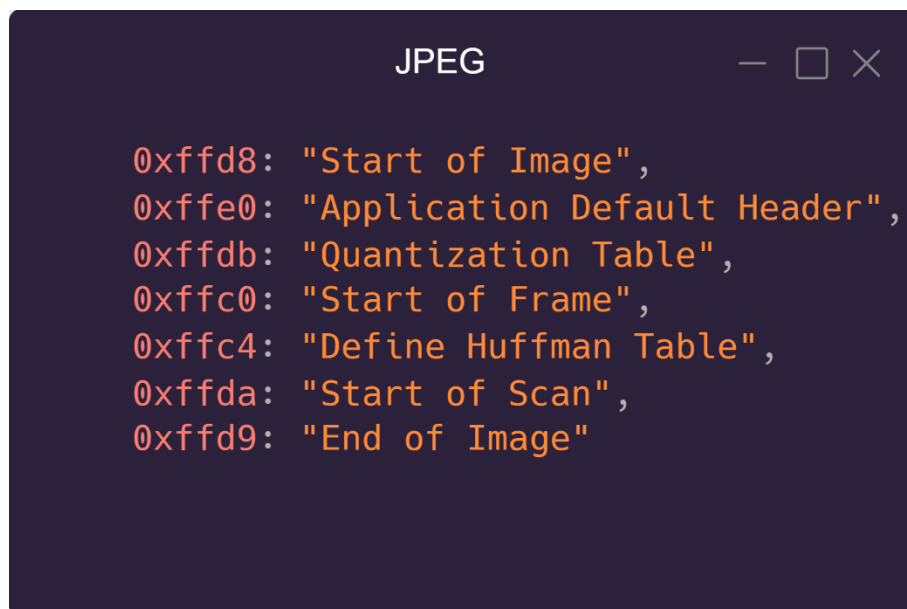
Krok 2 jest kluczowy w aspekcie wykorzystywania słabości ludzkiego wzroku. Okazuje się bowiem, że ludzkie oko jest dużo wrażliwsze na jasność (luminację), niż na kolory (chrominację). Z tego powodu możemy zmniejszyć rozdzielczość kolorowania obrazu, gdyż większość informacji pobieranych przez nas oczami i tak wyrażana jest w luminacji. Typowo możemy próbować co drugi lub co czwarty piksel, co skutecznie zmniejszy ilość informacji na temat koloru dwu lub czterokrotnie, zmniejszając rozmiar pliku.

Zygzakowanie jest sposobem uporządkowania współczynników otrzymanych po wykonaniu dyskretnej transformacji kosinusowej. Większość współczynników po kwantyzacji jest zerowa, a zygzakowanie (algorytm ZigZag) pozwala nam uporządkować je tak, aby zera znajdowały się na końcu tablicy.

Kodowanie Huffmana to metoda poradzenia sobie z tą dużą ilością zer. Kodowanie Huffmana polega na utworzeniu słów kodowych (ciągów bitowych), których długość jest odwrotnie proporcjonalna do prawdopodobieństwa ich wystąpienia. Tzn. im częściej dany symbol występuje, tym mniej zajmie bitów. Algorytm ten operuje na drzewach binarnych których węzły to słowa o określonym prawdopodobieństwie. Jest to rodzaj kompresji bezstratnej.

Plik JPEG składa się z kilku segmentów, podobnie jak plik PNG. Sekcje oznaczone są określonymi znacznikami (rysunek 1). Znacznik Start of Image oznacza początek pliku JPEG. Jest to identyfikator formatu. Application Default Header zawiera dodatkowe informacje o pliku, takie jak dane metadane, informacje o wersji formatu JPEG. Kolejny segment, Quantization Table zawiera informacje o używanych tablicach kwantyzacji. Segment Start of Frame zawiera informacje o szerokości i wysokości obrazu, liczbie komponentów kolorów w obrazie, rozdzielczości, długości segmentu oraz współczynniki próbkowania używane przy kompresji. Segment Define Huffman Table zawiera tablice kodowania (słowniki) Huffmana, używane przy kompresji. Segment Start of Scan określa początek danych skanowania obrazu (czyli właściwych danych, które będą kodowane w formie kompresji). Zawiera informacje o sposobie skanowania obrazu, takich jak które tablice Huffmana mają być użyte oraz które komponenty obrazu będą kodowane. SOS informuje o tym, że następują dane obrazu w postaci skompresowanej. Po zakończeniu segmentu SOS bezpośrednio zaczynają się

dane zakodowane za pomocą kodu Huffmana. Są to skompresowane współczynniki dyskretnej transformaty kosinusowej. Dane te trwają aż do ostatniego znacznika – End of Image.



Rysunek 1

## 1.2 Steganografia

Steganografia jest nauką o przekazywaniu informacji w taki sposób, aby obecność komunikatu nie została wykryta. Metody steganografia, mogą nam przywozić na myśl kryptografię, lecz te dwie nauki różnią się na zasadniczym polu: kryptografia zajmuje się ukrywaniem treści komunikacji, natomiast steganografia – ukrywaniem samego faktu odbywania się komunikacji. Przykładowo efektem pracy kryptograficznej, może być zakodowana wiadomość postaci „odvnd”, natomiast efektem pracy steganograficznej może być obraz PNG, z zakodowaną tą samą wiadomością, z tą różnicą, że w pierwszym przypadku wiemy, że jest to wiadomość – w drugim przypadku nie.

Steganografia ma swoje korzenie już w starożytności, przy użyciu analogowych narzędzi przekazu (np. atrament sympatyczny). W dzisiejszych czasach, rozwiązania cyfrowe umożliwiają ukrywanie informacji np. w plikach, w szczególności w plikach graficznych. Pliki graficzne kodują konkretne piksele, najczęściej w formacie RGB na 24 bitach. Małe zmiany tych bitów, nieznacznie wpłyną na odbierany obraz, a możemy w subtelny sposób zakodować na nich ukryte wiadomości, czy też całe obrazy.

Możliwe jest także ukrywanie informacji w plikach wideo, audio, czy choćby w plikach tekstowych.

Podstawowym algorytmem steganograficznym dla plików graficznych jest algorytm LSB, który zamienia odpowiednie bity kodujące piksele obrazu tak, aby różnica w zmodyfikowanym pliku nie była dostrzegalna dla ludzkiego oka. Konkretniej korzysta z określonej liczby najmłodszych bitów danej współrzędnej koloru w modelu RGB.

Jednym z najpopularniejszych zastosowań steganografii jest ukrywanie znaku wodnego, bądź innych danych umożliwiających weryfikację autorstwa danego pliku. Służy to zidentyfikowaniu źródła plików multimedialnych, które są udostępniane bez pozwolenia<sup>1</sup>.

---

<sup>1</sup> <https://www.techtarget.com/searchsecurity/definition/steganography>

## 2. Zadania

### 2.1 Pełny algorytm JPEG

W tym zadaniu należało dokończyć implementację algorytmu JPEG z laboratorium 2, tj. uzupełnić go o dyskretną transformację kosinusową oraz kwantyzację.

Z uwagi na niezorganizowaną strukturę operacji odwrotnych w laboratorium 2, zdecydowano się na częściową zmianę kodu oraz uporządkowanie go, aby zwiększyć przejrzystość kodu, w szczególności operacji algorytmu odwrotnego. Tak więc przeanalizujemy kod całościowo, z podziałem na poszczególne kroki.

Wszystkie kroki są odpowiednio wykonywane w funkcji `process()`, która będzie odpowiednio omawiana przy każdym z kroków algorytmu.

W programie skorzystano z bibliotek: `numpy` – do generalnych obliczeń, np. na tablicach współrzędnych RGB, `scipy` – do transformacji kosinusowych oraz interpolacji, `Pillow` – do obsługi plików graficznych oraz `collections` i `heapq` do implementacji kodowania Huffmana.

Opisywane zadanie można uruchomić przez program `main.exe` znajdujący się w folderze `dist`, wybierając opcję nr 1.

#### 2.1.1 Krok 1 – Konwersja z modelu RGB do YCbCr

Konwersję z modelu RGB do modelu luminancja-chrominancja realizuje funkcja `stepOne()`. Operację odwrotną realizuje funkcja `reverse_stepOne()`. Obie funkcje korzystają z tablic `numpy` i wykonują na nich operacje wg wzorów na konwersję między współrzędnymi w przestrzeniach RGB i YCbCr (rysunki 3 i 4<sup>2</sup>). Po wykonaniu obliczeń, wartości są zaokrąglane, aby mieściły się w zakresie 0-255. Funkcje te dane są rysunkiem nr 2.

W funkcji `process()` (rysunek 5) wpierw tworzymy tablicę `numpy`, będącą przetwarzanym obrazem, za pomocą używanej w poprzednich laboratoriach funkcji `colors()`, tworzącej spektrum przejść w przestrzeni RGB. Ta tablica jest konwertowana za pomocą funkcji `stepOne()`, a następnie następuje rozdzielenie jej na poszczególne kanały Y, Cb oraz Cr.

Warto w tym miejscu zaznaczyć, że funkcja `process()` przyjmuje jako argumenty, używane w kolejnych krokach parametry: współczynniki próbkowania i jakości.

#### 2.1.2 Krok 2 – Podpróbkowanie kanałów chrominancji

Funkcja `stepTwo()` (rysunek 6) realizuje podpróbkowanie kanałów chrominancji. Jest to prosta operacja na tablicach `numpy`, tworząca tablicę wyjściową z co factor-tych elementów (tzw. slicing z krokiem) tablicy wejściowej (gdzie jednostką w tablicy jest trójka współrzędnych).

W przypadku funkcji realizującą operację odwrotną, tj. nadpróbkowanie, czyli `reverse_stepTwo()` (rysunek 6), jej działanie jest nieco bardziej skomplikowane. Korzystamy z funkcji `zoom` z biblioteki `scipy`, która zmienia rozmiar tablicy, wypełniając brakujące punkty przy użyciu interpolacji liniowej.

---

<sup>2</sup> Wzór na konwersję z YCbCr na rysunku nr 4 jest zaokrąglony, w programie korzystamy z mniej zaokrąglonych wartości na podstawie <https://www.ijg.org/files/T-REC-T.871-201105-I!!PDF-E.pdf>

W funkcji process() (rysunek 7), próbujemy każdy z kanałów osobno. Następnie wyrównujemy tablice (funkcje dane rysunkiem 8), przed krokiem trzecim.

```
JPEG.py

#krok 1 : konwersja z RGB do YCbCr
def stepOne(RGBarray):
    print(f"stepOne: RGB -> YCbCr. Rozmiar obrazu: {RGBarray.shape}")
    width = RGBarray.shape[1]
    height = RGBarray.shape[0]

    ycbcr = np.zeros((height, width, 3), dtype=np.uint8)

    for j in range(height):
        for i in range(width):
            R, G, B = RGBarray[j, i]

            Y = 0.299 * R + 0.587 * G + 0.114 * B
            Cb = -0.1687 * R - 0.3313 * G + 0.5 * B + 128
            Cr = 0.5 * R - 0.4187 * G - 0.0813 * B + 128

            ycbcr[j, i] = [np.clip(Y, 0, 255), np.clip(Cb, 0, 255), np.clip(Cr, 0, 255)]

    return ycbcr

# odwrotny krok 1: konwersja z YCbCr do RGB
def reverse_stepOne(ycbcr_array):
    height, width = ycbcr_array.shape[:2]
    rgb = np.zeros((height, width, 3), dtype=np.uint8)

    for j in range(height):
        for i in range(width):
            Y, Cb, Cr = ycbcr_array[j, i]

            R = Y + 1.402 * (Cr - 128)
            G = Y - 0.344136 * (Cb - 128) - 0.714136 * (Cr - 128)
            B = Y + 1.772 * (Cb - 128)

            rgb[j, i] = [np.clip(R, 0, 255), np.clip(G, 0, 255), np.clip(B, 0, 255)]

    return rgb
```

Rysunek 2

$$\begin{aligned} Y &= 0,299 * R + 0,587 * G + 0,114 * B \\ C_b &= -0,1687 * R - 0,3133 * G + 0,5 * B + 128 \\ C_r &= 0,5 * R - 0,4187 * G - 0,0813 * B + 128 \end{aligned}$$

Rysunek 3

$$\begin{aligned}
 R &= \text{Min}(\text{Max}(0, \text{Round}(Y + 1.402 * (C_R - 128))), 255) \\
 G &= \text{Min}(\text{Max}(0, \text{Round}(Y - 0.3441 * (C_B - 128) - 0.7141 * (C_R - 128))), 255) \\
 B &= \text{Min}(\text{Max}(0, \text{Round}(Y + 1.772 * (C_B - 128))), 255)
 \end{aligned}$$

Rysunek 4

```

JPEG.py

def process(sampling_factor, QF):
    # --- 0. Generowanie obrazu ---
    img = colors()

    # --- 1. RGB -> YCbCr ---
    ycbcr = stepOne(img)

    #rozdzielenie kanałów
    Y = ycbcr[:, :, 0]
    Cb = ycbcr[:, :, 1]
    Cr = ycbcr[:, :, 2]

```

Rysunek 5

```

JPEG.py

#krok 2: podpróbkiwanie
def stepTwo(channel, factor):
    print(f"stepThree: Próbkowanie (factor={factor}). Rozmiar przed
    próbkowaniem: {channel.shape}")
    if factor == 1:
        return channel
    channel_sampled = channel[::factor, ::factor]
    print(f"stepThree: Rozmiar po próbkowaniu: {channel_sampled.shape}")
    return channel_sampled

#odwrotny krok 2: nadpróbkiwanie
def reverse_stepTwo(channel_sampled, factor):
    if factor == 1:
        return channel_sampled

    from scipy.ndimage import zoom
    channel_upsampled = zoom(channel_sampled, factor, order=1)
    return channel_upsampled

```

Rysunek 6



```
JPEG.py

# --- 2.Próbkowanie Cb i Cr ---
factor = sampling_factor
Cb_sub = stepTwo(Cb, factor)
Cr_sub = stepTwo(Cr, factor)

#wyrównanie
Y_padded = padding(Y)
Cb_padded = padding(Cb_sub)
Cr_padded = padding(Cr_sub)
Cb = ycbcr[:, :, 1]
Cr = ycbcr[:, :, 2]
```

Rysunek 7

```
JPEG.py

# wyrównanie
def padding(arrayChannel):
    print(f"stepTwo: Padding. Rozmiar przed paddingiem: {arrayChannel.shape}")
    height, width = arrayChannel.shape
    h_pad = (8 - height % 8) % 8
    w_pad = (8 - width % 8) % 8
    channel_padded = np.pad(arrayChannel, ((0, h_pad), (0, w_pad)), mode='reflect')
    print(f"stepTwo: Rozmiar po paddingu: {channel_padded.shape}")
    return channel_padded

#usunięcie wyrównania
def remove_padding(channel_padded, original_height, original_width):
    print(f"remove_padding: Rozmiar przed usunięciem paddingu: {channel_padded.shape}")
    channel_unpadded = channel_padded[:original_height, :original_width]
    print(f"remove_padding: Rozmiar po usunięciu paddingu: {channel_unpadded.shape}")
    return channel_unpaddedchannel_sampled, factor, order=1)
    return channel_upsampled
```

Rysunek 8

### 2.1.3 Krok 3 – Podział na bloki 8x8

Kolejnym krokiem jest podział kanałów na bloki rozmiaru 8x8, które będą dalej przetwarzane. Służy do tego funkcja `stepThree()` dana rysunkiem nr 9.

```
JPEG.py

def stepThree(channel):

    height, width = channel.shape
    blocks = []
    for i in range(0, height, 8):
        for j in range(0, width, 8):
            block = channel[i:i+8, j:j+8]
            blocks.append(block)
    return blocks

#odwrotny krok 3: scalenie bloków 8x8
def reverse_stepThree(blocks, original_height, original_width):
    print(f"reverse_stepThree: Scalanie bloków. Liczba bloków: {len(blocks)}")
    padded_height = ((original_height + 7) // 8) * 8
    padded_width = ((original_width + 7) // 8) * 8

    channel = np.zeros((padded_height, padded_width), dtype=np.float32)
    idx = 0
    for i in range(0, padded_height, 8):
        for j in range(0, padded_width, 8):
            channel[i:i + 8, j:j + 8] = blocks[idx]
            idx += 1
    return channel
```

Rysunek 9

Funkcja ta tworzy listę, a następnie uzupełnia ją blokami 8x8 utworzonymi z tablicy reprezentującej dany kanał.

Funkcja `reverse_stepThree()` jest funkcją odwrotną, która scala przetworzone bloki 8x8 w jeden kanał.

W funkcji `process()` (rysunek 10), dzielimy na bloki kolejno każdy kanał.

```
JPEG.py

# --- 3.Podział na bloki 8x8 ---
Y_blocks = stepThree(Y_padded)
Cb_blocks = stepThree(Cb_padded)
Cr_blocks = stepThree(Cr_padded)
```

Rysunek 10

### 2.1.4 Krok 4 – Dyskretna transformacja kosinusowa

Na każdym z bloków uzyskanym w kroku 3, wykonujemy dyskretną transformację kosinusową za pomocą funkcji `stepFour()` (rysunek 11).



```
JPEG.py

#dyskretna transformata kosinusowa
def dct2(array):
    return dct(dct(array, axis=0, norm='ortho'), axis=1, norm='ortho')

#odwrotna dyskretna transformata kosinusowa
def idct2(array):
    return idct(idct(array, axis=0, norm='ortho'), axis=1,
               norm='ortho')

#krok 4: transformata kosinusowa każdego bloku
def stepFour(blocks):
    blocks=[dct2(block) for block in blocks]
    return blocks

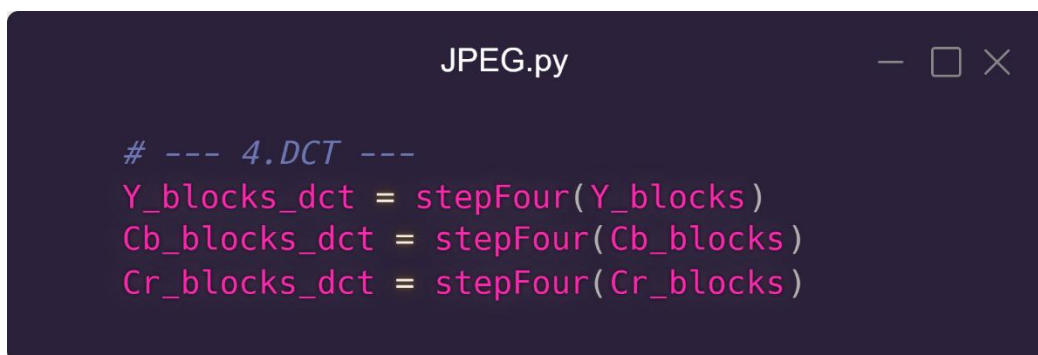
#odwrotny krok 4: odwrotna transformata kosinusowa każdego bloku
def reverse_stepFour(blocks):
    blocks = [idct2(block) for block in blocks]
    return blocks
```

Rysunek 11

Funkcja ta korzysta z funkcji `dct2()`, która to wywołuje funkcję z biblioteki `scipy` odpowiedzialną za transformatę kosinusową – najpierw po kolumnach, a następnie po wierszach. Dyskretna transformata kosinusowa pozwala na przejście z dziedziny przestrzennej (piksele), na dziedzinę częstotliwości (częstość zmiany pikseli).

Funkcja `reverse_stepFour()` jest operacją odwrotną do kroku czwartego i wykonuje odwrotną transformację kosinusową na przetworzonych blokach.

W funkcji `process()` (rysunek 12) wykonujemy funkcję `stepFour()` kolejno dla bloków każdego kanału.



```
JPEG.py

# --- 4.DCT ---
Y_blocks_dct = stepFour(Y_blocks)
Cb_blocks_dct = stepFour(Cb_blocks)
Cr_blocks_dct = stepFour(Cr_blocks)
```

Rysunek 12

### 2.1.5 Krok 5 – Dzielenie przez macierz kwantyzacji

Po zmianie dziedziny, dzielimy otrzymane bloki przez odpowiednie macierze kwantyzacji, zadane funkcjami `get_quantization_matrix_luminance()` oraz `get_quantization_matrix_chrominance()` (rysunek 13). Macierz kwantyzacji w JPEG jest inna dla luminancji i chrominancji, bo ludzkie oko jest dużo bardziej wrażliwe na zmiany jasności niż na zmiany koloru, stąd wartości w macierzy dla luminancji są średnio mniejsze od wartości w macierzy dla chrominancji, która w większości wypełniona jest wartością 99. Kwantyzacja ma na celu spłaszczenie lub wyzerowanie tych danych, które są mniej ważne dla oka.

Każdy blok jest dzielony w funkcji `stepFive()` (rysunek 14) przez macierz powstałą w zależności od QF (im mniejsze QF, tym większa stratność kompresji). Wykorzystujemy tu proste dzielenie tablic numpy. Zwracana jest lista bloków po kwantyzacji.

Operacja odwrotna, zadana funkcją `reverse_stepFive()` (rysunek 14), polega na wymnażaniu przetworzonych bloków, przez odpowiednie macierze kwantyzacji. Znow korzystając z tablic numpy, wymnażany jest każdy element tablicy, przez odpowiadający mu element macierzy kwantyzacji. Widzimy jasno, że jest to rodzaj kompresji stratnej – nie mamy możliwości odwrócenia procesu zaokrąglania w kroku 6. Co więcej samo dzielenie zmiennoprzecinkowe może być operacją stratną, więc nawet przed etapem zaokrąglania możemy utracić część informacji.

W funkcji `process()` (rysunek 15), wpierw tworzymy macierze kwantyzacji odpowiednio dla luminancji i chrominancji, a następnie wykonujemy funkcję `stepFive()` na każdej liście bloków otrzymanych po transformacji kosinusowej.

### 2.1.6 Krok 6 – Zaokrąglanie

Otrzymane w kroku 5 bloki, należy zaokrąglić. W algorytmie JPEG pracujemy na liczbach całkowitych. Wynika to z mniejszego rozmiaru tak zapisanych danych, prostszego odczytu oraz łatwiejszego przetwarzania, np. przy kodowaniu Huffmana, gdzie liczby całkowite są dużo łatwiejsze i tańsze do zakodowania.

Zaokrąglanie realizowane jest za pomocą funkcji `stepSix()` daną rysunkiem nr 16. Używamy w niej narzędzia biblioteki numpy, mianowicie `round`, które pozwala na zaokrąglenie wszystkich elementów zadanej tablicy. Tak więc zaokrąglamy każdy blok z listy i zwracamy listę zaokrąglonych bloków.

Krok 6 z oczywistych powodów nie posiada operacji odwrotnej. Nie jest możliwe odwrócenie zaokrąglania, bez żadnych dodatkowych informacji np. przyrost zaokrąglenia względem oryginału. Tak więc jest to operacja, przy której tracimy informacje na temat obrazu.

W funkcji `process()` (rysunek 17) zaokrąglamy kolejno listy bloków każdego kanału.

```

JPEG.py

# macierz kwantyzacji dla luminancji
def get_quantization_matrix_luminance(QF):
    Q_base = np.array([
        [16, 11, 10, 16, 24, 40, 51, 61],
        [12, 12, 14, 19, 26, 48, 60, 55],
        [14, 13, 16, 24, 40, 57, 69, 56],
        [14, 17, 22, 29, 51, 87, 80, 62],
        [18, 22, 37, 56, 68, 109, 103, 77],
        [24, 35, 55, 64, 81, 104, 113, 92],
        [49, 64, 78, 87, 103, 121, 120, 101],
        [72, 92, 95, 98, 112, 100, 103, 99]
    ], dtype=np.float64)

    if QF < 50:
        S = 5000 / QF
    else:
        S = 200 - 2 * QF

    Q = np.floor((Q_base * S + 50) / 100)
    Q = np.clip(Q, 1, 255)
    return Q

# macierz kwantyzacji dla chrominancji
def get_quantization_matrix_chrominance(QF):
    Q_base = np.array([
        [17, 18, 24, 47, 99, 99, 99, 99],
        [18, 21, 26, 66, 99, 99, 99, 99],
        [24, 26, 56, 99, 99, 99, 99, 99],
        [47, 66, 99, 99, 99, 99, 99, 99],
        [99, 99, 99, 99, 99, 99, 99, 99],
        [99, 99, 99, 99, 99, 99, 99, 99],
        [99, 99, 99, 99, 99, 99, 99, 99],
        [99, 99, 99, 99, 99, 99, 99, 99]
    ], dtype=np.float64)

    if QF < 50:
        S = 5000 / QF
    else:
        S = 200 - 2 * QF

    Q = np.floor((Q_base * S + 50) / 100)
    Q = np.clip(Q, 1, 255)
    return Q

```

Rysunek 13



```
JPEG.py

#krok 5: kwantyzacja
def stepFive(blocks, quant_matrix):

    quantized_blocks = []
    for block in blocks:
        quantized = block / quant_matrix
        quantized_blocks.append(quantized.astype(np.int32))
    return quantized_blocks

#odwrotny krok 5: dekwantyzacja
def reverse_stepFive(blocks, quant_matrix):

    dequantized_blocks = []
    for block in blocks:
        dequantized = block * quant_matrix
        dequantized_blocks.append(dequantized)
    return dequantized_blocks
```

Rysunek 14

```
JPEG.py

# --- 5.Kwantyzacja ---
QY = get_quantization_matrix_luminance(QF)
QC = get_quantization_matrix_chrominance(QF)

Y_blocks_quant = stepFive(Y_blocks_dct, QY)
Cb_blocks_quant = stepFive(Cb_blocks_dct, QC)
Cr_blocks_quant = stepFive(Cr_blocks_dct, QC)
```

Rysunek 15

```
JPEG.py

#krok 6: zaokrąglanie
def stepSix(blocks):

    rounded_blocks = []
    for block in blocks:
        rounded = np.round(block)
        rounded_blocks.append(rounded.astype(np.int32))
    return rounded_blocks
```

Rysunek 16

```
JPEG.py

# --- 6.Zaokrąglanie ---
rounded_Y_blocks = stepSix(Y_blocks_quant)
rounded_Cb_blocks = stepSix(Cb_blocks_quant)
rounded_Cr_blocks = stepSix(Cr_blocks_quant)
```

Rysunek 17

### 2.1.7 Krok 7 – Zygzakowanie

Kolejnym krokiem jest wykonanie algorytmu ZigZag na każdym bloku dla każdego kanału. Zygzakowanie polega na zmianie macierzy 8x8 na wektor, przy pomocy określonej indeksacji tychże elementów. W celu implementacji zygzakowania, wpierw tworzymy funkcję zwracającą kolejność indeksacji dla macierzy 8x8 (rysunek 18).

```
JPEG.py

#indeksy zigzag
def get_zigzag_index():
    return [
        (0,0),(0,1),(1,0),(2,0),(1,1),(0,2),(0,3),(1,2),
        (2,1),(3,0),(4,0),(3,1),(2,2),(1,3),(0,4),(0,5),
        (1,4),(2,3),(3,2),(4,1),(5,0),(6,0),(5,1),(4,2),
        (3,3),(2,4),(1,5),(0,6),(0,7),(1,6),(2,5),(3,4),
        (4,3),(5,2),(6,1),(7,0),(7,1),(6,2),(5,3),(4,4),
        (3,5),(2,6),(1,7),(2,7),(3,6),(4,5),(5,4),(6,3),
        (7,2),(7,3),(6,4),(5,5),(4,6),(3,7),(4,7),(5,6),
        (6,5),(7,4),(7,5),(6,6),(5,7),(6,7),(7,6),(7,7),
    ]
```

Rysunek 18

Zygzakowanie pojedynczego bloku polega na przypisaniu indeksów elementów danego bloku, za pomocą funkcji `get_zigzag_index()` w funkcji `zigzag_block` (rysunek 19).

```
JPEG.py

#zygzakowanie pojedynczego bloku
def zigzag_block(block):
    zigzag_index = get_zigzag_index()
    return [block[i, j] for i, j in zigzag_index]
```

Rysunek 19

Przy tak zdefiniowanych funkcjach, krok 7 realizowany jest przez funkcję `stepSeven()` (rysunek 20), która wykonuje funkcję `zigzag_block` na każdym bloku z wejściowej listy. Zwraca ona listę tablic jednowymiarowych utworzonych przez algorytm ZigZag.

Funkcją realizującą operację odwrotną jest `reverse_stepSeven()` (rysunek 20). Niejako „odbudowuje” ona blok 8x8 na podstawie tablicy jednowymiarowej.



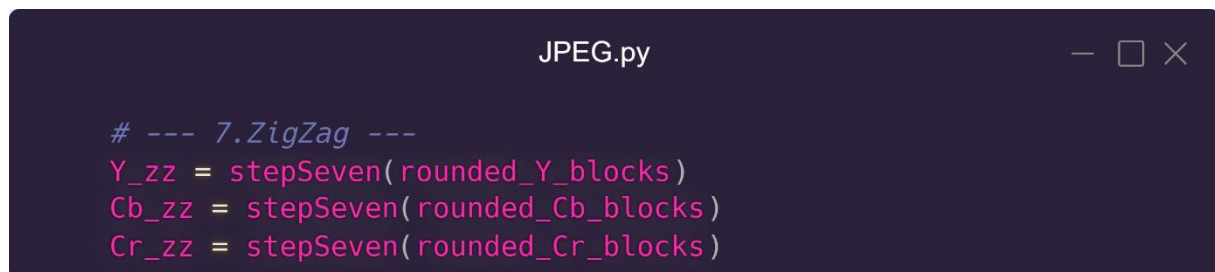
```
JPEG.py

#krok 7: zygzakowanie każdego bloku
def stepSeven(blocks):
    result=[zigzag_block(block) for block in blocks]
    return result

#odwrotny krok 7: dezygzakowanie każdego bloku
def reverse_stepSeven(zigzag):
    zigzag_index = get_zigzag_index()
    block = np.zeros((8, 8), dtype=np.int16)
    for idx, (i, j) in enumerate(zigzag_index):
        block[i, j] = zigzag[idx]
    return block
```

Rysunek 20

W funkcji `process()` wywołujemy funkcję `stepSeven()` na liście bloków dla każdego z kanałów (rysunek 21).



```
JPEG.py

# --- 7.ZigZag ---
Y_zz = stepSeven(rounded_Y_blocks)
Cb_zz = stepSeven(rounded_Cb_blocks)
Cr_zz = stepSeven(rounded_Cr_blocks)
```

Rysunek 21

### 2.1.8 Krok 8 – RLE i kodowanie Huffmana

Ostatnim krokiem jest RLE i kodowanie Huffmana. Krok 8 realizuje funkcja `stepEight()` z wykorzystaniem funkcji `build_huffman_tree()` (rysunek 22).

Funkcja `build_huffman_tree()` korzysta z biblioteki `heapq` aby stworzyć drzewo Huffmana, a następnie zwraca utworzony słownik kodów odpowiadających kodowanym symbolom.

Kopiec przechowuje w węzłach informacje w postaci `[częstotliwość, [symbol, kod]]`. Dopóki w kopcu są więcej niż dwa elementy, wyciągamy dwa najmniejsze elementy. Następnie dodajemy do nich prefiksy binarne, aby oznaczyć konkretne rozgałęzienia drzewa. Po tym, łączymy te elementy w jednym węźle z nową częstotliwością i wkładamy go ponownie do kopca.

Po zakończeniu, w kopcu pozostaje jeden element zawierający całe drzewo Huffmana. `heap[0][1:]` zawiera listę symboli i ich kodów bitowych. W wyniku tego powstaje słownik, który zawiera przypisanie każdego symbolu do jego kodu Huffmana.



W funkcji `stepEight()` zaczynamy od zakodowania wektorów będących wynikiem zygzakowania, przy pomocy RLE (**R**un-**L**ength-**E**ncoding). Jest to prosta metoda kompresji, polegająca na zapisywaniu ciągów powtarzających się znaków przy pomocy liczby i danego znaku. Przykładowo ciąg „AAABB” zapisalibyśmy jako „3A2B”. Dla każdego bloku (wektora) tworzymy listę RLE.

Następnie tworzymy słownik przy pomocy funkcji `build_huffman_tree()`. Przy jego pomocy kodujemy każdy blok RLE i zapisujemy jako strumień bitów.

```
JPEG.py

# Drzewo huffmana + słownik kodowy
def build_huffman_tree(symbols):
    freq = Counter(symbols)
    heap = [[weight, [symbol, "]] for symbol, weight in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]: pair[1] = '0' + pair[1]
        for pair in hi[1:]: pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    huffman_dict = {symbol: code for symbol, code in heap[0][1:]}
    return huffman_dict

#krok 8: odowanie RLE + Huffman dla bloków ZigZag
def stepEight(zigzag_blocks):
    encoded_blocks = []
    all_symbols = []

    for block in zigzag_blocks:
        rle = []
        zeros = 0

        for val in block:
            if val == 0:
                zeros += 1
            else:
                rle.append((zeros, val))
                all_symbols.append((zeros, val))
                zeros = 0
        rle.append((0, 0)) # EOB
        all_symbols.append((0, 0))
        encoded_blocks.append(rle)

    huffman_dict = build_huffman_tree(all_symbols)
    bitstream = "".join(huffman_dict[symbol] for rle in encoded_blocks for
symbol in rle)
    print(f"stepEight: Huffman. Liczba bitów: {len(bitstream)}")
    return bitstream, huffman_dict
```

Funkcją realizującą operację odwrotną jest `reverse_stepEight()` (rysunek 23), która w pierwszej dekoduje bloki zakodowane w kodzie Huffmana, przy pomocy słownika, a następnie dekompresuje listę RLE do postaci tablic po zygzakowaniu.

```
JPEG.py

#odwrotny krok 8: dekodowanie
def reverse_stepEight(bitstream, huffman_dict, num_blocks):
    reverse_dict = {v: k for k, v in huffman_dict.items()}
    decoded_blocks = []
    current_block = []
    buffer = ""
    blocks_decoded = 0

    for bit in bitstream:
        buffer += bit
        if buffer in reverse_dict:
            symbol = reverse_dict[buffer]
            if symbol == (0, 0): # EOB
                current_block += [0] * (64 - len(current_block))
                decoded_blocks.append(current_block)
                current_block = []
                blocks_decoded += 1
                if blocks_decoded >= num_blocks:
                    break
            else:
                zeros, val = symbol
                current_block += [0]*zeros + [val]
                buffer = ""

    print(f"decode_bitstream: Liczba zdekodowanych bloków: {len(decoded_blocks)}")
    return decoded_blocks
```

Rysunek 23

W funkcji `process()` (rysunek 24) wywołujemy funkcję `stepEight()` kolejno dla każdego z kanałów.

```
JPEG.py

# --- 8. RLE + Huffman ---
Y_encoded, Y_huffman = stepEight(Y_zz)
Cb_encoded, Cb_huffman = stepEight(Cb_zz)
Cr_encoded, Cr_huffman = stepEight(Cr_zz)
```

Rysunek 24

Otrzymujemy w ten sposób zakodowany binarnie obraz oraz słownik Huffmana (rysunek 25).

```
JPEG.py

Y_encoded:
00010001001010101100110100001000100110100001100100010011110011011111001111
0000001001111011101111100010001001111101101111100010001001111111011111010
11011000011000000001000011010000110000001101011010101011110100011000001010
10110101010111101000110000011000110000100000110000101101011001101000011000
01100100001100100011000011111011111001111000000110001000110111110010001100

Y_huffman:
{(0, 0): '00', (0, np.int32(-2)): '010000', (9, np.int32(-1)):
'010001000', (0, np.int32(7)): '0100010010', (0, np.int32(14)):
'0100010011', (12, np.int32(-2)): '01000101', (0, np.int32(-11)):
'0100011', (4, np.int32(-2)): '0100100', (13, np.int32(1)): '0100101',
(20, np.int32(1)): '01001100', (0, np.int32(-9)): '01001101', (0,
np.int32(11)): '01001110', (0, np.int32(21)): '0100111100', (0,
np.int32(28)): '0100111101', (0, np.int32(36)): '0100111110', (0,
np.int32(43)): '0100111111', (0, np.int32(1)): '01010', (12,
```

Rysunek 25

### 2.1.9 Dekodowanie

Tak zakodowany obraz, dekodujemy za pomocą operacji odwrotnych opisanych w punktach 2.1.1 – 2.1.8. Wykonanie tych operacji w funkcji `process()` dane jest rysunkiem nr 26.

Wynikiem funkcji `process()` jest obraz po kompresji i dekompresji.

Obrazy wejściowy i wyjściowy dane są rysunkami 27 i 28 (dla `sampling_factor=2`, `QF=100`, dodano niebieskie obramowanie w sprawozdaniu). Błąd średniokwadratowy pomiędzy tymi obrazami wynosi ok. 1,63. W folderze programu zapisane są odpowiednio jako `original_image.png` i `reconstructed_image.png`.



```

# --- [Dekodowanie] ---

# --- reverse 8.Huffman + RLE dekodowanie ---
Y_zz_decoded = reverse_stepEight(Y_encoded, Y_huffman, len(Y_blocks))
Cb_zz_decoded = reverse_stepEight(Cb_encoded, Cb_huffman, len(Cb_blocks))
Cr_zz_decoded = reverse_stepEight(Cr_encoded, Cr_huffman, len(Cr_blocks))

# --- reverse 7.ZigZag odwrotny ---
Y_blocks_dezigzag = [reverse_stepSeven(block) for block in Y_zz_decoded]
Cb_blocks_dezigzag = [reverse_stepSeven(block) for block in Cb_zz_decoded]
Cr_blocks_dezigzag = [reverse_stepSeven(block) for block in Cr_zz_decoded]

# --- reverse 6.brak ---

# --- reverse 5. Dekwantyzacja ---
Y_blocks_dequant = reverse_stepFive(Y_blocks_dezigzag, QY)
Cb_blocks_dequant = reverse_stepFive(Cb_blocks_dezigzag, QC)
Cr_blocks_dequant = reverse_stepFive(Cr_blocks_dezigzag, QC)

# --- reverse 4.IDCT ---
Y_blocks_idct = reverse_stepFour(Y_blocks_dequant)
Cb_blocks_idct = reverse_stepFour(Cb_blocks_dequant)
Cr_blocks_idct = reverse_stepFour(Cr_blocks_dequant)

# --- reverse 3.Scalanie bloków ---
width_Y = Y_padded.shape[1]
width_C = Cb_padded.shape[1]

Y_channel = reverse_stepThree(Y_blocks_idct, Y_padded.shape[0], Y_padded.shape[1])
Cb_channel = reverse_stepThree(Cb_blocks_idct, Cb_padded.shape[0], Cb_padded.shape[1])
Cr_channel = reverse_stepThree(Cr_blocks_idct, Cr_padded.shape[0], Cr_padded.shape[1])

#Usunięcie wyrównania
Y_channel = remove_padding(Y_channel, Y.shape[0], Y.shape[1])
Cb_channel = remove_padding(Cb_channel, Cb_sub.shape[0], Cb_sub.shape[1])
Cr_channel = remove_padding(Cr_channel, Cr_sub.shape[0], Cr_sub.shape[1])

# --- reverse 2.Nadpróbkowanie ---
Cb_upsampled = reverse_stepTwo(Cb_channel, factor)
Cr_upsampled = reverse_stepTwo(Cr_channel, factor)

#złączenie kanałów
ycbcr_reconstructed = np.stack([Y_channel, Cb_upsampled, Cr_upsampled], axis=2)

# --- reverse 1.Konwersja YCbCr -> RGB ---
rgb_reconstructed = reverse_stepOne(ycbcr_reconstructed)

return rgb_reconstructed

```

Rysunek 26



Rysunek 27

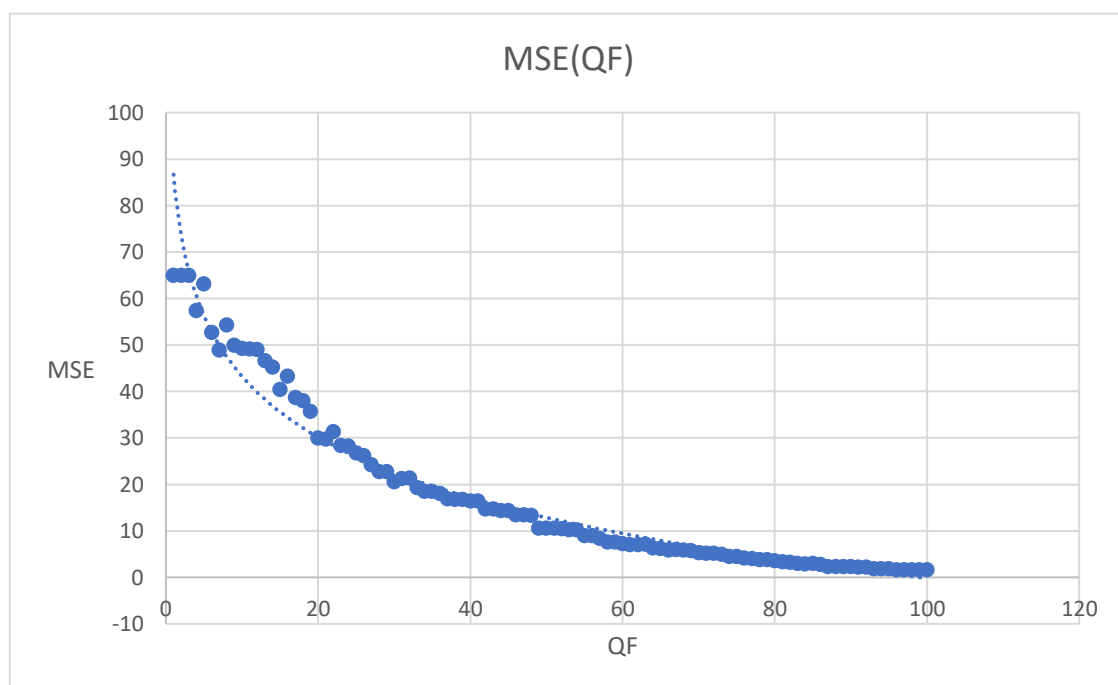


Rysunek 28

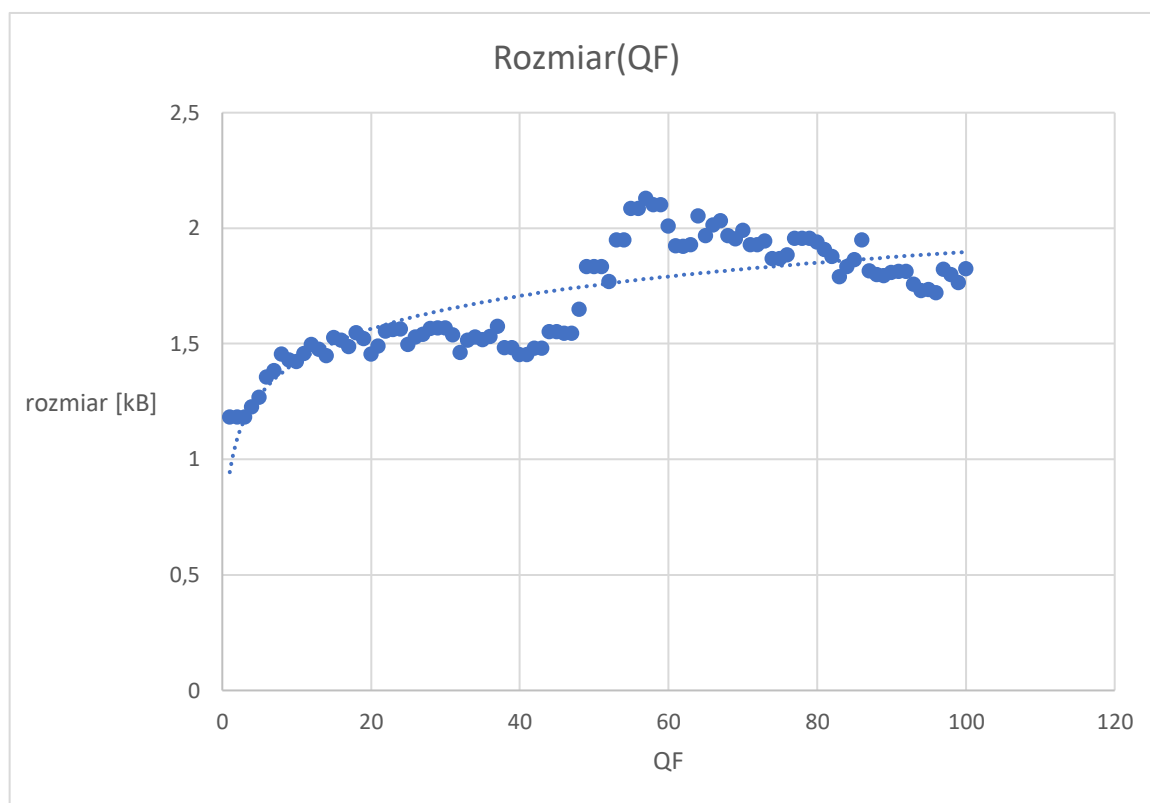
### 2.1.10 Badanie wpływu współczynnika jakości na jakość i rozmiar obrazu

Wykonano serię stu pomiarów dla QF z zakresu 1-100, gdzie mierzono błąd średniokwadratowy między obrazem wyjściowym, a wejściowym oraz rozmiar obrazu wyjściowego.

Wyniki badania dane są wykresami 1 i 2.



Wykres 1



**Wykres 2**

W przypadku zależności błędu średniokwadratowego od współczynnika jakości, widocznie maleje on logarytmicznie wraz ze wzrostem QF. Maksymalna wartość MSE to ok. 65 dla QF=1, a minimalna to 1,62 dla QF=99.

W przypadku zależności rozmiaru od współczynnika jakości, ciężko określić rodzaj zmiany; pojawiają się liczne fluktuacje, z silnie zaznaczonym minimum lokalnym dla QF=40. Maksymalny rozmiar pliku to 2128 bajtów dla QF=57. Minimalny to 1182 bajty dla QF=1-3. Tak więc różnica między maksymalnym, a minimalnym rozmiarem to 946 bajty tj. niecały kilobajt.

### **2.1.11 Wywołanie w programie main.exe**

W programie main.exe, wywołanie kompresji i dekompresji opisywanej w punktach 2.1.1-2.1.9, odbywa się w sposób pokazany na rysunku nr 29.

Parametry kompresji to współczynnik próbkowania równy 2 oraz współczynnik jakości równy 100.

Wywoływane jest także obliczenie błędu średniokwadratowego między obrazem wejściowym i wyjściowym oraz podanie rozmiaru obrazu wyjściowego w bajtach.

A screenshot of a dark-themed terminal window with the title bar 'main.py' and standard window controls (minimize, maximize, close). The window contains a Python script that uses the JPEG module for image processing. The script saves an original image, compresses it with a quality factor of 100, saves the reconstructed image, calculates the Mean Squared Error (MSE) between the original and reconstructed images, and prints the file size of the reconstructed image. It also includes print statements to identify the original and reconstructed images and to show them side-by-side.

```
main.py

JPEG.save_image(JPEG.colors(), "original_image.png")
reconstructed_image = JPEG.process(2, 100)
JPEG.save_image(reconstructed_image, "reconstructed_image.png")

JPEG.process_mse("original_image.png", "reconstructed_image.png")
size = os.path.getsize('reconstructed_image.png')
print(f"Rozmiar pliku na dysku: {size} bajtów")
print("Oryginalny obraz: original_image.png ")
print("Obraz po kompresji i dekompresji: reconstructed_image.png")
JPEG.show_image("original_image.png")
JPEG.show_image("reconstructed_image.png")
```

Rysunek 29

## 2.2 Ukrycie wiadomości w obrazie

W tym zadaniu należało ukryć wiadomość tekstową w dowolnym obrazie korzystając z funkcji określonych w instrukcji. Funkcje pochodzące z instrukcji zawarte są w pliku steganography.py. Funkcje te korzystają z bibliotek numpy – do obliczeń, binascii – do dekodowania ciągów binarnych na znaki ASCII, openCV – do obsługi i manipulacji obrazami oraz math – do wykorzystywania funkcji całkowitoliczbowych (sufit). Dodatkowo pierwotnie wykorzystywana była także biblioteka matplotlib do zapisu obrazu, lecz ze względu na jej problemy z dołączaniem jej przy tworzeniu pliku wykonywalnego, została ona zamieniona na Pillow.



```
lab3.py

def hide(img_name, msg, nLSB):
    original_image = steganography.load_image(img_name)
    message = msg
    message = steganography.encode_as_binary_array(message)

    image_with_message = steganography.hide_message(original_image, message, nLSB)
    steganography.save_image(img_name+"_with_message_"+str(nLSB)+".png", image_with_message)

def show(img_name, nLSB, length):

    image_with_message_png = steganography.load_image(img_name)
    secret_message_png =
    steganography.decode_from_binary_array(steganography.reveal_message(
        image_with_message_png, nbits=nLSB, length=length))
    print(secret_message_png)
```

Rysunek 30

W celu ukrycia i odkrycia wiadomości tekstowej w obrazku, zaimplementowano funkcje hide() oraz show() (rysunek 30), w pliku lab3.py.

Funkcja hide służy do ukrycia wiadomości msg, w pliku zadanym nazwą img\_name na nLSB najmniej znaczących bitach.

Korzystamy z funkcji ze steganography.py: wpierw ładujemy obraz, a następnie kodujemy wiadomość binarnie. Następnie ukrywamy obraz i zapisujemy go, ze stosowną nazwą.

Funkcja show(), również korzystając z funkcji ze steganography.py, ładuje obraz z wiadomością, a następnie odkrywa i dekoduje ją, wiedząc jaka jest jej długość oraz na ilu najmniej znaczących bitach została zapisana. Następnie wyświetla zdekodowaną wiadomość.

Ukrycie i odkrycie wiadomości odbywa się w pliku main.py (który koresponduje z plikiem main.exe). Operacja ukrycia i odkrycia wiadomości w main.py dana jest rysunkiem nr 31.



```
main.py

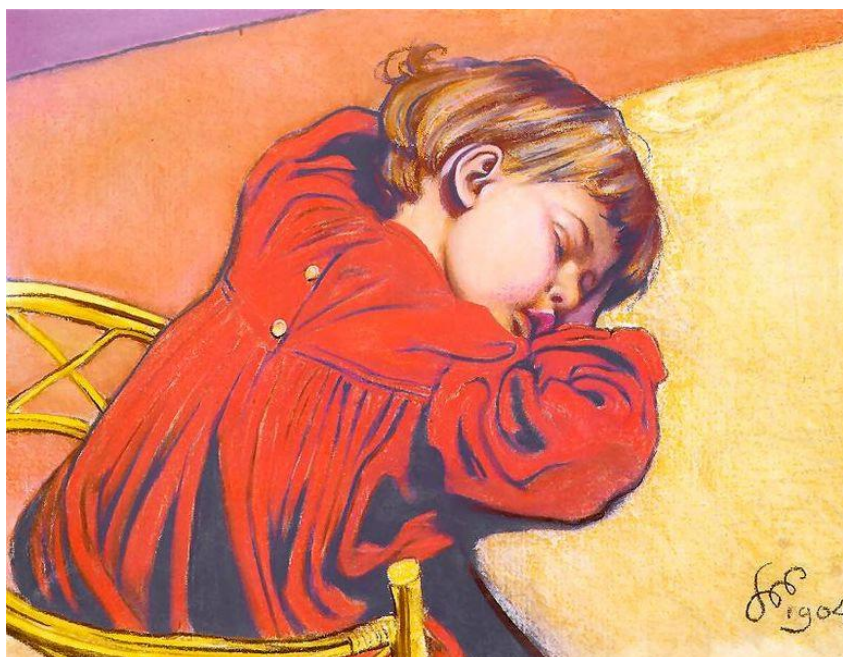
nLSB=2
msg = "Stanisław Wyspiański, Macierzyństwo, 1905"
binary=steganography.encode_as_binary_array(msg)
length=len(binary)
print("Liczba najmniej znaczących bitów użyta do ukrycia wiadomości: ", nLSB)
lab3.hide("img1.png",msg, nLSB)
print("Wiadomość ukryto w obrazie: img1.png_with_message_"+str(nLSB)+".png")
lab3.show("img1.png_with_message_"+str(nLSB)+".png",nLSB,length)
```

Rysunek 31

Liczba najmniej znaczących bitów służących do ukrycia wiadomości to 2. Ukrywana wiadomość to „Stanisław Wyspiański, Macierzyństwo, 1905”, co jest też tytułem obrazu, w którym ukrywamy wiadomość (plik img1.png). Używamy tu także steganography.py to zakodowania wiadomości i obliczenia jej długości, ponieważ długość podawana w funkcji show oznacza ilość bitów. W przypadku angielskich znaków moglibyśmy po prostu wymnażać długość wiadomości przez 7 (lub 8, w zależności od typu zapisu), lecz w przypadku polskich znaków, należałoby je zliczać i mnożyć przez 16 (UTF-8), gdyż polskie znaki nie są kodowane w sposób standardowy. Tak więc łatwiej jest zakodować wiadomość i obliczyć jej długość.

Program wyświetla nazwę pliku, w którym ukryta jest wiadomość, a następnie ją odkrywa i wyświetla na ekranie.

Wyniki działania programu pokazują rysunki 32 i 33.



Rysunek 32

```
Liczba najmniej znaczących bitów użyta do ukrycia wiadomości: 2
Wiadomość ukryto w obrazie: img1.png_with_message_2.png
Stanisław Wyspiański, Macierzyństwo, 1905
```

Rysunek 33

Wywołanie tej operacji odpowiada wybraniu opcji nr 2 w programie main.exe.

### 2.3 Badanie wpływu liczby użytych najmniej znaczących bitów na jakość obrazka

Kolejne zadanie polega na modyfikacji pierwszego, tak aby zbadać wpływ liczby użytych najmniej znaczących bitów do ukrycia wiadomości na obraz wyjściowy. W tym celu skorzystano z tych samych funkcji co w zadaniu pierwszym, z tą różnicą, że wywoływano je w pętli dla zakresu używanych LSB od 1 do 8. Następnie dla każdego z obrazów obliczono błąd średniokwadratowy względem oryginału. Funkcja obliczająca błąd średniokwadratowy pochodzi z laboratorium nr 1 (rysunek 34).

```
lab3.py

def calculate_mse(image1, image2):
    diff = image1 - image2
    mse = np.mean(np.square(diff))
    return mse

def file_to_array(name):
    img = Image.open(name).convert("RGB")
    img_array = np.array(img)
    return img_array

def process_mse(name1, name2):
    original_image = file_to_array(name1)
    transmitted_image = file_to_array(name2)
    mse = calculate_mse(original_image, transmitted_image)
    print(f"Mean Squared Error (MSE): {mse}")
```

Rysunek 34

Wywołania funkcji w pliku main.py dane są rysunkiem nr 35. Jest to także procedura wykonywana po wyborze opcji nr 3 w programie main.exe.

Co ważne, pierwotną wiadomość pomnożono przez 4000, aby zgodnie z poleceniem zajmowała więcej niż 75% obrazu dla  $n\text{LSB}=1$ . W takim wypadku pojemność obrazu to 1 402 200 bity ( $\text{liczba pikseli} \cdot \text{liczba kanałów} \cdot n\text{LSB}$ ), a rozmiar wiadomości to 1 376 000 bity. Tak więc wiadomość zajmuje ponad 98% dostępnego miejsca.

```
main.py

nLSB = 1
msg = "Stanisław Wyspiański, Macierzyństwo,1905"
msg = msg * 4000
binary = steganography.encode_as_binary_array(msg)
length = len(binary)

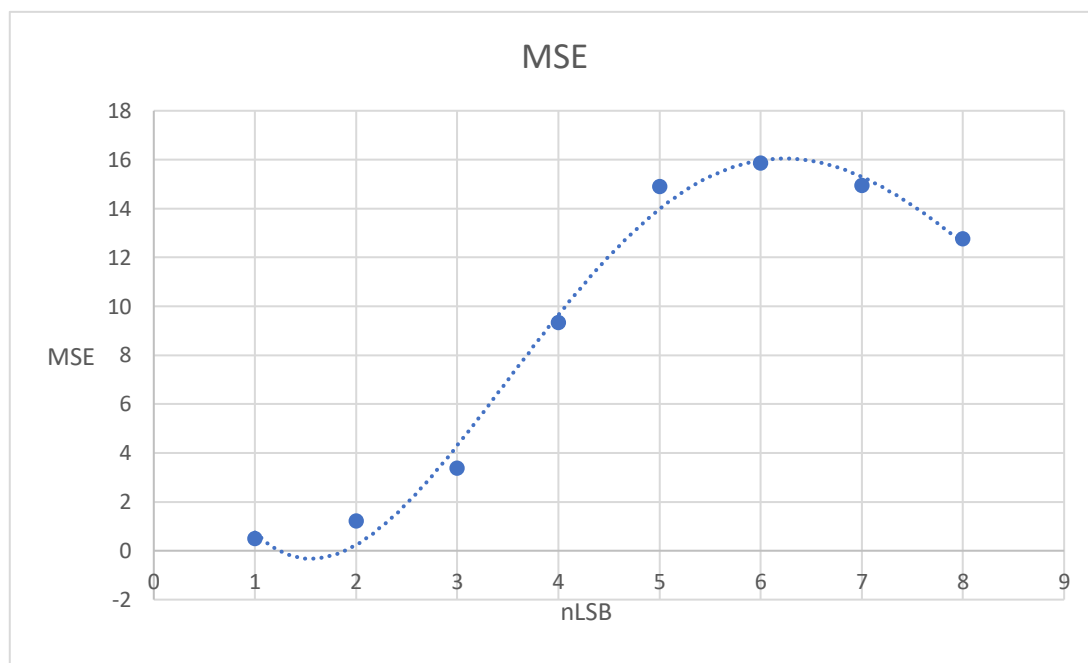
for i in range(8):
    lab3.hide("img1.png", msg, nLSB)
    lab3.show("img1.png_with_message_" + str(nLSB) + ".png", nLSB, length)

    nLSB = nLSB + 1

for i in range(8):
    print("Plik: img1.png_with_message_" + str(i+1) + ".png")
    print("n=", i + 1)
    lab3.process_mse("img1.png", "img1.png_with_message_" + str(i + 1) + ".png")
```

Rysunek 35

Wyniki pomiarów błędu średniokwadratowego w zależności od liczby użytych najmniej znaczących bitów prezentuje wykres nr 3.

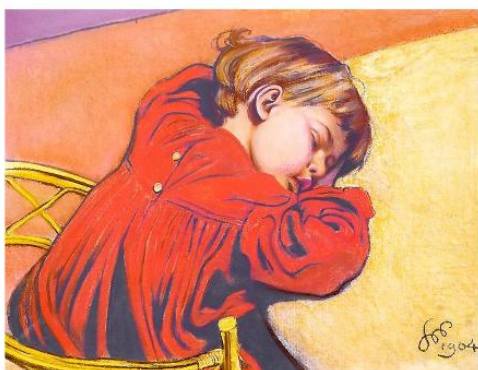


Wykres 3

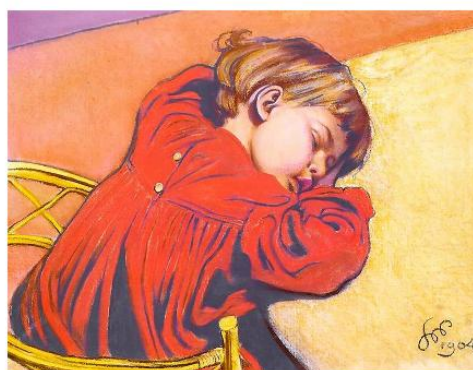
Porównanie wygenerowanych obrazów z ukrytą wiadomością dane jest rysunkiem nr 36.



nLSB=1



nLSB=2



nLSB=3



nLSB=4



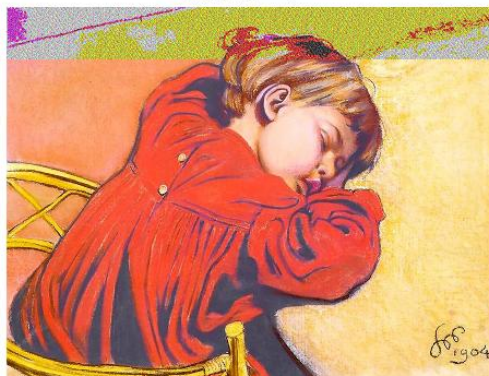
nLSB=5



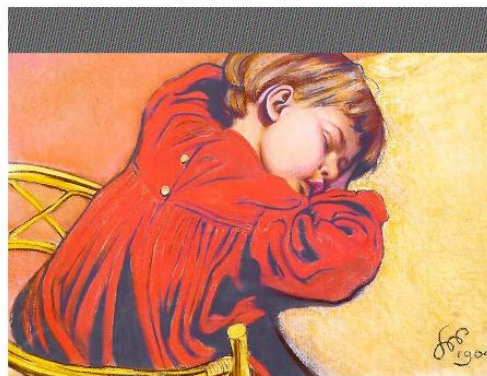
nLSB=6



nLSB=7



nLSB=8



Rysunek 36

Na rysunku nr 36 widzimy widoczne zniekształcenia od nLSB większego od 4. W przypadku nLSB=8 obserwujemy całkowite zniekształcenie górnej części obrazu.

Patrzac jednak na wykres, obserwujemy maksymalną wartość błędu przy nLSB=6. Następnie błąd maleje. Wynika to wprost z ważnej właściwości obliczanego przez nas błędu, mianowicie – uśrednianie. Jak sama nazwa wskazuje, błąd średniokwadratowy oblicza średnią z poszczególnych kwadratów różnic. Mimo poważnych zniekształceń na dwóch ostatnich obrazach, będzie on zatem niższy, ponieważ zniekształcenie to jest skumulowane na małym obszarze. W poprzednich przypadkach natomiast ukryta wiadomość jest rozsiłana po większej części obrazu, stąd więcej „przekłamanych” bitów. Szósty obrazek to punkt, w którym osiągnięte zostaje optimum maksymalizacji błędu: stosunek wielkości pojedynczego błędu do zajętego obszaru daje najwyższy błąd.

## **2.4 Zapis wiadomości od zadanej pozycji**

W tym zadaniu należało zmodyfikować funkcje `hide_message()` oraz `reveal_message()`, w taki sposób, aby ukrywały wiadomość od określonej pozycji. W tym celu w pliku `steganography.py` utworzono funkcje `hide_messagepos()` oraz `reveal_messagepos()`. Funkcje te dane są rysunkiem nr 37. W pliku `lab3.py` zostały utworzone funkcje `hide_pos()` oraz `show_pos`, które są identyczne jak funkcje `hide()` i `show()`, z tą różnicą, że wywołują funkcje `hide_messagepos()` oraz `reveal_messagepos()`, zamiast `hide_message()` i `reveal_message()`.

Na rysunku 38 wylistowano zmiany względem pierwotnych funkcji. W większości przypadków jest to najzwyczajniejsza zmiana indeksacji. Najważniejszą zmianą w używaniu funkcji jest to, że możemy określić teraz od której pozycji zaczynamy ukrywać wiadomość.

Wywołanie odpowiednich funkcji w pliku `main.py` dane jest rysunkiem 39. Są to kroki analogiczne do zadania nr 2, z tą różnicą, że ukrywanie wiadomości zaczyna się od 2025 pozycji. Kroki te wykonywane są przy wybraniu opcji nr 4 w programie `main.exe`.

```

def hide_messagepos(image, message, nbits=1, spos=0):

    nbits = clamp(nbits, 1, 8)
    shape = image.shape
    image = np.copy(image).flatten()
    available_bits = (len(image) - spos) * nbits
    if len(message) > available_bits:
        raise ValueError("Message is too long for the given starting position")

    chunks = [message[i:i + nbits] for i in range(0, len(message), nbits)]

    for i, chunk in enumerate(chunks):
        byte = "{:08b}".format(image[spos + i])
        new_byte = byte[:-nbits] + chunk
        image[spos + i] = int(new_byte, 2)
    return image.reshape(shape)

def reveal_messagepos(image, nbits=1, length=0, spos=0):

    nbits = clamp(nbits, 1, 8)
    shape = image.shape
    image = np.copy(image).flatten()
    available_bits = (len(image) - spos) * nbits

    if length <= 0:
        length = available_bits
    else:
        length = min(length, available_bits)

    length_in_pixels = math.ceil(length / nbits)
    message = ""
    i = 0
    while i < length_in_pixels and (spos + i) < len(image):
        byte = "{:08b}".format(image[spos + i])
        message += byte[-nbits:]
        i += 1

    if length > 0:
        message = message[:length]
    return message

```

Rysunek 37



```
steganography.py

def 1
available_bits = (len(image) - spos) * nbits # <-- -spos

def 2
for i, chunk in enumerate(chunks):
    byte = "{:08b}".format(image[spos + i]) # <-- spos+i
    new_byte = byte[:-nbits] + chunk
    image[spos + i] = int(new_byte, 2)

def 3
while i < length_in_pixels and (spos + i) < len(image):
    byte = "{:08b}".format(image[spos + i]) # <-- spos+i
    message += byte[-nbits:]
    i += 1

def 4
if length <= 0:
    length = available_bits
else:
    length = min(length, available_bits)
```

Rysunek 38

```
main.py

nLSB=2
spos=2025
msg = "Stanisław Wyspiański, Macierzyństwo,1905"
binary=steganography.encode_as_binary_array(msg)
length=len(binary)
print("Liczba najmniej znaczących bitów użyta do ukrycia wiadomości: ", nLSB)
lab3.hide_pos("img1.png",msg, nLSB,spos)
print("Wiadomość ukryto w obrazie: img1.png_with_message_"+str(nLSB)+".png")

lab3.show_pos("img1.png_with_message_pos_"+str(spos)+"_"+str(nLSB)+".png",nLSB,length,spos)
```

Rysunek 39

## 2.5 Ukrywanie obrazu w obrazie

Kolejnym zadaniem, było zaimplementowanie funkcji, która odzyskuje obraz ukryty w innym obrazie, za pomocą funkcji `hide_image()`.

Wpierw w pliku `lab3.py` zaimplementowano funkcję ukrywającą obraz w obrazie, korzystającą z funkcji `hide_image()`. Funkcja ta dana jest rysunkiem nr 40.

```
def image_in_image(name1, name2, nLSB):
    image = steganography.load_image(name1)
    image_with_secret, length_of_secret = steganography.hide_image(image, name2, nLSB)
    steganography.save_image(name2 + "_in_" + name1 + str(nLSB) + ".png", image_with_secret)
    return image_with_secret, length_of_secret
```

Rysunek 40

Funkcja ta ładuje obraz będący nośnikiem, ukrywa w nim drugi obraz, zapisuje go ze stosowną nazwą oraz zwraca ten obraz i długość zakodowanej wiadomości (w naszym przypadku obrazu).

Kolejnym krokiem było stworzenie funkcji analogicznej do `reveal_message()`, dla obrazów. Zaimplementowano zatem funkcję `reveal_image()` daną rysunkiem nr 41.

```
def reveal_image(image, length, nbits=1):

    binary_data = reveal_message(image, nbits=nbits, length=length)

    bytes_data = []
    for i in range(0, len(binary_data), 8):
        byte = binary_data[i:i + 8]
        if len(byte) < 8:
            byte = byte.ljust(8, '0')
        bytes_data.append(int(byte, 2))

    with open("recovered_secret.png", "wb") as f:
        f.write(bytes(bytes_data))

    print("Image recovered successfully!")
    return bytes(bytes_data)
```

Rysunek 41

Na początku odczytujemy odkryte bity za pomocą funkcji `reveal_message()`. Następnie otrzymany ciąg bitów dzielimy na bajty. Jeśli na końcu brakuje bitów, dopełniamy bajt zerami. Tak otrzymane bajty konwertowane są do liczb całkowitych. Otrzymujemy listę zwykłych liczb całkowitych. Następnie otwieramy plik `recovered_secret.png` w trybie zapisu binarnego i zapisujemy otrzymaną listę w postaci danych binarnych. Funkcja także zwraca te dane binarne. Jako, że ukryta została cała struktura pliku PNG, nie musimy się martwić żadną kompresją jak w laboratorium 1. Funkcja ta równie dobrze zatem mogłaby ukryć plik JPEG w pliku PNG (lecz nie na odwrót!).

Ostatnim krokiem było zaimplementowanie `show_image_in_image()` w pliku `lab3.py` danej rysunkiem 42, wywołująca funkcję `reveal_image()`.



```
lab3.py

def show_image_in_image(img, length, nLSB):
    steganography.reveal_image(img, length, nLSB)
```

Rysunek 42

Ukrycie i odkrycie obrazu w pliku main.py dane jest rysunkiem 43.

```
main.py

nLSB=2
image_with_secret, length_of_secret=lab3.image_in_image("img1.png", "img2.png", nLSB)
lab3.show_image_in_image(image_with_secret, length_of_secret, nLSB)
```

Rysunek 43

Operacje te wykonywane są przy wybraniu opcji nr 5 w programie main.exe.

## 2.6 Odkrywanie obrazu bez przekazywania długości wiadomości

Ostatnim zadaniem było utworzenie funkcji, umożliwiających ukrywanie i odkrywanie obrazu, w taki sposób, aby nie było potrzeby przekazywania długości wiadomości jako parametru funkcji. W tym celu należy przy dekodowaniu wykrywać stopkę pliku, w naszym przypadku PNG.

W pliku steganography.py zaimplementowano funkcję `reveal_bytes_until_footer()` daną rysunkiem nr 44. Jest to zmodyfikowana funkcja `reveal_image()`. Po pierwsze, argumentem staje się nazwa obrazka z wiadomością – jego wczytanie następuje wewnątrz funkcji. Po drugie, skopiowano tu kod z funkcji `hide_image()`, aby określić dokładną długość ciągu bitów obrazka, zgodnie z kodowaniem przy ukrywaniu. Następnie odkrywamy wiadomość za pomocą funkcji `reveal_message()`, tak jakby była ona ukryta w całości obrazka.

Finalnie dokonujemy zmiany w podziale ciągu bitów na bajty – zatrzymujemy pętlę, jeśli zostanie wykryta stopka pliku PNG (lub inna podana jako argument). Taki ciąg bajtów jest zapisywany jako obraz PNG.

W ten sposób otrzymaliśmy funkcję, która nie potrzebuje długości wiadomości jako parametru.

W pliku lab3.py znajduje się funkcja `reveal_image_with_0_save()` (rysunek 45), która wywołuje funkcję z rysunku nr 44 i wypisuje nazwę pliku wyjściowego.

```

steganography.py

def reveal_image_until_footer(image_name, nbits=1, footer=b'\x00\x00\x00\x00IEND\xaeB'\x82'):
    stego_image = load_image(image_name)

    with open(image_name, "rb") as file:
        secret_img = file.read()
    secret_img_hex = secret_img.hex()
    secret_img_binary = ["{:08b}".format(int(el, base=16)) for el in
                        [secret_img_hex[i:i + 2] for i in range(0, len(secret_img_hex), 2)]]
    secret_img_binary = "".join(secret_img_binary)

    binary_data = reveal_message(stego_image, nbits=nbits, length=len(secret_img_binary))

    bytes_data = []
    for i in range(0, len(binary_data), 8):
        byte = binary_data[i:i + 8]
        if len(byte) < 8:
            byte = byte.ljust(8, '0')
        bytes_data.append(int(byte, 2))

    if len(bytes_data) >= 12:
        f=bytes(bytes_data[-12:])
        if f == footer:

            print("Found footer!")
            break

    with open("recovered_secret0.png", "wb") as f:
        f.write(bytes(bytes_data))

    print("Image recovered successfully!")
    return bytes(bytes_data)

```

Rysunek 44

```

lab3.py

def reveal_image_with_0_save(stego_image_name, nLSB, output_filename):
    steganography.reveal_image_until_footer(stego_image_name, nLSB)

    print(f"Odkryty obrazek: {output_filename}")

```

Rysunek 45

W pliku main.py wywołane zostaje ukrycie obrazka za pomocą funkcji `hide_image()`, a następnie odkrycie jej przez funkcję `reveal_image_with_0_save()`.

Wywołania tych funkcji wraz z parametrami dane są rysunkiem nr 46.



```
main.py

nLSB = 4
name1="img1.png"
name2="img2.png"
image_with_secret= lab3.image_in_image(name1, name2, nLSB)
lab3.reveal_image_with_0_save(name2 + "_in_"+name1 + str(nLSB) + ".png", nLSB,
"recovered_img2_with0.png")
```

Rysunek 46

Operacje te zostaną wykonane przy wybraniu opcji nr 6 w programie main.exe.

## **Źródła**

<https://www.w3.org/Graphics/JPEG/itu-t81.pdf>

<https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python/>

<https://www.ijg.org/files/T-REC-T.871-201105-I!!PDF-E.pdf>

<https://datko.pl/IOb/lab5.zip>

<https://www.techtarget.com/searchsecurity/definition/steganography>

<https://www.ia.pw.edu.pl/~jurek/js/kody/>