

# Inżynieria obrazów

Implementacja formatów graficznych

Marcin Lasak 272886

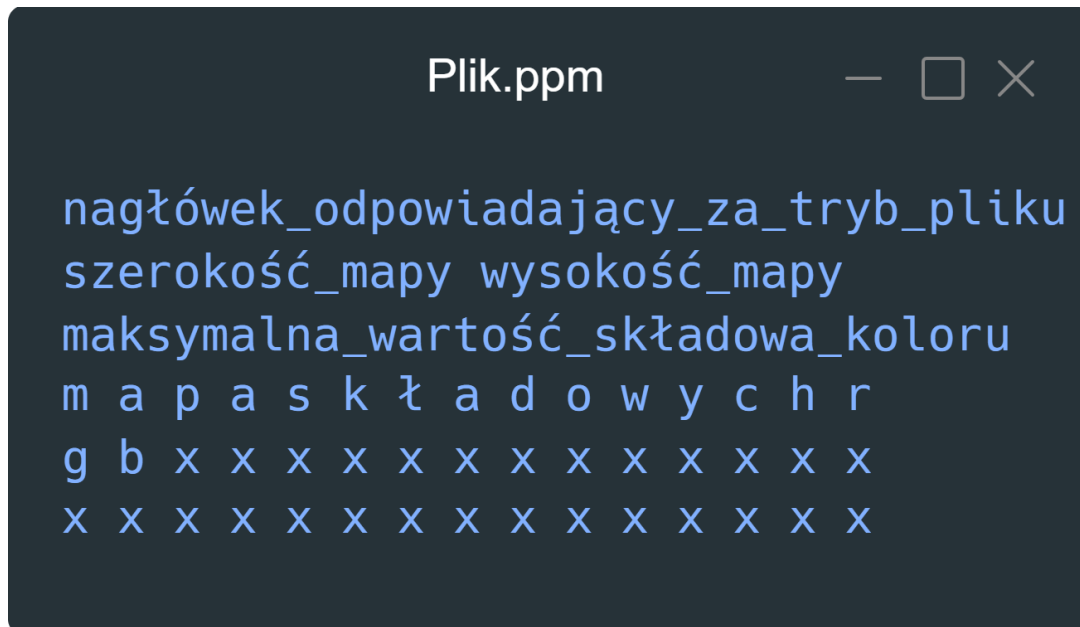
## Spis treści

1. Wstęp teoretyczny .....	3
1.1 PPM .....	3
1.2 PNG .....	3
1.3 JPEG .....	4
2. Zadania .....	5
2.1 Obsługa formatu PPM .....	5
2.2 Spektrum przejść w przestrzeni RGB .....	9
2.3 Zapis formatu PNG .....	12
2.4 Algorytm JPEG cz. I .....	15
3. Pliki źródłowe .....	24
Źródła .....	25

## 1. Wstęp teoretyczny

### 1.1 PPM

PPM (Portable **P**ixel**M**ap) to format zapisu grafiki rastrowej należący do tzw. PNM (Portable **a**n**y**Map). Wśród trzech formatów należących do tej grupy, PPM jest formatem zapisu obrazów kolorowych. Pozostałe dwa formaty służą do zapisu obrazu w odcieniach szarości bądź czerni i bieli. Plik PPM posiada uporządkowaną strukturę daną rysunkiem nr 1<sup>1</sup>.



Rysunek 1

Tryb pliku oznacza sposób zapisu; może być tekstowy(P3), lub binarny(P6). Szerokość i wysokość to rozmiary w pikselach, a mapa składowych to trójki współrzędnych przestrzeni RGB dla poszczególnych pikseli.

### 1.2 PNG

PNG (Portable Network Graphics) to format zapisu grafiki rastrowej oraz rodzaj bezstratnej kompresji plików graficznych. Obsługuje stopniowaną przezroczystość (tzw. kanał alfa) oraz 48-bitową głębię kolorów. Format ten jest dużo bardziej złożony w swojej strukturze niż PPM, lecz pliki w tym formacie mają znacznie mniejszy rozmiar. Plik PNG składa się z:

- Sygnatury
- Nagłówka
- Danych obrazu
- Zakończenia pliku

Sygnatura to podpis identyfikujący plik jako format PNG. Następnie części pliku to tzw. fragmenty (chunks): IHDR (koniecznie pierwszy), PLTE, IDAT oraz IEND(koniecznie ostatni)<sup>2</sup>.

<sup>1</sup> <https://netpbm.sourceforge.net/doc/ppm.html>

<sup>2</sup> <https://www.w3.org/TR/PNG-Chunks.html>

Fragment IHDR (nagłówek) zawiera szerokość, wysokość, głębokość bitową, typ koloru, metodę kompresji, metodę filtracji oraz metodę przeplatania.

Fragment PLTE odpowiada za paletę kolorów pliku PNG. Odpowiada za mapowanie wartości (indeksów) na konkretne kolory RGB.

Fragment IDAT (dane obrazu) zawiera właściwe dane obrazu (piksele). To właśnie tutaj zapisywany jest wynik kompresji obrazu wejściowego.

### 1.3 JPEG

JPEG (**J**oint **P**hotographic **E**xperts **G**roup) 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 z przestrzeni RGB do przestrzeni YCbCr
2. Podpróbkiowanie chrominancji
3. Podział obrazu na bloki 8x8
4. Wykonanie dyskretnej transformaty kosinusowej
5. Podzielenie każdego bloku obrazu przez macierz kwantyzacji
6. Zaokrąglenie wartości w każdym bloku do liczb całkowitych
7. Porządkowanie współczynników DCT w sposób liniowy poprzez zygzakowanie
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 próbkiować 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 transformaty 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.

## 2. Zadania

### 2.1 Obsługa formatu PPM

Celem tego zadania było zaimplementowanie obsługi odczytu i zapisu plików PPM. W tym celu napisano krótki program w języku Python. Program korzysta z paczki numpy, do przetwarzania obrazów oraz Pillow do szybkiego wgrywania i wyświetlania obrazów. Dodatkowo zaimportowano moduł os aby wyświetlać rozmiary wygenerowanych plików (rysunek 2).



```
lab2zad1.py

import numpy as np
import os
from PIL import Image
```

Rysunek 2

Zadanie rozpoczęto od zaimplementowania odczytu plików PPM. W tym celu stworzono funkcję read\_ppm() daną rysunkiem 3.



```
lab2zad1.py

def read_ppm(filename):
    with open(filename, "rb") as f:
        header = f.readline().decode().strip()
        width, height = map(int, f.readline().decode().split())
        max_color = int(f.readline().decode().strip())

        if max_color > 255:
            raise ValueError("Błędna głębokość bitowa! To nie jest plik PPM!")

        #dla p3
        if header == "P3":
            data = []
            for line in f:
                data.extend(map(int, line.decode().split()))
            image_array = np.array(data, dtype=np.uint8).reshape((height, width, 3))
        #dla p6
        elif header == "P6":
            data = np.frombuffer(f.read(), dtype=np.uint8)
            image_array = data.reshape((height, width, 3))
        else:
            raise ValueError("Błędny nagłówek!")

    return image_array
```

Rysunek 3

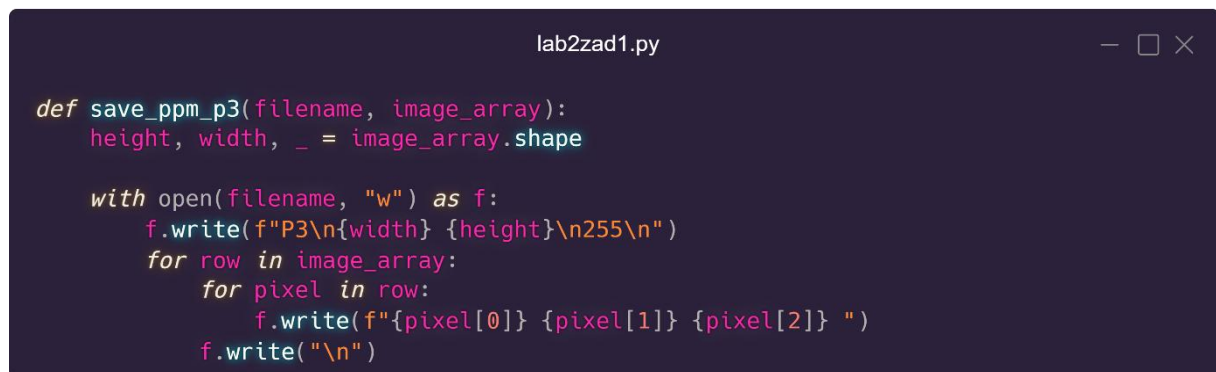
Funkcja ta działa w zarówno dla plików tekstowych jak i binarnych domyślne otworzenie pliku w trybie binarnym i późniejsze ewentualne dekodowanie. Po pobraniu trzech pierwszych linii pliku (oraz ich zdekodowaniu w przypadku typu tekstowego), sprawdzamy głębokość bitową (linia 3 pliku PPM), a następnie przechodzimy do rozróżnienia typu pliku. W przypadku typu P3, dane wczytywane są jako tekst i dekodowane. Takie dane zapisywane są w wynikowej tablicy numpy.

W przypadku P6 dane wczytywane są od razu jako bajty, a następnie analogicznie zapisywane w tablicy numpy.

Funkcja zwraca tak utworzoną tablicę numpy, na której możemy później swobodnie pracować.

Następnie zaimplementowane zostały funkcje zapisu formatu PPM. W tym wypadku jednak podzielono zapis typu tekstowego i binarnego na dwie osobne funkcje.

Rozpoczęto od funkcji `save_ppm_p3()` zapisującej tablicę numpy w PPM w trybie tekstowym (rysunek 4).

A screenshot of a code editor window titled 'lab2zad1.py'. The code defines a function `save_ppm_p3(filename, image_array):`. It first gets the dimensions of the image array: `height, width, _ = image_array.shape`. Then it opens the file in write mode ('w') and writes the PPM header: `f.write(f"P3\n{width} {height}\n255\n")`. It then iterates over each row and pixel, writing the RGB values: `f.write(f"{pixel[0]} {pixel[1]} {pixel[2]} ")` and a newline character `f.write("\n")` after each row.

```
def save_ppm_p3(filename, image_array):
    height, width, _ = image_array.shape

    with open(filename, "w") as f:
        f.write(f"P3\n{width} {height}\n255\n")
        for row in image_array:
            for pixel in row:
                f.write(f"{pixel[0]} {pixel[1]} {pixel[2]} ")
            f.write("\n")
```

Rysunek 4

Funkcja ta otwiera plik w trybie zapisu, zapisuje nagłówek pliku PPM (tryb, rozmiar, maks. wartość koloru), a następnie zapisuje trójki współrzędnych RGB dla każdego piksela.

Zapis w trybie binarnym został zaimplementowany w funkcji `save_ppm_p6()`, bliźniaczo podobnej do funkcji dla trybu tekstowego. Różni się zastąpieniem pętli prostą konwersją do tablicy binarnej za pomocą narzędzi numpy (rysunek 5).

A screenshot of a code editor window titled 'lab2zad1.py'. The code defines a function `save_ppm_p6(filename, image_array):`. It first gets the dimensions of the image array: `height, width, _ = image_array.shape`. Then it opens the file in write mode ('wb') and writes the PPM header: `f.write(f"P6\n{width} {height}\n255\n".encode())`. Finally, it writes the image data as bytes: `f.write(image_array.tobytes())`.

```
def save_ppm_p6(filename, image_array):
    height, width, _ = image_array.shape
    with open(filename, "wb") as f:
        f.write(f"P6\n{width} {height}\n255\n".encode())
        f.write(image_array.tobytes())
```

Rysunek 5

W celu przetestowania zaimplementowanych funkcjonalności stworzono kilka funkcji pomocniczych. Dane są one rysunkiem nr 6.

Testowanie programu polegało na wywołaniu kolejno funkcji jak na rysunku nr 7.

Skompilowany program w postaci pliku wykonywalnego `lab2zad1.exe` wykonuje dokładnie te operacje.

```

lab2zad1.py

#odczyt pliku jpg/png do numpy array
def read_image(filename):

    image = Image.open(filename)
    image_array = np.array(image)
    return image_array

#konwersja png/jpg do ppm p3
def save_as_ppm_p3(filename):

    save_ppm_p3(filename+".ppm", read_image(filename))

#konwersja png/jpg do ppm p6
def save_as_ppm_p6(filename):
    save_ppm_p6(filename+".ppm", read_image(filename))
#wyświetlenie obrazu
def show_image(filename):
    image = Image.open(filename)
    image.show()

```

Rysunek 6

```

lab2zad1.py

#testy
#1.zapis do ppm p3
save_as_ppm_p3("photo.jpg")
#2.zapis do ppm p6
save_as_ppm_p6("photo1.jpg")

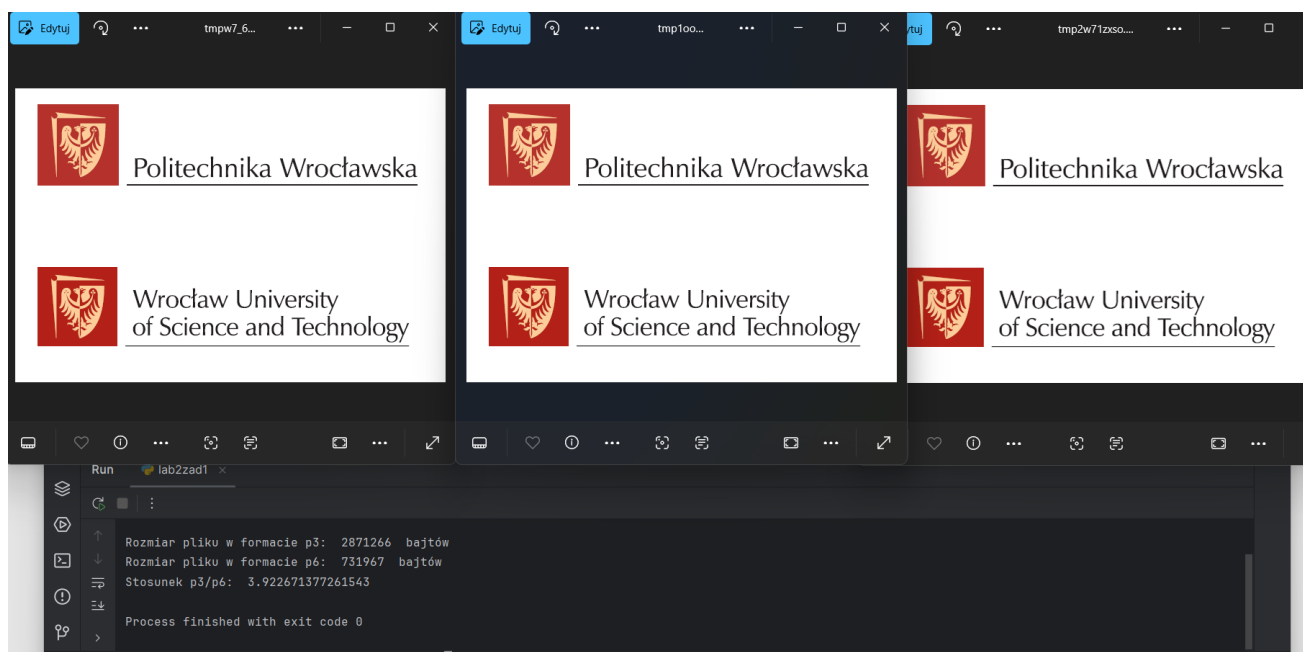
show_image("photo.jpg")
show_image("photo1.jpg")
#3.odczyt ww plików
array1 = read_ppm("photo.jpg.ppm")
array2 = read_ppm("photo1.jpg.ppm")
#4. porównanie rozmiaru
print("\n")
file1 = 'photo.jpg.ppm'
file2 = 'photo1.jpg.ppm'

# Pobranie rozmiarów plików
size1 = os.path.getsize(file1)
size2 = os.path.getsize(file2)
#Wyświetlenie rozmiaru plików
print("Rozmiar pliku w formacie p3: ", size1, " bajtów")
print("Rozmiar pliku w formacie p6: ", size2, " bajtów" )
print("Stosunek p3/p6: ", size1/size2)

```

Rysunek 7

Rysunek nr 8 prezentuje wynik działania programu. Wyświetlone obrazy to kolejno oryginał, PPM P3 i PPM P6. Jak widzimy jest to ten sam obraz. Co ciekawe, wbudowany w system Windows program Zdjęcia, służący do otwierania plików graficznych nie obsługuje plików PPM, więc po wymuszonym ich wyświetleniu przez kod Python, są one automatycznie konwertowane do PNG.



**Rysunek 8**

Widzimy także na rysunku nr 8, że rozmiar pliku w trybie binarnym jest prawie 4 razy mniejszy od pliku w trybie tekstowym.



## 2.2 Spektrum przejść w przestrzeni RGB

Kolejnym zadaniem było wygenerowanie w formacie PPM spektrum przejść wg rysunku nr 11 oraz spektrum przejścia czerni-biel. Program wykonano w języku Python z użyciem bibliotek numpy oraz Pillow (importy jak na rysunku 2 bez os). Wykorzystano także funkcje z poprzedniego zadania: `save_ppm_p3()` oraz `show_image()`. Wpierw wykonano funkcję `gray()` tworzącą tablicę numpy trójek współrzędnych RGB, gdzie każda kolejna trójka w kolumnie jest inkrementowana od 0 do 256 (stąd kolumna nr 1 zawiera piksele o barwie `[0,0,0]`, kolumna nr 2 `[1,1,1]` itd.). W ten sposób otrzymujemy płynne przejście tonalne od czerni do bieli. Funkcja `gray()` dana jest rysunkiem nr 9.

```
lab2zad2.py

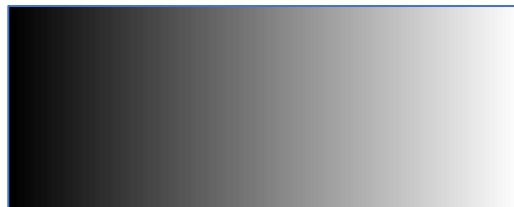
def gray():
    array = np.zeros((100, 256, 3), dtype=np.uint8)

    for i in range(256):
        for j in range(100):
            array[j, i] = [i,i,i]

    save_ppm_p3("gray.ppm", array)
```

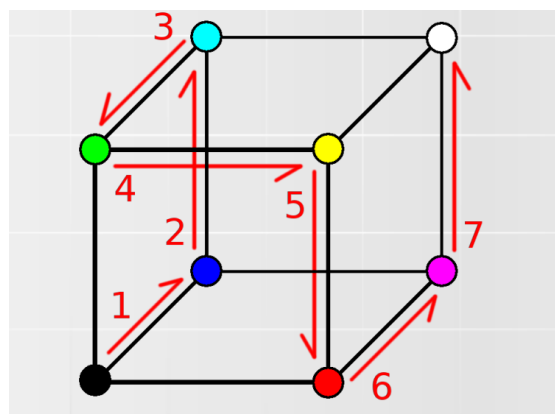
Rysunek 9

Wynik działania funkcji dany jest rysunkiem nr 10. Niebieskie obramowanie zostało dodane w raporcie, aby grafika nie zlewała się z białym tłem papieru.



Rysunek 10

Następnie wykonano funkcję generującą obraz PPM z płynnym przejściem kolorów wg rysunku nr 11.



Rysunek 11

Funkcja `colors()` działa analogicznie do funkcji `gray()`. Najpierw wyznaczono wszystkie skrajne współrzędne RGB w kolejności zgodnej z rysunkiem nr 11. Następnie cyklicznie inkrementowano lub dekrementowano jedną z współrzędnych. Funkcja `colors()` dana jest rysunkiem nr 12.

```
lab2zad2.py

def colors():
    transition_points = [
        (0, 0, 0), # 1 - czarny
        (0, 0, 255), # 2 - niebieski
        (0, 255, 255), # 3 - cyjan
        (0, 255, 0), # 4 - zielony
        (255, 255, 0), # 5 - żółty
        (255, 0, 0), # 6 - czerwony
        (255, 0, 255), # 7 - magenta
        (255, 0, 255), # 8 - biały
    ]

    array = np.zeros((100, 1792, 3), dtype=np.uint8)

    #czarny-niebieski
    for i in range(256):
        for j in range(100):
            array[j, i] = [0,0,i]

    #niebieski-cyjan
    for i in range(256):
        for j in range(100):
            array[j, 256+i] = [0,i,255]

    #cyjan-zielony
    for i in range(256):
        for j in range(100):
            array[j, 2*256+i] = [0,255,255-i]

    #zielony-żółty
    for i in range(256):
        for j in range(100):
            array[j, 3*256+i] = [i,255,0]

    #żółty-czerwony
    for i in range(256):
        for j in range(100):
            array[j, 4*256+i] = [255,255-i,0]

    # czerwony-magenta
    for i in range(256):
        for j in range(100):
            array[j, 5 * 256 + i] = [255, 0, i]

    # magenta-biały
    for i in range(256):
        for j in range(100):
            array[j, 6 * 256 + i] = [255, i, 255]

    save_ppm_p3("colors.ppm", array)
    return array
```

Rysunek 12

Przykładowo przy przejściu czarny-niebieski zaczynamy od współrzędnych [0,0,0] i inkrementujemy współrzędną B.

Program lab2zad2.py generuje i zapisuje kolejno spektrum przejścia między kolorami i przejścia czern-biel, a następnie wyświetla utworzone pliki PPM (rysunek 13).

```
lab2zad2.py

colors()
gray()
show_image("colors.ppm")
show_image("gray.ppm")
```

**Rysunek 13**

Plik wykonywalny utworzony na podstawie kodu Python wykonuje właśnie te działania.

Wynik działania funkcji colors() dany jest rysunkiem nr 14. Niebieskie obramowanie zostało dodane w raporcie ze względów opisanych przy funkcji gray().

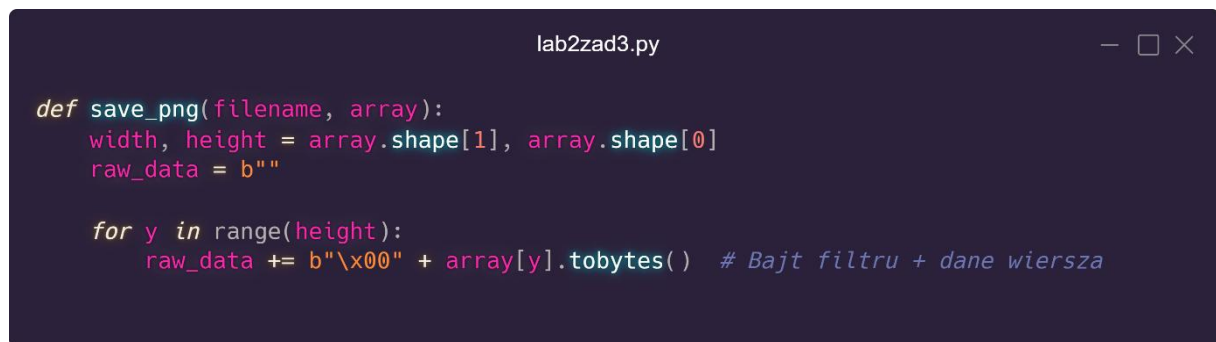


**Rysunek 14**

## 2.3 Zapis formatu PNG

W tym zadaniu należało zaimplementować zapis pliku graficznego w formacie PNG. Zapisywanym obrazem miało być spektrum przejść wygenerowane w zadaniu drugim. Stąd w programie korzystamy z funkcji `colors()`. Importowane biblioteki to tak jak wcześniej `numpy` oraz `Pillow` i dwie biblioteki wykorzystane do utworzenia pliku PNG. Pierwsza z nich to `zlib`, potrzebna do skompresowania danych obrazu, a druga to `struct` wykorzystana do wygenerowania nagłówka.

Zapis formatu PNG zaimplementowano jako funkcję `save_png()`. Jak wspomniano we wstępie teoretycznym, zapis do formatu PNG jest dużo bardziej skomplikowany od formatu PPM. Funkcja `save_png()` w pierwszej kolejności konwertuje każdy wiersz na ciąg bajtów i dodaje na jego początku bajt `\x00`, odpowiedzialny za metodę filtracji (rysunek 15). Taki bajt musi pojawić się w każdym wierszu; przyjmuje on wartości od 0 do 4. W naszym przypadku jest to 0, czyli brak filtracji.



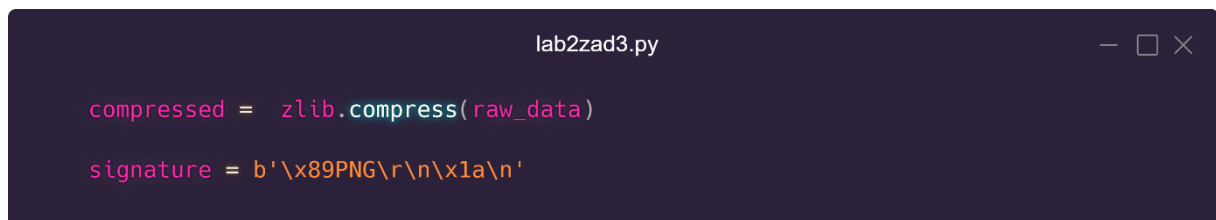
```
lab2zad3.py

def save_png(filename, array):
    width, height = array.shape[1], array.shape[0]
    raw_data = b""

    for y in range(height):
        raw_data += b"\x00" + array[y].tobytes() # Bajt filtru + dane wiersza
```

Rysunek 15

Następnie tablica surowych danych została skompresowana przy pomocy narzędzia `zlib.compress` oraz zdefiniowano sygnaturę tworzonego pliku (rysunek 16).



```
lab2zad3.py

compressed = zlib.compress(raw_data)

signature = b'\x89PNG\r\n\x1a\n'
```

Rysunek 16

Kolejnym krokiem było zdefiniowanie i utworzenie fragmentu IHDR. W tym celu najpierw określono parametry grafiki wg polecenia i wygenerowano nagłówek IHDR (rysunek 17).

Przy generowaniu nagłówka wykorzystano narzędzia `struct.pack` do generacji nagłówka oraz `zlib.crc32` do obliczenia sumy kontrolnej koniecznej do wygenerowania nagłówka.

Następnie przy pomocy narzędzia `struct.pack` (z wykorzystaniem `zlib.crc32` do obliczenia sumy kontrolnej) utworzono fragment IDAT, zawierający właściwe skompresowane dane obrazu (rysunek 18).

```
lab2zad3.py

# chunk IHDR
ihdr_chunk_type = b'IHDR'
bit_depth = 8
color_type = 2
compression_method = 0
filter_method = 0
interlace_method = 0

# dane IHDR,
ihdr_data = struct.pack(">2I5B", width, height, bit_depth, color_type,
compression_method, filter_method, interlace_method)

# Chunk IHDR
ihdr_chunk = struct.pack(">I", len(ihdr_data)) + ihdr_chunk_type + ihdr_data +
struct.pack(">I", zlib.crc32(
ihdr_chunk_type + ihdr_data) & 0xFFFFFFFF)
```

Rysunek 17

```
lab2zad3.py

# Chunk IDAT
idat_chunk = struct.pack(">I", len(compressed)) + idat_chunk_type + compressed +
struct.pack(">I",zlib.crc32(idat_chunk_type + compressed) & 0xFFFFFFFF)
```

Rysunek 18

Analogicznie jak w przypadku fragmentu IDAT utworzono fragment IEND (rysunek 19).

```
lab2zad3.py

# Chunk IEND (zakończenie)
iend_chunk_type = b'IEND'
iend_chunk = struct.pack(">I", 0) + iend_chunk_type + struct.pack(">I",
zlib.crc32(iend_chunk_type) & 0xFFFFFFFF)
```

Rysunek 19

Finalnie otwieramy (tworzymy) plik PNG (otwarcie w trybie zapisu binarnego) i kolejno zapisujemy w nim utworzone fragmenty (chunki) (rysunek 20).

```
lab2zad3.py

# Zapis pliku PNG
with open(filename, "wb") as f:
    f.write(signature)
    f.write(ihdr_chunk)
    f.write(idat_chunk)
    f.write(iend_chunk)

print(f"Obraz zapisany jako {filename}")
```

Rysunek 20

Program lab2zad3.py (oraz korespondujący mu plik wykonywalny lab2zad3.exe) tworzy tablicę współrzędnych RGB przy pomocy funkcji colors(), a następnie zapisuje ją jako plik PNG z użyciem funkcji save\_png() i wyświetla zapisany obraz używając omawianej wcześniej funkcji show\_image() (rysunek 21).

```
lab2zad3.py

array = colors()

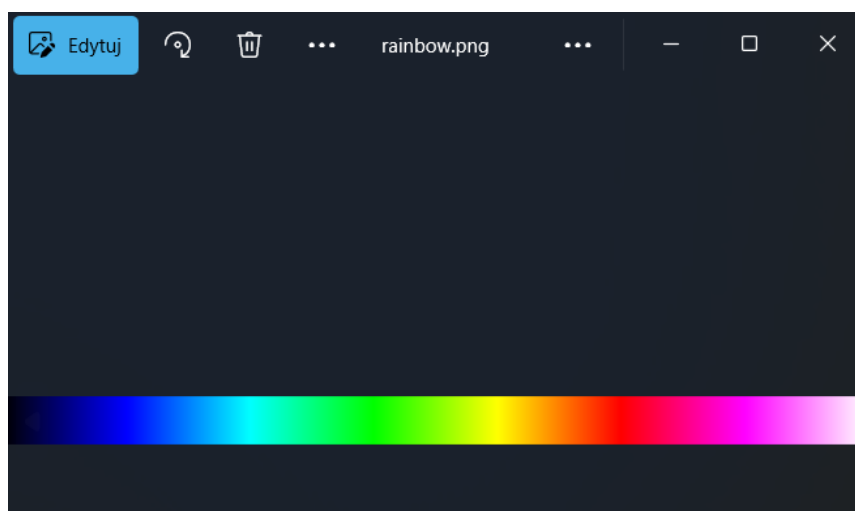
width, height = array.shape[1], array.shape[0]

save_png("rainbow.png", array)

show_image('rainbow.png')
```

Rysunek 21

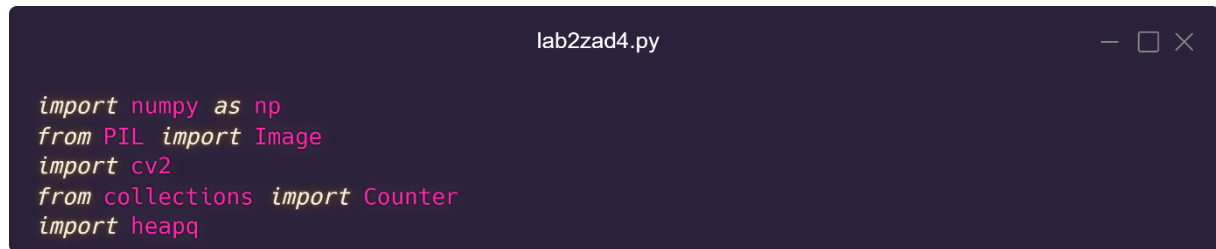
Wynikiem działania programu jest plik rainbow.jpg (rysunek 22).



Rysunek 22

## 2.4 Algorytm JPEG cz. I

W tym zadaniu należało zaimplementować trzy pierwsze kroki algorytmu JPEG oraz dwa ostatnie, a następnie odwrócić tenże proces. Program w języku Python implementujący te kroki, korzysta z większej ilości bibliotek niż rozwiązania poprzednich zadań (rysunek 23).



```
lab2zad4.py

import numpy as np
from PIL import Image
import cv2
from collections import Counter
import heapq
```

Rysunek 23

Paczki numpy oraz Pillow używane są tak jak w poprzednich zadaniach. Bibliotek openCV używana jest do konwersji z modelu luminacja-chrominancja na RGB oraz zmiany rozmiaru obrazów. Narzędzie Counter z modułu collections wykorzystywane jest przy kodowaniu Huffmana (zlicza wystąpienia słowa). Z kolei biblioteka heapq używana jest do tworzenia drzewa binarnego w algorytmie Huffmana.

W programie tym korzystamy z funkcji colors() (rysunek 12), używanej w poprzednich zadaniach, jako wejściowej tablicy. Wykorzystywana będzie także wcześniej omawiana funkcja show\_image().

Najpierw stworzono funkcję konwertującą tablicę współrzędnych koloru z modelu RGB na model luminacja-chrominancja, zgodnie ze wzorem z rysunku 24<sup>3</sup>.

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

Rysunek 24

Funkcja dokonująca tej konwersji to stepOne() dana rysunkiem nr 25. Konwersja jest przeprowadzana ręcznie (bezpośrednio dokonujemy przekształceń na tablicy numpy), stąd czas konwersji jest względnie długi. Z tego powodu, przy konwersji odwrotnej korzystać będziemy z narzędzi biblioteki openCV, które umożliwiają konwersje między przestrzeniami barw w zoptymalizowany, szybszy sposób. Współrzędne są normalizowane do zakresu 0-255.

Wynikiem działania tej funkcji jest tablica współrzędnych barw w modelu luminacja-chrominancja. Kanały chrominancji winny zostać następnie podpróbkowane.

Do tego celu stworzono funkcję stepTwo() daną rysunkiem nr 26.

<sup>3</sup> <https://pawelskaruz.pl/2017/04/kompresja-jpeg-oraz-jej-wykorzystanie-w-steganografii/>



```
lab2zad4.py

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):

            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
```

Rysunek 26

```
lab2zad4.py

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

Rysunek 25

Funkcja stepTwo() wykorzystuje tzw. slicing z krokiem, tzn. tworzy nową tablicę wykorzystując co factor-tą współrzędną.

Kolejnym krokiem było wyrównanie rozmiarów obrazu, aby był podzielny na bloki 8x8, które to następnie będą dalej przetwarzane. W tym celu zaimplementowano funkcję stepThree() daną rysunkiem nr 27.

```
lab2zad4.py

def stepThree(arrayChannel):
    print(f"stepThree: 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='constant')
    print(f"stepThree: Rozmiar po paddingu: {channel_padded.shape}")
    return channel_padded
```

Rysunek 27



W funkcji tej wykorzystano narzędzie numpy pad() o parametrze mode='constant', co skutkuje dopełnieniem tablicy zerami, w przypadku braku podzielności wysokości lub szerokości przez 8.

Tak wyrównaną tablicę możemy dalej przetwarzać. W tym zadaniu pomijamy kroki zawierające transformatę kosinusową i przechodzimy od razu do zygzakowania. Pominięcie tych kroków skutkować będzie artefaktami na obrazie, będącymi skutkiem podziału obrazu na bloki.

Tak więc w celu zaimplementowania algorytmu ZigZag stworzono funkcje zigzag\_block() oraz stepSeven() dane rysunkiem nr 28.

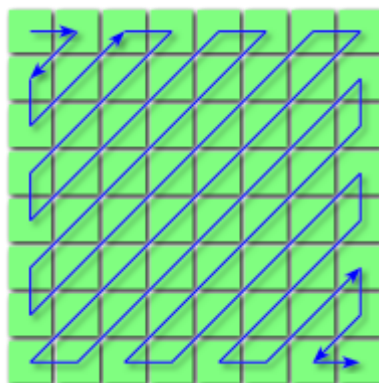
```
lab2zad4.py

def zigzag_block(block):
    zigzag_index = [
        (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),
    ]
    return [block[i, j] for i, j in zigzag_index]

def stepSeven(channel):
    h, w = channel.shape
    result = []
    for i in range(0, h, 8):
        for j in range(0, w, 8):
            block = channel[i:i+8, j:j+8]
            result.append(zigzag_block(block))
    print(f"stepSeven: ZigZag. Liczba bloków: {len(result)}")
    return result
```

Rysunek 28

Funkcja zigzag\_block() zawiera w sobie indeksy w kolejności zgodnej z algorytmem ZigZag (rysunek 29<sup>4</sup>).



Rysunek 29

<sup>4</sup> <https://mkramarczyk.zut.edu.pl/?cat=M&l=JPEG>

Funkcja ta wykorzystuje te współczynniki, aby przekształcić blok 8x8 (tablica dwuwymiarowa; macierz) na tablicę jednowymiarową (wektor).

Funkcja `stepSeven()` wywołuje funkcję `zigzag_block()` dla każdego bloku 8x8 i zapisuje wszystkie wyniki kolejno na liście `result`.

Końcowym etapem jest kodowanie Huffmana wektorów otrzymanych przy pomocy zygzakowania. W tym celu zaimplementowano funkcje `build_huffman_tree()` oraz `stepEight()` (rysunek 30).

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.

Operacje odwrotne tworzone były „w locie” i nie posiadają swoich własnych funkcji. Początkowo taki sposób implementacji wydawał się łatwiejszy, jednak po napisaniu programu i jego testowaniu, przy wykonywaniu zadania piątego zaimplementowane będą osobne funkcje implementujące operacje odwrotne, gdyż nieuporządkowane kroki odwrotne okazały się generować sporo problemów przy debugowaniu programu. Na ten moment omówimy jednak pierwsze rozwiązanie.

```

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

def stepEight(zigzag_blocks):
    encoded_blocks = []

    all_symbols = []

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

    huffman_dict = build_huffman_tree(all_symbols)

    bitstream = ""
    for rle in encoded_blocks:
        for symbol in rle:
            bitstream += huffman_dict[symbol]

    print(f"stepEight: Huffman. Liczba bitów: {len(bitstream)}")
    return bitstream, huffman_dict

```

Rysunek 30

Finalnie stworzono funkcję `process()` wykonującą kolejno kroki algorytmu JPEG oraz operacje odwrotne. Funkcja `process` dana jest rysunkiem nr 31. Tak więc tworzymy tablicę `numpy` przy pomocy funkcji `colors()`, konwertujemy ją do przestrzeni luminacja-chrominancja i zapisujemy jako `input.jpg`. Będziemy używać tego obrazu jako referencyjnego do obrazów wynikowych. Następnie rozdzielamy tak powstałą tablicę na poszczególne kanały i próbkujemy kanały chrominancji. Po tym kroku wyrównujemy wszystkie kanały do postaci wielokrotności bloku  $8 \times 8$ . Wtedy możemy przejść do zygzakowania, a po nim do kodowania Huffmana. W efekcie otrzymujemy zakodowane ciągi bitów oraz słowniki kodowe.

Następnie przechodzimy do operacji odwrotnych. Używamy funkcji pomocniczej `decode_bitstream` (rysunek 32), która przekształca strumień bitów na zdekodowane bloki danych. Używa odwrotnego słownika Huffmana, aby znaleźć odpowiednie symbole. Po tym, rekonstruujemy poszczególne kanały korzystając z funkcji pomocniczej `inverse_zigzag()` (rysunek 33), która jest operacją odwrotną do zygzakowania. Tak jak w pierwotnym algorytmie ZigZag, obraz przetwarzany jest blok po bloku. Jeśli współczynnik próbkowania wynosił więcej niż jeden, obraz jest nadpróbkowany przy pomocy interpolacji sześcienniej w narzędziu `resize` biblioteki `opencv`. Obraz jest także przycinany do pierwotnych wymiarów. Finalnie, obraz jest konwertowany z przestrzeni YCbCr do RGB i zapisywany jako `output.jpg`.

Funkcja `process()` testowana była dla współczynników 1, 2 oraz 4. Takie operacje wykonuje również korespondujący z kodem Python plik wykonywalny `lab2zad4.exe`.

```

lab2zad4.py

def process(sampling_factor):
    print(f"\n=== Próbkowanie {sampling_factor}x ===")

    img = colors()
    original_h, original_w = img.shape[:2]

    ycbcr = stepOne(img)
    save_ycbcr(ycbcr, "input.jpg")

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

    Y_s = Y
    Cb_s = stepTwo(Cb, sampling_factor)
    Cr_s = stepTwo(Cr, sampling_factor)

    Y_p = stepThree(Y_s)
    Cb_p = stepThree(Cb_s)
    Cr_p = stepThree(Cr_s)

    Y_zz = stepSeven(Y_p)
    Cb_zz = stepSeven(Cb_p)
    Cr_zz = stepSeven(Cr_p)

    y_stream, y_dict = stepEight(Y_zz)
    cb_stream, cb_dict = stepEight(Cb_zz)
    cr_stream, cr_dict = stepEight(Cr_zz)

    y_blocks = decode_bitstream(y_stream, y_dict, len(Y_zz))
    cb_blocks = decode_bitstream(cb_stream, cb_dict, len(Cb_zz))
    cr_blocks = decode_bitstream(cr_stream, cr_dict, len(Cr_zz))

    def rebuild_channel(blocks, h, w):
        channel = np.zeros((h, w), dtype=np.uint8)
        i = 0
        for y in range(0, h, 8):
            for x in range(0, w, 8):
                if i < len(blocks):
                    block = inverse_zigzag(blocks[i])
                    block = np.clip(block, 0, 255).astype(np.uint8)
                    if y + 8 <= h and x + 8 <= w:
                        channel[y:y+8, x:x+8] = block
                    i += 1
        return channel

    Y_rec = rebuild_channel(y_blocks, Y_p.shape[0], Y_p.shape[1])
    Cb_rec = rebuild_channel(cb_blocks, Cb_p.shape[0], Cb_p.shape[1])
    Cr_rec = rebuild_channel(cr_blocks, Cr_p.shape[0], Cr_p.shape[1])

    if sampling_factor > 1:
        Cb_rec = cv2.resize(Cb_rec, (original_w, original_h),
            interpolation=cv2.INTER_CUBIC)
        Cr_rec = cv2.resize(Cr_rec, (original_w, original_h),
            interpolation=cv2.INTER_CUBIC)
    else:
        Cb_rec = Cb_rec[:original_h, :original_w]
        Cr_rec = Cr_rec[:original_h, :original_w]

    Y_rec = Y_rec[:original_h, :original_w]

    ycbcr_rec = np.stack([Y_rec, Cb_rec, Cr_rec], axis=-1)
    save_ycbcr(ycbcr_rec, "output"+str(sampling_factor)+".jpg")
    show_image("output"+str(sampling_factor)+".jpg")

```

Rysunek 31



```

lab2zad4.py

def decode_bitstream(bitstream, huffman_dict, num_blocks):
    reverse_dict = {v: k for k, v in huffman_dict.items()}

    decoded_blocks = []
    current_block = [0]
    buffer = ""
    blocks_decoded = 0

    for bit in bitstream:
        buffer += bit
        if buffer in reverse_dict:
            symbol = reverse_dict[buffer]
            if symbol == (0, 0):
                current_block += [0] * (64 - len(current_block))
                decoded_blocks.append(current_block)
                current_block = [0]
                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 32

```

lab2zad4.py

def inverse_zigzag(zigzag):
    zigzag_index = [
        (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),
    ]
    block = np.zeros((8, 8), dtype=np.int16)

    for idx, (i, j) in enumerate(zigzag_index):
        block[i, j] = zigzag[idx]

    return block

```

Rysunek 33

Wyniki działania programu dane są rysunkami 34, 35 i 36 odpowiednio dla współczynnika próbkowania 1, 2 i 4.



**Rysunek 34**



**Rysunek 35**



**Rysunek 36**

Widzimy stopniową degradację, w miarę zwiększania współczynnika próbkowania (zmniejszania częstotliwości próbkowania). Siatka z kropek przy braku próbkowania spowodowana jest podziałem blokowym, którego konsekwencje normalnie byłyby łagodzone podczas transformacji kosinusowej. W przypadku większego współczynnika, próbkowanie chrominancji powoduje utratę informacji o kolorach, co może prowadzić do dominacji jednego koloru - w naszym przypadku zielonego.

### **3. Pliki źródłowe**

W załączonym folderze znajdują się kody źródłowe w języku Python, potrzebne pliki graficzne oraz folder dist zawierający pliki wykonywalne utworzone na podstawie ww. kodów. Pliki wykonywalne zostały utworzone przy użyciu narzędzia pyInstaller.



## **Źródła**

<https://mkramarczyk.zut.edu.pl/?cat=M&l=JPEG>

[https://datko.pl/IOb/lab4.pdf?fbclid=IwZXh0bgNhZW0CMTAAR2s4e3nIjo9ZXV75B9Q4-Q31U\\_9v90H2uE2TvflfKj0kEhFjx3-E0kZI\\_A\\_aem\\_y3urkb9jSojd1s10GoylIQ](https://datko.pl/IOb/lab4.pdf?fbclid=IwZXh0bgNhZW0CMTAAR2s4e3nIjo9ZXV75B9Q4-Q31U_9v90H2uE2TvflfKj0kEhFjx3-E0kZI_A_aem_y3urkb9jSojd1s10GoylIQ)

<https://www.w3.org/TR/png/>

<https://netpbm.sourceforge.net/doc/ppm.html>