

Inżynieria obrazów

Programowe przetwarzanie obrazów

Marcin Lasak 272886

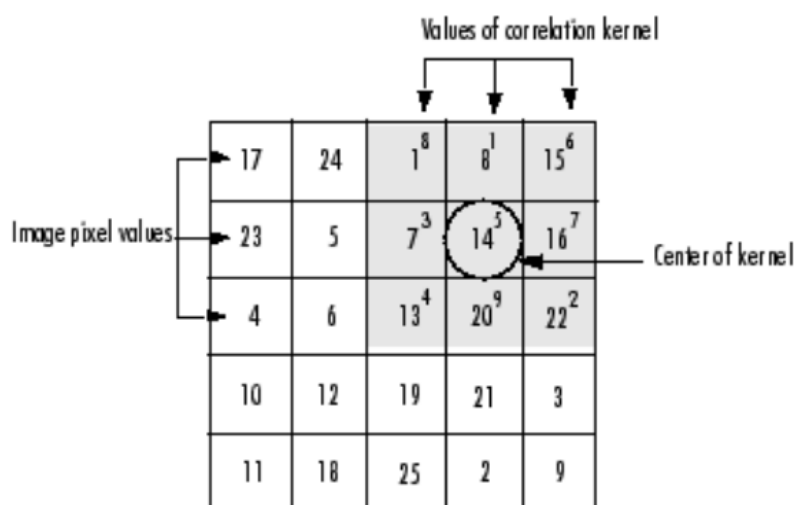
Spis treści

1. Wstęp teoretyczny	3
1.1 Filtracja obrazów	3
1.2 RGB	4
1.3 YCbCr – model luminancja-chrominancja.....	4
1.4 DVB	4
1.5 Błąd średniokwadratowy	5
2. Zadania	6
2.1 Filtr górnoprzepustowy	6
2.2 Przekształcenia kolorów	8
2.3 Konwersja z RGB do YCbCr	9
2.3 Symulacja transmisji DVB	12
2.5 Błąd średniokwadratowy	16
3. Zmiana struktury programów	17
Źródła	18

1. Wstęp teoretyczny

1.1 Filtracja obrazów

Jest to przekształcanie obrazu, w którym wartości pikseli wyjściowych są wynikami operacji matematycznych na pikselach sąsiednich. Piksele są traktowane jak sygnały, analogicznie do metod cyfrowego przetwarzania sygnałów zadanych funkcjami (filtrowanie np. sygnałów dźwiękowych).

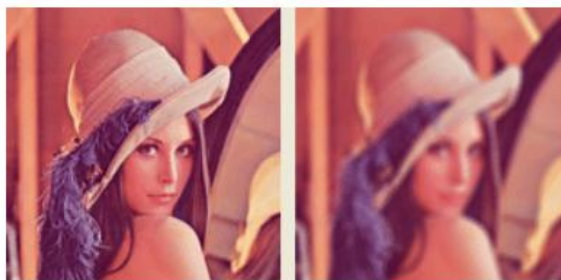


Rysunek 1

Przy zadanej masce (rysunek 1¹), obliczamy nową wartość piksela, poprzez obliczenie sumy ważonej składowej punktu i wszystkich sąsiadów zgodnie z wagami wskazanymi przez maskę filtra. Następnie tak otrzymana suma jest dzielona przez sumę wag.

W kontekście obrazów, częstotliwością nazywamy częstotliwość zmian wartości pikseli w obrazie, które są interpretowane w kontekście przestrzennym. Tj. o niskiej częstotliwości mówimy w przypadku wolnych zmian w obrazie; wartości pikseli zmieniają się w sposób łagodny i stopniowy. Możemy uprościć to zagadnienie do ogólnych kształtów, bez szczegółów. Natomiast o wysokiej częstotliwości mówimy, gdy zmiany wartości pikseli są nagłe, gwałtowne. Są to ostre krawędzie, detale, tekstura. To zagadnienie możemy uprościć do wszelkich szczegółów obrazu.

Tak więc filtry dolnoprzepustowe będą szczegóły z obrazu usuwać (rysunek 2²), a górnoprzepustowe – je uwypuklać.



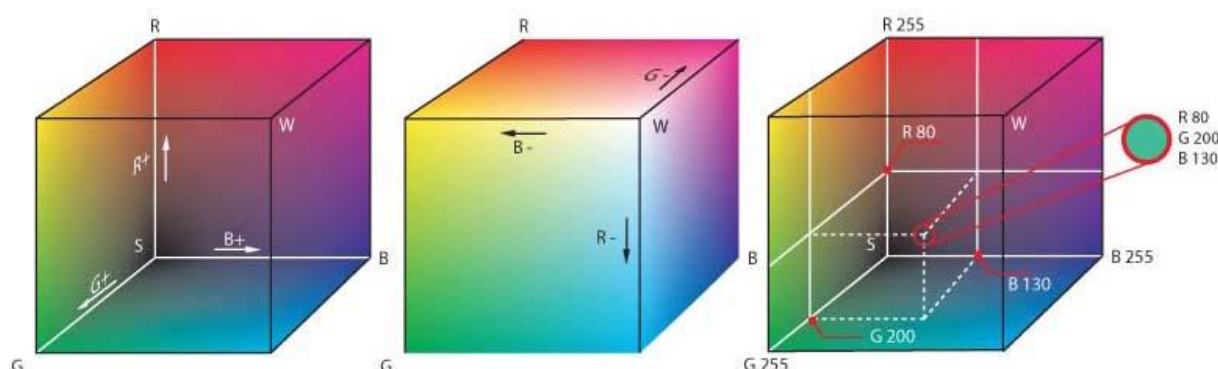
Rysunek 2

¹ https://multimed.org/student/pdio/pdio09_filtracja_obrazu_poprawa_jakosci_tekstu.pdf

² <http://www.algorytm.org/przetwarzanie-obrazow/filtrowanie-obrazow.html>

1.2 RGB

RGB (Red Green Blue) to model przestrzeni barw, opisywanej trzema współrzędnymi: Red – czerwień, Green – zieleń, Blue – niebieski. Każda współrzędna zapisywana jest na ośmiu bitach; w zakresie 0-255. Wartość 0 oznacza brak koloru, natomiast 255 jego pełną intensywność. Rysunek 3³ przedstawia wizualizację przestrzeni RGB.



Rysunek 3

System RGB posiada również wariant zmiennoprzecinkowy, gdzie wartości współrzędnych przyjmują wartości z zakresu 0-1.

1.3 YCbCr – model luminancja-chrominancja

YCbCr to model przestrzeni kolorów bazującą na modelu RGB, w której współrzędne przechowują informacje o luminancji – mierze natężenia światła, oraz chrominancji składowej odpowiedzialnej za odcień i nasycenie koloru. Przestrzeń luminancja-chrominancja, tak jak przestrzeń RGB, zadana jest trzema współrzędnymi:

- Y – składowa luminancji; obraz w czerni i bieli odpowiadający za natężenie światła;
- Cb – składowa różnicowa chrominancji odpowiadająca różnicy między luminancją, a kolorem niebieskim,
- Cr – składowa różnicowa chrominancji odpowiadająca różnicy między luminancją, a kolorem czerwonym.

Nałożenie takich trzech macierzy pikseli pozwala na uzyskanie pełnobarwnego obrazu.

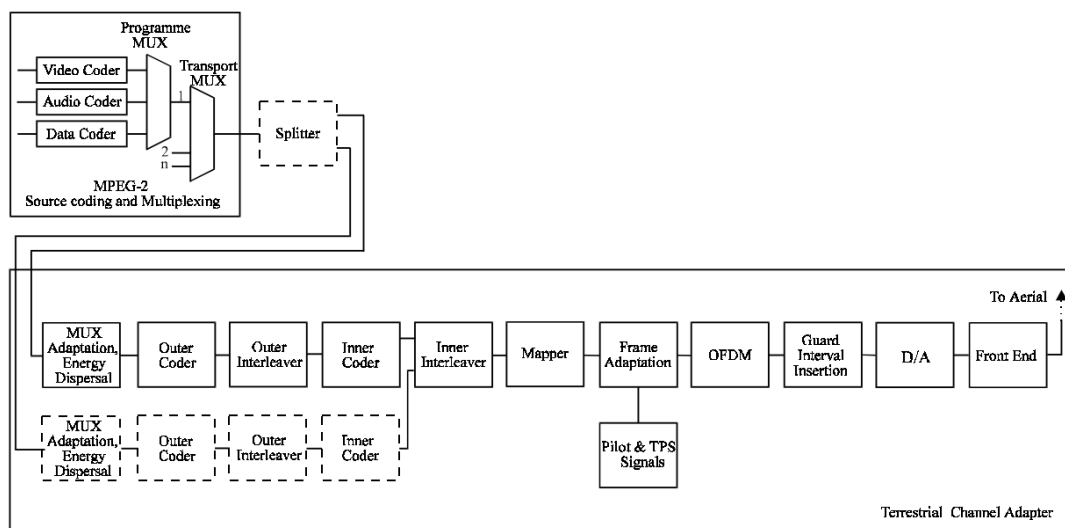
Model ten jest wykorzystywany przy kompresjach obrazu, ponieważ zmniejszanie rozdzielczości składowych chrominancji, przy zachowaniu wejściowej składowej luminancji, zmniejsza rozmiar obrazu w pamięci, zachowując pozornie swoją pierwotną jakość. Wynika to z faktu, iż ludzkie oko jest dużo bardziej wrażliwe na natężenie światła, niż na kolor.

1.4 DVB

DVB (Digital Video Broadcasting) to zbiór standardów transmisji telewizji cyfrowej. Przy takiej transmisji wykorzystywany jest model luminancja-chrominancja, ponieważ pozwala na kompresję obrazów, co jest konieczne aby efektywnie wykorzystać dostępne pasmo. Po redukcji chrominancji obrazy kompresowane są do formatu MPEG. Rysunek 4⁴ przedstawia schemat modelu transmisji DVB.

³ <https://pl.wikipedia.org/wiki/RGB>

⁴ https://www.etsi.org/deliver/etsi_en/300700_300799/300744/01.01.02_60/en_300744v010102p.pdf



Rysunek 4

1.5 Błąd średniokwadratowy

Błąd średniokwadratowy (Mean Squared Error) jest wartością oczekiwaną kwadratu błędu między estymatorem, a wartością estymowaną (rysunek 5⁵). W przypadku przetwarzania obrazów jest to miara różnicy między dwoma obrazami (np. oryginałem, a obrazem po kompresji lub rekonstrukcji). Oblicza się go jako średnią kwadratów różnic między odpowiadającymi sobie pikselami. Im niższa wartość MSE, tym lepsza jest jakość przetwarzania; obraz wynikowy jest bliższy oryginałowi. Jeśli wartość MSE wynosi 0, oznacza to, że obrazy są identyczne.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

Mean Error Squared

Rysunek 5

⁵ <https://suboptimal.wiki/explanation/mse/>

2. Zadania

2.1 Filtr górnoprzepustowy

W tym zadaniu należało wykonać filtr górnoprzepustowy, tzw. detektor krawędzi, o zadanej masce. Zadanie wykonano w języku Python. Skorzystano z bibliotek numpy, do właściwego przetwarzania tablic reprezentujących obraz w przestrzeni RGB oraz Pillow, do operacji na plikach graficznych (odczyt, zapis).

Filtracja odbywać się będzie przy pomocy zaimplementowanej funkcji `hpfiler()`, przyjmującą jako argument nazwę pliku graficznego poddawanego filtracji.

Funkcja ta wpierw otwiera zadany plik (rysunek 6), a następnie konwertuje go do modelu RGB, by następnie zapisać go w tablicy numpy. Dalej zapisujemy podaną maskę jako tablicę numpy, by móc ją wykorzystywać w późniejszych obliczeniach. Zapisujemy także wymiary maski oraz rozmiary wyrównania (padding). Tworzymy także wyjściową tablicę, wstępnie wypełnioną zerami.



```
lab1zad1.py

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

    kernel = np.array([
        [-1, -1, -1],
        [-1, 8, -1],
        [-1, -1, -1]
    ])

    kh, kw = kernel.shape
    pad_h, pad_w = kh // 2, kw // 2

    output = np.zeros_like(img_array, dtype=np.float32)
```

Rysunek 6

Przed obliczeniem nowych wartości pikseli (rysunek 7), każdy kanał jest wyrównywany: dodawane są piksele, aby poprawnie przetworzyć skrajne punkty obrazu. Zastosowano wypełnienie z odbiciem, zamiast dopełniania zerami.

Następnie obliczana jest nowa wartość danego kanału dla każdego piksela obrazu. Taki krok powtarzany jest dla każdego kanału, w rezultacie czego otrzymujemy tablicę z przetworzonymi współrzędnymi RGB.

```
lab1zad1.py

for channel in range(3):
    channel_data = img_array[:, :, channel]
    padded = np.pad(channel_data, ((pad_h, pad_w), (pad_w, pad_h)), mode='reflect')

    for i in range(channel_data.shape[0]):
        for j in range(channel_data.shape[1]):
            region = padded[i:i + kh, j:j + kw]
            value = np.sum(region * kernel)
            output[i, j, channel] = value
```

Rysunek 7

Finalnie normalizujemy wartości do zakresu 0-255 (ponieważ obliczone wartości mogły wyjść za ten zakres) i zapisujemy jako typ uint8 (liczba całkowita zapisana na 8 bitach).

Do sprawdzenia wyników filtrowania korzystamy z funkcji pomocniczych save() oraz show_image() (rysunek 8).

```
lab1zad1.py

def save(array, name):
    image = Image.fromarray(array)
    image.save(name + "-hpfoutput.jpg")

# Wyświetlanie obrazu
def show_image(filename):
    image = Image.open(filename)
    image.show()
```

Rysunek 8

Program Python lab1zad1.py i korespondujący z nim plik wykonywalny, wyświetlają najpierw obraz input1.jpg, przetwarzają go i zapisują, a następnie wyświetlają przetworzony obraz. Obraz wejściowy dany jest rysunkiem nr 9, natomiast wyjściowy rysunkiem nr 10.



Rysunek 9



Rysunek 10

2.2 Przekształcenia kolorów

Kolejnym zadaniem, było przekształcenie kolorów obrazu z wykorzystaniem konwersji na zmiennoprzecinkowy format RGB, z wykorzystaniem wzoru z rysunku nr 11.

$$\begin{bmatrix} R_{new} \\ G_{new} \\ B_{new} \end{bmatrix} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.689 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Rysunek 11

W tym zadaniu również korzystamy z bibliotek numpy oraz Pillow. W tym programie zdecydowano się na rozdzielenie etapu odczytu pliku graficznego z konwersją na tablicę numpy oraz właściwego przetwarzania obrazu. Dlatego też zaimplementowano funkcje `file_to_array()`, odpowiedzialną za odczyt i konwersję obrazu, oraz `inttofloat()`, przetwarzającą obraz w formacie zmiennoprzecinkowym (rysunek 12).

```
lab1zad2.py

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

def inttofloat(img_array):
    width=img_array.shape[1]
    height=img_array.shape[0]
    output = np.zeros_like(img_array, dtype=np.float32)

    for i in range(height):
        for j in range(width):
            r, g, b = img_array[i, j] / 255.0

            rnew = 0.393*r+0.769*g+0.189*b
            gnew = 0.349*r+0.689*g+0.168*b
            bnew = 0.272*r+0.534*g+0.131*b

            output[i, j] = [min(rnew, 1.0), min(gnew, 1.0), min(bnew,
1.0)]
    return output
```

Rysunek 12

W funkcji `inttofloat()`, w pierw tworzymy tablicę numpy wypełnioną zerami. Co ważne, używanym typem danych w tablicy jest `float32` (trzydziestodwubitowa liczba zmiennoprzecinkowa). Następnie w pętli dokonujemy konwersji każdego piksela na format zmiennoprzecinkowy oraz jego przekształcenia wg wzoru z rysunku 11. Wpierw obliczamy współrzędne RGB w nowym formacie, a następnie dokonujemy przekształceń kolorów. Tak

otrzymane dane zapisujemy w tablicy numpy, z uwzględnieniem możliwego przekroczenia wartości 1.

W celu sprawdzenia działania tej funkcji, przetwarzamy obraz z zadania pierwszego, zapisujemy go i wyświetlamy. W tym celu również korzystamy z ww. funkcji pomocniczych, z tym że funkcja zapisu została zmodyfikowana, aby przed zapisem format zmiennoprzecinkowy został zamieniony na całkowity (uint8). Takie operacje wykonuje również korespondujący z programem Python lab1zad2.py plik wykonywalny.

Wynikiem działania programu jest rysunek nr 13.



Rysunek 13

2.3 Konwersja z RGB do YCbCr

W tym punkcie, należało zaimplementować konwersję obrazu z przestrzeni RGB, do przestrzeni YCbCr wg wzoru danego rysunkiem nr 14.

$$\begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.229 & 0.587 & 0.114 \\ 0.500 & -0.418 & -0.082 \\ -0.168 & -0.331 & 0.500 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Rysunek 14

Ponownie skorzystano z bibliotek numpy i Pillow oraz ww. funkcji pomocniczych. Dodatkowo użyto biblioteki openCV do konwersji z YCbCr do RGB.

W celu konwersji tablicy numpy zawierającą współrzędne RGB, do tablicy zawierającej współrzędne YCbCr zaimplementowano funkcję RGBtoYCBCR() daną rysunkiem nr 15. Tworzymy w niej nową tablicę numpy wypełnioną zerami, a następnie w zagnieżdżonej pętli przechodzimy po każdym pikselu obrazu w modelu RGB, obliczamy współrzędne w modelu YCbCr i zapisujemy w odpowiednim miejscu tablicy. Współrzędne YCbCr są zaokrąglane tak, aby nie przekraczały wartości 1.

```
lab1zad3.py

def RGBtoYCBCR(RGBarray):
    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 = 0.229 * R + 0.587 * G + 0.114 * B
            Cb = 0.500 * R - 0.418 * G - 0.082 * B + 128
            Cr = -0.168 * R + 0.331 * G + 0.5 * B + 128

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

    return ycbcr
```

Rysunek 15

W celu sprawdzenia poprawności działania tej konwersji, zaimplementowano funkcję przeprowadzającą konwersję odwrotną i zapisującą tak przetworzony obraz oraz funkcję zapisującą każdą składową obrazu w modelu YCbCr w odcieniach szarości (rysunek 16) .

```
lab1zad3.py

def save_ycbcr(ycbcr_array, filename):
    rgb_image = cv2.cvtColor(ycbcr_array, cv2.COLOR_YCrCb2RGB)
    image = Image.fromarray(rgb_image)
    image.save(filename)

def save_image(array, filename):
    image = Image.fromarray(array)
    image.save(filename)

def show_image(filename):
    image = Image.open(filename)
    image.show()

def ycbcr(ycbcr_array):
    # Składowe Y, Cb, Cr
    Y = ycbcr_array[:, :, 0]
    Cb = ycbcr_array[:, :, 1]
    Cr = ycbcr_array[:, :, 2]

    save_image(Y, "Y.png")
    save_image(Cb, "Cb.png")
    save_image(Cr, "Cr.png")
```

Rysunek 16

Program wykonuje więc konwersję z RGB do YCbCr oraz konwersję odwrotną, zapisuje wynik tej operacji i składowe powstałe przy pierwszej konwersji. Obraz wejściowy dany jest rysunkiem nr 17. Wyniki działania programu dane są rysunkami nr 18, 19, 20, 21.



Rysunek 17 Oryginal



Rysunek 18 Przetworzony obraz



Rysunek 19 Składowa luminancji – Y



Rysunek 20 Składowa chrominancji – Cb



Rysunek 21 Składowa chrominancji – Cr

2.3 Symulacja transmisji DVB

W tym zadaniu należało zasymulować transmisję obrazu w systemie DVB. Symulowana będzie część kodowania i dekodowania sygnału. Symulację przeprowadzono w języku Python, z wykorzystaniem bibliotek numpy, Pillow oraz openCV.

W tym celu na pliku graficznym dokonujemy operację konwersji z modelu RGB na model YCbCr. Następnie przeprowadzamy operacje podpróbkowania kanałów chrominancji (po stronie nadawcy) oraz nadpróbkowania (po stronie odbiorcy). Po tym składowe są łączone w jeden obraz i konwertowane z powrotem do modelu RGB.

Do konwersji z modelu RGB do YCbCr wykorzystano funkcję z zadania 2 (rysunek 15).

W celu przeprowadzenia konwersji odwrotnej, zaimplementowano funkcję YCBCRtoRGB() daną rysunkiem nr 22.

W poprzednim zadaniu konwersja ta była przeprowadzana przy użyciu narzędzi biblioteki openCV. W tym przypadku zdecydowano się na ręczną implementację tej konwersji, co znacznie wydłuża działanie programu. Jest to spowodowane dużo lepszą optymalizacją narzędzi biblioteki openCV.

```

lab1zad4.py

def YBCRtoRGB(ybcr_array):
    height, width = ybcr_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 = ybcr_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 22

Następnie zaimplementowano funkcje wykonujące operacje podpróbkowania i nadpróbkowania dane rysunkiem nr 23.

```

lab1zad4.py

def downsample(Cb, Cr):
    Cb_downsampled = Cb[::2, ::2]
    Cr_downsampled = Cr[::2, ::2]
    return Cb_downsampled, Cr_downsampled

def upsample(Cb_down, Cr_down, height, width):
    Cb_upsampled = cv2.resize(Cb_down, (width, height), interpolation=cv2.INTER_LINEAR)
    Cr_upsampled = cv2.resize(Cr_down, (width, height), interpolation=cv2.INTER_LINEAR)
    return Cb_upsampled, Cr_upsampled

```

Rysunek 23

Funkcja `downsample()` wykonuje podpróbkowanie kanałów chrominancji. Operacja ta polega na zmniejszeniu rozdzielczości tablicy składowej chrominancji, przez pobieranie tylko współrzędnych znajdujących się w co drugim wierszu w co drugiej kolumnie (elementów macierzy a_{ij} , gdzie $i = 2k, j = 2l, i, j \in \{0, 1, \dots, \frac{\text{liczba wierszy/kolumn}}{2}\}$).

Funkcja `upsample()` wykonuje operację nadpróbkowania kanałów chrominancji. Jest to czynność dużo bardziej złożona, wykorzystującą interpolację. Z tego powodu korzystamy z narzędzia biblioteki `openCV`. Funkcja `resize` pozwala na wypełnienie brakujących punktów z wykorzystaniem interpolacji liniowej. Istnieje możliwość wykorzystania innych rodzajów interpolacji, które dały by lepszy wynik, lecz postanowiono wykorzystać najprostsze narzędzie, aby zobaczyć jakie negatywne efekty może za sobą nieść nadpróbkowanie.

Finalnie zaimplementowano funkcję `process()`, wykonującą kolejne operacje symulujące transmisję DVB (rysunek 24).


```
lab1zad4.py

def process(name):
    img_array = file_to_array(name)

    # konwersja na YCbCr
    ycbcr = RGBtoYCBGR(img_array)

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

    # downsampling kanałów Cb i Cr
    Cb_down, Cr_down = downsample(Cb, Cr)

    # upsampling kanałów Cb i Cr do oryginalnych rozmiarów
    Cb_upsampled, Cr_upsampled = upsample(Cb_down, Cr_down, img_array.shape[0], img_array.shape[1])

    # złożenie Y, Cb i Cr z powrotem do YCbCr
    ycbcr_transmitted = np.zeros_like(ycbcr)
    ycbcr_transmitted[:, :, 0] = Y
    ycbcr_transmitted[:, :, 1] = Cb_upsampled
    ycbcr_transmitted[:, :, 2] = Cr_upsampled

    # konwersja z YCbCr z powrotem do RGB
    rgb_transmitted = YCBGRtoRGB(ycbcr_transmitted)

    save_image(Y, "Y.jpg")
    save_image(Cb, "Cb.jpg")
    save_image(Cr, "Cr.jpg")
    save_image(rgb_transmitted, name+"-transmitted.jpg")
```

Rysunek 24

Wynikiem działania programu jest „przesłany” obraz oraz jego składowe w modelu YCbCr w odcieniach szarości.

Obraz wejściowy dany jest rysunkiem nr 9. Obrazy wyjściowe dane są rysunkami 25, 26, 27 i 28.



Rysunek 25 Obraz po transmisji



Politechnika Wrocławska



Wrocław University
of Science and Technology

Rysunek 26 Składowa luminancji – Y



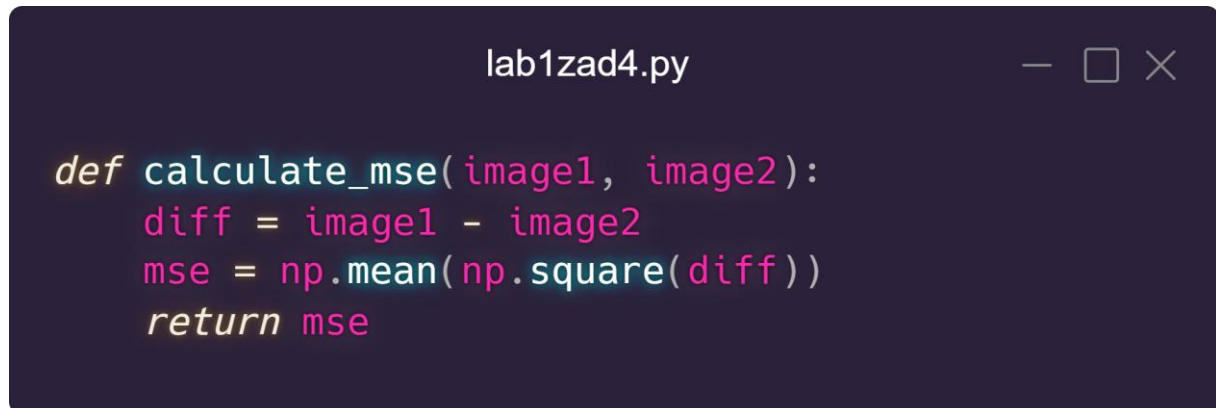
Rysunek 27 Składowa chrominancji – Cb



Rysunek 28 Składowa chrominancji – Cr

2.5 Błąd średniokwadratowy

Ostatnim zadaniem było zaimplementowanie obliczanie błędu średniokwadratowego pomiędzy dwoma obrazami. Biblioteka numpy pozwala nam na sprawne działania na macierzach, więc z jej wykorzystaniem stworzono funkcję `calculate_mse()` daną rysunkiem nr 29.

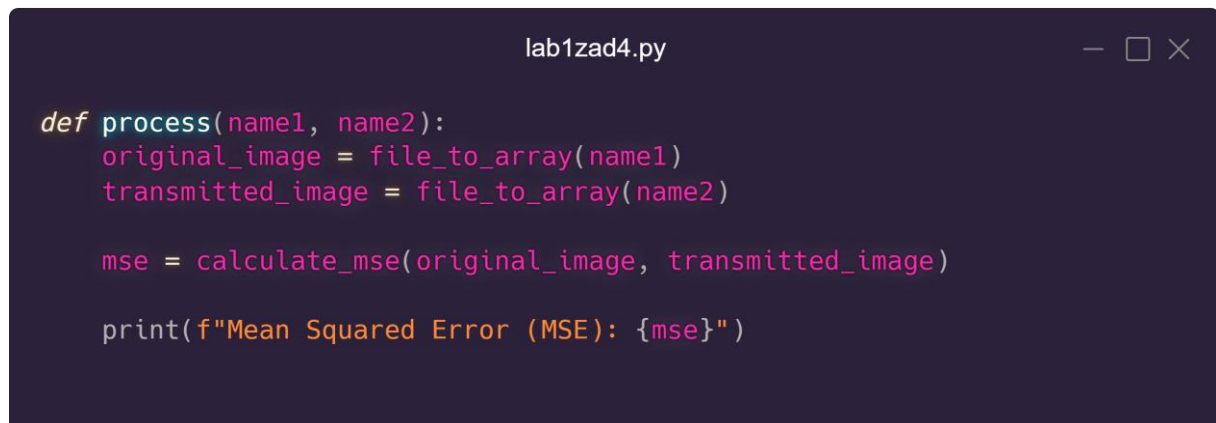
A screenshot of a code editor window titled 'lab1zad4.py'. The window has standard window controls (minimize, maximize, close) in the top right corner. The code defines a function 'calculate_mse' that takes two parameters, 'image1' and 'image2'. Inside the function, it calculates the difference 'diff' as 'image1 - image2', then calculates the mean squared error 'mse' as 'np.mean(np.square(diff))', and finally returns 'mse'.

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

Rysunek 29

Obiekty `image1` i `image2` to tablice numpy, które możemy swobodnie odejmować. Następnie podnosimy do kwadratu każdy element różnicy i obliczamy z nich średnią. W ten sposób otrzymujemy błąd średniokwadratowy między dwoma obrazami.

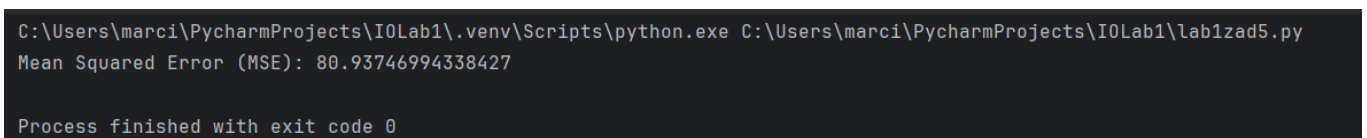
Następnie zaimplementowano funkcję `process()` (rysunek 30), konwertującą dwa obrazy do tablic numpy i obliczającą błąd średniokwadratowy.

A screenshot of a code editor window titled 'lab1zad4.py'. The window has standard window controls in the top right corner. The code defines a function 'process' that takes two parameters, 'name1' and 'name2'. Inside the function, it converts the image names to numpy arrays using 'file_to_array', calculates the MSE using the 'calculate_mse' function, and prints the result with a formatted string.

```
def process(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 30

Wynik działania programu dany jest rysunkiem nr 31.

A screenshot of a terminal window showing the execution of a Python script. The command line shows the path to the script 'C:\Users\marci\PycharmProjects\IOLab1\lab1zad5.py'. The output shows the Mean Squared Error (MSE) as 80.93746994338427. The terminal also shows the message 'Process finished with exit code 0'.

```
C:\Users\marci\PycharmProjects\IOLab1\.venv\Scripts\python.exe C:\Users\marci\PycharmProjects\IOLab1\lab1zad5.py  
Mean Squared Error (MSE): 80.93746994338427  
  
Process finished with exit code 0
```

Rysunek 31

3. Zmiana struktury programów

Zgodnie z zaleceniem prowadzącego, przekształcono programy dedykowane każdemu zadaniu, w jeden zbiorczy program, umożliwiający uruchomienie wybranego zadania. Stąd w folderze dist, brak wspomnianych plików wykonywalnych lab1zad1, lab1zad2 itd. Zamiast nich, znajduje się tam plik lab1.exe, umożliwiający wykonanie każdego z zadań. Co ważne, kody w języku Python zostały odpowiednio zakomentowane, aby umożliwić ich proste importy, więc same w sobie nie wykonują żadnej czynności. W przypadku chęci ręcznego uruchomienia kodów w Pythonie, należy odkomentować odpowiednie ostatnie linie.

Program lab1 działa w ciągłej pętli. Wybór zadania odbywa się przez podanie cyfry 1-5, natomiast zamknięcie programu możliwe jest przez podanie na wejściu znaku „q”. W folderze dist znajdują się wszystkie pliki potrzebne do poprawnego działania programu.

Plik lab1.exe został wygenerowany przy użyciu pyInstaller na podstawie kodu Python lab1.py.

Źródła

<https://fajne.studio/przestrzen-barwna-rgb-i-cmyk/>

https://pl.wikipedia.org/wiki/Digital_Video_Broadcasting

https://multimed.org/student/pdio/pdio09_filtracja_obrazu_poprawa_jakosci_tekstu.pdf

<http://www.algorytm.org/przetwarzanie-obrazow/filtrowanie-obrazow.html>

<https://pl.wikipedia.org/wiki/RGB>

https://www.etsi.org/deliver/etsi_en/300700_300799/300744/01.01.02_60/en_300744v010102p.pdf

<https://suboptimal.wiki/explanation/mse/>

<https://pwr.edu.pl/uczelnia/informacje-ogolne/materialy-promocyjne/logotyp>

<https://wmat.pwr.edu.pl/wydarzenia/uroczyste-otwarcie-budynku-wydzialu-matematyki-173.html>