

# Inżynieria obrazów - Laboratoria nr 2

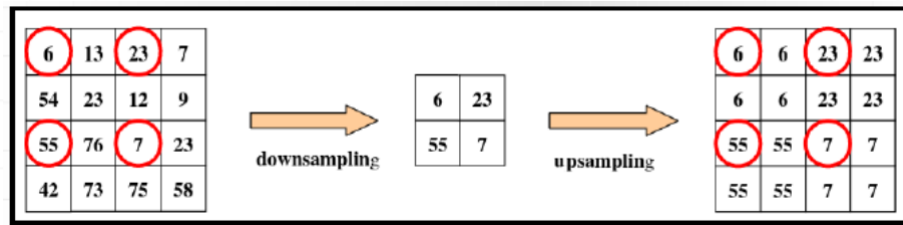
Szymon Wichrowski 263960

25 kwietnia 2024

## 1 Cele ćwiczenia

Celem ćwiczenia jest dalsza nauka programowego przetwarzania obrazów oraz zapoznanie się z podstawowymi modelami barw i ich praktycznym zastosowaniem, a ponadto zdobycie wiedzy o tym, jak można zapisywać dane graficzne w plikach.

## 2 Wykonanie symulacji transmisji obrazu w systemie DVB



Rysunek 1: Schemat obrazujący procesy downsamplingu oraz upsamplingu

Na początek, trzeba przeprowadzić konwersję wczytanego obrazu z modelu RGB do YCrCb. W ramach poprzednich zajęć dokonano takiej konwersji samodzielnie za pomocą odpowiednich operacji matematycznych na macierzach. Tym razem użyć można gotowej funkcji `cvtColor()` z biblioteki *OpenCV*.

Listing 1: Konwersja z formatu RGB do YCrCb

```
yCrCb = cv2.cvtColor(primaryImageRGB, cv2.COLOR_RGB2YCrCb)
```

Drugim krokiem jest realizacja operacji downsamplingu na kanałach Cr i Cb, a następnie przeprowadzenie upsamplingu na macierzach tych kanałów, tzn. najpierw zmniejszamy rozmiary macierzy, aby w dalszej kolejności przywrócić ich pierwotną rozdzielczość.

Początkowo zaimplementowano funkcje `downsampling()` i `upsampling()` samodzielnie. Funkcja `downsampling()` polegała na wyborze jednego piksela reprezentanta z każdego obszaru  $k \times k$ , gdzie  $k$  to współczynnik downsamplingu/upsamplingu, czyli ile razy zmniejszamy/zwiększamy daną macierz. Funkcja `upsampling()` zwiększała macierz  $k$  razy, aby uzyskać macierz o tym samym rozmiarze, co macierz pierwotna. Takie rozwiązanie działało, ale nie dla każdego przypadku, otóż zaimplementowana operacja upsamplingu faktycznie powiększała rozmiar macierzy po downsamplingu  $k$  razy, co niekoniecznie oznaczało powrót jej pierwotnej rozdzielczości. Na przykład, jeśli mielibyśmy macierz  $5 \times 5$ , a współczynnik  $k$  wynosiłby 2, to po downsamplingu otrzymalibyśmy macierz  $3 \times 3$ . Natomiast po wykonaniu funkcji `upsampling()`, długość oraz szerokość finalnej macierzy wzrosłyby  $k$  razy, czyli otrzymalibyśmy macierz o rozmiarze  $6 \times 6$  pikseli, zatem większą od macierzy pierwotnej.

Listing 2: Implementacja funkcji *downsampling()* oraz *upsampling()*

```
def downsampling(matrix, k):
    columns = len(matrix[0])
    elements_columns = math.ceil(columns / k)
    rows = len(matrix)
    elements_rows = math.ceil(rows / k)
    xNew = 0
    yNew = 0
    xOld = 0
    yOld = 0

    result_matrix = numpy.zeros((elements_rows, elements_columns))

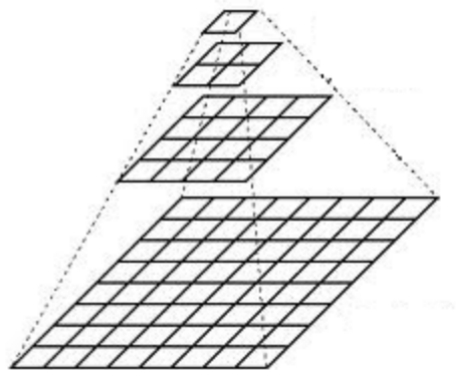
    while xNew < elements_rows:
        while yNew < elements_columns:
            result_matrix[xNew, yNew] = matrix[xOld, yOld]
            yOld = yOld + k
            yNew = yNew + 1
            xOld = xOld + k
            xNew = xNew + 1
            yOld = 0
            yNew = 0
        return result_matrix

def upsampling(matrix, k):
    columns = len(matrix[0])
    rows = len(matrix)
    xNew = 0
    yNew = 0
    xOld = 0
    yOld = 0

    result_matrix = numpy.zeros((rows * k, columns * k))

    while xOld < rows:
        while yOld < columns:
            for i in range(k):
                result_matrix[xNew, yNew + i] = matrix[xOld, yOld]
                yNew = yNew + k
                yOld = yOld + 1
            for i in range(k):
                result_matrix[xNew + i] = result_matrix[xNew]
                xNew = xNew + k
                xOld = xOld + 1
                yNew = 0
                yOld = 0
        return result_matrix
```

Skorzystano też z gotowych funkcji `pyrDown()` oraz `pyrUp()` z biblioteki *OpenCV*, które zmniejszają/zwiększają rozmiar macierzy dwukrotnie. Łatwo zauważyć, że w ten sposób sama idea operacji downsamplingu i upsamplingu ulega zmianie, w stosunku do poprzedniego rozwiązania. Nie mamy współczynnika  $k$  o dowolnie wybranej wartości. Do zwielokrotnienia rozmiaru macierzy należy powtórzyć wywołanie funkcji `pyrDown()`/`pyrUp()` dostatecznie wiele razy. Współczynnik  $k$  będzie tak naprawdę przyjmował wartości będące potęgami 2-ki, co widać na rysunku piramidy Gaussa, zaczerpniętym z dokumentacji biblioteki *OpenCV*.



Rysunek 2: Piramida Gaussa

Listing 3: Poprawiona implementacja funkcji *downsampling()* oraz *upsampling()*

```
def downsampling(image, k):
    if not k > 0:
        result = image
    else:
        result = cv2.pyrDown(image)
        for _ in range(k-1):
            result = cv2.pyrDown(result)
    return result

def upsampling(image, k):
    if not k > 0:
        result = image
    else:
        result = cv2.pyrUp(image)
        for _ in range(k-1):
            result = cv2.pyrUp(result)
    return result

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

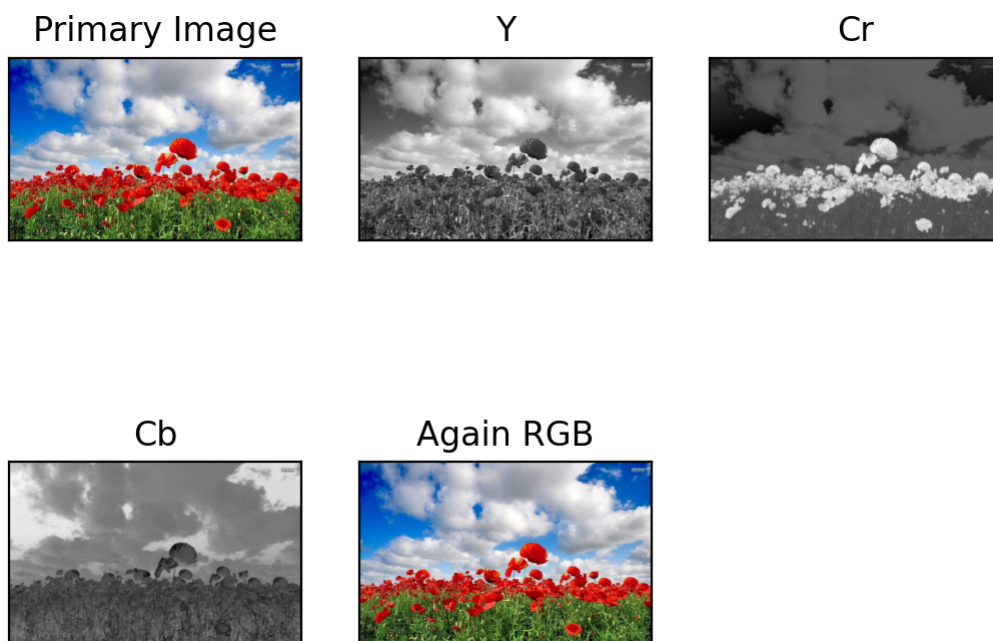
CrAfterDown = downsampling(Cr, 1)
CbAfterDown = downsampling(Cb, 1)

CrAfterUp = upsampling(CrAfterDown, 1)
CbAfterUp = upsampling(CbAfterDown, 1)

againYCrCb = cv2.merge([Y, CrAfterUp, CbAfterUp])

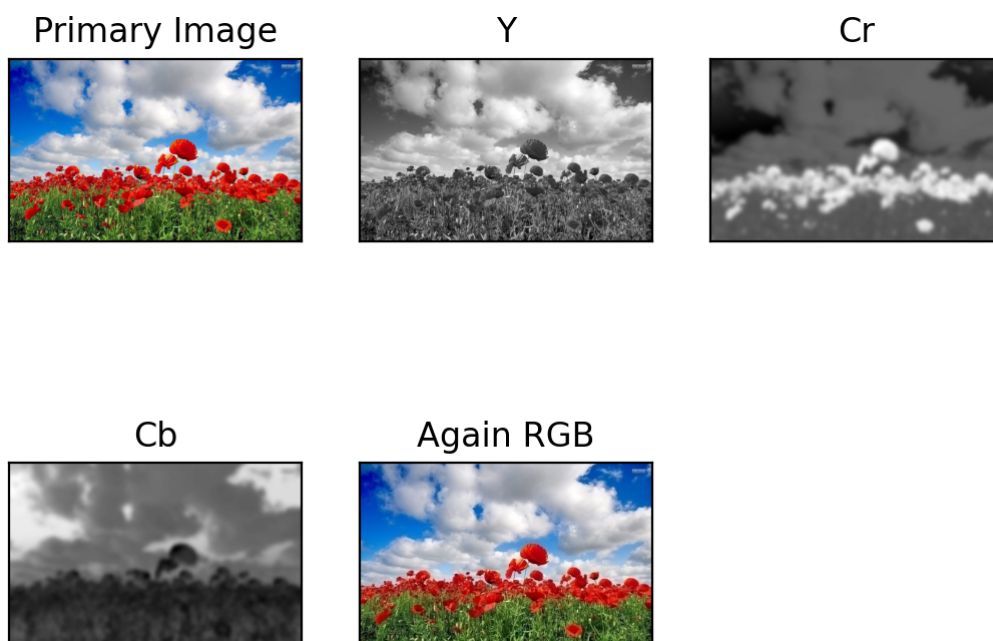
againRGB = cv2.cvtColor(againYCrCb, cv2.COLOR_YCrCb2RGB)
```

Ten sposób również nie rozwiązuje problemu z operacją upsamplingu, ale ostatecznie zdecydowano się właśnie na niego, z powodu merytorycznego potwierdzenia idei w oficjalnej dokumentacji, natomiast pierwsze rozwiązanie powstało bardziej na zasadzie intuicji.



Rysunek 3: Porównanie obrazu przed oraz po symulacji transmisji (dwukrotne zmniejszenie/powiększenie)

Na rysunku nr 3 nie widać znaczącego wpływu operacji na jakość obrazu. Sprawdźmy, co by się stało, gdyby macierze kanałów Cr oraz Cb zostały zmniejszone/powiększone wielokrotnie.



Rysunek 4: Porównanie obrazu przed oraz po symulacji transmisji (szesnastokrotne zmniejszenie/powiększenie)

Po wielokrotnym zmniejszeniu/powiększeniu macierzy kanałów Cb oraz Cr, wyraźnie widać spadek jakości obrazu. Nie jest on znaczący, jednak warto zauważyć, że tracimy w ten sposób wartości części pikseli z pierwotnego obrazu.

### 3 Obliczenie błędu średniokwadratowego (MSE) między obrazami z przed i po symulacji transmisji w systemie DVB

$$MSE = \frac{1}{m} \cdot \frac{1}{n} \cdot \sum_{i=1}^3 \sum_{j=1}^n (X_{ij} - \hat{X}_{ij})^2$$

Rysunek 5: Wzór na błąd średniokwadratowy

Na podstawie wzoru z rysunku nr 5 zaimplementowano funkcję `doMSE()`, która wyznacza błąd średniokwadratowy między dwoma obrazami, podanymi jako argumenty funkcji.

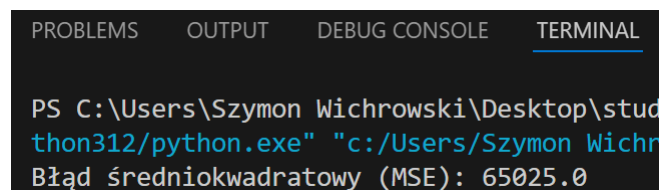
Listing 4: Implementacja funkcji wyznaczającej MSE dwóch obrazów

```
def doMSE(matrix1, matrix2):  
    return numpy.mean((matrix1 - matrix2)**2)
```

Jako, że wyznaczone MSE jest miarą średniej kwadratowej różnicy między pikselami dwóch obrazów, to im niższa wartość MSE, tym mniejsza różnica między obrazami. Jednakże interpretacja wyniku, czy wartość MSE jest niska, czy wysoka, może być na pierwszy rzut oka problematyczne, nieintuicyjne. Można temu zaradzić np. porównując ją z maksymalną możliwą wartością MSE dla danego zakresu wartości pikseli. Zależać ona będzie od maksymalnej możliwej wartości piksela, czyli 255 dla obrazów o zakresie [0, 255]. W takim wypadku maksymalna wartość MSE wynosi  $(255^2) = 65025$ . Do potwierdzenia tych założeń wykonano test, w którym porównano dwa obrazy, dla których każdy z pikseli przyjmuje jedną ze skrajnych wartości zakresu [0, 255] - jeden obraz cały biały, a drugi cały czarny. Wartość MSE wyznaczona między tymi obrazami powinna być równocześnie maksymalną możliwą wartością.

Listing 5: Test maksymalnej wartości MSE

```
white = np.array([255, 255, 255])  
black = np.array([0, 0, 0])  
  
image_black = np.array([[black, black, black, black],  
                        [black, black, black, black],  
                        [black, black, black, black],  
                        [black, black, black, black]])  
  
image_white = np.array([[white, white, white, white],  
                       [white, white, white, white],  
                       [white, white, white, white],  
                       [white, white, white, white]])  
  
mse = calculate_mse(image_black, image_white)  
  
print("Błąd średniokwadratowy (MSE):", mse)
```



The screenshot shows a terminal window with a dark background. At the top, there are four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is selected. The terminal output shows the command prompt 'PS C:\Users\Szymon Wichrowski\Desktop\stud' followed by the command 'thon312/python.exe "c:/Users/Szymon Wichr'. The output of the command is 'Błąd średniokwadratowy (MSE): 65025.0'.

Rysunek 6: Wynik testu maksymalnej wartości MSE wyświetlony w terminalu

Wynik testu potwierdził przyjętą maksymalną wartość MSE.

```
38 CrAfterDown = downsampling(Cr, 1)
39 CbAfterDown = downsampling(Cb, 1)
40
41 CrAfterUp = upsampling(CrAfterDown, 1)
42 CbAfterUp = upsampling(CbAfterDown, 1)
43
44 againYCrCb = cv2.merge([Y, CrAfterUp, CbAfterUp])
45
46 againRGB = cv2.cvtColor(againYCrCb, cv2.COLOR_YCrCb2RGB)
47
48 max_possible_mse = (255.0**2)
49
50 if(againRGB.size != primaryImageRGB.size):
51     print("Obrazy mają różną liczbę pikseli!!! Koniec programu")
52 else:
53     print("MSE = " + str("{:.2f}".format(doMSE(primaryImageRGB, againRGB))))
54     print("Maksymalne możliwe MSE:", max_possible_mse)
--
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\Szymon Wichrowski\Desktop\studia\6 semestr\inżynieria obrazow\lab2\IO\_lab  
thon312/python.exe" "c:/Users/Szymon Wichrowski/Desktop/studia/6 semestr/inżynieria o  
MSE = 103.99  
Maksymalne możliwe MSE: 65025.0

Rysunek 7: Obliczenie MSE między obrazami z przed i po symulacji transmisji w systemie DVB

Obliczono błąd średniokwadratowy między obrazami z rysunku nr 3 - wyniósł on 103,99 (zaokrąglając do dwóch miejsc po przecinku). Porównując otrzymaną wartość MSE z maksymalną możliwą do uzyskania wartością można stwierdzić, że różnica między obrazami jest niska.

## 4 Implementacja obsługi formatu PPM

Zadanie polega na zaimplementowaniu obsługi formatu PPM w zakresie odczytywania i zapisywania obrazu.

Na początek należy utworzyć dowolny szkic RGB i zapisać go w wariantach P3 oraz P6. W tym celu można utworzyć klasę *PPM*, reprezentującą taki szkic w formacie PPM, której metody będą odpowiadać za utworzenie szkicu i zapisanie go w odpowiednim wariancie.

Listing 6: Implementacja klasy *PPM*

```
class PPM:
    def __init__(self, width, height, color_range):
        self.width = width
        self.height = height
        self.color_range = color_range
        self.pixels = []

    def add_pixel(self, x, y, color_rgb):
        self.pixels.append((x, y, color_rgb))

    def saveP3(self, filename):
        with open(filename, 'w') as file:
            file.write(f'P3\n{self.width} {self.height}\n{self.color_range}\n')
            for pixel in self.pixels:
                file.write(f'{pixel[2][0]} {pixel[2][1]} {pixel[2][2]}\n')

    def saveP6(self, filename):
        with open(filename, 'wb') as file:
            header = f'P6\n{self.width} {self.height}\n{self.color_range}\n'
            header_bytes = header.encode('utf-8')
            file.write(bytearray(header_bytes))
            for pixel in self.pixels:
                file.write(bytes(pixel[2]))
```

Metoda `add_pixel()` odpowiada za dodawanie kolejnych pikseli do szkicu, natomiast metody `saveP3()` oraz `saveP6()` służą do zapisywania szkicu w odpowiednich wariantach formatu PPM. Dla wariantu P3, najpierw zapisujemy do pliku nagłówek, w którego skład wchodzi informacja o wariancie, wymiary szkicu oraz zakres wartości pikseli, a następnie dla każdego piksela szkicu, w osobnej linii, zapisywane są wartości poszczególnych składowych koloru (RGB - 3 wartości). Dla wariantu P6 również zaczynamy od nagłówka, ale trzeba zwrócić uwagę na to, że nowo powstały plik ma być plikiem binarnym. Używamy funkcji `encode('utf-8')` do przekonwertowania nagłówka na bajty.

Listing 7: Generowanie przykładowego szkicu oraz zapis do plików

```
sketch = PPM(2, 2, 255)
sketch.add_pixel(0, 0, (255, 0, 0))
sketch.add_pixel(0, 1, (0, 255, 0))
sketch.add_pixel(1, 0, (0, 0, 255))
sketch.add_pixel(1, 1, (255, 0, 255))

sketch.saveP3('szkicP3.ppm')
sketch.saveP6('szkicP6.ppm')
```

Otrzymane pliki PPM można wyświetlić za pomocą programu *Gimp*.

```

≡ szkicP3.ppm
1  P3
2  2 2
3  255
4  255 0 0
5  0 255 0
6  0 0 255
7  255 0 255
8

```

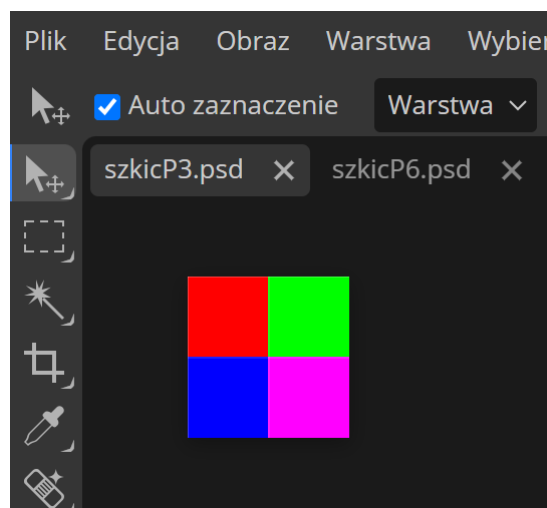
Rysunek 8: Plik *szkicP3.ppm*

```

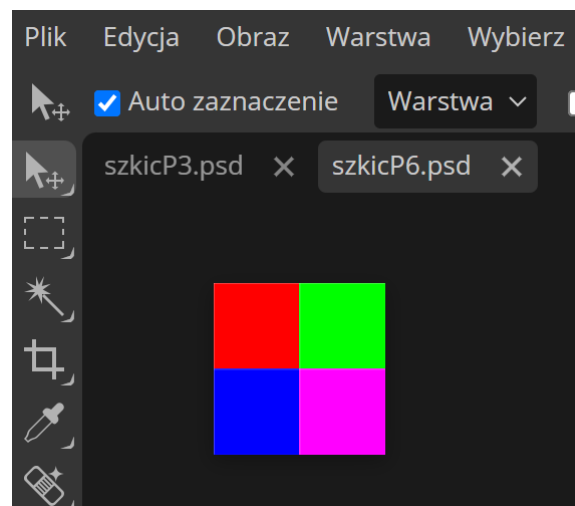
≡ szkicP6.ppm
1  P6
2  2 2
3  255
4  ? NULNULNUL ? NULNULNUL ? ? NUL ?

```

Rysunek 9: Plik *szkicP6.ppm*



Rysunek 10: Plik *szkicP3.ppm* wyświetlony w programie *Gimp*



Rysunek 11: Plik *szkicP6.ppm* wyświetlony w programie *Gimp*

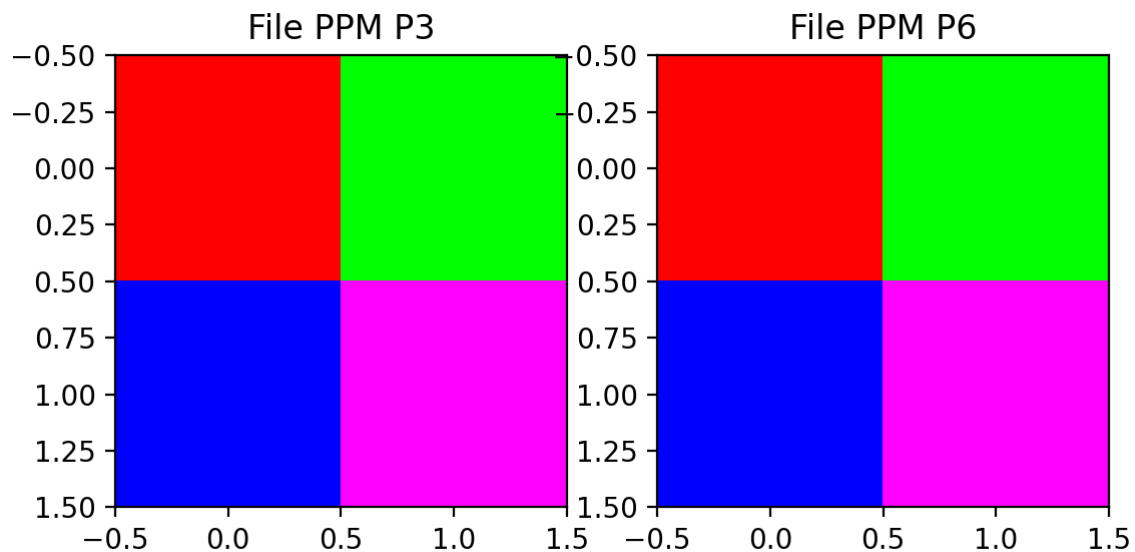


Kolejnym krokiem jest zaimplementowanie procedury odczytania zapisanych szkiców z plików i wyświetlenie ich na ekranie.

Listing 8: Procedura odczytania szkiców z plików *szkicP3.ppm* oraz *szkicP6.ppm*

```
sketch_from_file_P3 = cv2.imread('szkicP3.ppm')
sketch_from_file_P6 = cv2.imread('szkicP6.ppm')
sketch_from_file_P3 = cv2.cvtColor(sketch_from_file_P3, cv2.COLOR_BGR2RGB)
sketch_from_file_P6 = cv2.cvtColor(sketch_from_file_P6, cv2.COLOR_BGR2RGB)

pyplot.subplot(121), pyplot.imshow(sketch_from_file_P3), pyplot.title("File PPM P3")
pyplot.subplot(122), pyplot.imshow(sketch_from_file_P6), pyplot.title("File PPM P6")
pyplot.show()
```



Rysunek 12: Szkice odczytane z plików *szkicP3.ppm* oraz *szkicP6.ppm*

Następnie należy powtórzyć wszystkie kroki dla pobranej fotografii RGB. Procedury zapisu do formatu PPM oraz odczytania z niego obrazów są analogiczne, jak dla szkiców.

Listing 9: Implementacja zapisu obrazów do formatu PPM

```
def saveImage_P3(image, filename):
    height = len(image)
    width = len(image[0])
    color_range = 255
    with open(filename, 'w') as file:
        file.write(f'P3\n{width} {height}\n{color_range}\n')
        for i in range(height):
            for j in range(width):
                file.write(f'{image[i, j, 0]} {image[i, j, 1]} {image[i, j, 2]}\n')

def saveImage_P6(image, filename):
    height = len(image)
    width = len(image[0])
    color_range = 255
    with open(filename, 'wb') as file:
        header = f'P6\n{width} {height}\n{color_range}\n'
        header_bytes = header.encode('utf-8')
        file.write(bytearray(header_bytes))
        for i in range(height):
            for j in range(width):
                file.write(bytes(image[i, j]))
```

```

≡ photoP3.ppm
1  P3
2  2200 1376
3  255
4  91 139 188
5  69 113 158
6  108 149 181
7  108 143 165
8  102 131 145
9  112 139 148
10 102 124 137
11 130 152 166
12 110 130 154

```

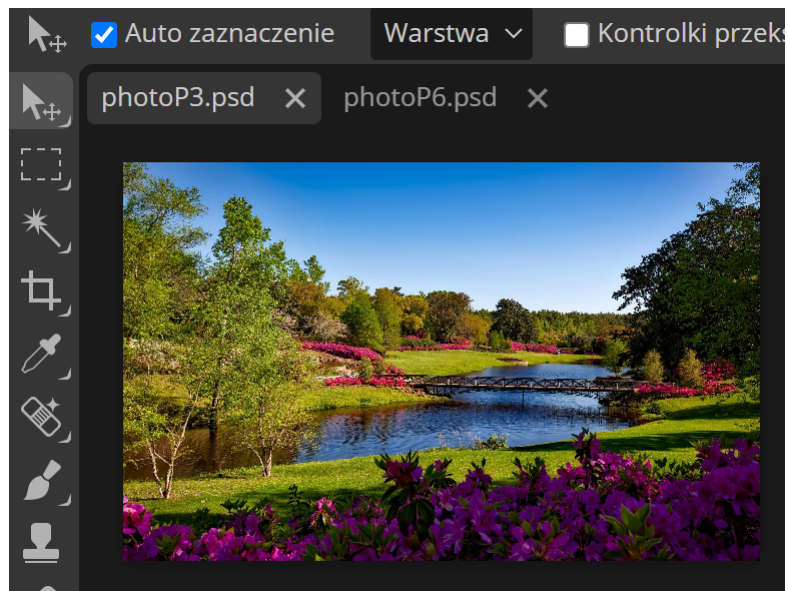
Rysunek 13: Plik *photoP3.ppm*

```

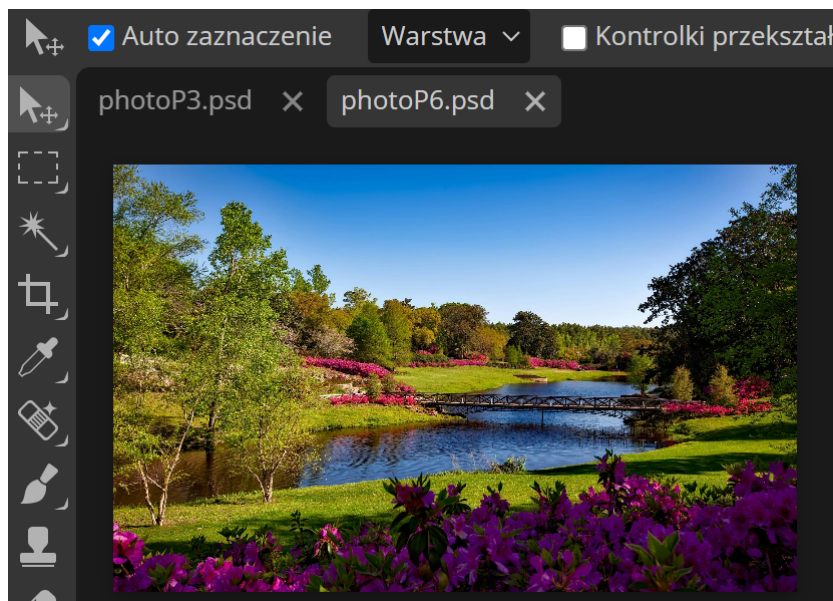
≡ photoP6.ppm
1  P6
2  2200 1376
3  255
4  [??Eq?l??l??f??p??f|???n??Pezrs0>3CBR`IE
5  x?
6  x?
7  x?
8  x?
9  x?
10 x?VTX?VTX?BELU?BELU?ACKt?ACKt?EOTt?ETXs?STXr?NULr?NULt?NUL
11 z?
12 z?
13 z?
14 z?FFZ?FFZ?FFZ?FFZ?FFZ?FFZ?FFZ?FFZ?FFZ?
15 {?

```

Rysunek 14: Plik *photoP6.ppm*



Rysunek 15: Plik *photoP3.ppm* wyświetlony w programie *Gimp*



Rysunek 16: Plik *photoP6.ppm* wyświetlony w programie *Gimp*

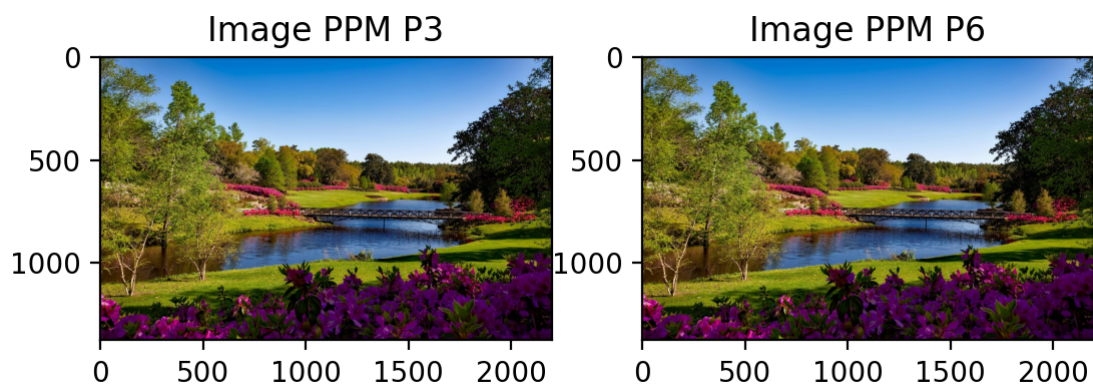
Listing 10: Procedura odczytania obrazów z plików *photoP3.ppm* oraz *photoP6.ppm*

```
primaryImageBGR = cv2.imread("images/krajobraz.jpg")
primaryImageRGB = cv2.cvtColor(primaryImageBGR, cv2.COLOR_BGR2RGB)

saveImage_P3(primaryImageRGB, 'photoP3.ppm')
saveImage_P6(primaryImageRGB, 'photoP6.ppm')

image_from_file_P3 = cv2.imread('photoP3.ppm')
image_from_file_P6 = cv2.imread('photoP6.ppm')
image_from_file_P3 = cv2.cvtColor(image_from_file_P3, cv2.COLOR_BGR2RGB)
image_from_file_P6 = cv2.cvtColor(image_from_file_P6, cv2.COLOR_BGR2RGB)

pyplot.subplot(221), pyplot.imshow(image_from_file_P3), pyplot.title("Image PPM P3")
pyplot.subplot(222), pyplot.imshow(image_from_file_P6), pyplot.title("Image PPM P6")
pyplot.show()
```



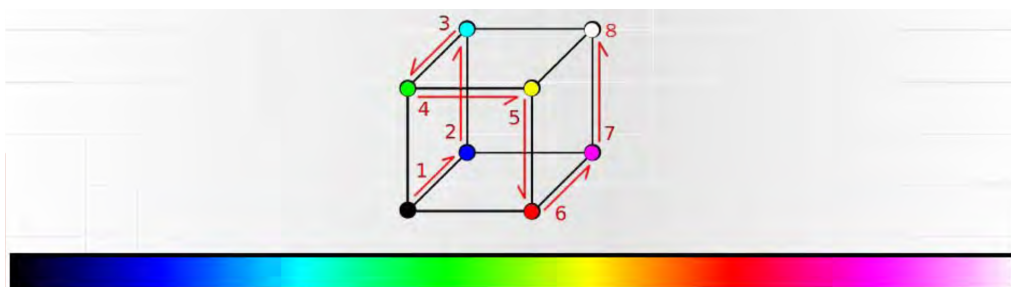
Rysunek 17: Obrazy odczytane z plików *photoP3.ppm* oraz *photoP6.ppm*

Na koniec warto porównać rozmiary zbiorów dla fotografii RGB. Plik tekstowy, wariant P3 ma 31509 KB, a plik binarny, wariant P6 ma tylko 8869 KB. Udowadnia to, że zapisanie danych obrazu w postaci binarnej zmniejsza rozmiar pliku.

 photoP3	31 509 KB
 photoP6	8 869 KB

Rysunek 18: Rozmiary plików *photoP3.ppm* oraz *photoP6.ppm*

## 5 Schemat przestrzeni barw RGB w formacie PPM



Rysunek 19: Wzorzec ze schematem przejść w przestrzeni RGB oraz spektrum tych przejść

Zadanie polega na umieszczeniu schematu przestrzeni barw RGB w formacie PPM. Ograniczając się do jednego wariantu PPM, należy zaimplementować spektrum przejść według wzorca z rysunku nr 19 oraz dodatkowo dodać spektrum przejścia (1-8).

Tak, jak w poprzednim zadaniu można utworzyć klasę *PPM*, która będzie odpowiadała za generowanie szkicu i zapisywanie go w formacie PPM. Należy zaimplementować metody `setRainbow()` oraz `setBlackAndWhite()`, które przy gotowej już metodzie `add_pixel()` (listing nr 6), będą miały wygenerować szkice odpowiadające spektrum przejść (1,...,8) i (1-8).

Listing 11: Implementacja metod do szkicowania spektrum przejść (1,...,8) i (1-8)

```
def setRainbow(self):
    transition = (self.width - 1) / 7
    term = self.color_range / transition
    term = int(term)
    for i in range(self.height):
        red = 0
        green = 0
        blue = 0
        self.add_pixel(i, 0, (red, green, blue))
        for j in range(self.width - 1):
            if j < transition:
                blue += term
            elif j >= transition and j < transition*2:
                green += term
            elif j >= transition*2 and j < transition*3:
                blue -= term
            elif j >= transition*3 and j < transition*4:
                red += term
            elif j >= transition*4 and j < transition*5:
                green -= term
            elif j >= transition*5 and j < transition*6:
                blue += term
            elif j >= transition*6 and j < transition*7:
                green += term
            else:
                red = 0
                green = 0
                blue = 0
            self.add_pixel(i, j+1, (red, green, blue))

def setBlackAndWhite(self):
    for i in range(self.height):
        for j in range(self.width):
            self.add_pixel(i, j, (j, j, j))
```

Metoda `setRainbow()` iteruje po kolejnych kolumnach tablicy pikseli. W zależności od aktualnego numeru kolumny, inkrementuje lub dekrementuje wartość odpowiedniego kanału: *red*, *green* albo *blue*. Metoda `setBlackAndWhite()` również iteruje po kolumnach tablicy pikseli, ale przy każdej kolejnej iteracji inkrementowane są wszystkie składowe wektora koloru. Zapis do pliku i wyświetlanie wygląda identycznie jak do tej pory (listingi nr 7 i nr 8).

```

≡ spektrum-teczowe.ppm
1    P3
2    1786 100
3    255
4    0 0 0
5    0 0 1
6    0 0 2
7    0 0 3
8    0 0 4
9    0 0 5
10   0 0 6
11   0 0 7
12   0 0 8
13   0 0 9
14   0 0 10

```

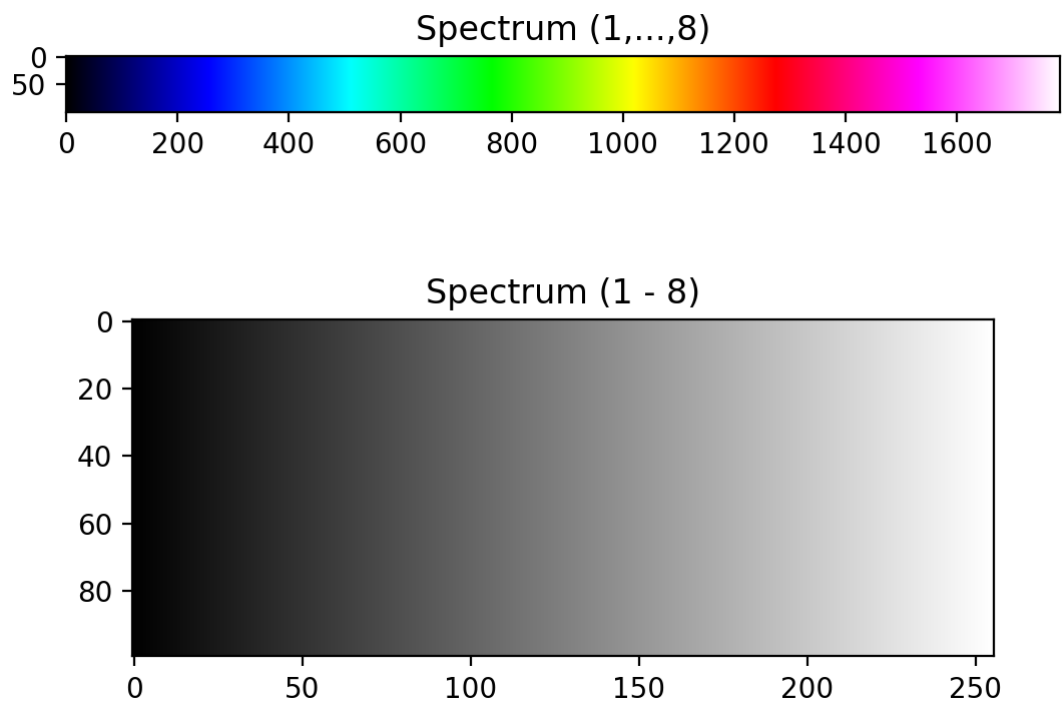
Rysunek 20: Plik *spektrum-teczowe.ppm*

```

≡ spektrum-czarno-biale.ppm
1    P3
2    256 100
3    255
4    0 0 0
5    1 1 1
6    2 2 2
7    3 3 3
8    4 4 4
9    5 5 5
10   6 6 6
11   7 7 7
12   8 8 8
13   9 9 9
14   10 10 10
15   11 11 11
16   12 12 12

```

Rysunek 21: Plik *spektrum-czarno-biale.ppm*



Rysunek 22: Wizualizacja przestrzeni RGB - spektrum przejść (1,...,8) i (1-8)

## 6 Źródła

- dokumentacja biblioteki OpenCV
- dokumentacja formatu PPM
- Wikipedia - Netpbm