

UNIWERSYTET KARDYNAŁA STEFANA WYSZYŃSKIEGO  
W WARSZAWIE

WYDZIAŁ MATEMATYCZNO-PRZYRODNICZY  
SZKOŁA NAUK ŚCISŁYCH

Katarzyna Mitrus

Michał Słotwiński

Wprowadzenie do Przetwarzania Obrazów

Sprawozdanie z laboratorium

Prowadzący:  
prof. Wojciech Mokrzycki

Warszawa, 2018

# Spis treści

<b>Spis rysunków</b> . . . . .	4
<b>Rozdział 1. Wstęp</b> . . . . .	8
1.1 Specyfikacja wykorzystanego formatu obrazu . . . . .	8
1.2 Instrukcja obsługi programu . . . . .	8
<b>Rozdział 2. Operacje ujednoliciania obrazów</b> . . . . .	9
<b>Rozdział 3. Operacje sumowania arytmetycznego obrazów szarych</b> . . . . .	10
3.1 Sumowanie (określonej) stałej z obrazem . . . . .	10
3.2 Sumowanie dwóch obrazów . . . . .	12
3.3 Mnożenie obrazu przez zadaną liczbę . . . . .	15
3.4 Mnożenie obrazu przez inny obraz . . . . .	17
3.5 Mieszanie obrazów z określonym współczynnikiem . . . . .	20
3.6 Potegowanie obrazu (z zadanaą potegą) . . . . .	22
3.7 Dzielenie obrazu przez (zadaną) liczbę . . . . .	24
3.8 Dzielenie obrazu przez przez inny obraz . . . . .	26
3.9 Pierwiastkowanie obrazu . . . . .	29
3.10 Logarytmowanie obrazu . . . . .	31
<b>Rozdział 4. Operacje sumowania arytmetycznego obrazów barwowych</b> . . . . .	34
4.1 Sumowanie (określonej) stałej z obrazem . . . . .	34
4.2 Sumowanie dwóch obrazów . . . . .	36
4.3 Mnożenie obrazu przez zadaną liczbę . . . . .	40
4.4 Mnożenie obrazu przez inny obraz . . . . .	42
4.5 Mieszanie obrazów z określonym współczynnikiem . . . . .	46
4.6 Potegowanie obrazu . . . . .	48
4.7 Dzielenie obrazu przez (zadaną) liczbę . . . . .	51
4.8 Dzielenie obrazu przez inny obraz . . . . .	53
4.9 Pierwiastkowanie obrazu . . . . .	57
4.10 Logarytmowanie obrazu . . . . .	59
<b>Rozdział 5. Operacje geometryczne na obrazie</b> . . . . .	63
5.1 Przemieszczenie obrazu o zadany wektor . . . . .	63
5.2 Jednorodne skalowanie obrazu . . . . .	64
5.3 Niejednorodne skalowanie obrazu . . . . .	66
5.4 Obracanie obrazu o dowolny kąt . . . . .	68
5.5 Symetrie względem osi układu . . . . .	71
5.6 Symetrie względem zadanej prostej . . . . .	73

<i>Spis treści</i>	3
5.7    Wycinanie fragmentów obrazu . . . . .	75
5.8    Kopiowanie fragmentów obrazów . . . . .	76
<b>Rozdział 6. Operacje na histogramie obrazu szarego . . . . .</b>	78
<b>Rozdział 7. Operacje na histogramie obrazu barwowego . . . . .</b>	79
<b>Rozdział 8. Operacje morfologiczne na obrazach binarnych . . . . .</b>	80
8.1    Okrawanie (erozja) . . . . .	80
8.2    Nakładanie (dylatacja) . . . . .	82
8.3    Otwarcie . . . . .	84
8.4    Zamknięcie . . . . .	86
<b>Rozdział 9. Operacje morfologiczne na obrazach szarych . . . . .</b>	89
9.1    Okrawanie (erozja) . . . . .	89
9.2    Nakładanie (dylatacja) . . . . .	90
9.3    Otwarcie . . . . .	92
9.4    Zamknięcie . . . . .	94
<b>Rozdział 10. Filtrowanie liniowe i nielinowe . . . . .</b>	97
<b>Rozdział 11. Podsumowanie . . . . .</b>	98
<b>Bibliografia . . . . .</b>	99

# Spis rysunków

3.1 (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji . . . . .	10
3.2 (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji . . . . .	11
3.3 (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji . . . . .	13
3.4 (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji . . . . .	13
3.5 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=50, obraz po normalizacji . . . . .	15
3.6 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=100, obraz po normalizacji . . . . .	16
3.7 (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	18
3.8 (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	18
3.9 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.5$ , poniżej obraz wynikowy po normalizacji . . . . .	20
3.10 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$ , poniżej obraz wynikowy po normalizacji . . . . .	21
3.11 (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=2$ , obraz po normalizacji . . . . .	22
3.12 (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=3$ , obraz po normalizacji . . . . .	23
3.13 (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji . . . . .	25
3.14 (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji . . . . .	25
3.15 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	27
3.16 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	27
3.17 (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ( $\alpha=1/2$ ), obraz po normalizacji . . . . .	29
3.18 (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ( $\alpha=1/3$ ), obraz po normalizacji . . . . .	29

3.19 (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji . . . . .	31
3.20 (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji . . . . .	32
4.1 (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji . . . . .	34
4.2 (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji . . . . .	35
4.3 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji . . . . .	37
4.4 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji . . . . .	38
4.5 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 50, obraz po normalizacji . . . . .	40
4.6 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 100, obraz po normalizacji . . . . .	41
4.7 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	43
4.8 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	44
4.9 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.5$ , poniżej obraz wynikowy po normalizacji . . . . .	46
4.10 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$ , poniżej obraz wynikowy po normalizacji . . . . .	47
4.11 (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=2$ , obraz po normalizacji . . . . .	49
4.12 (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=3$ , obraz po normalizacji . . . . .	49
4.13 (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji . . . . .	51
4.14 (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji . . . . .	52
4.15 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	54
4.16 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji . . . . .	55
4.17 (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ( $\alpha=1/2$ ), obraz po normalizacji . . . . .	57
4.18 (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ( $\alpha=1/3$ ), obraz po normalizacji . . . . .	57
4.19 (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji . . . . .	59
4.20 (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji . . . . .	60
5.1 (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [40, 70] . . . . .	63

5.2	(Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [200, 100] . . . . .	64
5.3	(Od lewej) Obraz wejściowy, obraz skalowania jednorodnym ze współczynnikiem $S=1.5$ , obraz po interpolacji . . . . .	65
5.4	(Od lewej) Obraz wejściowy, obraz skalowania jednorodnym ze współczynnikiem $S=2$ , obraz po interpolacji . . . . .	65
5.5	(Od lewej) Obraz wejściowy, obraz skalowania niejednorodnym ze współczynnikiem $S_x=2$ oraz współczynnikiem $S_y=1$ , obraz po interpolacji . . . . .	67
5.6	(Od lewej) Obraz wejściowy, obraz skalowania niejednorodnym ze współczynnikiem $S_x=1$ oraz współczynnikiem $S_y=2$ , obraz po interpolacji . . . . .	67
5.7	(Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 40 stopni, obraz po interpolacji . . . . .	69
5.8	(Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 110 stopni, obraz po interpolacji . . . . .	69
5.9	(Od lewej) Obraz wejściowy, obraz symetryczny względem osi X . . . . .	71
5.10	(Od lewej) Obraz wejściowy, obraz symetryczny względem osi X . . . . .	71
5.11	(Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y . . . . .	72
5.12	(Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y . . . . .	72
5.13	(Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu . . . . .	73
5.14	(Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu . . . . .	73
5.15	(Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu . . . . .	73
5.16	(Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu . . . . .	74
5.17	(Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$ . . . . .	75
5.18	(Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$ . . . . .	75
5.19	(Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$ , Skopiowany fragment (151x426) . . . . .	76
5.20	(Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$ , Skopiowany fragment (201x201) . . . . .	76
8.1	(Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji) . . . . .	81
8.2	(Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji) . . . . .	81
8.3	(Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji) . . . . .	82
8.4	(Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji) . . . . .	83
8.5	(Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja) . . . . .	84
8.6	(Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja) . . . . .	84
8.7	(Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja) . . . . .	86

8.8 (Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja) . . . . .	86
9.1 (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji okrawania (erozji) . . . . .	89
9.2 (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji okrawania (erozji) . . . . .	90
9.3 (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji nakładania (dylatacji) . . . . .	91
9.4 (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji nakładania (dylatacji) . . . . .	91
9.5 (Od lewej) Obraz wejściowy (256x256), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja) . . . . .	92
9.6 (Od lewej) Obraz wejściowy (512x512), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja) . . . . .	93
9.7 (Od lewej) Obraz wejściowy (256x256), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja) . . . . .	94
9.8 (Od lewej) Obraz wejściowy (512x512), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja) . . . . .	95

## Rozdział 1

# Wstęp

Laboratoria... [1]

## 1.1 Specyfikacja wykorzystanego formatu obrazu

## 1.2 Instrukcja obsługi programu

## **Rozdział 2**

# **Operacje ujednolicania obrazów**

1. ujednolicenie obrazów szarych geometryczne (liczba wierszy i kolumn piksli)
2. ujednolicenie obrazów szarych rozdzielczościowe (w rastrze)
3. ujednolicenie obrazów RGB geometryczne (liczba wierszy i kolumn piksli)
4. ujednolicenie obrazów RGB rozdzielczościowe (w rastrze)

## Rozdział 3

# Operacje sumowania arytmetycznego obrazów szarych

Arytmetyczne operacje między pikslami  $p$  i  $q$  dwóch obrazów są używane w wielu dzia-łach przetwarzania obrazów. Przeprowadzane się je wykonując działania na pojedynczych pikslach i są uwarunkowane wymaganiami zależnymi od typu operacji. Po operacjach arytmetycznych zwykle niezbędna jest normalizacja. W przedstawionych zadaniach do normalizacji wykorzystano wzór:

$$f_{norm} = Z_{rep}[(f - f_{min}) / (f_{max} - f_{min})]$$

### 3.1 Sumowanie (określonej) stałej z obrazem

Algorytm sumowania obrazu szarego z określona stałą polega na dodaniu do każdej wartości pojedynczego piksla stałej liczby. Po operacji sumowania następuje normalizacja obrazu.

1. Policz sumy wartości kazdego piksla ze stałą (*const*).
2. Jeżeli jedna z tych sum jest większa niż 255 to:
3. Wybierz największą sumę  $Q_{max}$  i policz  $D_{max}$  ze wzoru:  $D_{max}[i, j] = (Q_{max}[i, j] - 255)$
4. Oblicz  $X = D_{max}/255$
5. Policz sumy ze wzoru

$$Q[i, j] = P[i, j] - (P[i, j] * X) + const - (const * X)$$



Rysunek 3.1: (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji



Rysunek 3.2: (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji

Listing 3.1: Sumowanie obrazu szarego ze stałą

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):
        # Obliczanie sumy
        L = int(image_matrix[x][y]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < L:
            Q_max = L

# Sprawdzenie czy przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255)

```

```

# Obliczenie sumy z uwzględnieniem zakresu
for y in range(height):
    for x in range(width):
        L = (image_matrix[x][y] - (image_matrix[x][y] * X)) + (
            const - (const * X))

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y] = math.ceil(L)

        # Poszukiwanie minimum i maksimum
        if f_min > L:
            f_min = L
        if f_max < L:
            f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) /
            (f_max - f_min))

```

## 3.2 Sumowanie dwóch obrazów

Algebraiczne sumowanie obrazów  $f$  i  $f'$  jest określone jedynie dla obrazów o tych samych wymiarach  $M \times N$  i strukturze ich macierzy. Dodawanie obrazów jest użyteczne w uśrednianiu obrazów, wykonywanym w celu zredukowania na nich szumu. Algorytm sumowania obrazu z obrazem polega na dodaniu do wartości piksla z pierwszego obrazu, wartości odpowiadającego piksla z drugiego obrazu. Po operacji sumowania następuje normalizacja obrazu.

1. Policz sumy wartości kazdego piksla obrazu pierwszego  $P1[i,j]$  z pikslem obrazu drugiego  $P2[i,j]$ .
2. Jeżeli jedna z tych sum jest większa niż 255 to:
3. Wybierz największą sumę  $Q_{max}$  i policz  $D_{max}$  ze wzoru:  $D_{max}[i,j] = (Q_{max}[i,j] - 255)$
4. Oblicz  $X = D_{max}/255$
5. Policz sumy ze wzoru

$$Q[i,j] = P1[i,j] - (P1[i,j] * X) + P2[i,j] - (P2[i,j] * X)$$



Rysunek 3.3: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.4: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.2: Sumowanie obrazów szarych

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])

        # Poszukiwanie maksimum
        if Q_max < L:
            Q_max = L

# Sprawdzenie czy przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255)

# Obliczenie sumy z uwzglednieniem zakresu
for y in range(height):
    for x in range(width):
        L = (image1_matrix[x][y] - (image1_matrix[x][y] * X)) +
            (image2_matrix[x][y] - (image2_matrix[x][y] * X))

        # Zaokroglenie do najblzszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum

```

```

if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

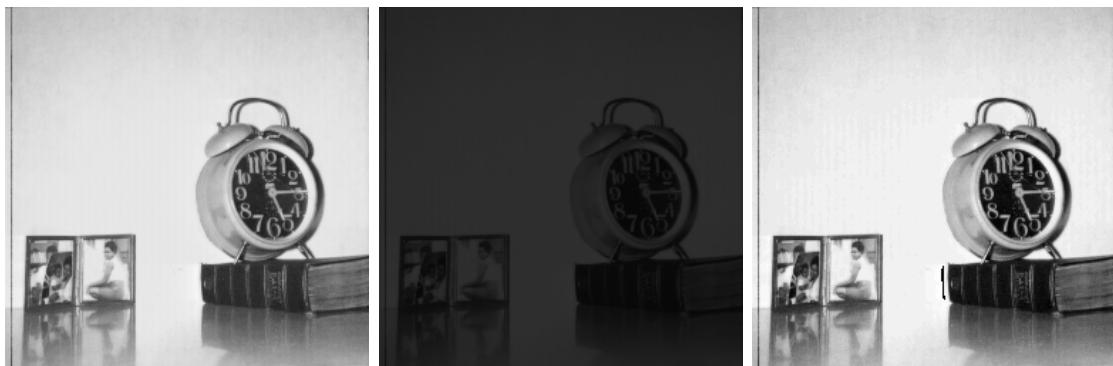
# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min
            ) / (f_max - f_min))

```

### 3.3 Mnożenie obrazu przez zadaną liczbę

Mnożenie obrazu  $f$  przez skalar wykonuje się mnożąc każdy element obrazu  $f_{i,j}$  (wartość funkcji obrazowej piksla) przez ten skalar.

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa piksela  $P_1[i, j]$  ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości stałej.
3. W przeciwnym przypadku, jeżeli składowa barwy piksla  $P_1[i, j]$  ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 3.5: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=50, obraz po normalizacji



Rysunek 3.6: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=100, obraz po normalizacji

Listing 3.3: Mnożenie obrazu szarego przez zadaną liczbę

```
iimage1_matrix = self.im1
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

# Mnożenie
for y in range(height):
    for x in range(width):

        L = int(image1_matrix[x][y])
        if L == 255:
            L = const
        elif L == 0:
            L = 0
        else:
            L = (int(image1_matrix[x][y]) * const)/255

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y] = math.ceil(L)
```

```

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

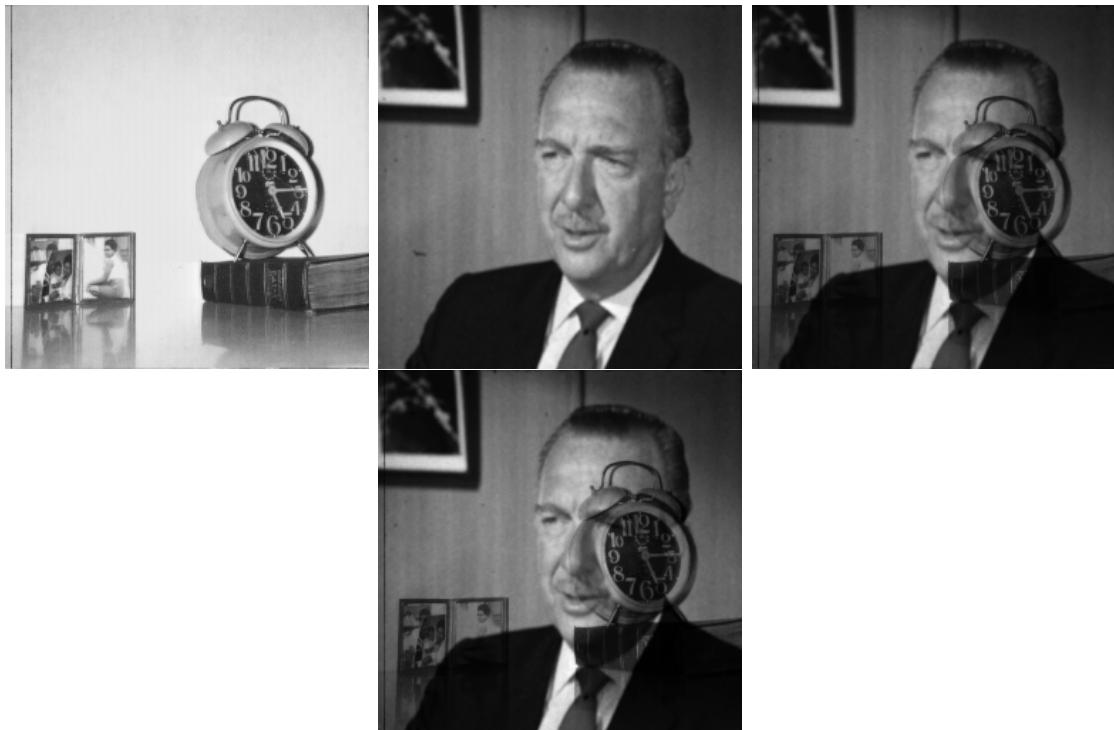
# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))

```

## 3.4 Mnożenie obrazu przez inny obraz

Mnożenie obrazu  $f$  przez inny obraz wykonuje się mnożąc każdy element obrazu  $P1_{i,j}$  (wartość funkcji obrazowej piksla) przez odpowiadającego piksla drugiego obrazu  $P2_{i,j}$

1. Weź dwa identycznych rozmiarów obrazy  $P_1$  i  $P_2$ .
2. Dla wszystkich pikseli w obrazie wykonaj:
3. Jeżeli składowa piksela  $P_1[i, j]$  ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości składowej  $P_2[i, j]$ .
4. W przeciwnym przypadku, jeżeli składowa piksela  $P_1[i, j]$  ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
5. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 3.7: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.8: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.4: Mnożenie obrazu szarego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        L = int(image1_matrix[x][y])
        if L == 255:
            L = image2_matrix[x][y]
        elif L == 0:
            L = 0
        else:
            L = (int(image1_matrix[x][y]) * int(image2_matrix[x][y])) / 255

# Zaokroglenie do najblzszej wartosci calkowitej z gory
# i przypisanie wartosci
result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))

```

### 3.5 Mieszanie obrazów z określonym współczynnikiem

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami  $\alpha$  i  $(1 - \alpha)$ , odpowiednio, wg wzoru:

$$f_m = f\alpha + f^I(1 - \alpha),$$

gdzie  $\alpha \in [0, 1]$ . Płynna zmiana parametru  $\alpha$  w przedziale  $[0, 1]$  powoduje efekt przecho- dzenia obrazu  $f^I$  w obraz  $f$ .

1. Weź dwa identycznych rozmiarów obrazy  $P_1$  i  $P_2$ .
2. Określ współczynnik mieszania  $\alpha$  wyrażony jako liczba rzeczywista z zakresu  $<0, 1>$ ; 0 reprezentuje pewną przezroczystość, 1 - nieprzezroczystości.
3. Dla wszystkich pikseli w obrazach wejściowych wykonuj  $Q(i, j) = \alpha * P_1(i, j) + (1 - \alpha) * P_2(i, j)$



Rysunek 3.9: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mie- szania obrazów ze współczynnikiem  $\alpha=0.5$ , poniżej obraz wynikowy po normalizacji



Rysunek 3.10: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem  $\alpha=0.8$ , poniżej obraz wynikowy po normalizacji

Listing 3.5: Mieszanie obrazów szarych z określonym współczynnikiem

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        L = float(image1_matrix[x][y]) * alfa + (1-alfa) *
            float(image2_matrix[x][y])

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory

```

```

# i przypisanie wartosci
result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

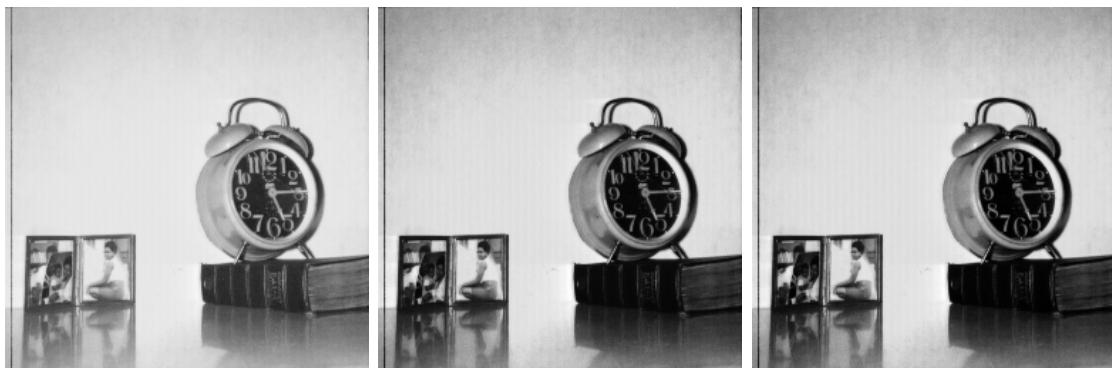
# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))

```

### 3.6 Potęgowanie obrazu (z zadaną potęgą)

Potęgowanie obrazu jest szczególnym przypadkiem operacji mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

$$f_m = 255 \left( \frac{f(x,y)}{f_{max}} \right)^\alpha, \alpha > 0$$



Rysunek 3.11: (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi  $\alpha=2$ , obraz po normalizacji



Rysunek 3.12: (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi  $\alpha=3$ , obraz po normalizacji

Listing 3.6: Potęgowanie obrazu szarego (z zadana potęgą)

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])

        # Poszukiwanie maksimum
        if f_img_max < L:
            f_img_max = L

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])
        if L == 255:
            L = 255
        elif L == 0:

```

```

L = 0
else:
    L = math.pow(int(image_matrix[x][y]) / f_img_max,
                 alfa) * 255

# Zaokrąglenie do najbliższej wartości całkowitej z
# gory
# i przypisanie wartości
result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)
                                    / (f_max - f_min))

```

### 3.7 Dzielenie obrazu przez (zadaną) liczbę

Dzielenie obrazów stosuje się w celu korekcji cieniowania między poziomami szarości.

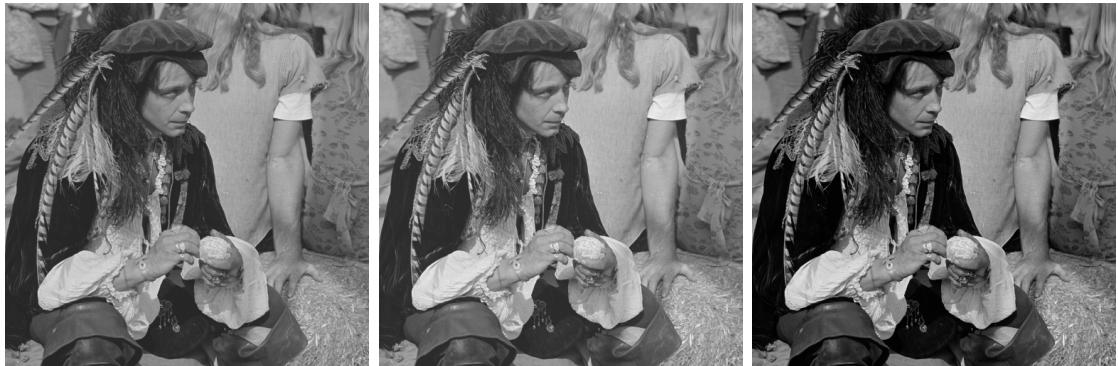
1. Dla wszystkich pikseli w tych obrazach wykonaj:
2. Policz sumy pikseli ze stałą.
3. Wybierz największą sumę  $Q_{max}$  i policz równania:

$$Q[i, j] = (S * 255) / Q_{max},$$

Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 3.13: (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji



Rysunek 3.14: (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji

Listing 3.7: Dzielenie obrazu szarego przez (zadaną) liczbę

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):
        
```

```

L = int(image_matrix[x][y]) + int(const)

# Poszukiwanie maksimum
if Q_max < L:
    Q_max = L

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y]) + int(const)
        Q_L = (L * 255) / Q_max

        # Zaokroglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y] = math.ceil(Q_L)

        # Poszukiwanie minimum i maksimum
        if f_min > Q_L:
            f_min = Q_L
        if f_max < Q_L:
            f_max = Q_L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)
                                   / (f_max - f_min))

```

### 3.8 Dzielenie obrazu przez inny obraz

1. Weź dwa identycznych rozmiarów obrazy  $P1$  i  $P2$
2. Dla wszystkich pikseli w tych obrazach wykonaj:
3. Policz sumy pikseli ze stałą.
4. Wybierz największą sumę  $Q_{max}$  i policz równania:  

$$Q[i, j] = (S * 255)/Q_{max},$$
 Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 3.15: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.16: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.8: Dzielenie obrazu szarego przez przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):
        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])

        # Poszukiwanie maksimum
        if Q_max < L:
            Q_max = L

for y in range(height):
    for x in range(width):

        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])
        Q_L = (L * 255) / Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(Q_L)

        # Poszukiwanie minimum i maksimum
        if f_min > Q_L:
            f_min = Q_L
        if f_max < Q_L:
            f_max = Q_L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)

```

```

for y in range( height ):
    for x in range( width ):
        norm_matrix [x] [y] = 255 * (( result_matrix [x] [y] - f_min
            ) / (f_max - f_min))
    
```

## 3.9 Pierwiastkowanie obrazu

Pierwiastkowanie obrazu jest szczególnym przypadkiem operacji potęgowania obrazów, gdzie wykładnikiem jest ułamek. Aby uniknąć wykroczenia poza zakres, skorzystano ze znalezionego wzoru [2]:

$$f_m = 255 \left( \frac{f(x,y)}{f_{max}} \right)^\alpha, \alpha > 0$$



Rysunek 3.17: (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ( $\alpha=1/2$ ), obraz po normalizacji



Rysunek 3.18: (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ( $\alpha=1/3$ ), obraz po normalizacji

Listing 3.9: Pierwiastkowanie obrazu szarego

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereokosc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

alfa = 1/deg # Zamiana stopnia pierwiastka na ulamek

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])

        # Poszukiwanie maksimum
        if f_img_max < L:
            f_img_max = L

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])
        if L == 255:
            L = 255
        elif L == 0:
            L = 0
        else:
            L = math.pow(int(image_matrix[x][y]) / f_img_max,
                        alfa) * 255

        # Zaokroglenie do najblizszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(L)

        # Poszukiwanie minimum i maksimum
        if f_min > L:
            f_min = L
        if f_max < L:

```

$$f_{\max} = L$$

```
# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))
```

## 3.10 Logarytmowanie obrazu

Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Wykorzystano wzór z normalizacją /citeWykRat:

$$f_m = 255 \left( \frac{\log(1 + f(x, y))}{\log(1 + f_{\max})} \right)$$

Przesunięcie funkcji obrazowej  $f$  do góry o 1 przed jej logarytmowaniem wynika z nieokreśloności logarytmu w zerze. Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. .



Rysunek 3.19: (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji



Rysunek 3.20: (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji

Listing 3.10: Logarytmowanie obrazu szarego

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])

        # Poszukiwanie maksimum
        if f_img_max < L:
            f_img_max = L

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])

        if L == 0:
            L = 0

```

```
else:  
    L = (math.log(1 + L) / math.log(1 + f_img_max)) *  
        255  
  
    # Zaokroglenie do najblizszej wartosci calkowitej z  
    # gory  
    # i przypisanie wartosci  
    result_matrix[x][y] = math.ceil(L)  
  
    # Poszukiwanie minimum i maksimum  
    if f_min > L:  
        f_min = L  
    if f_max < L:  
        f_max = L  
  
# Normalizacja  
norm_matrix = np.zeros((width, height), dtype=np.uint8)  
for y in range(height):  
    for x in range(width):  
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)  
            / (f_max - f_min))
```

## Rozdział 4

# Operacje sumowania arytmetycznego obrazów barwowych

Arytmetyczne operacje na obrazach barwowych przeprowadza się wykonując działania na pojedynczych pikslach i są uwarunkowane wymaganiami zależnymi od typu operacji. W przedstawionych zadaniach poruszamy się w przestrzeni barw RGB, a do normalizacji wykorzystano wzór:

$$f_{norm} = Z_{rep}[(f - f_{min}) / (f_{max} - f_{min})]$$

## 4.1 Sumowanie (określonej) stałej z obrazem

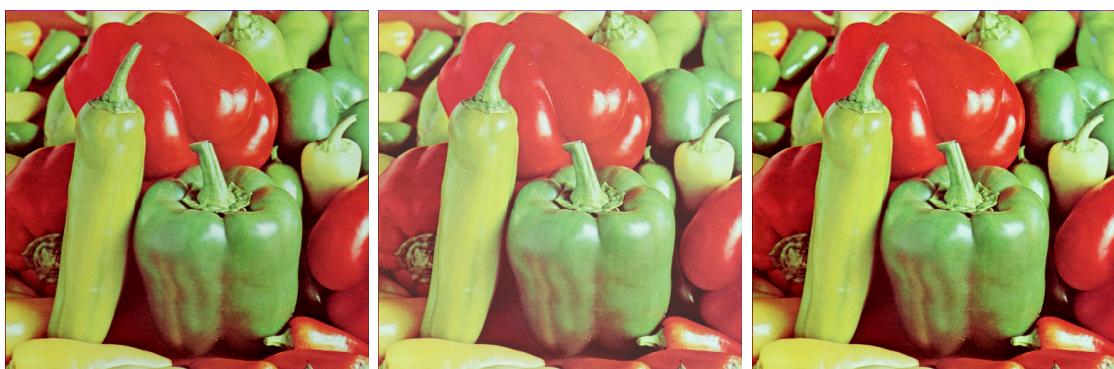
Algorytm sumowania obrazu barwowego z określona stałą polega na dodaniu do każdej składowej barwowej pojedynczego piksla stałej liczby. Po operacji sumowania obraz poddawany jest normalizacji.

1. Policz sumy wartości kazdego piksla ze stałą (*const*).
2. Jeżeli jedna z tych sum jest większa niż 255 to:  
3. Wybierz największą sumę  $Q_{max}$  i policz  $D_{max}$  ze wzoru:  $D_{max}[i, j] = (Q_{max}[i, j] - 255)$
4. Oblicz  $X = D_{max}/255$
5. Policz sumy ze wzoru

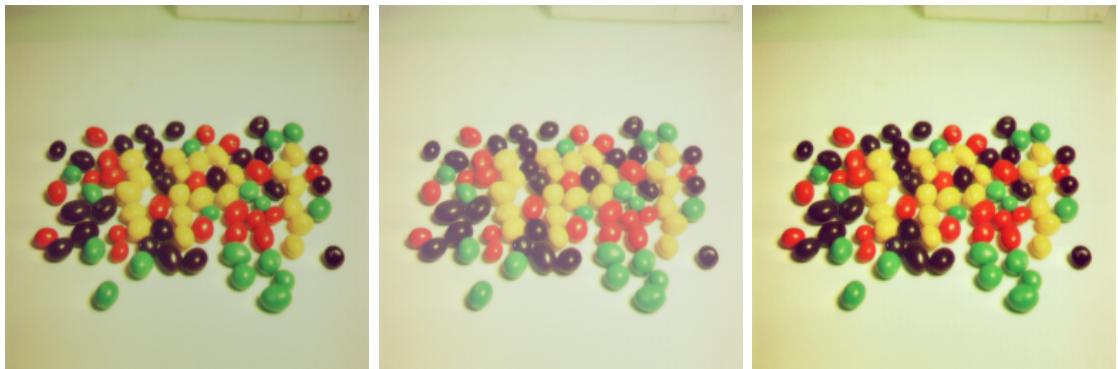
$$Q_R[i, j] = P_R[i, j] - (P_R * X) + const - (const * X) - 1$$

$$Q_G[i, j] = P_G[i, j] - (P_G * X) + const - (const * X) - 1$$

$$Q_B[i, j] = P_B[i, j] - (P_B * X) + const - (const * X) - 1$$



Rysunek 4.1: (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji



Rysunek 4.2: (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji

Listing 4.1: Sumowanie obrazu barwowego ze stałą

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R = int(image_matrix[x][y][0]) + int(const)
        G = int(image_matrix[x][y][1]) + int(const)
        B = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R, G, B]):
            Q_max = max([R, G, B])

# Sprawdzenie czy maksimum przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255

```

```

X = (D_max/255) # Obliczenie proporcji

# Obliczenie sum z uwzględnieniem zakresu
for y in range(height):
    for x in range(width):
        R = (image_matrix[x][y][0] - (image_matrix[x][y][0] * X
            )) + (const - (const * X))
        G = (image_matrix[x][y][1] - (image_matrix[x][y][1] * X
            )) + (const - (const * X))
        B = (image_matrix[x][y][2] - (image_matrix[x][y][2] * X
            )) + (const - (const * X))

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y][0] = math.ceil(R)
        result_matrix[x][y][1] = math.ceil(G)
        result_matrix[x][y][2] = math.ceil(B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([R, G, B]):
            f_min = min([R, G, B])
        if f_max < max([R, G, B]):
            f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
            f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
            f_min) / (f_max - f_min))

```

## 4.2 Sumowanie dwóch obrazów

Algebraiczne sumowanie obrazów  $f$  i  $f'$  jest określone jedynie dla obrazów o tych samych wymiarach  $M \times N$  i strukturze ich macierzy. Dodawanie obrazów jest użyteczne w uśrednianiu obrazów, wykonywanym w celu zredukowania na nich szumu. Algorytm sumowania obrazu z obrazem polega na dodaniu do wartości piksla z pierwszego obrazu, wartości odpowiadającego piksla z drugiego obrazu. Po operacji sumowania następuje normalizacja obrazu.

1. Weź dwa identycznych rozmiarów obrazy  $P1$  i  $P2$
2. Dla wszystkich pikseli w tych obrazach wykonaj:
3. Policz sumy odpowiadających składowych barwy.
4. Wybierz największą sumę  $Q_{max} = \max(R_s, G_s, B_s)$  i policz równania:

$$Q_R[i, j] = (R_s * 255) / Q_{max},$$

$$Q_G[i, j] = (G_s * 255) / Q_{max},$$

$$Q_B[i, j] = (B_s * 255) / Q_{max}.$$

Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 4.3: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.4: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.2: Sumowanie obrazów barwowych

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum

```

```

R = int(image1_matrix[x][y][0]) + int(image2_matrix[x][y][0])
G = int(image1_matrix[x][y][1]) + int(image2_matrix[x][y][1])
B = int(image1_matrix[x][y][2]) + int(image2_matrix[x][y][2])

# Poszukiwanie maksimum
if Q_max < max([R, G, B]):
    Q_max = max([R, G, B])

# Sprawdzenie czy maximum przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255) # Obliczenie proporcji

# Obliczenie sum z uwzglednieniem zakresu
for y in range(height):
    for x in range(width):
        R = (image1_matrix[x][y][0] - (image1_matrix[x][y][0] * X)) + (image2_matrix[x][y][0] - (image2_matrix[x][y][0] * X))
        G = (image1_matrix[x][y][1] - (image1_matrix[x][y][1] * X)) + (image2_matrix[x][y][1] - (image2_matrix[x][y][1] * X))
        B = (image1_matrix[x][y][2] - (image1_matrix[x][y][2] * X)) + (image2_matrix[x][y][2] - (image2_matrix[x][y][2] * X))

        # Zaokroglenie do najblizszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(R)
        result_matrix[x][y][1] = math.ceil(G)
        result_matrix[x][y][2] = math.ceil(B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([R, G, B]):
            f_min = min([R, G, B])
        if f_max < max([R, G, B]):
            f_max = max([R, G, B])

# Normalizacja

```

```

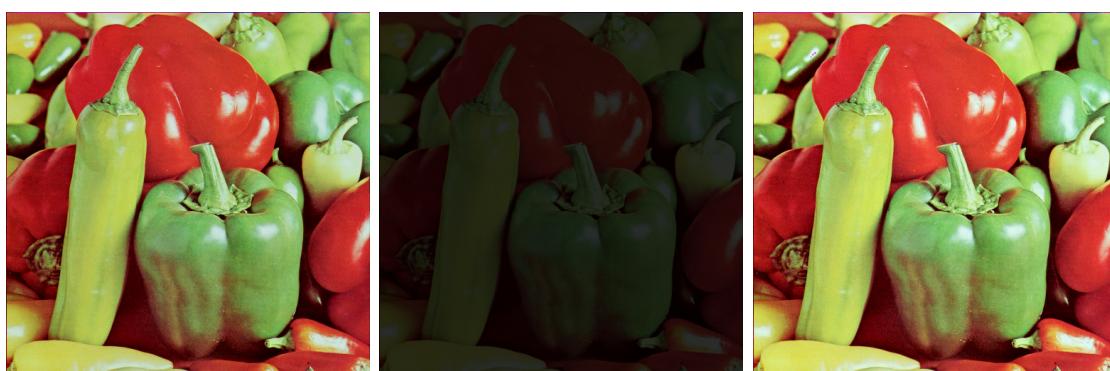
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

### 4.3 Mnożenie obrazu przez zadaną liczbę

Mnożenie obrazu  $f$  przez skalar wykonuje się mnożąc każdy element obrazu  $f_{i,j}$  (wartość funkcji obrazowej piksla) przez ten skalar. Barwa wynikowa jest zawsze barwą ciemniejszą.

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa barwy piksela  $P_1[i, j]$  ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości stalej.
3. W przeciwnym przypadku, jeżeli składowa barwy piksela  $P_1[i, j]$  ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 4.5: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 50, obraz po normalizacji



Rysunek 4.6: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 100, obraz po normalizacji

Listing 4.3: Mnożenie obrazu barwowego przez zadaną liczbę

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = int(image_matrix[x][y][0])
        G = int(image_matrix[x][y][1])
        B = int(image_matrix[x][y][2])

        if R == 255:
            R = const
        elif R == 0:
            R = 0
        else:
            R = (int(image_matrix[x][y][0]) * int(const)) / 255

        if G == 255:
            G = const
        elif G == 0:
            G = 0
        else:
            G = (int(image_matrix[x][y][1]) * int(const)) / 255

        if B == 255:
            B = const
        elif B == 0:
            B = 0
        else:
            B = (int(image_matrix[x][y][2]) * int(const)) / 255

        result_matrix[x][y] = [R, G, B]
    
```

```

G = 0
else:
    G = (int(image_matrix[x][y][1]) * int(const))/255

if B == 255:
    B = const
elif B == 0:
    B = 0
else:
    B = (int(image_matrix[x][y][2]) * int(const))/255

# Zaokroglenie do najblizszej wartosci calkowitej z
# gory
# i przypisanie wartosci
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

## 4.4 Mnożenie obrazu przez inny obraz

Mnożenie obrazu  $f$  przez inny obraz wykonuje się mnożąc każdy element obrazu  $P1_{i,j}$  (wartość funkcji obrazowej piksla) przez odpowiadającego piksla drugiego obrazu  $P2_{i,j}$

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa piksela  $P1[i, j]$  ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości składowej  $P2[i, j]$ .

3. W przeciwnym przypadku, jeśli składowa piksela  $P_1[i, j]$  ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 4.7: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstający w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.8: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.4: Mnożenie obrazu barwowego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        if R == 255:

```

```

R = image2_matrix[x][y][0]
elif R == 0:
    R = 0
else:
    R = (int(image1_matrix[x][y][0]) * int(
        image2_matrix[x][y][0]))/255

if G == 255:
    G = image2_matrix[x][y][1]
elif G == 0:
    G = 0
else:
    G = (int(image1_matrix[x][y][1]) * int(
        image2_matrix[x][y][1]))/255

if B == 255:
    B = image2_matrix[x][y][2]
elif B == 0:
    B = 0
else:
    B = (int(image1_matrix[x][y][2]) * int(
        image2_matrix[x][y][2]))/255

# Zaokrąglenie do najbliższej wartości całkowitej z
# gory
# i przypisanie wartości
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))

```

```

norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
    f_min) / (f_max - f_min))
norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
    f_min) / (f_max - f_min))

```

## 4.5 Mieszanie obrazów z określonym współczynnikiem

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami  $\alpha$  i  $(1-\alpha)$ , odpowiednio, wg wzoru:

$$f_m = f\alpha + f^I(1 - \alpha),$$

gdzie  $\alpha \in [0, 1]$ . Płynna zmiana parametru  $\alpha$  w przedziale  $[0, 1]$  powoduje efekt przechodzenia obrazu  $f^I$  w obraz  $f$ . W mieszaniu obrazów nie ma problemu normalizacji, a jedynie jednolitości struktur i typów obrazowych.

1. Weź dwa identycznych rozmiarów obrazy  $P_1$  i  $P_2$ .
2. Określ współczynnik mieszania  $\alpha$  wyrażony jako liczba rzeczywista z zakresu  $<0, 1>$ ; 0 reprezentuje pewną przezroczystość, 1 - nieprzezroczystości.
3. Dla wszystkich pikseli w obrazach wejściowych wykonuj  $Q(i, j) = \alpha * P_1(i, j) + (1 - \alpha) * P_2(i, j)$



Rysunek 4.9: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem  $\alpha=0.5$ , poniżej obraz wynikowy po normalizacji



Rysunek 4.10: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem  $\alpha=0.8$ , poniżej obraz wynikowy po normalizacji

Listing 4.5: Mieszanie obrazów barwowych z określonym współczynnikiem

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = float(image1_matrix[x][y][0]) * alfa + (1-alfa) *
            float(image2_matrix[x][y][0])
        G = float(image1_matrix[x][y][1]) * alfa + (1-alfa) *
            float(image2_matrix[x][y][1])
        B = float(image1_matrix[x][y][2]) * alfa + (1-alfa) *
            float(image2_matrix[x][y][2])

        result_matrix[x][y] = [R, G, B]
    
```

```

B = float(image1_matrix[x][y][2]) * alfa + (1-alfa) *
    float(image2_matrix[x][y][2])

# Zaokroglenie do najblizszej wartosci całkowitej z gory
# i przypisanie wartosci
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

## 4.6 Potęgowanie obrazu

Potęgowanie obrazu jest szczególnym przypadkiem operacji mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

$$f_m = 255 \left( \frac{f(x,y)}{f_{max}} \right)^\alpha, \alpha > 0$$



Rysunek 4.11: (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi  $\alpha=2$ , obraz po normalizacji



Rysunek 4.12: (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi  $\alpha=3$ , obraz po normalizacji

Listing 4.6: Potęgowanie obrazu barwowego

```

image1_matrix = self.im1
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):
        
```

```

R = int(image1_matrix[x][y][0])
G = int(image1_matrix[x][y][1])
B = int(image1_matrix[x][y][2])

if f_img_max < max([R, G, B]):
    f_img_max = max([R, G, B])

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        if R == 0:
            R = 0
        else:
            R = 255 * (math.pow(int(image1_matrix[x][y][0]) / f_img_max, alfa))

        if G == 0:
            G = 0
        else:
            G = 255 * (math.pow(int(image1_matrix[x][y][1]) / f_img_max, alfa))

        if B == 0:
            B = 0
        else:
            B = 255 * (math.pow(int(image1_matrix[x][y][2]) / f_img_max, alfa))

# Zaokroglenie do najblizszej wartosci calkowitej z gory
# i przypisanie wartosci
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):

```

```

f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

## 4.7 Dzielenie obrazu przez (zadaną) liczbę

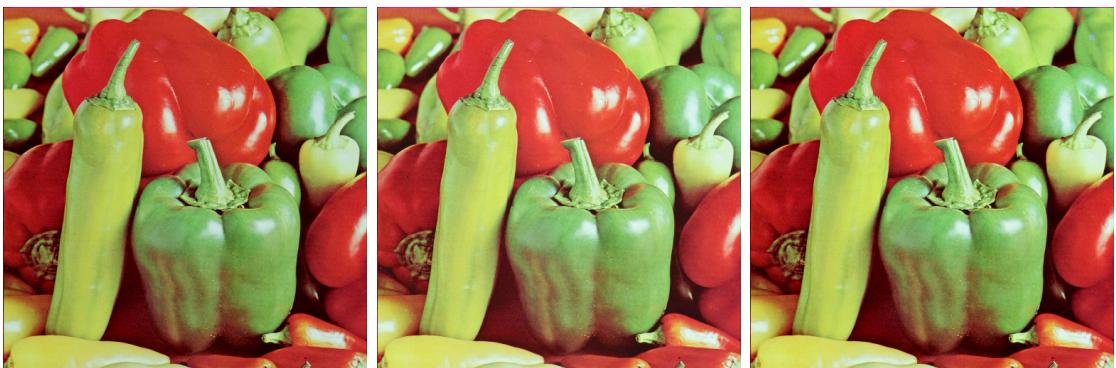
Dzielenie obrazów stosuje się w celu korekcji cieniowania między poziomami szarości.

1. Dla wszystkich pikseli w tych obrazach wykonaj:
  2. Policz sumy piksli ze stałą.
  3. Wybierz największą sumę  $Q_{max} = \max(R_s, G_s, B_s)$  i policz równania:  

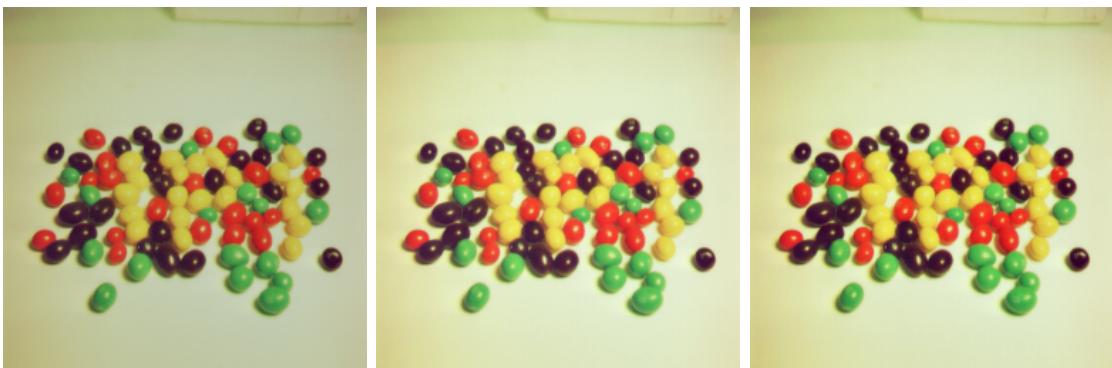
$$Q_R[i, j] = (R_s * 255)/Q_{max},$$
  

$$Q_G[i, j] = (G_s * 255)/Q_{max},$$
  

$$Q_B[i, j] = (B_s * 255)/Q_{max}.$$
- Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 4.13: (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji



Rysunek 4.14: (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji

Listing 4.7: Dzielenie obrazu barwowego przez (zadaną) liczbę

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R_S, G_S, B_S]):
            Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum

```

```

R_S = int(image_matrix[x][y][0]) + int(const)
G_S = int(image_matrix[x][y][1]) + int(const)
B_S = int(image_matrix[x][y][2]) + int(const)

Q_R = (R_S * 255)/Q_max
Q_G = (G_S * 255)/Q_max
Q_B = (B_S * 255)/Q_max

# Zaokroglenie do najblizszej wartosci calkowitej z
# gory
# i przypisanie wartosci
result_matrix[x][y][0] = math.ceil(Q_R)
result_matrix[x][y][1] = math.ceil(Q_G)
result_matrix[x][y][2] = math.ceil(Q_B)

# Poszukiwanie minimum i maksimum
if f_min > min([Q_R, Q_G, Q_B]):
    f_min = min([Q_R, Q_G, Q_B])
if f_max < max([Q_R, Q_G, Q_B]):
    f_max = max([Q_R, Q_G, Q_B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

## 4.8 Dzielenie obrazu przez inny obraz

1. Dla wszystkich pikseli w tych obrazach wykonaj:
2. Weź dwa identycznych rozmiarów obrazy  $P1$  i  $P2$
3. Policz sumy piksli ze stałą.
4. Wybierz największą sumę  $Q_{max} = \max(R_s, G_s, B_s)$  i policz równania:  

$$Q_R[i, j] = (R_s * 255)/Q_{max}$$
  

$$Q_G[i, j] = (G_s * 255)/Q_{max}$$

$$Q_B[i, j] = (B_S * 255) / Q_{max}$$

Wynik zaokrągluj do najbliższej górnej liczby całkowitej.



Rysunek 4.15: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.16: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.8: Dzielenie obrazu barwowego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image1_matrix[x][y][0]) + int(image2_matrix[x][y][0])

```

```

G_S = int(image1_matrix[x][y][1]) + int(image2_matrix[x]
    ][y][1])
B_S = int(image1_matrix[x][y][2]) + int(image2_matrix[x]
    ][y][2])

# Poszukiwanie maksimum
if Q_max < max([R_S, G_S, B_S]):
    Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image1_matrix[x][y][0]) + int(image2_matrix[x]
            ][y][0])
        G_S = int(image1_matrix[x][y][1]) + int(image2_matrix[x]
            ][y][1])
        B_S = int(image1_matrix[x][y][2]) + int(image2_matrix[x]
            ][y][2])

        Q_R = (R_S * 255)/Q_max
        Q_G = (G_S * 255)/Q_max
        Q_B = (B_S * 255)/Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(Q_R)
        result_matrix[x][y][1] = math.ceil(Q_G)
        result_matrix[x][y][2] = math.ceil(Q_B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([Q_R, Q_G, Q_B]):
            f_min = min([Q_R, Q_G, Q_B])
        if f_max < max([Q_R, Q_G, Q_B]):
            f_max = max([Q_R, Q_G, Q_B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))

```

```

norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
    f_min) / (f_max - f_min))
norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
    f_min) / (f_max - f_min))

```

## 4.9 Pierwiastkowanie obrazu

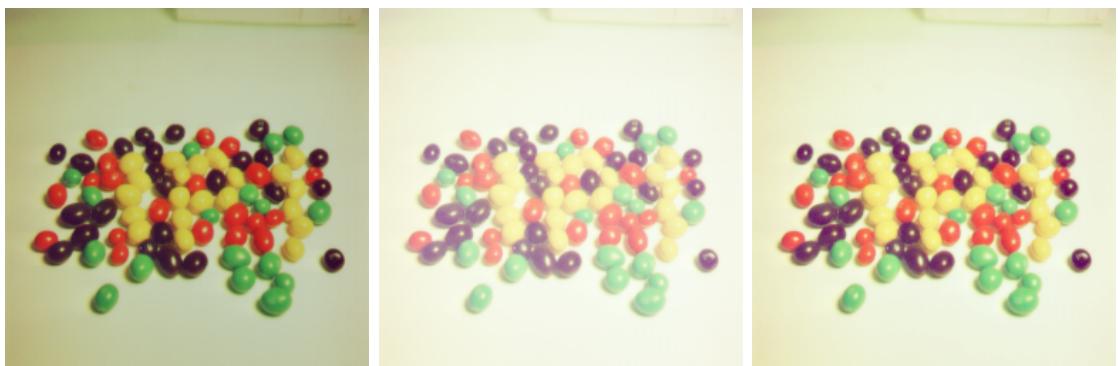
Pierwiastkowanie obrazu jest szczególnym przypadkiem operacji potęgowania obrazów, gdzie wykładnikiem jest ułamek. Aby uniknąć wykroczenia poza zakres, skorzystano ze

znormalizowanego wzoru [2]:

$$f_m = 255 \left( \frac{f(x,y)}{f_{max}} \right)^\alpha, \alpha > 0$$



Rysunek 4.17: (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ( $\alpha=1/2$ ), obraz po normalizacji



Rysunek 4.18: (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ( $\alpha=1/3$ ), obraz po normalizacji

Listing 4.9: Pierwiastkowanie obrazu barwowego

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R_S, G_S, B_S]):
            Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        Q_R = (R_S * 255)/Q_max
        Q_G = (G_S * 255)/Q_max
        Q_B = (B_S * 255)/Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(Q_R)
        result_matrix[x][y][1] = math.ceil(Q_G)
        result_matrix[x][y][2] = math.ceil(Q_B)

```

```

# Poszukiwanie minimum i maksimum
if f_min > min([Q_R, Q_G, Q_B]):
    f_min = min([Q_R, Q_G, Q_B])
if f_max < max([Q_R, Q_G, Q_B]):
    f_max = max([Q_R, Q_G, Q_B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
                                         f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
                                         f_min) / (f_max - f_min))

```

## 4.10 Logarytmowanie obrazu

Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Wykorzystano wzór z normalizacją /citeWykRat:

$$f_m = 255(\frac{1}{\log(1 + f(x, y))} \log(1 + f_{max}))$$

Przesunięcie funkcji obrazowej  $f$  do góry o 1 przed jej logarytmowaniem wynika z nieokreśloności logarytmu w zerze. Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu.



Rysunek 4.19: (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji



Rysunek 4.20: (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji

Listing 4.10: Logarytmowanie obrazu barwowego

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

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

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        # Poszukiwanie maksimum
        if f_img_max < max([R, G, B]):
            f_img_max = max([R, G, B])

for y in range(height):
    for x in range(width):

```

```

R = int(image1_matrix[x][y][0])
G = int(image1_matrix[x][y][1])
B = int(image1_matrix[x][y][2])

if R == 0:
    R = 0
else:
    R = math.log(1 + int(image1_matrix[x][y][0])) /
        math.log(1 + int(f_img_max)) * 255

if G == 0:
    G = 0
else:
    G = math.log(1 + int(image1_matrix[x][y][1])) /
        math.log(1 + int(f_img_max)) * 255

if B == 0:
    B = 0
else:
    B = math.log(1 + int(image1_matrix[x][y][2])) /
        math.log(1 + int(f_img_max)) * 255

# Zaokroglenie do najblizszej wartosci calkowitej z
# gory
# i przypisanie wartosci
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))

```

```
norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -  
    f_min) / (f_max - f_min))  
norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -  
    f_min) / (f_max - f_min))
```

## Rozdział 5

# Operacje geometryczne na obrazie

Transformacje geometryczne są szczególnie wykorzystywane w przypadku dopasowywania obrazu do układu współrzędnych oraz w przypadku eliminowania zniekształceń geometrycznych obrazu. W przedstawionych operacjach obrazy umieszczone są w pierwszej ćwiartce układu współrzędnych.

## 5.1 Przemieszczenie obrazu o zadany wektor

Przesuwanie obrazu polega na zmianie współrzędnych każdego piksela obrazu o określoną wartość zgodnie z zależnościami:

$$\begin{aligned}x^I &= x_o + \Delta x \\y^I &= y_o + \Delta y\end{aligned}$$

gdzie  $(x_o, y_o)$  - współrzędnych początkowe piksla;  $(\Delta x, \Delta y)$  - wartości przesunięcia;  $(x^I, y^I)$  - współrzędne piksla po przesunięciu;



Rysunek 5.1: (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [40, 70]



Rysunek 5.2: (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [200, 100]

Listing 5.1: Przemieszczenie obrazu o zadany wektor

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

delta_y = 0 - delta_y # Poruszamy sie w pierwszej cwartce
                      ukladu wspolrzednych

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

for y in range(height):
    for x in range(width):
        if 0 < y+delta_y < height and 0 < x+delta_x < width:
            result_matrix[y+delta_y][x+delta_x] = image_matrix[
                y][x]

```

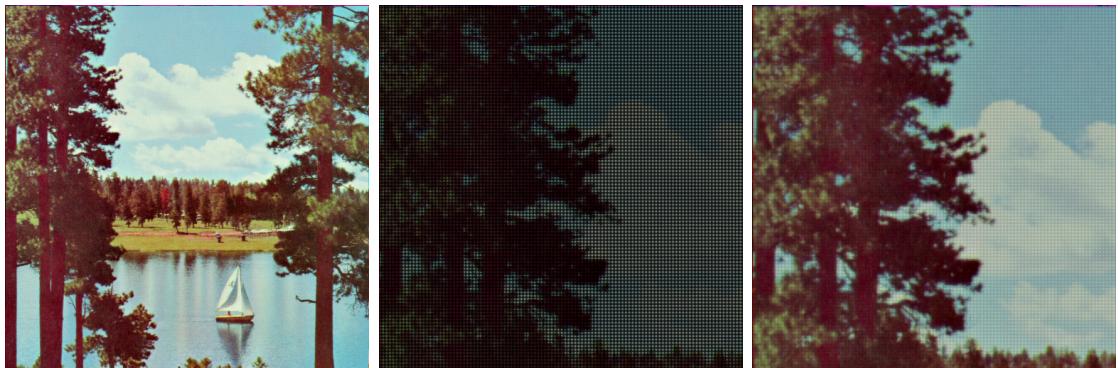
## 5.2 Jednorodne skalowanie obrazu

Skalowanie jednorodne obrazu polega na pomnożeniu współrzędnych każdego piksła obrazu przez współczynnik skalowania  $S$  wg. wzoru:

$$\begin{aligned} x^I &= x_o * S \\ y^I &= y_o * S \end{aligned}$$



Rysunek 5.3: (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem  $S=1.5$  , obraz po interpolacji



Rysunek 5.4: (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem  $S=2$ , obraz po interpolacji

Listing 5.2: Jednorodne skalowanie obrazu

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

# result_matrix = np.empty((math.ceil(height/scale_y), math.
#                           ceil(width/scale_x), 3), dtype=np.uint8)
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        if scale*y < height and scale*x < width:
            # result_matrix[y][x] = image_matrix[scale*y][scale
            *x]

```

```

result_matrix[int(scale*y)][int(scale*x)] =
    image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype=np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) | ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (resultImage2[iSafe, jSafe][1] > 0) | (resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        g += resultImage2[iSafe, jSafe][1]
                        b += resultImage2[iSafe, jSafe][2]
                        n += 1
        tmp[i, j] = (r/n, g/n, b/n)
        resultImage2[i, j] = tmp[i, j]

```

### 5.3 Niejednorodne skalowanie obrazu

Skalowanie niejednorodne obrazu polega na pomnożeniu współrzędnych każdego piksła obrazu przez współczynniki skalowania  $S_x, S_y$  wg. wzoru:

$$\begin{aligned} x^I &= x_o * S_x \\ y^I &= y_o * S_y \end{aligned}$$

gdzie  $(x_o, y_o)$  - współrzędne początkowe piksela;  $(S_x, S_y)$  - wartości współczynników skalowania;  $(x^I, y^I)$  - współrzędne piksła po skalowaniu;



Rysunek 5.5: (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem  $S_x=2$  oraz współczynnikami  $S_y=1$ , obraz po interpolacji



Rysunek 5.6: (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem  $S_x=1$  oraz współczynnikami  $S_y=2$ , obraz po interpolacji

Listing 5.3: Niejednorodne skalowanie obrazu

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

# result_matrix = np.empty((math.ceil(height/scale_y), math.
#                           ceil(width/scale_x), 3), dtype=np.uint8)
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        if scale_y*y < height and scale_x*x < width:
            # result_matrix[y][x] = image_matrix[scale_y*y][
            #                           scale_x*x]

```

```

result_matrix[int(scale_y*y)][int(scale_x*x)] =
    image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype=np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) | ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (resultImage2[iSafe, jSafe][1] > 0) | (resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        g += resultImage2[iSafe, jSafe][1]
                        b += resultImage2[iSafe, jSafe][2]
                        n += 1
        tmp[i, j] = (r/n, g/n, b/n)
        resultImage2[i, j] = tmp[i, j]

```

## 5.4 Obracanie obrazu o dowolny kąt

Operację obrotu dookoła początku układu współrzędnych wykonuje się zgodnie ze wzorem:

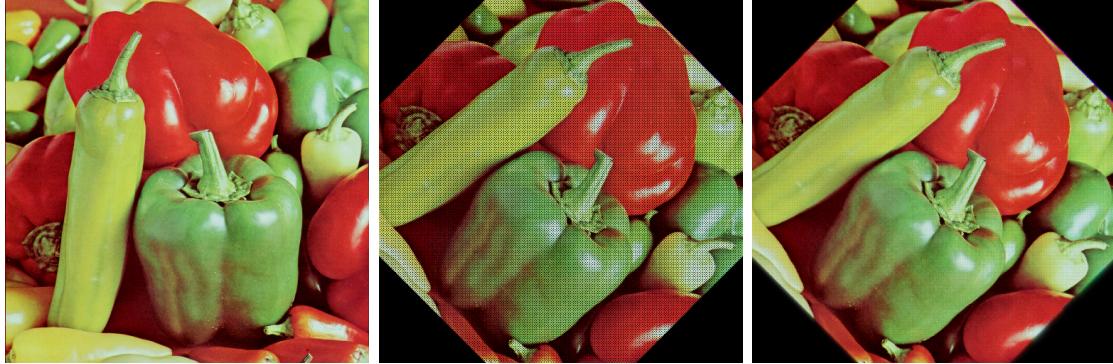
$$\begin{aligned} x^I &= x_o * \cos\alpha - y_o * \sin\alpha \\ y^I &= x_o * \sin\alpha + y_o * \cos\alpha \end{aligned}$$

gdzie  $(x_o, y_o)$  - współrzędne początkowe piksla;  $\alpha$  - kąt obrotu;  $(x^I, y^I)$  - współrzędne piksela po obrocie;

W przedstawionych przykładach punkt obrotu został przesunięty na środek obrazu według wzoru:

$$new_x = (x - width/2) * \mathit{math.cos}(alfa_r) - (y - height/2) * \mathit{math.sin}(alfa_r) + (width/2)$$

$$\text{new}_y = (x - \text{width}/2) * \text{math.sin}(\text{alfa}_r) + (y - \text{height}/2) * \text{math.cos}(\text{alfa}_r) + (\text{height}/2)$$



Rysunek 5.7: (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 40 stopni, obraz po interpolacji



Rysunek 5.8: (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 110 stopni, obraz po interpolacji

Listing 5.4: Obracanie obrazu o dowolny kąt

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

# Przekształcenie na radiany
alfa_r = math.radians(alfa)

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

for y in range(height):

```

```

for x in range(width):
    new_x = (x - width/2) * math.cos(alfa_r) - (y - height
        /2) * math.sin(alfa_r) + (width/2)
    new_y = (x - width/2) * math.sin(alfa_r) + (y - height
        /2) * math.cos(alfa_r) + (height/2)
    if new_y < height and new_y >= 0 and new_x >= 0 and
        new_x < width:
        result_matrix[int(new_y)][int(new_x)] =
            image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype = np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1]
            < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) |
                        ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) |
                        ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (
                        resultImage2[iSafe, jSafe][1] > 0) | (
                        resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        g += resultImage2[iSafe, jSafe][1]
                        b += resultImage2[iSafe, jSafe][2]
                    n += 1
        tmp[i, j] = (r/n, g/n, b/n)
        resultImage2[i, j] = tmp[i, j]

```

## 5.5 Symetrie względem osi układu

### Symetria względem osi X

Względem osi OX piksem symetrycznym do piksla  $P_1(x, y)$  jest piksel  $P_2(x, -y)$



Rysunek 5.9: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X



Rysunek 5.10: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X

Listing 5.5: Symetrie względem osi X

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_height = height-1 #array height last index

for y in range(height):
    for x in range(width):
        result_matrix[y][x] = image_matrix[_height-y][x]

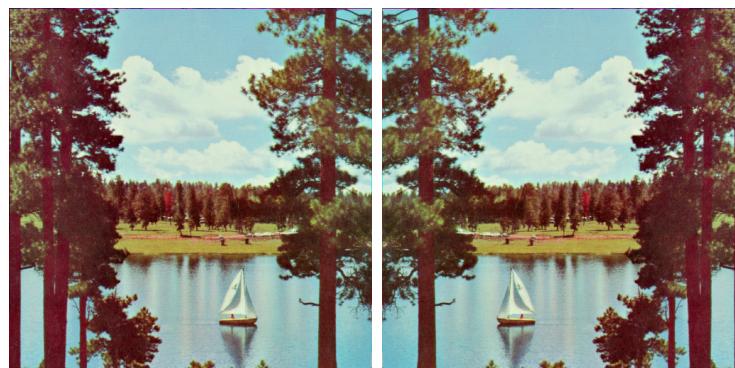
```

## Symetria względem osi Y

Względem osi OY piksem symetrycznym do piksla  $P_1(x, y)$  jest piksel  $P_2(-x, y)$



Rysunek 5.11: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y



Rysunek 5.12: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y

Listing 5.6: Symetrie względem osi Y

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_width = width - 1 #array width last index

for y in range(height):
    for x in range(width):
        result_matrix[y][x] = image_matrix[y][_width - x]
```

## 5.6 Symetrie względem zadanej prostej



Rysunek 5.13: (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu



Rysunek 5.14: (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu



Rysunek 5.15: (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu



Rysunek 5.16: (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu

Listing 5.7: Symetrie względem zadanej prostej

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

#Symetria wzgledem poziomej prostej poprowadzonej przez srodek
#obrazu
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_height = height - 1 #array height last index

param_y = height/2

for y in range(height):
    for x in range(width):
        if y < param_y:
            result_matrix[y][x] = image_matrix[y][x]
        else:
            result_matrix[y][x] = image_matrix[_height - y][x]

#Symetria wzgledem pionowej prostej poprowadzonej przez srodek
#obrazu
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_width = width - 1 #array width last index

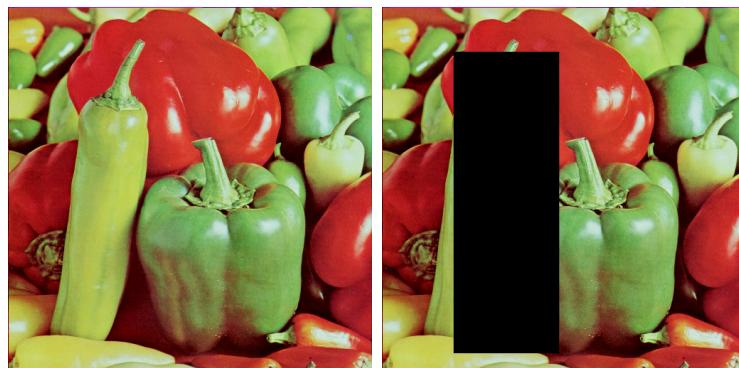
param_x = width/2

for y in range(height):

```

```
for x in range(width):
    if x < param_x:
        result_matrix[y][x] = image_matrix[y][x]
    else:
        result_matrix[y][x] = image_matrix[y][-width - x]
```

## 5.7 Wycinanie fragmentów obrazu



Rysunek 5.17: (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych  $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$



Rysunek 5.18: (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych  $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$

Listing 5.8: Wycinanie fragmentów obrazu

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc
```

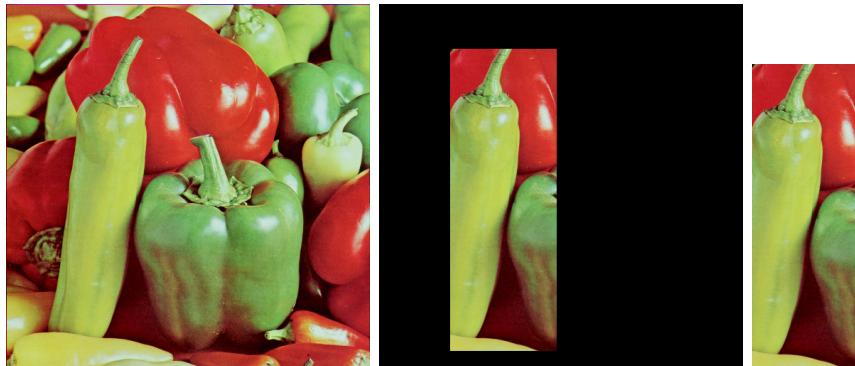
```

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

for y in range(height):
    for x in range(width):
        # Poruszamy sie w pierwszej cwiartce osi układu
        # współrzędnych
        if x > x_min and x < x_max and y < height-y_min and y >
            height-y_max:
            result_matrix [y] [x] = 0
        else:
            result_matrix [y] [x] = image_matrix [y] [x]

```

## 5.8 Kopiowanie fragmentów obrazów



Rysunek 5.19: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych  $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$ , Skopiowany fragment (151x426)



Rysunek 5.20: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych  $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$ , Skopiowany fragment (201x201)

Listing 5.9: Kopiowanie fragmentów obrazów

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

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

# Macierz o wymiarach wycinanego fragmentu
cut_matrix = np.zeros((y_max-y_min + 1, x_max-x_min + 1, 3),
                      dtype=np.uint8)

cut_y = 0
for y in range(height):
    cut_x = 0
    for x in range(width):
        # Poruszamy sie w pierwszej cwiartce osi ukladu
        # wspolrzednych
        if x >= x_min and x <= x_max and y <= height-y_min and
           y >= height-y_max:
            result_matrix[y][x] = image_matrix[y][x]
            cut_matrix[cut_y][cut_x] = image_matrix[y][x]
            cut_x+=1
    if cut_x > 0:
        cut_y+=1
```

## Rozdział 6

# Operacje na histogramie obrazu szarego

1. obliczanie histogramu
2. przemieszczanie histogramu
3. rozciąganie histogramu
4. progowanie lokalne
5. progowanie globalne

## **Rozdział 7**

# **Operacje na histogramie obrazu barwowego**

1. obliczanie histogramu
2. przemieszczanie histogramu
3. rozciąganie histogramu
4. progowanie 1-progowe
5. progowanie wieloprogowe
6. progowanie lokalne
7. progowanie globalne

## Rozdział 8

# Operacje morfologiczne na obrazach binarnych

W obrazie binarnym piksele mogą przybierać tylko dwie wartości. Zazwyczaj kodowane są za pomocą pojedynczego bitu i przyjmują wartość 0 lub 1. Spotyka się także reprezentacje wykorzystujące inne pary wartości: (0, 255), (-1, 1), (True, False).

W przedstawionych przykładach została użyta reprezentacja (0,255), gdzie 0 oznacza czerń a 255 biały.

W morfologicznym przetwarzaniu obrazów ważne jest określenie, kiedy dwa piksele sąsiadują ze sobą. W tym celu definiuje się dla każdego piksla jego sąsiedztwo. Przy implementacji wykorzystano sąsiedztwo czterospójne:

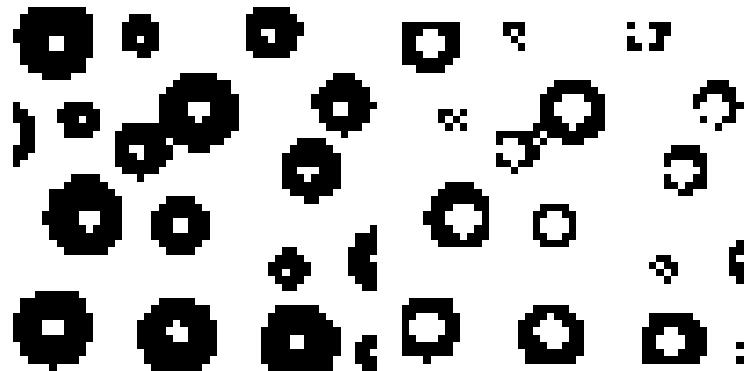
Sąsiedztwo czterospójne (von Neumanna) - obejmuje cztery piksele przyległe do danego z góry, dołu i po bokach

$$N_4(p) = ((x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y))$$

## 8.1 Okrawanie (erozja)

Przyjęto, że wartości wykraczające poza granice (wysokość/szerokość) obrazu są białe (maja wartość 255)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli  $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$  od piksla  $(x, y)$ .
3. Jeżeli którykolwiek z sąsiadów ma wartość równą 255 to środkowy piksel  $(x, y)$  ma przyjąć wartość 255.
4. Jeżeli wszyscy sąsiedzi mają wartość równą 0 to środkowy piksel  $(x, y)$  ma przyjąć wartość 0.



Rysunek 8.1: (Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)



Rysunek 8.2: (Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)

Listing 8.1: Operacja okrawania (erozji) na obrazie binarnym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        # Przyjeto, ze wartosci wykraczajace poza granice
        # obrazu sa biale (maja wartosc 255)
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x][0])
    
```

```

if x + 1 < width:
    neighbour_pix[2]=(image_matrix[y][x+1][0])
if y + 1 < height:
    neighbour_pix[3]=(image_matrix[y+1][x][0])

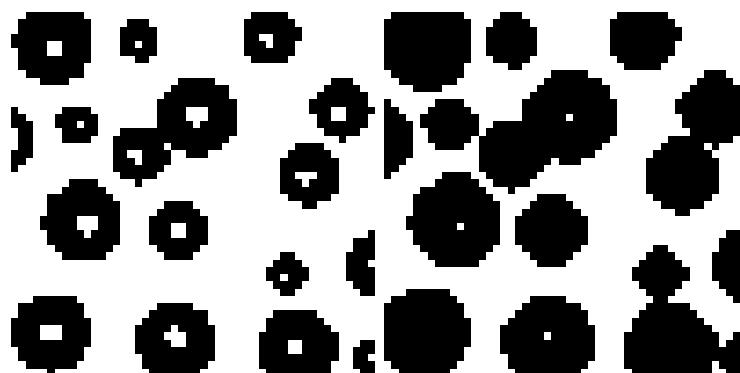
if 255 in neighbour_pix:
    result_matrix[y][x] = 255 #biały
else:
    result_matrix[y][x] = 0 #czarny

```

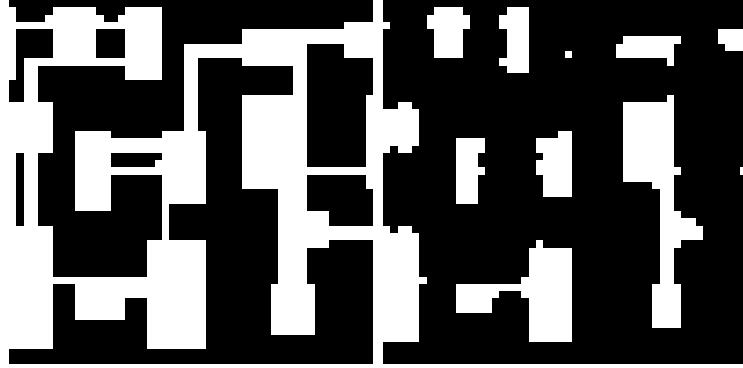
## 8.2 Nakładanie (dylatacja)

Przyjeto, że wartości wykraczające poza granice (wysokość/szerokość) obrazu są białe (maja wartość 255)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli  $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$  od piksla  $(x, y)$ .
3. Jeżeli którykolwiek z sąsiadów ma wartość równą 0 to środkowy piksel  $(x, y)$  ma przyjmować wartość 0.
4. Jeżeli wszyscy sąsiedzi mają wartość równą 255 to środkowy piksel  $(x, y)$  ma przyjmować wartość 255.



Rysunek 8.3: (Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)



Rysunek 8.4: (Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)

Listing 8.2: Operacja nakładania (dylatacji) na obrazie binarnym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

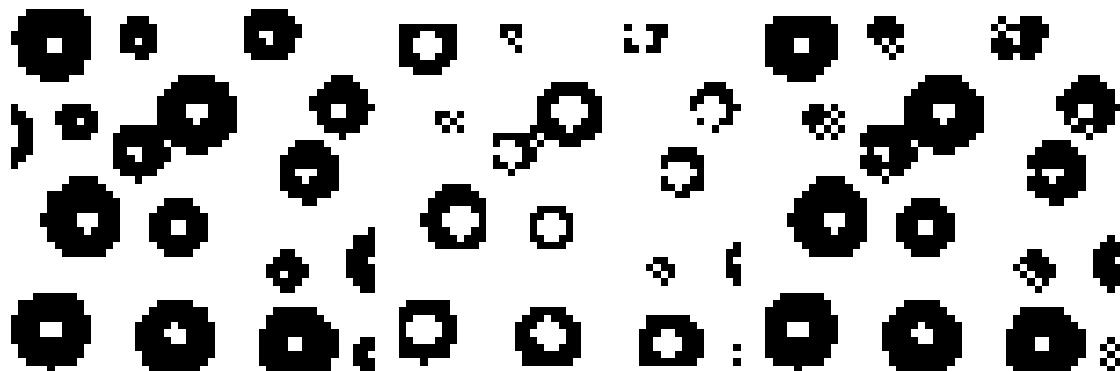
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x][0])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1][0])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x][0])

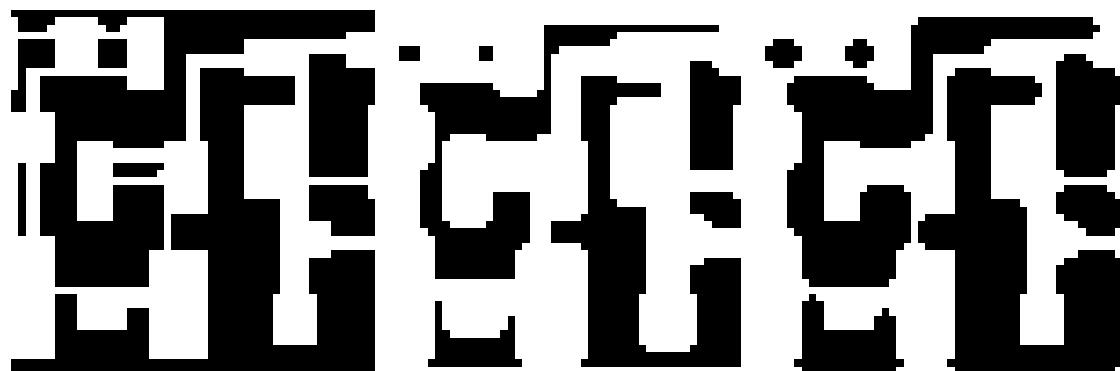
        if 0 in neighbour_pix:
            result_matrix[y][x] = 0
        else:
            result_matrix[y][x] = 255
    
```

### 8.3 Otwarcie .

Otwarcie morfologiczne jest równoważne nałożeniu operacji dylatacji na wynik erozji obrazu pierwotnego.



Rysunek 8.5: (Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)



Rysunek 8.6: (Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)

Listing 8.3: Operacja otwarcia na obrazie binarnym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#erozja
```

```

for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x][0])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1][0])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x][0])

        if 255 in neighbour_pix:
            e_result_matrix[y][x] = 255
        else:
            e_result_matrix[y][x] = 0

Image.fromarray(e_result_matrix).show()
#dylatacja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(e_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(e_result_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(e_result_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(e_result_matrix[y+1][x])

        if 0 in neighbour_pix:
            d_result_matrix[y][x] = 0

```

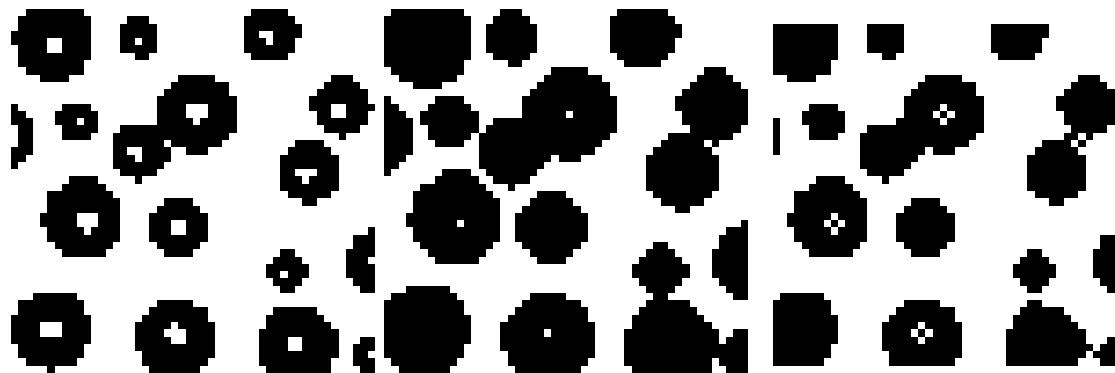
```

else :
    d_result_matrix [y] [x] = 255

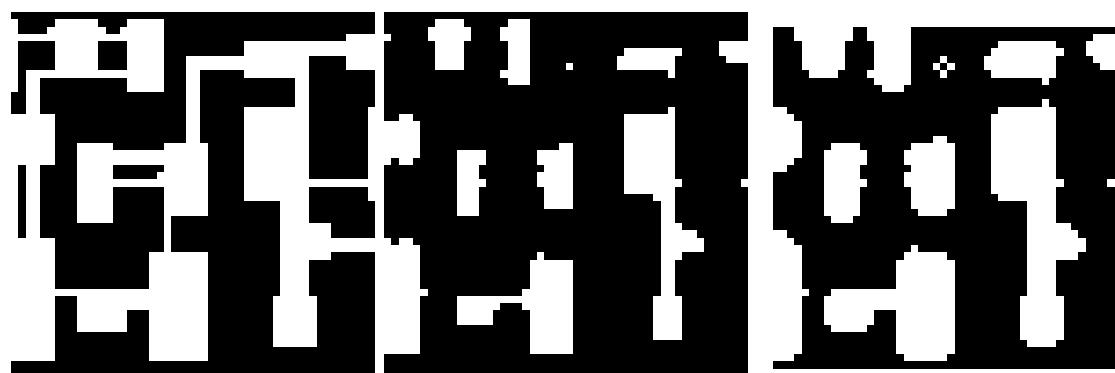
```

## 8.4 Zamknięcie ..

Zamknięcie morfologiczne jest równoważne nałożeniu operacji erozji na wynik dylatacji obrazu pierwotnego.



Rysunek 8.7: (Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)



Rysunek 8.8: (Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)

## Kod źródłowy ..

Listing 8.4: Operacja zamknięcia na obrazie binarnym

```

image_matrix = self .im1
width = image_matrix .shape [1]      # szereoksc
height = image_matrix .shape [0]     # wysokosc

```

```

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#dylatacja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x-1][0])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1][0])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x][0])

        if 0 in neighbour_pix:
            d_result_matrix[y][x] = 0
        else:
            d_result_matrix[y][x] = 255

```

```
Image.fromarray(e_result_matrix).show()
```

```

#erozja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(d_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(d_result_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(d_result_matrix[y][x+1])
        if y + 1 < height:

```

```
neighbour_pix[3]=( d_result_matrix[y+1][  
x] )  
  
if 255 in neighbour_pix:  
    e_result_matrix[y][x] = 255  
else:  
    e_result_matrix[y][x] = 0
```

## Rozdział 9

# Operacje morfologiczne na obrazach szarych

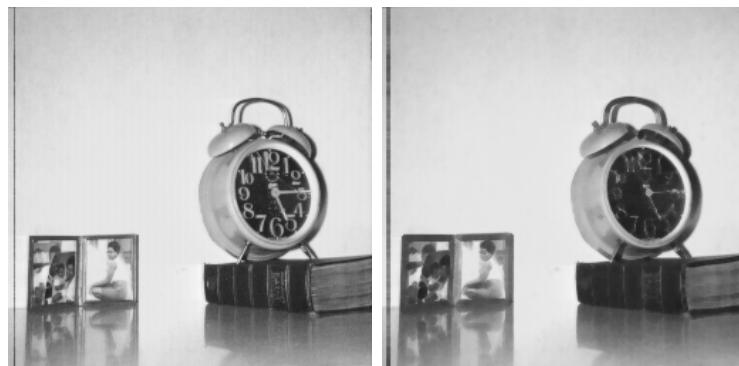
W morfologicznym przetwarzaniu obrazów ważne jest określenie, kiedy dwa piksele sąsiadują ze sobą. W tym celu definiuje się dla każdego piksla jego sąsiedztwo. Przy implementacji wykorzystano sąsiedztwo czterospójne:

Sąsiedztwo czterospójne (von Neumanna) - obejmuje cztery piksele przyległe do danego z góry, dołu i po bokach

$$N_4(p) = ((x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y))$$

## 9.1 Okrawanie (erozja)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli do wektora  $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$  od piksla  $(x, y)$ .
3. Posortuj wektor.
4. Przypisz wartość piksla o najniższej jasności do piksla  $(x, y)$ .



Rysunek 9.1: (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji okrawania (erozji)



Rysunek 9.2: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji okrawania (erozji)

Listing 9.1: Operacja okrawania (erozji) na obrazie szarym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

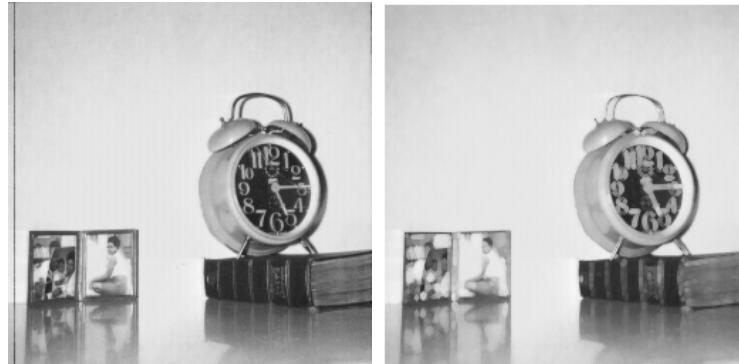
        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        min_pix = min(neighbour_pix)
        result_matrix[y][x] = min_pix
    
```

## 9.2 Nakładanie (dylatacja) .

1. Dla wszystkich pikseli wykonaj:

2. Wczytaj wartości sąsiadujących pikseli do wektora  $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$  od piksela  $(x, y)$ .
3. Posortuj wektor.
4. Przypisz wartość piksela o najwyższej jasności do piksela  $(x, y)$ .



Rysunek 9.3: (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji nakładania (dylatacji)



Rysunek 9.4: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji nakładania (dylatacji)

Listing 9.2: Operacja nakładania (dylatacji) na obrazie szarym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szerokość
height = image_matrix.shape[0]     # wysokość

result_matrix = np.zeros((height, width), dtype=np.uint8)
print(image_matrix)
print(result_matrix)
```

```

for y in range( height ):
    for x in range( width ):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        max_pix = max(neighbour_pix)
        result_matrix[y][x] = max_pix
    
```

### 9.3 Otwarcie .

Otwarcie morfologiczne jest równoważne nałożeniu operacji dylatacji na wynik erozji obrazu pierwotnego.



Rysunek 9.5: (Od lewej) Obraz wejściowy (256x256), obraz po dylacjii, obraz po operacji otwarcia (erozja → dylatacja)



Rysunek 9.6: (Od lewej) Obraz wejściowy (512x512), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)

Listing 9.3: Operacja otwarcia na obrazie szarym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#erozja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        min_pix = min(neighbour_pix)
        e_result_matrix[y][x] = min_pix

Image.fromarray(e_result_matrix).show()
#dylatacja
for y in range(height):

```

```

for x in range(width):
    neighbour_pix = [255, 255, 255, 255]

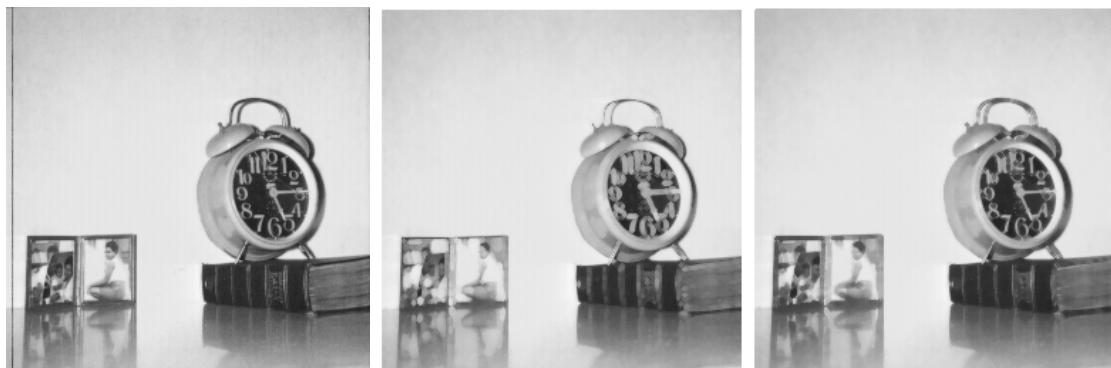
    if x - 1 > 0:
        neighbour_pix[0]=( e_result_matrix[y][x-1])
    if y - 1 > 0:
        neighbour_pix[1]=( e_result_matrix[y-1][x])
    if x + 1 < width:
        neighbour_pix[2]=( e_result_matrix[y][x+1])
    if y + 1 < height:
        neighbour_pix[3]=( e_result_matrix[y+1][x])

    max_pix = max(neighbour_pix)
    d_result_matrix[y][x] = max_pix

```

## 9.4 Zamknięcie .

Zamknięcie morfologiczne jest równoważne nałożeniu operacji erozji na wynik dylatacji obrazu pierwotnego.



Rysunek 9.7: (Od lewej) Obraz wejściowy (256x256), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)



Rysunek 9.8: (Od lewej) Obraz wejściowy (512x512), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)

## Kod źródłowy .

Listing 9.4: Operacja zamknięcia na obrazie szarym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#dylatacja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        max_pix = max(neighbour_pix)
        d_result_matrix[y][x] = max_pix

#erozja

```

```
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=( d_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=( d_result_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=( d_result_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=( d_result_matrix[y+1][x])

        min_pix = min(neighbour_pix)
        e_result_matrix[y][x] = min_pix
```

1. okrawanie(erózja)
2. nakładanie (dylatacja)
3. otwarcie
4. zamknięcie

## **Rozdział 10**

# **Filtrowanie liniowe i nieliniowe**

1. dolnoprzepustowe (dwa do wyboru)
2. górnoprzepustowe (Roberts, Prewitta, Sobella, ....)
3. gradientowe (kompasowe, płaskorzeźbowe kierunkowe, gradientu wektorowego VGO, gradientu wektora kierunkowego VDG).
4. medianowe
5. ekstremalne

## Rozdział 11

# Podsumowanie

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

# Bibliografia

- [1] Wojciech S. Mokrzycki. *Wprowadzenie do przetwarzania informacji wizualnej Tom II*. Akademicka Oficyna Wydawnicza EXIT, 2012.
- [2] J. Ratajczaki. *Cyfrowe przetwarzanie obrazów i sygnałów - wykład 3*. Politechnika Wrocławskiego, 2015.