

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>	7
1.1 Specyfikacja wykorzystanego formatu obrazu	7
1.2 Instrukcja obsługi programu	8
<b>Rozdział 2. Operacje ujednoliciania obrazów</b>	9
2.1 Ujednolicenie obrazów szarych geometryczne	10
2.2 Ujednolicenie obrazów szarych rozdzielczościowe	13
2.3 Ujednolicenie obrazów RGB geometryczne	17
2.4 Ujednolicenie obrazów RGB rozdzielczościowe	20
<b>Rozdział 3. Operacje sumowania arytmetycznego obrazów szarych</b>	24
<b>Rozdział 4. Operacje sumowania arytmetycznego obrazów barwowych</b>	25
<b>Rozdział 5. Operacje geometryczne na obrazie</b>	26
<b>Rozdział 6. Operacje na histogramie obrazu szarego</b>	27
6.1 Obliczanie histogramu	28
6.2 Przeszczepianie histogramu	30
6.3 Rozciąganie histogramu	33
6.4 Progowanie lokalne	36
6.5 Progowanie globalne	39
<b>Rozdział 7. Operacje na histogramie obrazu barwowego</b>	42
7.1 Obliczanie histogramu	43
7.2 Przeszczepianie histogramu	45
7.3 Rozciąganie histogramu	48
7.4 Progowanie 1-progowe lokalne	51
7.5 Progowanie wielo-progowe lokalne	54
7.6 Progowanie 1-progowe globalne	58
7.7 Progowanie wielo-progowe globalne	61
<b>Rozdział 8. Operacje morfologiczne na obrazach binarnych</b>	64
<b>Rozdział 9. Operacje morfologiczne na obrazach szarych</b>	65
<b>Rozdział 10. Filtrowanie liniowe i nielinowe</b>	66
10.1 Filtr dolnoprzepustowy uśredniający	67
10.2 Filtr dolnoprzepustowy Gaussowski	71
10.3 Operator Roberts'a	75

10.4 Operator Prewitt'a . . . . .	79
10.5 Operator Sobel'a . . . . .	83
10.6 Filtr kompasowy . . . . .	88
10.7 Gradient wektora kierunkowego . . . . .	96
10.8 Filtr medianowy . . . . .	102
10.9 Filtr maksymalny . . . . .	106
10.10 Filtr minimalny . . . . .	110
10.11 Filtr płaskorzeźbowy . . . . .	114
<b>Rozdział 11. Podsumowanie . . . . .</b>	<b>119</b>
<b>Bibliografia . . . . .</b>	<b>120</b>

# Spis rysunków

2.1	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512) . . . . .	10
2.2	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	10
2.3	Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512) . . . . .	11
2.4	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	11
2.5	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	13
2.6	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	13
2.7	Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	14
2.8	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	14
2.9	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512) . . . . .	17
2.10	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	17
2.11	Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512) . . . . .	18
2.12	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	18
2.13	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	20
2.14	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	20
2.15	Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	21
2.16	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512) . . . . .	21
6.1	Obraz szary, histogram szarości tego obrazu . . . . .	28
6.2	Obraz szary, histogram szarości tego obrazu . . . . .	28
6.3	Obraz szary wejściowy, histogram szarości tego obrazu . . . . .	30
6.5	Obraz szary wejściowy, histogram szarości tego obrazu . . . . .	31
6.4	Obraz szary przesunięty o 100, histogram szarości tego obrazu . . . . .	31
6.6	Obraz szary przesunięty o -100, histogram szarości tego obrazu . . . . .	32
6.7	Obraz wejściowy, histogram szarości tego obrazu . . . . .	33
6.8	Obraz po rozciagnięciu, histogram szarości tego obrazu . . . . .	33
6.9	Obraz wejściowy, histogram szarości tego obrazu . . . . .	34
6.10	Obraz po rozciagnięciu, histogram szarości tego obrazu . . . . .	34
6.11	Obraz wejściowy, histogram szarości tego obrazu . . . . .	36
6.12	Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu . . . . .	36
6.13	Obraz wejściowy, histogram szarości tego obrazu . . . . .	37
6.14	Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu . . . . .	37
6.15	Obraz wejściowy, histogram szarości tego obrazu . . . . .	39
6.16	Obraz po progowaniu globalnym, histogram szarości tego obrazu . . . . .	40
6.17	Obraz wejściowy, histogram szarości tego obrazu . . . . .	40
6.18	Obraz po progowaniu globalnym, histogram szarości tego obrazu . . . . .	41
7.1	Obraz barwny, histogram barw tego obrazu . . . . .	43

7.2	Obraz barwny, histogram barw tego obrazu . . . . .	43
7.3	Obraz wejściowy, histogram barw tego obrazu . . . . .	45
7.4	Obraz wyjściowy przesunięty o 50, histogram barw tego obrazu . . . . .	45
7.5	Obraz wejściowy, histogram barw tego obrazu . . . . .	46
7.6	Obraz wyjściowy przesunięty o -50, histogram barw tego obrazu . . . . .	46
7.7	Obraz wejściowy, histogram barw tego obrazu . . . . .	48
7.8	Obraz po rozciągnięciu, histogram barw tego obrazu . . . . .	49
7.9	Obraz wejściowy, histogram barw tego obrazu . . . . .	49
7.10	Obraz po rozciągnięciu, histogram barw tego obrazu . . . . .	49
7.11	Obraz wejściowy, histogram szarości tego obrazu . . . . .	51
7.12	Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu . .	52
7.13	Obraz wejściowy, histogram szarości tego obrazu . . . . .	52
7.14	Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu . .	52
7.15	Obraz wejściowy, histogram barw tego obrazu . . . . .	54
7.16	Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu .	55
7.17	Obraz wejściowy, histogram barw tego obrazu . . . . .	55
7.18	Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu .	55
7.19	Obraz wejściowy, histogram barw tego obrazu . . . . .	58
7.20	Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu . . . . .	58
7.21	Obraz wejściowy, histogram barw tego obrazu . . . . .	59
7.22	Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu . . . . .	59
7.23	Obraz wejściowy, histogram barw tego obrazu . . . . .	61
7.24	Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu . . . . .	62
7.25	Obraz wejściowy, histogram barw tego obrazu . . . . .	62
7.26	Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu . . . . .	62
10.1	Obraz wejściowy, obraz uśredniony . . . . .	67
10.2	Obraz wejściowy, obraz uśredniony . . . . .	67
10.3	Obraz wejściowy, obraz uśredniony . . . . .	68
10.4	Obraz wejściowy, obraz uśredniony . . . . .	68
10.5	Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	71
10.6	Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	72
10.7	Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	72
10.8	Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	72
10.9	Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	75
10.10	Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	76
10.11	Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	76
10.12	Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	76
10.13	Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	79
10.14	Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	80
10.15	Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	80
10.16	Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	80
10.17	Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	83
10.18	Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	84

10.19 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	84
10.20 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	84
10.21 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	89
10.22 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	89
10.23 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	89
10.24 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	90
10.25 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	97
10.26 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	97
10.27 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	97
10.28 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	98
10.29 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	102
10.30 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	102
10.31 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	103
10.32 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	103
10.33 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	106
10.34 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	106
10.35 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	107
10.36 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	107
10.37 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	110
10.38 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	110
10.39 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	111
10.40 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	111
10.41 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym . . . . .	114
10.42 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym . . . . .	115
10.43 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym . . . . .	115
10.44 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym . . . . .	115

## Rozdział 1

# Wstęp

Laboratoria oh oh... [1]

### 1.1 Specyfikacja wykorzystanego formatu obrazu

**Tagged Image File Format (TIFF or TIF)** jest formatem pliku komputerowego do przechowywania obrazów grafiki rastrowej (oraz osadzania elementów grafiki wektorowej). Jest rastrowym formatem uniwersalnym, tzn. może być zapisany w trybie kolorów:

- CMYK
- YCbCr
- CIELab
- RGB
- skala szarości
- kolor oparty na indeksowaniu
- kolor oparty na bitmapie z dowolną głębokością bitową

Również można dobrać dowolną rozdzielcość z opcją kanału przezroczystości alfa lub bez. Format TIFF wiele algorytmów kompresji bezstratnej:

- PackBits
- LZW (Lempel-Ziv-Welch)
- CCITT Fax group 3 & 4

Plik TIFF podzielony jest na trzy części:

1. nagłówek pliku obrazowego (Image File Header, IFH)
2. katalog pliku obrazowego (Image File Directory, IFD)
3. część danych obrazu

Plik TIFF może zawierać wiele obrazów. Nagłówek pliku składa się z ośmiu bajtów. Jak sama nazwa wskazuje, format TIFF używa struktur danych noszących nazwę Tag, definiujących cechy zawartych w nim obrazów. Np. dla obrazu 320x240 piksli, szerokość była by oznaczona tagiem "width", a wysokość "height", po którym następuje liczba 320 lub 240. Poniżej znajdują się trzy możliwe formy wewnętrznej struktury danych pliku TIFF (zawierającej trzy obrazy).

header	header	header
IFD 0	IFD 0	Image 0
IFD 1	Image 0	Image 1
IFD n	IFD 1	Image 2
Image 0	Image 1	IFD 0
Image 1	IFD 2	IFD 1
Image n	Image 2	IFD 2

W każdym przykładzie nagłówek pojawia się na początku. W pierwszym przykładzie katalog(IFD) jest zapisany na początku jeden za drugim, taki sposób pozwala na szybki odczyt danych.

W drugim przykładzie po każdym katalogu(IFD) są dane bitmapowe, co jest najczęściej spotykanym rozowiązaniem przy plikach z wieloma obrazami.

W ostatnim przykładzie najpierw są zapisane dane bitmapowe, a następnie katalogi(IFD). Aby sprawdzić, czy obraz jest zapisany w formacie TIFF wystarczy odczytać z nagłówka pierwsze 4 bajty, jeśli mają wartość 49h 49h 00h 2Ah lub 4Dh 4Dh 00h 2Ah, to jest to plik TIFF.

## 1.2 Intstrukcja obsługi programu

## Rozdział 2

# Operacje ujednolicania obrazów

Operacja ujednolicania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe. W tym programie ujednolicane są dwa obrazy rastrowe w taki sposób, że mniejszy doprowadzany jest do większego, przez co generowany jest nowy obraz o większe liczbie pikseli niż początkowo. Taki sposób ujednolicania nie powoduje widocznego spadku jakości.

## 2.1 Ujednolicenie obrazów szarych geometryczne

### Opis algorytmu

Operacja geometrycznego ujednolicenia obrazów polega na doprowadzeniu obydwu obrazów do takiej samej liczby wierszy pikseli w każdym obrazie i takiej samej liczby kolumn pikseli w każdym obrazie.

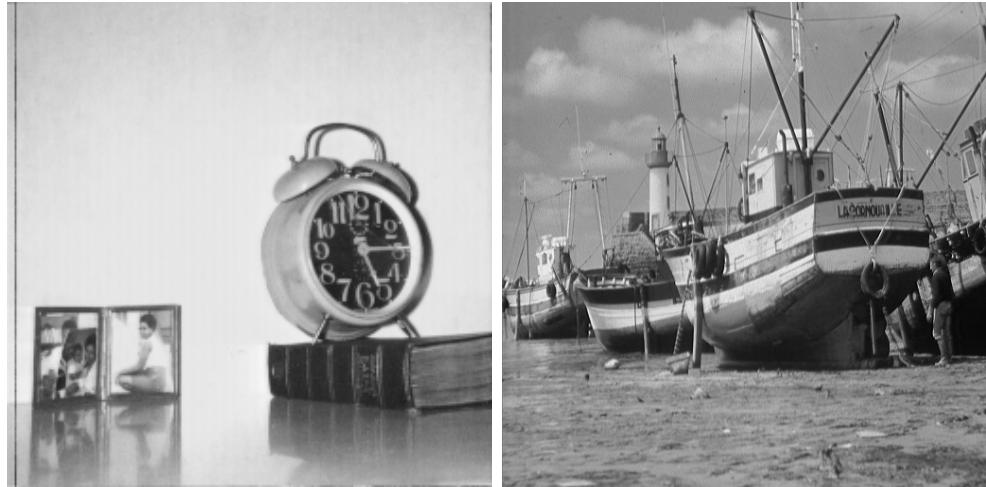
1. Wybierz największą wysokość i największą szerokość z dwóch obrazów.
2. Jeśli dany obraz ma mniejszą szerokość albo wysokość, wypełnij różnicę pikslami o wartości 1 (aby uniknąć dzielenia przez 0).



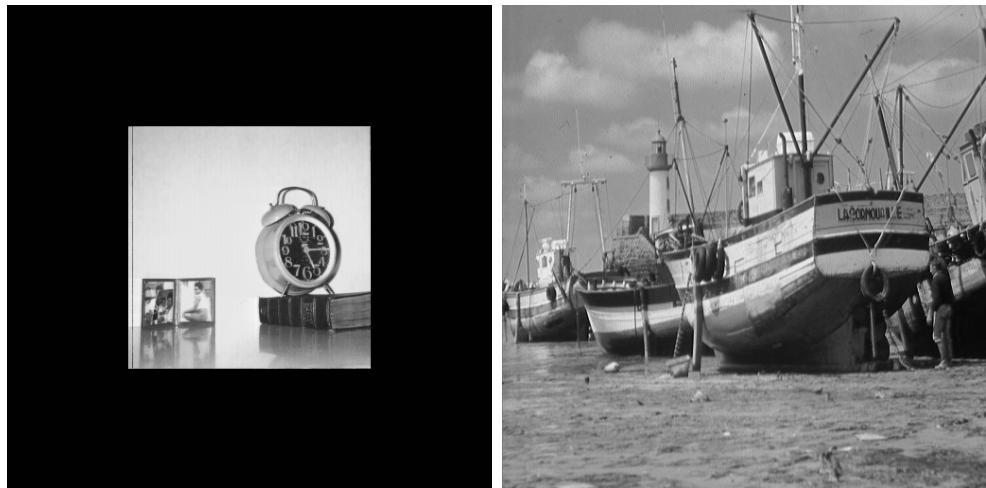
Rysunek 2.1: Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)



Rysunek 2.2: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.3: Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.4: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.1: Geometryczne ujednolicianie obrazów szarych

```

def geometricGray( self , show = False ) :
    # najwieksza szerokosc sposrod dwuch obrazow
    width1 = self .im1 .shape [1]
    width2 = self .im2 .shape [1]
    maxWidth = width1 if width1 > width2 else width2

    # najwieksza wysokosc sposrod dwuch obrazow
    height1 = self .im1 .shape [0]
    height2 = self .im2 .shape [0]
    maxHeight = height1 if height1 > height2 else height2

```

```

# alokacja pamieci na obrazy wynikowe
resultImage1 = np.empty((maxHeight, maxWidth), dtype = np.
    uint8)
resultImage2 = np.empty((maxHeight, maxWidth), dtype = np.
    uint8)

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width1) / 2))
startHeightCoord = int(round((maxHeight - height1) / 2))

# wypelnienie obrazu czarny kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage1[i, j] = 1

# narysowanie wysrodkowanego obrazu
for i in range(0, height1):
    for j in range(0, width1):
        resultImage1[i + startHeightCoord, j + startWidthCoord] =
            self.im1[i, j]

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width2) / 2))
startHeightCoord = int(round((maxHeight - height2) / 2))

# wypelnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage2[i, j] = 1

# narysowanie wysrodkowanego obrazu
for i in range(0, height2):
    for j in range(0, width2):
        resultImage2[i + startHeightCoord, j + startWidthCoord] =
            self.im2[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "L"), Image.
        fromarray(resultImage2, "L"))
    self.save(resultImage1, self.im1Name, "unificationGeo")
    self.save(resultImage2, self.im2Name, "unificationGeo")

```

## 2.2 Ujednolicenie obrazów szarych rozdzielczościowe

### Opis algorytmu

Operacja rozdzielczościowego ujednolicenia obrazów następuje po ujednoliceniu geometrycznym i polega na wypełnieniu obrazu pikslami, a brakujące piksele powinny być zinterpolowane.

1. Wypełnij cały obraz pikslami o znanej wartości zachowując pewien odstęp między nimi, gdzie odstępem będą piksele o wartości 0.
2. Każdemu pikselowi o nieznanej wartości przypisz średnią wartość znanych ( $> 0$ ) pikseli z jego bezpośredniego otoczenia.



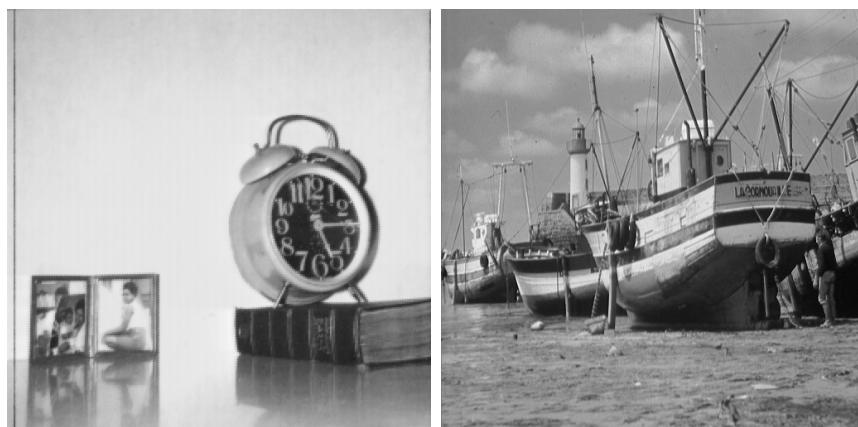
Rysunek 2.5: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.6: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.7: Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)



Rysunek 2.8: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.2: Rastrowe ujednolicianie obrazów szarych

```
def rasterGray( self , show = False ):
    # pierwszy jest ZAWSZE wiekszy
    width1 = self .im1 .shape [1]
    width2 = self .im2 .shape [1]

    height1 = self .im1 .shape [0]
    height2 = self .im2 .shape [0]

    scaleW = width1 / width2
    scaleH = height1 / height2

    # alokacja pamieci na obrazy wynikowe
    resultImage1 = np .zeros((height1 , width1) , dtype = np .uint8 )
```

```

resultImage2 = np.zeros((height1, width1), dtype = np.uint8)
tmp = np.zeros((height1, width1), dtype = np.uint8)

for i in range(height1):
    for j in range(width1):
        resultImage1[i, j] = self.im1[i, j]

# wypelnianie
count = 0
for i in range(height2):
    for j in range(width2):
        if count == 0:
            resultImage2[int(scaleH*i), int(round(scaleW*j)) + 1] =
                self.im2[i, j]
            count += 1
        if count == 1:
            resultImage2[int(round(scaleH*i)) + 1, int(scaleW*j)] =
                self.im2[i, j]
            count = 0

# interpolacja
for i in range(height1):
    for j in range(width1):
        value = 0
        n = 0
        tmp[i, j] = resultImage2[i, j]
        if resultImage2[i, j] < 1:
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height1 - 2)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width1 - 2)) | ((j +
                        jOff) < 0) else (j + jOff)
                    if resultImage2[iSafe, jSafe] > 0:
                        value += resultImage2[iSafe, jSafe]
                        n += 1
                    tmp[i, j] = value / n
                    resultImage2[i, j] = tmp[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "L"), Image.
              fromarray(resultImage2, "L"))
self.save(resultImage1, self.im1Name, "unificationRas")

```

```
self.save(resultImage2, self.im2Name, "unificationRas")
```

## 2.3 Ujednolicenie obrazów RGB geometryczne

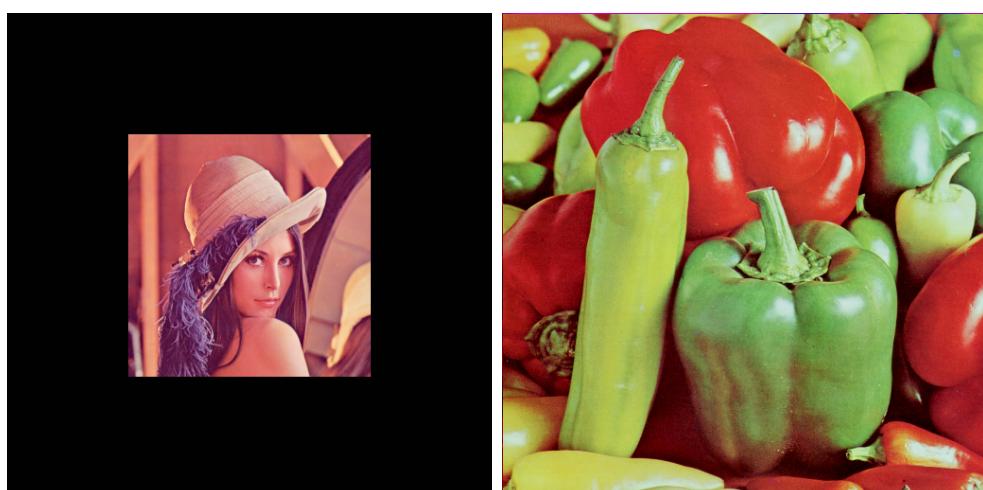
### Opis algorytmu

Operacja geometrycznego ujednolicenia obrazów polega na doprowadzeniu obydwu obrazów do takiej samej liczby wierszy pikseli w każdym obrazie i takiej samej liczby kolumn pikseli w każdym obrazie.

1. Wybierz największą wysokość i największą szerokość z dwóch obrazów.
2. Jeśli dany obraz ma mniejszą szerokość albo wysokość, wypełnij różnicę pikslami o wartości 1 dla każdego z kanałów R, G i B (aby uniknąć dzielenia przez 0).



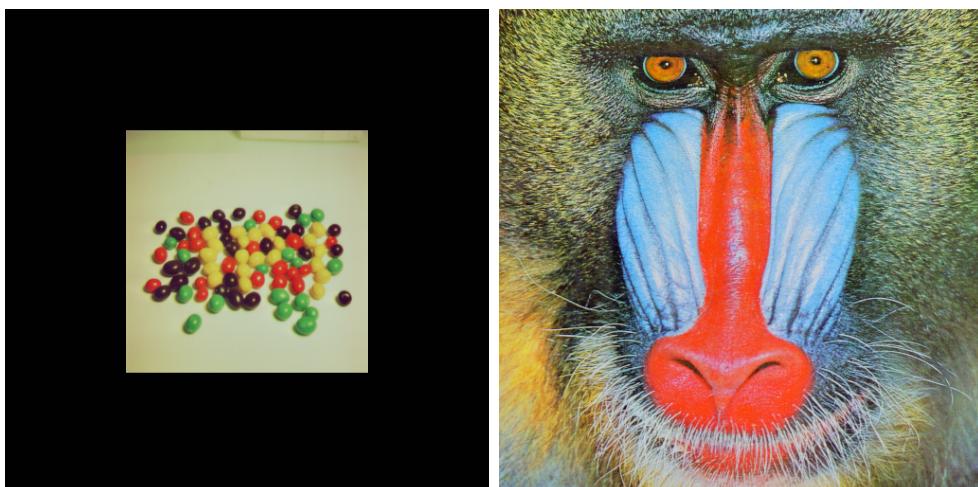
Rysunek 2.9: Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)



Rysunek 2.10: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.11: Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.12: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.3: Geometryczne ujednolicianie obrazów barwnych

```
def geometricColor(self , show = False):
    # najwieksza szerokosc sposrod dwuch obrazow
    width1 = self.im1.shape[1]
    width2 = self.im2.shape[1]
    maxWidth = width1 if width1 > width2 else width2

    # najwieksza wysokosc sposrod dwuch obrazow
    height1 = self.im1.shape[0]
    height2 = self.im2.shape[0]
    maxHeight = height1 if height1 > height2 else height2
```

```

# alokacja pamieci na obrazy wynikowe
resultImage1 = np.empty((maxHeight, maxWidth, 3), dtype = np.
    uint8)
resultImage2 = np.empty((maxHeight, maxWidth, 3), dtype = np.
    uint8)

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width1) / 2))
startHeightCoord = int(round((maxHeight - height1) / 2))

# wypelnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage1[i, j] = (1, 1, 1)

# narysowanie wysrodkowanego obrazu
for i in range(0, height1):
    for j in range(0, width1):
        resultImage1[i + startHeightCoord, j + startWidthCoord] =
            self.im1[i, j]

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width2) / 2))
startHeightCoord = int(round((maxHeight - height2) / 2))

# wypelnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage2[i, j] = (1, 1, 1)

# narysowanie wysrodkowanego obrazu
for i in range(0, height2):
    for j in range(0, width2):
        resultImage2[i + startHeightCoord, j + startWidthCoord] =
            self.im2[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "RGB"), Image.
        fromarray(resultImage2, "RGB"))
self.save(resultImage1, self.im1Name, "unificationGeo")
self.save(resultImage2, self.im2Name, "unificationGeo")

```

## 2.4 Ujednolicenie obrazów RGB rozdzielczościowe

### Opis algorytmu

Operacja rozdzielczościowego ujednolicenia obrazów następuje po ujednoliceniu geometrycznym i polega na wypełnieniu obrazu pikslami, a brakujące piksele powinny być zinterpolowane.

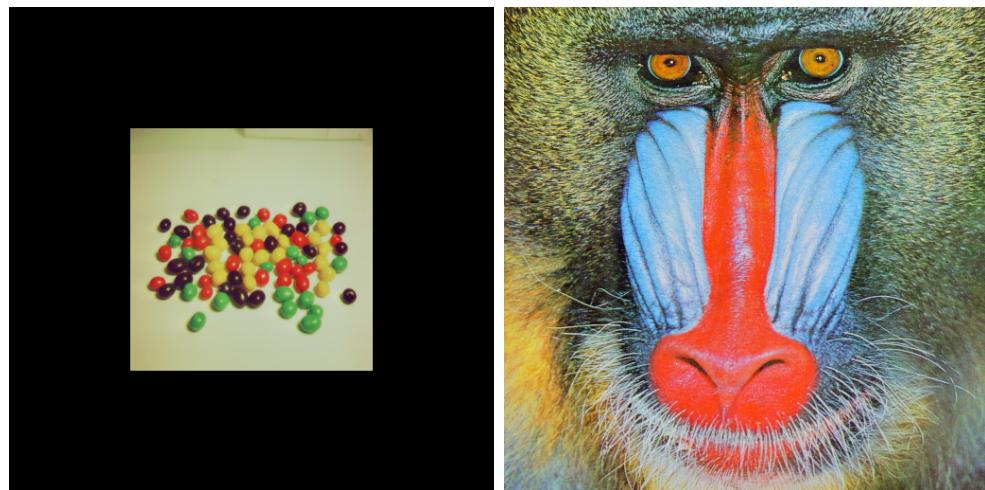
1. Wypełnij cały obraz pikslami o znanej wartości zachowując pewien odstęp między nimi.
2. Każdemu pikselowi o nieznanej wartości przypisz zinterpolowaną wartość (dla każdego z kanałów R, G, B) znanych piksli z jego bezpośredniego otoczenia.



Rysunek 2.13: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.14: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.15: Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)



Rysunek 2.16: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.4: Rastrowe ujednolicianie obrazów barwnych

```
def rasterColor(self, show = False):
    # pierwszy jest ZAWSZE wiekszy
    width1 = self.im1.shape[1]
    width2 = self.im2.shape[1]

    height1 = self.im1.shape[0]
    height2 = self.im2.shape[0]

    scaleW = width1 / width2
    scaleH = height1 / height2
```

```

# alokacja pamieci na obrazy wynikowe
resultImage1 = np.zeros((height1, width1, 3), dtype = np.
    uint8)
resultImage2 = np.zeros((height1, width1, 3), dtype = np.
    uint8)
tmp = np.zeros((height1, width1, 3), dtype = np.uint8)

for i in range(height1):
    for j in range(width1):
        resultImage1[i, j] = self.im1[i, j]

# wypelnianie
count = 0
for i in range(height2):
    for j in range(width2):
        if count == 0:
            resultImage2[int(scaleH*i), int(round(scaleW*j)) + 1] =
                self.im2[i, j]
        count += 1
        if count == 1:
            resultImage2[int(round(scaleH*i)) + 1, int(scaleW*j)] =
                self.im2[i, j]
        count = 0

# interpolacja
for i in range(height1):
    for j in range(width1):
        r, g, b = 0, 0, 0
        n = 0
        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) > (height1 - 2)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width1 - 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
            r /= n
            g /= n
            b /= n
        resultImage1[i, j] = [r, g, b]

```

```
    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 ]

if show:
    self . show( Image . fromarray( resultImage1 , "RGB" ) , Image .
        fromarray( resultImage2 , "RGB" ) )
    self . save( resultImage1 , self . im1Name , "unificationRas" )
    self . save( resultImage2 , self . im2Name , "unificationRas" )
```

## Rozdział 3

# Operacje sumowania arytmetycznego obrazów szarych

1. Sumowanie (określonej) stałej z obrazem oraz dwóch obrazów
2. Mnożenie obrazu przez zadawaną liczbę oraz przez inny obraz
3. Mieszanie obrazów z określonym współczynnikiem
4. Potęgowanie obrazu (z zadanaą potegą)
5. Dzielenie obrazu przez (zadaną) liczbę oraz przez inny obraz
6. Pierwiastkowanie obrazu
7. Logarytmowanie obrazu

## Rozdział 4

# Operacje sumowania arytmetycznego obrazów barwowych

1. sumowanie (określonej) stałej z obrazem oraz dwóch obrazów
2. mnożenie obrazu przez zadaną liczbę oraz przez inny obraz
3. mieszanie obrazów z określonym współczynnikiem
4. potęgowanie obrazu (z zadaną potęgą)
5. dzielenie obrazu przez (zadaną) liczbę oraz przez inny obraz
6. pierwiastkowanie obrazu
7. logarytmowanie obrazu

## Rozdział 5

# Operacje geometryczne na obrazie

1. przemieszczenie obrazu o zadany wektor
2. jednorodne i niejednorodne skalowanie obrazu
3. obracanie obrazu o dowolny kąt
4. symetrie względem osi układu i zadanej prostej
5. wycinanie fragmentów obrazu
6. kopiowanie fragmentów obrazów

## Rozdział 6

# Operacje na histogramie obrazu szarego

Histogram to najprostszy opis całościowy obrazu. Dlatego stosuje się go, by rozpoznać, jakie dalsze metody i operacje należy zastosować na przetwarzanym obrazie, by osiągnąć założony cel. Jego obliczenie polega na odczytaniu szarości każdego piksla i rejestraniu jej wystąpienia o danym poziomie.

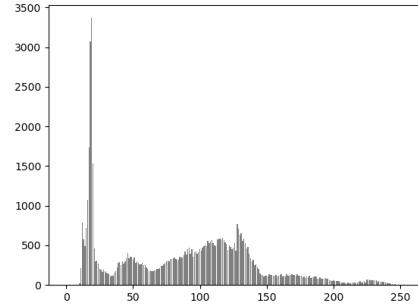
Histogram to funkcja przypisująca każdemu poziomowi szarości obrazu, liczbę piksli z danym poziomem szarości, czyli jest to wykres częstości występowania wartości piksli w obrazie.

## 6.1 Obliczanie histogramu

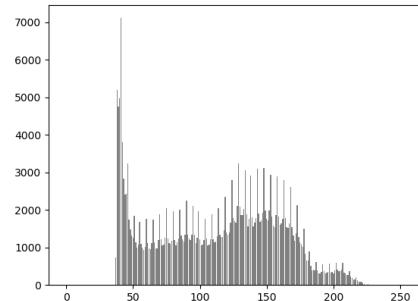
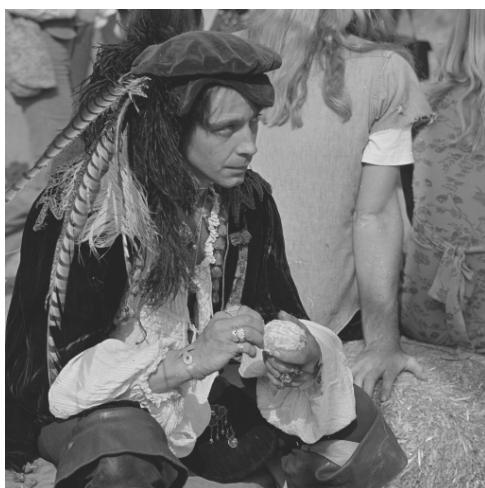
### Opis algorytmu

Histogram obrazu szarego jest wykresem częstości występowania wartości szarości piksli w obrazie tj. przyporządkowuje liczbę pikseli do danego poziomu szarości.

1. Zaalokuj tablicę 256 elementową (tyle, ile poziomów szarości w obrazie)
2. Dla każdego piksla:
  3. Zwięksź element tablicy o indeksie równym pozycji szarości danego piksla



Rysunek 6.1: Obraz szary, histogram szarości tego obrazu



Rysunek 6.2: Obraz szary, histogram szarości tego obrazu

Listing 6.1: Obliczanie histogramu

```
def calculate(self, plot = False, image = None):
    if image is None:
        image = self.im

    width = image.shape[1]          # szereoksc
    height = image.shape[0]         # wysokosc
    hist = [0] * 256               # histogram szarosci

    for i in range(height):
        for j in range(width):
            bin = image[i, j]      # odcien szarosci
            hist[bin] += 1

    if plot:
        # tablica [0, 1, ... , 254, 255]
        bins = np.arange(256)
        self.plotHistogram(bins, hist)

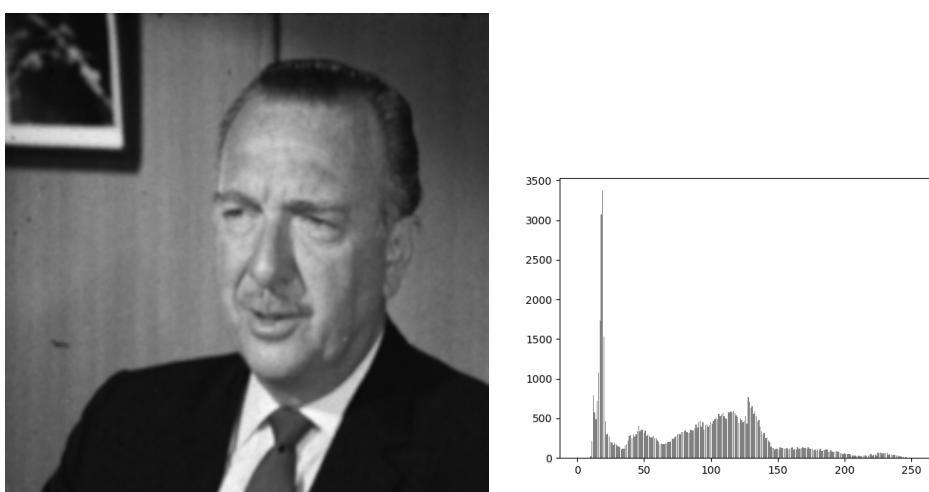
    return hist
```

## 6.2 Przemieszczanie histogramu

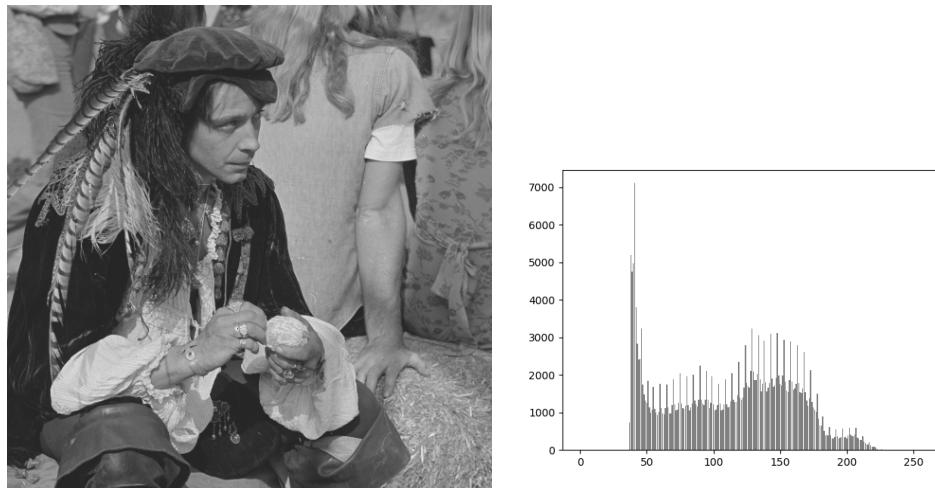
### Opis algorytmu

Przemieszczenie histogramu polega na dodaniu lub odjęciu tej samej wartości od poziomu szarości każdego piksla w obrazie. W rezultacie obraz jest odpowiednio równomiernie rozjaśniony bądź przyciemniony. Nie można przekroczyć przyjętego zakresu poziomów szarości.

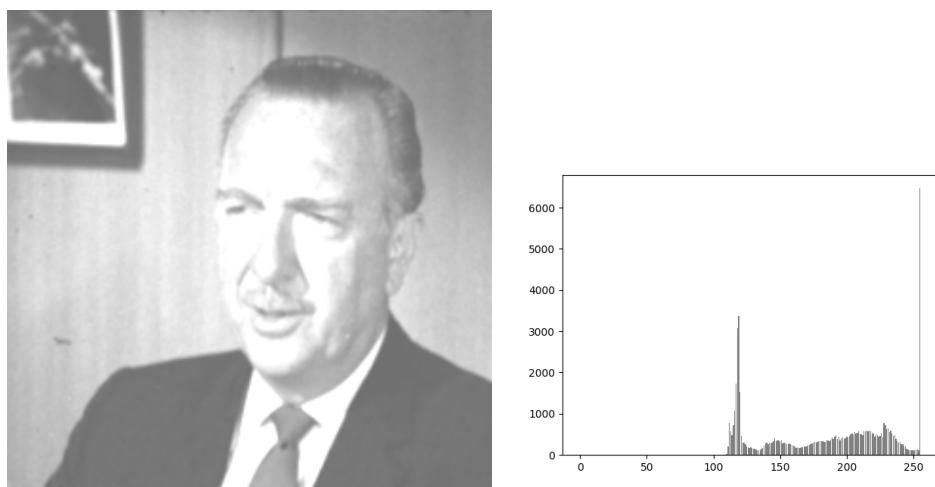
1. Do każdej wartości piksla dodaj podaną stałą o którą chcesz przemieścić histogram.
2. Jeśli wartość piksla po operacji dodawania wykracza poza zakres 255:
  3. Przypisz jej wartość 255.
4. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0:
  5. Przypisz jej wartość 0.



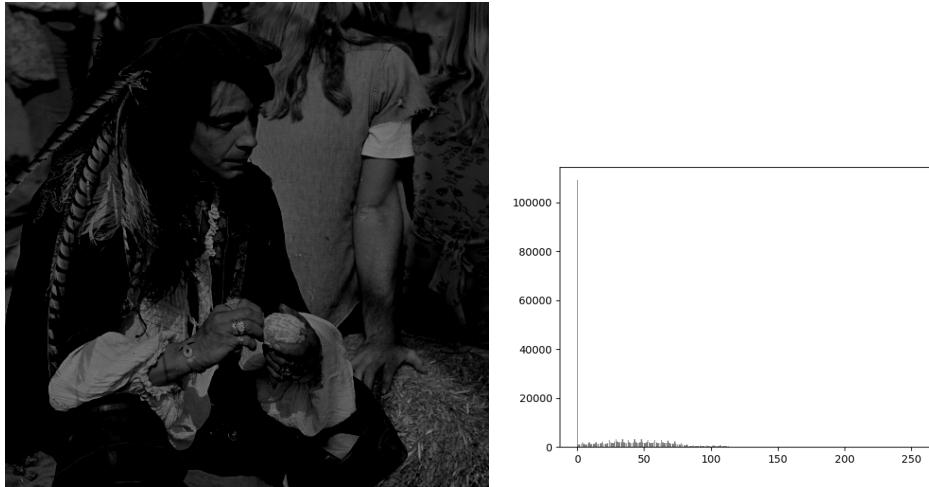
Rysunek 6.3: Obraz szary wejściowy, histogram szarości tego obrazu



Rysunek 6.5: Obraz szary wejściowy, histogram szarości tego obrazu



Rysunek 6.4: Obraz szary przesunięty o 100, histogram szarości tego obrazu



Rysunek 6.6: Obraz szary pryesunięty o -100, histogram szarości tego obrazu

Listing 6.2: Przemieszczanie histogramu

```

def move( self , const = 0 , show = False , plot = False ):
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    # przemieszczanie
    for i in range(height):
        for j in range(width):
            value = int(self .im [i , j ]) + const
            if value < 0:
                value = 0
            elif value > 255:
                value = 255
            resultImage [i , j ] = value

    if show:
        self .show(Image .fromarray(resultImage , "L"))
    self .calculate(plot , resultImage)
    self .save(resultImage , self .imName , "moveHist")

```

## 6.3 Rozciąganie histogramu

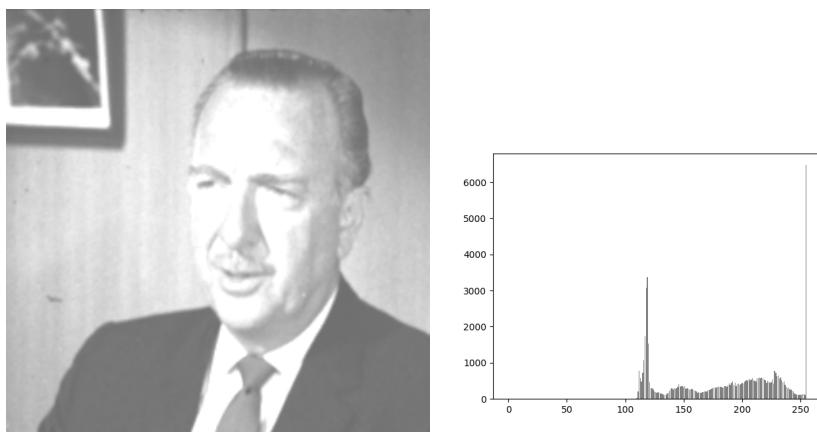
### Opis algorytmu

Rozciągania histogramu dokonuje się na obrazie, którego poziomy szarości nie są rozpięte na cały możliwy zakres np. [51, 233]. Operacja rozciągnięcia histogramu rozciągnie histogram tak, aby był rozpięty na cały możliwy zakres poziomów szarości np. [0, 255].

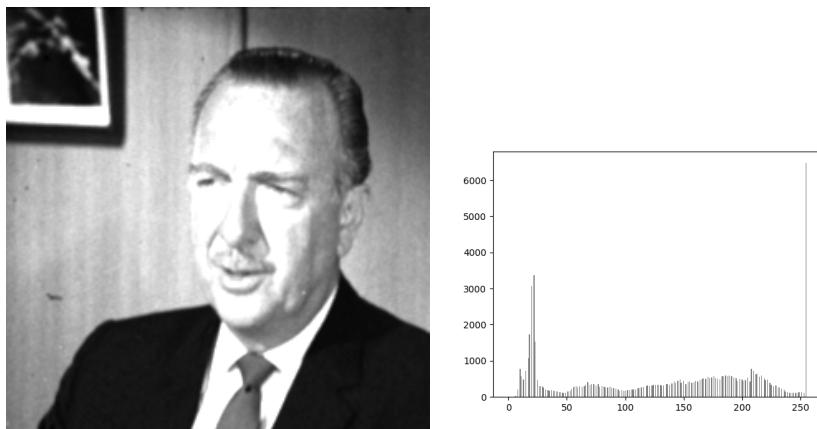
1. Znajdź w obrazie największą( $\max$ ) i najmniejszą( $\min$ ) wartość piksla
2. Dla każdego piksla:
3. Oblicz nową wartość piksla stosując wzór:

$$P_n = 255 / (\max - \min) * (P_o - \min).$$

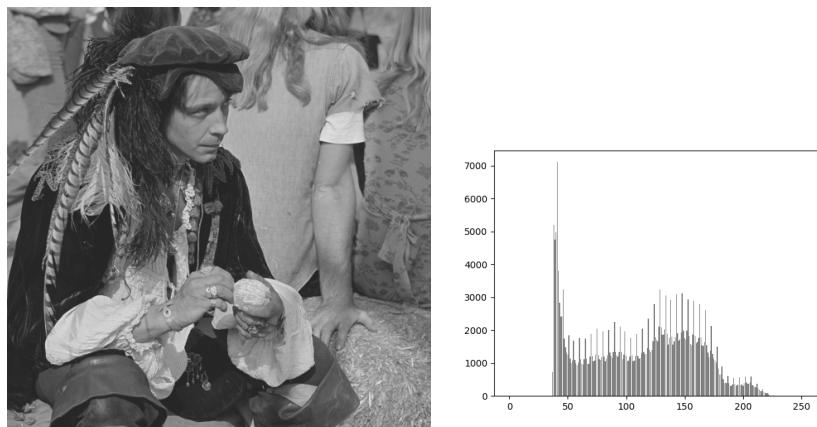
W taki sposób, jeśli odcienie szarości obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, odcienie szarości będą w zakresie [0, 255]



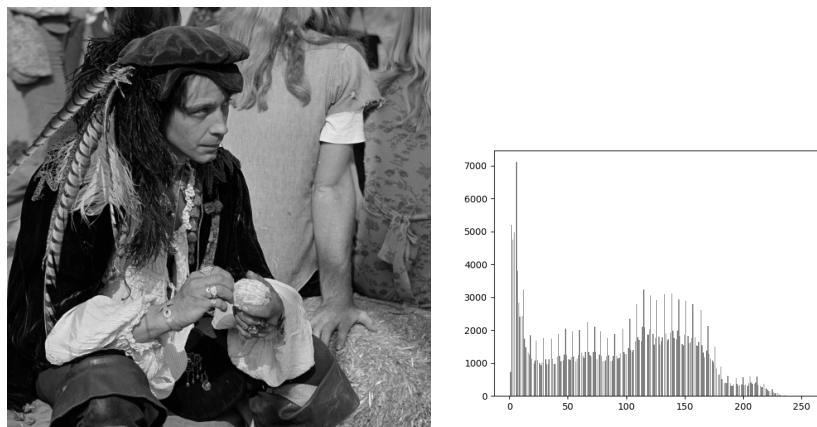
Rysunek 6.7: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.8: Obraz po rozciagnięciu, histogram szarości tego obrazu



Rysunek 6.9: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.10: Obraz po rozciagnięciu, histogram szarości tego obrazu

Listing 6.3: Rozciąganie histogramu

```
def stretch( self , show = False , plot = False ) :
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)
    for i in range(height):
        for j in range(width):
            resultImage [i , j] = self .im [i , j]

    # wartosc min i max w obrazie
    maxValue = 0
    minValue = 255
    while maxValue != 255:
```

```
# wartosci max i min w obrazie
for i in range(height):
    for j in range(width):
        currValue = resultImage[i, j]
        maxValue = max(maxValue, currValue)
        minValue = min(minValue, currValue)

# rozciąganie
for i in range(height):
    for j in range(width):
        pix = resultImage[i, j]
        resultImage[i, j] = ((255 / (maxValue - minValue)) * (pix
            - minValue))

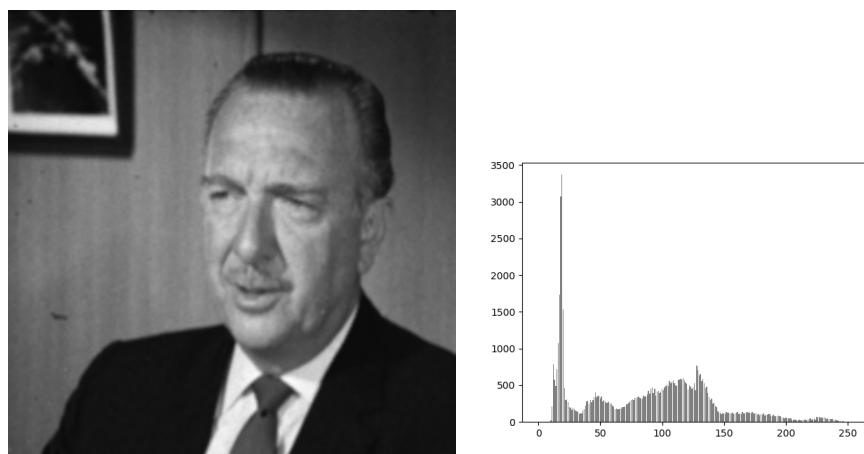
if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "stretchHist")
```

## 6.4 Progowanie lokalne

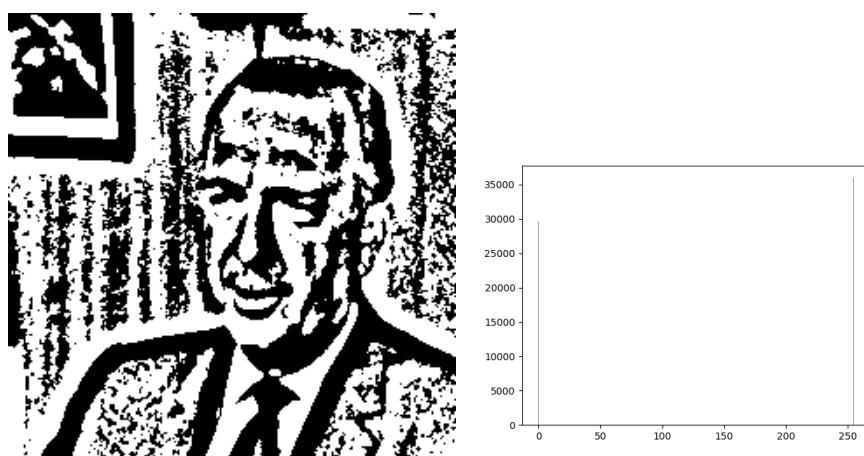
### Opis algorytmu

Progowanie lokalne oblicza wartość progową dla każdego piksla z osobna. Jest to jedna z metod binaryzacji obrazu, która w wyniku dokładniej odwzorowuje kształt obiektu na obrazie.

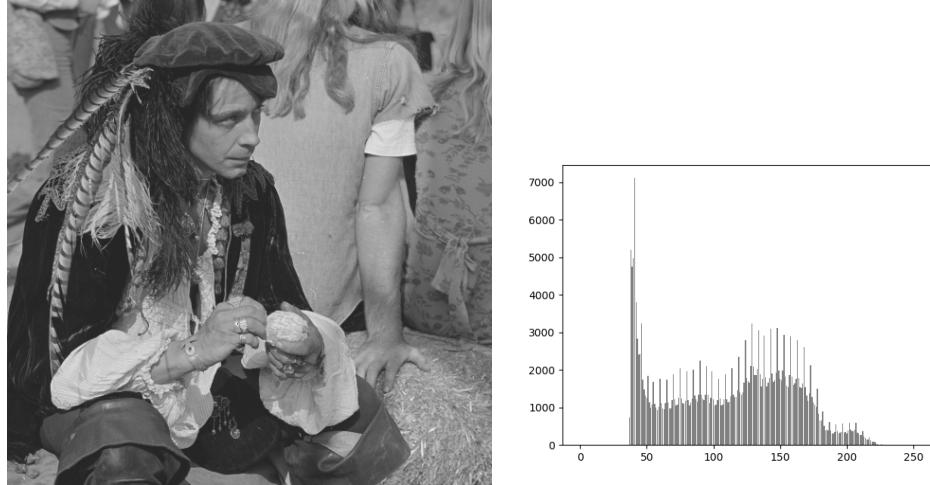
1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla:
3. Oblicz wartość progową jako średnią wartość piksli w otoczeniu danego piksla.
4. Jeśli wartość piksla środkowego jest  $< 0$ :
5. Przypisz mu wartość 0.
6. W przeciwnym przypadku przypisz mu wartość 255.



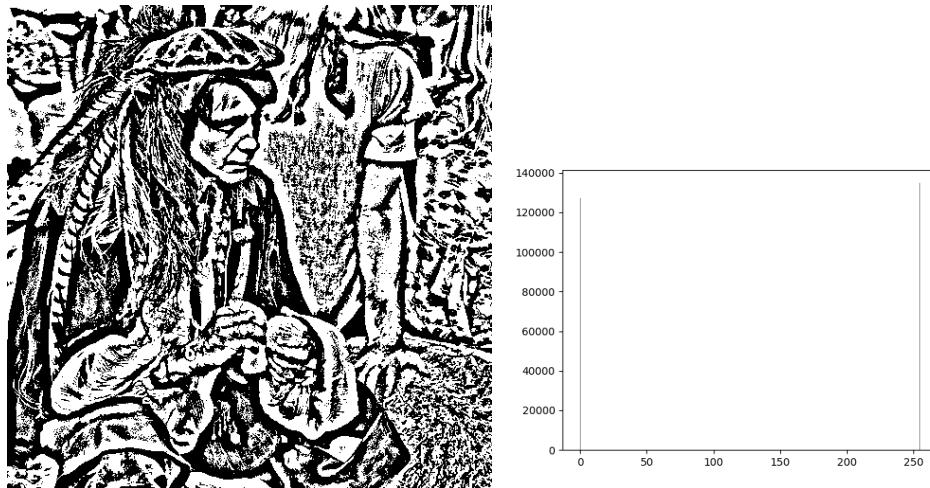
Rysunek 6.11: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.12: Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu



Rysunek 6.13: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.14: Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu

Listing 6.4: Progowanie lokalne

```

def localThreshold(self , dim = 3, show = False, plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]    # wysokosc
    l , r = -(int(round(dim / 2))), int(round(dim / 2) + 1)  # wsp
        . sasiadow

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width) , dtype=np.uint8)

# progowanie lokalne
for i in range(height):

```

```
for j in range(width):
    n = 0
    threshold = 0
    currPix = self.im[i, j]
    for iOff in range(1, r):
        for jOff in range(1, r):
            iSafe = i if ((i + iOff) > (height + 1)) else (i + iOff)
            jSafe = j if ((j + jOff) > (width + 1)) else (j + jOff)
            threshold += self.im[iSafe, jSafe]
            n += 1
    threshold = int(round(threshold / n))
    resultImage[i, j] = 0 if (currPix < threshold) else 255

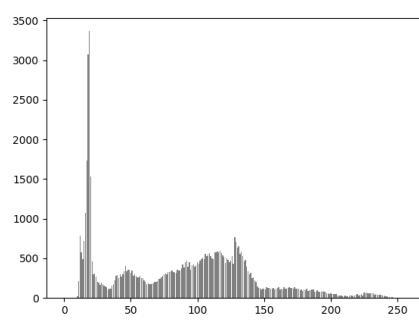
if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "locThreshold")
```

## 6.5 Progowanie globalne

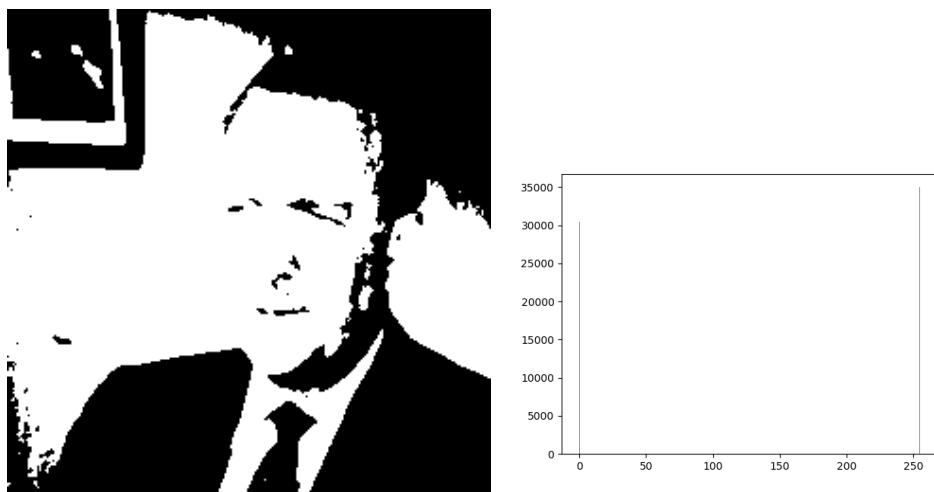
### Opis algorytmu

Progowanie globalne jest jedną z metod binaryzacji obrazu. Wartość progowa jest ustalana globalnie biorąc pod uwagę wartość każdego piksla w obrazie, po czym stosując wyliczony próg aby nadać nową wartość każdemu poikslowi. Obraz w wyniku jest binarny.

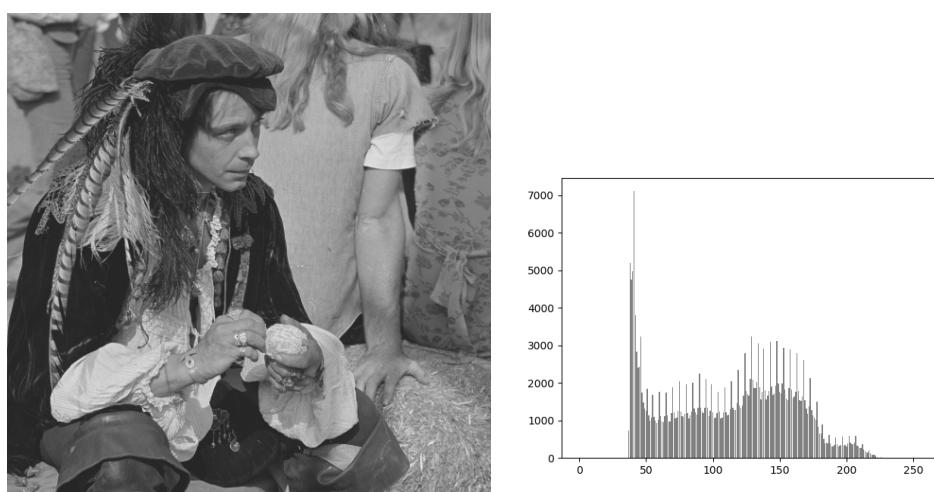
1. Oblicz wartość progową  $T$ , jako średnią wartość z wszystkich piskli w obrazie
2. Dla każdego piskla:
  3. Jeśli wartość danego piskla jest  $< T$ :
  4. przypisz mu wartość 0.
  5. W przeciwnym przypadku przypisz mu wartość 255.



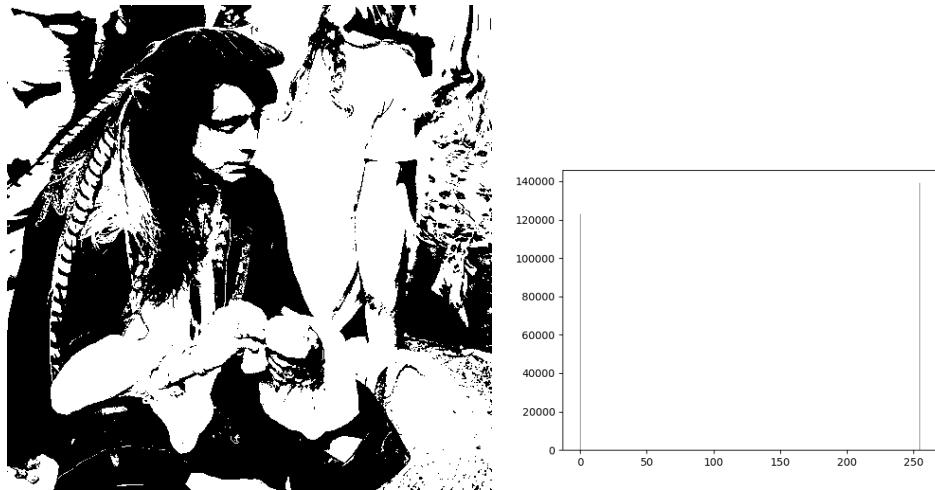
Rysunek 6.15: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.16: Obraz po progowaniu globalnym, histogram szarości tego obrazu



Rysunek 6.17: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.18: Obraz po progowaniu globalnym, histogram szarości tego obrazu

Listing 6.5: Progowanie globalne

```

def globalThreshold(self , show = False , plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width) , dtype=np.uint8)

    # prog globalny
    threshold = 0
    n = 0
    for i in range(height):
        for j in range(width):
            threshold += self.im[i , j]
            n += 1
            threshold = int(round(threshold / n))

    # binarizacja
    for i in range(height):
        for j in range(width):
            resultImage[i , j] = 0 if (self.im[i , j] < threshold) else
                255

    if show:
        self.show(Image.fromarray(resultImage , "L"))
    self.calculate(plot , resultImage)
    self.save(resultImage , self.imName , "globThreshold")

```

## Rozdział 7

# Operacje na histogramie obrazu barwowego

Histogram to najprostszy opis całościowy obrazu. Dlatego stosuje się go, by rozpoznać, jakie dalsze metody i operacje należy zastosować na przetwarzanym obrazie, by osiągnąć założony cel. Jego obliczenie polega na odczytaniu barwy każdego piksla i rejestraniu jej wystąpienia o danym poziomie.

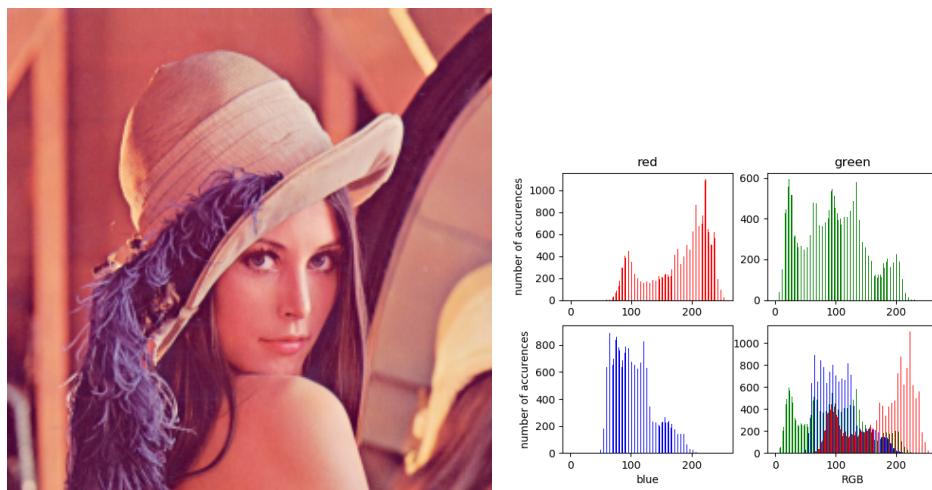
Histogram to funkcja przypisująca każdemu poziomowi barwy obrazu, liczbę piksli z danym poziomem barwy, czyli jest to wykres częstości występowania wartości piksli w obrazie.

## 7.1 Obliczanie histogramu

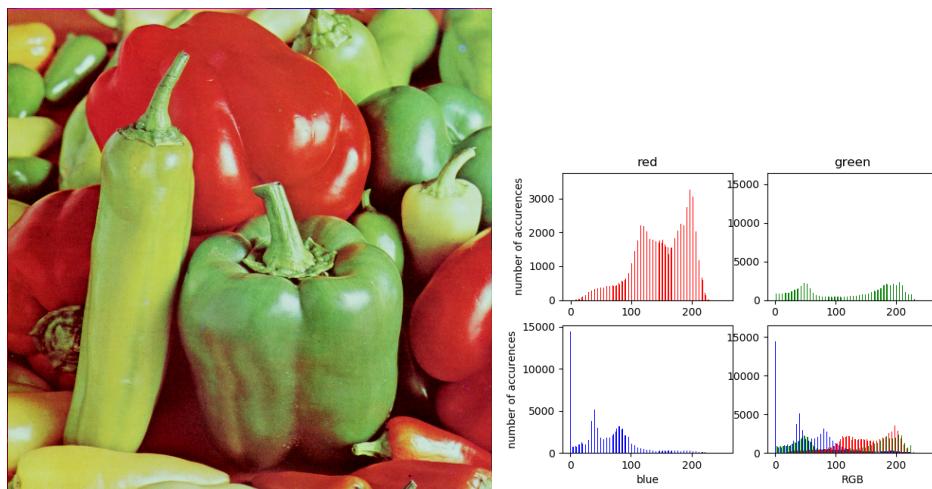
### Opis algorytmu

Histogram obrazu barwnego jest wykresem częstości występowania wartości barwy piksli w obrazie tj. przyporządkowuje liczbę pikseli do danego poziomu barwy.

1. Zaalokuj 3 tablice 256 elementowe (tyle, ile poziomów barw w obrazie)
2. Dla każdego piksela:
3. Dla każdej barwy:
4. Zwięksź element tablicy danej barwy o indeksie równym poziomie tej barwy danego piksela



Rysunek 7.1: Obraz barwny, histogram barw tego obrazu



Rysunek 7.2: Obraz barwny, histogram barw tego obrazu

Listing 7.1: Obliczanie histogramu

```

def calculate(self , plot = False , image = None):
    if image is None:
        image = self.im

    width = image.shape[1]           # szerokosc
    height = image.shape[0]          # wysokosc
    hist = [0] * 3                  # histogram RGB
    hist[0] = [0] * 256              # histogram R
    hist[1] = [0] * 256              # histogram G
    hist[2] = [0] * 256              # histogram B

    for i in range(height):
        for j in range(width):
            bin = image[i , j]
            for k in range(3):
                hist[k][bin[k]] += 1

    if plot:
        # tablica [0, 1, ... , 254, 255]
        bins = np.arange(256)
        self.plotHistogram(bins , hist)

    return hist                   # [0] - hist R, [1] - hist G, [2]
                                  - hist B

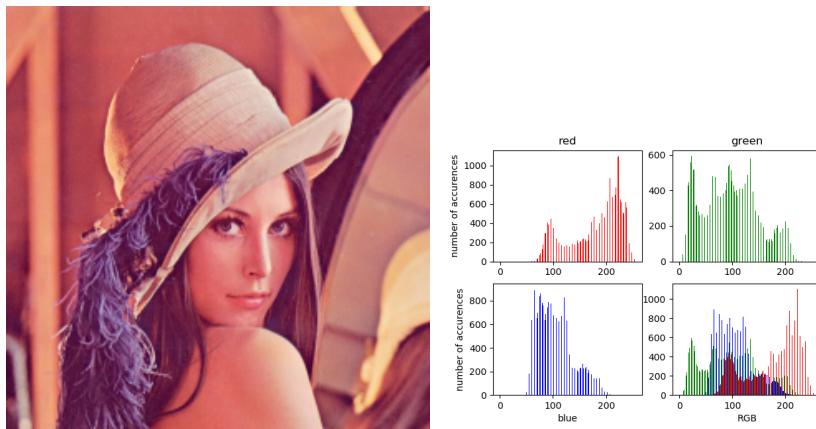
```

## 7.2 Przemieszczanie histogramu

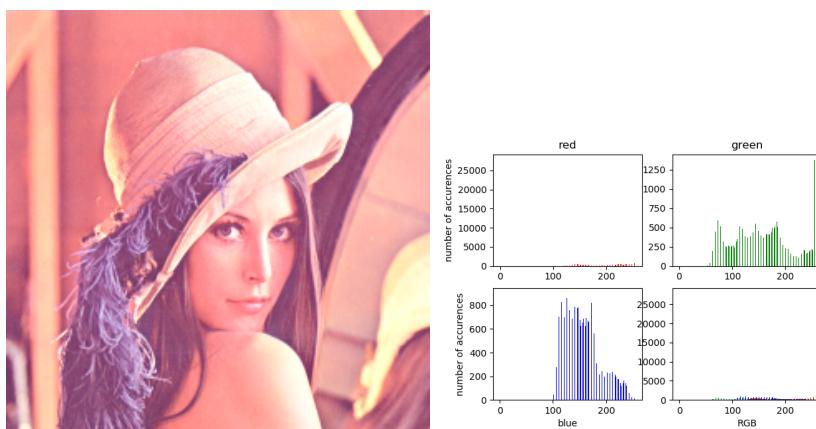
### Opis algorytmu

Przemieszczenie histogramu polega na dodaniu lub odjęciu tej samej wartości od poziomu każdej z barw każdego piksla w obrazie. W rezultacie obraz jest odpowiednio równomiernie rozjaśniony bądź przyciemniony. Nie można przekroczyć przyjętego zakresu poziomu barwy.

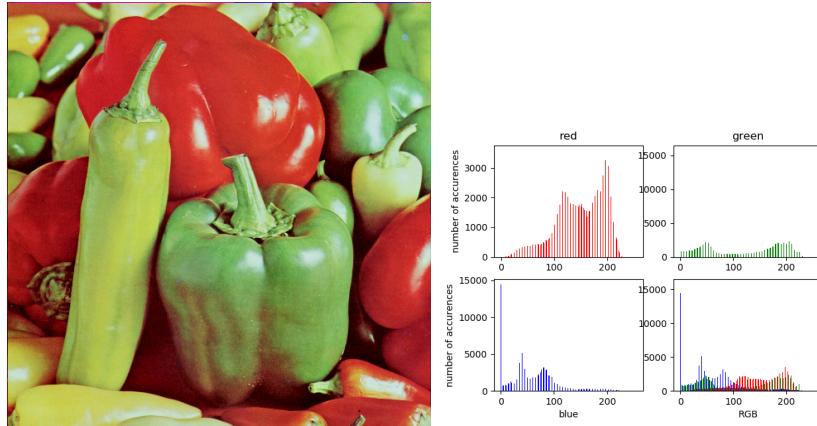
1. Do każdej wartości barwy piksla dodaj podaną stałą, o którą chcesz przemieścić histogram.
2. Jeśli wartość barwy piksla po operacji dodawania wykracza poza zakres 255: Przypisz jej wartość 255.
3. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0: Przypisz jej wartość 0.



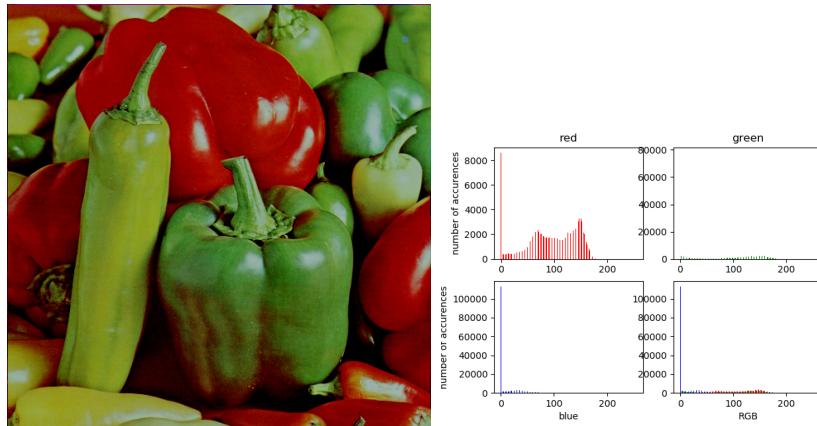
Rysunek 7.3: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.4: Obraz wyjściowy przesunięty o 50, histogram barw tego obrazu



Rysunek 7.5: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.6: Obraz wyjściowy przesunięty o -50, histogram barw tego obrazu

Listing 7.2: Przemieszczanie histogramu

```

def move( self , const = 0 , show = False , plot = False ):
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty (( height , width , 3 ) , dtype=np .uint8 )

    # przemieszczanie
    for i in range (height):
        for j in range (width):
            value = self .im [i , j]
            for k in range (len (value)):
                v = value [k]
                v += const

```

```
if v < 0:  
    v = 0  
elif v > 255:  
    v = 255  
value[k] = v  
resultImage[i, j] = value  
  
if show:  
    self.show(Image.fromarray(resultImage, "RGB"))  
self.calculate(plot, resultImage)  
self.save(resultImage, self.imName, "moveHist")
```

## 7.3 Rozciąganie histogramu

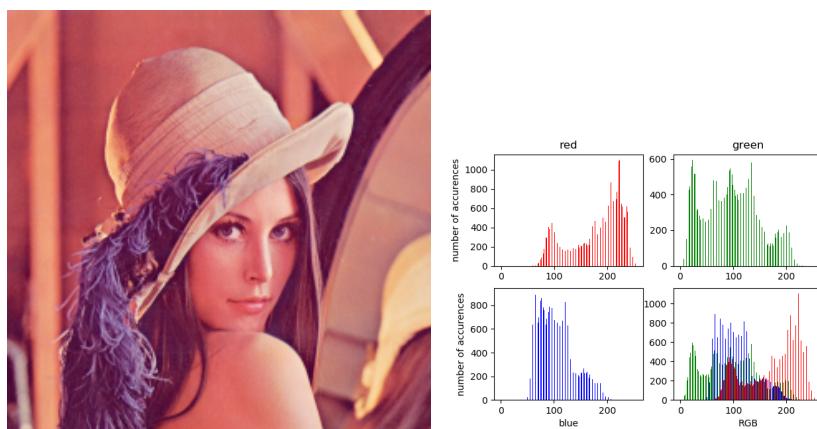
### Opis algorytmu

Rozciągania histogramu dokonuje się na obrazie, którego poziomy barw nie są rozpięte na cały możliwy zakres np. [51, 233]. Operacja rozciągnięcia histogramu rozciągnie histogram tak, aby był rozpięty na cały możliwy zakres poziomów barw np. [0, 255].

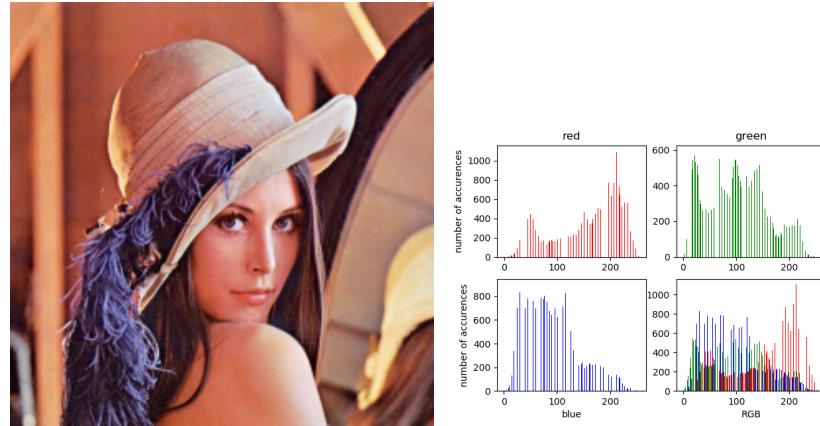
1. Znajdź w obrazie największą( $\max_c$ ) i najmniejszą( $\min_c$ ) wartość piksla dla każdej z barw( $c$ )
2. Dla każdego piksla( $P_o$ ):
3. Dla każdej z barw( $c$ ):
4. Oblicz nową wartość piksla( $P_n$ ) stosując wzór:

$$P_n = 255 / (\max_c - \min_c) * (P_o - \min_c).$$

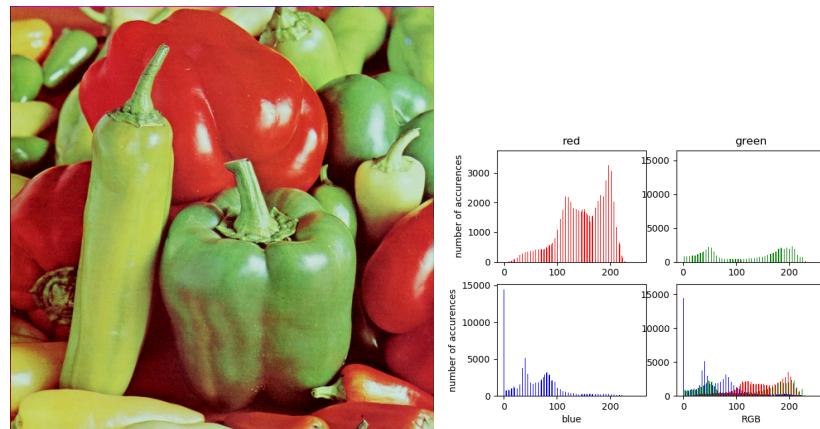
W taki sposób, jeśli barwy obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, będą w zakresie [0, 255].



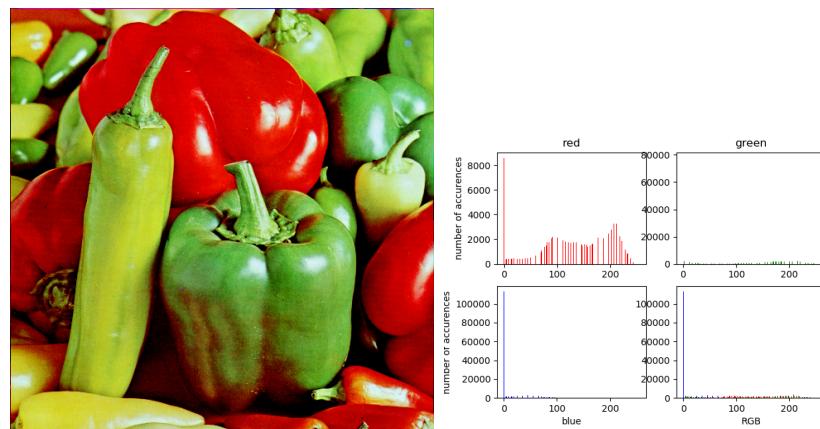
Rysunek 7.7: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.8: Obraz po rozciagnięciu, histogram barw tego obrazu



Rysunek 7.9: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.10: Obraz po rozciagnięciu, histogram barw tego obrazu

Listing 7.3: Rozciąganie histogramu

```

def stretch(self , show = False , plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)
for i in range(height):
    for j in range(width):
        resultImage[i , j] = self.im[i , j]

# wartosci min i max w obrazie
maxValue = [0] * 3
minValue = [255] * 3
while (maxValue[0] != 255) & (maxValue[1] != 255) & (maxValue[2] != 255):
# wartosci max i min w obrazie
for i in range(height):
    for j in range(width):
        currValue = resultImage[i , j]
        for k in range(3):
            maxValue[k] = max(maxValue[k] , currValue[k])
            minValue[k] = min(minValue[k] , currValue[k])

# rozciaganie
for i in range(height):
    for j in range(width):
        pix = resultImage[i , j]
        for k in range(3):
            pix[k] = ((255 / (maxValue[k] - minValue[k])) * (pix[k] - minValue[k]))
        resultImage[i , j] = pix

if show:
    self.show(Image.fromarray(resultImage , "RGB"))
    self.calculate(plot , resultImage)
    self.save(resultImage , self.imName , "stretchHist")

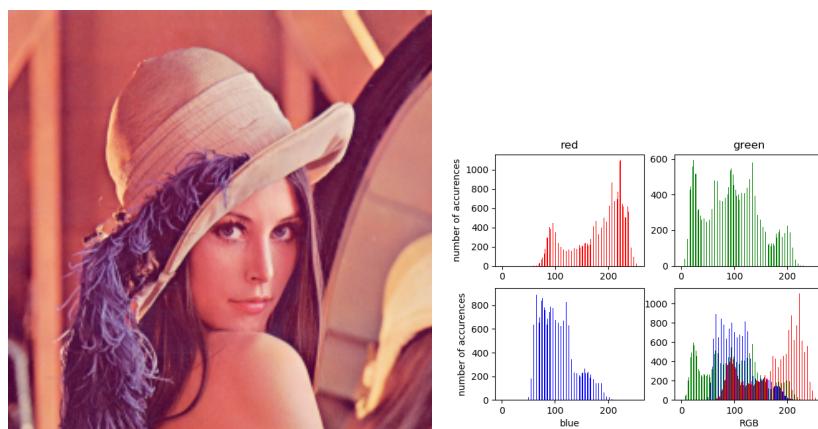
```

## 7.4 Progowanie 1-progowe lokalne

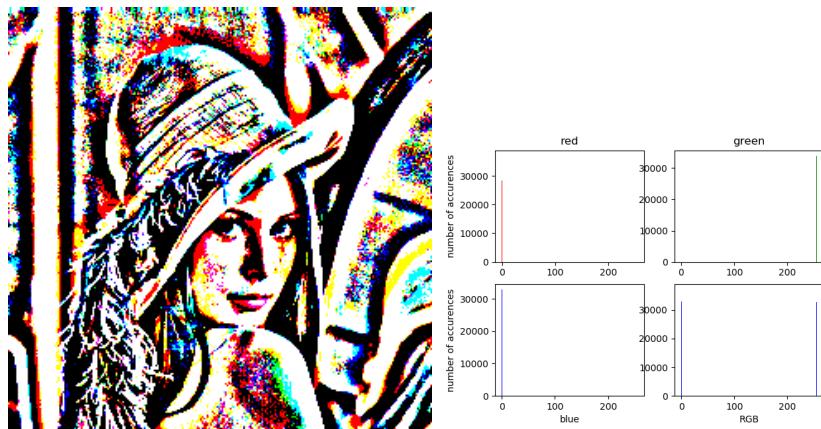
### Opis algorytmu

Progowanie 1-progowe lokalne oblicza wartość progową dla każdego piksla z osobna. W wyniku takiego progowania obraz dokładniej odwzorowuje kształt obiektu. Próg obliczany jest jako średnia wartość piksli w obrazie, dla każdego kanału z osobna.

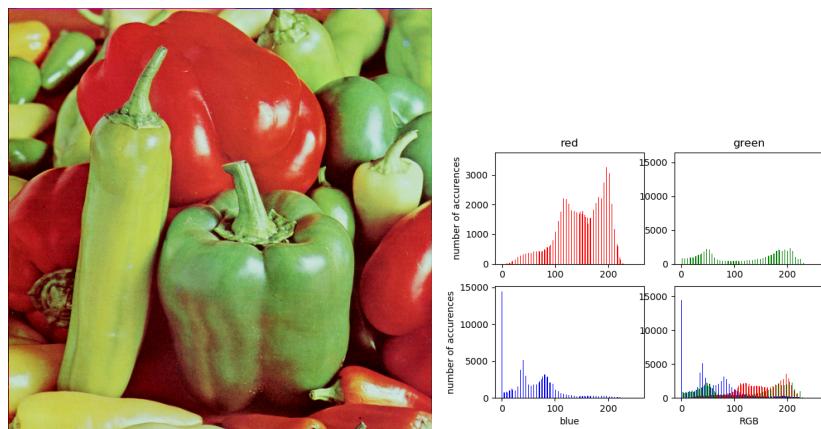
1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla( $P$ ):
3. Dla każdego kanału( $C$ ):
4. Oblicz wartość progową  $T_C$  jako średnią wartość piksli  $P_C$  w otoczeniu danego piksla( $P$ ).
5. Jeśli wartość piksla  $P_C$  jest  $< T_C$ :
6. Przypisz mu wartość 0.
7. W przeciwnym przypadku przypisz mu wartość 255.



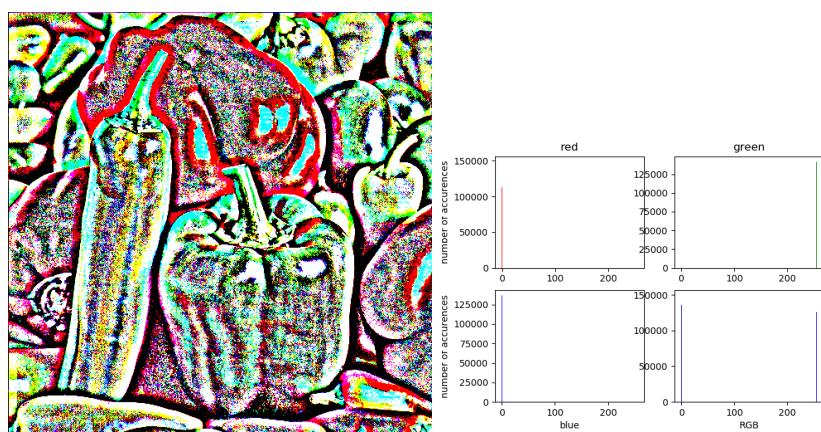
Rysunek 7.11: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 7.12: Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu



Rysunek 7.13: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 7.14: Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu

Listing 7.4: Progowanie 1-progowe lokalne

```

def localSingleThreshold ( self , dim = 3 , show = False , plot =
    False ) :
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc
    low , up = -(int(dim / 2)) , (int(dim / 2) + 1)   # wsp.
        sasiadow

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty((height , width , 3) , dtype=np .uint8)
    tmp = np .empty((height , width , 3))
    tmp2 = np .empty((height , width , 3))

    # progowanie lokalne
    for i in range (height) :
        for j in range (width) :
            n = 0
            r = 0
            g = 0
            b = 0
            currPix = self .im [i , j]
            for iOff in range (low , up) :
                for jOff in range (low , up) :
                    iSafe = i if ((i + iOff) > (height + low)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width + low)) | ((j +
                        jOff) < 0) else (j + jOff)
                    r += int(self .im [iSafe , jSafe ][0])
                    g += int(self .im [iSafe , jSafe ][1])
                    b += int(self .im [iSafe , jSafe ][2])
                    n += 1
                    r = int(round(r / n))
                    g = int(round(g / n))
                    b = int(round(b / n))
                    resultImage [i , j] = (0 if (currPix [0] < r) else 255 , 0 if
                        (currPix [1] < g) else 255 , 0 if (currPix [2] < b)
                        else 255)

    if show :
        self .show (Image .fromarray (resultImage , "RGB"))
        self .calculate (plot , resultImage)
        self .save (resultImage , self .imName , "localSingleThreshold")

```

## 7.5 Progowanie wielo-progowe lokalne

### Opis algorytmu

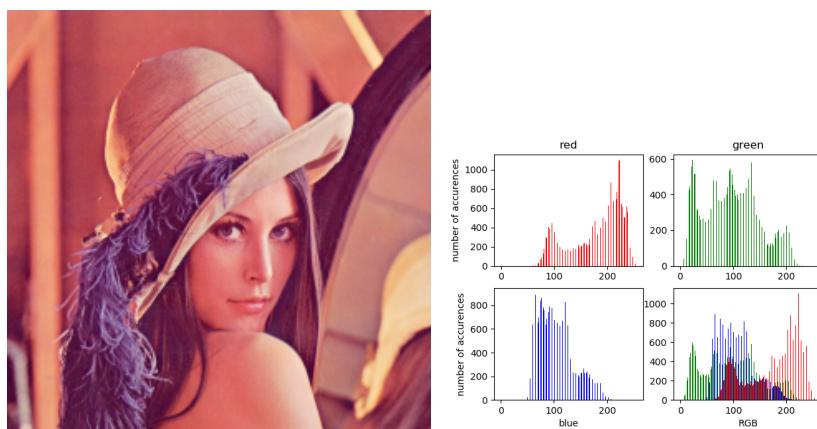
Progowanie wielo-progowe lokalne oblicza wartości progowe dla każdego piksła z osobna. W wyniku takiego progowania obraz ma mniejszą ilość kolorów w obrazie. Progi obliczane są dla każdego kanału z osobna.

1. Zdefiniuj wielkość otoczenia piksła (musi być nieparzysta, po to aby mógł istnieć piksel środkowy).
2. Zdefiniuj ilość progów( $T$ ).
3. Dla każdego piksła( $P$ ):
4. Dla każdego kanału( $C$ ):
5. Znajdź  $MAX(P_C)$  i  $MIN(P_C)$ .
6. Oblicz skalę( $S_C$ ) wg. wzoru:

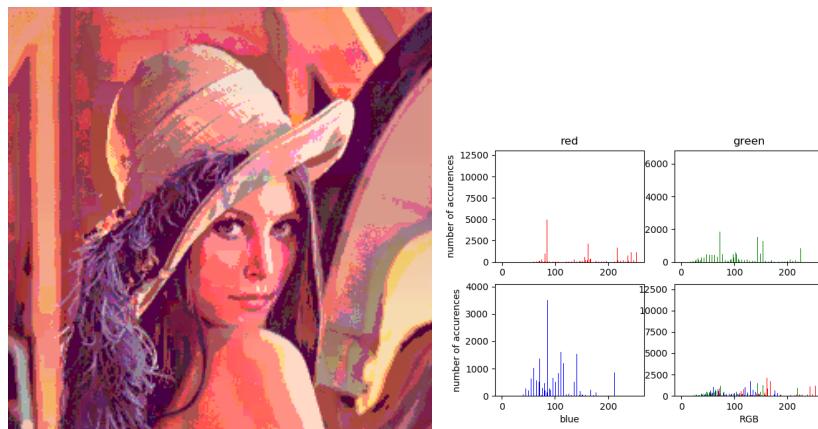
$$S_C = \frac{MAX(P_C)}{(T-1)}.$$

7. Jeśli  $S_C = 0$ :
8. Przypisz  $S_C$  wartość 1 (aby uniknąć dzielenia przez 0).
9. Wylicz nową wartość piksła( $P_{C_n}$ ) wg. wzoru:

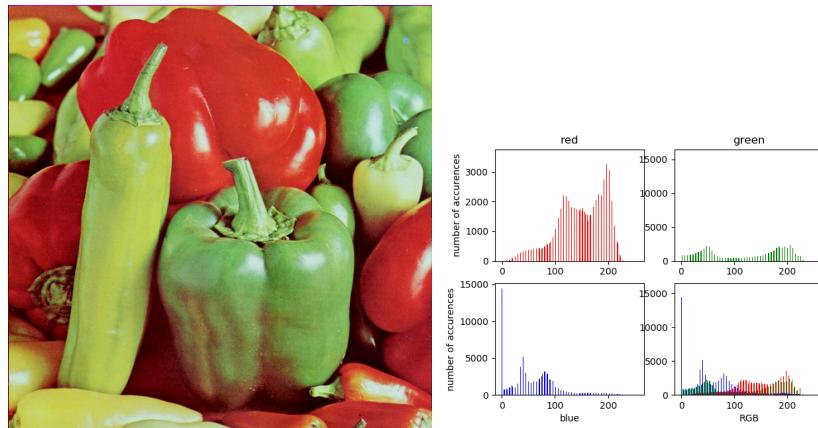
$$P_{C_n} = \lceil \frac{P_C}{S_C} \rceil * S_C.$$



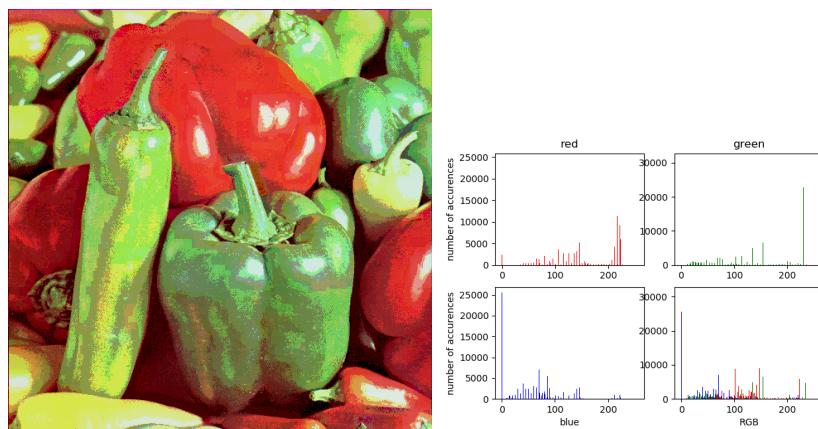
Rysunek 7.15: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.16: Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu



Rysunek 7.17: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.18: Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu

Listing 7.5: Progowanie wielo-progowe lokalne

```

def localMultiThreshold(self, dim=3, bins=4, show=False, plot=False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc
    low, up = -(int(dim / 2)), (int(dim / 2) + 1) # wsp.
        sasiadow

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width, 3), dtype=np.uint8)

    # progowanie lokalne
    for i in range(height):
        for j in range(width):
            n = 0
            r = 0
            g = 0
            b = 0
            currPix = self.im[i, j]
            maxValue = [0] * 3
            minValue = [255] * 3
            for iOff in range(low, up):
                for jOff in range(low, up):
                    iSafe = i if ((i + iOff) > (height + low)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width + low)) | ((j + jOff) < 0) else (j + jOff)
                    currValue = self.im[iSafe, jSafe]
                    for k in range(3):
                        maxValue[k] = max(maxValue[k], currValue[k])
                        minValue[k] = min(minValue[k], currValue[k])
            scale = [0] * 3
            for k in range(3):
                scale[k] = maxValue[k] / (bins - 1)
                if scale[k] == 0:
                    scale[k] = 1
            for k in range(3):
                v = int(round(currPix[k] / scale[k])) * scale[k]
                currPix[k] = v
            resultImage[i, j] = currPix

    if show:
        self.show(Image.fromarray(resultImage, "RGB"))

```

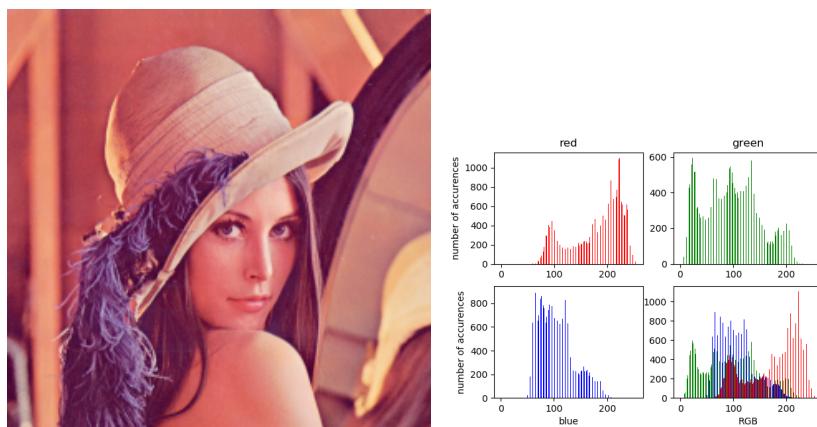
```
self.calculate(plot, resultImage)
self.save(resultImage, self.imName, "localMultiThreshold")
```

## 7.6 Progowanie 1-progowe globalne

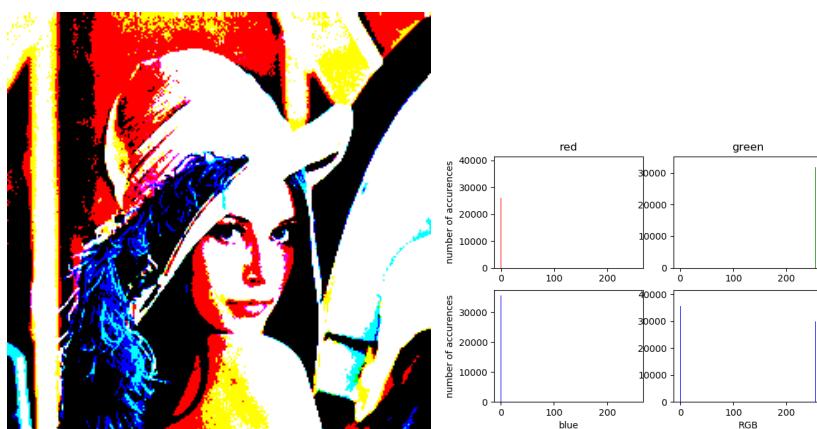
### Opis algorytmu

W progowaniu 1-progowym globalnym wartość progowa jest ustalana globalnie dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksela w obrazie. Następnie, tak wyliczona wartość progowa, jest stosowana do nadania każdemu pikselowi nową wartość.

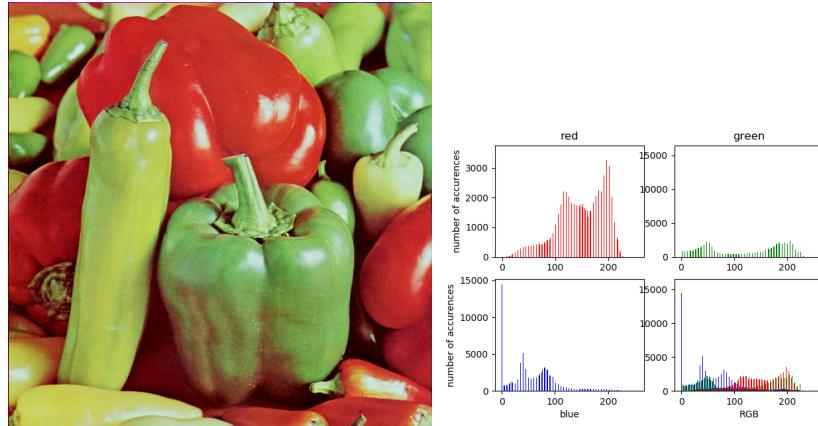
1. Dla każdego kanału( $C$ ):
2. Oblicz wartość progową  $T_C$ , jako średnią wartość z wszystkich pikseli  $P_C$  w obrazie.
3. Dla każdego piksela( $P$ ):
4. Dla każdego kanału( $C$ ):
5. Jeśli wartość  $P_C$  danego piksela jest  $< T_C$ :
6. przypisz mu wartość 0.
7. W przeciwnym przypadku przypisz mu wartość 255.



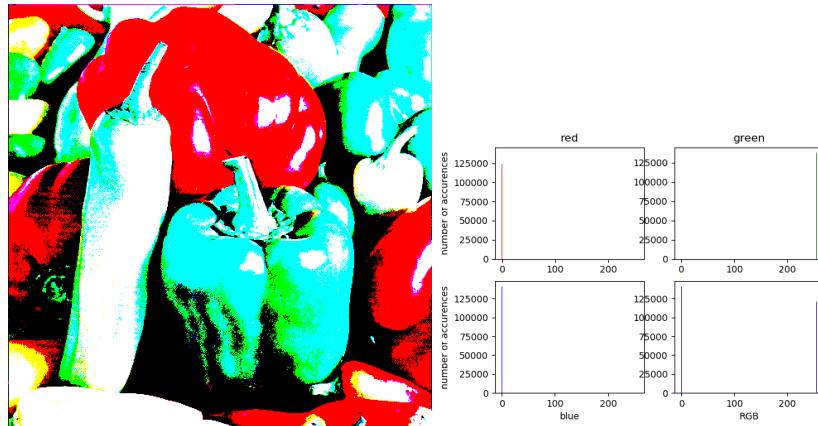
Rysunek 7.19: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.20: Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu



Rysunek 7.21: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.22: Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu

Listing 7.6: Progowanie 1-progowe globalne

```

def globalSingleThreshold(self , show = False , plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)
    tmp = np.empty((height , width , 3))
    tmp2 = np.empty((height , width , 3))

    # prog globalny
    globalR = 0
    globalG = 0
    globalB = 0
    nR = 0

```

```

nG = 0
nB = 0
for i in range(height):
    for j in range(width):
        globalR += self .im[ i , j ][ 0 ]
        nR += 1
        globalG += self .im[ i , j ][ 1 ]
        nG += 1
        globalB += self .im[ i , j ][ 2 ]
        nB += 1
    globalR = int(round(globalR / nR))
    globalG = int(round(globalG / nG))
    globalB = int(round(globalB / nB))

# kwantzyacja
for i in range(height):
    for j in range(width):
        resultImage[ i , j ] = (0 if ( self .im[ i , j ][ 0 ] < globalR )
                                else 255, 0 if ( self .im[ i , j ][ 1 ] < globalG ) else 255,
                                0 if ( self .im[ i , j ][ 2 ] < globalB ) else 255)

if show:
    self .show( Image .fromarray( resultImage , "RGB" ) )
    self .calculate( plot , resultImage )
    self .save( resultImage , self .imName , "globalSingleThreshold" )

```

## 7.7 Progowanie wielo-progowe globalne

### Opis algorytmu

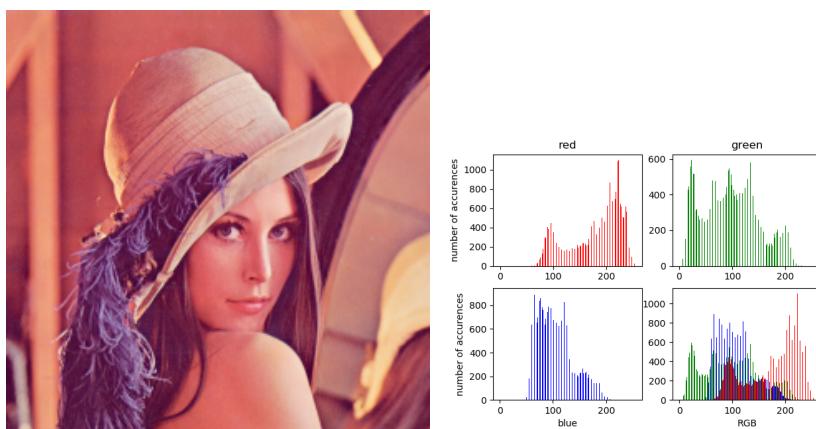
W progowaniu wielo-progowym globalnym wartości progowe są ustalane dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa, jest stosowana do nadania każdemu pikselowi nowej wartości.

1. Zdefiniuj ilość progów( $T$ ).
2. Dla każdego piksla( $P$ ):
3. Dla każdego kanału( $C$ ):
4. Znajdź  $\text{MAX}(P_C)$  i  $\text{MIN}(P_C)$ .
5. Oblicz skalę( $S_C$ ) wg. wzoru:

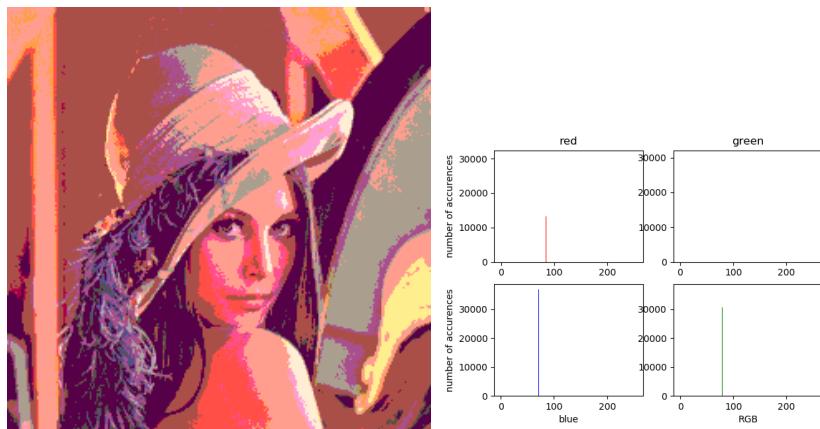
$$S_C = \frac{\text{MAX}(P_C)}{(T-1)}.$$

6. Dla każdego piksla( $P$ ):
7. Dla każdego kanału( $C$ ):
8. Wylicz nową wartość piksla( $P_{C_n}$ ) wg. wzoru:

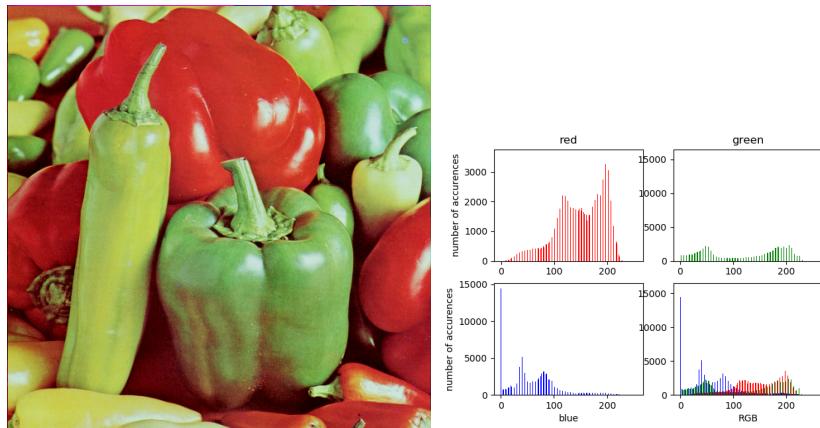
$$P_{C_n} = \lceil \frac{P_C}{S_C} \rceil * S_C.$$



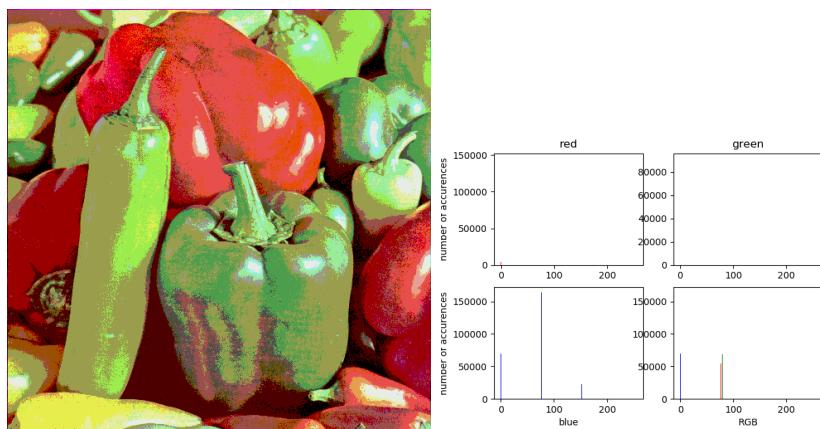
Rysunek 7.23: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.24: Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu



Rysunek 7.25: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.26: Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu

Listing 7.7: Progowanie wielo-progowe globalne

```

def globalMultiThreshold(self , bins = 4, show = False , plot =
False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)

    maxValue = [0] * 3
    minValue = [255] * 3
# wartosci max i min w obrazie
    for i in range(height):
        for j in range(width):
            currValue = self.im[i , j]
            for k in range(3):
                maxValue[k] = max(maxValue[k] , currValue[k])
                minValue[k] = min(minValue[k] , currValue[k])

    scale = [0] * 3
    for k in range(3):
        scale[k] = maxValue[k] / (bins - 1)

    for i in range(height):
        for j in range(width):
            pix = self.im[i , j]
            for k in range(3):
                pix[k] = int(round(pix[k] / scale[k])) * scale[k]
            resultImage[i , j] = pix

    if show:
        self.show(Image.fromarray(resultImage , "RGB"))
        self.calculate(plot , resultImage)
        self.save(resultImage , self.imName, "globalMultiThreshold")

```

## **Rozdział 8**

# **Operacje morfologiczne na obrazach binarnych**

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

## Rozdział 9

# Operacje morfologiczne na obrazach szarych

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

## Rozdział 10

# Filtrowanie liniowe i nieliniowe

Filtrowanie obrazów oznacza, że do obliczenia nowej wartości danego piksela brane są pod uwagę wartości pikseli z jego otoczenia. Każdy punkt z otoczenia wnosi swój wkład (wagę) podczas przeprowadzania końcowych obliczeń. Wagi te są zapisywane w postaci maski. Najczęściej spotykane rozmiary filtrów to maski o rozmiarach 3x3, 5x5. Ale filtry o rozmiarach 7x7, 9x9 również są nierzadko używane. Filtrowanie odbywa się w ten sposób, że maskę przemieszcza się w obrębie obrazu z krokiem równym odległości międzypikselowej. Poniżej przykład dla maski o wymiarach 3x3:

$$\begin{matrix} f_{-1,-1} & f_{0,-1} & f_{1,-1} \\ f_{-1,0} & f_{0,0} & f_{1,0} \\ f_{-1,1} & f_{0,1} & f_{1,1} \end{matrix}$$

By obliczyć nową wartość piksła należy wpierw obliczyć sumę ważoną składowych punktu oraz wszystkich sąsiadujących punktów zgodnie z wagami wskazanymi przez maskę.

$$s = f_{-1,-1} * a_{i-1,j-1} + f_{0,-1} * a_{i,j-1} + f_{1,-1} * a_{i+1,j-1} + f_{-1,0} * a_{i-1,j} + f_{0,0} * a_{i,j} + f_{1,0} * a_{i+1,j} + f_{-1,1} * a_{i-1,j+1} + f_{0,1} * a_{i,j+1} + f_{1,1} * a_{i+1,j+1}$$

Otrzymaną sumę dzielimy przez sumę wszystkich wag maski, jeżeli jest ona różna od 0.

$$a'_{i,j} = \frac{s}{f_{-1,-1} + f_{0,-1} + f_{1,-1} + f_{-1,0} + f_{0,0} + f_{1,0} + f_{-1,1} + f_{0,1} + f_{1,1}}$$

## 10.1 Filtr dolnoprzepustowy uśredniający

### Opis algorytmu

Filtr uśredniający jest podstawowym filtrem dolnoprzepustowym, jego wynikiem jest uśrednienie każdego piksla razem ze swoimi sąsiadami. Maska:

	1	1	1
$\frac{1}{9}$	1	1	1
	1	1	1

1. Dla każdego piksla( $P$ ):
2. Dla każdej barwy:
3. Zsumuj wartości barwy piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Sumę barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikslowi  $P$ .



Rysunek 10.1: Obraz wejściowy, obraz uśredniony



Rysunek 10.2: Obraz wejściowy, obraz uśredniony



Rysunek 10.3: Obraz wejściowy, obraz uśredniony



Rysunek 10.4: Obraz wejściowy, obraz uśredniony

Listing 10.1: Filtr dolnoprzepustowy uśredniający (obraz szary)

```

def averageGray( self , show = False ):
    width = self .im .shape [1]      # szereokosc
    height = self .im .shape [0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np .empty(( height , width ) , dtype=np .uint8)

mask = np .ones ((3 , 3))

# wygladzanie
for i in range (height):
    for j in range (width):
        avg = 0
        n = 0
        for iOff in range (-1, 1):
            for jOff in range (-1, 1):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                avg += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff +
                    1]
                n += mask [iOff + 1 , jOff + 1]
        avg = int (round (avg / n))
        resultImage [i , j] = avg

    if show:
        self .show (Image .fromarray (resultImage , "L"))
    self .save (resultImage , self .imName , "lowpassAvg")

```

Listing 10.2: Filtr dolnoprzepustowy uśredniający (obraz barwny)

```

def averageColor( self , show = False ) :
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8 )

    mask = np .ones((3 , 3))

    # wygladzanie
    for i in range( height ) :
        for j in range( width ) :
            avgr = 0
            avgg = 0
            avgb = 0
            n = 0
            for iOff in range(-1, 1) :
                for jOff in range(-1, 1) :
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    avgr += self .im[ iSafe , jSafe ][0] * mask[ iOff + 1 ,
                        jOff + 1]
                    avgg += self .im[ iSafe , jSafe ][1] * mask[ iOff + 1 ,
                        jOff + 1]
                    avgb += self .im[ iSafe , jSafe ][2] * mask[ iOff + 1 ,
                        jOff + 1]
                    n += mask[ iOff + 1 , jOff + 1]
            avgr = int(round(avgr / n))
            avgg = int(round(avgg / n))
            avgb = int(round(avgb / n))
            resultImage [i , j] = (avgr , avgg , avgb)

    if show :
        self .show( Image .fromarray( resultImage , "RGB" ))
        self .save( resultImage , self .imName , "lowpassAvg" )

```

## 10.2 Filtr dolnoprzepustowy Gaussowski

### Opis algorytmu

Filtr Gaussa jest filtrem uśredniającym. Jego maska aproksymuje 2-wymiarową krzywą Gaussa. W odrużnieniu od filtru uśredniającego efekt rozmycia przez ten filtr jest mniejszy. Maska:

	1	1	1	1	1
	1	4	6	4	1
$\frac{1}{47}$	1	1	1	1	1
	1	4	6	4	1
	1	1	1	1	1

1. Dla każdego piksla( $P$ ):
2. Dla każdej barwy:
3. Zsumuj wartości barwy piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Sumę wartości barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikslowi  $P$ .



Rysunek 10.5: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.6: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.7: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.8: Obraz wejściowy, obraz po filtracji filtrem Gaussa

Listing 10.3: Filtr dolnoprzepustowy Gaussowski (obraz szary)

```

def gaussGray( self , show = False):
    width = self .im .shape [1]  # szereokosc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    mask = np .ones ((5 , 5))
    mask [1 , 1] = mask [3 , 3] = mask [1 , 3] = mask [3 , 1] = 4
    mask [1 , 2] = mask [3 , 2] = 6

    # filtracja
    for i in range (height):
        for j in range (width):
            n = 0
            value = 0
            for iOff in range (-2, 3):
                for jOff in range (-2, 3):
                    iSafe = i if ((i + iOff) > (height - 2)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 2 , jOff
                        + 2]
                    n += mask [iOff + 2 , jOff + 2]
            value = int(round(value / n))
            resultImage [i , j] = value

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "lowpassGauss")

```

Listing 10.4: Filtr dolnoprzepustowy Gaussowski (obraz barwny)

```

def gaussColor( self , show = False ):
    width = self .im .shape [1]      # szereokosc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    mask = np .ones((5 , 5))
    mask [1 , 1] = mask [3 , 3] = mask [1 , 3] = mask [3 , 1] = 4
    mask [1 , 2] = mask [3 , 2] = 6

    # filtracja
    for i in range (height):
        for j in range (width):
            n = 0
            r , g , b = 0 , 0 , 0
            for iOff in range (-2, 3):
                for jOff in range (-2, 3):
                    iSafe = i if ((i + iOff) > (height - 2)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ][0] * mask [iOff + 2 , jOff +
                        2]
                    g += self .im [iSafe , jSafe ][1] * mask [iOff + 2 , jOff +
                        2]
                    b += self .im [iSafe , jSafe ][2] * mask [iOff + 2 , jOff +
                        2]
                    n += mask [iOff + 2 , jOff + 2]
            r = int (round (r / n))
            g = int (round (g / n))
            b = int (round (b / n))
            resultImage [i , j] = (r , g , b)

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "lowpassGauss")

```

## 10.3 Operator Roberts'a

### Opis algorytmu

Filtr Roberts'a jest jednym z najbardziej znanych filtrów do wykrywania krawędzi w obrazie. Wynikowa wartość składowej po zastosowaniu owego filtra może wyjść ujemna, aby temu zapobiec należy użyć wartości bezwzględnej. Filtr Roberts'a jest bardzo wrażliwy na szum i ma niski poziom reakcji na krawędź obrazu. Maska:

0	0	0
0	0	-1
0	1	0

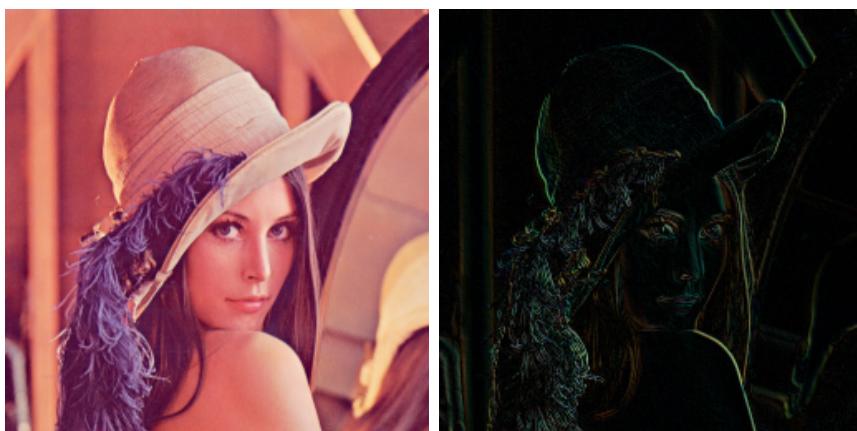
1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Zsumuj wartości pikseli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Przypisz nową wartość barwy pikselowi  $P$ .



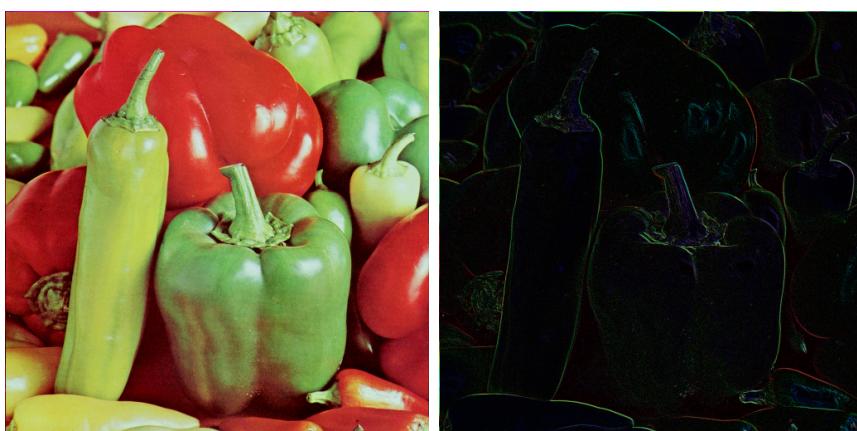
Rysunek 10.9: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.10: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.11: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.12: Obraz wejściowy, obraz po filtracji filtrem Roberts'a

Listing 10.5: Operator Roberts'a (obraz szary)

```

def robertsGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [2 , 1] = 1
    mask [1 , 2] = -1

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff
                        + 1]
            resultImage [i , j] = abs (value)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "highpassRoberts")

```

Listing 10.6: Operator Roberts'a (obraz barwny)

```

def robertsColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    mask = np .zeros ((3 , 3))
    mask [2 , 1] = 1
    mask [1 , 2] = -1

    # filtracja
    for i in range (height):
        for j in range (width):
            r , g , b = 0 , 0 , 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ][0] * mask [iOff + 1 , jOff +
                        1]
                    g += self .im [iSafe , jSafe ][1] * mask [iOff + 1 , jOff +
                        1]
                    b += self .im [iSafe , jSafe ][2] * mask [iOff + 1 , jOff +
                        1]
            resultImage [i , j] = (abs (r) , abs (g) , abs (b))

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "highpassRoberts")

```

## 10.4 Operator Prewitt'a

### Opis algorytmu

Filtr Prewitt'a, podobnie jak filtr Roberts'a, służy do wykrywania krawędzi i może w wyniku wygenerować wartość ujemną, aby temu zapobiec należy użyć wartości bezwzględnej. Maska Prewitt'a jest rozszerzeniem maski Roberts'a i nie jest tak wrażliwa na szum. Maska:

	-1	0	1
$\frac{1}{2}$	-1	0	1
	-1	0	1

1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Zsumuj wartości piksli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 2.
6. Jeśli otrzymany wynik jest  $> 255$ .
7. Przypisz mu wartość 255.
8. Przypisz nową wartość barwy pikslowi  $P$ .



Rysunek 10.13: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.14: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.15: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.16: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a

Listing 10.7: Operator Prewitt'a (obraz szary)

```

def prewittGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [1 , 0] = mask [2 , 0] = -1
    mask [0 , 2] = mask [1 , 2] = mask [2 , 2] = 1

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff
                        + 1]
            resultImage [i , j] = abs (value / 2)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "highpassPrewitt")

```

Listing 10.8: Operator Prewitt'a (obraz barwny)

```

def prewittColor( self , show = False ):
    width = self .im .shape [1]  # szereokosc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [1 , 0] = mask [2 , 0] = -1
    mask [0 , 2] = mask [1 , 2] = mask [2 , 2] = 1

    # filtracja
    for i in range (height):
        for j in range (width):
            r , g , b = 0 , 0 , 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ][0] * mask [iOff + 1 , jOff +
                        1]
                    g += self .im [iSafe , jSafe ][1] * mask [iOff + 1 , jOff +
                        1]
                    b += self .im [iSafe , jSafe ][2] * mask [iOff + 1 , jOff +
                        1]
            r , g , b = (abs (r)/2 , abs (g)/2 , abs (b)/2)
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage [i , j] = (r , g , b)

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "highpassPrewitt")

```

## 10.5 Operator Sobel'a

### Opis algorytmu

Filtr Prewitt'a, podobnie jak filtr Roberts'a, służy do wykrywania krawędzi i może w wyniku wygenerować wartość ujemną, aby temu zapobiec należy użyć wartości bezwzględnej. Maska Prewitt'a jest rozszerzeniem maski Roberts'a i nie jest tak wrażliwa na szum. Maska:

$\frac{1}{4}$	1	2	1
	0	0	0
	-1	-2	-1

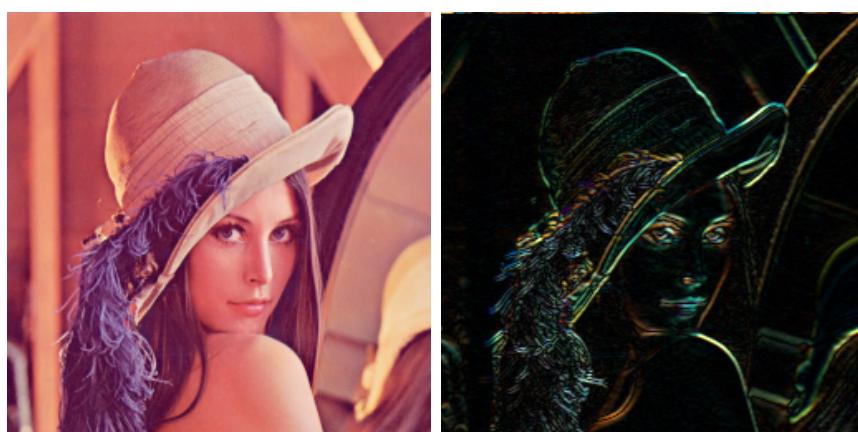
1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Zsumuj wartości piksli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 4.
6. Jeśli otrzymana wartość jest  $> 255$ .
7. Przypisz jej wartość 255.
8. Przypisz nową wartość barwy pikslowi  $P$ .



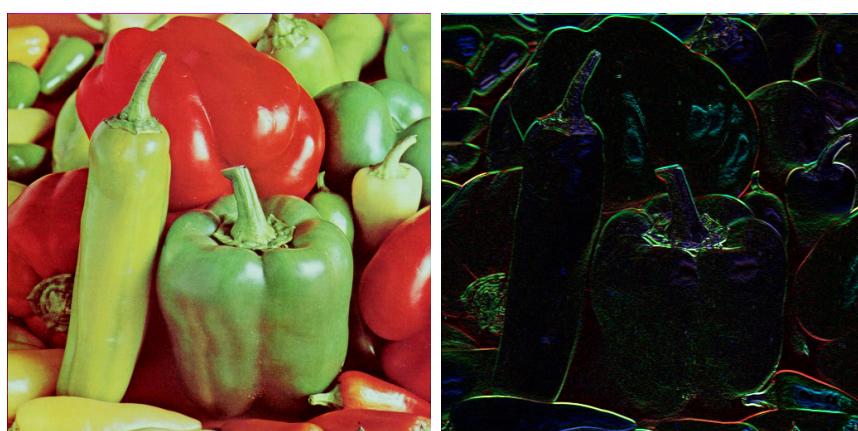
Rysunek 10.17: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.18: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.19: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.20: Obraz wejściowy, obraz po filtracji filtrem Sobel'a

Listing 10.9: Operator Sobel'a (obraz szary)

```

def sobolGray( self , show = False):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [0 , 2] = 1
    mask [2 , 0] = mask [2 , 2] = -1
    mask [0 , 1] = 2
    mask [2 , 1] = -2

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff
                        + 1]
            resultImage [i , j] = abs (value / 4)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "highpassSobel")

```

Listing 10.10: Operator Sobel'a (obraz barwny)

```

def sobolColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [0 , 2] = 1
    mask [2 , 0] = mask [2 , 2] = -1
    mask [0 , 1] = 2
    mask [2 , 1] = -2

    # filtracja
    for i in range (height):
        for j in range (width):
            r , g , b = 0 , 0 , 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ][0] * mask [iOff + 1 , jOff +
                        1]
                    g += self .im [iSafe , jSafe ][1] * mask [iOff + 1 , jOff +
                        1]
                    b += self .im [iSafe , jSafe ][2] * mask [iOff + 1 , jOff +
                        1]
            r = abs(r) / 2
            g = abs(g) / 2
            b = abs(b) / 2
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage [i , j] = (r , g , b)

```

```
if show:  
    self.show( Image.fromarray( resultImage , "RGB" ) )  
    self.save( resultImage , self.imName , "highpassSobel" )
```

## 10.6 Filtr kompasowy

### Opis algorytmu

Filtr kompasowy polega na splecieniu zbioru 8 masek wzornikowych, gdzie każda z nich jest czuła w innym kierunku. Dla każdego piksla wybierana jest maska o maksymalnej reakcji. Maski Sobel'a:

$\frac{1}{4}$	-1 0 1 -2 0 2 -1 0 1
$\frac{1}{4}$	0 1 2 -1 0 1 -2 -1 0
$\frac{1}{4}$	1 2 1 0 0 0 -1 -2 -1
$\frac{1}{4}$	2 1 0 1 0 -1 0 -1 -2
$\frac{1}{4}$	1 0 -1 2 0 -2 1 0 -1
$\frac{1}{4}$	0 -1 -2 1 0 -1 2 1 0
$\frac{1}{4}$	-1 -2 -1 0 0 0 1 2 1
$\frac{1}{4}$	-2 -1 0 -1 0 1 0 1 2

1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Dla każdej z masek( $M$ ):
4. Zsumuj wartości piksli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski  $M$ .
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Następnie podziel ją przez 4.

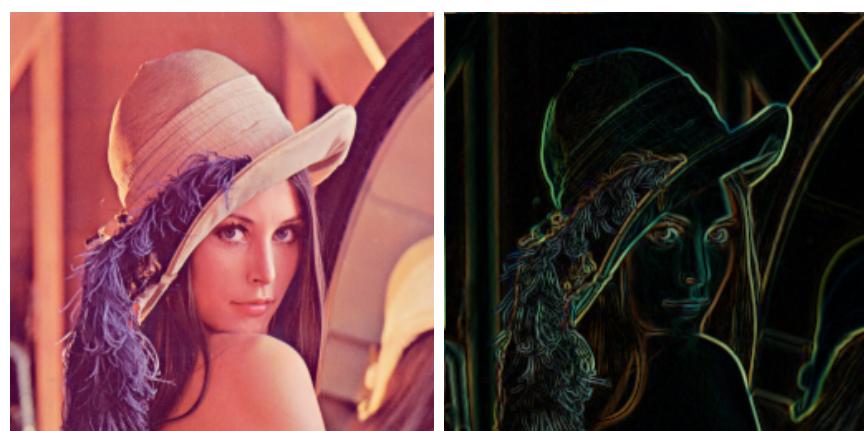
7. Wybierz największą z wartości, która jest maksymalną reakcją gradientu.
8. Przypisz nową wartość barwy pikselowi  $P$ .



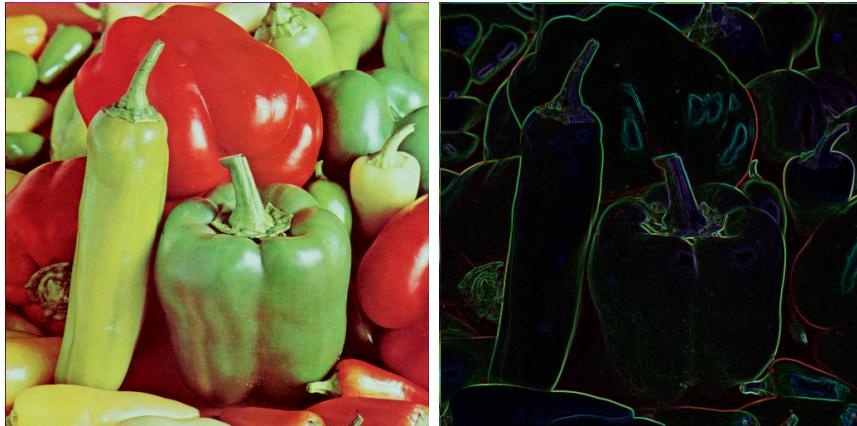
Rysunek 10.21: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.22: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.23: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.24: Obraz wejściowy, obraz po filtracji filtrem kompasowym

Listing 10.11: Filtr kompasowy (obraz szary)

```

def compassGray(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width) , dtype=np.uint8)

#maska
mask = [0] * 8

mask[0] = np.zeros((3 , 3))
mask[0][0 , 2] = mask[0][2 , 2] = 1
mask[0][0 , 0] = mask[0][2 , 0] = -1
mask[0][1 , 2] = 2
mask[0][1 , 0] = -2

mask[1] = np.zeros((3 , 3))
mask[1][0 , 1] = mask[1][1 , 2] = 1
mask[1][1 , 0] = mask[1][2 , 1] = -1
mask[1][0 , 2] = 2
mask[1][2 , 0] = -2

mask[2] = np.zeros((3 , 3))
mask[2][0 , 0] = mask[2][0 , 2] = 1
mask[2][2 , 0] = mask[2][2 , 2] = -1
mask[2][0 , 1] = 2
mask[2][2 , 1] = -2

```

```

mask[3] = np.zeros((3, 3))
mask[3][0, 1] = mask[3][1, 0] = 1
mask[3][1, 2] = mask[3][2, 1] = -1
mask[3][0, 0] = 2
mask[3][2, 2] = -2

mask[4] = np.zeros((3, 3))
mask[4][0, 0] = mask[4][2, 0] = 1
mask[4][0, 2] = mask[4][2, 2] = -1
mask[4][1, 0] = 2
mask[4][1, 2] = -2

mask[5] = np.zeros((3, 3))
mask[5][1, 0] = mask[5][2, 1] = 1
mask[5][0, 1] = mask[5][1, 2] = -1
mask[5][2, 0] = 2
mask[5][0, 2] = -2

mask[6] = np.zeros((3, 3))
mask[6][2, 0] = mask[6][2, 2] = 1
mask[6][0, 0] = mask[6][0, 2] = -1
mask[6][2, 1] = 2
mask[6][0, 1] = -2

mask[7] = np.zeros((3, 3))
mask[7][1, 2] = mask[7][2, 1] = 1
mask[7][0, 1] = mask[7][1, 0] = -1
mask[7][2, 2] = 2
mask[7][0, 0] = -2

# filtracja
for i in range(height):
    for j in range(width):
        value = [0] * 8
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                for k in range(8):

```

```
    value[k] += self.im[iSafe, jSafe] * mask[k][iOff +
        1, jOff + 1]

    resultImage[i, j] = max(map(abs, value)) / 4

if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "compassSobol")
```

Listing 10.12: Filtr kompasowy (obraz barwny)

```

def compassColor(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)
    tmp = np.empty((height , width , 3))
    tmp2 = np.empty((height , width , 3))

#maska
    mask = [0] * 8

    mask[0] = np.zeros((3 , 3))
    mask[0][0 , 2] = mask[0][2 , 2] = 1
    mask[0][0 , 0] = mask[0][2 , 0] = -1
    mask[0][1 , 2] = 2
    mask[0][1 , 0] = -2

    mask[1] = np.zeros((3 , 3))
    mask[1][0 , 1] = mask[1][1 , 2] = 1
    mask[1][1 , 0] = mask[1][2 , 1] = -1
    mask[1][0 , 2] = 2
    mask[1][2 , 0] = -2

    mask[2] = np.zeros((3 , 3))
    mask[2][0 , 0] = mask[2][0 , 2] = 1
    mask[2][2 , 0] = mask[2][2 , 2] = -1
    mask[2][0 , 1] = 2
    mask[2][2 , 1] = -2

    mask[3] = np.zeros((3 , 3))
    mask[3][0 , 1] = mask[3][1 , 0] = 1
    mask[3][1 , 2] = mask[3][2 , 1] = -1
    mask[3][0 , 0] = 2
    mask[3][2 , 2] = -2

    mask[4] = np.zeros((3 , 3))
    mask[4][0 , 0] = mask[4][2 , 0] = 1
    mask[4][0 , 2] = mask[4][2 , 2] = -1
    mask[4][1 , 0] = 2
    mask[4][1 , 2] = -2

```

```

mask[5] = np.zeros((3, 3))
mask[5][1, 0] = mask[5][2, 1] = 1
mask[5][0, 1] = mask[5][1, 2] = -1
mask[5][2, 0] = 2
mask[5][0, 2] = -2

mask[6] = np.zeros((3, 3))
mask[6][2, 0] = mask[6][2, 2] = 1
mask[6][0, 0] = mask[6][0, 2] = -1
mask[6][2, 1] = 2
mask[6][0, 1] = -2

mask[7] = np.zeros((3, 3))
mask[7][1, 2] = mask[7][2, 1] = 1
mask[7][0, 1] = mask[7][1, 0] = -1
mask[7][2, 2] = 2
mask[7][0, 0] = -2

# filtracja
for i in range(height):
    for j in range(width):
        r = [0] * 8
        g = [0] * 8
        b = [0] * 8
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                for k in range(8):
                    r[k] += self.im[iSafe, jSafe][0] * mask[k][iOff + 1, jOff + 1]
                    g[k] += self.im[iSafe, jSafe][1] * mask[k][iOff + 1, jOff + 1]
                    b[k] += self.im[iSafe, jSafe][2] * mask[k][iOff + 1, jOff + 1]

        resultImage[i, j] = (max(map(abs, r)) / 4, max(map(abs, g)) / 4, max(map(abs, b)) / 4)

if show:

```

```
self.show(Image.fromarray(resultImage, "RGB"))
self.save(resultImage, self.imName, "compassSobol")
```

## 10.7 Gradient wektora kierunkowego

### Opis algorytmu

Filtr ten służy do wykrywania krawędzi w obrazie. Podobnie jak dla filtru kompasowego, filtr wektora kierunkowego polega na splecieniu 4 masek wzornikowych, gdzie każda z nich jest czuła w innym kierunku, a dla każdego piksla wybierana jest maska o maksymalnej reakcji. Maski Prewitt'a:

$\frac{1}{2}$	-1	0	1
	-1	0	1
	-1	0	1

$\frac{1}{2}$	1	1	1
	0	0	0
	-1	-1	-1

$\frac{1}{2}$	1	0	-1
	1	0	-1
	1	0	-1

$\frac{1}{2}$	-1	-1	-1
	0	0	0
	1	1	1

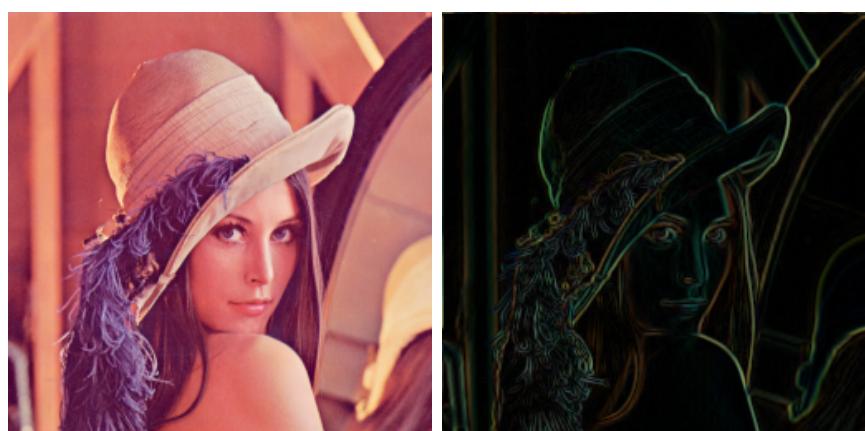
1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Dla każdej z masek( $M$ ):
4. Zsumuj wartości piksli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski  $M$ .
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Następnie podziel ją przez 4.
7. Wybierz największą z wartości, która jest maksymalną reakcją gradientu.
8. Przypisz nową wartość barwy pikslowi  $P$ .



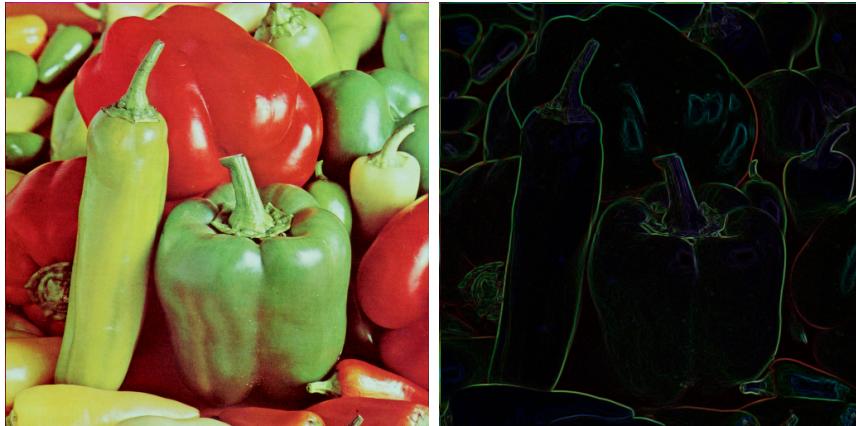
Rysunek 10.25: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.26: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.27: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.28: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego

Listing 10.13: Gradient wektora kierunkowego (obraz szary)

```

def VDGGray( self , show = False):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty (( height , width ) , dtype=np .uint8 )

    #maska
    mask = [0] * 4

    mask [0] = np .zeros ((3 , 3))
    mask [0][0 , 2] = mask [0][2 , 2] = 1
    mask [0][0 , 0] = mask [0][2 , 0] = -1
    mask [0][1 , 2] = 1
    mask [0][1 , 0] = -1

    mask [1] = np .zeros ((3 , 3))
    mask [1][0 , 0] = mask [1][0 , 2] = 1
    mask [1][2 , 0] = mask [1][2 , 2] = -1
    mask [1][0 , 1] = 1
    mask [1][2 , 1] = -1

    mask [2] = np .zeros ((3 , 3))
    mask [2][0 , 0] = mask [2][2 , 0] = 1
    mask [2][0 , 2] = mask [2][2 , 2] = -1
    mask [2][1 , 0] = 1
    mask [2][1 , 2] = -1

```

```

mask[3] = np.zeros((3, 3))
mask[3][2, 0] = mask[3][2, 2] = 1
mask[3][0, 0] = mask[3][0, 2] = -1
mask[3][2, 1] = 1
mask[3][0, 1] = -1

# filtracja
for i in range(height):
    for j in range(width):
        value = [0] * 4
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                for k in range(4):
                    value[k] += self.im[iSafe, jSafe] * mask[k][iOff + 1,
                        jOff + 1]

        resultImage[i, j] = max(map(abs, value)) / 2

if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "vdgSobol")

```

Listing 10.14: Gradient wektora kierunkowego (obraz barwny)

```

def VDGColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    #maska
    mask = [0] * 4

    mask [0] = np .zeros ((3 , 3))
    mask [0][0 , 2] = mask [0][2 , 2] = 1
    mask [0][0 , 0] = mask [0][2 , 0] = -1
    mask [0][1 , 2] = 1
    mask [0][1 , 0] = -1

    mask [1] = np .zeros ((3 , 3))
    mask [1][0 , 0] = mask [1][0 , 2] = 1
    mask [1][2 , 0] = mask [1][2 , 2] = -1
    mask [1][0 , 1] = 1
    mask [1][2 , 1] = -1

    mask [2] = np .zeros ((3 , 3))
    mask [2][0 , 0] = mask [2][2 , 0] = 1
    mask [2][0 , 2] = mask [2][2 , 2] = -1
    mask [2][1 , 0] = 1
    mask [2][1 , 2] = -1

    mask [3] = np .zeros ((3 , 3))
    mask [3][2 , 0] = mask [3][2 , 2] = 1
    mask [3][0 , 0] = mask [3][0 , 2] = -1
    mask [3][2 , 1] = 1
    mask [3][0 , 1] = -1

    # filtracja
    for i in range (height):
        for j in range (width):
            r = [0] * 4
            g = [0] * 4

```

```

b = [0] * 4
for iOff in range(-1, 2):
    for jOff in range(-1, 2):
        iSafe = i if ((i + iOff) > (height - 1)) else (i +
            iOff)
        jSafe = j if ((j + jOff) > (width - 1)) else (j +
            jOff)
        for k in range(4):
            r[k] += self.im[iSafe, jSafe][0] * mask[k][iOff +
                1, jOff + 1]
            g[k] += self.im[iSafe, jSafe][1] * mask[k][iOff +
                1, jOff + 1]
            b[k] += self.im[iSafe, jSafe][2] * mask[k][iOff +
                1, jOff + 1]

resultImage[i, j] = (max(map(abs, r)) / 4, max(map(abs, g
)) / 4, max(map(abs, b)) / 4)

if show:
    self.show(Image.fromarray(resultImage, "RGB"))
    self.save(resultImage, self.imName, "vdgSobol")

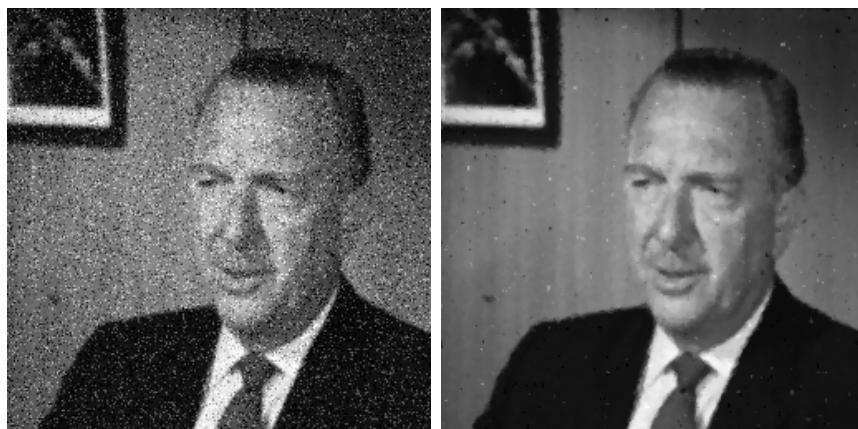
```

## 10.8 Filtr medianowy

### Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Filtr medianowy (środkowy) opiera się na medianie, czyli wartości środkowej spośród uporządkowanych wartości piksli z otoczenia badanego piksła. Filtr ten stosuje się do redukcji szumu w obrazie.

1. Dla każdego piksła ( $P$ ):
2. Dla każdej z barw ( $C$ ):
3. Umieść wartości barwy  $C$  piksli z otoczenia piksła  $P$  w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy  $C$ , a następnie wybierz medianę.
5. Przypisz znalezioną medianę jako nową wartość barwy piksła  $P$ .



Rysunek 10.29: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.30: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.31: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.32: Obraz wejściowy, obraz po filtracji filtrem medianowym

Listing 10.15: Filtr medianowy (obraz szary)

```

def medianGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            median = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    median[3*(1 + iOff) + jOff + 1] = self .im [ iSafe ,
                        jSafe ]
            median .sort ()
            u = int (round (len (median)/2))
            resultImage [i , j] = median [u] if ((u*2) % 2 == 0) else ((
                median [u - 1] + median [u])/2)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "median")

```

Listing 10.16: Filtr medianowy (obraz brawny)

```

def medianColor( self , show = False ) :
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    for i in range( height ):
        for j in range( width ):
            r = [0] * 9
            g = [0] * 9
            b = [0] * 9
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][0]
                    g[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][1]
                    b[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][2]
            r .sort()
            g .sort()
            b .sort()
            ur = int (round(len(r) / 2))
            ug = int (round(len(g) / 2))
            ub = int (round(len(b) / 2))
            resultImage [i , j] = (r [ur] if ((ur*2) % 2 == 0) else ((r [
                ur - 1] + r [ur]) /2) , g [ug] if ((ug*2) % 2 == 0) else
                ((g [ug - 1] + g [ug]) /2) , b [ub] if ((ub*2) % 2 == 0)
                else ((b [ub - 1] + b [ub]) /2))

    if show:
        self .show( Image .fromarray( resultImage , "RGB" ))
    self .save( resultImage , self .imName , "median" )

```

## 10.9 Filtr maksymalny

### Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Zwany jest także filtrem dekompresującym albo ekspansywnym. Jego działanie polega na wybraniu z pod maski punktu o wartości największej. Jego działanie powoduje zwiększenie jasności obrazu, daje to efekt powiększania się obiektów.

1. Dla każdego piksla ( $P$ ):
2. Dla każdej z barw ( $C$ ):
3. Umieść wartości barwy  $C$  piksli z otoczenia piksla  $P$  w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy  $C$ , a następnie wybierz ostatni(największy) element.
5. Przypisz znalezioną wartość jako nową wartość barwy piksla  $P$ .



Rysunek 10.33: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.34: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.35: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.36: Obraz wejściowy, obraz po filtracji filtrem maksymalnym

Listing 10.17: Filtr maksymalny (obraz szary)

```

def maxGray( self , show=False ) :
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            median = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    median[3 * (1 + iOff) + jOff + 1] = self .im [iSafe ,
                        jSafe]
            median .sort ()
            resultImage [i , j] = median [len (median) - 1]

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "max")

```

Listing 10.18: Filtr maksymalny (obraz barwny)

```

def maxColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)

    for i in range(height):
        for j in range(width):
            r = [0] * 9
            g = [0] * 9
            b = [0] * 9
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][0]
                    g[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][1]
                    b[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][2]
            r .sort()
            g .sort()
            b .sort()
            resultImage[i , j] = (r[len(r) - 1] , g[len(g) - 1] , b[len(b) - 1])

    if show:
        self .show(Image .fromarray(resultImage , "RGB"))
        self .save(resultImage , self .imName , "max")

```

## 10.10 Filtr minimalny

### Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Zwany jest także filtrem kompresującym albo erozyjnym. Jego działanie polega na wybraniu z pod maski punktu o wartości najmniejszej. Jego działanie powoduje zmniejszenie jasności obrazu, daje to efekt erozji obiektów.

1. Dla każdego piksla ( $P$ ):
2. Dla każdej z barw ( $C$ ):
3. Umieść wartości barwy  $C$  piksli z otoczenia piksla  $P$  w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy  $C$ , a następnie wybierz pierwszy(najmniejszy) element.
5. Przypisz znalezioną wartość jako nową wartość barwy piksla  $P$ .



Rysunek 10.37: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.38: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.39: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.40: Obraz wejściowy, obraz po filtracji filtrem minimalnym

Listing 10.19: Filtr minimalny (obraz szary)

```

def minGray( self , show=False ) :
    width = self .im .shape [1]  # szerokość
    height = self .im .shape [0]  # wysokość

    # alokacja pamięci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            median = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    median[3 * (1 + iOff) + jOff + 1] = self .im [iSafe ,
                        jSafe]
            median .sort ()
            resultImage [i , j] = median [0]

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "min")

```

Listing 10.20: Filtr minimalny (obraz barwny)

```

def minColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            r = [0] * 9
            g = [0] * 9
            b = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][0]
                    g[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][1]
                    b[3 * (1 + iOff) + jOff + 1] = self .im[iSafe , jSafe
                        ][2]
            r .sort ()
            g .sort ()
            b .sort ()
            resultImage [i , j] = (r [0] , g [0] , b [0])

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "min")

```

## 10.11 Filtr płaskorzeźbowy

### Opis algorytmu

Filtr płaskorzeźbowy swoją nazwę zdobył od efektu jaki generuje jego maska, obrazy przepuszczone przez ten filtr, w efekcie przypominają płaskorzeźbę. Sposób definiowania kierunków jest podobny jak w przypadku kierunkowych filtrów gradientowych. Różnicą jest jedynie wartość środkowa mająca wartość 1. W tym programie została użyta maska Prewitt'a. Maska:

$$\frac{1}{2} \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 1 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

1. Dla każdego piksla( $P$ ):
2. Dla każdego kanału( $C$ ):
3. Zsumuj wartości piksli  $P_C$ , otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 2.
6. Jeśli otrzymana wartość jest  $> 255$ .
7. Przypisz jej wartość 255.
8. Przypisz nową wartość barwy pikslowi  $P$ .



Rysunek 10.41: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.42: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.43: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.44: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym

Listing 10.21: Filtr płaskorzeźbowy (obraz szary)

```

def reliefGray( self , show = False):
    width = self .im .shape [1]  # szerokość
    height = self .im .shape [0]  # wysokość

    # alokacja pamięci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [1 , 0] = mask [2 , 0] = -1
    mask [0 , 2] = mask [1 , 2] = mask [2 , 2] = 1
    mask [1 , 1] = 1

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff
                        )
                    jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff +
                        1]
            value = abs (value) / 2
            if value > 255:
                value = 255
            resultImage [i , j] = value

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "reliefPrewitt")

```

Listing 10.22: Filtr płaskorzeźbowy (obraz barwny)

```

def reliefColor(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)
    tmp = np.empty((height , width , 3))
    tmp2 = np.empty((height , width , 3))

    mask = np.zeros((3 , 3))
    mask[0 , 0] = mask[1 , 0] = mask[2 , 0] = -1
    mask[0 , 2] = mask[1 , 2] = mask[2 , 2] = 1
    mask[1 , 1] = 1

    # filtracja
    for i in range(height):
        for j in range(width):
            r , g , b = 0 , 0 , 0
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self.im[iSafe , jSafe][0] * mask[iOff + 1 , jOff +
                        1]
                    g += self.im[iSafe , jSafe][1] * mask[iOff + 1 , jOff +
                        1]
                    b += self.im[iSafe , jSafe][2] * mask[iOff + 1 , jOff +
                        1]
            r = abs(r) / 2
            g = abs(g) / 2
            b = abs(b) / 2
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage[i , j] = (r , g , b)

```

```
if show:  
    self.show(Image.fromarray(resultImage, "RGB"))  
self.save(resultImage, self.imName, "reliefPrewitt")
```

## 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.