

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> . . . . .	3
<b>Rozdział 1. Wstęp</b> . . . . .	5
1.1 Specyfikacja wykorzystanego fortformatu obrazu . . . . .	5
1.2 Instrukcja obsługi programu . . . . .	5
<b>Rozdział 2. Operacje ujednoliciania obrazów</b> . . . . .	6
<b>Rozdział 3. Operacje sumowania arytmetycznego obrazów szarych</b> . . . . .	17
3.1 Sumowanie (określonej) stałej z obrazem oraz dwóch obrazów . . . . .	17
3.2 Mnożenie obrazu przez zadaną liczbę oraz przez inny obraz . . . . .	17
3.3 Mieszanie obrazów z określonym współczynnikiem . . . . .	17
3.4 Potęgowanie obrazu (z zadaną potegą) . . . . .	17
3.5 Dzielenie obrazu przez (zadaną) liczbę oraz przez inny obraz . . . . .	17
3.6 Pierwiastkowanie obrazu . . . . .	17
3.7 Logarytmowanie obrazu . . . . .	17
<b>Rozdział 4. Operacje sumowania arytmetycznego obrazów barwowych</b> . . . . .	18
<b>Rozdział 5. Operacje geometryczne na obrazie</b> . . . . .	19
<b>Rozdział 6. Operacje na histogramie obrazu szarego</b> . . . . .	20
<b>Rozdział 7. Operacje na histogramie obrazu barwowego</b> . . . . .	35
<b>Rozdział 8. Operacje morfologiczne na obrazach binarnych</b> . . . . .	52
<b>Rozdział 9. Operacje morfologiczne na obrazach szarych</b> . . . . .	53
<b>Rozdział 10. Filtrowanie liniowe i nielinowe</b> . . . . .	54
<b>Rozdział 11. Podsumowanie</b> . . . . .	99
<b>Bibliografia</b> . . . . .	100

# Spis rysunków

2.1	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512) . . . . .	7
2.2	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	7
2.3	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	9
2.4	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	9
2.5	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512) . . . . .	12
2.6	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	12
2.7	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	14
2.8	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512) . . . . .	14
6.1	Obraz szary, histogram szarości tego obrazu . . . . .	21
6.2	Obraz szary, histogram szarości tego obrazu . . . . .	21
6.3	Obraz szary wejściowy, histogram szarości tego obrazu . . . . .	23
6.4	Obraz szary przesunięty o 100, histogram szarości tego obrazu . . . . .	23
6.5	Obraz szary wejściowy, histogram szarości tego obrazu . . . . .	24
6.6	Obraz szary przesunięty o -100, histogram szarości tego obrazu . . . . .	24
6.7	Obraz wejściowy, histogram szarości tego obrazu . . . . .	26
6.8	Obraz po rozciągnięciu, histogram szarości tego obrazu . . . . .	26
6.9	Obraz wejściowy, histogram szarości tego obrazu . . . . .	27
6.10	Obraz po rozciągnięciu, histogram szarości tego obrazu . . . . .	27
6.11	Obraz wejściowy, histogram szarości tego obrazu . . . . .	29
6.12	Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu . . . . .	29
6.13	Obraz wejściowy, histogram szarości tego obrazu . . . . .	30
6.14	Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu . . . . .	30
6.15	Obraz wejściowy, histogram szarości tego obrazu . . . . .	32
6.16	Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu . . . . .	32
6.17	Obraz wejściowy, histogram szarości tego obrazu . . . . .	33
6.18	Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu . . . . .	33
7.1	Obraz barwny, histogram barw tego obrazu . . . . .	36
7.2	Obraz barwny, histogram barw tego obrazu . . . . .	36
7.3	Obraz wejściowy, histogram barw tego obrazu . . . . .	38
7.4	Obraz wyjściowy przesunięty o 50, histogram barw tego obrazu . . . . .	38
7.5	Obraz wejściowy, histogram barw tego obrazu . . . . .	39
7.6	Obraz wyjściowy przesunięty o -50, histogram barw tego obrazu . . . . .	39
7.7	Obraz wejściowy, histogram barw tego obrazu . . . . .	41
7.8	Obraz po rozciągnięciu, histogram barw tego obrazu . . . . .	41
7.9	Obraz wejściowy, histogram barw tego obrazu . . . . .	42

7.10 Obraz po rozciagnięciu, histogram barw tego obrazu . . . . .	42
7.11 Obraz wejściowy, histogram szarości tego obrazu . . . . .	46
7.12 Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu . .	47
7.13 Obraz wejściowy, histogram szarości tego obrazu . . . . .	47
7.14 Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu . .	47
7.15 Obraz wejściowy, histogram barw tego obrazu . . . . .	49
7.16 Obraz po progowaniu globalnym, histogram barw tego obrazu . . . . .	49
7.17 Obraz wejściowy, histogram barw tego obrazu . . . . .	50
7.18 Obraz po progowaniu globalnym, histogram barw tego obrazu . . . . .	50
10.1 Obraz wejściowy, obraz uśredniony . . . . .	55
10.2 Obraz wejściowy, obraz uśredniony . . . . .	55
10.3 Obraz wejściowy, obraz uśredniony . . . . .	56
10.4 Obraz wejściowy, obraz uśredniony . . . . .	56
10.5 Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	59
10.6 Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	60
10.7 Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	60
10.8 Obraz wejściowy, obraz po filtracji filtrem Gaussa . . . . .	60
10.9 Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	63
10.10 Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	64
10.11 Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	64
10.12 Obraz wejściowy, obraz po filtracji filtrem Roberts'a . . . . .	64
10.13 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	67
10.14 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	68
10.15 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	68
10.16 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a . . . . .	68
10.17 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	71
10.18 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	72
10.19 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	72
10.20 Obraz wejściowy, obraz po filtracji filtrem Sobel'a . . . . .	72
10.21 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	76
10.22 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	76
10.23 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	76
10.24 Obraz wejściowy, obraz po filtracji filtrem kompasowym . . . . .	77
10.25 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	82
10.26 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	82
10.27 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	82
10.28 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego . . . . .	83
10.29 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	87
10.30 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	87
10.31 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	88
10.32 Obraz wejściowy, obraz po filtracji filtrem medianowym . . . . .	88
10.33 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	91
10.34 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	91
10.35 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	92
10.36 Obraz wejściowy, obraz po filtracji filtrem maksymalnym . . . . .	92

10.37 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	95
10.38 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	95
10.39 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	96
10.40 Obraz wejściowy, obraz po filtracji filtrem minimalnym . . . . .	96

## Rozdział 1

# Wstęp

Laboratoria oh oh... [1]

## 1.1 Specyfikacja wykorzystanego formatu obrazu

## 1.2 Instrukcja obsługi programu

## Rozdział 2

# Operacje ujednolicania obrazów

## 1. Ujednolicenie obrazów szarych geometryczne

### Opis algorytmu

Operacja ujednoliciania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe.

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)

## Kod źródłowy

```

def geometricGray(self, show = False):
    width1 = self.im1.shape[1]
    width2 = self.im2.shape[1]
    maxWidth = width1 if width1 > width2 else width2

    height1 = self.im1.shape[0]
    height2 = self.im2.shape[0]
    maxHeight = height1 if height1 > height2 else height2

    resultImage1 = np.empty((maxHeight, maxWidth), dtype = np.uint8)
    resultImage2 = np.empty((maxHeight, maxWidth), dtype = np.uint8)

    startWidthCoord = int(round((maxWidth - width1) / 2))
    startHeightCoord = int(round((maxHeight - height1) / 2))

    for i in range(0, maxHeight):
        for j in range(0, maxWidth):
            resultImage1[i, j] = 1

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

            startWidthCoord = int(round((maxWidth - width2) / 2))
            startHeightCoord = int(round((maxHeight - height2) / 2))

            for i in range(0, maxHeight):
                for j in range(0, maxWidth):
                    resultImage2[i, j] = 1

                    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. Ujednolicenie obrazów szarych rozdzielczościowe

### Opis algorytmu

Operacja ujednoliciania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe.

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 znanych pikseli z jego bezpośredniego otoczenia.



Rysunek 2.3: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.4: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)

## Kod źródłowy

```

def rasterGray(self, show = False):
    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

    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]

    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

    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")
```

### 3. Ujednolicenie obrazów RGB geometryczne

#### Opis algorytmu

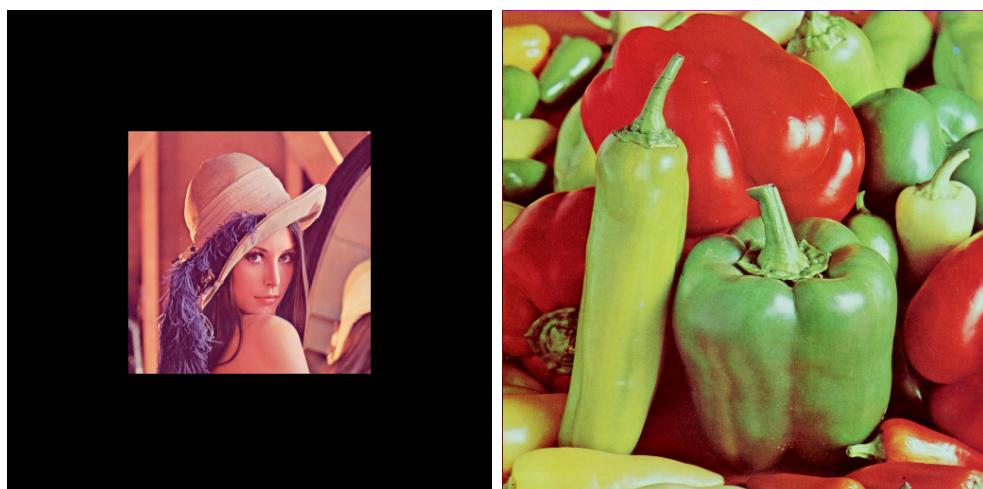
Operacja ujednolicania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe.

Operacja geometrycznego ujednolicenia obrazów polega na doprowadzeniu obydwu obrazów do takiej samej liczby wierszy piksli w każdym obrazie i takiej samej liczby kolumn piksli 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 2.5: Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)



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

## Kod źródłowy

```

def geometricColor(self, show = False):
    width1 = self.im1.shape[1]
    width2 = self.im2.shape[1]
    maxWidth = width1 if width1 < width2 else width2

    height1 = self.im1.shape[0]
    height2 = self.im2.shape[0]
    maxHeight = height1 if height1 < height2 else height2

    resultImage1 = np.empty((maxHeight, maxWidth, 3), dtype = np.uint8)
    resultImage2 = np.empty((maxHeight, maxWidth, 3), dtype = np.uint8)

    startWidthCoord = int(round((maxWidth - width1) / 2))
    startHeightCoord = int(round((maxHeight - height1) / 2))

    for i in range(0, maxHeight):
        for j in range(0, maxWidth):
            resultImage1[i, j] = (1, 1, 1)

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

    startWidthCoord = int(round((maxWidth - width2) / 2))
    startHeightCoord = int(round((maxHeight - height2) / 2))

    for i in range(0, maxHeight):
        for j in range(0, maxWidth):
            resultImage2[i, j] = (1, 1, 1)

    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")

```

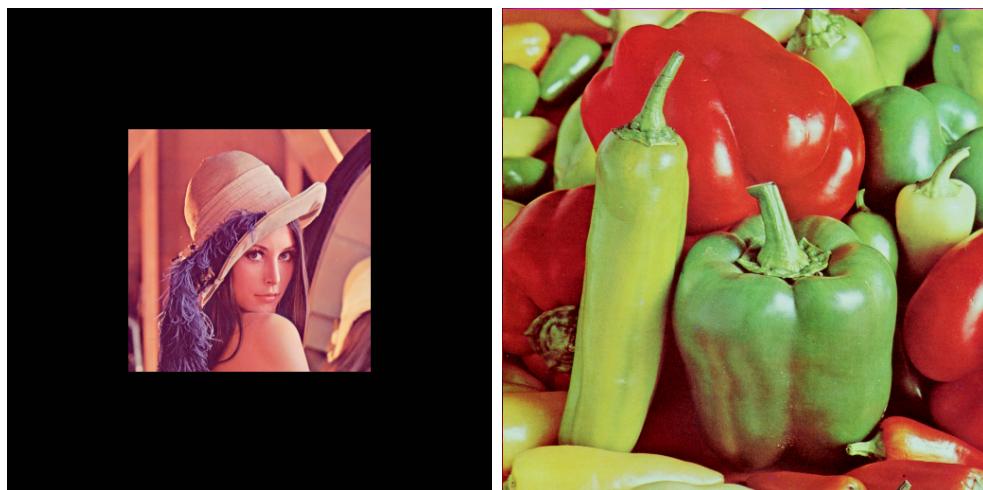
## 4. Ujednolicenie obrazów RGB rozdzielczościowe

### Opis algorytmu

Operacja ujednolicania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe.

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

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



Rysunek 2.7: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.8: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)

## Kod źródłowy

```

def rasterColor(self, show = False):
    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

    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]

    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

    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]
                            n += 1
            resultImage1[i, j] = [r/n, g/n, b/n]

```

```
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

**3.1 Sumowanie (określonej) stałej z obrazem oraz dwóch obrazów**

**3.2 Mnożenie obrazu przez zadaną liczbę oraz przez inny obraz**

**3.3 Mieszanie obrazów z określonym współczynnikiem**

**3.4 Potęgowanie obrazu (z zadaną potęgą)**

**3.5 Dzielenie obrazu przez (zadaną) liczbę oraz przez inny obraz**

**3.6 Pierwiastkowanie obrazu**

**3.7 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

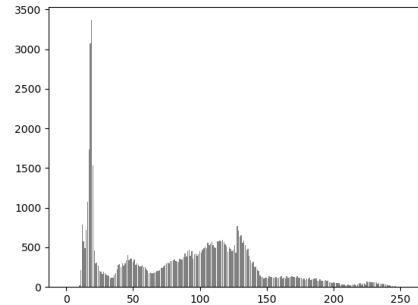
# Operacje na histogramie obrazu szarego

## 1. obliczanie histogramu

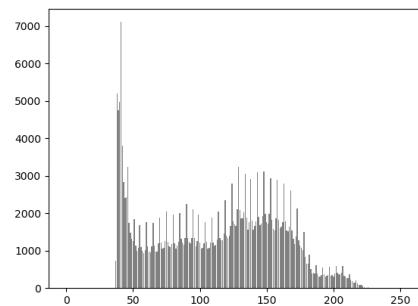
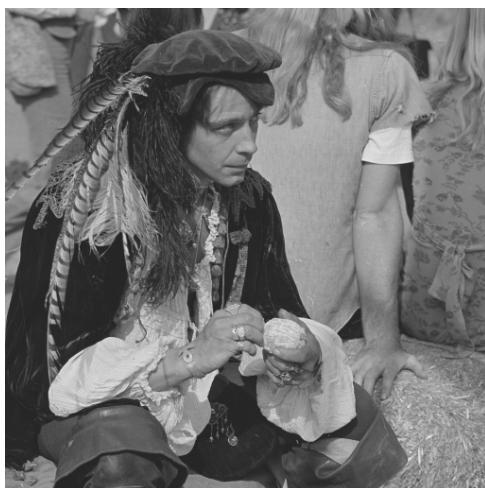
### Opis algorytmu

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

## Kod źródłowy

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

    width = image.shape[1]
    height = image.shape[0]
    hist = [0] * 256

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

    if plot:
        bins = np.arange(256)
        self.plotHistogram(bins, hist)

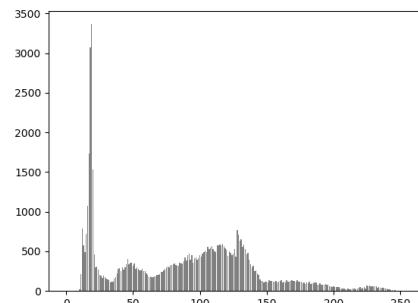
    return hist
```

## 2. przemieszczanie histogramu

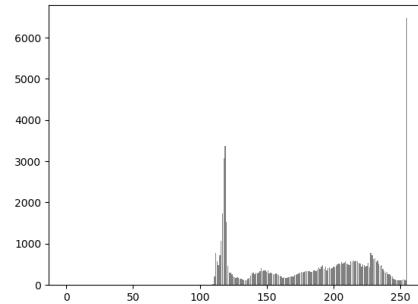
### Opis algorytmu

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.

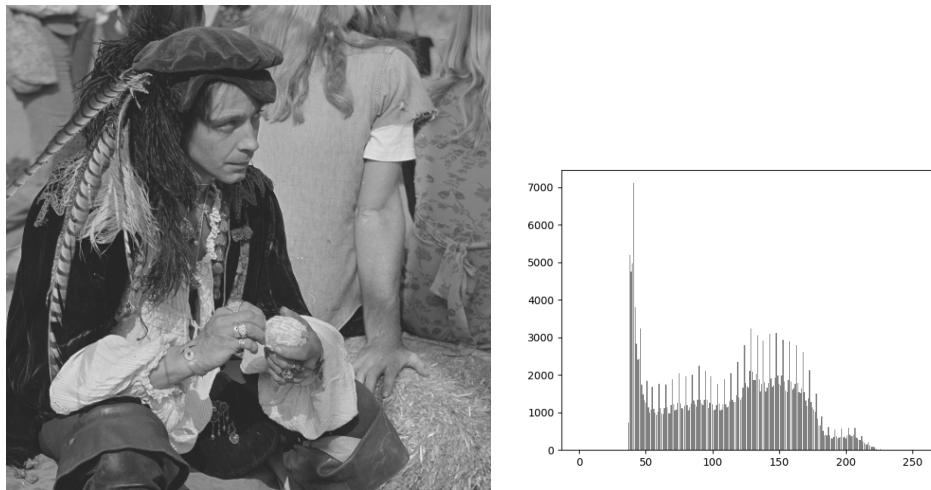
1. Do każdej wartości piksla dodaj podaną stałą o którą chcesz przemieścić histogram.
2. Jeśli wartość piksla po operacji dodawania wykracza poza zakres 255:
  3. Przypisz jej wartość 255.
  4. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0:
    5. Przypisz jej wartość 0.



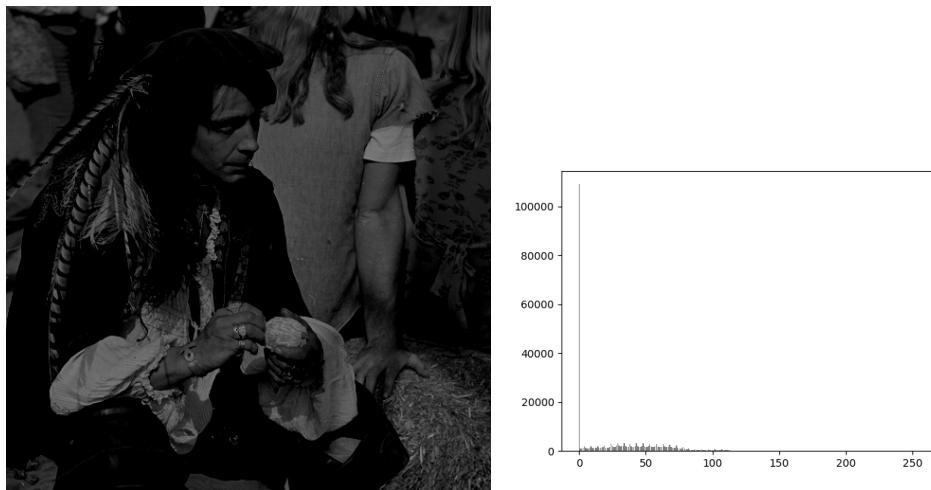
Rysunek 6.3: Obraz szary wejściowy, histogram szarości tego obrazu



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



Rysunek 6.5: Obraz szary wejściowy, histogram szarości tego obrazu



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

## Kod źródłowy

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

    resultImage = np.empty((height, width), dtype=np.uint8)

    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")
```

### 3. rozciąganie histogramu

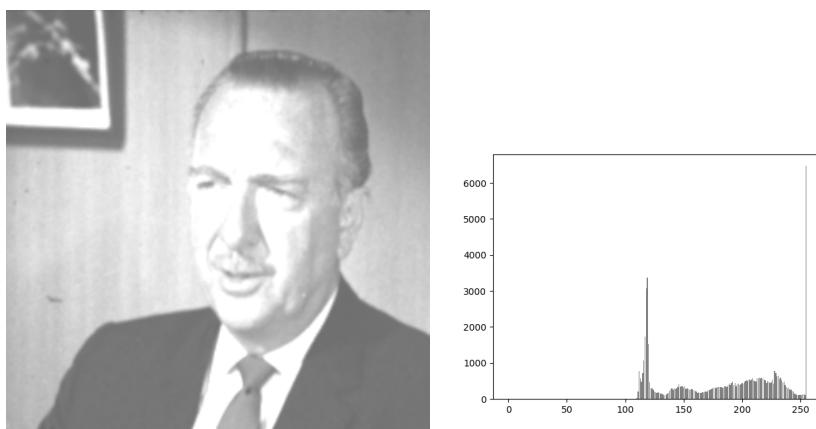
#### Opis algorytmu

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].

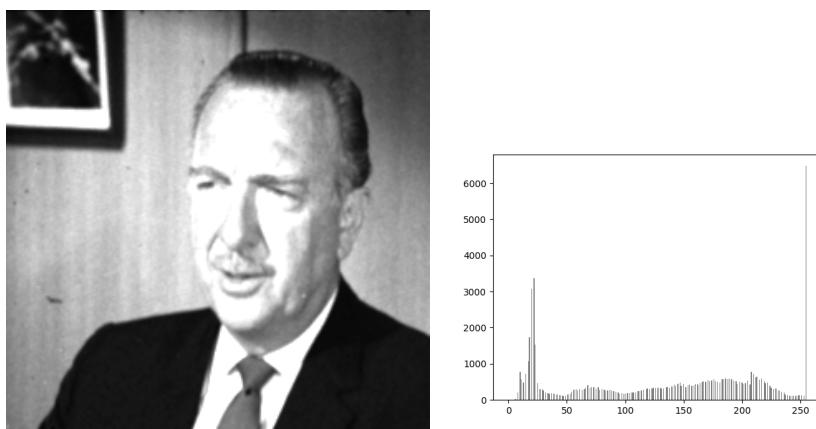
1. Znajdź w obrazie największą( $\max$ ) i najmniejszą( $\min$ ) wartość piksla
2. Dla każdego piksla:
3. Oblicz nową wartość piskla stosując wzór:

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

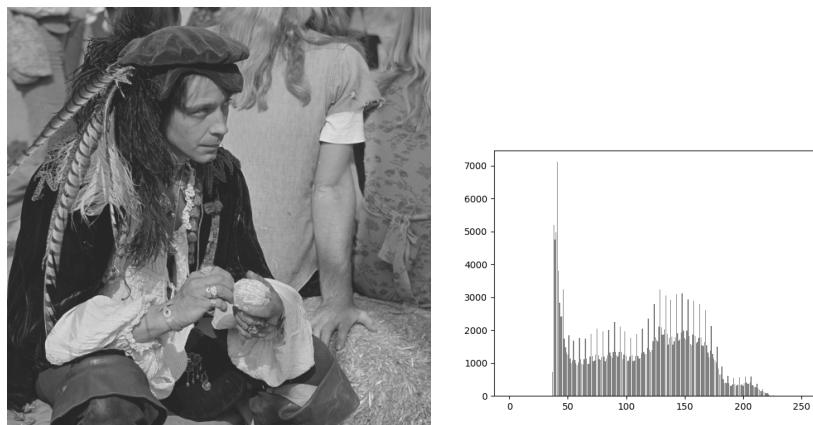
W taki sposób, jeśli odcienie szarości obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, odcienie szarości będą w zakresie [0, 255]



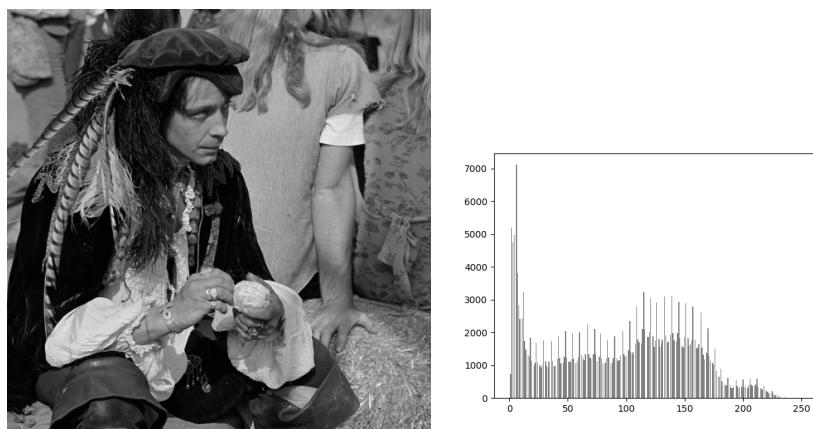
Rysunek 6.7: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.8: Obraz po rozciagnięciu, histogram szarości tego obrazu



Rysunek 6.9: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.10: Obraz po rozciągnięciu, histogram szarości tego obrazu

## Kod źródłowy

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

    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]

    maxValue = 0
    minValue = 255
    while maxValue != 255:

        for i in range(height):
            for j in range(width):
```

```
currValue = resultImage[i, j]
maxValue = max(maxValue, currValue)
minValue = min(minValue, currValue)

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

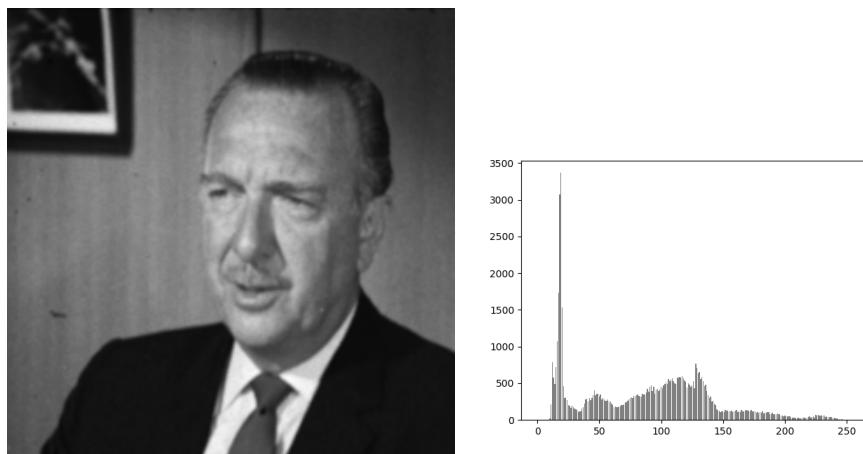
if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "stretchHist")
```

## 4. progowanie lokalne

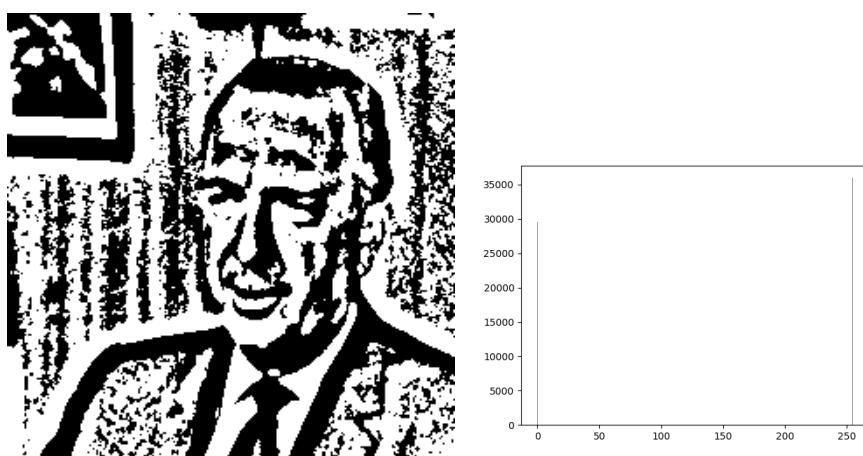
### Opis algorytmu

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.

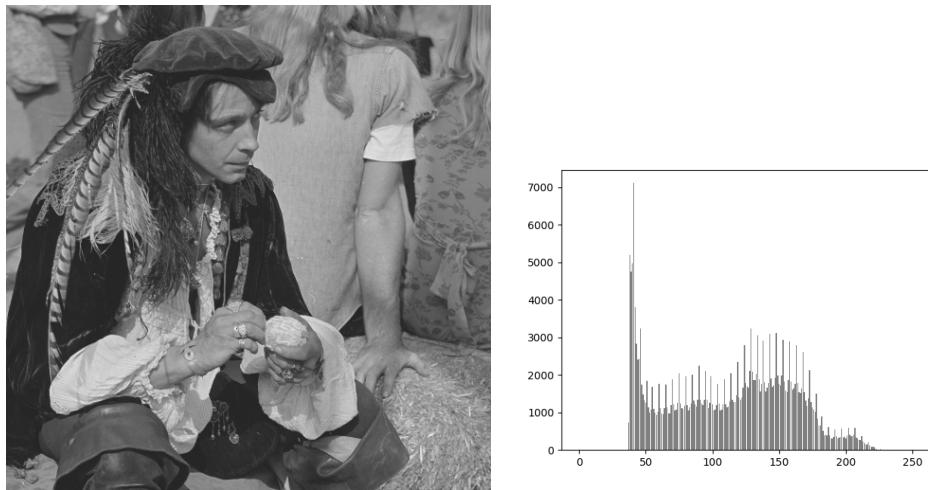
1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla:
3. Oblicz wartość progową jako średnią wartość piksli w otoczeniu danego piksla.
4. Jeśli wartość piksla środkowego jest  $< 0$ :
5. Przypisz mu wartość 0.
6. W przeciwnym przypadku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



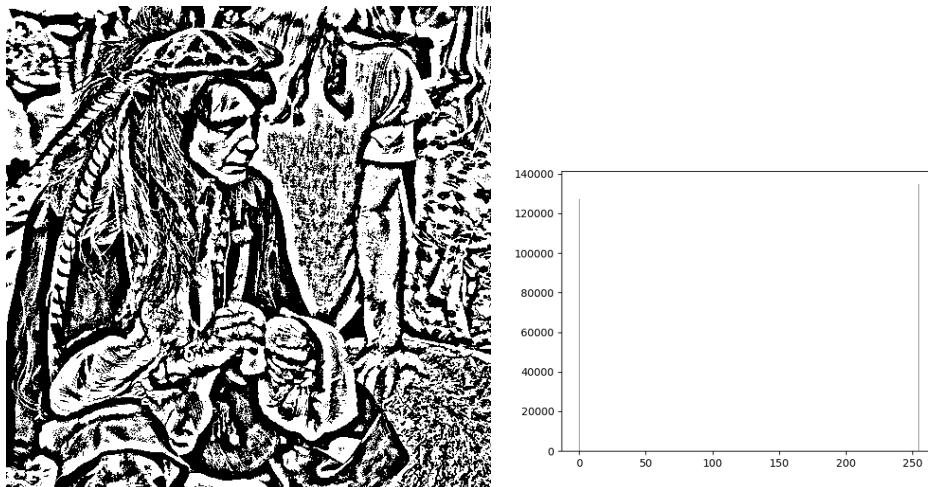
Rysunek 6.11: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.12: Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu



Rysunek 6.13: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.14: Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu

## Kod źródłowy

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

    resultImage = np.empty((height, width), dtype=np.uint8)

    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 + l)) else (i + iOff)
        jSafe = j if ((j + jOff) > (width + l)) else (j + jOff)
        threshold += self.im[iSafe, jSafe]
    n += 1
threshold = int(round(threshold / n))
resultImage[i, j] = 0 if (currPix < threshold) else 255

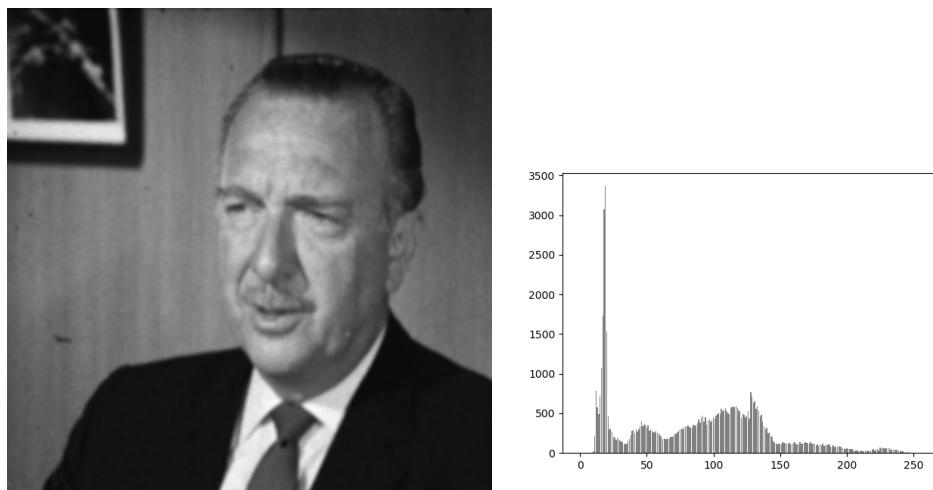
if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "locThreshold")
```

## 5. progowanie globalne

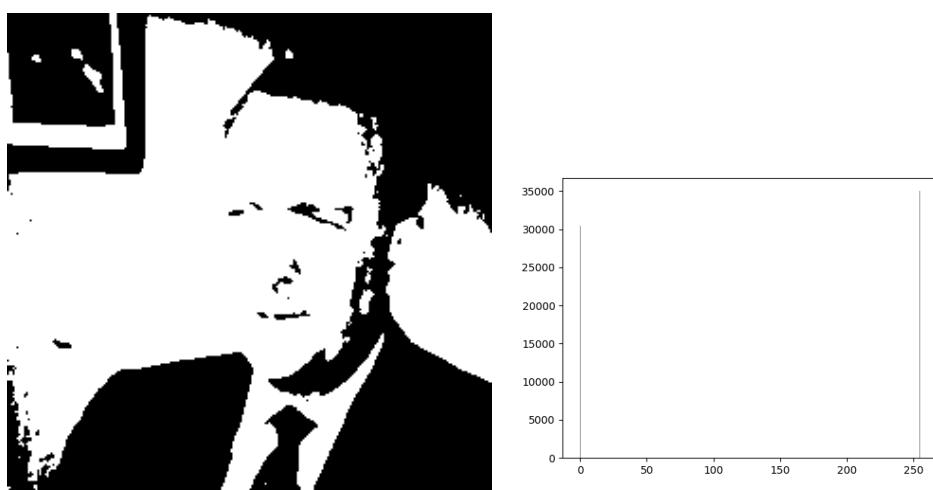
### Opis algorytmu

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 pikslowi obraz w wyniku jest binarny.

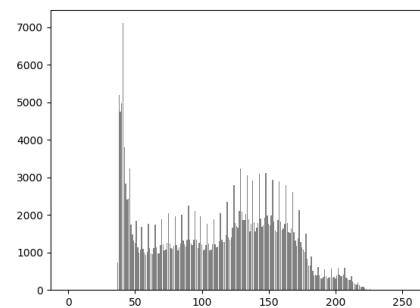
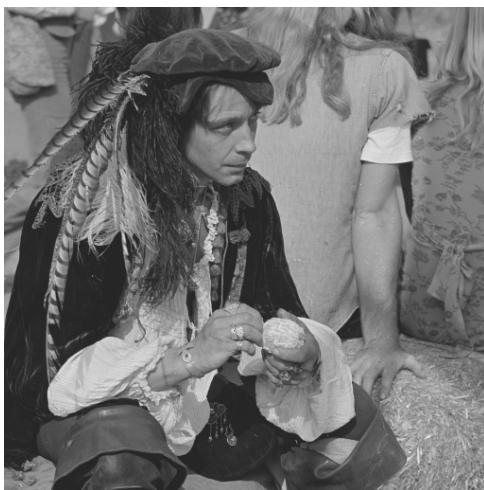
1. Oblicz wartość progową  $T$ , jako średnią wartość z wszystkich pikseli w obrazie
2. Dla każdego piksla:
3. Jeśli wartość danego piksla jest  $< T$ :
4. przypisz mu wartość 0.
5. W przeciwnym przypadku przypisz mu wartość 255.



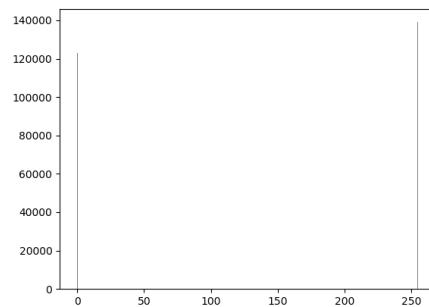
Rysunek 6.15: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.16: Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu



Rysunek 6.17: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.18: Obraz po progowaniu z parametrem 20, histogram szarości tego obrazu

## Kod źródłowy

```

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

    resultImage = np.empty((height, width), dtype=np.uint8)

    threshold = 0
    n = 0
    for i in range(height):
        for j in range(width):
            threshold += self.im[i, j]
            n += 1

    threshold = threshold / n
    for i in range(height):
        for j in range(width):
            if self.im[i, j] > threshold:
                resultImage[i, j] = 255
            else:
                resultImage[i, j] = 0

    if show:
        plt.imshow(resultImage, cmap='gray')
        plt.show()

    if plot:
        plt.hist(resultImage.ravel(), bins=256)
        plt.show()
  
```

```
n += 1
threshold = int(round(threshold / n))

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

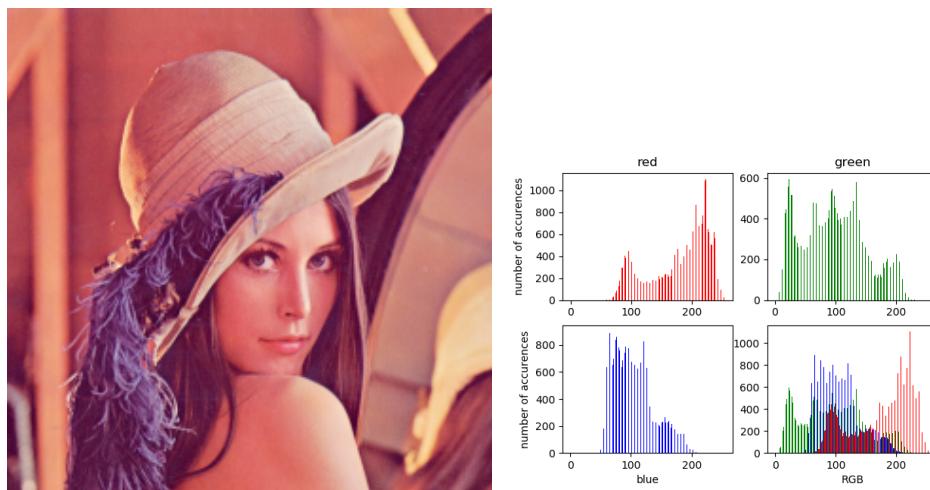
# Operacje na histogramie obrazu barwowego

## 1. obliczanie histogramu

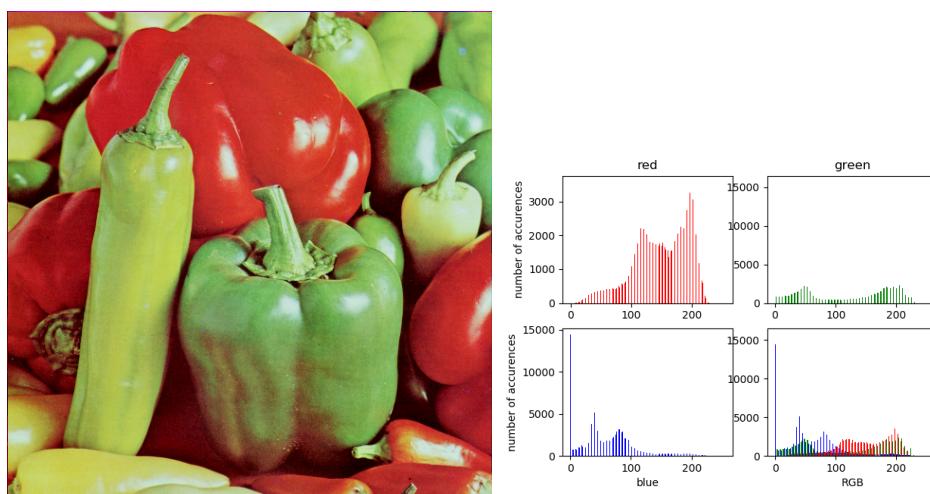
### Opis algorytmu

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

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



Rysunek 7.1: Obraz barwny, histogram barw tego obrazu



Rysunek 7.2: Obraz barwny, histogram barw tego obrazu

## Kod źródłowy

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

    width = image.shape[1]
    height = image.shape[0]
    hist = [0] * 3
    hist[0] = [0] * 256
    hist[1] = [0] * 256
    hist[2] = [0] * 256

    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:
        bins = np.arange(256)
        self.plotHistogram(bins, hist)

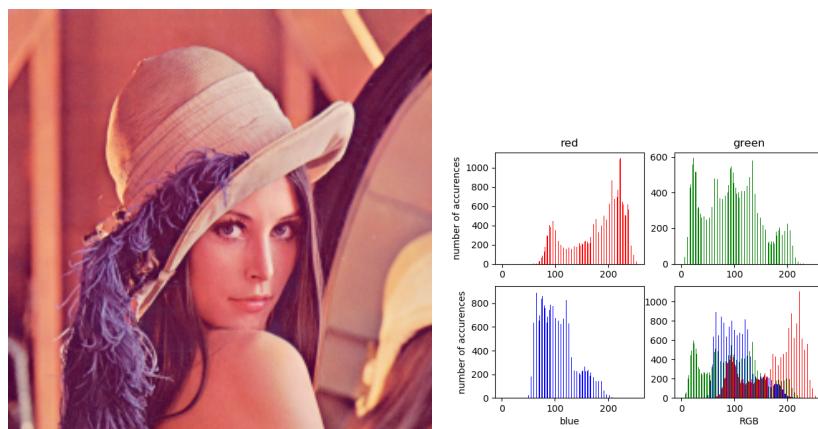
    return hist
```

## 2. Przemieszczanie histogramu

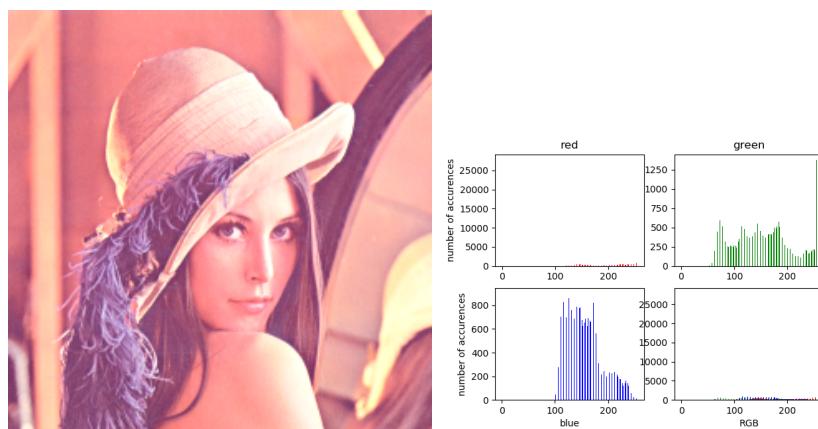
### Opis algorytmu

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.

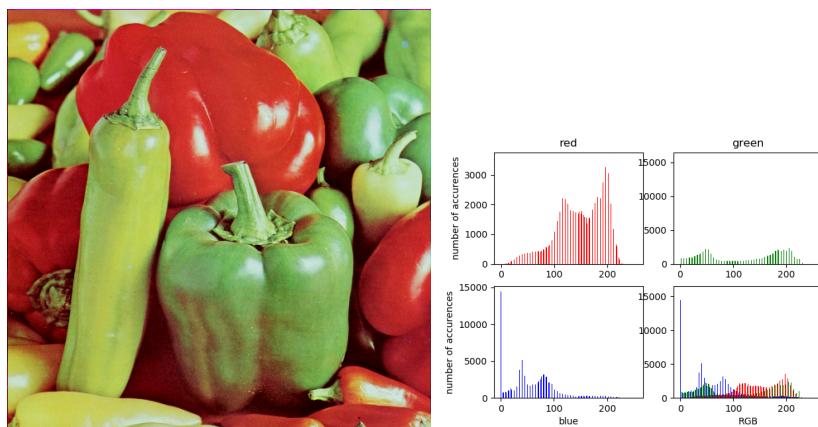
1. Do każdej wartości barwy piksla dodaj podaną stałą, o którą chcesz przemieścić histogram.
2. Jeśli wartość barwy piksla po operacji dodawania wykracza poza zakres 255: Przypisz jej wartość 255.
3. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0: Przypisz jej wartość 0.



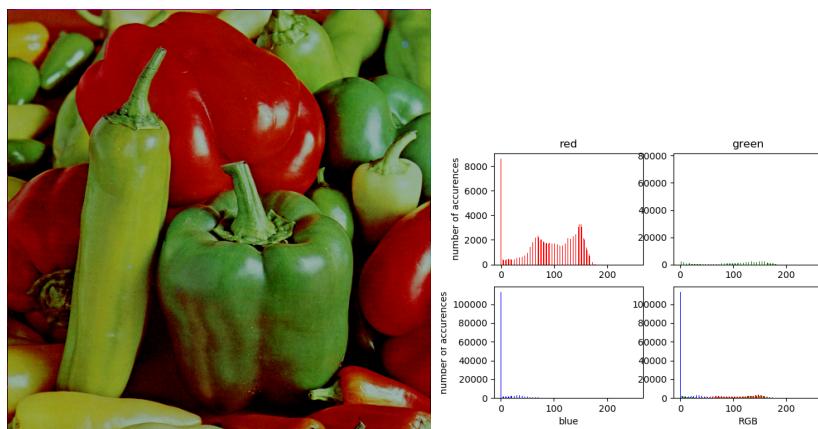
Rysunek 7.3: Obraz wejściowy, histogram barw tego obrazu



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

## Kod źródłowy

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

    resultImage = np.empty((height, width, 3), dtype=np.uint8)

    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")
```

### 3. Rozciąganie histogramu

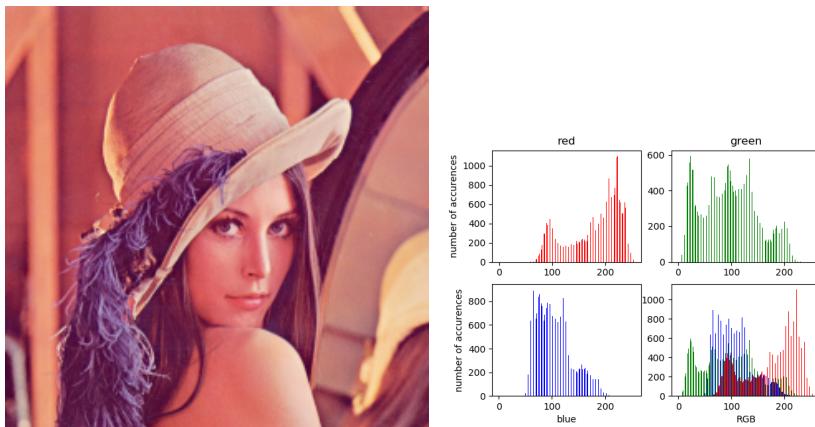
#### Opis algorytmu

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].

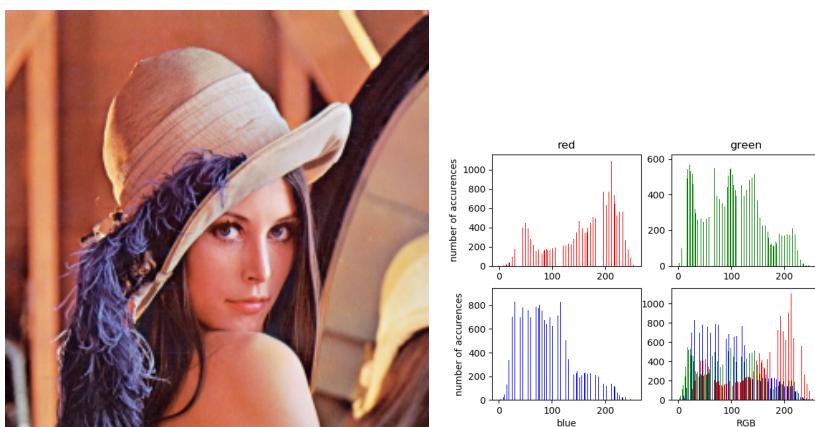
1. Znajdź w obrazie największą( $\max_c$ ) i najmniejszą( $\min_c$ ) wartość piksla dla każdej z barw( $c$ )
2. Dla każdego piksla( $P_o$ ):
3. Dla każdej z barw( $c$ ):
4. Oblicz nową wartość piksla( $P_n$ ) stosując wzór:

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

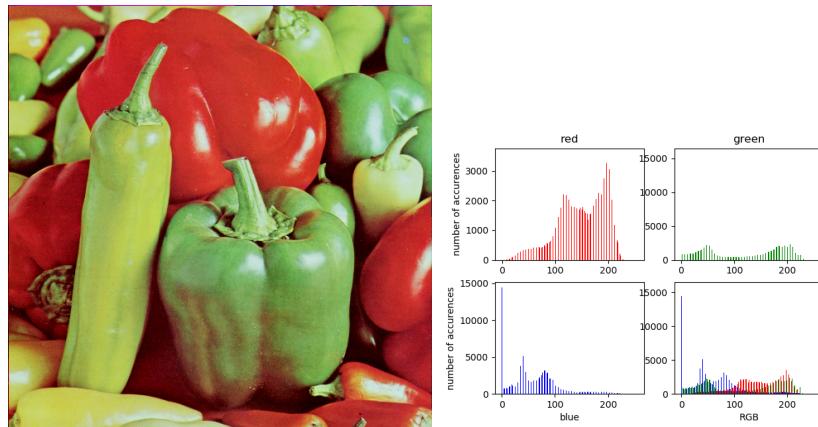
W taki sposób, jeśli barwy obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, będą w zakresie [0, 255].



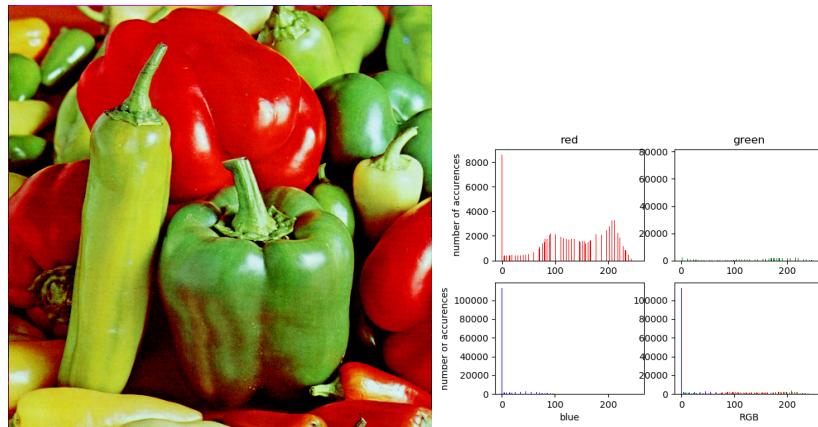
Rysunek 7.7: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.8: Obraz po rozciągnięciu, histogram barw tego obrazu



Rysunek 7.9: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.10: Obraz po rozcięgnięciu, histogram barw tego obrazu

## Kod źródłowy

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

    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]

    maxValue = [0] * 3
    minValue = [255] * 3
    while (maxValue[0] != 255) & (maxValue[1] != 255) & (maxValue[2] != 255):
```

```
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])

        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")
```

## 4. Progowanie 1-progowe

### Opis algorytmu

1.

## 5. Progowanie wielo-progowe

### Opis algorytmu

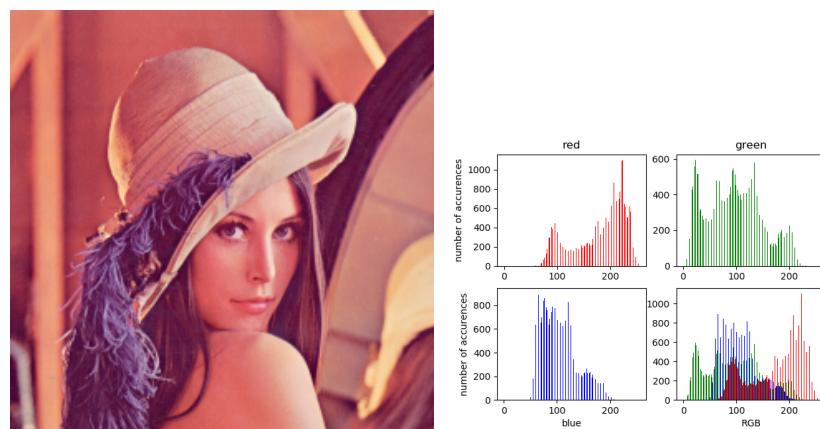
1.

## 6. progowanie lokalne

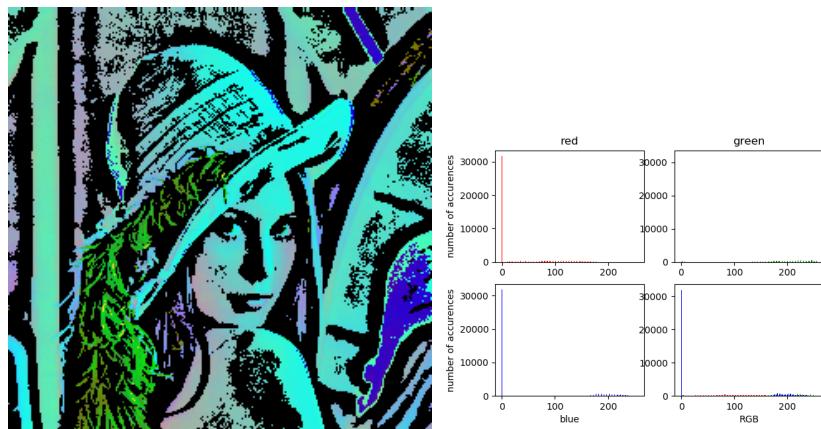
### Opis algorytmu

Progowanie lokalne oblicza wartość progową dla każdego piksla z osobna. W wyniku takiego progowania obraz dokładniej odwzorowuje kształt obiektu.

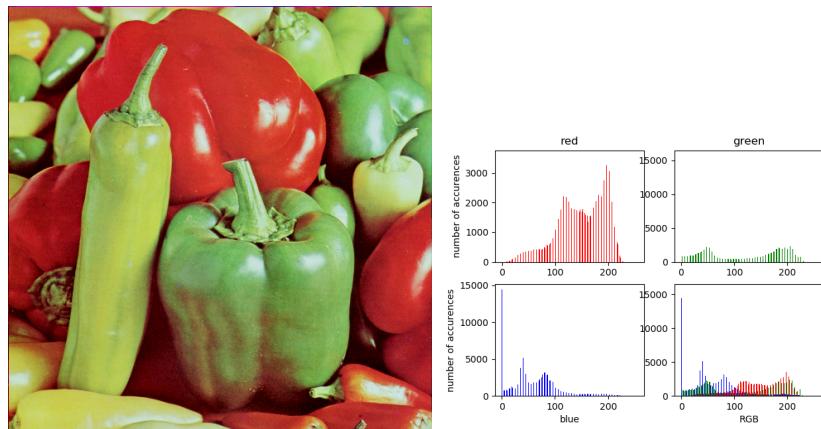
1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Przenieś obraz do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Oblicz wartość progową składowej  $I$  jako średnią wartość piksli w otoczeniu danego piksla( $P$ ).
5. Jeśli wartość składowej  $I$  piksla  $P$  jest  $< 0$ :
6. Przypisz mu wartość 0.
7. W przeciwnym przypadku przypisz mu wartość 1.
8. Przenieś obraz spowrotem do przestrzeni RGB.



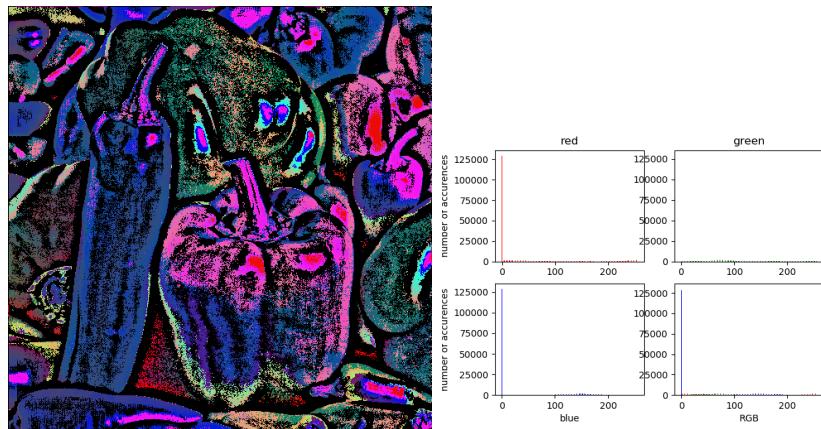
Rysunek 7.11: Obraz wejściowy, histogram szarości tego obrazu



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

## Kod źródłowy

```

def localThreshold(self, dim = 3, show = False, plot = False):
    width = self.im.shape[1]
    height = self.im.shape[0]
    low, up = -(int(round(dim / 2))), int(round(dim / 2) + 1)

    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    tmp = np.empty((height, width, 3))
    tmp2 = np.empty((height, width, 3))

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))

    for i in range(height):
        for j in range(width):
            H = 0
            n = 0
            currPix = tmp[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)
                    H += tmp[iSafe, jSafe][2]
                    n += 1
            H = H / n
            tmp2[i, j] = currPix
            tmp2[i, j][2] = 0 if currPix[2] < H else 1

    for i in range(height):
        for j in range(width):
            h, s, I = tmp2[i, j]
            resultImage[i, j] = self.HSItorRGB((h, s, I))

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

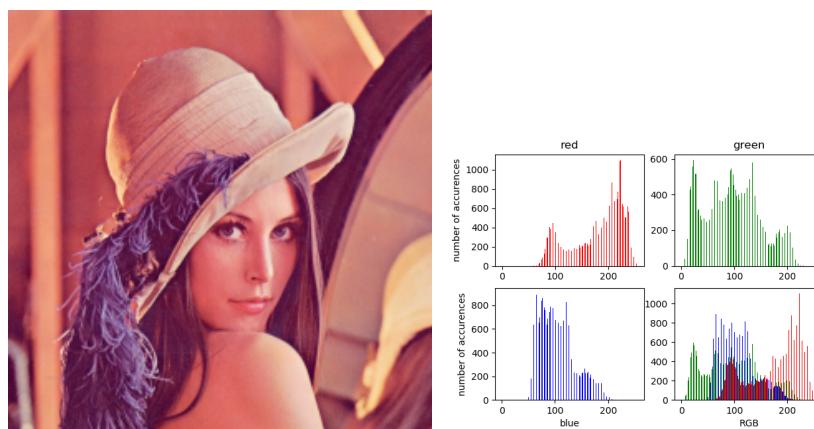
```

## 7. Progowanie globalne

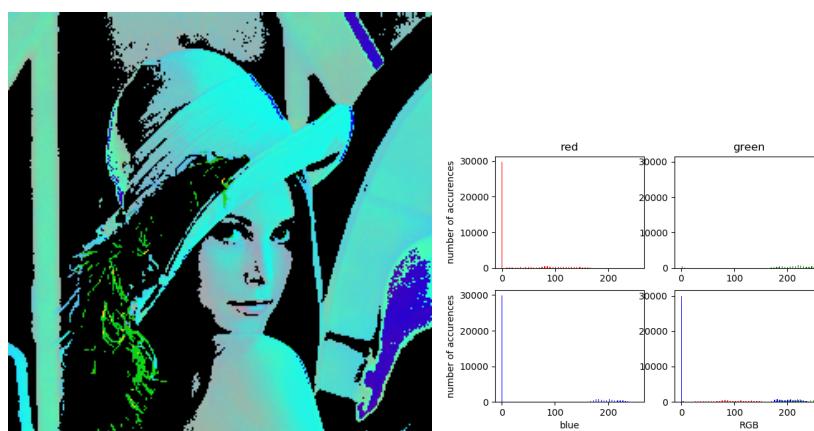
### Opis algorytmu

W progowaniu globalnym wartość progowa jest ustalana globalnie, biorąc pod uwagę wartość składowej intensywnościowej (z przestrzeni HSI) każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa, jest stosowana do nadania każdemu pikselowi nową wartość.

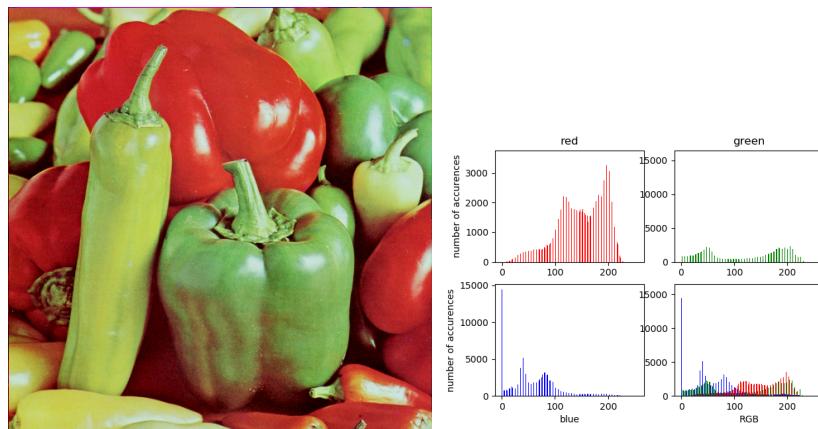
1. Przenieś obraz do przestrzeni HSI.
2. Oblicz wartość progową  $T$ , jako średnią wartość składowej intensywnościowej ( $I$ ) z wszystkich piskli w obrazie.
3. Dla każdego piskla:
4. Jeśli wartość  $I$  danego piskla jest  $< T$ :
5. przypisz mu wartość 0.
6. W przeciwnym przypadku przypisz mu wartość 1.
7. Przenieś obraz spowrotem do przestrzeni RGB.



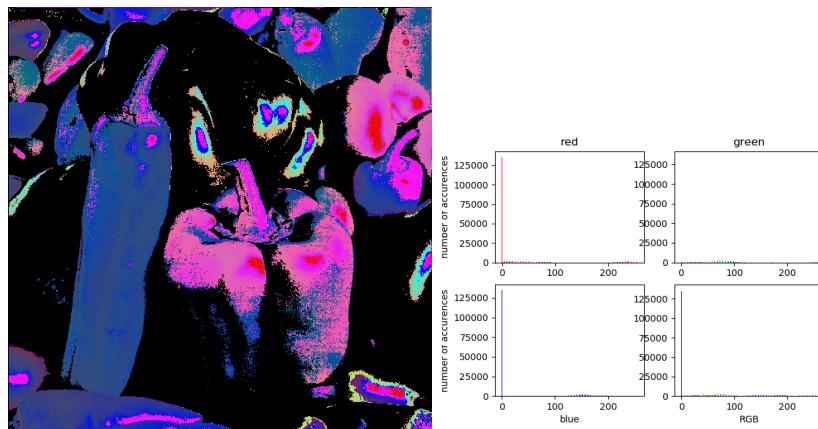
Rysunek 7.15: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.16: Obraz po progowaniu globalnym, histogram barw tego obrazu



Rysunek 7.17: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.18: Obraz po progowaniu globalnym, histogram barw tego obrazu

## Kod źródłowy

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

    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    tmp = np.empty((height, width, 3))
    tmp2 = np.empty((height, width, 3))

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))
            tmp2[i, j] = self.globalThreshold(tmp[i, j])

    resultImage = tmp2
```

```
H = 0
n = 0
for i in range(height):
    for j in range(width):
        H += tmp[i, j][2]
        n += 1
        H = H / n

for i in range(height):
    for j in range(width):
        tmp2[i, j] = tmp[i, j]
        tmp2[i, j][2] = 0 if tmp[i, j][2] < H else 1

for i in range(height):
    for j in range(width):
        h, s, I = tmp2[i, j]
        resultImage[i, j] = self.HSItoRGB((h, s, I))

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

## Rozdział 8

# **Operacje morfologiczne na obrazach binarnych**

1. okrawanie(erodzja)
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**

## 1.1 Filtr dolnoprzepustowy uśredniający

### Opis algorytmu

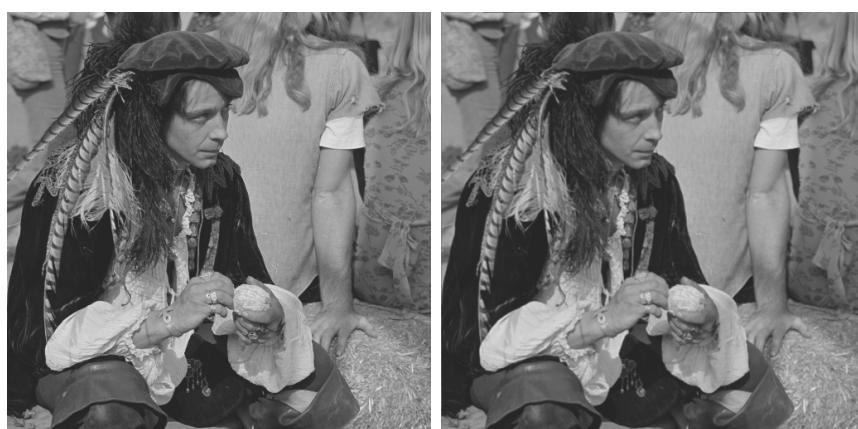
Filtr uśredniający jest podstawowym filtrem dolnoprzepustowym, jego wynikiem jest uśrednienie każdego piksla razem ze swoimi sąsiadami. Maska:

	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 pikseli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Sumę barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikselowi  $P$ .



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

**Kod źródłowy dla obrazów szarych**

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

    resultImage = np.empty((height, width), dtype=np.uint8)

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

    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")
```

## Kod źródłowy dla obrazów barwnych

```

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

    resultImage = np.empty((height, width, 3), dtype=np.uint8)

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

    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")

```

## 1.2 Filtr dolnoprzepustowy Gaussowski

### Opis algorytmu

Filtr Gaussa jest filtrem uśredniającym. Jego maska aproksymuje 2-wymiarową krzywą Gaussa. W odniesieniu do 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

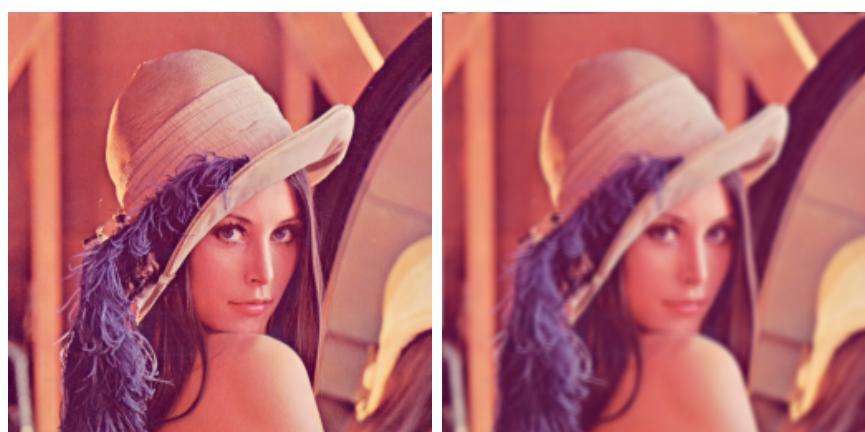
1. Dla każdego piksla( $P$ ):
2. Dla każdej barwy:
3. Zsumuj wartości barwy piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
4. Sumę wartości barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikselowi  $P$ .



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

## Kod źródłowy dla obrazów szarych

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

    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

    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")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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

    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")

```

## 2.1 Operator Roberts'a

### Opis algorytmu

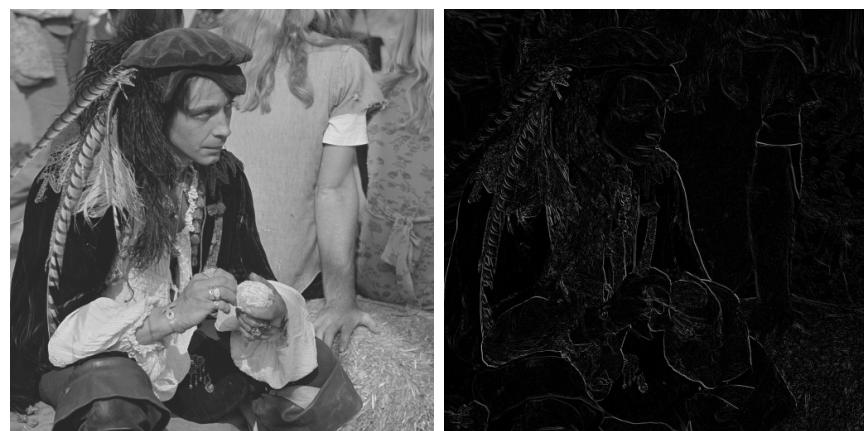
Filtr Roberts'a jest jednym z najbardziej znanych filtrów do wykrywania krawędzi w obrazie. Wynikowa wartość składowej po zastosowaniu owego filtra może wyjść ujemna, aby temu zapobiec należy użyć wartości bezwzględnej. Filtr Roberts'a jest bardzo wrażliwy na szum i ma niski poziom reakcji na krawędź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

0	0	0
0	0	-1
0	1	0

1. Jeśli obraz kolorowy:
2. Przejdź do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Zsumuj wartości (składowej intensywnościowej dla obrazu barwnego) piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Przypisz nową wartość barwy pikslowi  $P$ .



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

## Kod źródłowy dla obrazów szarych

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

    resultImage = np.empty((height, width), dtype=np.uint8)

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

    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")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))

    for i in range(height):
        for j in range(width):
            I = 0
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) | ((j + jOff) < 0) else (j + jOff)
                    I += tmp[iSafe, jSafe][2] * mask[iOff + 1, jOff + 1] * 255
            tmp2[i, j] = tmp[i, j]
            tmp2[i, j][2] = abs(I / 255)

    for i in range(height):
        for j in range(width):
            h, s, I = tmp2[i, j]
            resultImage[i, j] = self.HSItoRGB((h, s, I))

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

```

## 2.2 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
-1	0	1
-1	0	1

1. Jeśli obraz kolorowy:
2. Przejdź do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Zsumuj wartości (składowej intensywnościowej dla obrazu barwnego) piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Przypisz nową wartość barwy pikslowi  $P$ .



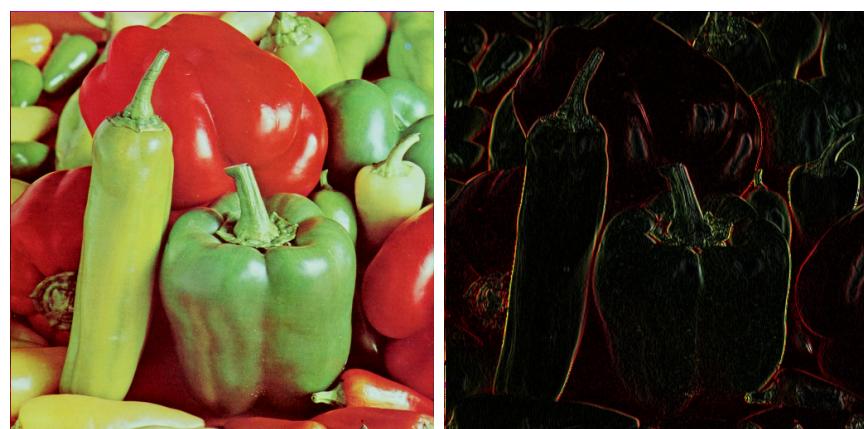
Rysunek 10.13: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



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

## Kod źródłowy dla obrazów szarych

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

    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

    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, "highpassPrewitt")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))

    for i in range(height):
        for j in range(width):
            I = 0
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) | ((j + jOff) < 0) else (j + jOff)
                    I += tmp[iSafe, jSafe][2] * mask[iOff + 1, jOff + 1] * 255
            tmp2[i, j] = tmp[i, j]
            tmp2[i, j][2] = abs(I / 255)

    for i in range(height):
        for j in range(width):
            h, s, l = tmp2[i, j]
            resultImage[i, j] = self.HSItoRGB((h, s, l))

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

```

## 2.3 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:

1	2	1
0	0	0
-1	-2	-1

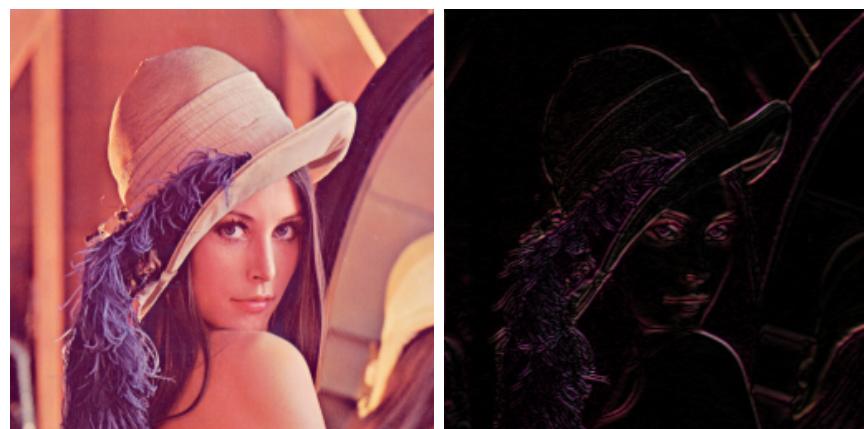
1. Jeśli obraz kolorowy:
2. Przejdź do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Zsumuj wartości (składowej intensywnościowej dla obrazu barwnego) piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski.
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Przypisz nową 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

## Kod źródłowy dla obrazów szarych

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

    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

    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, "highpassSobol")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))

    for i in range(height):
        for j in range(width):
            I = 0
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) | ((j + jOff) < 0) else (j + jOff)
                    I += tmp[iSafe, jSafe][2] * mask[iOff + 1, jOff + 1] * 255
            tmp2[i, j] = tmp[i, j]
            tmp2[i, j][2] = abs(I / 255)

    for i in range(height):
        for j in range(width):
            h, s, I = tmp2[i, j]
            resultImage[i, j] = self.HSItoRGB((h, s, I))

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

```

### 3.1 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:

-1	0	1
-2	0	2
-1	0	1

0	1	2
-1	0	1
-2	-1	0

1	2	1
0	0	0
-1	-2	-1

2	1	0
1	0	-1
0	-1	-2

1	0	-1
2	0	-2
1	0	-1

0	-1	-2
1	0	-1
2	1	0

-1	-2	-1
0	0	0
1	2	1

-2	-1	0
-1	0	1
0	1	2

1. Jeśli obraz kolorowy:
2. Przejdź do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Dla każdej z masek( $M$ ):
5. Zsumuj wartości (składowej intensywnościowej dla obrazu barwnego) piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski  $M$ .
6. Zastosój wartość bezwzględną na otrzymanej sumie.

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



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

## Kod źródłowy dla obrazów szarych

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

    resultImage = np.empty((height, width), dtype=np.uint8)
    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
```

```
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))
```

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

## Kod źródłowy dla obrazów barwnych

```

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

    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    tmp = np.empty((height, width, 3))
    tmp2 = np.empty((height, width, 3))
    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

for i in range(height):
    for j in range(width):
        r, g, b = self.im[i, j]
        tmp[i, j] = self.RGBtoHSI((r, g, b))

        for i in range(height):
            for j in range(width):
                I = [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):
                            I[k] += tmp[iSafe, jSafe][2] * mask[k][iOff + 1, jOff + 1]

                tmp2[i, j] = tmp[i, j]
                tmp2[i, j][2] = max(I)

            for i in range(height):
                for j in range(width):
                    h, s, l = tmp2[i, j]
                    resultImage[i, j] = self.HSItoRGB((h, s, l))

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

```

## 3.2 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:

-1	0	1
-1	0	1
-1	0	1

1	1	1
0	0	0
-1	-1	-1

1	0	-1
1	0	-1
1	0	-1

-1	-1	-1
0	0	0
1	1	1

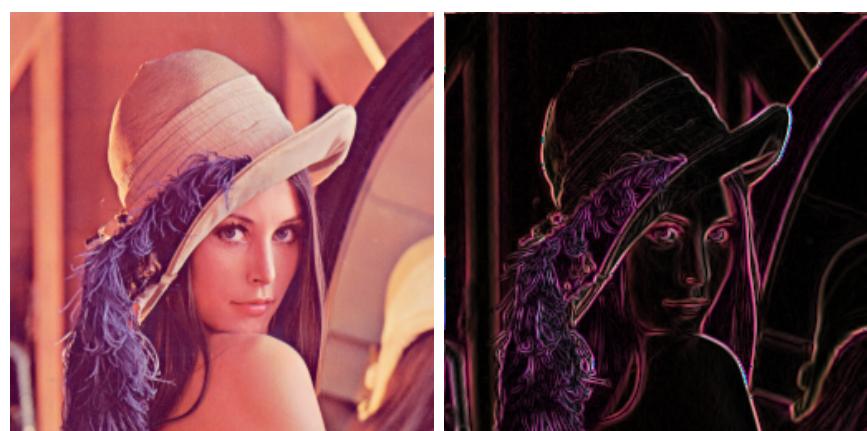
1. Jeśli obraz kolorowy:
2. Przejdź do przestrzeni HSI.
3. Dla każdego piksla( $P$ ):
4. Dla każdej z masek( $M$ ):
5. Zsumuj wartości (składowej intensywnościowej dla obrazu barwnego) piksli, otaczających piksel  $P$  pomnożonych przez odpowiednią wagę maski  $M$ .
6. Zastosuj wartość bezwzględną na otrzymanej sumie.
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

## Kod źródłowy dla obrazów szarych

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

    resultImage = np.empty((height, width), dtype=np.uint8)

    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

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, "vdgPrewitt")
```

## Kod źródłowy dla obrazów barwnych

```

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

    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    tmp = np.empty((height, width, 3))
    tmp2 = np.empty((height, width, 3))

    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

    for i in range(height):
        for j in range(width):
            r, g, b = self.im[i, j]
            tmp[i, j] = self.RGBtoHSI((r, g, b))

    for i in range(height):
        for j in range(width):

```

```
I = [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):
            I[k] += tmp[iSafe, jSafe][2] * mask[k][iOff + 1, jOff + 1]

tmp2[i, j] = tmp[i, j]
tmp2[i, j][2] = max(I) / 2

for i in range(height):
    for j in range(width):
        h, s, l = tmp2[i, j]
        resultImage[i, j] = self.HSItoRGB((h, s, l))

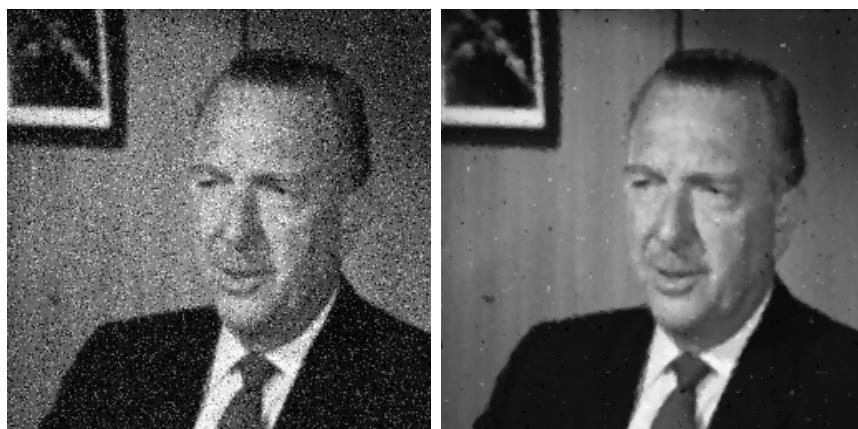
if show:
    self.show(Image.fromarray(resultImage, "RGB"))
    self.save(resultImage, self.imName, "vdgPrewitt")
```

## 4. 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  $P$ .



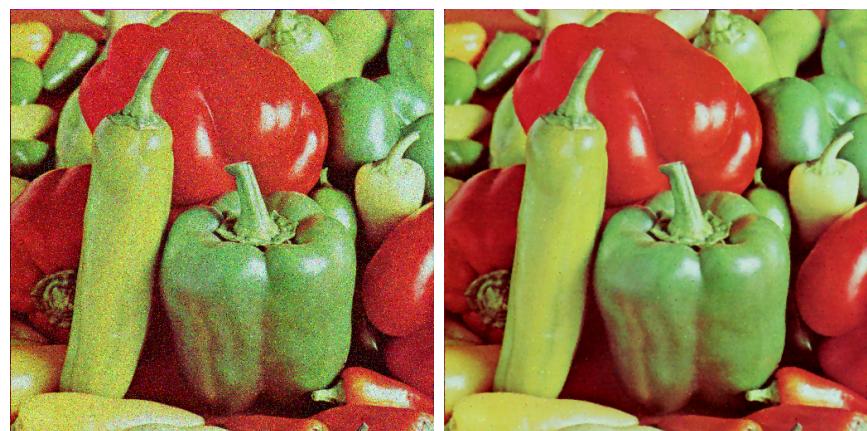
Rysunek 10.29: Obraz wejściowy, obraz po filtracji filtrem medianowym



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

## Kod źródłowy dla obrazów szarych

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

    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")
```

## Kod źródłowy dla obrazów szarych

```

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

    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")

```

## 5.1. Filtr maksymalny

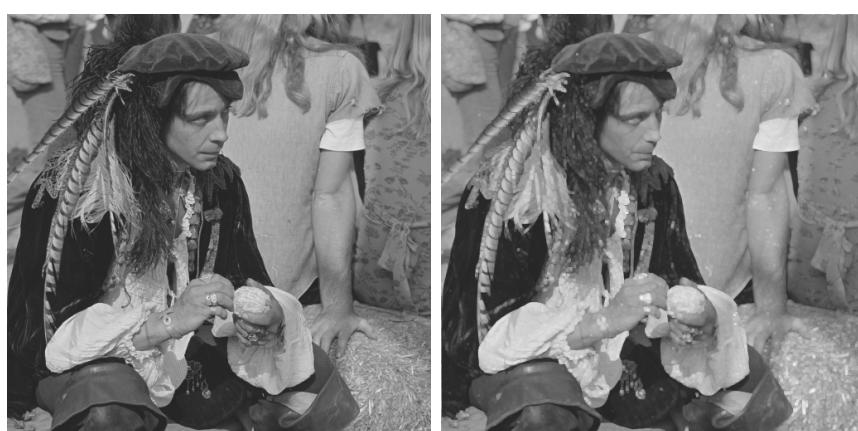
### Opis algorytmu

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

## Kod źródłowy dla obrazów szarych

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

    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")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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")

```

## 5.2. 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

## Kod źródłowy dla obrazów szarych

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

    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")
```

## Kod źródłowy dla obrazów barwnych

```

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

    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")

```

## 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.