

Przetwarzanie Obrazów: Sprawozdanie

Damian Ubowski

Warszawa, 2020

Spis treści

1 Wstęp	5
1.1 Format obrazu	5
1.1.1 Struktura formatu	5
1.1.2 Instrukcja obsługi programu	6
2 Operacje ujednoliciania obrazów	7
2.1 Ujednolicenie obrazów szarych geometryczne	8
2.2 Ujednolicenie obrazów szarych rozdzielczościowe	12
2.3 Ujednolicenie obrazów RGB geometryczne	16
2.4 Ujednolicenie obrazów RGB rozdzielczościowe	20
3 Operacje sumowania arytmetycznego obrazów szarych	25
3.1 Sumowanie określonej stałej z obrazem	26
3.2 Sumowanie dwóch obrazów	28
3.3 Mnożenie obrazu przez zadaną liczbę	31
3.4 Mnożenie obrazu przez inny obraz	35
3.5 Mieszanie obrazów z określonym współczynnikiem	38
3.6 Potęgowanie obrazu z zadaną potegą	40
3.7 Dzielenie obrazu przez liczbę	42
3.8 Dzielenie obrazu przez inny obraz	43
3.9 Pierwiastkowanie obrazu	46
3.10 Logarytmowanie obrazu	48
4 Operacje sumowania arytmetycznego obrazów barwowych	51
4.1 Sumowanie (określonej) stałej z obrazem	52
4.2 Sumowanie dwóch obrazów	54
4.3 Mnożenie obrazu przez zadaną liczbę	57
4.4 Mnożenie obrazu przez inny obraz	59
4.5 Mieszanie obrazów z określonym współczynnikiem	62
4.6 Potęgowanie obrazu (z zadaną potegą)	64
4.7 Dzielenie obrazu przez liczbę	66
4.8 Dzielenie obrazu przez inny obraz	67
4.9 Pierwiastkowanie obrazu	70
4.10 Logarytmowanie obrazu	72
5 Operacje geometryczne na obrazie	75
5.1 Przesunięcie obrazu o zadany wektor	76
5.2 Jednorodne skalowanie obrazu	78
5.3 Niejednorodne skalowanie obrazu	80
5.4 Obracanie obrazu o dowolny kąt	82
5.5 Symetrie względem osi układu	84
5.6 Symetrie względem zadanej prostej	87
5.7 Wycinanie fragmentów obrazu	89

5.8 Kopiowanie fragmentów obrazów	91
6 Operacje na histogramie obrazu szarego	93
6.1 Obliczanie histogramu	94
6.2 Przemieszczanie histogramu	96
6.3 Rozciąganie histogramu	99
6.4 Progowanie lokalne	102
6.5 Progowanie globalne	107
7 Operacje na histogramie obrazu barwowego	111
7.1 Obliczanie histogramu	112
7.2 Przemieszczanie histogramu	114
7.3 Rozciąganie histogramu	118
7.4 Progowanie 1-progowe lokalne	120
7.5 Progowanie wielo-progowe lokalne	123
7.6 Progowanie 1-progowe globalne	126
7.7 Progowanie wielo-progowe globalne	128

Rozdział 1

Wstęp

1.1 Format obrazu

Wybranym formatem obrazów cyfrowych jest PNG (*ang. The Portable Network Graphics*), a jego możliwości sprowadzają się do: obsługi kanału alfa (w celu imitacji przezroczystości obrazu), korekcji gamma (w celu dostosowania jasności obrazu), czymś co nazywa się *two-dimensional interlacing*, oraz dobrej jakości kompresję bezstratną, która w porównaniu wypada lepiej na korzyść PNG, a nie GIF. Natomiast wybrany format PNG nie wspiera żadnego rodzaju animacji, co jest korzyścią dla GIF. PNG jest formatem rastrowym co oznacza, że jego strukturę można sprowadzić do dwuwymiarowej tablicy (macierzy) w której komórkach (pikselach) trzymane są wartości dotyczące kolorów.

1.1.1 Struktura formatu

Pliki PNG rozpoczynają się od swojej “Magic number” potwierdzającej rodzaj pliku i mającej wartość *137 80 78 71 13 10 26 10*. Następnie czerpiąc inspirację ze struktury IFF (**Interchange File Format**) plik dzieli się na kawałki (*ang. chunks*) zawierające interesujące nas cenne dane - takie jak szerokość lub wysokość obrazu, dpi, informacje o kolorach, rozmieszczeniu pikseli, etc.

Każda informacja jest włożona pomiędzy parę *IHDR-IEND*. Każdy kawałek składając się z długości, typu i danych, a nad jego integralnością sprawuje pieczę sumą kontrolną *CRC*. Wszystko to tworzy zwarty format.

Identyfikator typu określa rolę w jakiej przyjdzie służyć kawałkowi.

Przykładowa struktura prostego pliku PNG

Structure of a very simple PNG file			
PNG signature	IHDR	IDAT	IEND
Contents of a minimal PNG file representing just one red pixel			
			As characters
89 50 4E 47 0D 0A 1A 0A 00 00 00 01 00 00 00 01 08 02 00 00 00 90 77 53 DE 00 00 00 0C 49 44 41 54 08 D7 63 F8 CF C0 00 00 03 01 01 00 18 DD 8D B0 00 00 00 00 49 45 4E 44 AE 42 60 82			.PNG.....IHDRwSIDAT..c....IEN D.B`.

Rysunek 1.1: Przykładowa struktura pliku PNG [źródło: wikipedia.org]

1.1.2 Instrukcja obsługi programu

W celu uruchomienia kodu źródłowego będzie niezbędny:

- Python (≥ 2.7 , 32 bit)
- NumPy ($\geq 1.16.3$)
- matplotlib ($\geq 2.2.5$)
- Pillow ($\geq 6.0.0$)

Uruchomienie programu sprowadza się do użycia komendy `python source.py` w folderze `./Exercises/Source` co spowoduje wygenerowanie się odpowiednich obrazów w folderze `./Exercises/Source/Resources/result`.

Rozdział 2

Operacje ujednolicania obrazów

Ujednolicanie obrazów oznacza sprowadzenie ich do wspólnego gruntu pod względem określonego parametru. W tym wypadku będziemy ujednolić obrazy pod względem geometrycznym (ilości kolumn i wierszy pikseli) i następnie rozdzielczościowym (wypełnienia pikselami). Sekwencyjność tych operacji jak i one same nie są w stanie spowodować spadku jakości obrazu.

2.1 Ujednolicenie obrazów szarych geometryczne

Algorytm

Opis

Algorytm geometrycznego ujednolicenia obrazów ma za zadanie sprowadzić oba obrazy do tej samej liczby pikseli w każdym wierszu i każdej kolumnie.

Kroki

1. Porównaj szerokości i wysokości obu obrazów i wybierz największe.
2. Jeśli pierwszy lub drugi obraz mają szerokość lub wysokość mniejszą od największej dostępnej to:
 - (a) Utwórz czarne tło
 - (b) Przenieś z wyśrodkowaniem piksle na czarne tło
3. Jeśli żaden z warunków jest niespełniony to nie rób nic

Rysunek 2.1: Przed uruchomieniem algorytmu (od lewej): obraz 1 (712x712), obraz 2 (1423x1423)



Rysunek 2.2: Po uruchomieniu algorytmu (od lewej): obraz 1 (1423x1423), obraz 2 (1423x1423)



Rysunek 2.3: Przed uruchomieniem algorytmu (od lewej): obraz 1 (567x567), obraz 2 (712x712)



Rysunek 2.4: Po uruchomieniu algorytmu (od lewej): obraz 1 (712x712), obraz 2 (712x712)



Kod źródłowy algorytmu

```

def geometricGray(self):
    print('Geometric gray unification for image {} and {}'.format(self.
                                                                    firstDecoder.name, self.
                                                                    secondDecoder.name))
    firstResult, secondResult = self.grayUnification()
    ImageHelper.Save(firstResult.astype(numpy.uint8), self.imageType, ,
                     'geometric-gray', False, self.
                     firstDecoder, self.secondDecoder
                     )
    ImageHelper.Save(secondResult.astype(numpy.uint8), self.imageType, ,
                     'geometric-gray', False, self.
                     secondDecoder, self.firstDecoder
                     )

def grayUnification(self):
    firstResult = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
    width, height = self.firstDecoder.width, self.firstDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        # Copy smaller image to bigger
        startWidthIndex = int(round((self.maxWidth - width) / 2))
        startHeightIndex = int(round((self.maxHeight - height) / 2))
        pixelsBuffer = self.firstDecoder.getPixels()
        for h in range(0, height):
            for w in range(0, width):
                firstResult[h + startHeightIndex, w + startWidthIndex] =
                    pixelsBuffer[h, w]
    else:
        firstResult = self.firstDecoder.getPixels()

    secondResult = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
    width, height = self.secondDecoder.width, self.secondDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        # Copy smaller image to bigger
        startWidthIndex = int(round((self.maxWidth - width) / 2))
        startHeightIndex = int(round((self.maxHeight - height) / 2))
        pixelsBuffer = self.secondDecoder.getPixels()
        for h in range(0, height):
            for w in range(0, width):

```

```
secondResult[h + startHeightIndex, w + startWidthIndex] =  
    pixelsBuffer[h, w]  
else:  
    secondResult = self.secondDecoder.getPixels()  
  
return firstResult, secondResult
```

2.2 Ujednolicenie obrazów szarych rozdzielczościowe

Algorytm

Opis

Po użyciu ujednolicenia geometrycznego można użyć ujednolicenia rozdzielczościowego, które przeskaliuje obraz z mniejszej postaci do większej dzięki czemu nie zostanie nam czarna ramka wokół obrazu. Wynikiem będzie większy obraz niż początkowo bez czarnego obwodu wokół. Mniejszy obraz można przeskalać do większych wymiarów przenosząc wszystkie piksele z uwzględnieniem luk pomiędzy nimi i następnie użycia interpolacji do zamazania tych luk. Interpolacja działa na zasadzie pobierania wartości z okolicznych pikseli i wyciągania z nich średniej, która posłuży jako baza koloru dla nowego piksela.

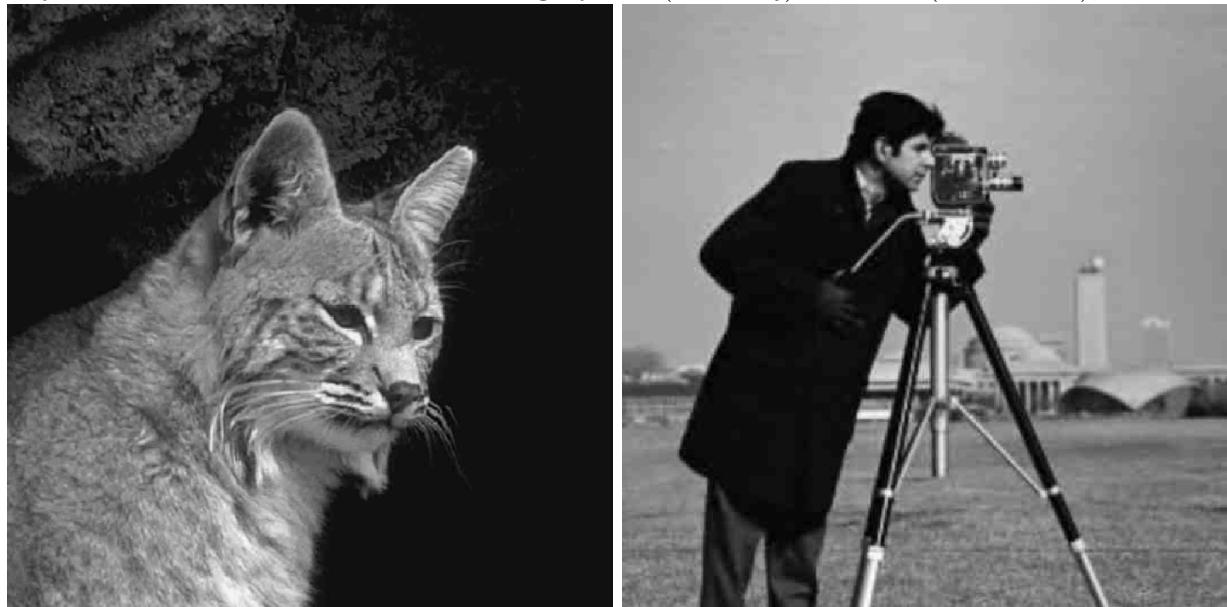
Kroki

1. Ustalenie nowych wymiarów obrazu
2. Obliczenie odległości pomiędzy pikselami ($scaleFactoryH$, $scaleFactoryW$)
3. Naniesienie pikseli z mniejszego obrazu na większy z uwzględnieniem luk
4. Interpolacja

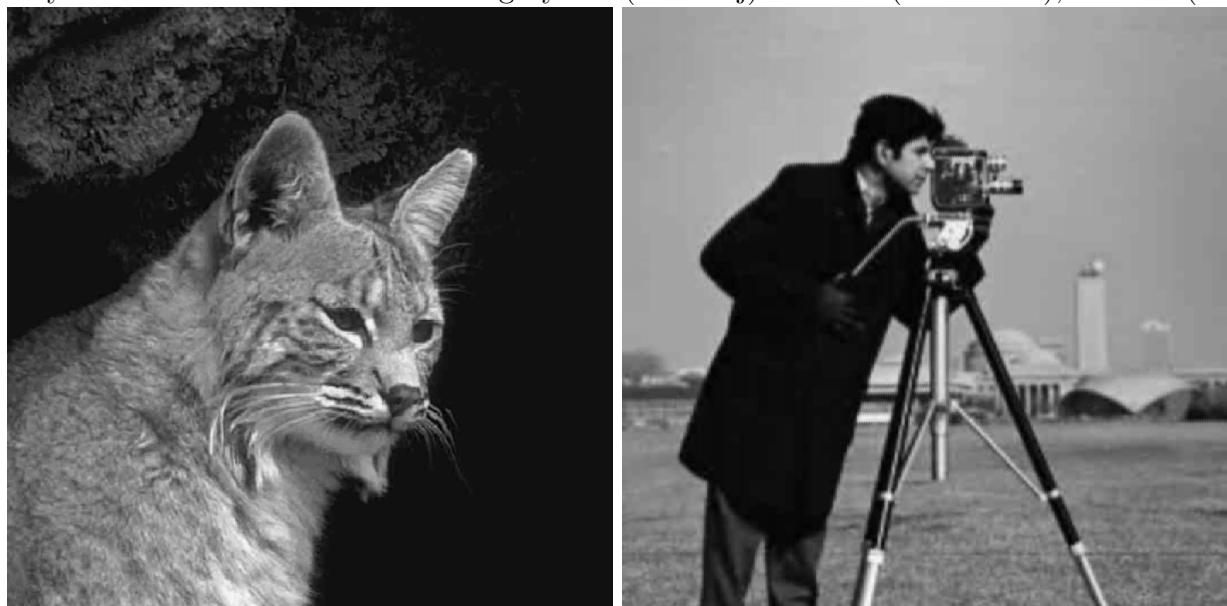
Rysunek 2.5: Skutki braku interpolacji



Rysunek 2.6: Przed uruchomieniem algorytmu (od lewej): obraz 1 (1423x1423), obraz 2 (712x712)



Rysunek 2.7: Po uruchomieniu algorytmu (od lewej): obraz 1 (1423x1423), obraz 2 (1423x1423)



Rysunek 2.8: Przed uruchomieniem algorytmu (od lewej): obraz 1 (567x567), obraz 2 (712x712)



Rysunek 2.9: Po uruchomieniu algorytmu (od lewej): obraz 1 (712x712), obraz 2 (712x712)



Kod źródłowy algorytmu

```

def rasterGray(self):
    print('Raster gray unification for image {} and {}'.format(self.
                                                               firstDecoder.name, self.
                                                               secondDecoder.name))
    firstResult = self.scaleUpGray(self.firstDecoder)
    secondResult = self.scaleUpGray(self.secondDecoder)
    ImageHelper.Save(firstResult.astype(numpy.uint8), self.imageType, 'raster
                           -gray', False, self.firstDecoder
                           , self.secondDecoder)
    ImageHelper.Save(secondResult.astype(numpy.uint8), self.imageType, 'raster-gray', False, self.
                           secondDecoder, self.firstDecoder)

```

```

def scaleUpGray(self, decoder):

```

```

width, height = decoder.width, decoder.height
scaleFactoryW = float(self.maxWidth) / width
scaleFactoryH = float(self.maxHeight) / height
if width < self.maxWidth or height < self.maxHeight:
    pixelsBuffer = decoder.getPixels()
    result = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
    # Fill values
    for h in range(height - 1):
        for w in range(width - 1):
            if w%2 == 0:
                result[int(round(scaleFactoryH * h)), int(round(scaleFactoryW *
                                                               w)) + 1] =
                pixelsBuffer[h, w]
            if w%2 == 1:
                result[int(round(scaleFactoryH * h)) + 1, int(
                                                               scaleFactoryW *
                                                               w)] =
                pixelsBuffer[h, w]
    # Interpolate
    self._interpolateGray(result)
    return result
else:
    return decoder.getPixels()

def _interpolateGray(self, result):
    for h in range(self.maxHeight):
        for w in range(self.maxWidth):
            value = 0
            count = 0
            if result[h, w] == 0:
                for hOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        hSafe = h if ((h + hOff) > (self.maxHeight - 2)) | ((h +
                                                               hOff) < 0) else (h +
                                                               hOff)
                        wSafe = w if ((w + wOff) > (self.maxWidth - 2)) | ((w +
                                                               wOff) < 0) else (w +
                                                               wOff)
                        if result[hSafe, wSafe] != 0:
                            value += result[hSafe, wSafe]
                            count += 1
            result[h, w] = value / count

```

2.3 Ujednolicenie obrazów RGB geometryczne

Algorytm

Opis

Algorytm geometrycznego ujednolicenia obrazów ma za zadanie sprowadzić oba obrazy do tej samej liczby pikseli w każdym wierszu i każdej kolumnie. Różnica pomiędzy tym przypadkiem a szarym sprawia, że ważne jest użycie odpowiednich struktur danych w taki sposób aby każdy z kanałów RGB był w stanie się pomieścić. Niewątpliwie ważne jest struktura danych uwzględniała kolejność w jakim kolory są przechowywane, inaczej może dojść do sytuacji w której nie dostaniemy oczekiwanej rezultatu.

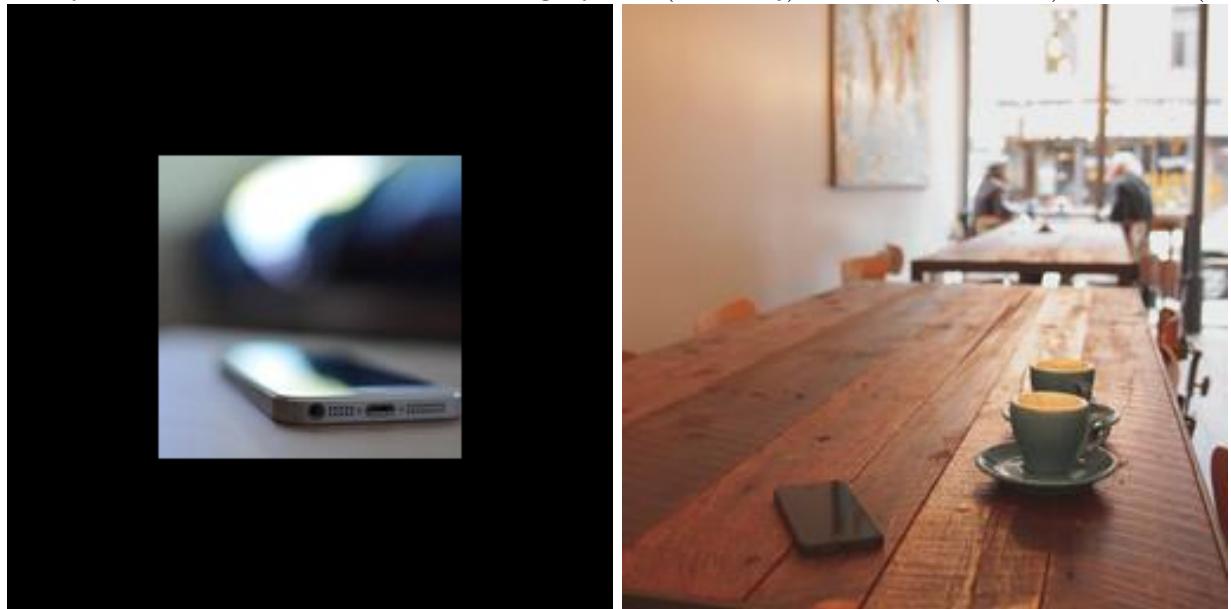
Kroki

1. Porównaj szerokości i wysokości obu obrazów i wybierz największe.
2. Jeśli pierwszy lub drugi obraz mają szerokość lub wysokość mniejszą od największej dostępnej to:
 - (a) Utwórz czarne tło
 - (b) Przenieś z wyśrodkowaniem piksle na czarne tło z uwzględnieniem każdego z kanałów RGB
3. Jeśli żaden z warunków jest niespełniony to nie rób nic

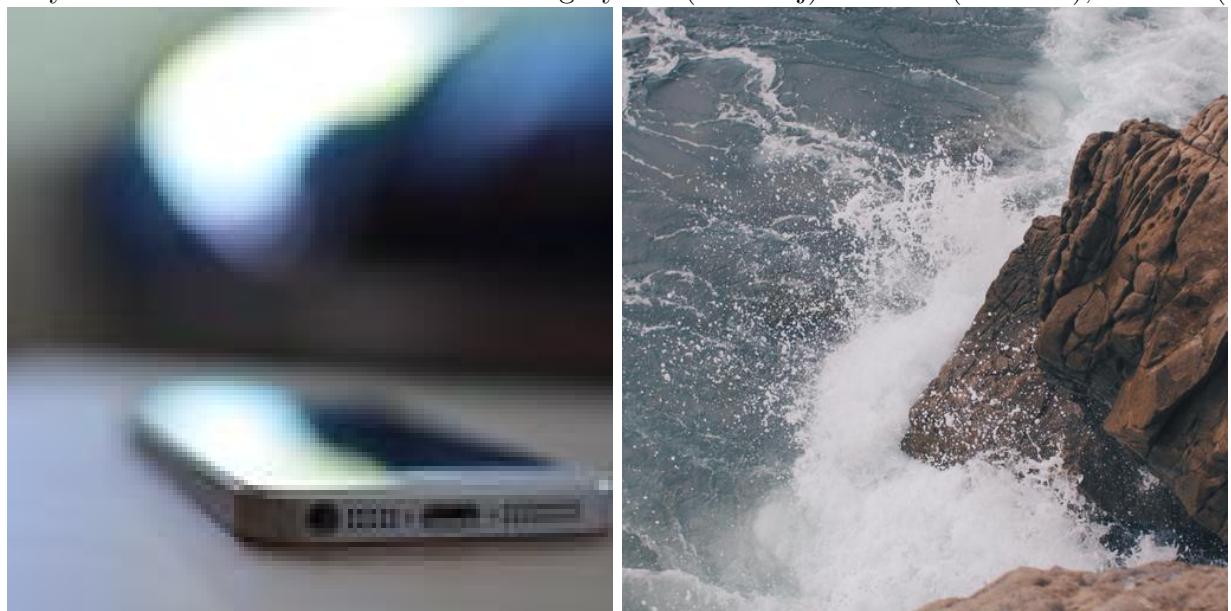
Rysunek 2.10: Przed uruchomieniem algorytmu (od lewej): obraz 1 (128x128), obraz 2 (256x256)



Rysunek 2.11: Po uruchomieniu algorytmu (od lewej): obraz 1 (256x256), obraz 2 (256x256)



Rysunek 2.12: Przed uruchomieniem algorytmu (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.13: Po uruchomieniu algorytmu (od lewej): obraz 3 (512x512), obraz 4 (512x512)



Kod źródłowy algorytmu

```

def geometricColor(self):
    print('Geometric color unification for image {} and {}'.format(self.
                                                                     firstDecoder.name, self.
                                                                     secondDecoder.name))
    firstResult, secondResult = self.colorUnification()
    ImageHelper.Save(firstResult.astype(numpy.uint8), self.imageType, ,
                     'geometric-color', False, self.
                     firstDecoder, self.secondDecoder
                     )
    ImageHelper.Save(secondResult.astype(numpy.uint8), self.imageType, ,
                     'geometric-color', False, self.
                     secondDecoder, self.firstDecoder
                     )

def colorUnification(self):
    width, height = self.firstDecoder.width, self.firstDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        firstResult = self._paintInMiddleColor(self.firstDecoder)
    else:
        firstResult = self.firstDecoder.getPixels()

    width, height = self.secondDecoder.width, self.secondDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        secondResult = self._paintInMiddleColor(self.secondDecoder)
    else:
        secondResult = self.secondDecoder.getPixels()

    return firstResult, secondResult

def _paintInMiddleColor(self, decoder):
    result = numpy.full((self.maxHeight, self.maxWidth, 3), 0, numpy.uint8)
    # Copy smaller image to bigger
    width, height = decoder.width, decoder.height
    startWidthIndex = int(round((self.maxWidth - width) / 2))
    startHeightIndex = int(round((self.maxHeight - height) / 2))
    pixelsBuffer = decoder.getPixels()
    for h in range(0, height):
        for w in range(0, width):
            result[h + startHeightIndex][w + startWidthIndex] = pixelsBuffer[h][w]
    return result

```


2.4 Ujednolicenie obrazów RGB rozdzielczościowe

Algorytm

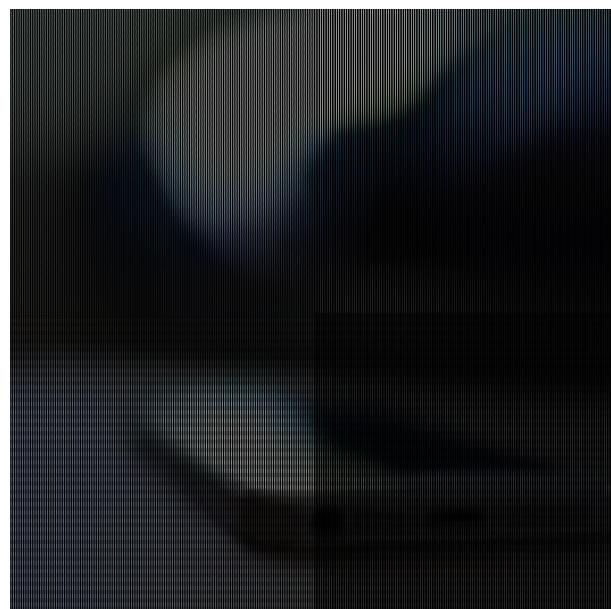
Opis

Po użyciu ujednolicenia geometrycznego można użyć ujednolicenia rozdzielczościowego, które przeskaliuje obraz z mniejszej postaci do większej dzięki czemu nie zostanie nam czarna ramka wokół obrazu. Wynikiem będzie większy obraz niż początkowo bez czarnego obwodu wokół. Mniejszy obraz można przeskalać do większych wymiarów przenosząc wszystkie piksele z uwzględnieniem luk pomiędzy nimi i następnie użycia interpolacji do zamazania tych luk. Interpolacja działa na zasadzie pobierania wartości z okolicznych pikseli i wyciągania z nich średniej, która posłuży jako baza koloru dla nowego piksela.

Kroki

1. Ustalenie nowych wymiarów obrazu
2. Obliczenie odległości pomiędzy pikselami ($scaleFactoryH$, $scaleFactoryW$)
3. Nanieśenie pikseli z mniejszego obrazu na większy z uwzględnieniem luk
4. Interpolacja

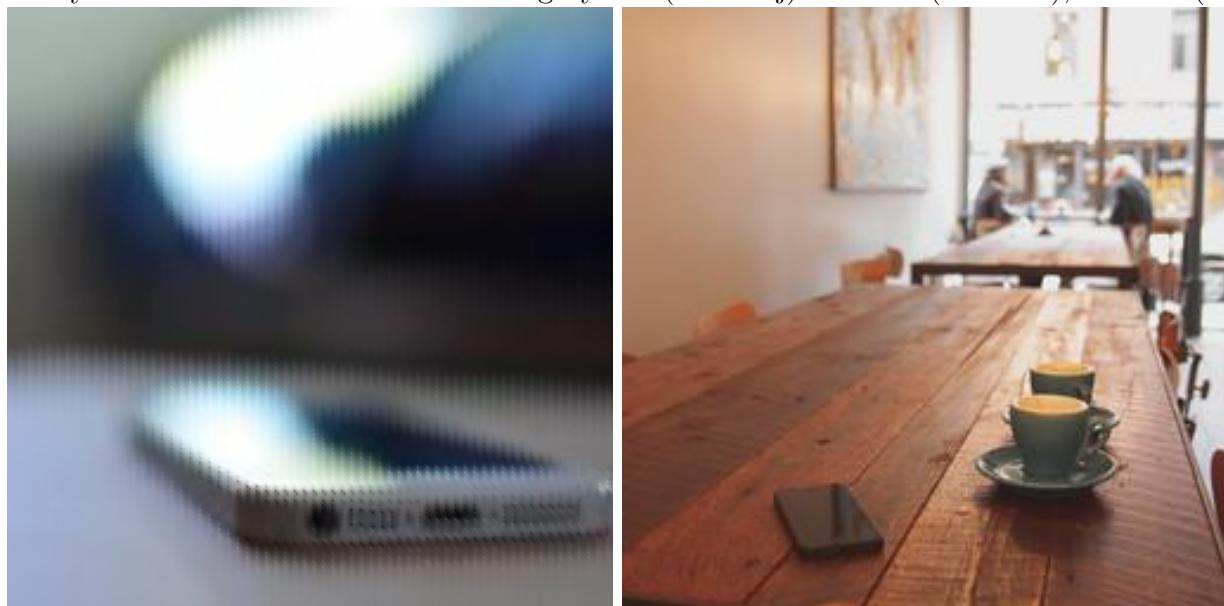
Rysunek 2.14: Skutki braku interpolacji



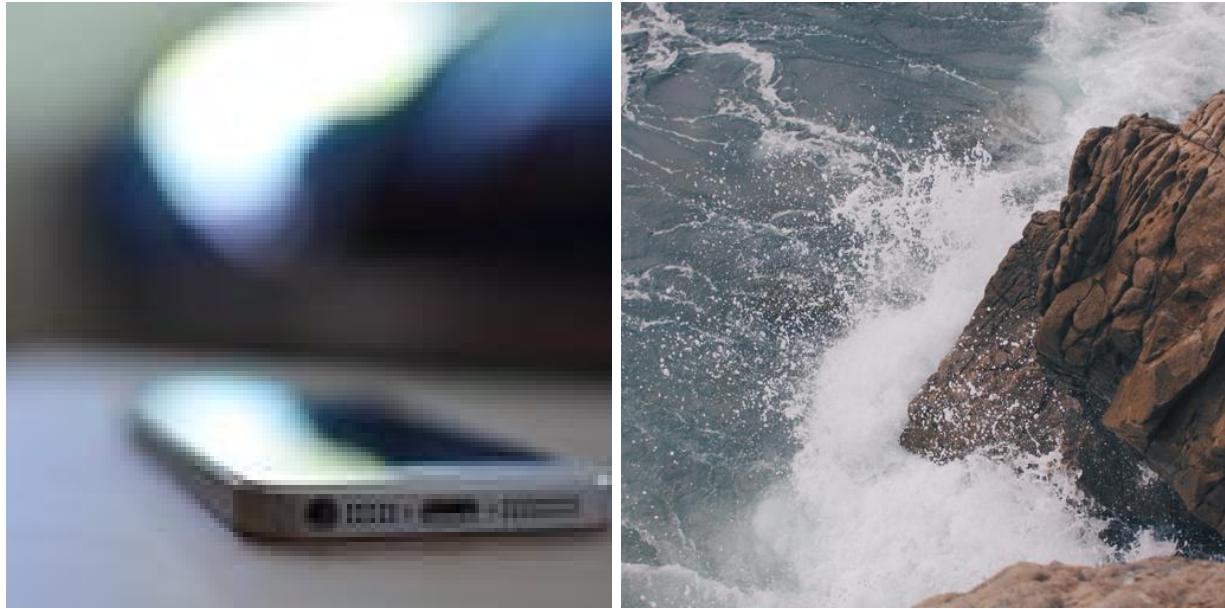
Rysunek 2.15: Przed uruchomieniem algorytmu (od lewej): obraz 1 (128x128), obraz 2 (256x256)



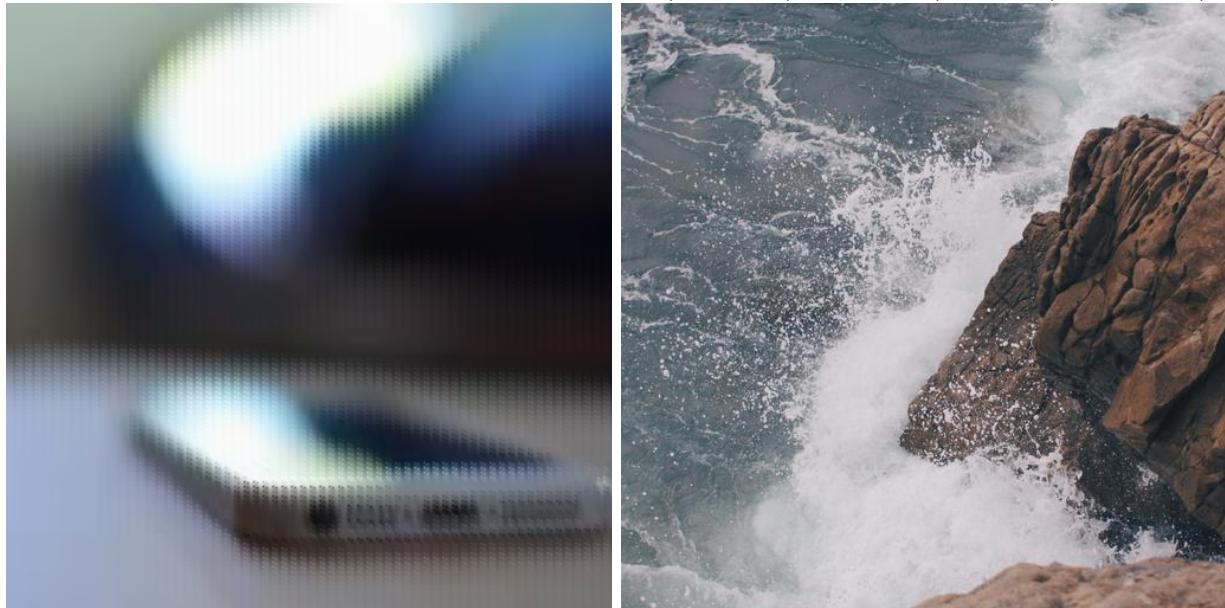
Rysunek 2.16: Po uruchomieniu algorytmu (od lewej): obraz 1 (256x256), obraz 2 (256x256)



Rysunek 2.17: Przed uruchomieniem algorytmu (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.18: Po uruchomieniu algorytmu (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Kod źródłowy algorytmu

```
def rasterColor(self):
    print('Raster color unification for image {} and {}'.format(self.
                                                               firstDecoder.name, self.
                                                               secondDecoder.name))
    firstResult = self.scaleUpColor(self.firstDecoder)
    secondResult = self.scaleUpColor(self.secondDecoder)
    ImageHelper.Save(firstResult.astype(numpy.uint8), self.imageType, 'raster
                           -color', False, self.
                           firstDecoder, self.secondDecoder
                           )
    ImageHelper.Save(secondResult.astype(numpy.uint8), self.imageType, 'raster-color', False, self.
                           secondDecoder, self.firstDecoder
                           )
```

```

def scaleUpColor(self, decoder):
    width, height = decoder.width, decoder.height
    scaleFactoryW = float(self.maxWidth) / width
    scaleFactoryH = float(self.maxHeight) / height
    if width < self.maxWidth or height < self.maxHeight:
        pixelsBuffer = decoder.getPixels()
        result = numpy.full((self.maxHeight, self.maxWidth, 3), 1, numpy.
                             uint8)
        # Fill values
        for h in range(height):
            for w in range(width):
                if w%2 == 0:
                    result[int(scaleFactoryH * h), int(round(scaleFactoryW *
                        w)) + 1] =
                        pixelsBuffer[h, w]
                if w%2 == 1:
                    result[int(round(scaleFactoryH * h)) + 1, int(
                        scaleFactoryW *
                        w)] =
                        pixelsBuffer[h, w]
        # Interpolate
        self._interpolateColor(result)
        return result;
    else:
        return decoder.getPixels()

def _interpolateColor(self, result):
    for h in range(self.maxHeight):
        for w in range(self.maxWidth):
            r, g, b = 0, 0, 0
            n = 0
            if (result[h, w][0] == 1) & (result[h, w][1] == 1) & (result[h, w
                ][2] == 1):
                for hOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        hSafe = h if ((h + hOff) > (self.maxHeight - 2)) | ((h
                            + hOff) < 0) else (h +
                            hOff)
                        wSafe = w if ((w + wOff) > (self.maxWidth - 2)) | ((w
                            + wOff) < 0) else (w +
                            wOff)
                        if (result[hSafe, wSafe][0] > 1) | (result[hSafe,
                            wSafe][1] >
                            1) | (result
                            [hSafe,
                            wSafe][2] >
                            1):
                            r += result[hSafe, wSafe][0]
                            g += result[hSafe, wSafe][1]
                            b += result[hSafe, wSafe][2]
                            n += 1
            result[h, w] = (r/n, g/n, b/n)

```


Rozdział 3

Operacje sumowania arytmetycznego obrazów szarych

Obraz jest macierzą wartości co pozwala nam wykonywać na nich operacje arytmetyczne tak samo jak na zwykłych macierzach. Operacje takie jak:

- * dodawanie,
- * odejmowanie,
- * mnożenie (wyjątkowo wykonywane inaczej niż w przypadku dwóch macierzy),
- * dzielenie

obrazów może odbywać się w różnych kombinacjach rodzajów wartości:

- * obraz z obrazem
- * obraz ze stałą

Operacje te odbywają się na poziomie komórek macierzy i są też nazywane **operatorami punktowymi**. Co oznacza, że przy przetwarzaniu dwóch obrazów liczą się tylko wartości znajdujące się na tej samej pozycji wysokości i szerokości w obrazie $P_1(i, j)$ i $P_2(i, j)$.

Po przeprowadzeniu niektórych operacji algebraicznych należy przeprowadzić normalizację w celu zmiany zakresu wartości (\min, \max) na ($\text{newMin}, \text{newMax}$). Do przeprowadzenia normalizacji w poniższych przykładach będziemy używać wzoru:

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

Gdzie Z_{rep} oznacza maksymalną wartość dla naszej struktury piksela. Dla obrazów szarych może przybrać wartość `#FF` (system szesnastkowy) lub dla wersji kolorowej `#FFFFFF`. W celu uniknięcia powtórzeń w kodzie źródłowym wydzieliłem funkcję normalizacji, która dla obrazów szarych prezentuje się następująco:

```
def Normalization(image, result):  
    maxValue = numpy.iinfo(image.dtype).max  
    fmin = numpy.amin(result)  
    fmax = numpy.amax(result)  
    result = result.astype(numpy.float32)  
    result = maxValue * ((result - fmin) / (fmax - fmin))  
    result = result.astype(numpy.uint8)  
    return result
```

3.1 Sumowanie określonej stałej z obrazem

Algorytm

Opis

W operacji sumowania obrazów szarych ze stałą ważne jest doprowadzenie do stanu w którym będziemy mogli dodać wartość nie martwiąc się o przepełnienie zmiennej co mogłoby spowodować zniekształcenie obrazu.

Aby tego uniknąć przeskalujemy wszystkie wartości macierzy obrazu tak, aby suma stałej i największej wartości macierzy nie przekroczyła maksymalnej wartości dla zmiennej.

Rysunek 3.1: Przed uruchomieniem algorytmu (lewy obraz), po dodaniu wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.2: Przed uruchomieniem algorytmu (lewy obraz), po dodaniu wartości 300 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.3: Przed uruchomieniem algorytmu (lewy obraz), po dodaniu wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.4: Przed uruchomieniem algorytmu (lewy obraz), po dodaniu wartości 300 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

def sumWithConst(self, constValue):
    print('Sum gray image {} with const {}'.format(self.firstDecoder.name,
                                                    constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxSum = float(numpy.amax(numpy.add(image.astype(numpy.uint32),
                                         constValue)))
    maxValue = float(numpy.iinfo(image.dtype).max)
    scaleFactor = (maxSum - maxValue) / maxValue if maxSum > maxValue else 0

    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            pom = (image[h, w] - (image[h, w] * scaleFactor)) + (constValue -
                                                               (constValue *
                                                               scaleFactor))
            result[h, w] = numpy.ceil(pom)

    ImageHelper.Save(result, self.imageType, 'sum-gray-const', False, self.
                     firstDecoder, None, constValue)
    result = Commons.Normalization(image, result)
    ImageHelper.Save(result, self.imageType, 'sum-gray-const', True, self.
                     firstDecoder, None, constValue)

```

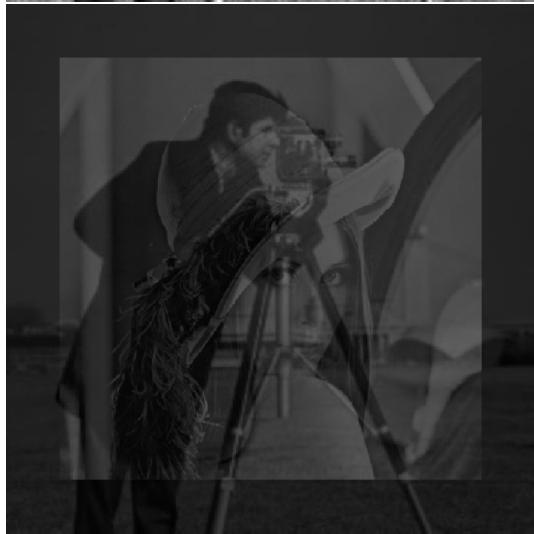
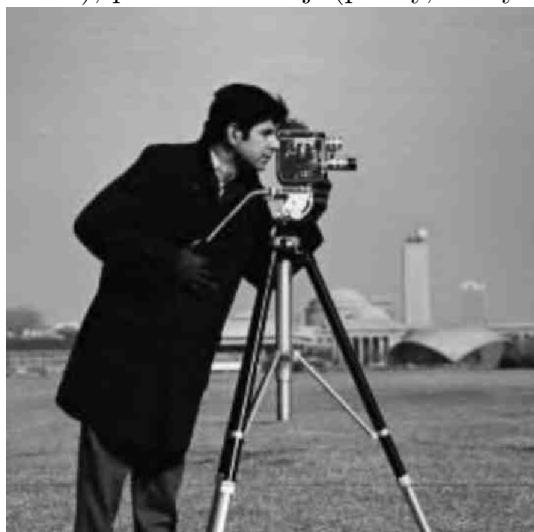
3.2 Sumowanie dwóch obrazów

Algorytm

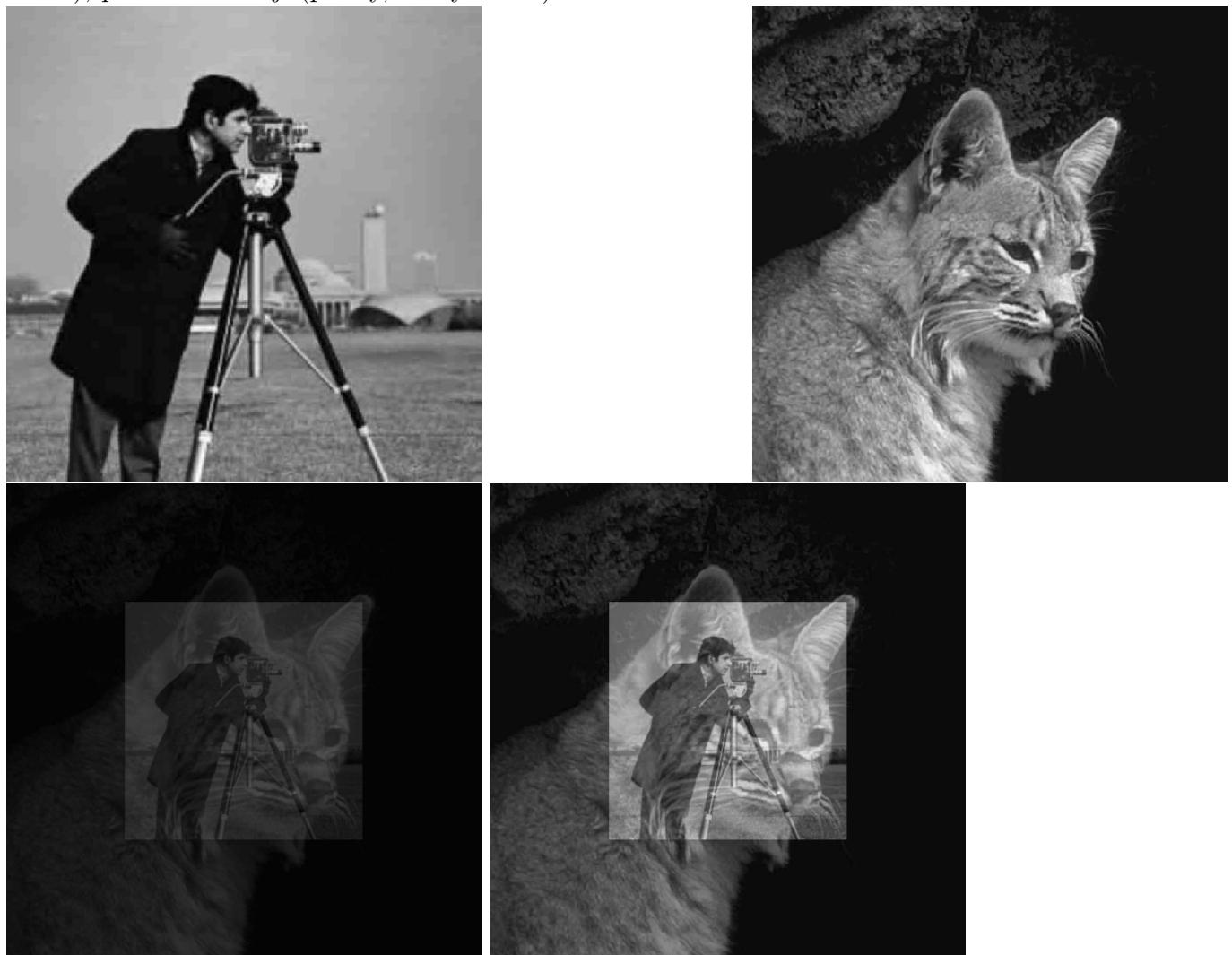
Opis

Dodanie dwóch obrazów jest analogiczne jak przy dodaniu stałej do obrazu, ale z tą różnicą, że maksymalnej wartości mogącej spowodować przepełnienie szukamy we wstępny obrazie wynikowym sumy obu obrazów.

Rysunek 3.5: Przed uruchomieniem algorytmu (obrazy na górze), po dodaniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 3.6: Przed uruchomieniem algorytmu (obrazy na górze), po dodaniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Kod źródłowy programu

```

def sumImages(self):
    print('Sum gray image {} with image {}'.format(self.firstDecoder.name,
                                                    self.secondDecoder.name))
    unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
                               'L')
    firstImage, secondImage = unification.grayUnification()
    width, height = firstImage.shape[0], firstImage.shape[1]

    maxSum = float(
        numpy.amax(
            numpy.add(firstImage.astype(numpy.uint32),
                      secondImage.astype(numpy.uint32))))
    maxValue = float(numpy.iinfo(firstImage.dtype).max)
    scaleFactor = (maxSum - maxValue) / maxValue if maxSum > maxValue else 0

    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            pom = (firstImage[h, w] - (firstImage[h, w] * scaleFactor)) + (
                secondImage[h, w] - (secondImage[h, w] * scaleFactor))
            result[h, w] = numpy.ceil(pom)

```

```

ImageHelper.Save(result, self.imageType, 'sum-gray-images', False, self.
                  firstDecoder, self.secondDecoder
)
result = Commons.Normalization(firstImage, result)
ImageHelper.Save(result, self.imageType, 'sum-gray-images', True, self.
                  firstDecoder, self.secondDecoder
)

```

3.3 Mnożenie obrazu przez zadaną liczbę

Algorytm

Opis

Mnożenie obrazu przychodzi w dwóch formach. Jednym z nich jest wykonanie tej operacji z użyciem stałej jako mnoźnika. Na wykonanie tej czynności składa się mnożenie każdego z pikseli przez stałą, w ten sam sposób jak przy zwykłych macierzach:

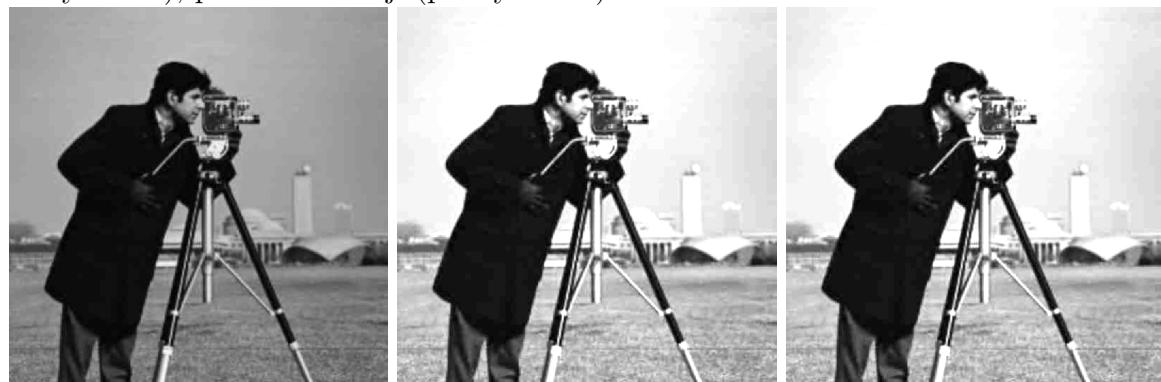
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * X = \begin{bmatrix} a_{11} * X & a_{12} * X & a_{13} * X \\ a_{21} * X & a_{22} * X & a_{23} * X \\ a_{31} * X & a_{32} * X & a_{33} * X \end{bmatrix}$$

Mnożeniem obrazów szarych można nazwać też ich *skalowaniem* i dla wartości $\{x : 1 < x\}$ rozjaśnić go. Jednym z zagrożeń przy tej operacji jest przepełnienie, które objawia się czarnymi plamami (Rysunek 3.7). Aby go uniknąć dla wartości powyżej Z_{rep} (tzn. maksymalna wielkość zakresu reprezentacji tonalnej obrazu) program przytnie wartości do Z_{rep} co będzie skutkowało coraz jaśniejszymi obrazami, a uniemożliwi czarne plamy.

Rysunek 3.7: Przykład przekroczenia maksymalnej wartości



Rysunek 3.8: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 1.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.9: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 3.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.10: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 1.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.11: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 3.5 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

def multiplyWithConst(self, constValue):
    print('Multiply gray image {} with const {}'.format(self.firstDecoder.
                                                          name, constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = numpy.iinfo(image.dtype).max
    result = numpy.ones((height, width), numpy.uint8)

    for h in range(height):
        for w in range(width):
            result[h][w] = image[h][w] * constValue

```

```
for w in range(width):
    pom = image[h, w] * constValue
    result[h, w] = pom if pom <= maxValue else maxValue

ImageHelper.Save(result, self.imageType, 'multiply-gray-const', False,
                  self.firstDecoder, None,
                  constValue)
result = Commons.Normalization(image, result)
ImageHelper.Save(result, self.imageType, 'multiply-gray-const', True,
                  self.firstDecoder, None,
                  constValue)
```

3.4 Mnożenie obrazu przez inny obraz

Algorytm

Opis

Drugą formą mnożenia obrazu jest sytuacja gdy mnożnikiem jest inny obraz. Operacja ta przebiega analogicznie jak przy stałej, ale z tą różnicą że każdy piksel mnożymy przez odpowiadający mu piksel w drugim obrazie, zgodnie ze wzorem:

$$Q(i, j) = P_1(i, j) \times P_2(i, j) \quad (3.1)$$

W zależności od tego jakich wartości w obrazach użyjemy otrzymamy różne rezultaty:

1. Dla dwóch obrazów, które wartości posiadają z przedziału $(0, Z_{rep})$ obrazy będą się na siebie nakładać (Rysunek 3.12 i 3.13).
2. Dla obrazów w których jeden z nich jest używany jako maska otrzymamy wyciętą zawartość drugiego obrazu (Rysunek 3.14).

Przydatną właściwością tej operacji jest możliwość wyciągania części wspólnych z obrazów, dzięki czemu wykrywanie ruch na obrazach po-klatkowych jest łatwiejsze.

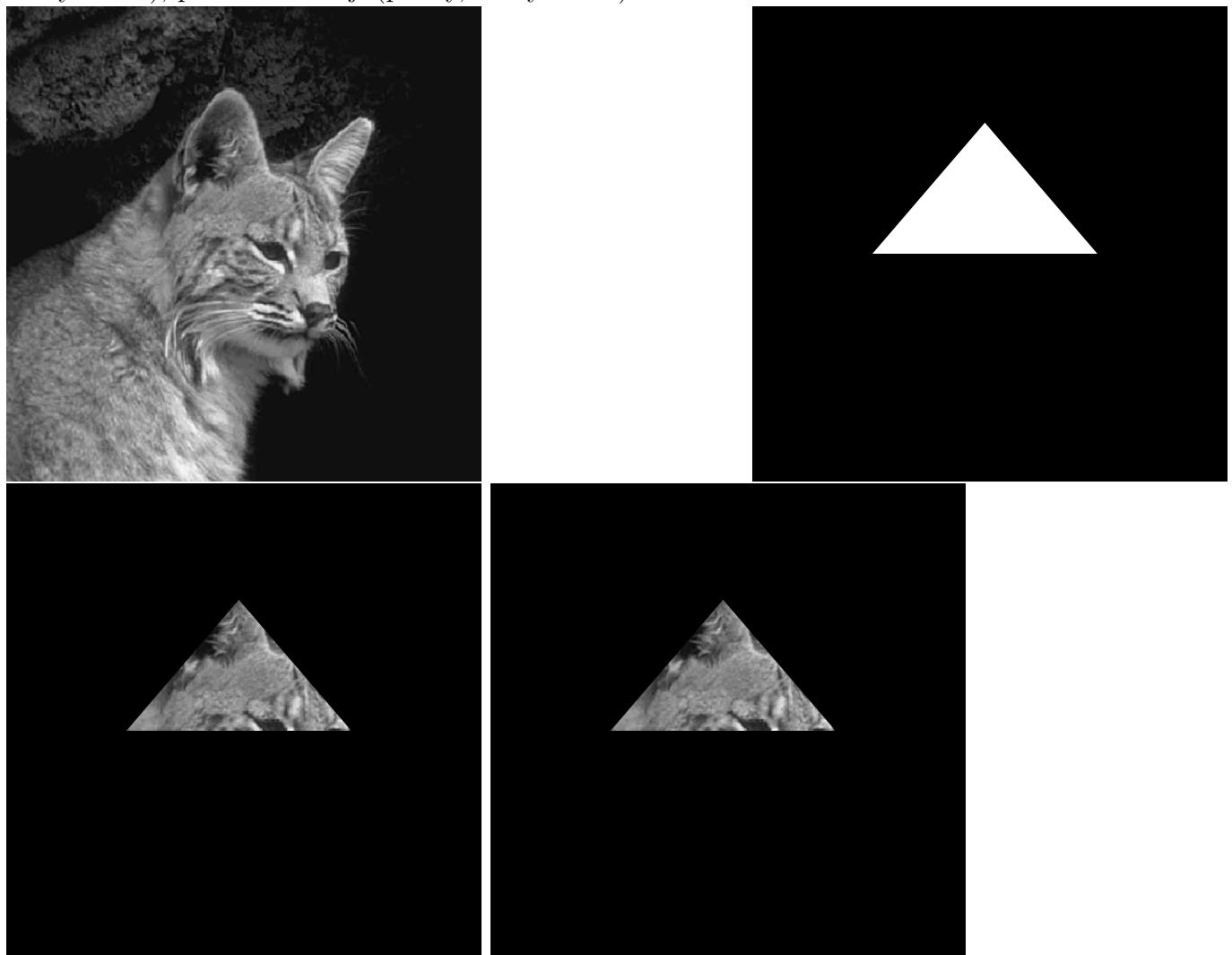
Rysunek 3.12: Przed uruchomieniem algorytmu (obrazy na górze), po przemnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 3.13: Przed uruchomieniem algorytmu (obrazy na górze), po przemnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 3.14: Przed uruchomieniem algorytmu (obrazy na górze), po przemnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Kod źródłowy programu

```

def multiplyImages(self):
    print('Multiply gray image {} with image {}'.format(self.firstDecoder.
        name, self.secondDecoder.name))
    unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
        , 'L')
    firstImage, secondImage = unification.grayUnification()
    width, height = firstImage.shape[0], firstImage.shape[1]

    maxValue = float(numpy.iinfo(firstImage.dtype).max)
    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            pom = int(firstImage[h, w]) * int(secondImage[h, w]) / maxValue
            result[h, w] = pom

    ImageHelper.Save(result, self.imageType, 'multiply-gray-images', False,
        self.firstDecoder, self.
        secondDecoder)
    result = Commons.Normalization(firstImage, result)
    ImageHelper.Save(result, self.imageType, 'multiply-gray-images', True,
        self.firstDecoder, self.
        secondDecoder)

```

3.5 Mieszanie obrazów z określonym współczynnikiem

Algorytm

Opis

Operator liniowego mieszania przyjmuje dwa obrazy tych samych rozmiarów, co wymaga skalowania geometrycznego, i zwraca liniową kombinację odpowiadających sobie pikseli (tak samo jak w przypadku dodawania). Wynikiem jest efekt przechodzenia obrazu f w obraz f' , a jego natężenie ustala się z pomocą specjalnego współczynnika.

Współczynnik α jest wartością określanaą przez użytkownika z zakresu $[0, 1]$. Jego zadaniem jest skalowanie wartości każdego piksela w obu obrazach przed ich połączeniem, dzięki czemu raz mogą być faworyzowane wartości z f' i raz z f .

Do obliczeń operatora będzie używany wzór:

$$f_{result}(i, j) = \alpha \times f(i, j) + (1 - \alpha) \times f'(i, j) \quad (3.2)$$

Wadą mieszania względem dodawania obrazów jest sam proces skalowania wartości pikseli, który sprawia, że przy wybraniu zbyt małej wartości α kontrast między pikselami jest zatracany. Problem występuje głównie, gdy kontrast na samych obrazach jest dość ubogi. Aby podtrzymać kontrast możemy zwiększyć współczynnik α lub zdefiniować odpowiednią maskę dla naszego obrazu. Maska f_m jest rozmiaru f_{result} i składa się z wartości z zakresu $[0, Z_p]$. Każda wartość takiej maski odpowiada współczynnikowi α dla każdego piksela w f i f' . Wzór wygląda wtedy tak:

$$f_{result}(i, j) = \frac{f_m(i, j)}{Z_p} \times f(i, j) + \left(1 - \frac{f_m(i, j)}{Z_p}\right) \times f'(i, j) \quad (3.3)$$

Rysunek 3.15: Przed uruchomieniem algorytmu



Rysunek 3.16: Po mieszaniu o wartość 0.2 (lewy obraz), po mieszaniu o wartość 0.5 (środkowy obraz), po mieszaniu o wartość 0.8 (prawy obraz)



Kod źródłowy programu

```
def blendImages(self, ratio):
    print('Blending gray image {} with image {} and ratio {}'.format(self.
        firstDecoder.name, self.
        secondDecoder.name, ratio))

    if ratio < 0 or ratio > 1.0:
        raise ValueError('ratio is wrong')

    unification = Unification(self.firstDecoder.name, self.secondDecoder.name
        , 'L')
    firstImage = unification.scaleUpGray(self.firstDecoder)
    secondImage = unification.scaleUpGray(self.secondDecoder)
    width, height = firstImage.shape[0], firstImage.shape[1]

    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            pom = ratio * firstImage[h, w] + (1 - ratio) * secondImage[h, w]
            result[h, w] = pom

    ImageHelper.Save(result, self.imageType, 'blend-gray-images', False,
        self.firstDecoder, None, ratio)
```

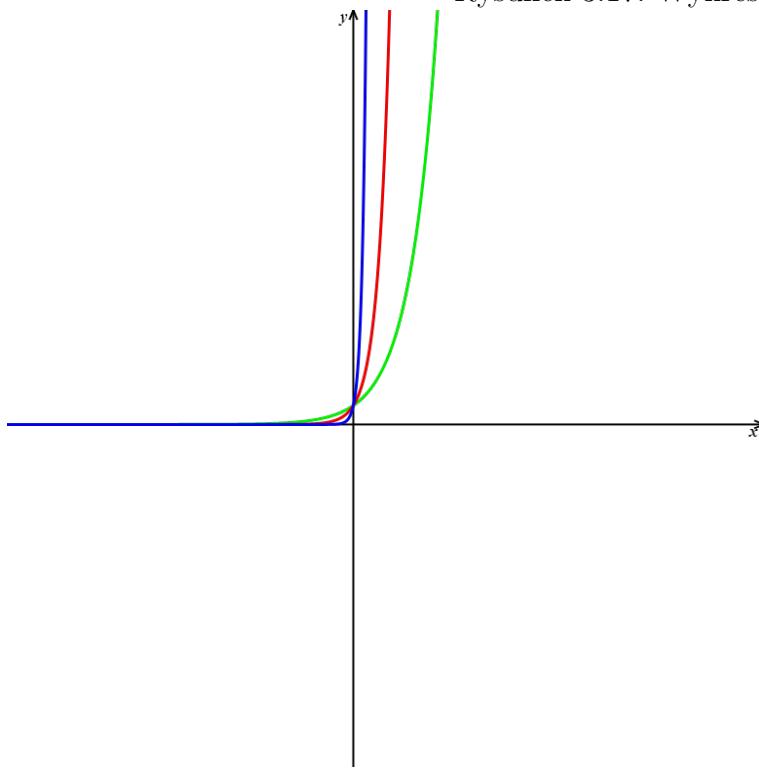
3.6 Potęgowanie obrazu z zadaną potęgą

Algorytm

Opis

Efektem operatora potęgowania jest zmniejszenie lub zwiększenie kontrastu pomiędzy pikselami. W przeciwieństwie do operacji dodawania, która zmniejszała lub zwiększała wartość wszystkich pikseli, potęgowanie wpływa na odległość pomiędzy wartościami na osi. Zachowanie wynika z własności funkcji wykładniczej (Rysunek 3.16) - im mniejsza wartość bazowa tym wolniejszy przyrost, a dla większych szybszy przyrost.

Rysunek 3.17: Wykres eksponenty



W zależności od dobrania odpowiedniej wartości *wykładnika* możemy:

1. zwiększyć kontrast dla małych jaskrawości; wybierając $\alpha \in (0, 1)$.

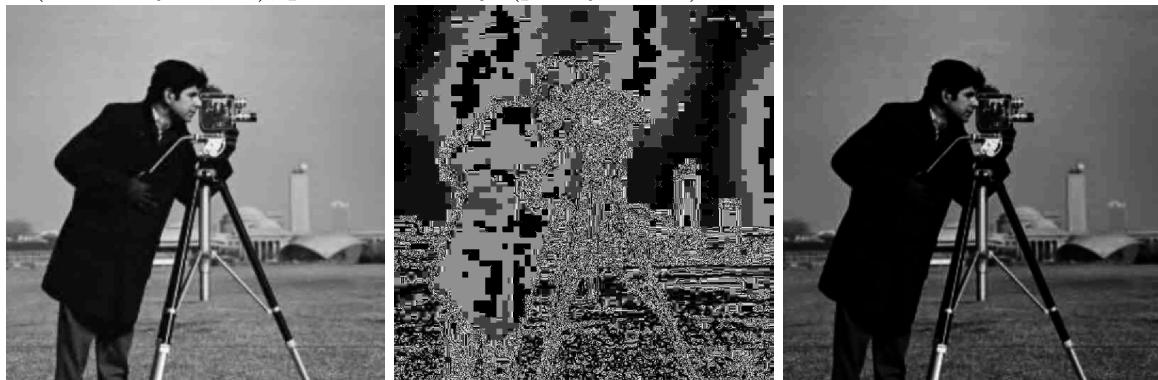
2. zwiększyć kontrast dla dużych jaskrawości; wybierając $\alpha \in (1, \infty)$.

Obliczając wartości pikseli użyjemy wzoru:

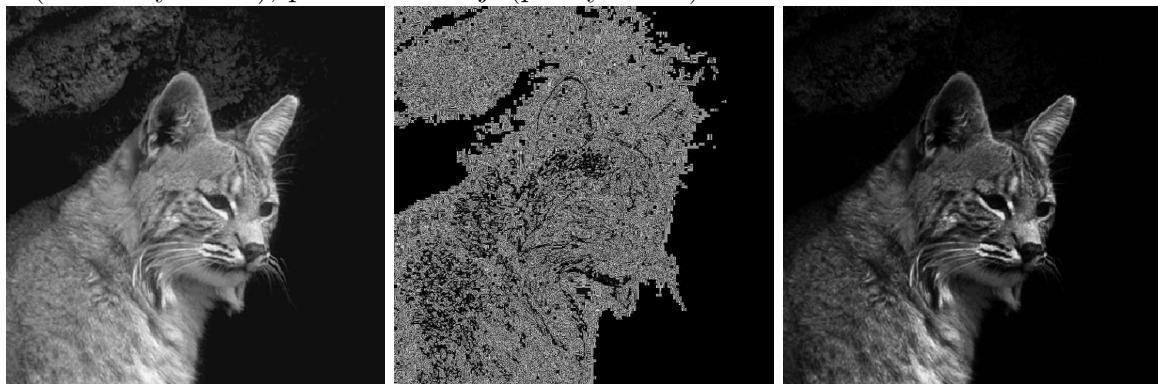
$$f_m = f^\alpha \quad (3.4)$$

Po wykonaniu tej operacji będzie potrzebne zastosowanie normalizacji do uwidocznienie wyników.

Rysunek 3.18: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.19: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

def powerFirstImage(self, powerIndex):
    print('Power gray image {} with index {}'.format(self.firstDecoder.name,
                                                       powerIndex))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxValue = float(numpy.iinfo(image.dtype).max)
    result = numpy.ones((height, width), numpy.uint32)
    for h in range(height):
        for w in range(width):
            result[h, w] = image[h, w]**powerIndex

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'power-gray',
                    False, self.firstDecoder, None,
                    powerIndex)
    result = Commons.Normalization(image, result)
    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'power-gray',
                    True, self.firstDecoder, None,
                    powerIndex)

```

3.7 Dzielenie obrazu przez liczbę

Wstęp

Operator dzielenia obrazu przez liczbę przyjmuje dwie wartości: macierz obrazu P_1 i stałą liczbową C i polega na podzieleniu każdego piksla w obrazie przez stałą. Wynikiem tej operacji będzie liczba zmiennoprzecinkowa, więc należy ją zaokrąglić do najbliższej wartości całkowitej.

$$Q(i, j) = P_1(i, j) \div C \quad (3.5)$$

Efektem tej operacji będzie przyciemnienie obrazu i zmniejszenie kontrastu pomiędzy pikselami. Z tego względu po operacji dzielenia warto zastosować *normalizację*, aby wartości stłoczone bliżej zera "rozciągnąć" na cały możliwy przedział wartości od 0 do Z_p .

Kod algorytmu

```
def divideWithConst(self, constValue):
    print('Divide gray image {} with const {}'.format(self.firstDecoder.name,
                                                       constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = numpy.iinfo(image.dtype).max
    result = numpy.ones((height, width), numpy.uint8)

    for h in range(height):
        for w in range(width):
            result[h, w] = image[h, w] / constValue

    ImageHelper.Save(result, self.imageType, 'divide-gray-const', False, self.
                     .firstDecoder, None, constValue)
    result = Commons.Normalization(image, result)
    ImageHelper.Save(result, self.imageType, 'divide-gray-const', True, self.
                     .firstDecoder, None, constValue)
```

Wynik algorytmu

Rysunek 3.20: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.21: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.22: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 10 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.23: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 10 (środkowy obraz), po normalizacji (prawy obraz)



3.8 Dzielenie obrazu przez inny obraz

Wstęp

Dzielenie dwóch obrazów odbywa się z pomocą formuły:

$$Q(i, j) = P_1(i, j) \div P_2(i, j) \quad (3.6)$$

Aby operacja się powiodła oba obrazy muszą być tych samych rozmiarów, a dla wartości w których dzielnik ma wartość 0 przyjmuje się że wynikiem dzielenia jest dzielna tego działania.

Pierwszy problem pojawia się gdy chcemy zwyczajnie podzielić odpowiadające sobie piksele P_1 i P_2 , gdyż dostaniemy stosunek jednej wartości do drugiej. Wynik takiej operacji będzie bardzo mały jeśli $P_1(i, j) < P_2(i, j)$ lub proporcjonalnie większy jeśli $P_1(i, j) > P_2(i, j)$ co niszczy harmoniczny kontrast pikseli i w rezultacie otrzymujemy to co na rysunku poniżej:

Rysunek 3.24: Rezultat dzielenia obrazu przez inny bez zachowania odpowiedniego balansu pomiędzy pikslami



Remedium na ten błąd jest znalezienie największej sumy P_1 i P_2 określonej jako Q_{max} . Następnie dla każdej pary odpowiadających sobie pikseli używamy wzoru:

$$Q(i, j) = (P_1(i, j) + P_2(i, j)) \times \frac{Z_p}{Q_{max}} \quad (3.7)$$

Wzór ten pozwoli na zachowanie odpowiednich zależności pomiędzy wartościami pikseli.

Jednym z częstszych zastosowań dzielenia obrazów jest wykrywanie zmian (np w filmach po-klatkowych), w podobny sposób jak używane jest odejmowanie obrazów w tym celu. Ponad to dzielenie obrazów nie może zostać użyte jako operacja maskowania przy użyciu odpowiedniego drugiego obrazu - w przeciwnieństwie do operacji mnożenia

Kod algorytmu

```

def divideImages(self):
    print('Divide gray image {} with image {}'.format(self.firstDecoder.name,
                                                       self.secondDecoder.name))
    unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
                               'L')
    firstImage, secondImage = unification.grayUnification()
    width, height = firstImage.shape[0], firstImage.shape[1]

    maxValue = float(numpy.iinfo(firstImage.dtype).max)
    maxSum = float(
        numpy.amax(
            numpy.add(firstImage.astype(numpy.uint32),
                      secondImage.astype(numpy.uint32))))
    scaleFactor = maxValue / maxSum

    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):

```

```
for w in range(width):
    pom = (int(firstImage[h, w]) + int(secondImage[h, w])) *
          scaleFactor
    result[h, w] = math.ceil(pom)

ImageHelper.Save(result, self.imageType, 'divide-gray-images', False,
                  self.firstDecoder, self.
                  secondDecoder)
result = Commons.Normalization(firstImage, result)
ImageHelper.Save(result, self.imageType, 'divide-gray-images', True, self.
                  .firstDecoder, self.
                  secondDecoder)
```

Wynik algorytmu

Rysunek 3.25: Przed uruchomieniem algorytmu (obrazy na górze), po podzieleniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 3.26: Przed uruchomieniem algorytmu (obrazy na górze), po podzieleniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



3.9 Pierwiastkowanie obrazu

Wstęp

Pierwiastkowanie obrazu jest szczególnym rodzajem jego potęgowania gdy wykładnik potęgi jest mniejszy od jedności. Efektem tej operacji jest zwiększenie intensywności piksli, których wartości są bliższe zeru (tzn. zwiększenie kontrastu) i jednocześnie zmniejszenie intensywności piksli wyższych pasm (tzn. zmniejszenie kontrastu). Po operacji zwykle jest niezbędna normalizacja ze względu na ciemniejszy wąski zakres wartości obrazu wyjściowego.

Kod algorytmu

```

def rootFirstImage(self, rootIndex):
    print('Root gray image {} with index {}'.format(self.firstDecoder.name,
                                                    rootIndex))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxValue = float(numpy.iinfo(image.dtype).max)
    result = numpy.ones((height, width), numpy.uint32)
    for h in range(height):
        for w in range(width):
            result[h][w] = int(maxValue * pow(image[h][w], 1 / rootIndex))

```

```

        result[h, w] = image[h, w]**(1.0/rootIndex)

ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'root-gray',
                  False, self.firstDecoder, None,
                  rootIndex)
result = Commons.Normalization(image, result)
ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'root-gray',
                  True, self.firstDecoder, None,
                  rootIndex)

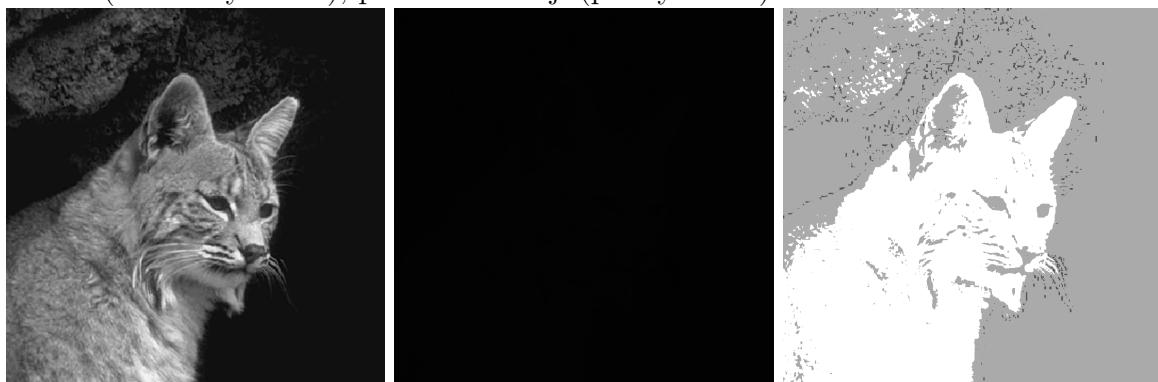
```

Wynik algorytmu

Rysunek 3.27: Przed uruchomieniem algorytmu (lewy obraz), po użyciu pierwiastkowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.28: Przed uruchomieniem algorytmu (lewy obraz), po użyciu pierwiastkowania o wykładniku 4 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.29: Przed uruchomieniem algorytmu (lewy obraz), po użyciu pierwiastkowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.30: Przed uruchomieniem algorytmu (lewy obraz), po użyciu pierwiastkowania o wykładniku 4 (środkowy obraz), po normalizacji (prawy obraz)



3.10 Logarytmowanie obrazu

Wstęp

Przydatność tej operacji wynika z jej efektu. Polega on na rozjaśnieniu i zwiększeniu kontrastu w ciemniejszych rejonach obrazu. Ten efekt bierze się z kompresji *rozpiętości tonalnej* z użyciem funkcji logarytmicznej. Każdy piksel w obrazie jest zamieniany ze swoim logarytmem. Użycie tego operatora punktowego może być przydatne gdy *rozpiętość tonalna* jest zbyt duża aby ją wyświetlić na ekranie. **Logarytmowanie obrazu** odbywa się z pomocą wzoru, który uwzględnia nieograniczonosć logarytmu w zerze poprzez dodanie wartości w argumencie logarytmu:

$$f_m = \log(1 + f) \quad (3.8)$$

Przy logarytmowaniu wartości piksli nie trudno o wyjście poza zakres, więc z tego powodu będziemy używać znormalizowanej wersji wzoru:

$$f_m = 255 * \left(\frac{\log(1 + f(i, j))}{\log(1 + f_{max})} \right) \quad (3.9)$$

W tych przykładach będziemy używać logarytmu o bazie 10, ale można też używać tego z podstawą naturalną gdyż nie wpływa to na kształt krzywej logarytmu. Jedyne co na niego wpływa to skala (wartość 255), która jest stosowana na każdą wartość wychodzącą z logarytmu. Użycie tej skali jest wymagane, aby prawidłowo wyświetlić obraz w systemie 8-bitowym.

Kod algorytmu

```

def logarithm(self):
    print('Logarithm gray image {}'.format(self.firstDecoder.name))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxValue = float(numpy.iinfo(image.dtype).max)
    maxImageValue = numpy.amax(image)
    result = numpy.ones((height, width), numpy.uint32)
    for h in range(height):
        for w in range(width):
            result[h, w] = maxValue * (math.log10(1 + image[h, w]) / math.
                                      log10(1 + maxImageValue))

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'logarithm-
gray', False, self.firstDecoder)

```

Wynik algorytmu

Rysunek 3.31: Przed uruchomieniem algorytmu (lewy obraz), po użyciu logarytmu (prawy obraz)



Rysunek 3.32: Przed uruchomieniem algorytmu (lewy obraz), po użyciu logarytmu (prawy obraz)



Rozdział 4

Operacje sumowania arytmetycznego obrazów barwowych

Rodzaje operatorów arytmetycznych używane na obrazach barwowych różnią się względem ich szarych odpowiedników tylko strukturą na której pracują. Piksel obrazów szarych składa się z jednej wartości z zakresu [0, 255]. W przypadku kolorowych obrazów potrzebne są trzy wartości na każdą barwę *RGB*.

Funkcja normalizacji w przypadku tego rodzaju obrazu wygląda tak:

```
def Normalization(image, result):
    maxValue = numpy.iinfo(image.dtype).max
    fmin = numpy.amin(result)
    fmax = numpy.amax(result)
    result = result.astype(numpy.float32)
    result = maxValue * ((result - fmin) / (fmax - fmin))
    result = result.astype(numpy.uint8)
    return result
```

Pobiera ona informacje o maksymalnej dostępnej wartości dla kanału piksela *maxValue* (w tym wypadku 255). Po czym wśród wszystkich pikselach i ich składowych znajduje najmniejszą i największą wartość *fmin* i *fmax*. Następnie wykonywane są obliczenia mające na celu przeskalowanie wartości do innego przedziału. W wypadku gdy *fmin* i *fmax* znajdują się niedaleko od siebie, normalizacja zapewni aby wartości obrazu wynikowego będą rozpięte na przedział [0, 255] z *fmin* i *fmax*, który może być mniejszy (np z zakresu [50, 90]). Dzięki temu zabiegowi proporcje między barwami zostaną zachowane i obraz ożywi się trochę kolorami.

4.1 Sumowanie (określonej) stałej z obrazem

Algorytm

Opis

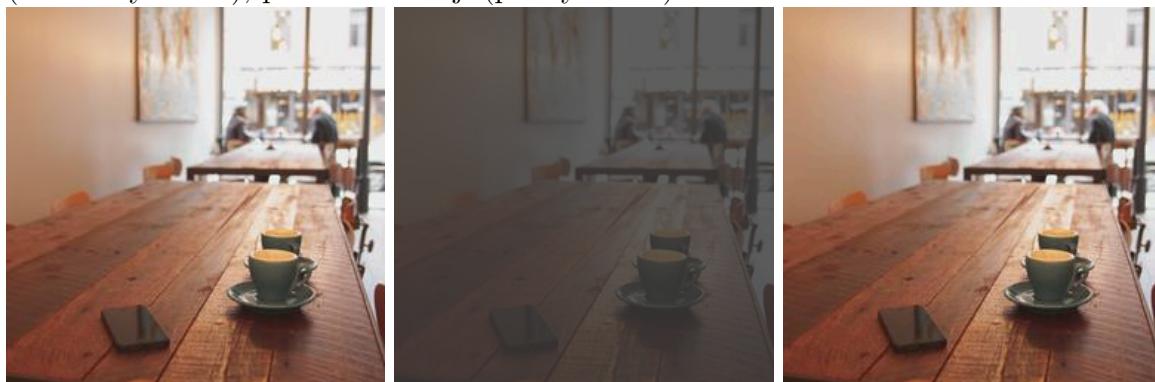
Rysunek 4.1: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



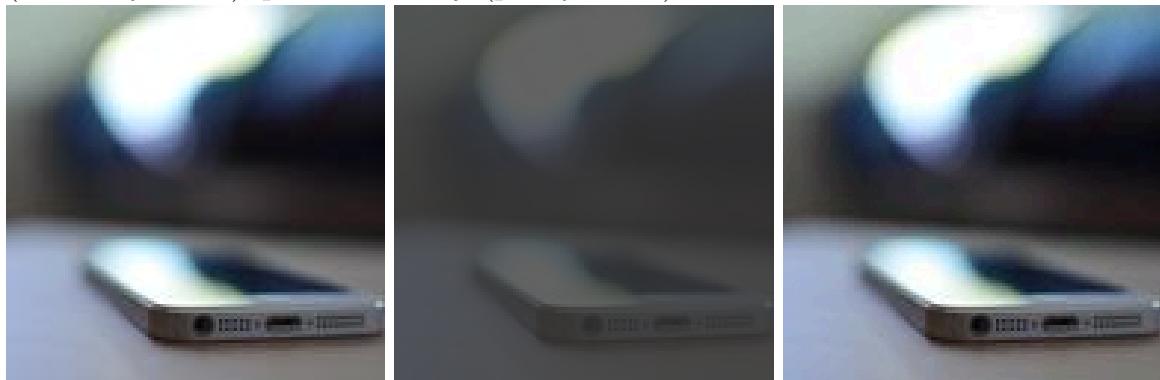
Rysunek 4.2: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.3: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 200 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.4: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 200 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

def sumWithConst(self, constValue):
    print('Sum color image {} with const {}'.format(self.firstDecoder.name,
                                                    constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxSum = float(numpy.amax(numpy.add(image.astype(numpy.uint32),
                                         constValue)))
    maxValue = float(numpy.iinfo(image.dtype).max)
    scaleFactor = (maxSum - maxValue) / maxValue if maxSum > maxValue else 0

    result = numpy.ones((height, width, 3), numpy.uint8)
    for h in range(height):
        for w in range(width):
            R = (image[h, w, 0] - (image[h, w, 0] * scaleFactor)) + (
                constValue - (constValue * scaleFactor))
            G = (image[h, w, 1] - (image[h, w, 1] * scaleFactor)) + (
                constValue - (constValue * scaleFactor))
            B = (image[h, w, 2] - (image[h, w, 2] * scaleFactor)) + (
                constValue - (constValue * scaleFactor))
            result[h, w] = [numpy.ceil(R), numpy.ceil(G), numpy.ceil(B)]

    ImageHelper.Save(result, self.imageType, 'sum-color-const', False, self.
                     firstDecoder, None, constValue)
    result = Commons.Normalization(image, result)
    ImageHelper.Save(result, self.imageType, 'sum-color-const', True, self.
                     firstDecoder, None, constValue)

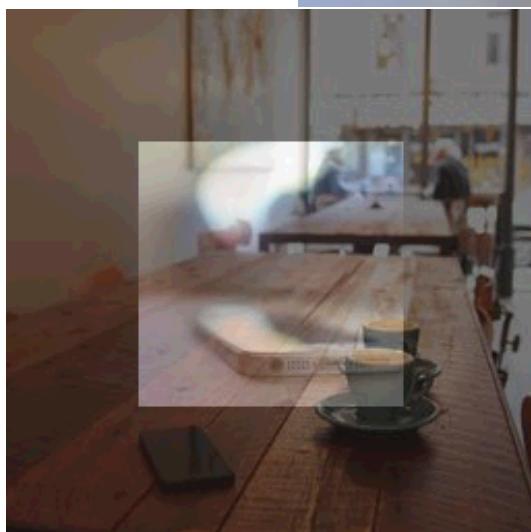
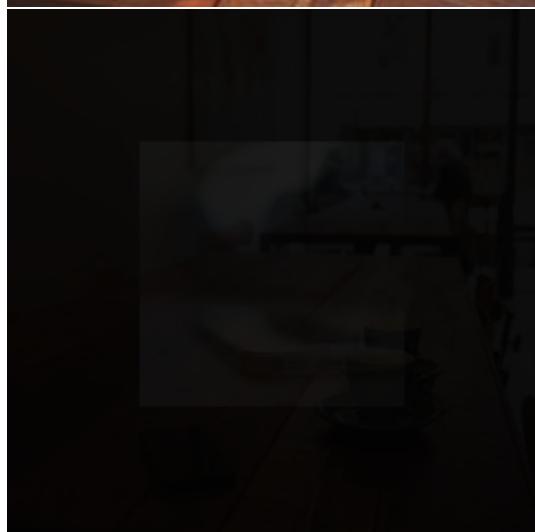
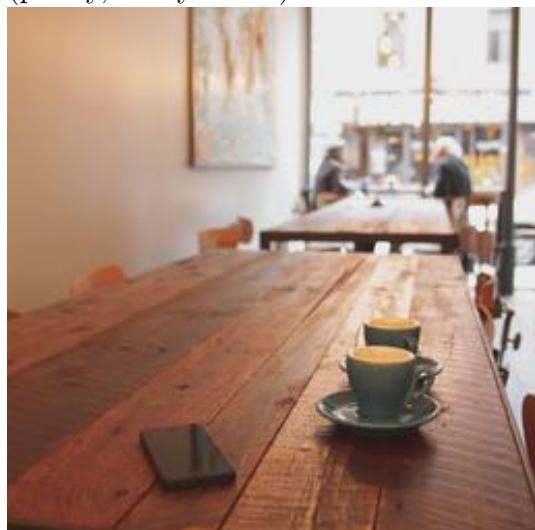
```

4.2 Sumowanie dwóch obrazów

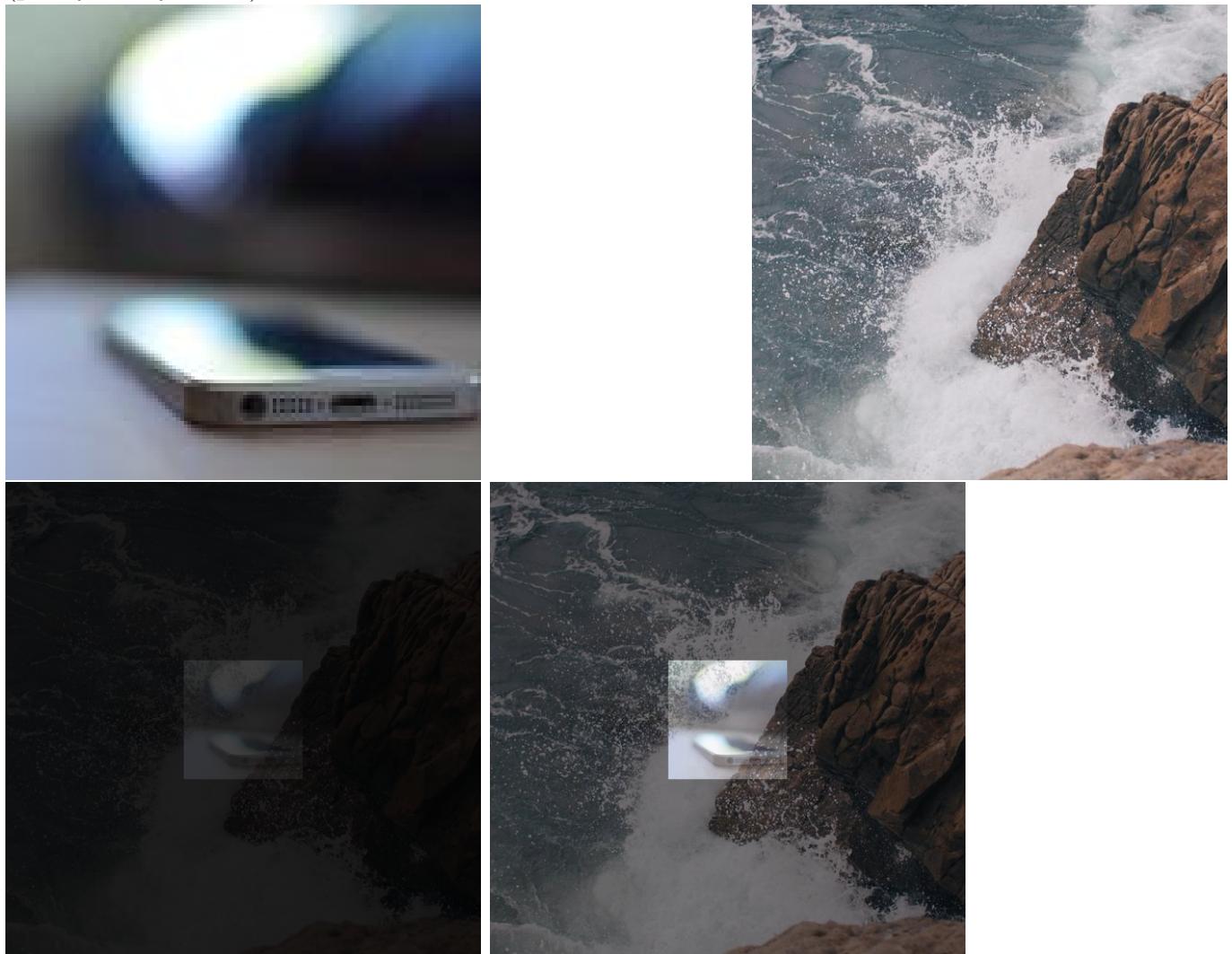
Algorytm

Opis

Rysunek 4.5: Przed (obrazy na górze), po dodaniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 4.6: Przed (obrazy na górze), po dodaniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Kod źródłowy programu

```

def sumImages(self):
    print('Sum color image {} with image {}'.format(self.firstDecoder.name,
                                                    self.secondDecoder.name))
    unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
                                , self.imageType)
    firstImage, secondImage = unification.colorUnification()
    width, height = firstImage.shape[0], firstImage.shape[1]

    maxSum = float(
        numpy.amax(
            numpy.add(firstImage.astype(numpy.uint32),
                      secondImage.astype(numpy.uint32))))
    maxValue = float(numpy.iinfo(firstImage.dtype).max)
    scaleFactor = (maxSum - maxValue) / maxValue if maxSum > maxValue else 0

    result = numpy.ones((height, width, 3), numpy.uint8)
    for h in range(height):
        for w in range(width):
            R = (firstImage[h, w, 0] - (firstImage[h, w, 0] * scaleFactor)) +
                (secondImage[h, w, 0] - (secondImage[h, w, 0] * scaleFactor))

```

```
G = (firstImage[h, w, 1] - (firstImage[h, w, 1] * scaleFactor)) +
      (secondImage[h, w, 1] -
       (secondImage[h, w, 1] * scaleFactor))
B = (firstImage[h, w, 2] - (firstImage[h, w, 2] * scaleFactor)) +
      (secondImage[h, w, 2] -
       (secondImage[h, w, 2] * scaleFactor))
result[h, w] = [numpy.ceil(R), numpy.ceil(G), numpy.ceil(B)]

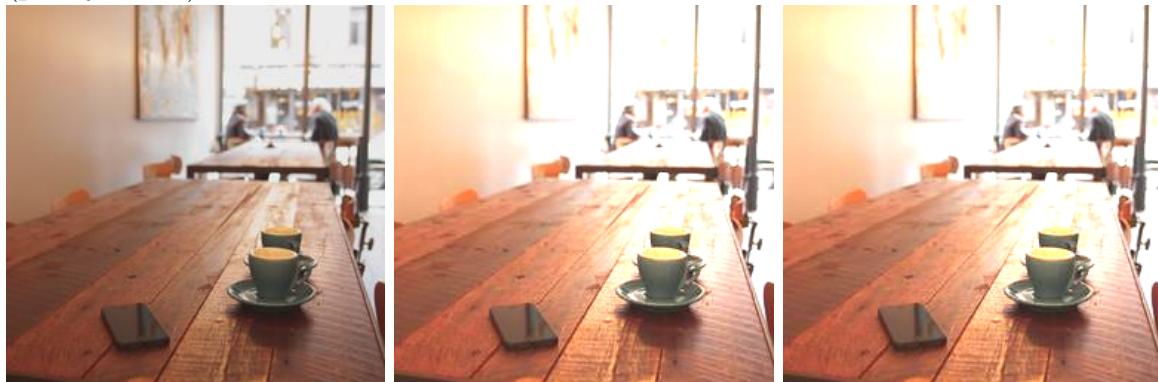
ImageHelper.Save(result, self.imageType, 'sum-color-images', False, self.
                  firstDecoder, self.secondDecoder
)
result = Commons.Normalization(firstImage, result)
ImageHelper.Save(result, self.imageType, 'sum-color-images', True, self.
                  firstDecoder, self.secondDecoder
)
```

4.3 Mnożenie obrazu przez zadaną liczbę

Algorytm

Opis

Rysunek 4.7: Przed (lewy obraz), po pomnożeniu przez wartości 1.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.8: Przed (lewy obraz), po pomnożeniu przez wartości 3.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.9: Przed (lewy obraz), po pomnożeniu przez wartości 1.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.10: Przed (lewy obraz), po pomnożeniu przez wartości 3.5 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

print('Multiply color image {} with const {}'.format(self.firstDecoder.
    name, constValue))
height, width = self.firstDecoder.height, self.firstDecoder.width
image = self.firstDecoder.getPixels()
maxValue = numpy.iinfo(image.dtype).max
result = numpy.ones((height, width, 3), numpy.uint8)

for h in range(height):
    for w in range(width):
        R = image[h, w, 0] * constValue
        G = image[h, w, 1] * constValue
        B = image[h, w, 2] * constValue
        result[h, w, 0] = R if R <= maxValue else maxValue
        result[h, w, 1] = G if G <= maxValue else maxValue
        result[h, w, 2] = B if B <= maxValue else maxValue

ImageHelper.Save(result, self.imageType, 'multiply-color-const', False,
    self.firstDecoder, None,
    constValue)
result = Commons.Normalization(image, result)
ImageHelper.Save(result, self.imageType, 'multiply-color-const', True,
    self.firstDecoder, None,
    constValue)

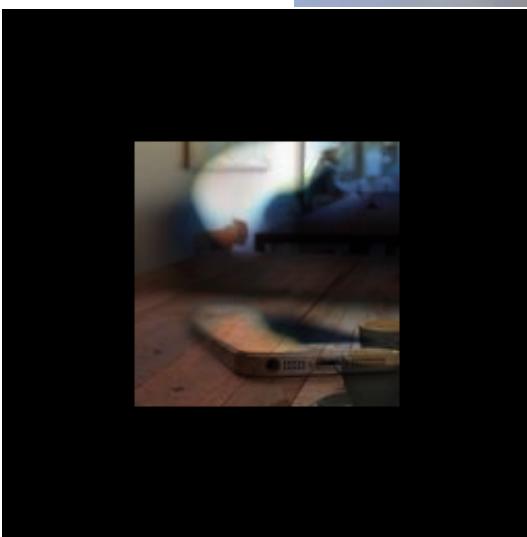
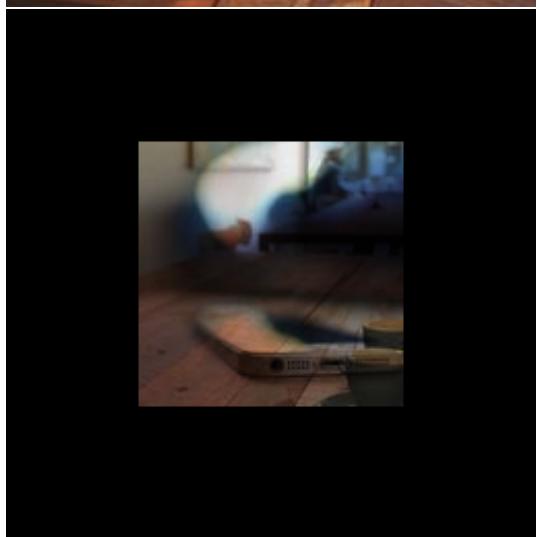
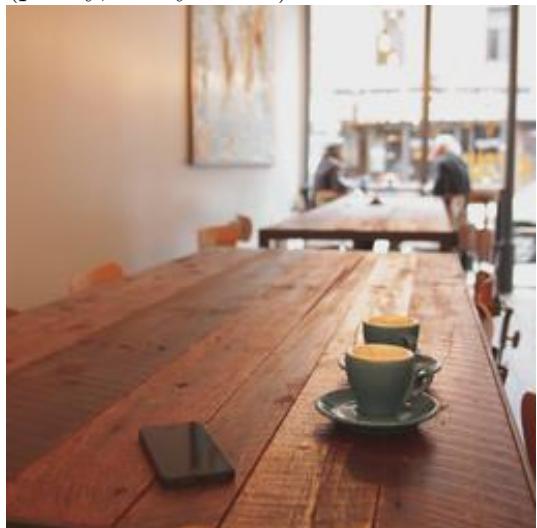
```

4.4 Mnożenie obrazu przez inny obraz

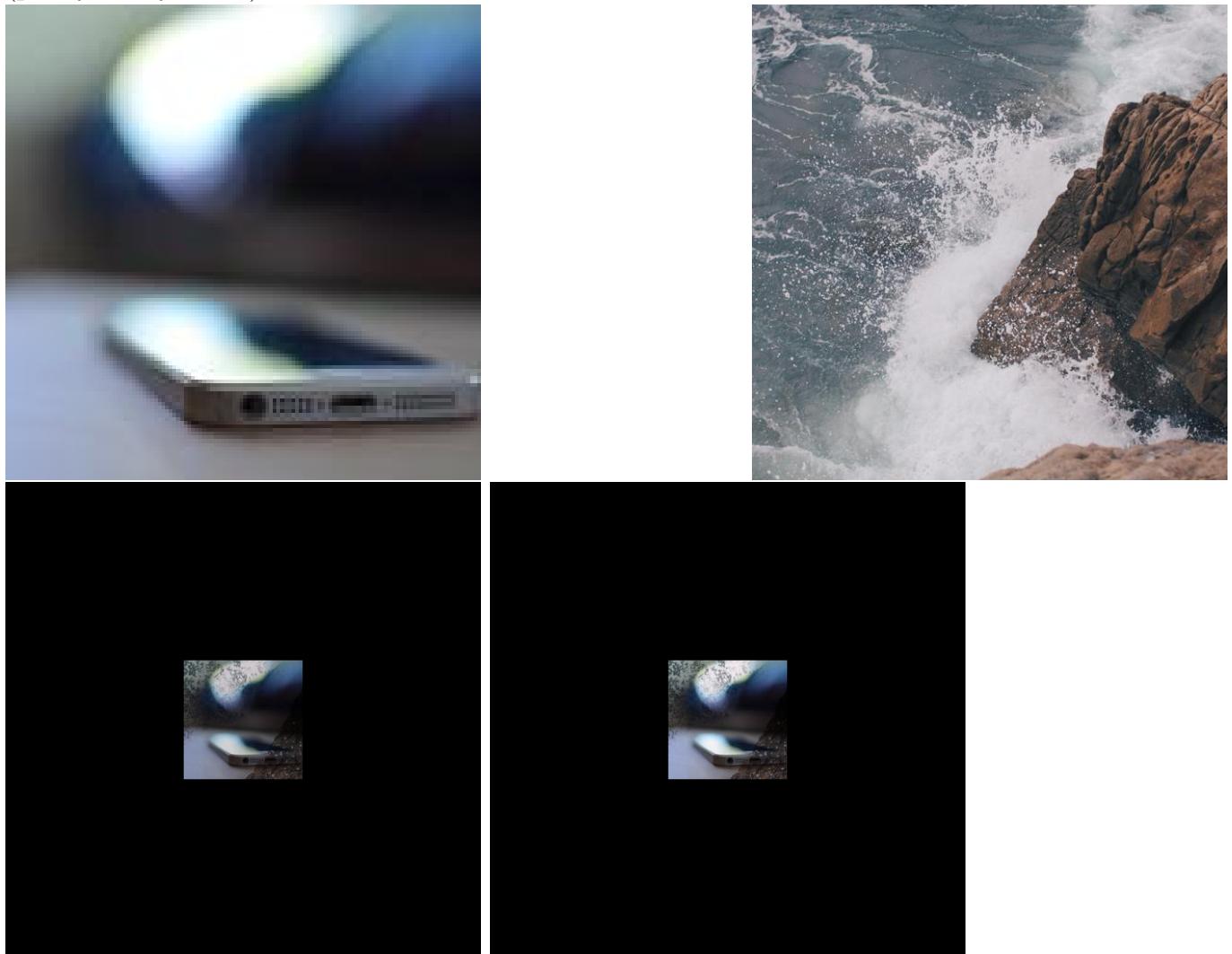
Algorytm

Opis

Rysunek 4.11: Przed (górnego obrazy), po pomnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 4.12: Przed (górnne obrazy), po pomnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Kod źródłowy programu

```

print('Multiply color image {} with image {}'.format(self.firstDecoder.
                                                       name, self.secondDecoder.name))
unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
                           self.imageType)
firstImage, secondImage = unification.colorUnification()
width, height = firstImage.shape[0], firstImage.shape[1]

maxValue = float(numpy.iinfo(firstImage.dtype).max)
result = numpy.ones((height, width, 3), numpy.uint8)
for h in range(height):
    for w in range(width):
        result[h, w, 0] = int(firstImage[h, w, 0]) * int(secondImage[h, w,
                                                                     0]) / maxValue
        result[h, w, 1] = int(firstImage[h, w, 1]) * int(secondImage[h, w,
                                                                     1]) / maxValue
        result[h, w, 2] = int(firstImage[h, w, 2]) * int(secondImage[h, w,
                                                                     2]) / maxValue

ImageHelper.Save(result, self.imageType, 'multiply-color-images', False,
                  self.firstDecoder, self.
                  secondDecoder)
result = Commons.Normalization(firstImage, result)

```

```
ImageHelper.Save(result, self.imageType, 'multiply-color-images', True,  
self.firstDecoder, self.  
secondDecoder)
```

4.5 Mieszanie obrazów z określonym współczynnikiem

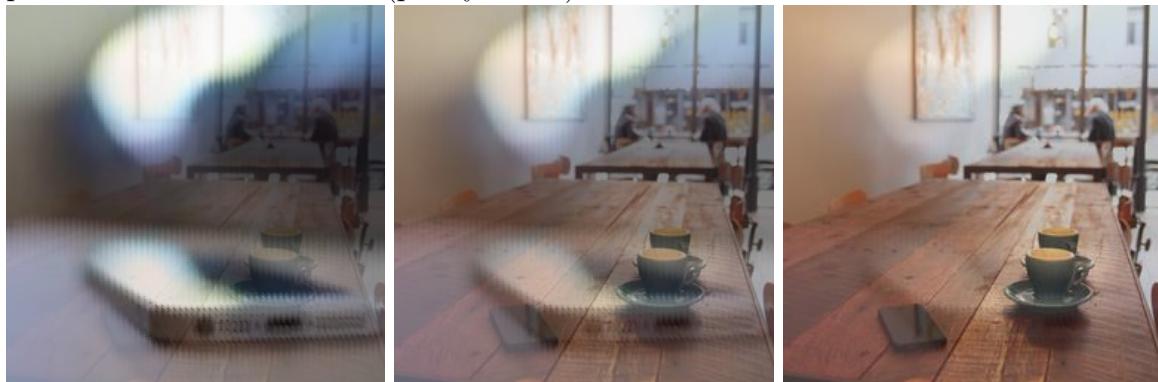
Algorytm

Opis

Rysunek 4.13: Przed uruchomieniem algorytmu



Rysunek 4.14: Po mieszaniu o wartość 0.2 (lewy obraz), po mieszaniu o wartość 0.5 (środkowy obraz), po mieszaniu o wartość 0.8 (prawy obraz)



Kod źródłowy programu

```

print('Blending color image {} with image {} and ratio {}'.format(self.
                     firstDecoder.name, self.
                     secondDecoder.name, ratio))

if ratio < 0 or ratio > 1.0:
    raise ValueError('ratio is wrong')

unification = Unification(self.firstDecoder.name, self.secondDecoder.name
                           , self.imageType)
firstImage = unification.scaleUpColor(self.firstDecoder)
secondImage = unification.scaleUpColor(self.secondDecoder)
width, height = firstImage.shape[0], firstImage.shape[1]

result = numpy.ones((height, width, 3), numpy.uint8)

```

```
for h in range(height):
    for w in range(width):
        result[h, w, 0] = ratio * firstImage[h, w, 0] + (1 - ratio) *
                           secondImage[h, w, 0]
        result[h, w, 1] = ratio * firstImage[h, w, 1] + (1 - ratio) *
                           secondImage[h, w, 1]
        result[h, w, 2] = ratio * firstImage[h, w, 2] + (1 - ratio) *
                           secondImage[h, w, 2]

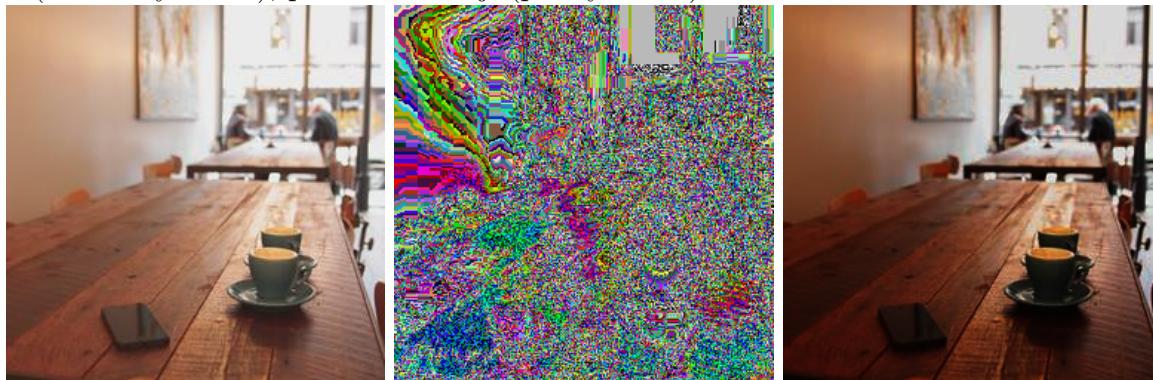
ImageHelper.Save(result, self.imageType, 'blend-color-images', False,
                 self.firstDecoder, None, ratio)
```

4.6 Potęgowanie obrazu (zadaną potęgą)

Algorytm

Opis

Rysunek 4.15: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.16: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Kod źródłowy programu

```

print('Power color image {} with image {} and index {}'.format(self.
    firstDecoder.name, self.
    secondDecoder.name, powerIndex))

height, width = self.firstDecoder.height, self.firstDecoder.width
image = self.firstDecoder.getPixels()

maxValue = float(numpy.iinfo(image.dtype).max)
result = numpy.ones((height, width, 3), numpy.uint32)
for h in range(height):
    for w in range(width):
        result[h, w, 0] = image[h, w, 0]**powerIndex
        result[h, w, 1] = image[h, w, 1]**powerIndex
        result[h, w, 2] = image[h, w, 2]**powerIndex

ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'power-color'
    , False, self.firstDecoder,
    None, powerIndex)

result = Commons.Normalization(image, result)

```

```
ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'power-color',
                 True, self.firstDecoder, None,
                 powerIndex)
```

4.7 Dzielenie obrazu przez liczbę

Wstęp

W wypadku gdy wartość piksla jest skalarem - tak jak w przypadku obrazów kolorowych. Stała C jest dzielona przez każdy z trzech kanałów.

Kod algorytmu

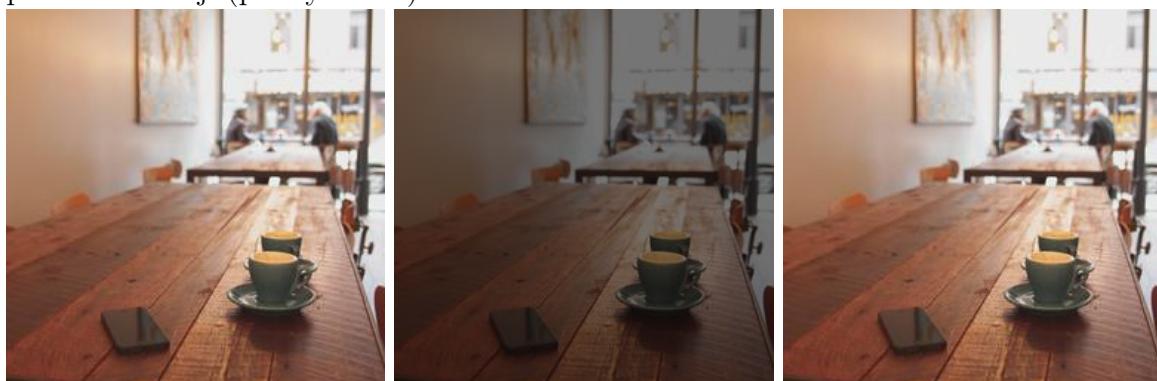
```
def divideWithConst(self, constValue):
    print('Divide color image {} with const {}'.format(self.firstDecoder.name,
                                                       , constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = numpy.iinfo(image.dtype).max
    result = numpy.ones((height, width, 3), numpy.uint8)

    for h in range(height):
        for w in range(width):
            result[h, w, 0] = image[h, w, 0] / constValue
            result[h, w, 1] = image[h, w, 1] / constValue
            result[h, w, 2] = image[h, w, 2] / constValue

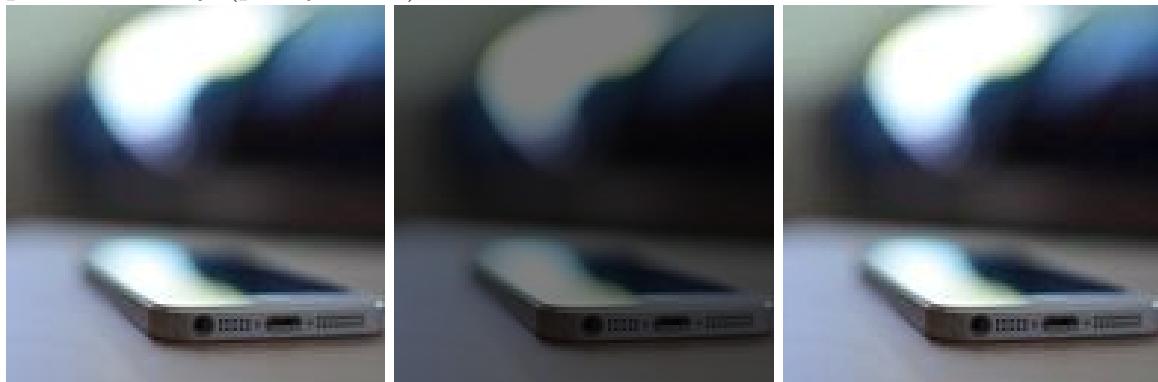
    ImageHelper.Save(result, self.imageType, 'divide-color-const', False,
                     self.firstDecoder, None,
                     constValue)
    result = Commons.Normalization(image, result)
    ImageHelper.Save(result, self.imageType, 'divide-color-const', True, self
                     .firstDecoder, None, constValue)
```

Wynik algorytmu

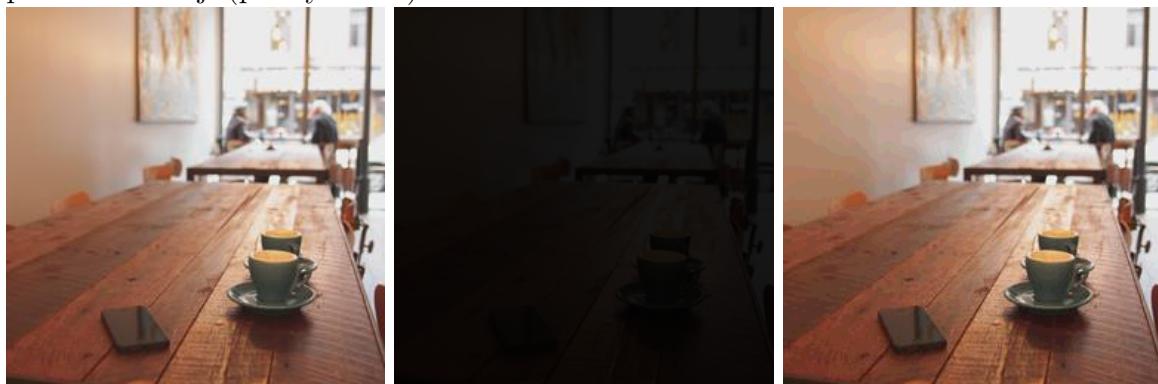
Rysunek 4.17: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 2 (środkowy obraz), po normalizacji (prawy obraz)



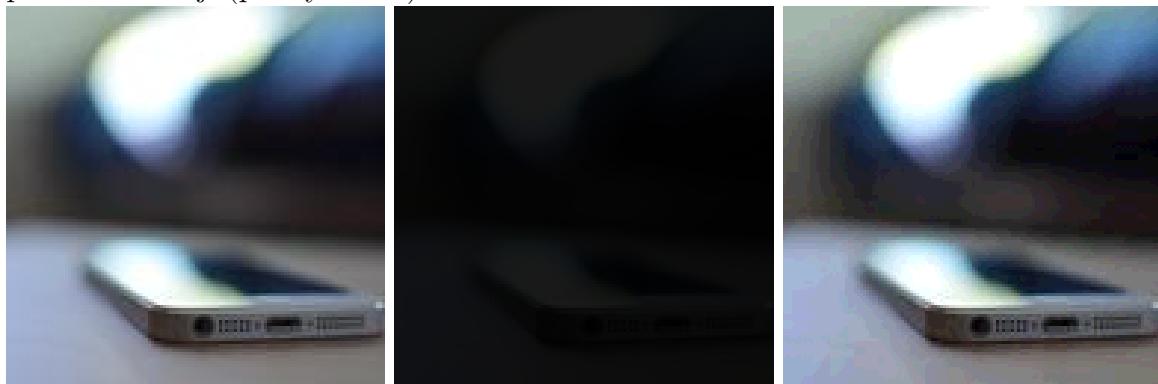
Rysunek 4.18: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.19: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 10 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.20: Przed uruchomieniem (lewy obraz), po dzieleniu przez wartość 10 (środkowy obraz), po normalizacji (prawy obraz)



4.8 Dzielenie obrazu przez inny obraz

Wstęp

Jedyna różnica pomiędzy sposobem dzielenia obrazów szarych od barwnych polega na innej wartości piksla, który w przypadku tych drugich jest wektorem trzech wartości. Wymusza to znalezienie maksymalnej sumy na przestrzeni każdej z trzech składowych RGB i na ich podstawie wytyczyć można odpowiednia wartość dzielenia za pomocą wzoru:

$$Q(i, j) = (P_1(i, j) + P_2(i, j)) \times \frac{Z_p}{Q_{max}} \quad (4.1)$$

Kod algorytmu

```

def divideImages(self):
    print('Divide color image {} with image {}'.format(self.firstDecoder.name,
                                                       self.secondDecoder.name))
    unification = Unification(self.firstDecoder.name, self.secondDecoder.name,
                               self.imageType)
    firstImage, secondImage = unification.colorUnification()
    width, height = firstImage.shape[0], firstImage.shape[1]

    maxValue = float(numpy.iinfo(firstImage.dtype).max)
    sumR = float(
        numpy.amax(
            numpy.add(firstImage[:, :, 0].astype(numpy.uint32),
                      secondImage[:, :, 0].astype(numpy.uint32))))
    sumG = float(
        numpy.amax(
            numpy.add(firstImage[:, :, 1].astype(numpy.uint32),
                      secondImage[:, :, 1].astype(numpy.uint32))))
    sumB = float(
        numpy.amax(
            numpy.add(firstImage[:, :, 2].astype(numpy.uint32),
                      secondImage[:, :, 2].astype(numpy.uint32))))
    scaleR = maxValue / sumR
    scaleG = maxValue / sumG
    scaleB = maxValue / sumB

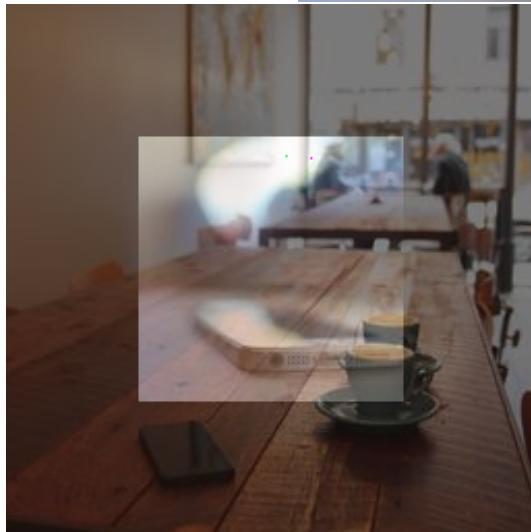
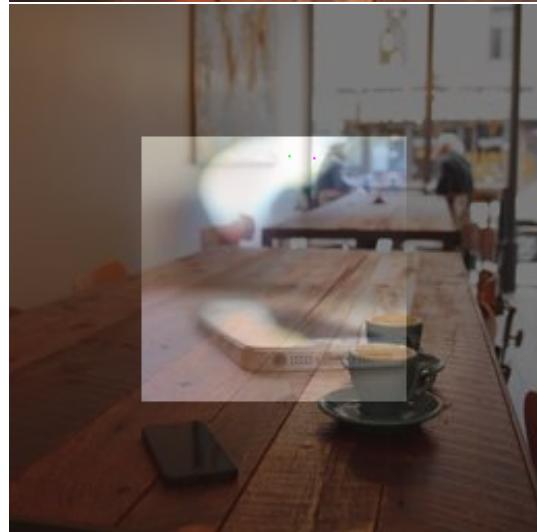
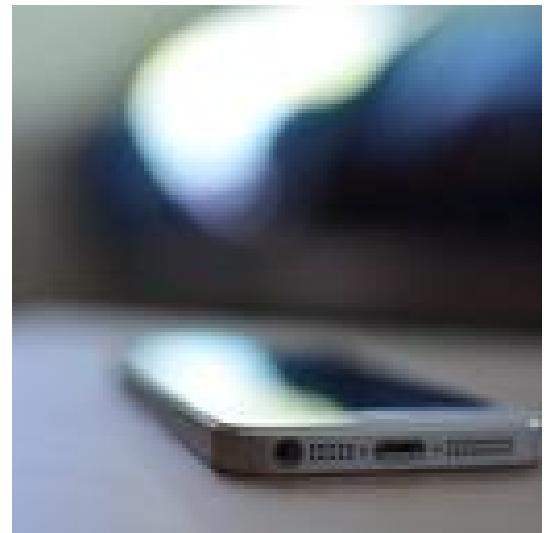
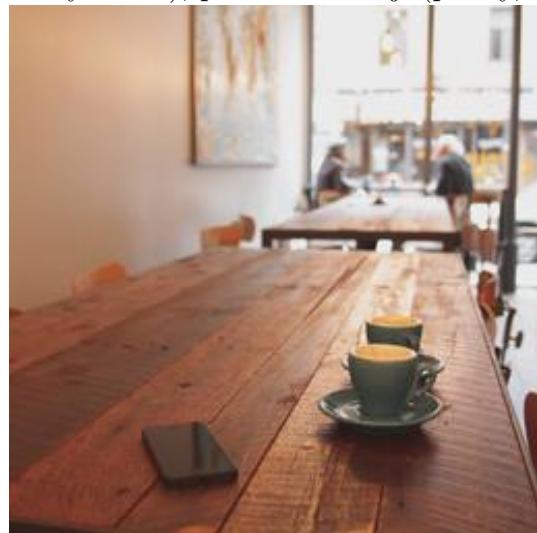
    result = numpy.ones((height, width, 3), numpy.uint8)
    for h in range(height):
        for w in range(width):
            R = (int(firstImage[h, w, 0]) + int(secondImage[h, w, 0])) *
                scaleR
            G = (int(firstImage[h, w, 1]) + int(secondImage[h, w, 1])) *
                scaleG
            B = (int(firstImage[h, w, 2]) + int(secondImage[h, w, 2])) *
                scaleB
            result[h, w, 0] = numpy.ceil(R)
            result[h, w, 1] = numpy.ceil(G)
            result[h, w, 2] = numpy.ceil(B)

    ImageHelper.Save(result, self.imageType, 'divide-color-images', False,
                     self.firstDecoder, self.secondDecoder)
    result = Commons.Normalization(firstImage, result)
    ImageHelper.Save(result, self.imageType, 'divide-color-images', True,
                     self.firstDecoder, self.secondDecoder)

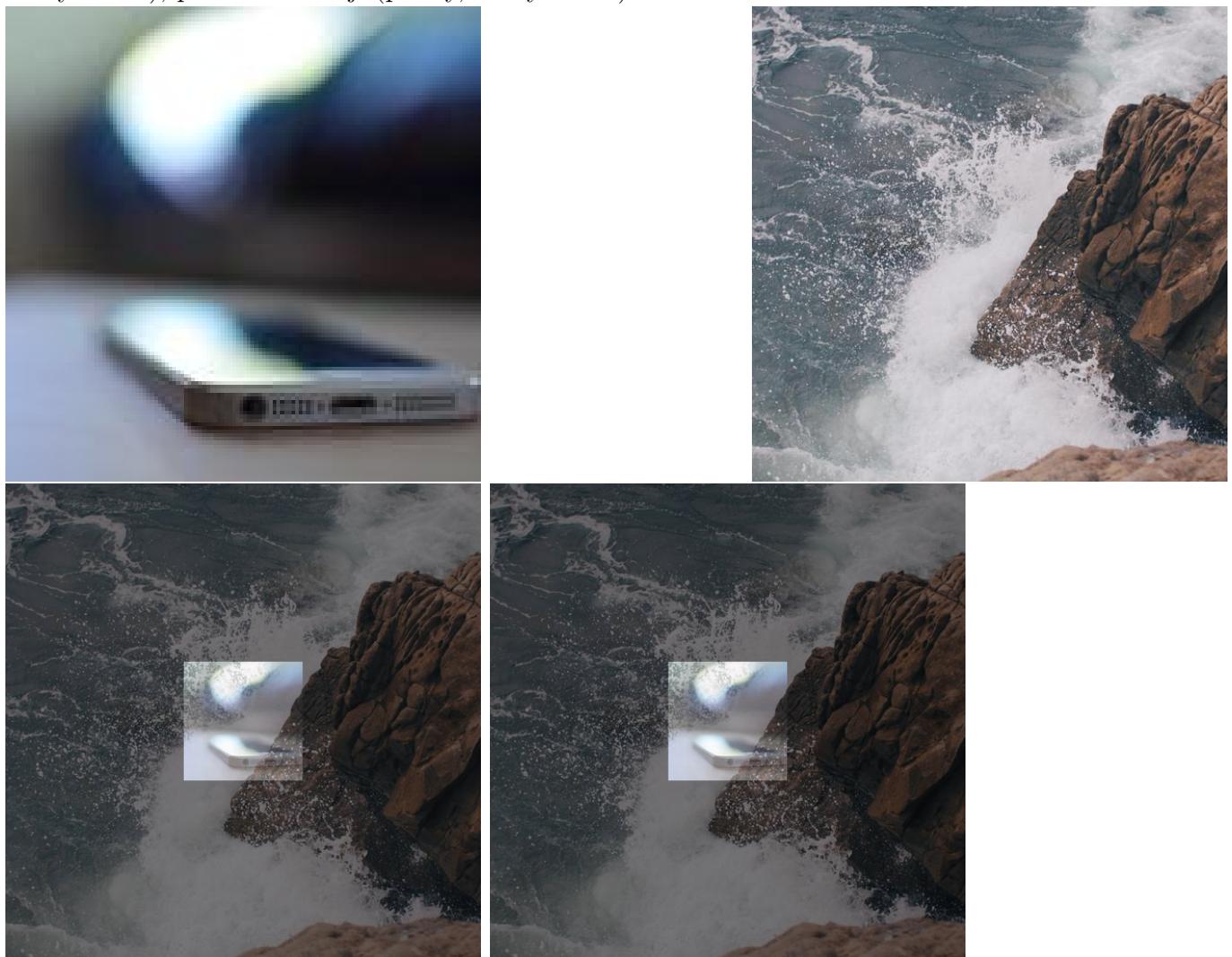
```

Wynik algorytmu

Rysunek 4.21: Przed uruchomieniem algorytmu (obrazy na górze), po podzieleniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



Rysunek 4.22: Przed uruchomieniem algorytmu (obrazy na górze), po podzieleniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



4.9 Pierwiastkowanie obrazu

Wstęp

Pierwiastkowanie obrazu jest szczególnym rodzajem jego potęgowania gdy wykładnik potęgi jest mniejszy od jedności. Efektem tej operacji jest zwiększenie intensywności piksli, których wartości są bliższe zeru (tzn. zwiększenie kontrastu) i jednocześnie zmniejszenie intensywności piksli wyższych pasm (tzn. zmniejszenie kontrastu). Po operacji zwykle jest niezbędna normalizacja ze względu na ciemniejszy wąski zakres wartości obrazu wyjściowego.

Kod algorytmu

```
def rootFirstImage(self, rootIndex):
    print('Root color image {} with image {} and index {}'.format(self.
        firstDecoder.name, self.
        secondDecoder.name, rootIndex))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

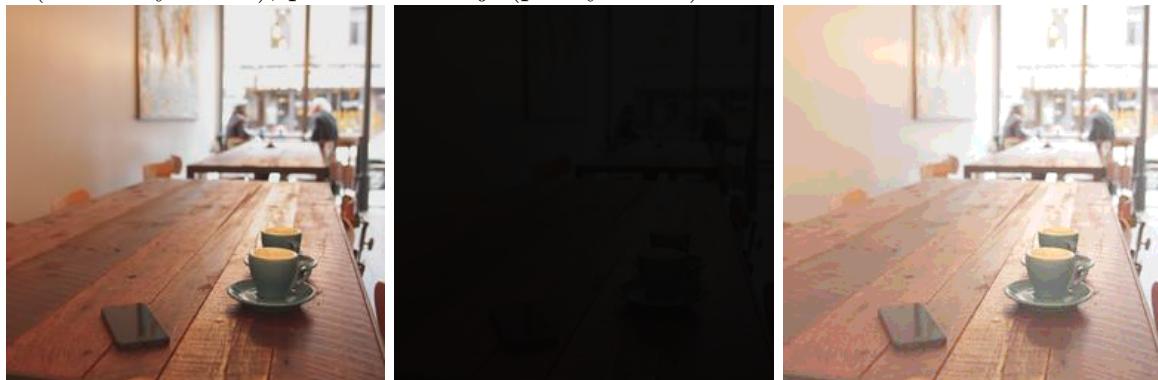
    maxValue = float(numpy.iinfo(image.dtype).max)
    result = numpy.ones((height, width, 3), numpy.uint32)
    for h in range(height):
```

```
for w in range(width):
    result[h, w, 0] = image[h, w, 0]**(1.0/rootIndex)
    result[h, w, 1] = image[h, w, 1]**(1.0/rootIndex)
    result[h, w, 2] = image[h, w, 2]**(1.0/rootIndex)

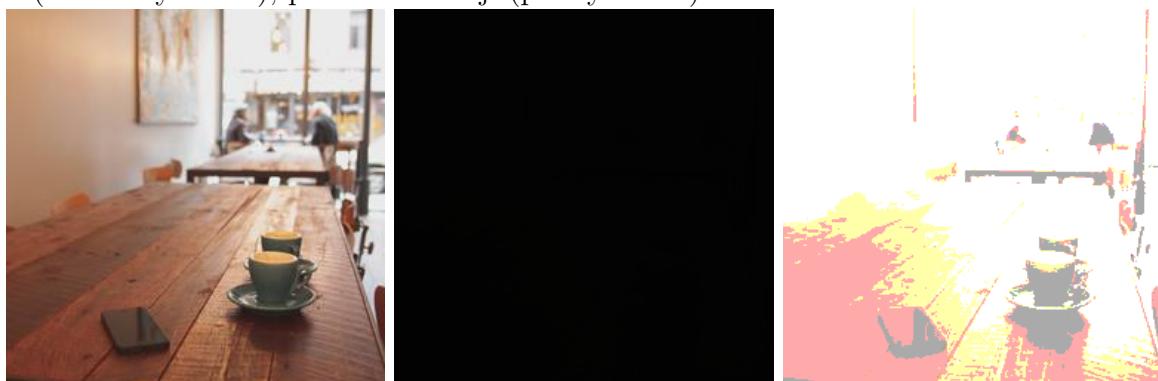
ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'root-color',
                  , False, self.firstDecoder, None,
                  , rootIndex)
result = Commons.Normalization(image, result)
ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'root-color',
                  , True, self.firstDecoder, None,
                  , rootIndex)
```

Wynik algorytmu

Rysunek 4.23: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.24: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 4 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.25: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 4.26: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 4 (środkowy obraz), po normalizacji (prawy obraz)



4.10 Logarytmowanie obrazu

Wstęp

Przydatność tej operacji wynika z jej efektu. Polega on na rozjaśnieniu i zwiększeniu kontrastu w ciemniejszych rejonach obrazu. Ten efekt bierze się z kompresji *rozpiętości tonalnej* z użyciem funkcji logarytmicznej. Każdy piksel w obrazie jest zamieniany ze swoim logarytmem. Użycie tego operatora punktowego może być przydatne gdy *rozpiętość tonalna* jest zbyt duża aby ją wyświetlić na ekranie. **Logarytmowanie obrazu** odbywa się z pomocą wzoru, który uwzględnia nieograniczonosć logarytmu w zerze poprzez dodanie wartości w argumencie logarytmu:

$$f_m = \log(1 + f) \quad (4.2)$$

Przy logarytmowaniu wartości piksli nie trudno o wyjście poza zakres, więc z tego powodu będziemy używać znormalizowanej wersji wzoru:

$$f_m = 255 * \left(\frac{\log(1 + f(i, j))}{\log(1 + f_{max})} \right) \quad (4.3)$$

W tych przykładach będziemy używać logarytmu o bazie 10, ale można też używać tego z podstawą naturalną gdyż nie wpływa to na kształt krzywej logarytmu. Jedyne co na niego wpływa to skala (wartość 255), która jest stosowana na każdą wartość wychodzącą z logarytmu. Użycie tej skali jest wymagane, aby prawidłowo wyświetlić obraz w systemie 8-bitowym.

Kod algorytmu

```

def logarithm(self):
    print('Logarithm color image {}'.format(self.firstDecoder.name))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()

    maxValue = float(numpy.iinfo(image.dtype).max)
    maxR = numpy.amax(image[:, :, 0])
    maxG = numpy.amax(image[:, :, 1])
    maxB = numpy.amax(image[:, :, 2])
    result = numpy.ones((height, width, 3), numpy.uint32)
    for h in range(height):
        for w in range(width):
            result[h, w, 0] = maxValue * (math.log10(1 + image[h, w, 0]) /
                                         math.log10(1 + maxR))
            result[h, w, 1] = maxValue * (math.log10(1 + image[h, w, 1]) /
                                         math.log10(1 + maxG))
            result[h, w, 2] = maxValue * (math.log10(1 + image[h, w, 2]) /
                                         math.log10(1 + maxB))

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'logarithm-
        color', False, self.firstDecoder
    )

```

Wynik algorytmu

Rysunek 4.27: Przed uruchomieniem algorytmu (lewy obraz), po użyciu logarytmu (prawy obraz)



Rysunek 4.28: Przed uruchomieniem algorytmu (lewy obraz), po użyciu logarytmu (prawy obraz)



74 ROZDZIAŁ 4. OPERACJE SUMOWANIA ARYTMETYCZNEGO OBRAZÓW BARWOWYCH

Rozdział 5

Operacje geometryczne na obrazie

Operacje geometryczne przekształcają położenie pikseli (x_1, y_1) w obrazie wejściowym do nowej lokalizacji (x_2, y_2) w obrazie wynikowym. Dzięki temu możemy dopasować obraz do odpowiedniego układu współrzędnych lub użyć tych operacji do eliminacji geometrycznych zakłóceń obrazu (dystorsji).

5.1 Przemieszczenie obrazu o zadany wektor

Opis

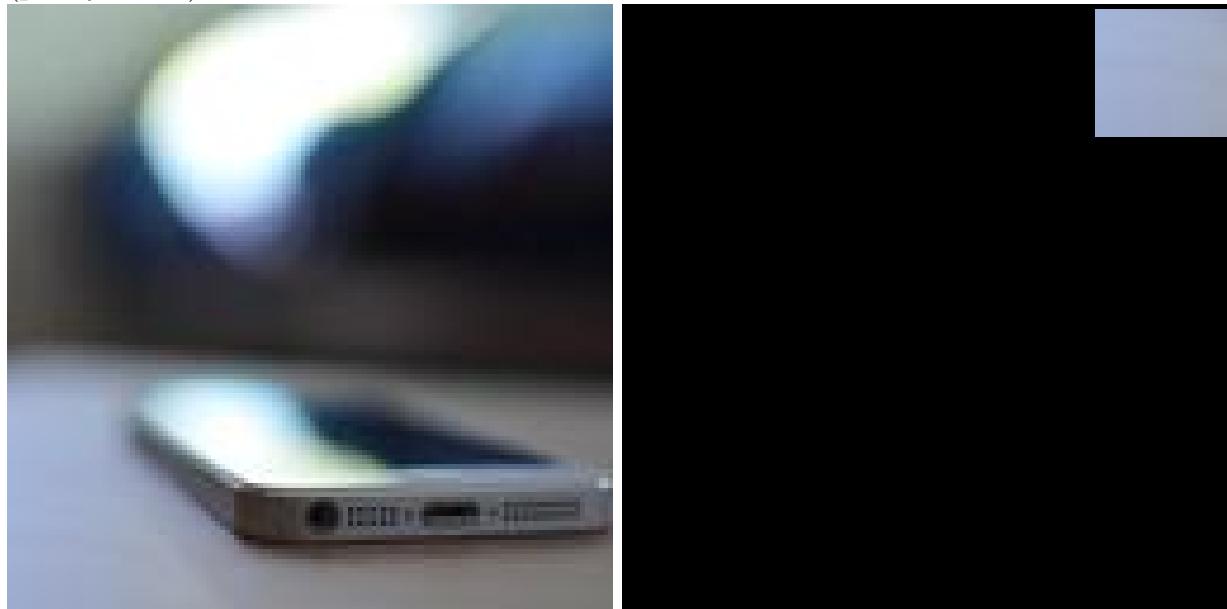
Operacja translacji wykonuje transformację geometryczną polegającą na przeniesieniu każdego z punktów obrazu wejściowego w nowe miejsce na obrazie wynikowym. Pod wpływem translacji element obrazu zlokalizowany na (x_1, y_1) zostanie przesunięty na nową pozycję (x_2, y_2) . Różnicą pomiędzy (x_1, y_1) i (x_2, y_2) jest wektor (bx, by) , który jest określony przez użytkownika.

Operacja przemieszczenia przybiera postać:

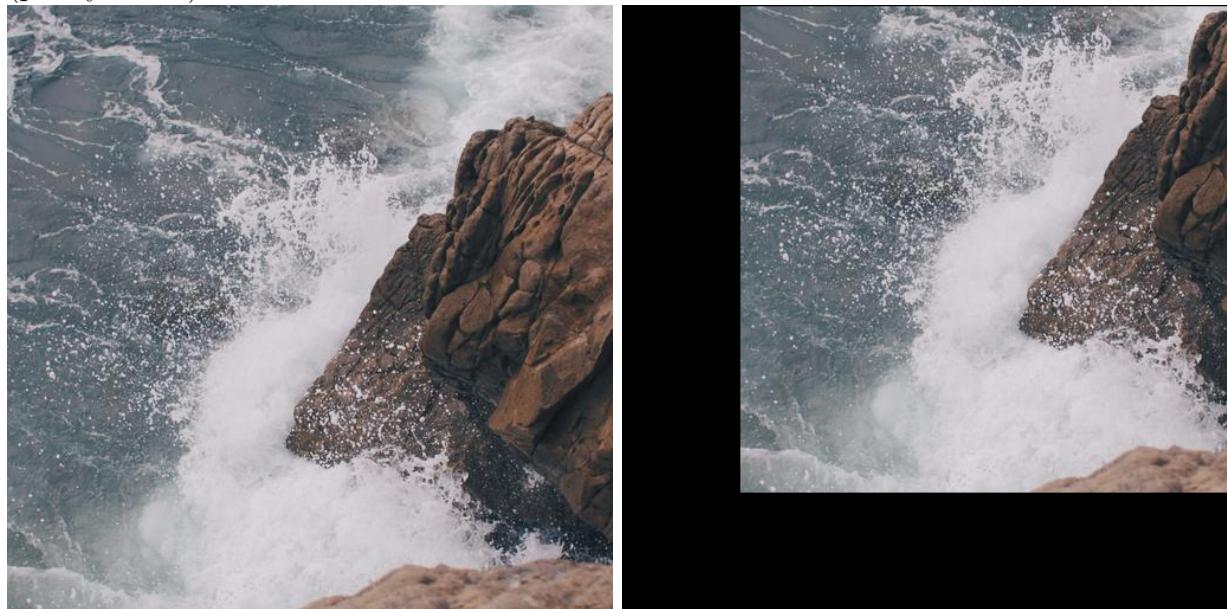
$$x_2 = x_1 + b_x \quad (5.1)$$

$$y_2 = y_1 + b_y \quad (5.2)$$

Rysunek 5.1: Przed uruchomieniem algorytmu (lewy obraz), po przesunięciu o wektor [100, 100] (prawy obraz)



Rysunek 5.2: Przed uruchomieniem algorytmu (lewy obraz), po przesunięciu o wektor [100, 100] (prawy obraz)



Kod źródłowy algorytmu

```
def translate(self, deltaX = 0, deltaY = 0):
    print('Translation color image {} to point ({},{}).format(self.decoder.
        name, deltaX, deltaY)')
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()
    result = numpy.zeros((height, width, 3), numpy.uint8)

    for y in range(height):
        for x in range(width):
            if 0 < y + deltaY < height and 0 < x + deltaX < width:
                result[y + deltaY][x + deltaX] = image[y][x]

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'translate',
        False, self.decoder)
```

5.2 Jednorodne skalowanie obrazu

Opis

Skalowanie jednorodne obrazu składa się na pomnożenie współrzędnych każdego piksela przez określoną wartość.

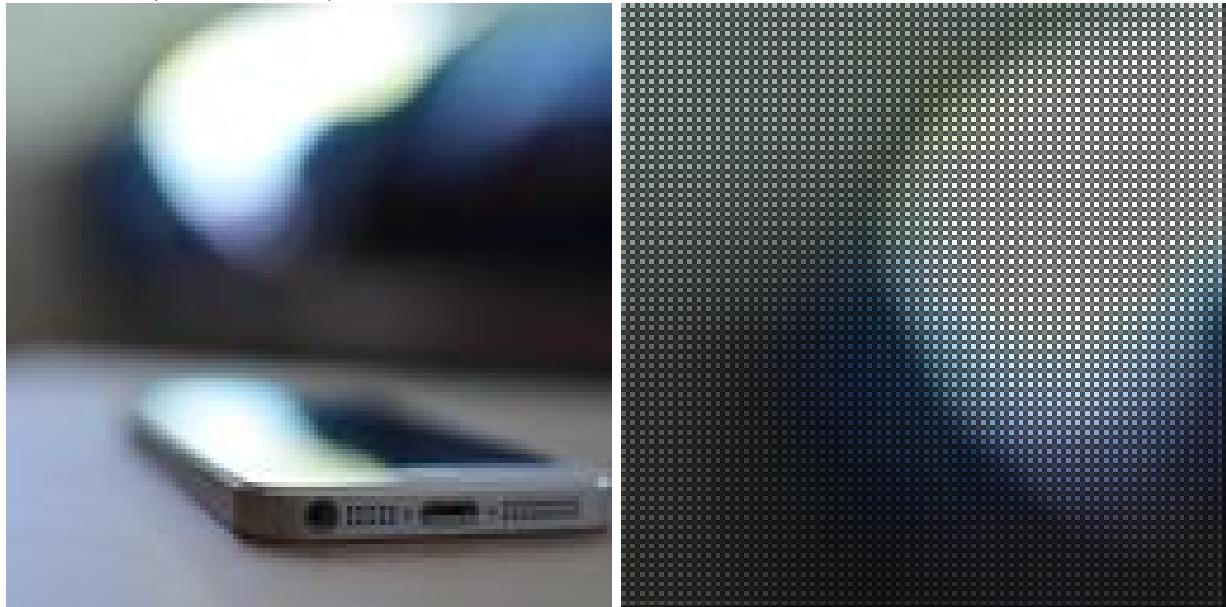
$$x_2 = S_x * x_1 \quad (5.3)$$

$$y_2 = S_y * y_1 \quad (5.4)$$

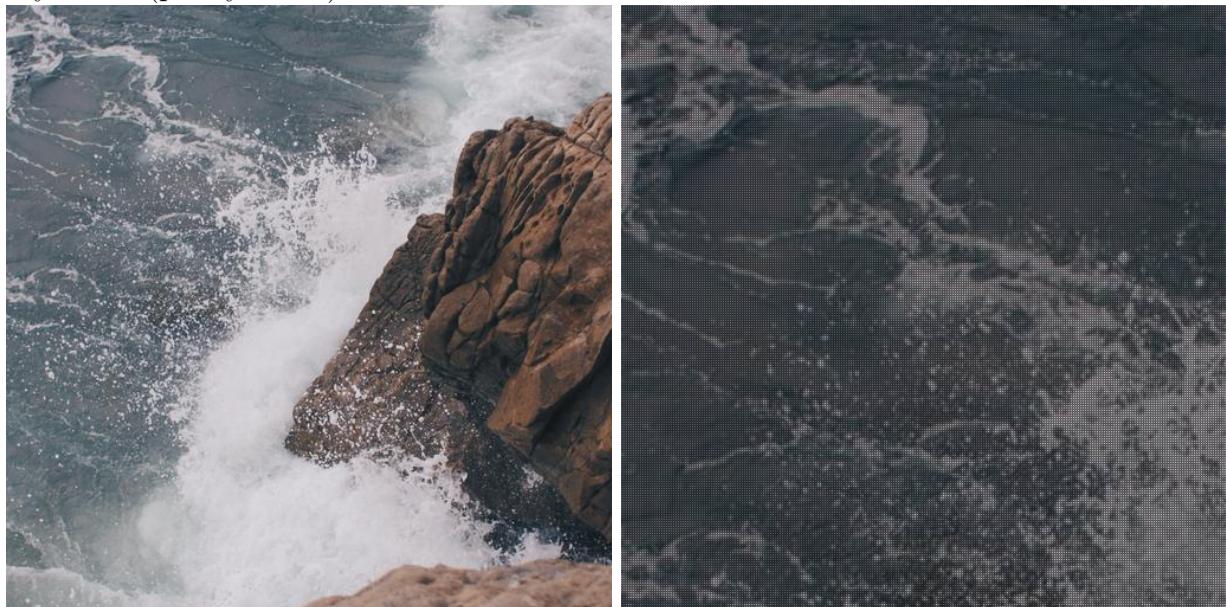
Przy czym skalowanie jednorodne oznacza, że po zmianie wartości współrzędnych nasz obraz zachowuje dawne proporcje. Czyli:

$$S_x = S_y \quad (5.5)$$

Rysunek 5.3: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik 2 (prawy obraz)



Rysunek 5.4: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik 2 (prawy obraz)



Kod źródłowy algorytmu

```
def homogeneousScaling(self, scale = 1.0):
    print('Homogeneous scaling color image {} about scale {}'.format(self.decoder.name, scale))
    image = self.decoder.getPixels()

    result = self._scale(image, scale, scale)
    self._interpolateColor(result)

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'homogeneous
                           -scaling', False, self.decoder)
```

```
def _scale(self, matrix, scaleX, scaleY):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.full((height, width, 3), 1, numpy.uint8)
    for y in range(height):
        for x in range(width):
            if scaleY * y < height and scaleX * x < width:
                result[int(scaleY * y)][int(scaleX * x)] = matrix[y][x]
    return result
```

5.3 Niejednorodne skalowanie obrazu

Opis

Skalowanie niejednorodne obrazu składa się na pomnożenie współrzędnych każdego piksela przez określoną wartość.

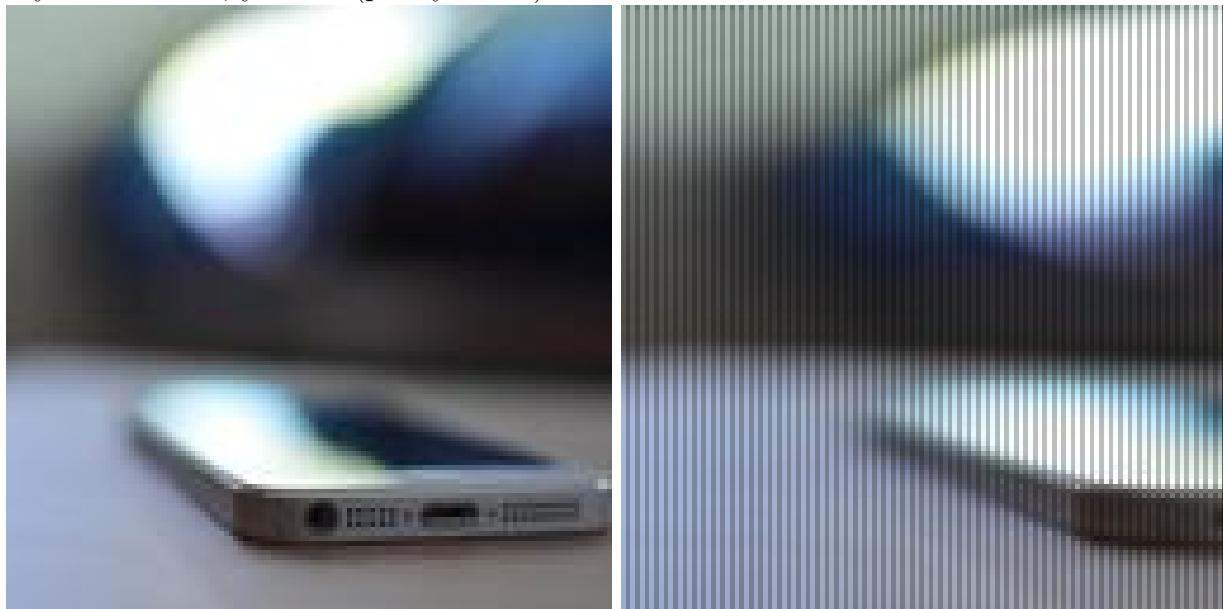
$$x_2 = S_x * x_1 \quad (5.6)$$

$$y_2 = S_y * y_1 \quad (5.7)$$

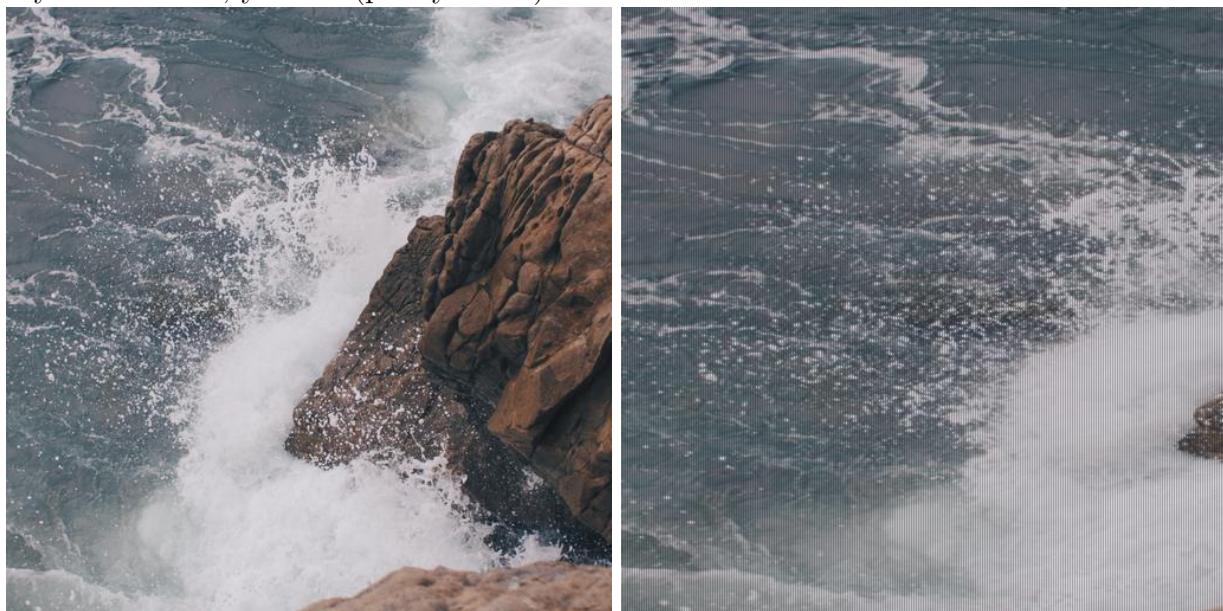
Przy czym skalowanie niejednorodne oznacza, że po zmianie wartości współrzędnych nasz obraz będzie miał zachwiane proporcje. Czyli:

$$S_x \neq S_y \quad (5.8)$$

Rysunek 5.5: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik $x = 2.0, y = 1.0$ (prawy obraz)



Rysunek 5.6: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik $x = 2.0, y = 1.0$ (prawy obraz)



Kod źródłowy algorytmu

```
def nonUniformScaling(self, scaleX = 1.0, scaleY = 1.0):
    print('Non-uniform scaling color image {} with scale ({},{}).format(self
        .decoder.name, scaleX, scaleY))
    image = self.decoder.getPixels()

    result = self._scale(image, scaleX, scaleY)
    self._interpolateColor(result)

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'nonuniform-
        scaling', False, self.decoder)

def _scale(self, matrix, scaleX, scaleY):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.full((height, width, 3), 1, numpy.uint8)
    for y in range(height):
        for x in range(width):
            if scaleY * y < height and scaleX * x < width:
                result[int(scaleY * y)][int(scaleX * x)] = matrix[y][x]
    return result
```

5.4 Obracanie obrazu o dowolny kąt

Opis

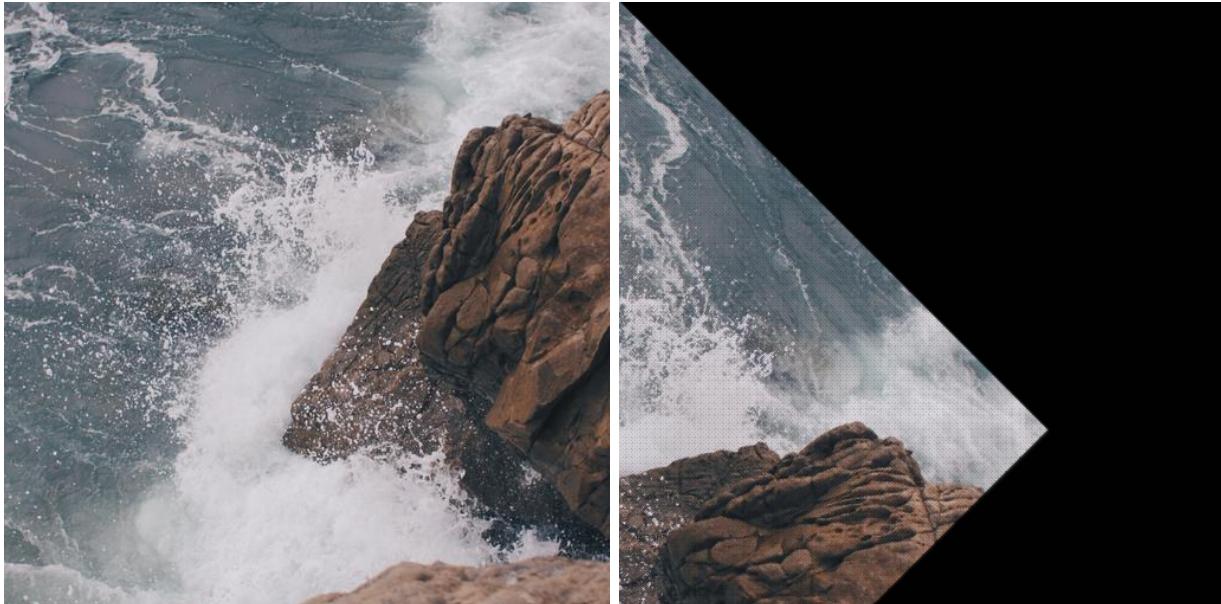
Operacja obrotu wykonywana jest wokół początku układu współrzędnych o kąt φ w taki sposób aby odległość od początku układu do punktu pozostała bez zmian, oraz aby pomiędzy odcinkami danych punków był kąt φ . Właściwości te można pozyskać dzięki wzorom:

$$x' = x \cos(\varphi) - y \sin(\varphi) \quad (5.9)$$

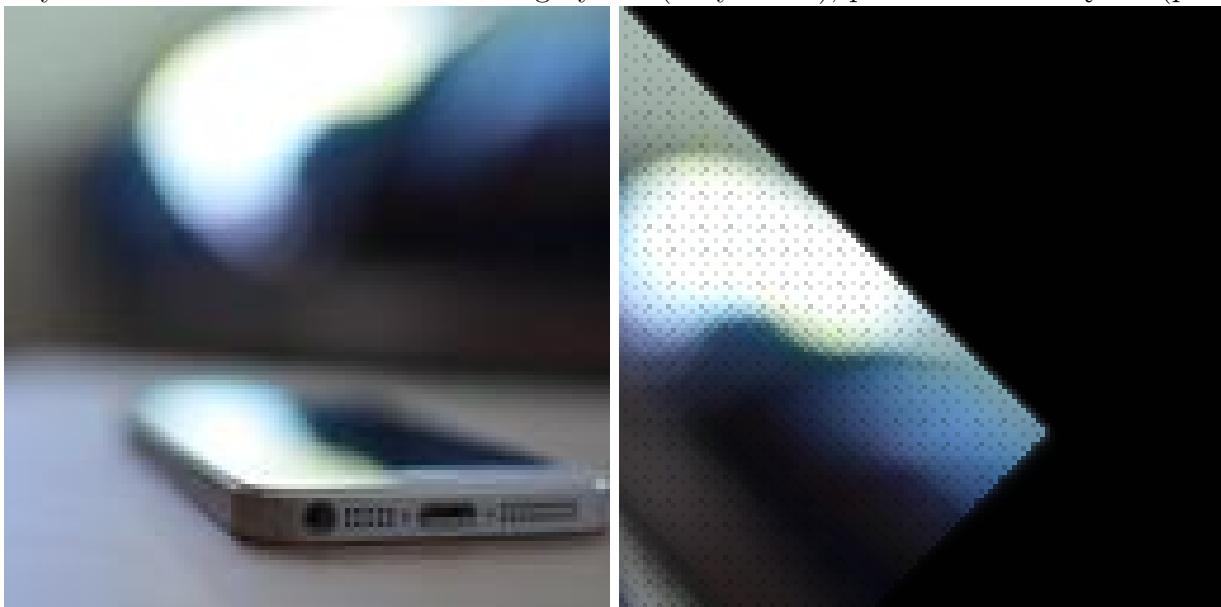
$$y' = x \sin(\varphi) + y \cos(\varphi) \quad (5.10)$$

gdzie (x', y') to nowe współrzędne wyznaczone po obrocie punktu (x, y) o kąt φ .

Rysunek 5.7: Przed uruchomieniem algorytmu (lewy obraz), po obróceniu o kąt 45° (prawy obraz)



Rysunek 5.8: Przed uruchomieniem algorytmu (lewy obraz), po obróceniu o kąt 45° (prawy obraz)



Kod źródłowy algorytmu

```
def rotation(self, phi):
    print('Rotation color image {} about angle {}'.format(self.decoder.name,
                                                            phi))
    image = self.decoder.getPixels()

    result = self._rotate(image, phi)
    self._interpolateColor(result)

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'rotate',
                     False, self.decoder)

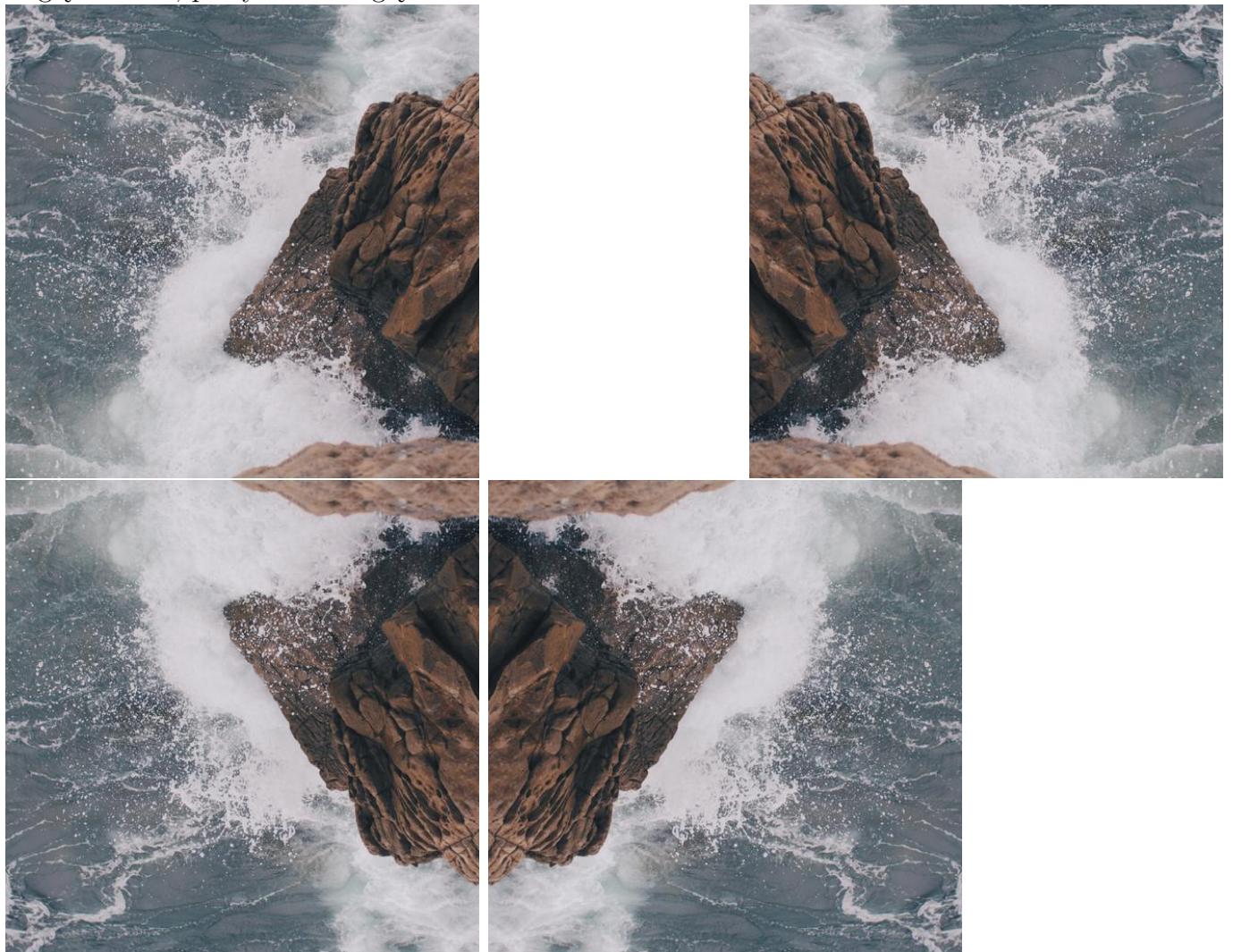
def _rotate(self, image, phi):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.full((height, width, 3), 1, numpy.uint8)
    radian = math.radians(phi)
    for y in range(height):
        for x in range(width):
            newX = x * math.cos(radian) - y * math.sin(radian)
            newY = x * math.sin(radian) + y * math.cos(radian)
            if newY < height and newY >= 0 and newX >= 0 and newX < width:
                result[int(newY)][int(newX)] = image[y][x]
    return result
```

5.5 Symetrie względem osi układu

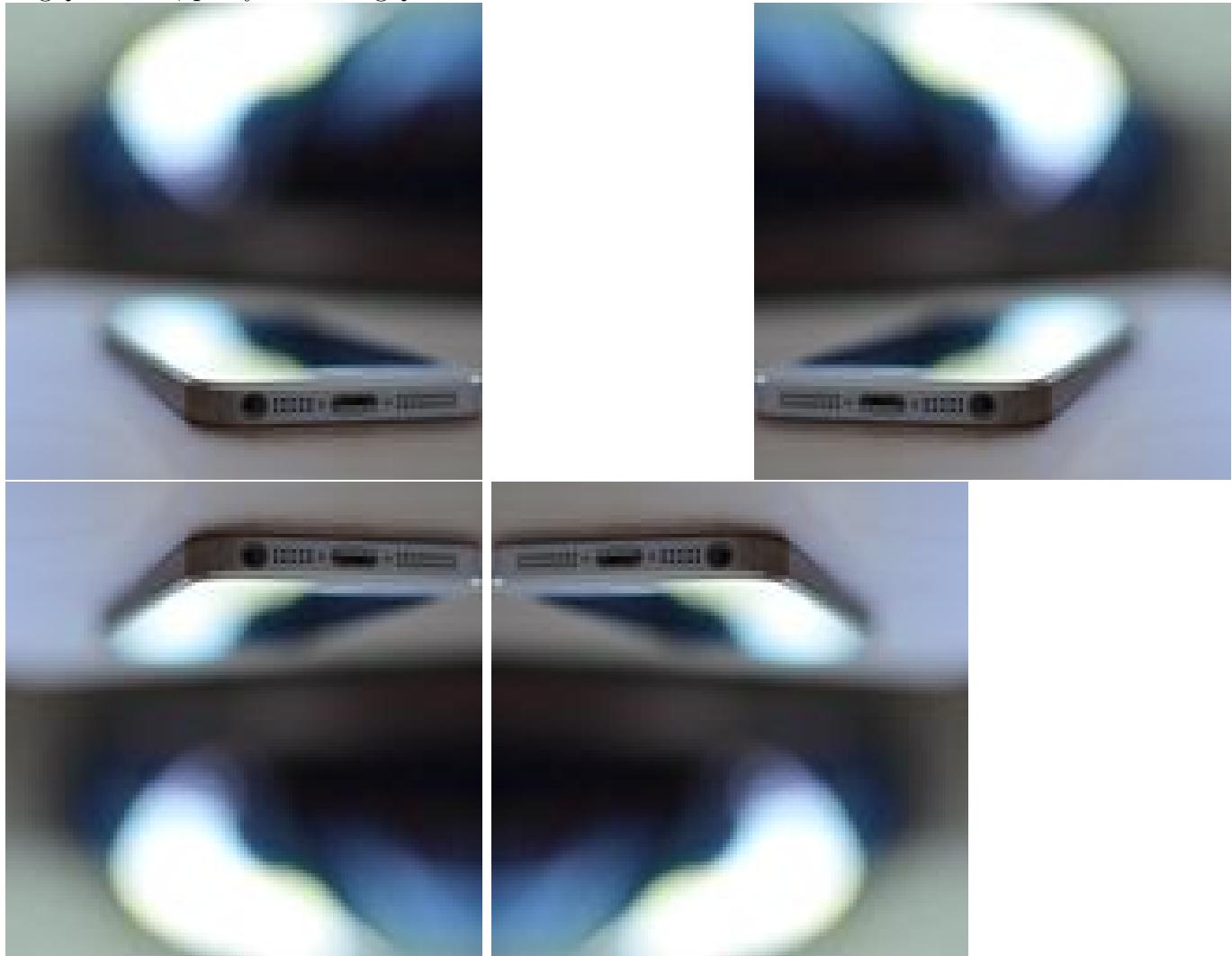
Opis

Symetriaosiowa względem osi OX lub OY sprawia, że punkt (x, y) zmienia się w $(x, -y)$ lub $(-x, y)$ w zależności czy symetria dotyczyła osi OX lub OY. W tej pracy przyjmuję, że lewa dolna krawędź obrazu znajduje się w punkcie $(0, 0)$.

Rysunek 5.9: Od lewej: przed uruchomieniem algorytmu, po symetrii względem OX, po symetrii względem OY, po symetrii względem OX oraz OY



Rysunek 5.10: Od lewej: przed uruchomieniem algorytmu, po symetrii względem OX, po symetrii względem OY, po symetrii względem OX oraz OY



Kod źródłowy algorytmu

```

def axisSymmetry(self, ox, oy):
    print('Axis symmetry color image {} on axis {} and {}'.format(self.
                                                                     decoder.name, ox, oy))
    image = self.decoder.getPixels()

    result = self._symmetryOXorOY(image, ox, oy)

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'axis-
symmetry-{}-{}'.format(ox, oy),
                  False, self.decoder)

def _symmetryOXorOY(self, image, ox, oy):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width, 3), numpy.uint8)
    for y in range(height):
        for x in range(width):
            if ox and not oy:
                result[y][x] = image[y][(width-1)-x]
            elif not ox and oy:
                result[y][x] = image[(height-1)-y][x]
            elif ox and oy:
                result[y][x] = image[(height-1)-y][(width-1)-x]
    return result

```


5.6 Symetrie względem zadanej prostej

Opis

Przypadek podobny do poprzedniego, lecz tym razem użytkownik podaje wiersz lub kolumnę względem której będzie przebiegała oś symetrii.

Rysunek 5.11: Przed uruchomieniem algorytmu (lewy), po symetrii względem prostej $x=356$ (środkowy), po symetrii względem prostej $y=356$ (prawy)



Rysunek 5.12: Przed uruchomieniem algorytmu (lewy), po symetrii względem prostej $x=64$ (środkowy), po symetrii względem prostej $y=64$ (prawy)



Kod źródłowy algorytmu

```

def customSymmetryX(self, ox):
    print('Custom axis symmetry X color image {} on axis {}'.format(self.decoder.name, ox))
    if not self._validateSymmetryAxisX(ox):
        return

    image = self.decoder.getPixels()
    height, width = self.decoder.height, self.decoder.width
    resultWidth = ox*2
    result = numpy.zeros((height, resultWidth, 3), numpy.uint8)
    for y in range(height):
        for x in range(ox):
            result[y][x] = image[y][x]

```

```
        result[y][resultWidth-1-x] = image[y][x]

ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'x-symmetry'
                  , False, self.decoder, None, ox)

def _validateSymmetryAxisX(self, ox):
    width = self.decoder.width
    if ox <= 0 or ox > width:
        return False
    return True

def customSymmetryY(self, oy):
    print('Custom axis symmetry Y color image {} on axis {}'.format(self.
        decoder.name, oy))
    if not self._validateSymmetryAxisY(oy):
        return

    image = self.decoder.getPixels()
    height, width = self.decoder.height, self.decoder.width
    resultHeight = oy*2
    result = numpy.zeros((resultHeight, width, 3), numpy.uint8)
    for y in range(oy):
        for x in range(width):
            result[y][x] = image[y][x]
            result[resultHeight-1-y][x] = image[y][x]

    ImageHelper.Save(result.astype(numpy.uint8), self.imageType, 'y-symmetry'
                  , False, self.decoder, None, oy)

def _validateSymmetryAxisY(self, oy):
    height = self.decoder.height
    if oy <= 0 or oy > height:
        return False
    return True
```

5.7 Wycinanie fragmentów obrazu

Opis

Wycięcie części obrazu jest zaimplementowane za pomocą kopowania pikseli do obrazu pomocniczego. Skopiowane zostają tylko te piksele, które znajdują się wewnątrz podanego zakresu.

Rysunek 5.13: Przed uruchomieniem algorytmu (lewy), po wycięciu kwadratu od piksela $(50, 50)$ do $(100, 100)$ (prawy)



Rysunek 5.14: Przed uruchomieniem algorytmu (lewy), po wycięciu kwadratu od piksela $(50, 50)$ do $(100, 100)$ (prawy)



Kod źródłowy algorytmu

```
def crop(self, (x1, y1), (x2, y2)):  
    print('Crop image {} from ({}, {}) to ({}, {})'.format(self.decoder.name,  
                                                       x1, y1, x2, y2))  
    image = self.decoder.getPixels()  
  
    for x in range(x1, x2+1):  
        for y in range(y1, y2+1):  
            image[y, x] = (0, 0, 0)
```

```
ImageHelper.Save(image, self.imageType, 'crop', False, self.decoder)
```

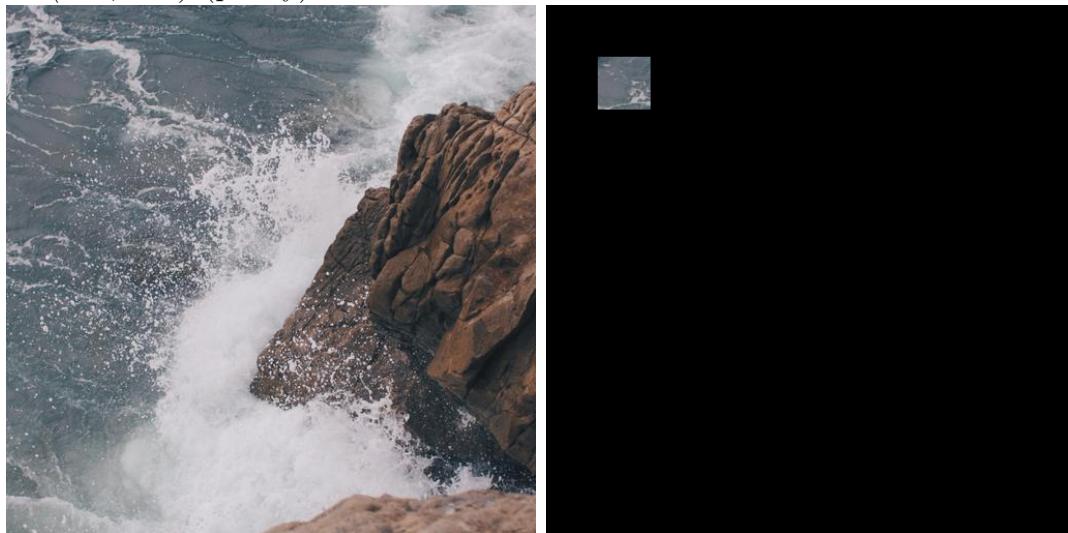
5.8 Kopiowanie fragmentów obrazów

Opis

Rysunek 5.15: Przed uruchomieniem algorytmu (lewy), po skopiowaniu kwadratu od piksela $(50, 50)$ do $(100, 100)$ (prawy)



Rysunek 5.16: Przed uruchomieniem algorytmu (lewy), po skopiowaniu kwadratu od piksela $(50, 50)$ do $(100, 100)$ (prawy)



Kod źródłowy algorytmu

```
def copy(self, (x1, y1), (x2, y2)):  
    print('Copy image {} from {} to {}'.format(self.decoder.name,  
                                              x1, y1, x2, y2))  
  
    image = self.decoder.getPixels()  
    height, width = self.decoder.height, self.decoder.width  
    result = numpy.zeros((height, width, 3), numpy.uint8)  
  
    for x in range(x1, x2+1):  
        for y in range(y1, y2+1):  
            result[y, x] = image[y, x]  
  
    ImageHelper.Save(result, self.imageType, 'copy', False, self.decoder)
```


Rozdział 6

Operacje na histogramie obrazu szarego

Histogram obrazu to typ histogramu, który na osi poziomej przyjmuje rozpiętość tonalną obrazu szarego, a na pionowej zliczone występowania wartości tych tonów. Dzięki temu zabiegowi można na jednym wykresie przedstawić rozkład kolorów danego obrazu i odczytać dane o jego jasności i intensywności. Umożliwia to świadome manipulowanie jego cechami w celu ulepszenia obrazu (zmiana jasności lub kontrastu z pomocą normalizacji) lub uwydatnienia jego cech (na przykład przy progowaniu obrazu).

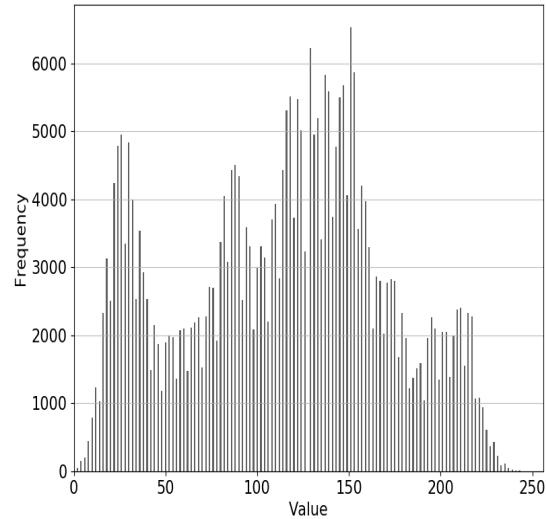
6.1 Obliczanie histogramu

Algorytm

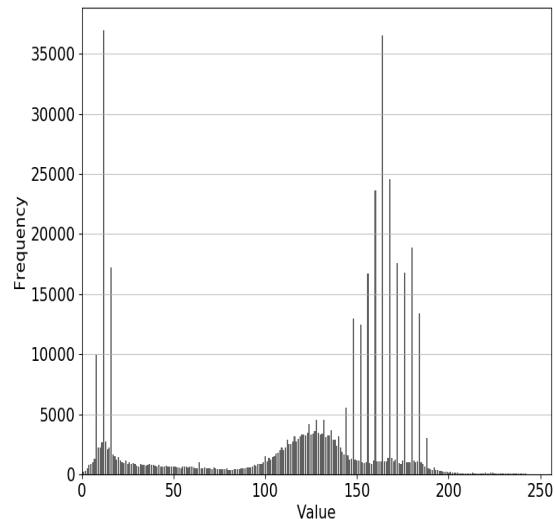
Opis

Przy obliczaniu wartości histogramu należy wziąć pod uwagę zakres jaki mogą przyjmować piksele. Przy obrazie szarym zakres ten wynosi od 0 do 255. Dla każdej liczby całkowitej w tym zakresie jest zliczana ilość jej wystąpień w obrazie co daje nam pełny pogląd jakie wartości w nim dominują. Na rysunku 6.1 można zaobserwować dominujące wartości. Są to głównie jasne odcienie szarości widoczne w tle obrazu, ale można też zobaczyć, że histogram posiada jeden słupek przewyższający inne w ciemniejszym zakresie odcieni - jest to mężczyzna przedstawiony na pierwszym planie fotografii. Natomiast rysunek 6.1 jest bardziej stonowany pod względem rozłożenia pikseli na skali szarości. Ta operacja jest wstępem do następnych ponieważ samo obliczenie histogramu daje inny pogląd na analizę obrazu, ale nie wpływa bezpośrednio na jego cechy. Dopiero operacje świadome zmieniające wartości histogramu są użyteczne.

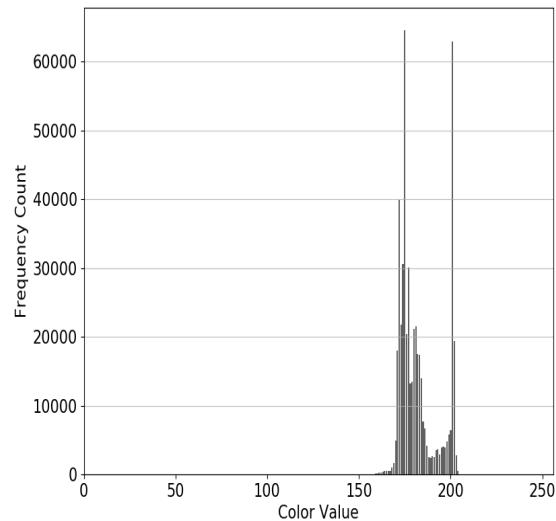
Rysunek 6.1: Obraz bazowy i jego obliczony histogram



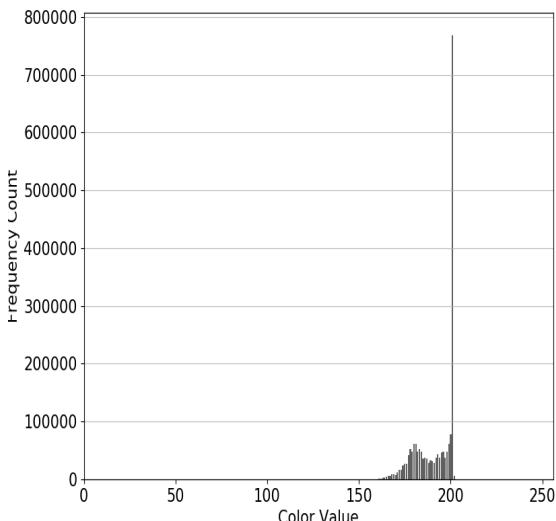
Rysunek 6.2: Obraz bazowy i jego obliczony histogram



Rysunek 6.3: Obraz bazowy i jego obliczony histogram



Rysunek 6.4: Obraz bazowy i jego obliczony histogram



Kod źródłowy programu

```

def CalculateGrayHistogram(image, height, width):
    maxValue = numpy.iinfo(image.dtype).max
    histogram = numpy.zeros(maxValue+1, numpy.uint32)
    for h in range(height):
        for w in range(width):
            bin = image[h, w]
            histogram[bin] += 1

    bins = numpy.arange(maxValue+2).astype(numpy.uint32)
    return bins, histogram

```

6.2 Przemieszczenie histogramu

Wstęp

Przemieszczenie histogramu jest efektem po zastosowaniu operatora punktowego, który dodaje lub odejmuje stałą wartość od każdego piksela. Wykres histogramu w tym wypadku przemieszcza się wzdłuż osi OX, zawierającej zakres wartości tonalnej obrazu.

Kod algorytmu

```
def moveHistogram(self, constValue):
    print('Move histogram in gray image {} about {}'.format(self.firstDecoder
                                                               .name, constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = float(numpy.iinfo(image.dtype).max)
    minValue = float(numpy.iinfo(image.dtype).min)
    result = numpy.ones((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            computed = int(image[h, w]) + constValue
            computed = maxValue if computed > maxValue else computed
            computed = minValue if computed < minValue else computed
            result[h, w] = computed

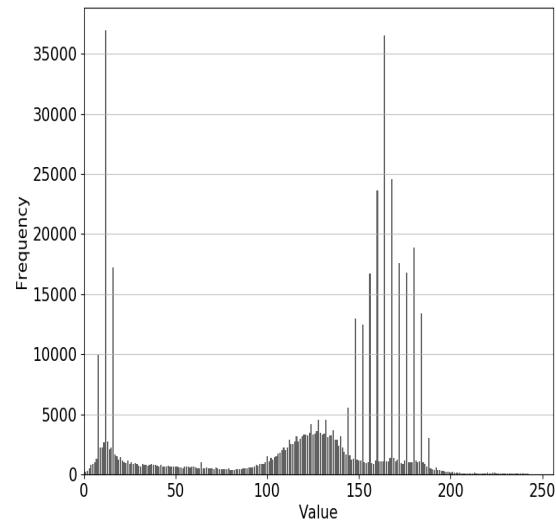
    ImageHelper.Save(result, self.imageType, 'move-histogram-image', False,
                     self.firstDecoder, None,
                     constValue)
    bins, histogram = Commons.CalculateGrayHistogram(result, height, width)
    ImageHelper.SaveGrayHistogram(bins, histogram, 'move-histogram', False,
                                 self.firstDecoder, None,
                                 constValue)
```

Wynik algorytmu

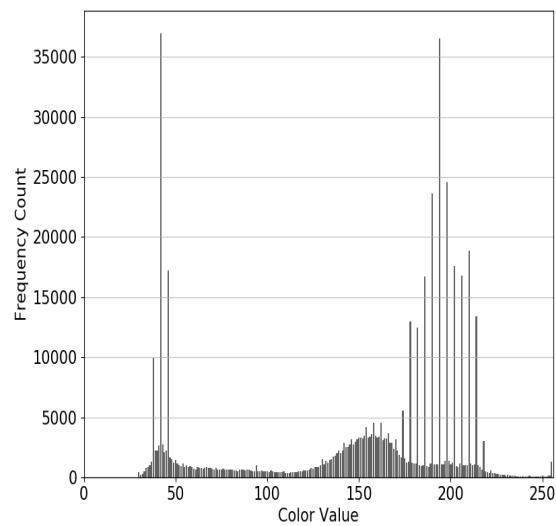
Rozjaśnienie obrazu spowodowało przesunięcie histogramu w prawą stronę, a przyciemnienie go spowodowało efekt odwrotny.

W miejscu gdzie wartości chcieli wyjść poza dostępny zakres można zaobserwować znaczący wzrost występowania pikseli o wartościach granicznych (0 i 255).

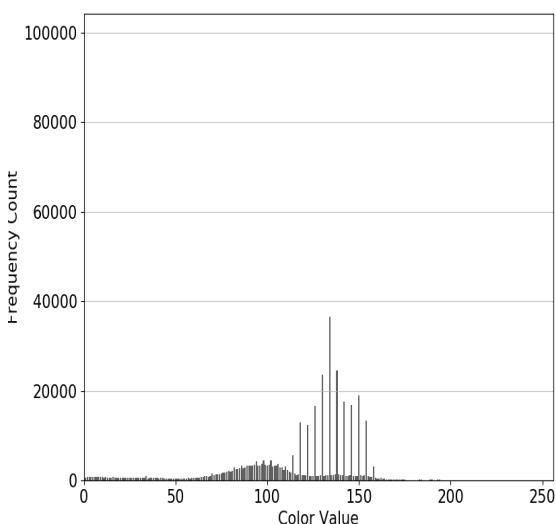
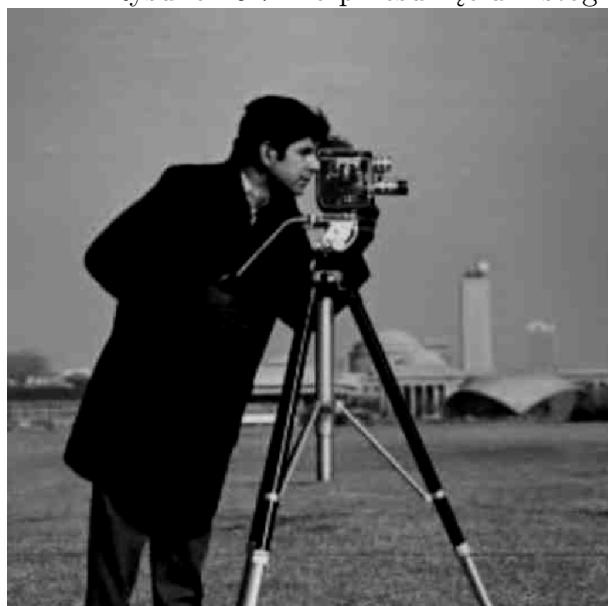
Rysunek 6.5: Obraz bazowy i jego obliczony histogram



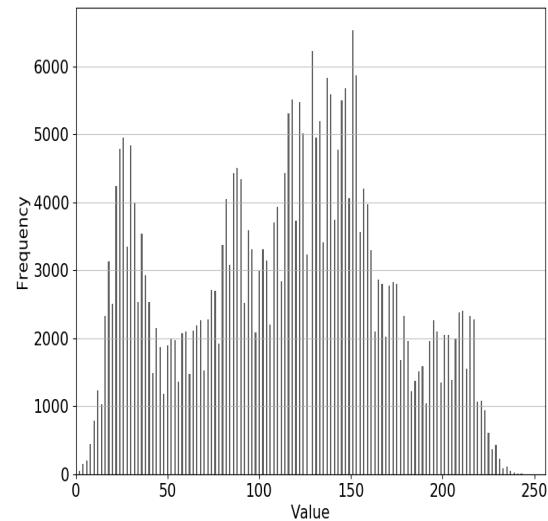
Rysunek 6.6: Po przesunięciu histogramu w górę o 30 wraz z jego obliczonym histogramem



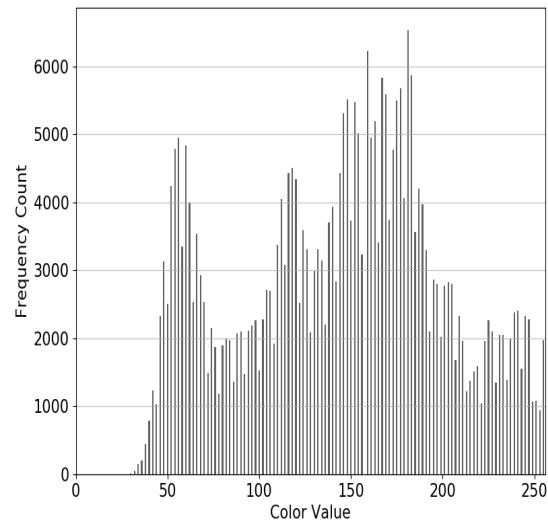
Rysunek 6.7: Po przesunięciu histogramu w dół o 30 wraz z jego obliczonym histogramem



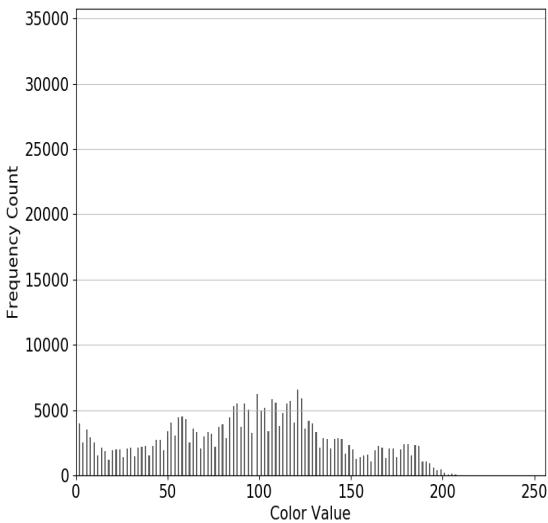
Rysunek 6.8: Obraz bazowy i jego obliczony histogram



Rysunek 6.9: Po przesunięciu histogramu w góre o 30 wraz z jego obliczonym histogramem



Rysunek 6.10: Po przesunięciu histogramu w dół o 30 wraz z jego obliczonym histogramem



6.3 Rozciąganie histogramu

Wstęp

Wyróżnia się dwie metody **skalowania histogramu**, mogące wpływać na jego kształt. Pierwszą z nich jest **skalowanie pionowe**, które rozciąga histogram z użyciem operatora mnożenia obrazu przez skalar większy od jedności lub zwęźa histogram gdy użyjemy operatora mnożenia z wartością mniejszą od jedności. Drugą jest **skalowanie poziome**, które przekształci wąski zakres wartości tonalnych do jego odpowiednika o pełnym zakresie.

Na przykład jeśli w pewnym obrazie najmniejsza wartość (*min*) wynosi 60, a największa (*max*) wynosi 180 rozciągnięcie histogramu z pomocą *normalizacji* przekształci zbiór [60, 180] na [0, MAX_{ton}], czyli w tym wypadku [0, 255].

Kod algorytmu

```
def extendHistogram(self):
    print('Extend histogram in gray image {}'.format(self.firstDecoder.name))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    result = Commons.Normalization(image, image)

    ImageHelper.Save(result, self.imageType, 'extend-histogram-image', False,
                     self.firstDecoder)
    bins, histogram = Commons.CalculateGrayHistogram(result, height, width)
    ImageHelper.SaveGrayHistogram(bins, histogram, 'extend-histogram', False,
                                  self.firstDecoder)
```

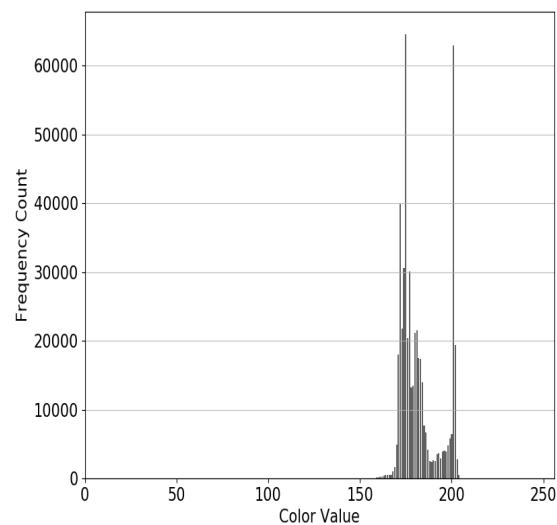
Używałem tej samej funkcji *normalizacji* co w innych przykładach:

```
def Normalization(image, result):
    maxValue = numpy.iinfo(image.dtype).max
    fmin = numpy.amin(result)
    fmax = numpy.amax(result)
    result = result.astype(numpy.float32)
    result = maxValue * ((result - fmin) / (fmax - fmin))
    result = result.astype(numpy.uint8)
    return result
```

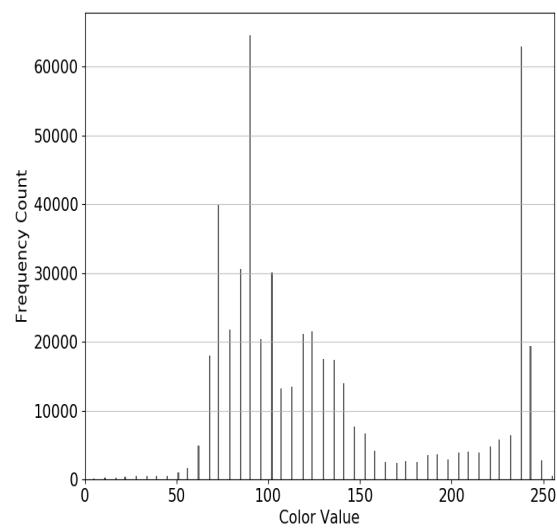
Wynik algorytmu

Na przedstawionych rysunkach 6.3 i 6.3 widać zwiększenie się poziomu kontrastu wraz z rozszerzeniem się histogramu i wykorzystaniem większej gradacji barw. Funkcja *normalizacji* zostawiła tą samą ilość słupków, ale zmieniła ich rozmieszczenie na osi OX. Dzięki temu na zdjęciach wynikowych widać nawet odcienie czerni i brudnej bieli, których obrazy bazowe nie posiadają.

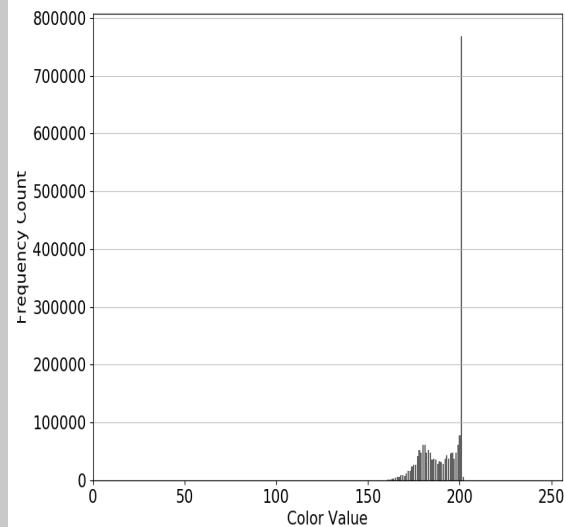
Rysunek 6.11: Obraz bazowy i jego obliczony histogram



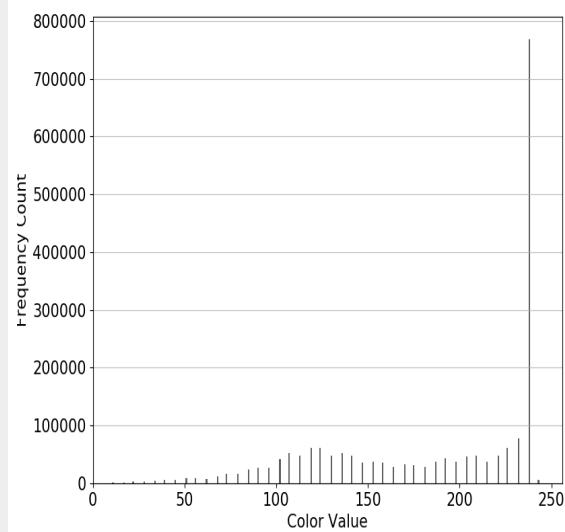
Rysunek 6.12: Po rozszerzeniu histogramu z pomocą normalizacji



Rysunek 6.13: Obraz bazowy i jego obliczony histogram



Rysunek 6.14: Po rozszerzeniu histogramu z pomocą normalizacji



6.4 Progowanie lokalne

Wstęp

Progowanie jest operatorem służącym do konwersji obrazu szarego lub kolorowego do jego binarnej formy. Jest to też rodzaj **segmentacji obrazu**, w którym próg zależy od wartości danego piksela $f(x, y)$ i od jego lokalnej właściwości $p(x, y)$.

Obrazy binarne posiadają tylko dwie wartości *prawdę* i *fałsz*, więc aby zmienić macierz w której komórki mogą przyjmować więcej niż dwie wartości niezbędne jest ustalenie **progu jasności**. Liczby większe od progu będą przyjmowały wartość oznaczoną jako *prawda*, a mniejsze bądź równe otrzymają *fałsz*. Inaczej można też powiedzieć, że wartości mniejsze od progu będą tłem, a większe obiektem. W świecie obrazów szarych *prawdą* będzie wartość maksymalna MAX_{ton} czyli 255, a *fałszem* wartość minimalna MIN_{ton} czyli 0.

Przy **progowaniu lokalnym** dla każdego piksela ustala się *próg jasności* indywidualnie. W tym przykładzie użyję implementacji progowania Bernsen'a.

Progowanie Bernsen'a używa w swojej metodzie jednej z wartości podanej przez użytkownika - *kontrastu progowego* l . Jeżeli *kontrast lokalny* ($f_{high} - f_{low}$) jest większy bądź równy *kontrastowi progowemu*, wtedy próg $T(x, y)$ jest ustawiany do średniej wartości maksymalnego i minimalnego elementu w oknie. W przypadku gdy prawdziwe jest stwierdzenie $C(x, y)$ całe sąsiedztwo piksela uznaje się jako obiekt lub tło w zależności od $T(x, y) \geq 128$.

$$T(x, y) = \frac{f_{low} + f_{high}}{2} \quad (6.1)$$

$$C(x, y) = f_{high} - f_{low} < l \quad (6.2)$$

Kod algorytmu

```

def localThresholding(self, contrastThreshold, windowSize=3):
    print('Local thresholding gray image {} with contrast threshold of {}'.format(self.firstDecoder.name, contrastThreshold))

    if windowSize % 2 == 0:
        raise ValueError("Window size can't be even")

    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = int(numpy.iinfo(image.dtype).max)
    minValue = int(numpy.iinfo(image.dtype).min)
    if contrastThreshold <= minValue or contrastThreshold >= maxValue:
        raise ValueError("Contrast threshold has to be in range of ({},{}).".format(minValue, maxValue))

    result = numpy.zeros((height, width), numpy.uint8)
    overlap = int(math.ceil(windowSize/2))
    for h in range(height):
        for w in range(width):
            minH = 0 if h-overlap < 0 else h-overlap
            maxH = height if h+overlap+1 > height else h+overlap+1
            minW = 0 if w-overlap < 0 else w-overlap
            maxW = width if w+overlap+1 > width else w+overlap+1
            if minH >= maxH or minW >= maxW:
                continue

            window = image[minH:maxH, minW:maxW]
            localMin = numpy.amin(window)
            localMax = numpy.amax(window)

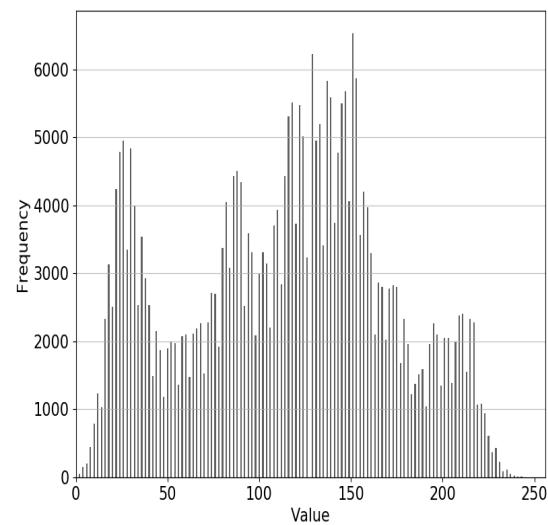
```

```
localContrast = localMax-localMin
midGray = (int(localMax)+int(localMin))/2
if localContrast < contrastThreshold:
    if midGray >= maxValue/2:
        result[minH:maxH, minW:maxW] = numpy.full((maxH-minH,
                                                       maxW-minW),
                                                       maxValue)
    else:
        result[minH:maxH, minW:maxW] = numpy.full((maxH-minH,
                                                       maxW-minW),
                                                       minValue)
else:
    if image[h, w] >= midGray:
        result[h, w] = maxValue
    else:
        result[h, w] = minValue

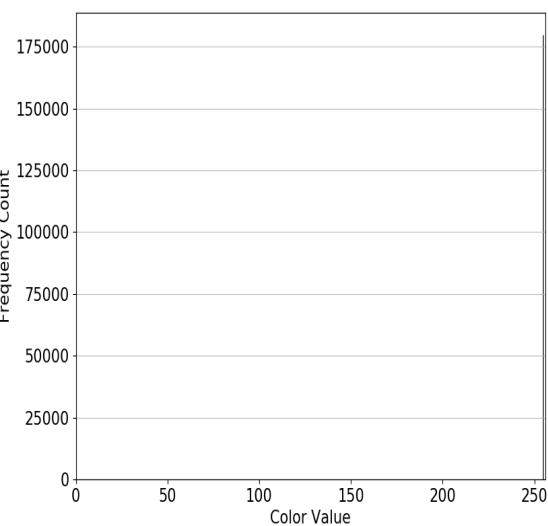
ImageHelper.Save(result, self.imageType, 'local-threshold-image', False,
                 self.firstDecoder, None,
                 contrastThreshold)
bins, histogram = Commons.CalculateGrayHistogram(result, height, width)
ImageHelper.SaveGrayHistogram(bins, histogram, 'local-threshold', False,
                             self.firstDecoder, None,
                             contrastThreshold)
```

Wynik algorytmu

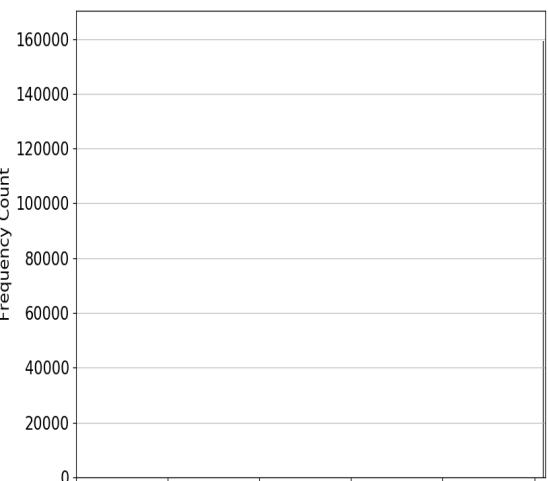
Rysunek 6.15: Obraz bazowy i jego obliczony histogram



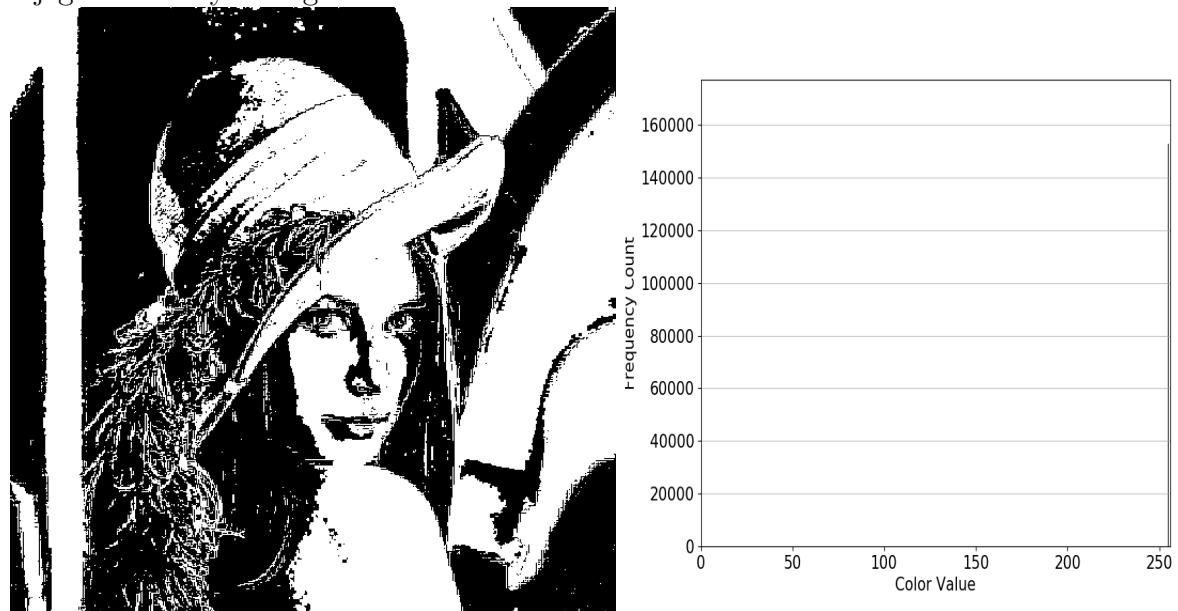
Rysunek 6.16: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 5 wraz z jego obliczony histogram



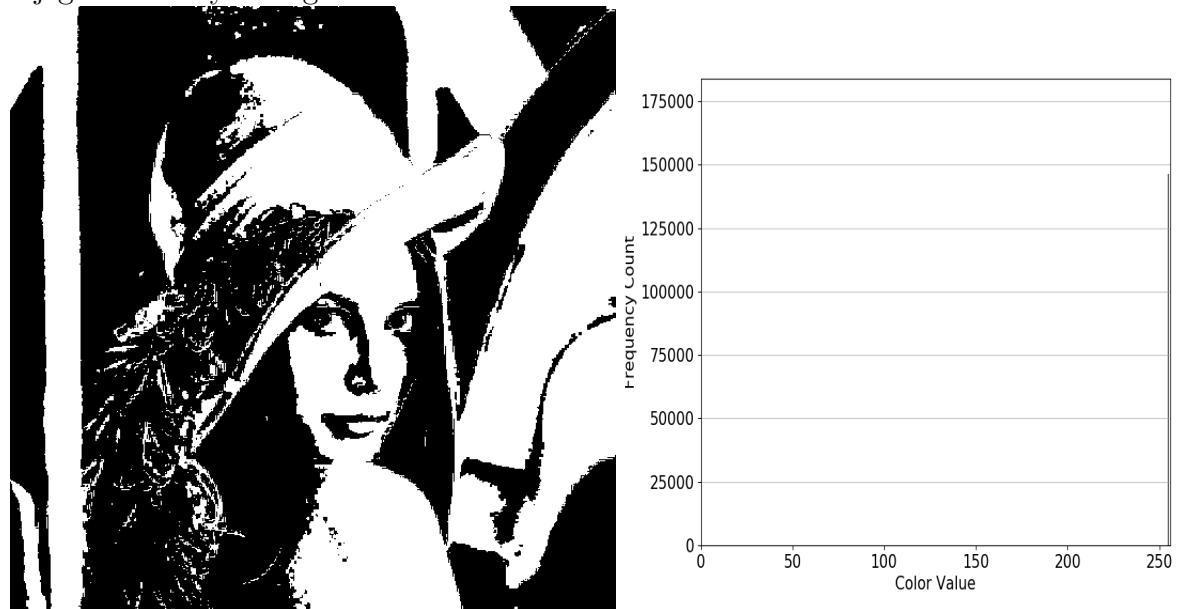
Rysunek 6.17: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 15 wraz z jego obliczony histogram



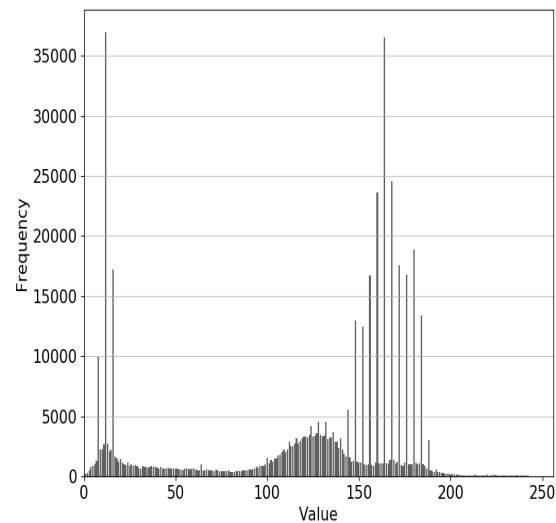
Rysunek 6.18: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 25 wraz z jego obliczony histogram



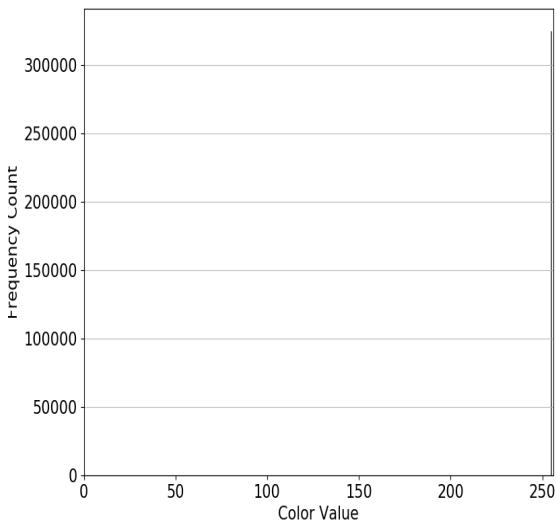
Rysunek 6.19: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 50 wraz z jego obliczony histogram



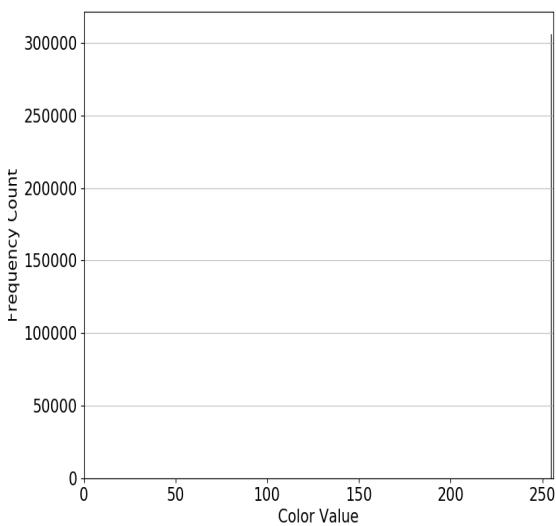
Rysunek 6.20: Obraz bazowy i jego obliczony histogram



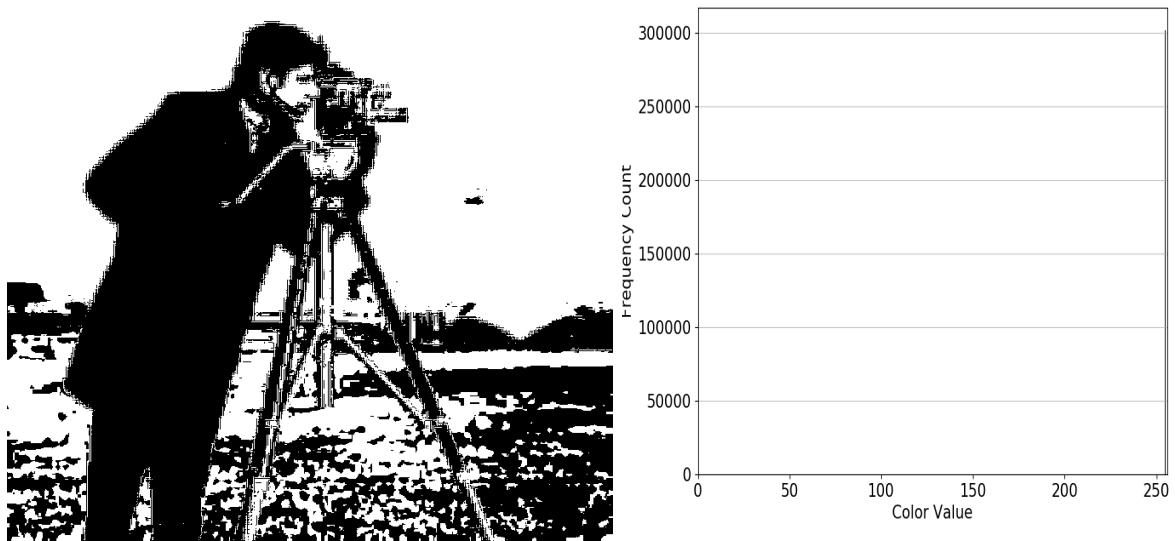
Rysunek 6.21: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 5 wraz z jego obliczony histogram



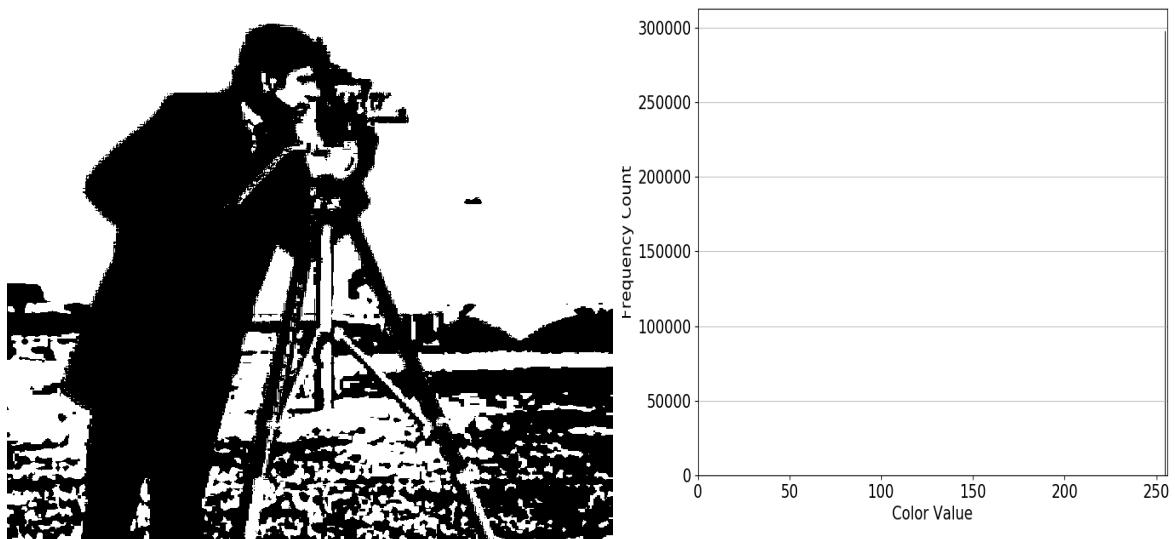
Rysunek 6.22: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 15 wraz z jego obliczony histogram



Rysunek 6.23: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 25 wraz z jego obliczony histogram



Rysunek 6.24: Obraz binarny po zastosowaniu progowania Bernstena o kontraście progowym 50 wraz z jego obliczony histogram



6.5 Progowanie globalne

Wstęp

Dla **progowania globalnego** ustala się jeden *próg jasności* T , które ma zastosowanie do wszystkich pikseli w obrazie. Wtedy *progiem jasności* nazywamy wartość, która będzie klinem wbijanym w histogram obrazu dzieląc go na dwie części - część obiektów i tła.

Dla wszystkich wartości:

1. $f(x, y) \geq T$ otrzymają wartość obiektu
2. $f(x, y) < T$ otrzymają wartość tła

Kod algorytmu

```

def globalThresholding(self, threshold):
    print('Global thresholding gray image {} with threshold value of {}'.format(
        self.firstDecoder.name, threshold))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = int(numpy.iinfo(image.dtype).max)
    minValue = int(numpy.iinfo(image.dtype).min)
    if threshold <= minValue or threshold >= maxValue:
        raise ValueError("Threshold has to be in range of [{},{}].format(
            minValue, maxValue)")

    result = numpy.zeros((height, width), numpy.uint8)
    for h in range(height):
        for w in range(width):
            result[h, w] = maxValue if image[h, w] >= threshold else minValue

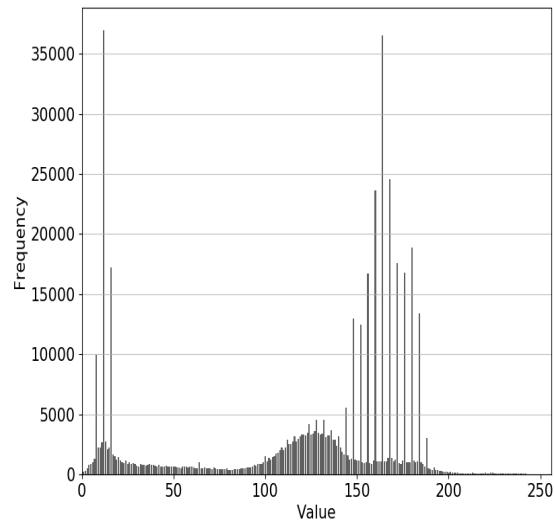
    ImageHelper.Save(result, self.imageType, 'global-threshold-image', False,
                     self.firstDecoder, None,
                     threshold)
    bins, histogram = Commons.CalculateGrayHistogram(result, height, width)
    ImageHelper.SaveGrayHistogram(bins, histogram, 'global-threshold', False,
                                 self.firstDecoder, None,
                                 threshold)

```

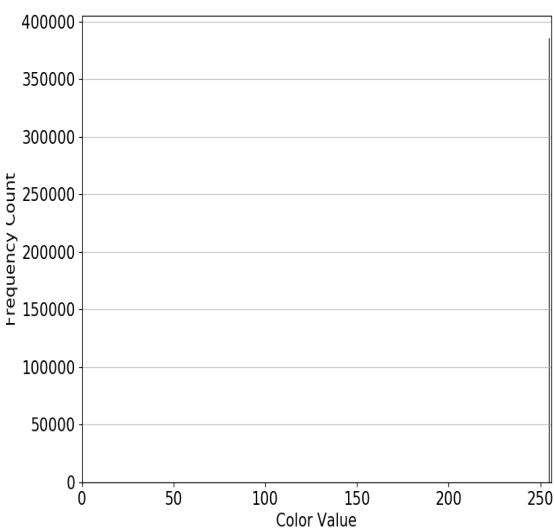
Wynik algorytmu

Można zaobserwować zależność pomiędzy wielkością wartości progowej T a ilością detali w obrazie binarnym. Im wyższe T tym więcej detali, co również oznacza wyższy słupek koloru czarnego na histogramie.

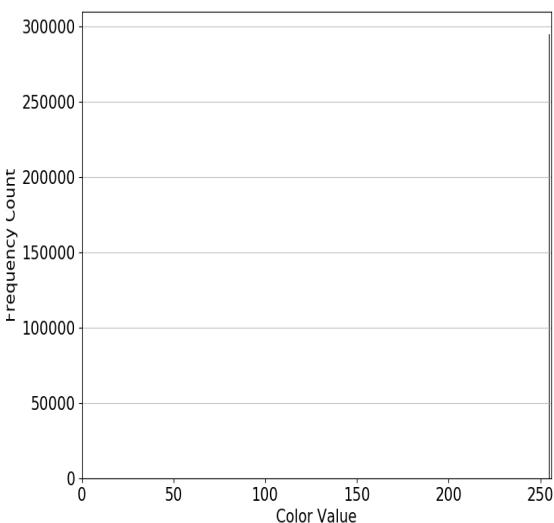
Rysunek 6.25: Obraz bazowy i jego obliczony histogram



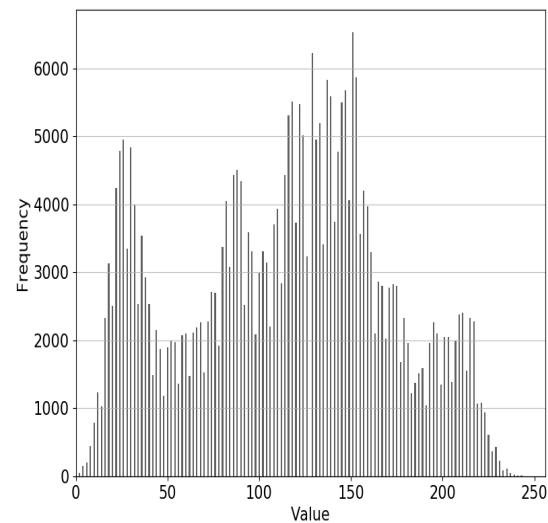
Rysunek 6.26: Obraz binarny po zastosowaniu progowania globalnego o wartości 64 wraz z jego obliczony histogram



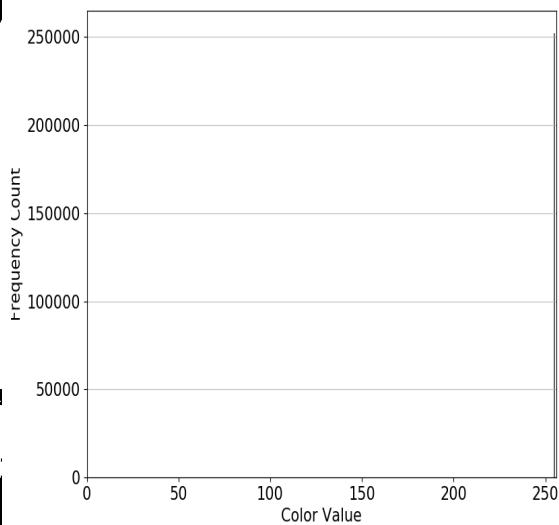
Rysunek 6.27: Obraz binarny po zastosowaniu progowania globalnego o wartości 128 wraz z jego obliczony histogram



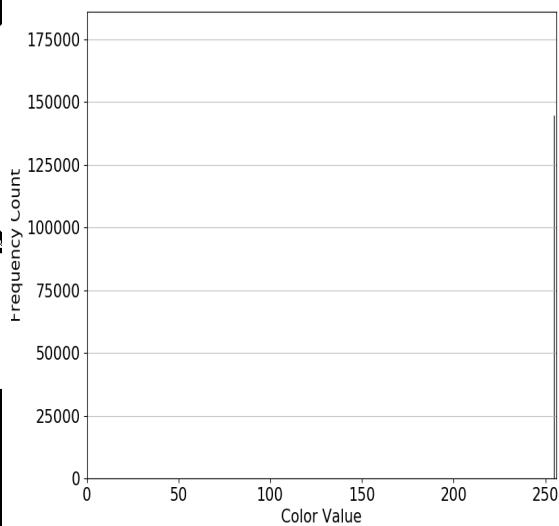
Rysunek 6.28: Obraz bazowy i jego obliczony histogram



Rysunek 6.29: Obraz binarny po zastosowaniu progowania globalnego o wartości 64 wraz z jego obliczony histogram



Rysunek 6.30: Obraz binarny po zastosowaniu progowania globalnego o wartości 128 wraz z jego obliczony histogram



Rozdział 7

Operacje na histogramie obrazu barwowego

Zasadniczą różnicą w przedstawieniu histogramu obrazu barwowego jest ilość kanałów, które trzeba przetworzyć w celu otrzymania ilości wystąpień różnych tonów barw. Dla każdej barwy RGB należy bowiem z osobna obliczyć histogram w celu wyciągnięcia pełnej informacji o stanie barw obrazu jak i również samo przetwarzanie musi się odbywać z uwzględnieniem tych trzech barw.

Ilość barw w typ wypadku jest sposobnością, aby użyć *progowania wielo-progowego*, gdyż na każdą barwę będzie mógł przypadać jeden próg.

7.1 Obliczanie histogramu

Wstęp

Obliczenie histogramu obrazu barwnego odbywa się poprzez zliczenie ilości występowania każdej wartości tonalnej dla każdej z trzech barw RGB. Oznacza to, że dla barwy czerwonej będziemy liczyć histogram osobno tak jakbyśmy to robili dla obrazu szarego, z tym wyjątkiem że będzie to powtórzyć również dla koloru zielonego i niebieskiego.

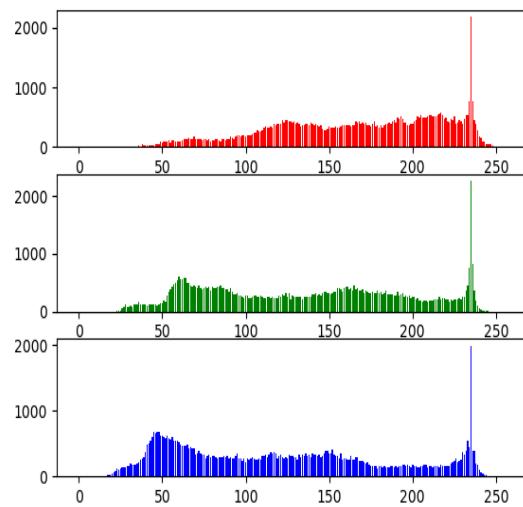
Kod algorytmu

```
def CalculateColorHistogram(image, height, width):
    maxValue = numpy.iinfo(image.dtype).max
    histogram = numpy.zeros((3, maxValue+1), numpy.uint32)
    for h in range(height):
        for w in range(width):
            r = image[h, w, 0]
            g = image[h, w, 1]
            b = image[h, w, 2]
            histogram[0, r] += 1
            histogram[1, g] += 1
            histogram[2, b] += 1

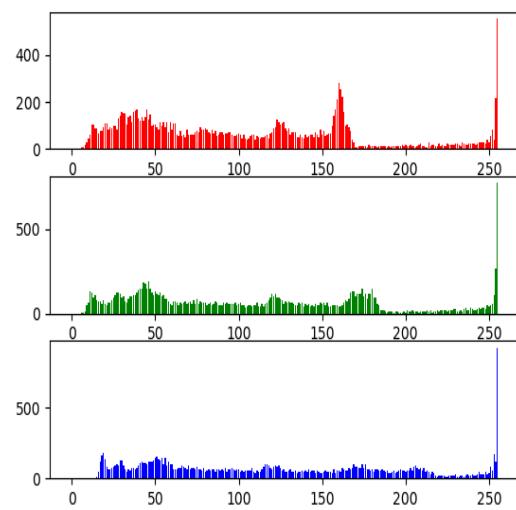
    bins = numpy.arange(maxValue+1).astype(numpy.uint32)
    return bins, histogram
```

Wynik algorytmu

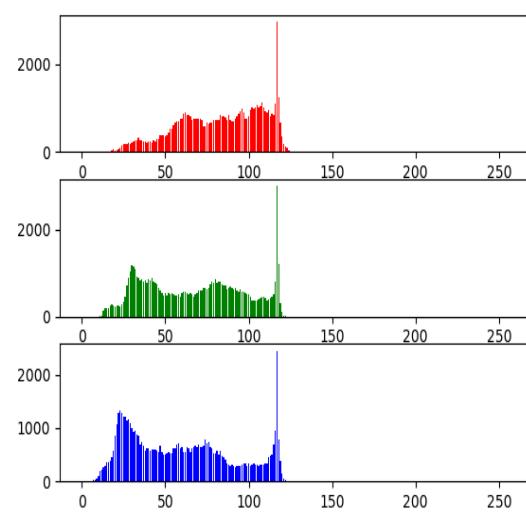
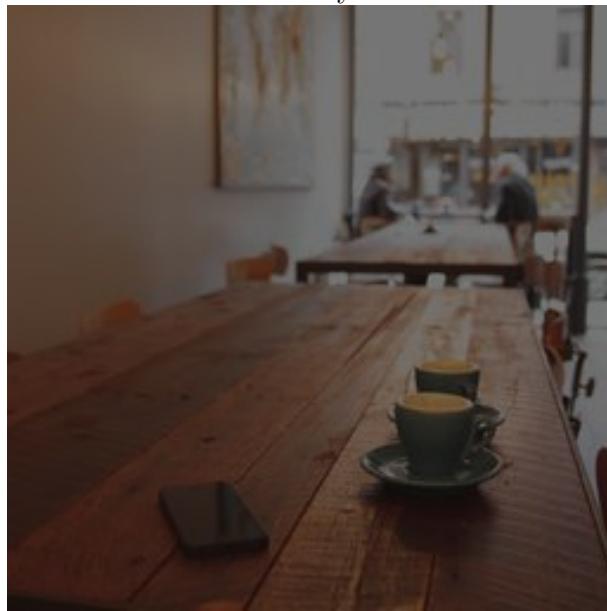
Rysunek 7.1: Obraz bazowy i jego obliczony histogram



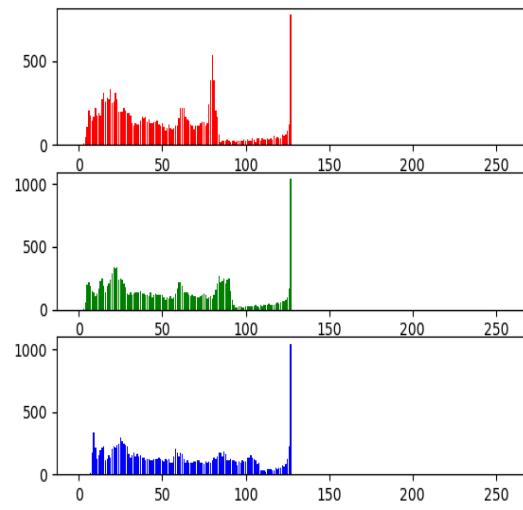
Rysunek 7.2: Obraz bazowy i jego obliczony histogram



Rysunek 7.3: Obraz bazowy i jego obliczony histogram



Rysunek 7.4: Obraz bazowy i jego obliczony histogram



7.2 Przemieszczanie histogramu

Wstęp

Przemieszczanie histogramu odbywa się poprzez zmniejszenie lub zwiększenie wartości dla wszystkich pikseli w obrazie. W ramach piksela mogą zostać zmodyfikowane wartości z trzech barw RGB. Zwiększenie wartości dla trzech barw przyciemni obraz, a zmniejszenie rozjaśni. Przesuwanie histogramu może zostać też zastosowane do pojedynczych barw RGB, gdzie zwiększenie jednej z nich spowoduje dominację przypisanego koloru w pikselach co będzie skutkowało zmianą kolorów pikseli.

Kod algorytmu

```

def moveHistogram(self, constValue=(0,0,0)):
    print('Move histogram in color image {} about {}'.format(self.
                                                               firstDecoder.name, constValue))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
    maxValue = float(numpy.iinfo(image.dtype).max)
    minValue = float(numpy.iinfo(image.dtype).min)
    result = numpy.ones((height, width, 3), numpy.uint8)
    for h in range(height):
        for w in range(width):
            computedR = int(image[h, w, 0]) + constValue[0]
            computedR = maxValue if computedR > maxValue else computedR
            computedR = minValue if computedR < minValue else computedR
            result[h, w, 0] = computedR
            computedG = int(image[h, w, 1]) + constValue[1]
            computedG = maxValue if computedG > maxValue else computedG
            computedG = minValue if computedG < minValue else computedG
            result[h, w, 1] = computedG
            computedB = int(image[h, w, 2]) + constValue[2]
            computedB = maxValue if computedB > maxValue else computedB
            computedB = minValue if computedB < minValue else computedB
            result[h, w, 2] = computedB

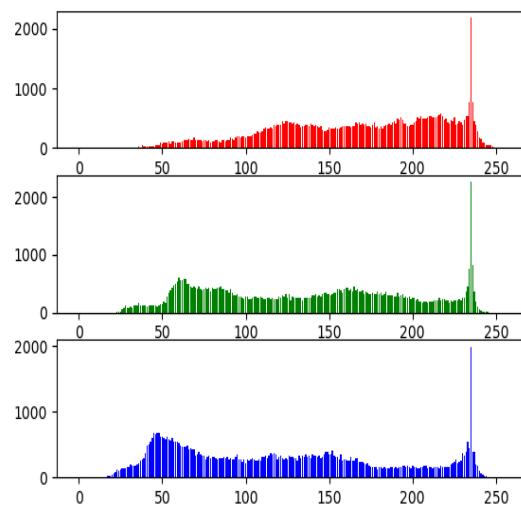
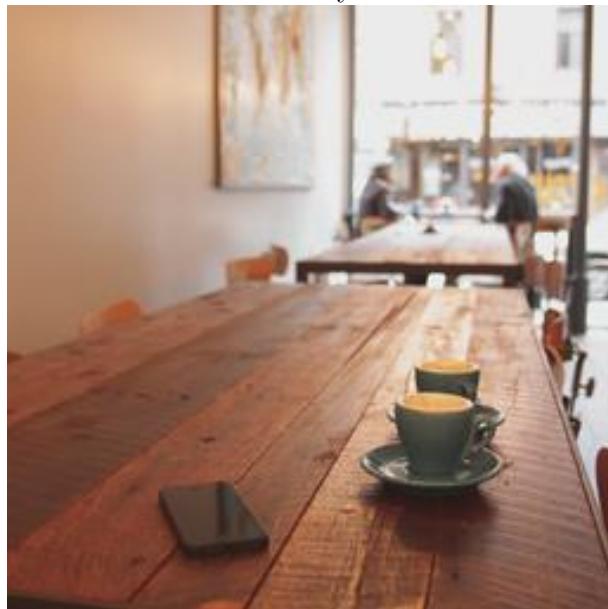
    ImageHelper.Save(result, self.imageType, 'move-histogram-image', False,
                     self.firstDecoder, None,
                     constValue)

```

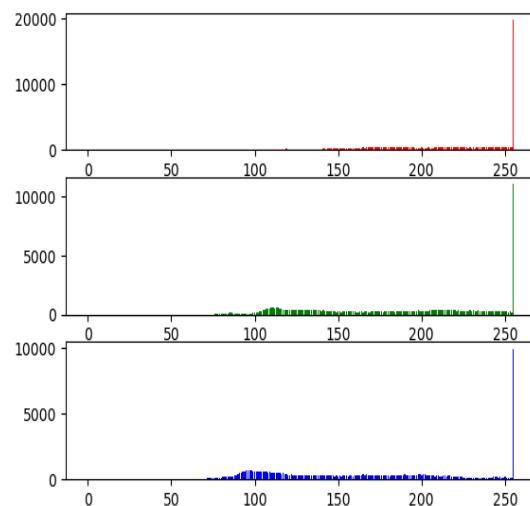
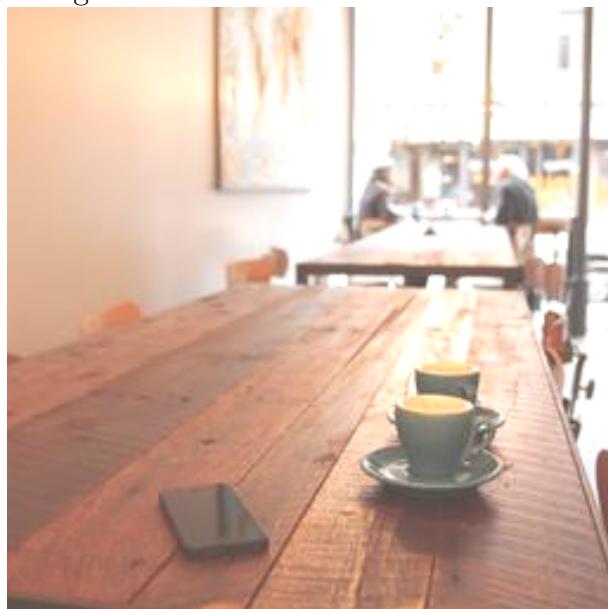
```
bins, histogram = Commons.CalculateColorHistogram(result, height, width)
ImageHelper.SaveColorHistogram(bins, histogram, 'move-histogram', self.
                                firstDecoder, constValue)
```

Wynik algorytmu

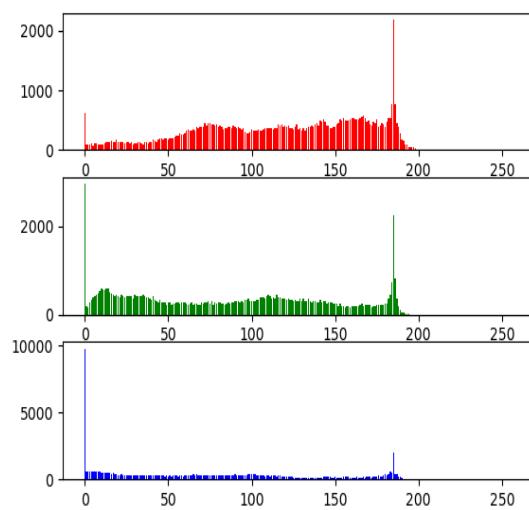
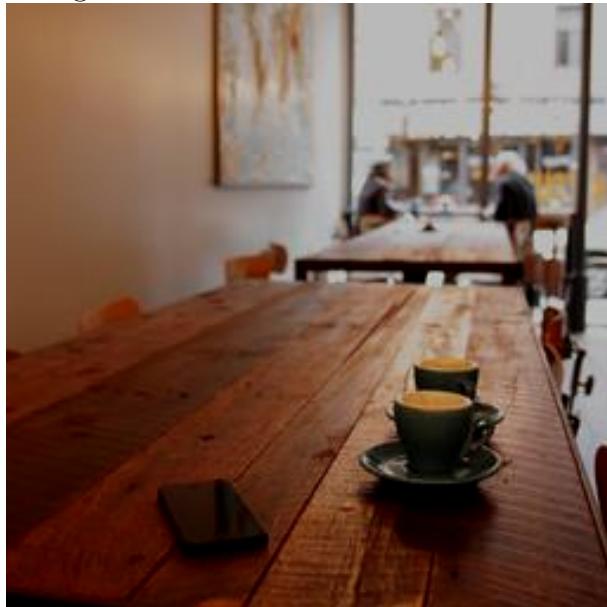
Rysunek 7.5: Obraz bazowy i jego obliczony histogram



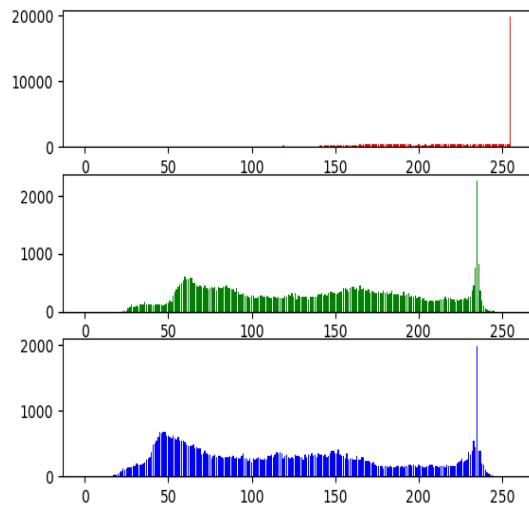
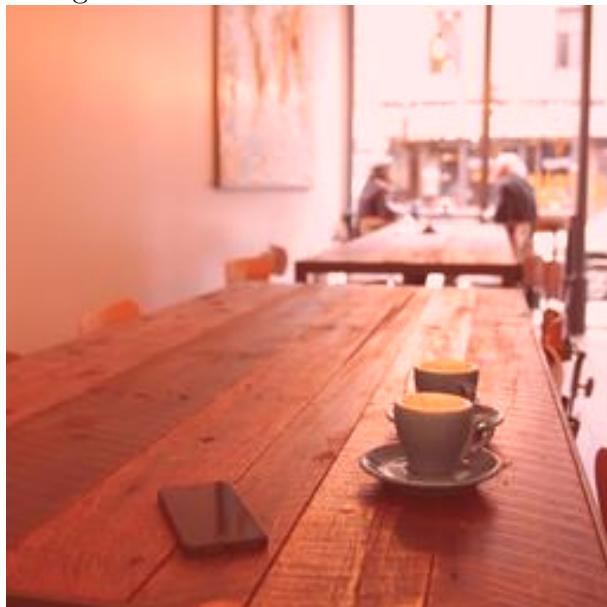
Rysunek 7.6: Obraz po przesunięciu histogramu o wartość 50 dla wszystkich barw i jego obliczony histogram



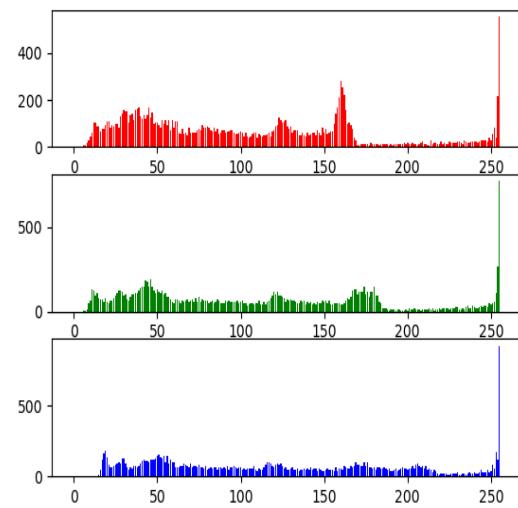
Rysunek 7.7: Obraz po przesunięciu histogramu o wartość -50 dla wszystkich barw i jego obliczony histogram



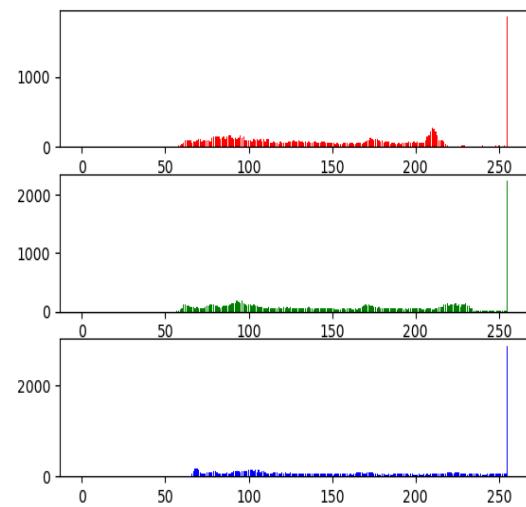
Rysunek 7.8: Obraz po przesunięciu histogramu o wartość 50 dla barwy czerwonej i jego obliczony histogram



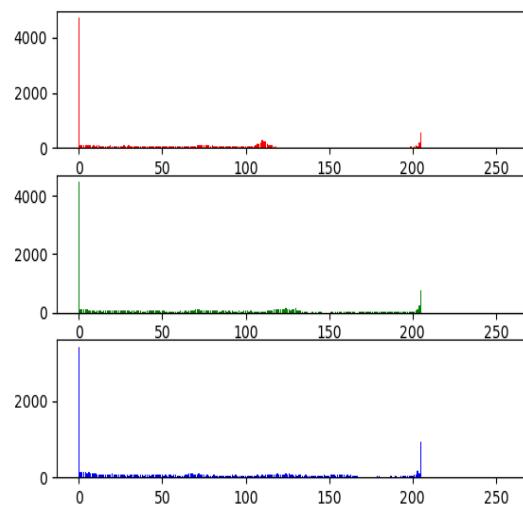
Rysunek 7.9: Obraz bazowy i jego obliczony histogram



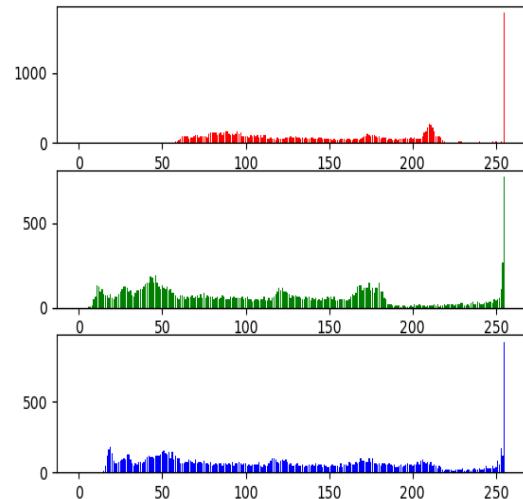
Rysunek 7.10: Obraz po przesunięciu histogramu o wartość 50 dla wszystkich barw i jego obliczony histogram



Rysunek 7.11: Obraz po przesunięciu histogramu o wartość -50 dla wszystkich barw i jego obliczony histogram



Rysunek 7.12: Obraz po przesunięciu histogramu o wartość 50 dla barwy czerwonej i jego obliczony histogram



7.3 Rozciąganie histogramu

Wstęp

Rozszerzanie histogramu odbywa się w tym przykładzie poprzez *normalizację*.

Kod algorytmu

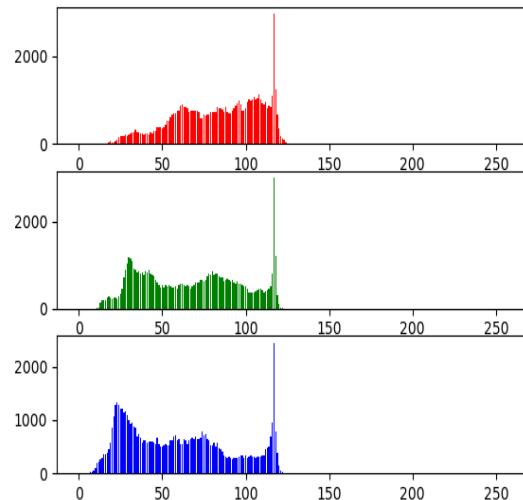
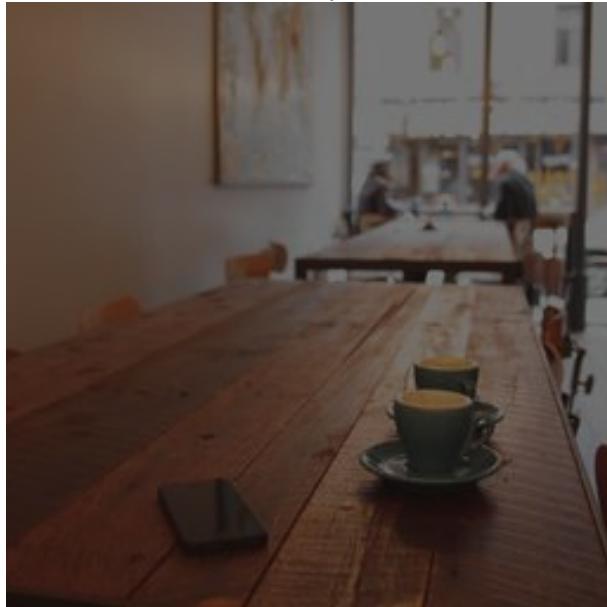
```
def extendHistogram(self):
    print('Extend histogram in color image {}'.format(self.firstDecoder.name))
    height, width = self.firstDecoder.height, self.firstDecoder.width
    image = self.firstDecoder.getPixels()
```

```
result = Commons.Normalization(image, image)

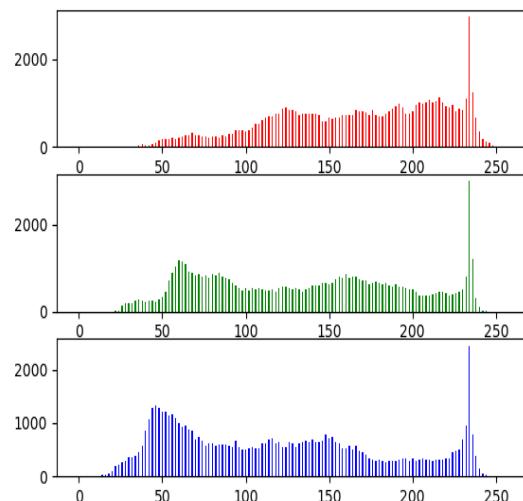
ImageHelper.Save(result, self.imageType, 'extend-histogram-image', False,
                  self.firstDecoder)
bins, histogram = Commons.CalculateColorHistogram(result, height, width)
ImageHelper.SaveColorHistogram(bins, histogram, 'extend-histogram', self.
                                firstDecoder)
```

Wynik algorytmu

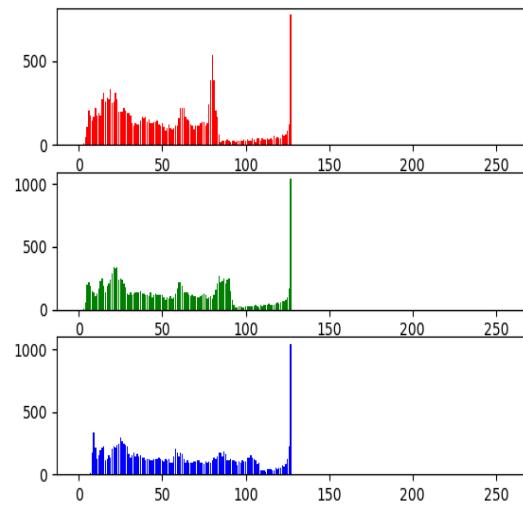
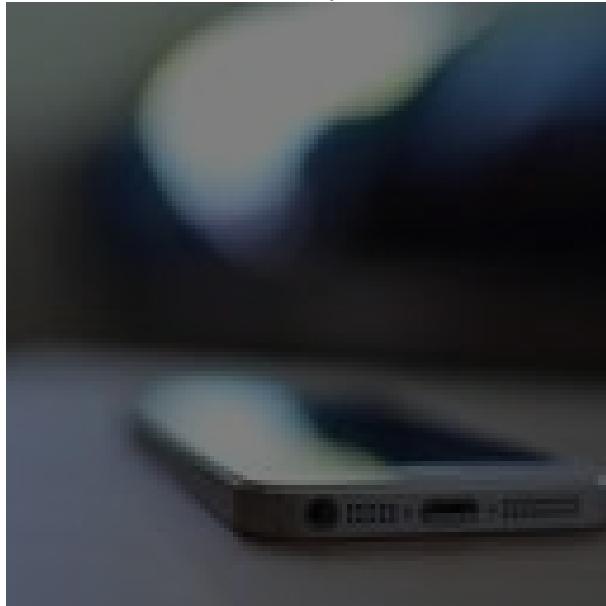
Rysunek 7.13: Obraz bazowy i jego obliczony histogram



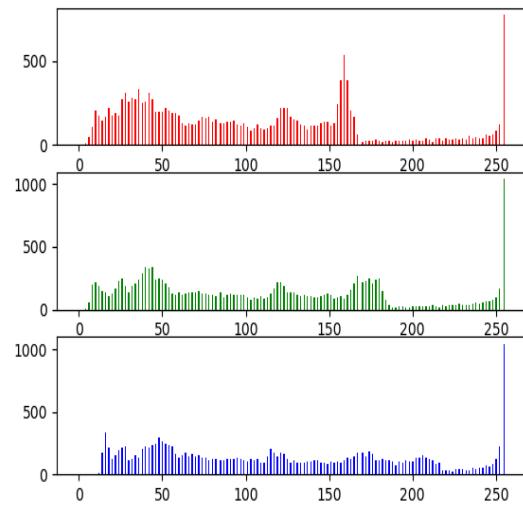
Rysunek 7.14: Po rozszerzeniu histogramu z pomocą normalizacji



Rysunek 7.15: Obraz bazowy i jego obliczony histogram



Rysunek 7.16: Po rozszerzeniu histogramu z pomocą normalizacji



7.4 Progowanie 1-progowe lokalne

Wstęp

Progowanie 1-progowe lokalne odbywa się w tym przykładzie z pomocą *metody Bernsen'a* zmodyfikowanej pod kątem wielu kanałów formatu RGB.

Kod algorytmu

```

def localSingleThresholding(self, contrastThreshold=15, windowSize=3):
    print('Local single thresholding color image {} with contrast threshold
          of {}'.format(self.firstDecoder.
                    name, contrastThreshold))

    if windowSize % 2 == 0:
        raise ValueError("Window size can't be even")

    height, width = self.firstDecoder.height, self.firstDecoder.width

```

```

image = self.firstDecoder.getPixels()
maxValue = int(numpy.iinfo(image.dtype).max)
minValue = int(numpy.iinfo(image.dtype).min)
if contrastThreshold <= minValue or contrastThreshold >= maxValue:
    raise ValueError("Contrast threshold has to be in range of ({},{}).
format(minValue, maxValue))

result = numpy.zeros((height, width, 3), numpy.uint8)
overlap = int(math.ceil(windowSize/2))
for h in range(height):
    for w in range(width):
        minH = 0 if h-overlap < 0 else h-overlap
        maxH = height if h+overlap+1 > height else h+overlap+1
        minW = 0 if w-overlap < 0 else w-overlap
        maxW = width if w+overlap+1 > width else w+overlap+1
        if minH >= maxH or minW >= maxW:
            continue

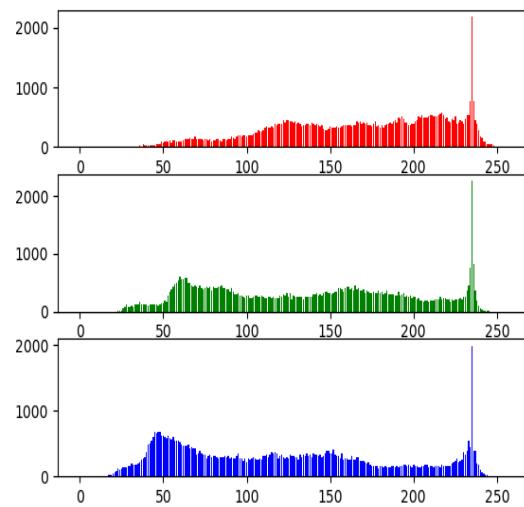
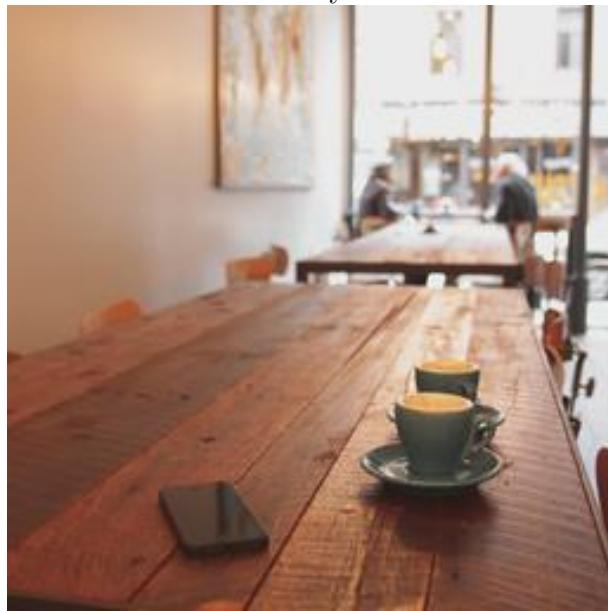
        window = image[minH:maxH, minW:maxW]
        for color in range(3):
            localMin = numpy.amin(window[:, :, color])
            localMax = numpy.amax(window[:, :, color])
            localContrast = localMax-localMin
            midColor = (int(localMax)+int(localMin))/2
            if localContrast < contrastThreshold:
                if midColor >= maxValue/2:
                    result[minH:maxH, minW:maxW, color] = numpy.full((
                        maxH-minH,
                        maxW-minW),
                        maxValue)
                else:
                    result[minH:maxH, minW:maxW, color] = numpy.full((
                        maxH-minH,
                        maxW-minW),
                        minValue)
            else:
                if image[h, w, color] >= midColor:
                    result[h, w, color] = maxValue
                else:
                    result[h, w, color] = minValue

ImageHelper.Save(result, self.imageType, 'single-local-threshold-image',
                False, self.firstDecoder, None,
                contrastThreshold)
bins, histogram = Commons.CalculateColorHistogram(result, height, width)
ImageHelper.SaveColorHistogram(bins, histogram, 'single-local-threshold',
                               self.firstDecoder)

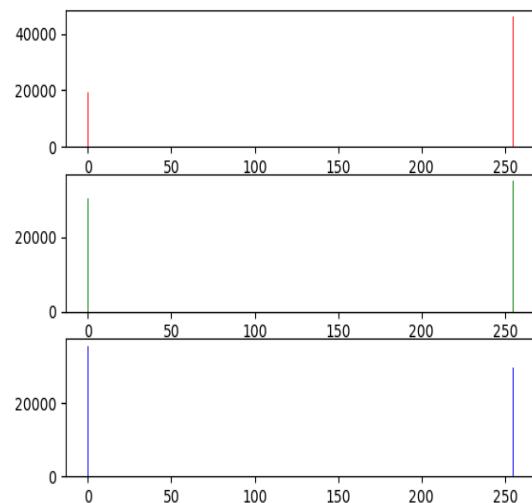
```

Wynik algorytmu

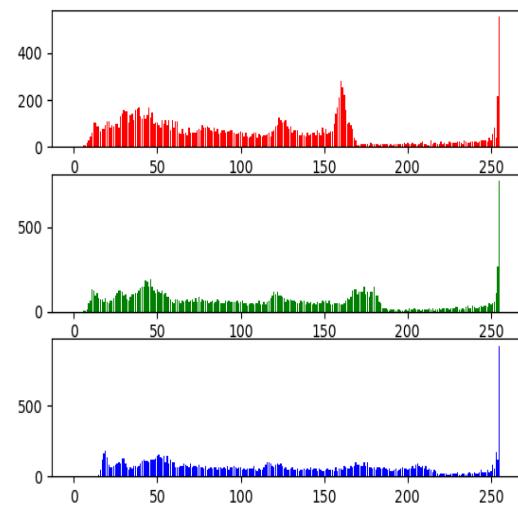
Rysunek 7.17: Obraz bazowy i jego obliczony histogram



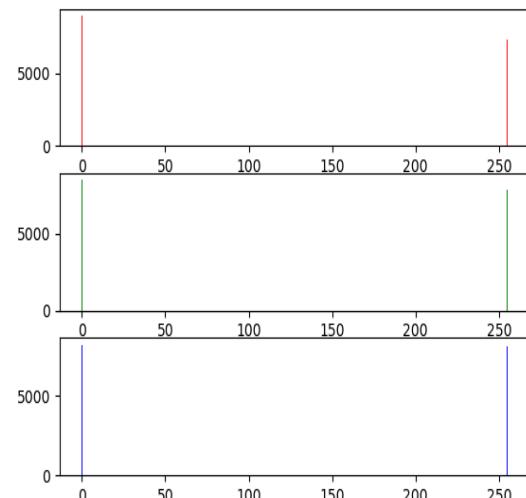
Rysunek 7.18: Obraz po zastosowaniu progowania Bernstena o kontraście progowym 15 wraz z jego obliczonym histogramem



Rysunek 7.19: Obraz bazowy i jego obliczony histogram



Rysunek 7.20: Obraz po zastosowaniu progowania Bernstena o kontraście progowym 15 wraz z jego obliczonym histogramem



7.5 Progowanie wielo-progowe lokalne

Wstęp

Progowanie wielo-progowe lokalne używa wielu progów w celu podzielenia zawartości przesuwającego się okna na $T - 1$ przedziałów w celu redukcji ilości kolorów w obrazie. Wynikiem tej operacji jest zmniejszenie gęstości histogramu.

Kod algorytmu

```
def localMultiThresholding(self, amountOfThresholds=4, windowSize=3):
    print('Local multi thresholding color image {} with amount of thresholds
          equals to {}'.format(self.
                           firstDecoder.name,
                           amountOfThresholds))
```

```

if windowSize % 2 == 0:
    raise ValueError("Window size can't be even")

height, width = self.firstDecoder.height, self.firstDecoder.width
image = self.firstDecoder.getPixels()

result = numpy.zeros((height, width, 3), numpy.uint8)
overlap = int(math.ceil(windowSize/2))
for h in range(height):
    for w in range(width):
        minH = 0 if h-overlap < 0 else h-overlap
        maxH = height if h+overlap+1 > height else h+overlap+1
        minW = 0 if w-overlap < 0 else w-overlap
        maxW = width if w+overlap+1 > width else w+overlap+1
        if minH >= maxH or minW >= maxW:
            continue

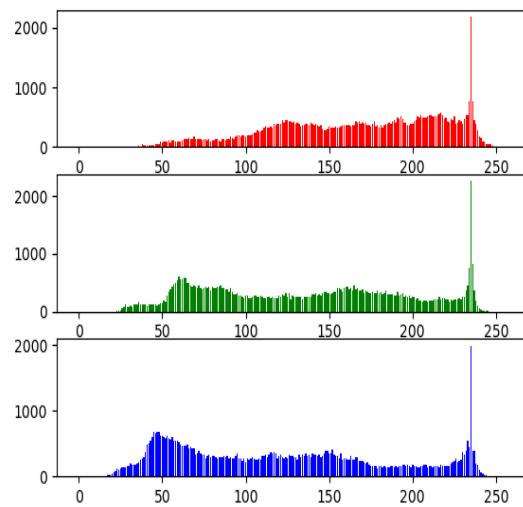
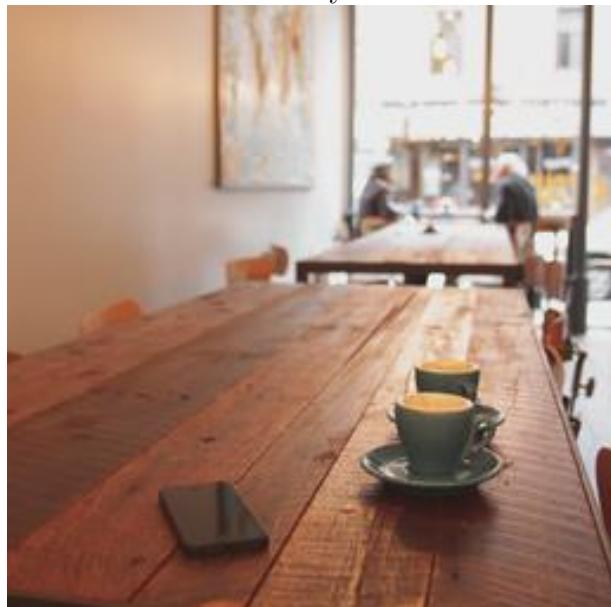
        window = image[minH:maxH, minW:maxW]
        for color in range(3):
            localMax = float(numpy.amax(window[:, :, color]))
            scale = localMax / (amountOfThresholds-1) if localMax != 0
                                         else 1
            result[h, w, color] = int(round(image[h, w, color] / scale))
                                         * int(scale)

ImageHelper.Save(result, self.imageType, 'multi-local-threshold-image',
                 False, self.firstDecoder, None,
                 amountOfThresholds)
bins, histogram = Commons.CalculateColorHistogram(result, height, width)
ImageHelper.SaveColorHistogram(bins, histogram, 'multi-local-threshold',
                               self.firstDecoder,
                               amountOfThresholds)

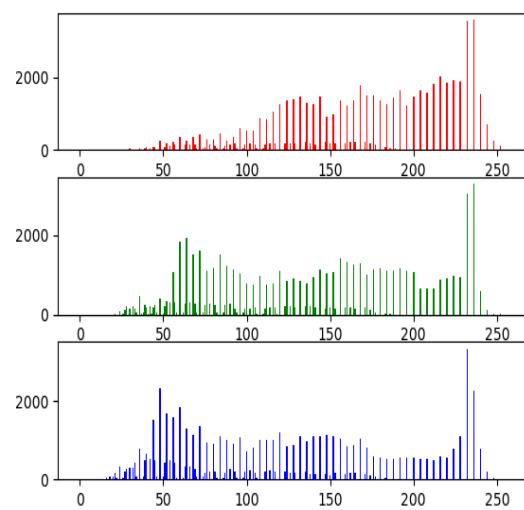
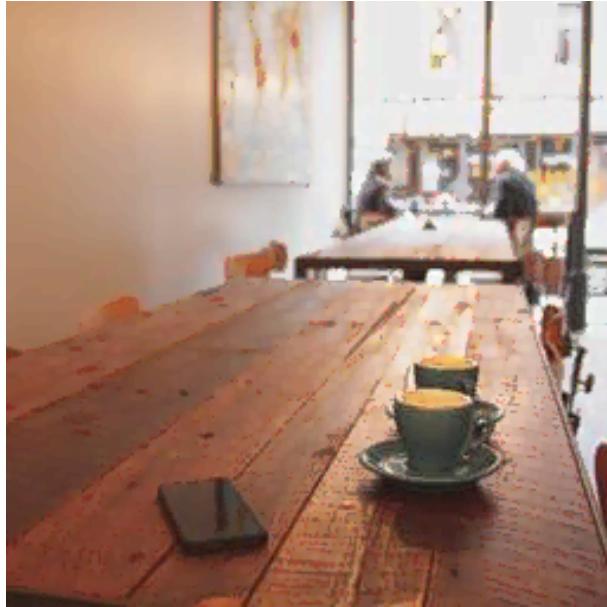
```

Wynik algorytmu

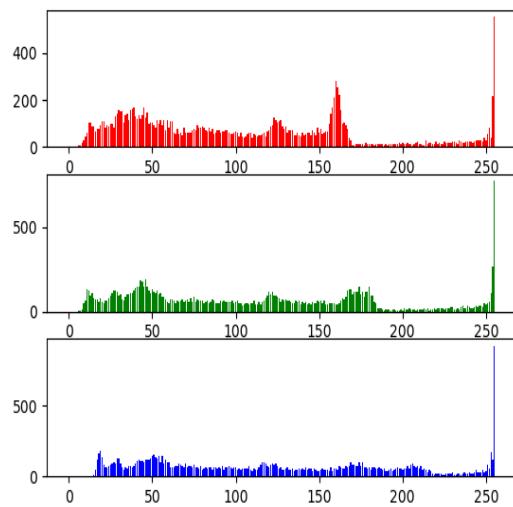
Rysunek 7.21: Obraz bazowy i jego obliczony histogram



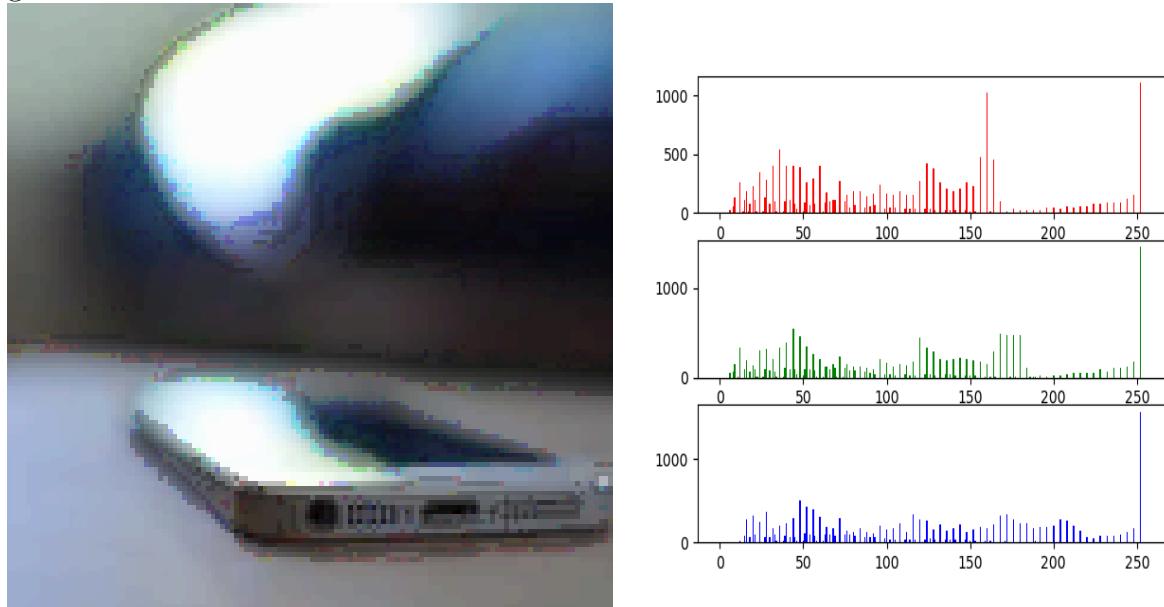
Rysunek 7.22: Obraz po zastosowaniu progowania z czterema progami wraz z jego obliczonym histogramem



Rysunek 7.23: Obraz bazowy i jego obliczony histogram



Rysunek 7.24: Obraz po zastosowaniu progowania z czterema progami wraz z jego obliczonym histogramem



7.6 Progowanie 1-progowe globalne

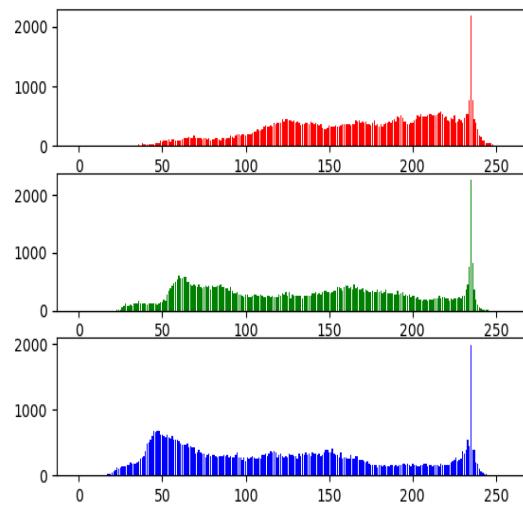
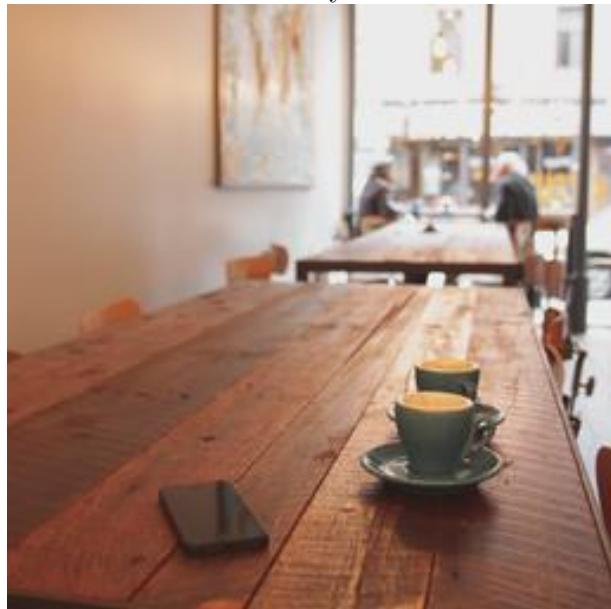
Wstęp

Progowanie 1-progowe globalne dla obrazów barwnych opera się o ustalenie progu dla każdej z trzech barw RGB. Próg jest wyliczany ze wszystkich pikseli na podstawie średniej arytmetycznej.

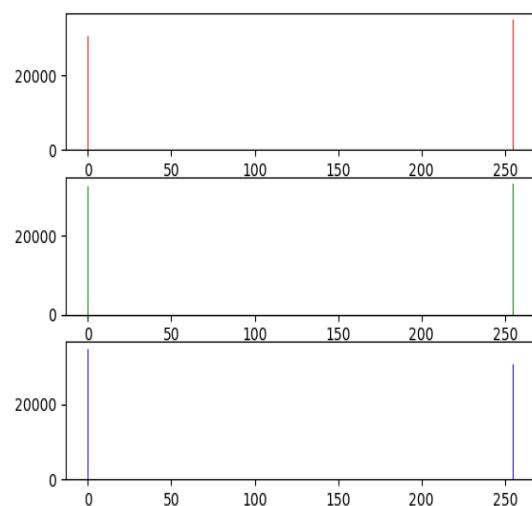
Kod algorytmu

Wynik algorytmu

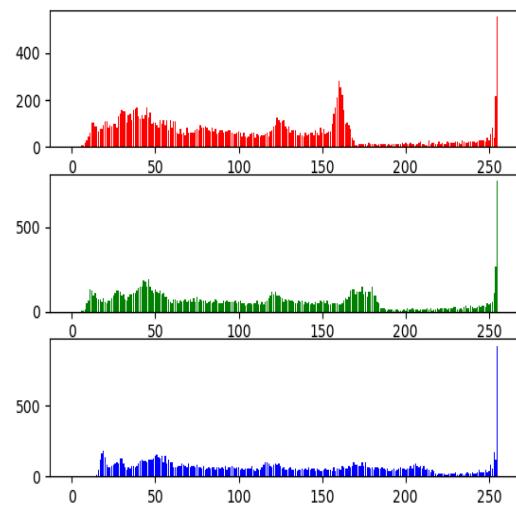
Rysunek 7.25: Obraz bazowy i jego obliczony histogram



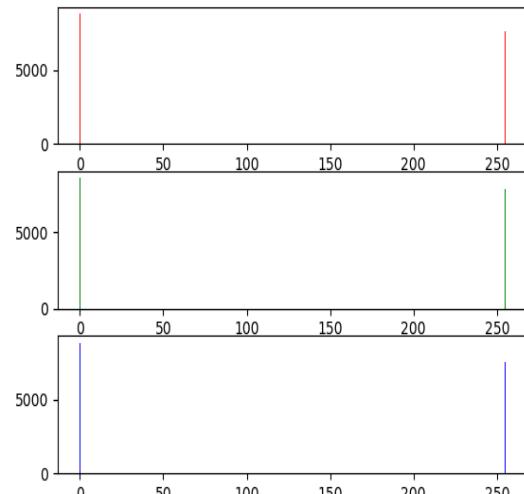
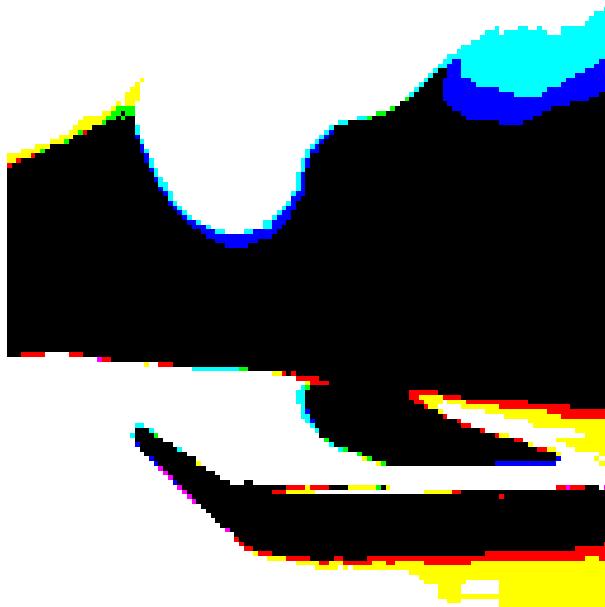
Rysunek 7.26: Obraz po zastosowaniu progowania globalnego wraz z jego obliczonym histogramem



Rysunek 7.27: Obraz bazowy i jego obliczony histogram



Rysunek 7.28: Obraz po zastosowaniu progowania globalnego wraz z jego obliczonym histogramem



7.7 Progowanie wielo-progowe globalne

Wstęp

Progowanie wielo-progowe globalne do stworzenia progów używa maksymalnej wartości ze wszystkich piksli, którą to wartość dzieli później na $T - 1$ odcinków i każdy piksel z tych odcinków przypisuje do jednej z $T + 1$ wartości progowych.

Kod algorytmu

```
def globalMultiThresholding(self, amountOfThresholds=4):
    print('Global multi thresholding color image {} with amount of thresholds
          equals to {}'.format(self.
                           firstDecoder.name,
                           amountOfThresholds))
```

```

height, width = self.firstDecoder.height, self.firstDecoder.width
image = self.firstDecoder.getPixels()

maxR = numpy.amax(image[:, :, 0])
maxG = numpy.amax(image[:, :, 1])
maxB = numpy.amax(image[:, :, 2])
scaleR = maxR / (amountOfThresholds-1)
scaleG = maxG / (amountOfThresholds-1)
scaleB = maxB / (amountOfThresholds-1)

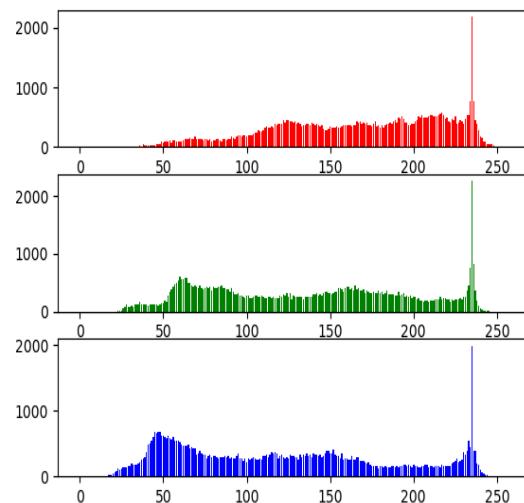
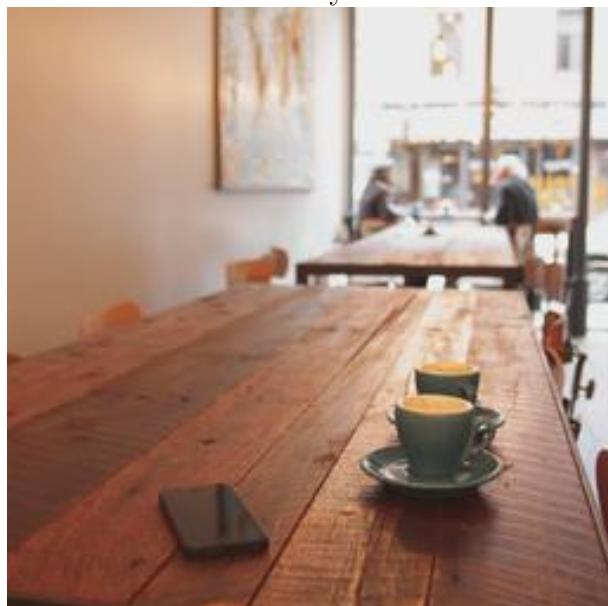
result = numpy.zeros((height, width, 3), numpy.uint8)
for h in range(height):
    for w in range(width):
        result[h, w, 0] = int(round(image[h, w, 0] / scaleR)) * int(
            scaleR)
        result[h, w, 1] = int(round(image[h, w, 1] / scaleG)) * int(
            scaleG)
        result[h, w, 2] = int(round(image[h, w, 2] / scaleB)) * int(
            scaleB)

ImageHelper.Save(result, self.imageType, 'multi-global-threshold-image',
                 False, self.firstDecoder, None,
                 amountOfThresholds)
bins, histogram = Commons.CalculateColorHistogram(result, height, width)
ImageHelper.SaveColorHistogram(bins, histogram, 'multi-global-threshold',
                               self.firstDecoder,
                               amountOfThresholds)

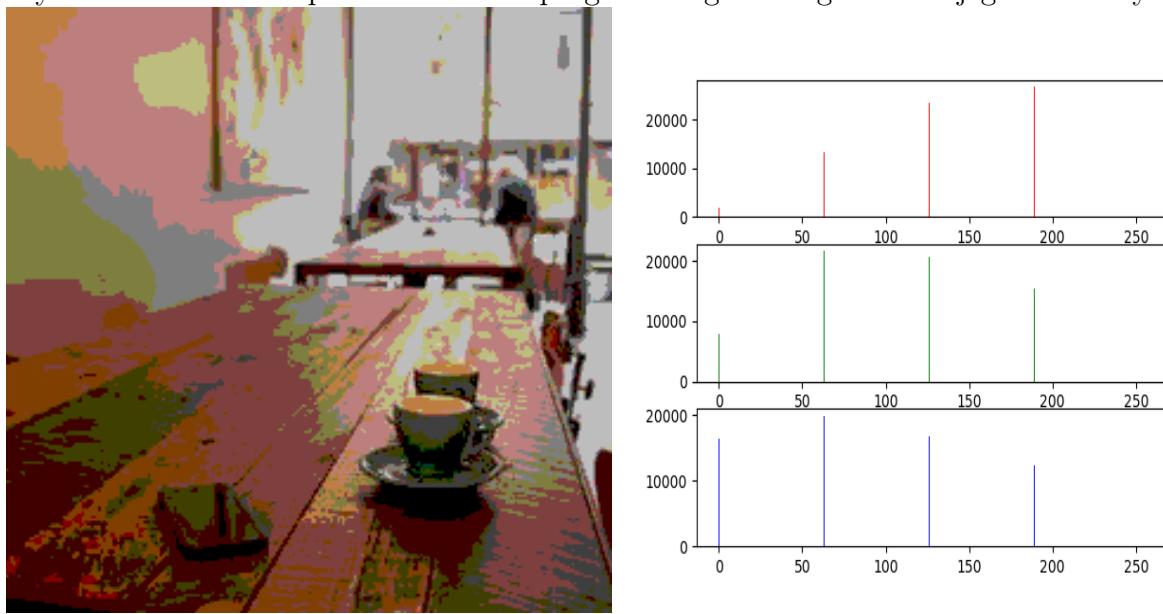
```

Wynik algorytmu

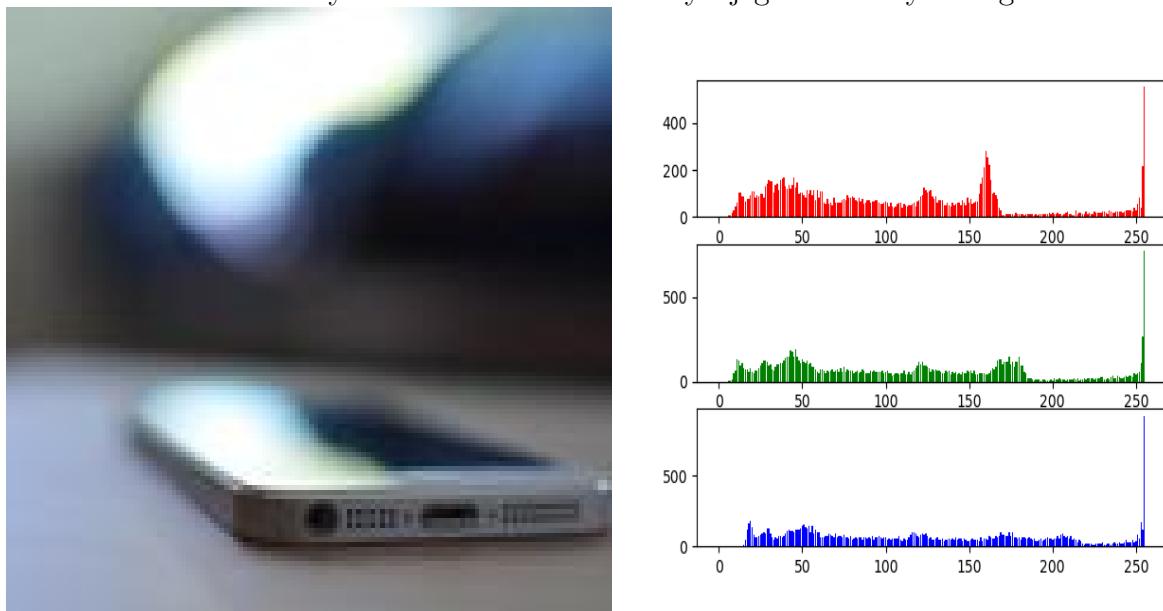
Rysunek 7.29: Obraz bazowy i jego obliczony histogram



Rysunek 7.30: Obraz po zastosowaniu progowania globalnego wraz z jego obliczonym histogramem



Rysunek 7.31: Obraz bazowy i jego obliczony histogram



Rysunek 7.32: Obraz po zastosowaniu progowania globalnego wraz z jego obliczonym histogramem

