

# **Przetwarzanie Obrazów: Sprawozdanie**

Damian Ubowski

Warszawa, 2020



# Spis treści

<b>1 Wstęp</b>	<b>5</b>
1.1 Format obrazu . . . . .	5
1.1.1 Struktura formatu . . . . .	5
1.1.2 Przykładowa struktura IFF . . . . .	5
1.1.3 Instrukcja obsługi programu . . . . .	6
<b>2 Operacje ujednoliciania obrazów</b>	<b>7</b>
2.1 Ujednolicenie obrazów szarych geometryczne . . . . .	8
2.2 Ujednolicenie obrazów szarych rozdzielczościowe . . . . .	10
2.3 Ujednolicenie obrazów RGB geometryczne . . . . .	13
2.4 Ujednolicenie obrazów RGB rozdzielczościowe . . . . .	16
<b>3 Operacje sumowania arytmetycznego obrazów szarych</b>	<b>19</b>
3.1 Sumowanie określonej stałej z obrazem . . . . .	20
3.2 Sumowanie dwóch obrazów . . . . .	22
3.3 Mnożenie obrazu przez zadaną liczbę . . . . .	25
3.4 Mnożenie obrazu przez inny obraz . . . . .	29
3.5 Mieszanie obrazów z określonym współczynnikiem . . . . .	32
3.6 Potęgowanie obrazu z zadaną potęgą . . . . .	34
<b>4 Operacje sumowania arytmetycznego obrazów barwowych</b>	<b>37</b>
4.1 Sumowanie (określonej) stałej z obrazem . . . . .	38
4.2 Sumowanie dwóch obrazów . . . . .	40
4.3 Mnożenie obrazu przez zadaną liczbę . . . . .	43
4.4 Mnożenie obrazu przez inny obraz . . . . .	43
4.5 Mieszanie obrazów z określonym współczynnikiem . . . . .	43
4.6 Potęgowanie obrazu (z zadaną potegą) . . . . .	43
<b>5 Operacje geometryczne na obrazie</b>	<b>45</b>
5.1 Przesunięcie obrazu o zadany wektor . . . . .	46
5.2 Jednorodne skalowanie obrazu . . . . .	48
5.3 Niejednorodne skalowanie obrazu . . . . .	50
5.4 Obracanie obrazu o dowolny kąt . . . . .	52
5.5 Symetrie względem osi układu . . . . .	54
5.6 Symetrie względem zadanej prostej . . . . .	56
5.7 Wycinanie fragmentów obrazu . . . . .	58
5.8 Kopiowanie fragmentów obrazów . . . . .	59
<b>6 Operacje na histogramie obrazu szarego</b>	<b>61</b>
6.1 Obliczanie histogramu . . . . .	61
6.2 Przesunięcie histogramu . . . . .	61
6.3 Rozciąganie histogramu . . . . .	61
6.4 Progowanie lokalne . . . . .	61

6.5 Progowanie globalne . . . . .	61
<b>7 Operacje na histogramie obrazu barwowego</b>	<b>63</b>
7.1 Obliczanie histogramu . . . . .	63
7.2 Przemieszczanie histogramu . . . . .	63
7.3 Rozciąganie histogramu . . . . .	63
7.4 Progowanie 1-progowe lokalne . . . . .	63
7.5 Progowanie wielo-progowe lokalne . . . . .	63
7.6 Progowanie 1-progowe globalne . . . . .	63
7.7 Progowanie wielo-progowe globalne . . . . .	63

# Rozdział 1

## Wstęp

### 1.1 Format obrazu

Wybranym przez nas formatem obrazów cyfrowych jest DjVu, który jest oparty na zaawansowanej metodzie segmentacji obrazu. Tworzenie pliku DjVu polega na rozdzieleniu dowolnie skomplikowanego obrazu na odrębne warstwy, a następnie poddaniu warst odrębnym optymalizacjom i kompresjom. Format ten stosuje ładowanie progresywne, kodowanie arytmetyczne, oraz kompresję stratną dzięki czemu przy minimalnej ilości przestrzeni dyskowej można delektować się obrazami i dokumentami w wysokiej jakości.

#### 1.1.1 Struktura formatu

Pliki DjVu rozpoczynają się od swojej “Magic number” potwierdzającej rodzaj pliku i mającej wartość *0x41 0x54 0x26 0x54*. Następnie czerpiąc inspirację ze struktury IFF (**I**nterchange **F**ile **F**ormat) 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. Każdy kawałek składający się z ID typu, długości zawartości i samej zawartości tworzy zwarty format. Identyfikator typu określa rolę w jakiej przyjdzie służyć kawałkowi. Do dyspozycji ma ich całkiem sporo, ale uwzględniając najbardziej przydatne w naszym kontekście to ograniczymy liczbę do:

- \* BGjp - warstwa tylna przechowywana przy użyciu kodowania JPEG.
- \* BFjp - warstwa przednia w formacie JPEG.
- \* INFO - opisuje wysokość, szerokość, rozdzielczość, wersję kodera, oraz flagi wskazujące na obrót obrazu.

#### 1.1.2 Przykładowa struktura IFF

FORM:DJVU [14260]

INFO [10]

Sjbz [13133]

FG44 [181]

BG44 [935]

Powyższa struktura przedstawia dokument składający się z jednej strony, na co wskazuje *FORM:DJVU*, wraz z grafiką. Ten znacznik informuje, że mamy do czynienia z kontenerem o długości 14260 bajtów, który może zawierać inne kawałki dokumentu. Zgodnie z konwencją, po identyfikatorze typu i informacji o długości znajduje się zawartość kawałka. W tym wypadku jak i w każdym innym po *FORM:DJVU* powinno znaleźć się *INFO* z podstawowymi informacjami. Jeśli konwencji i wymagań specyfikacyjnych stało się zadość wtedy czas nastął na jakieś wizualne atrakcje takie jak *Sjbz*, czyli masce wyboru pomiędzy kolorami z warstwy przedniej (*FG44*) i tylnej (*BG44*).

### 1.1.3 Instrukcja obsługi programu

W celu uruchomienia kodu źródłowego będzie niezbędny:

- \* [Python](#) ( $\geq 2.6$  lub 3.X)

## 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.

Rysunek 2.1: Przed uruchomieniem algorytmu (od lewej): obraz 1 (1067x1067, 300dpi), obraz 2 (2133x2133, 300dpi)



## 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

### Kod źródłowy algorytmu

```
def geometricGray(self):
    print('geometric gray unification start')
    width, height = self.firstDecoder.width, self.firstDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        # Create black background
        firstResult = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
        # 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):
```

Rysunek 2.2: Po uruchomieniu algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



```

firstResult[h + startHeightIndex, w + startWidthIndex] = pixelsBuffer[h, w]
img = Image.fromarray(firstResult, mode='L')
img.save('Resources/ggUnification_1.png')
print('first image done')

width, height = self.secondDecoder.width, self.secondDecoder.height
if width < self.maxWidth or height < self.maxHeight:
    # Create black background
    secondResult = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
    # 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]
    img = Image.fromarray(secondResult, mode='L')
    img.save('Resources/ggUnification_2.png')
    print('second image done')
print('geometric gray unification done')

```

Rysunek 2.3: Skutki braku interpolacji



## 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. Nanieśenie pikseli z mniejszego obrazu na większy z uwzględnieniem luk
4. Interpolacja

### Kod źródłowy algorytmu

```
def rasterGray(self):
    print('raster gray unification start')
    self._scaleUpGray(self.firstDecoder, 'Resources/rgUnification_1.png')
    print('first image done')
    self._scaleUpGray(self.secondDecoder, 'Resources/rgUnification_2.png')
    print('second image done')
    print('raster gray unification done')
```

Rysunek 2.4: Przed uruchomieniem algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



Rysunek 2.5: Po uruchomieniu algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



```

def _scaleUpGray(self, decoder, outputPath):
    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):
            for w in range(width):
                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(round(scaleFactoryW * w))] =
                        pixelsBuffer[h, w]
        # Interpolate
        self._interpolateGray(result)
        img = Image.fromarray(result, mode='L')
        img.save(outputPath)

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

```

Rysunek 2.6: Przed uruchomieniem algorytmu (od lewej): obraz 1 (512x512, 300dpi), obraz 2 (1024x1024, 300dpi)



## 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 piksele na czarne tło z uwzględnieniem każdego z kanałów RGB
3. Jeśli żaden z warunków jest niespełniony to nie rób nic

Rysunek 2.7: Po uruchomieniu algorytmu (od lewej): obraz 1 (1024x1024, 300dpi), obraz 2 (1024x1024, 300dpi)



Rysunek 2.8: Przed uruchomieniem algorytmu (od lewej): obraz 3 (126x126, 300dpi), obraz 4 (256x256, 300dpi)



Rysunek 2.9: Po uruchomieniu algorytmu (od lewej): obraz 3 (126x126, 300dpi), obraz 4 (256x256, 300dpi)



## Kod źródłowy algorytmu

```

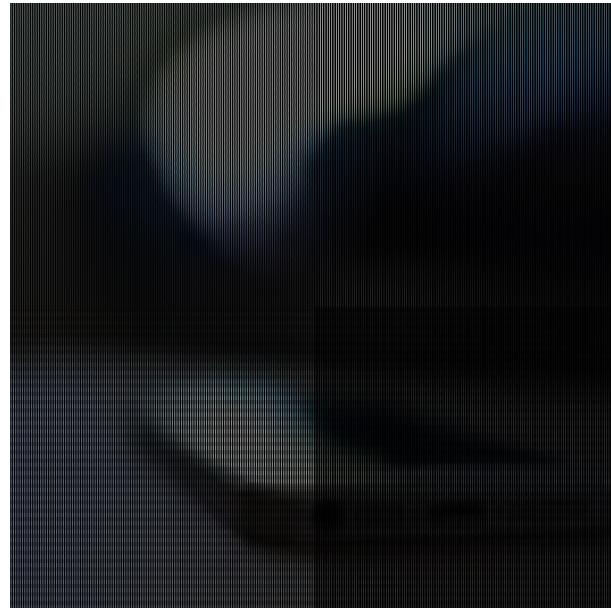
def geometricColor(self):
    print('geometric color unification start')
    self.firstDecoder.setColor()
    width, height = self.firstDecoder.width, self.firstDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        result = self._paintInMiddleColor(self.firstDecoder)
    img = Image.fromarray(result, 'RGB')
    img.save('Resources/gcUnification_1.png')
    print('first image done')

    self.secondDecoder.setColor()
    width, height = self.secondDecoder.width, self.secondDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        result = self._paintInMiddleColor(self.secondDecoder)
    img = Image.fromarray(result, 'RGB')
    img.save('Resources/gcUnification_2.png')
    print('second image done')
    print('geometric color unification done')

def _paintInMiddleColor(self, decoder):
    # Create black background
    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.getPixels24Bits()
    for h in range(0, height):
        for w in range(0, width):
            result[h + startHeightIndex, w + startWidthIndex] = pixelsBuffer[h, w]
    return result

```

Rysunek 2.10: Skutki braku interpolacji



## 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 2.11: Przed uruchomieniem algorytmu (od lewej): obraz 1 (256x256, 300dpi), obraz 2 (256x256, 300dpi)



Rysunek 2.12: Po uruchomieniu algorytmu (od lewej): obraz 1 (256x256, 300dpi), obraz 2 (256x256, 300dpi)



## Kod źródłowy algorytmu

```

def rasterColor(self):
    print('rastar color unification start')
    self.firstDecoder.setColor()
    self._scaleUpColor(self.firstDecoder, 'Resources/rcUnification_1.png')
    print('first image done')
    self.secondDecoder.setColor()
    self._scaleUpColor(self.secondDecoder, 'Resources/rcUnification_2.png')
    print('second image done')
    print('rastar color unification done')

def _scaleUpColor(self, decoder, outputPath):
    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.getPixels24Bits()
        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)
        img = Image.fromarray(result, mode='RGB')
        img.save(outputPath)

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)



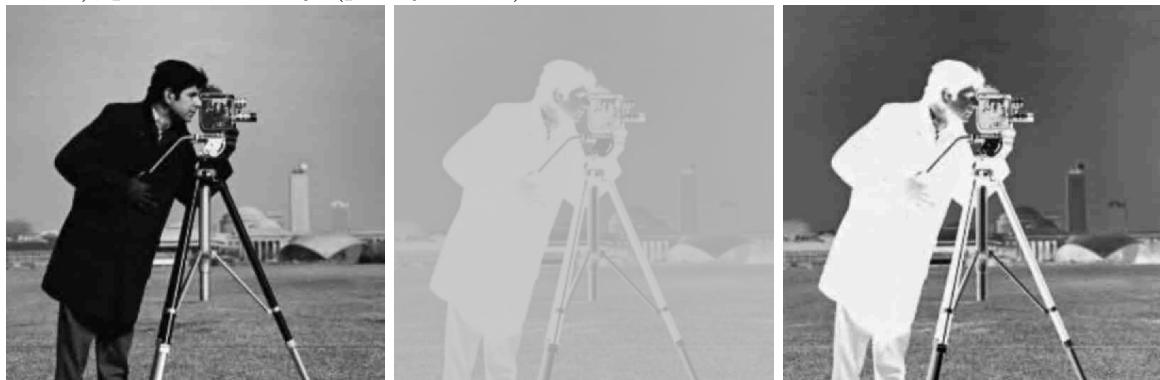
Rysunek 3.2: Przed uruchomieniem algorytmu (lewy obraz), po dodaniu wartości 300 (środkowy obraz), po normalizacji (prawy obraz)



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  $\{x : 0 < x < 1\}$  przyjemniej obrazy, a 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 0.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.9: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 1.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.10: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 0.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.11: Przed uruchomieniem algorytmu (lewy obraz), po pomnożeniu przez wartość 1.5 (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.

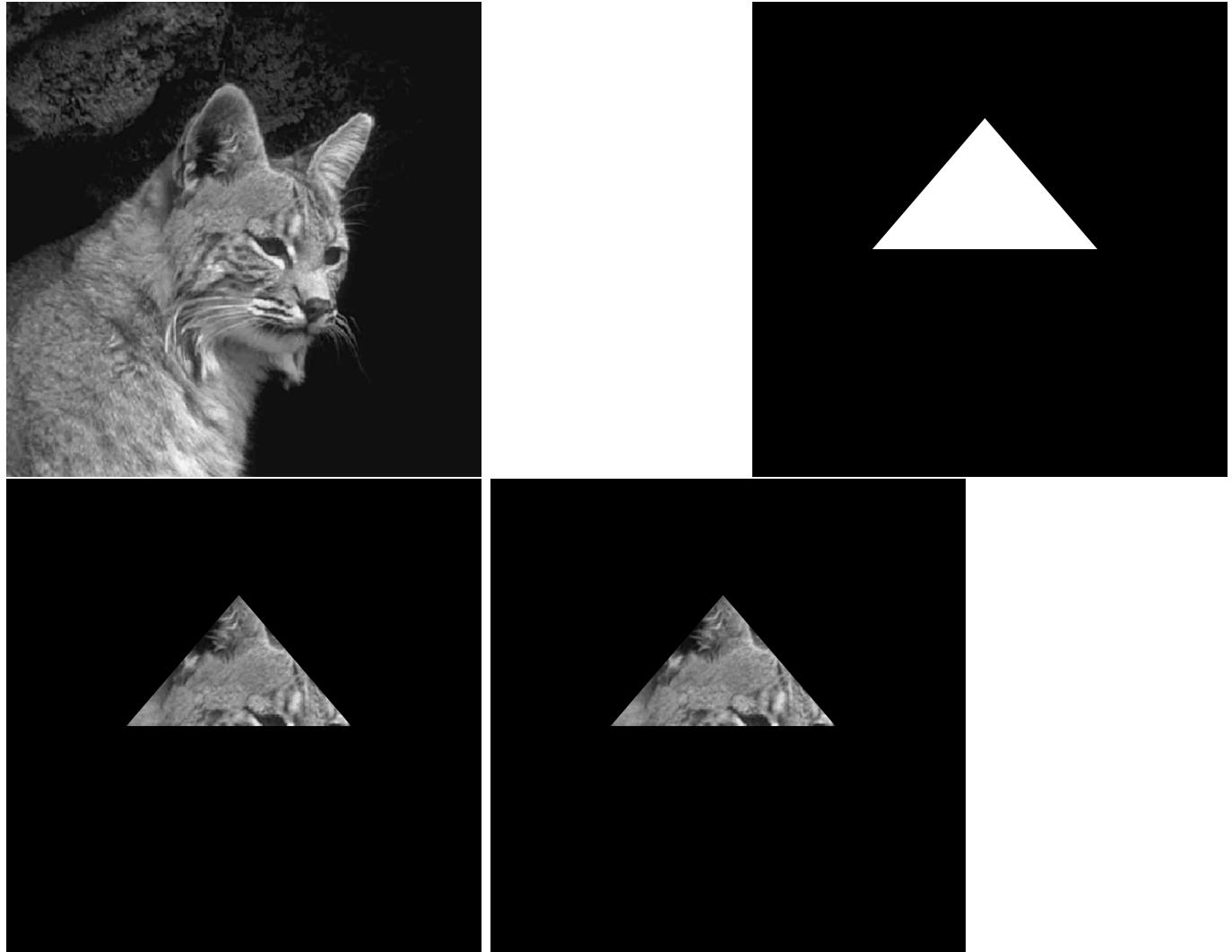
Rysunek 3.12: Przed uruchomieniem algorytmu (obrazy na górze), po przemnożeniu obrazów (lewy, dolny obraz), po normalizacji (prawy, dolny obraz)



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  $\alpha$  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  $\alpha$  kontrast między pikselami jest zatracany. Problem występuje głównie gdy kontrast na samych obrazach jest dość ubogi. Aby podtrzymać kontrast możemy zwiększyć współczynnik  $\alpha$  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  $\alpha$  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 (lewy, górny obraz), po mieszaniu o wartość 0.2 (prawy, górny obraz), po mieszaniu o wartość 0.5 (lewy, dolny obraz), po mieszaniu o wartość 0.8 (prawy, dolny 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)

```

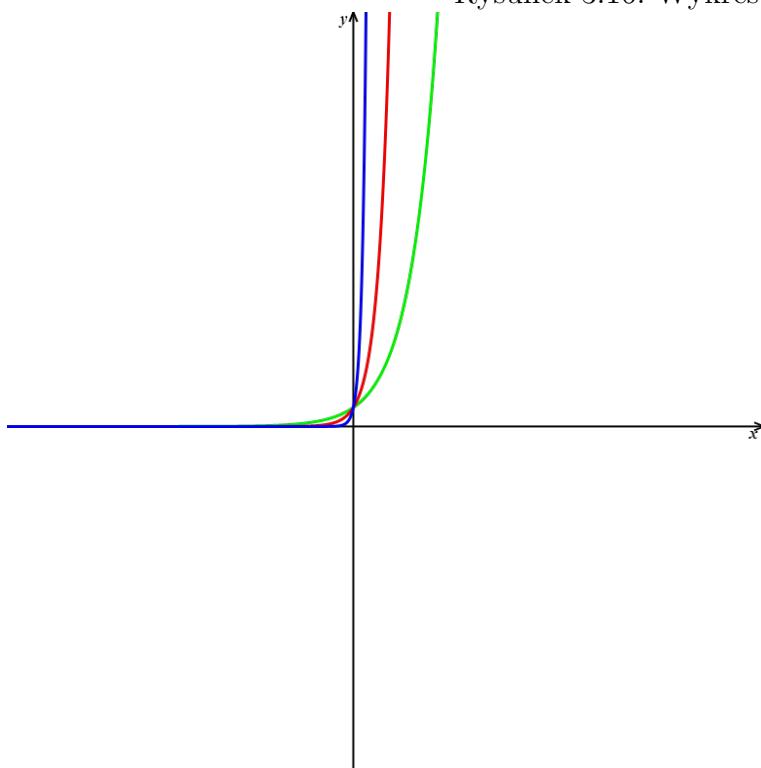
## 3.6 Potęgowanie obrazu z zadaną potęgą

### 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 3.16: Wykres eksponenty



W zależności od dobrania odpowiedniej wartości *wykładnika* możemy:

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

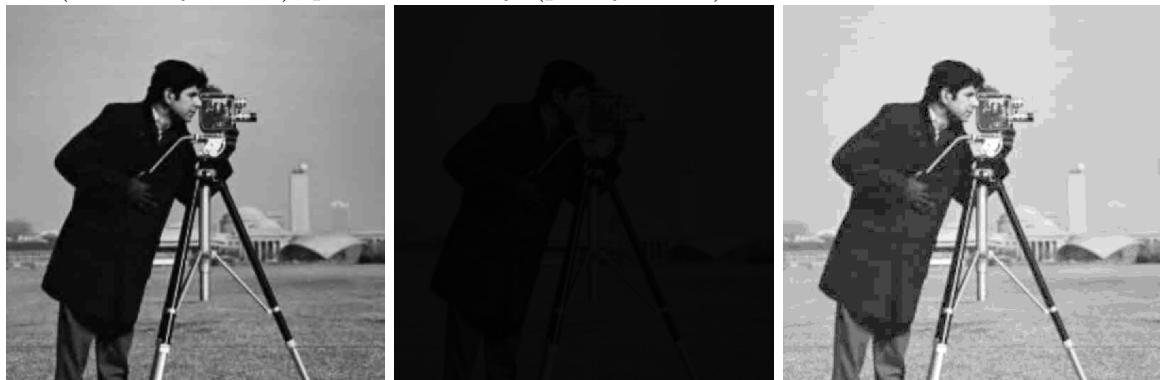
2. zwiększyć kontrast dla dużych jaskrawości; wybierając  $\alpha \in (1, \infty)$ .

Obliczając wartości pikseli użyjemy wzoru:

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

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

Rysunek 3.17: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 0.5 (środkowy obraz), po normalizacji (prawy obraz)



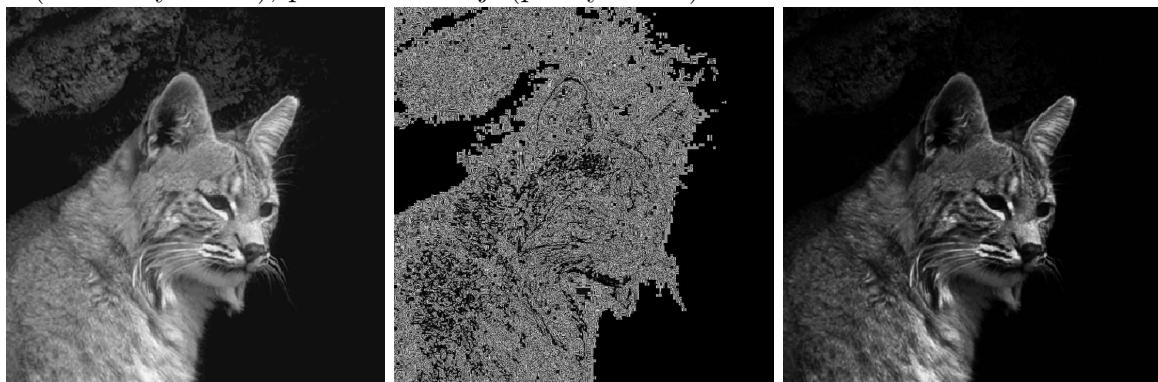
Rysunek 3.18: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 2 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.19: Przed uruchomieniem algorytmu (lewy obraz), po użyciu potęgowania o wykładniku 0.5 (środkowy obraz), po normalizacji (prawy obraz)



Rysunek 3.20: 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 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), 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)

```

## 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

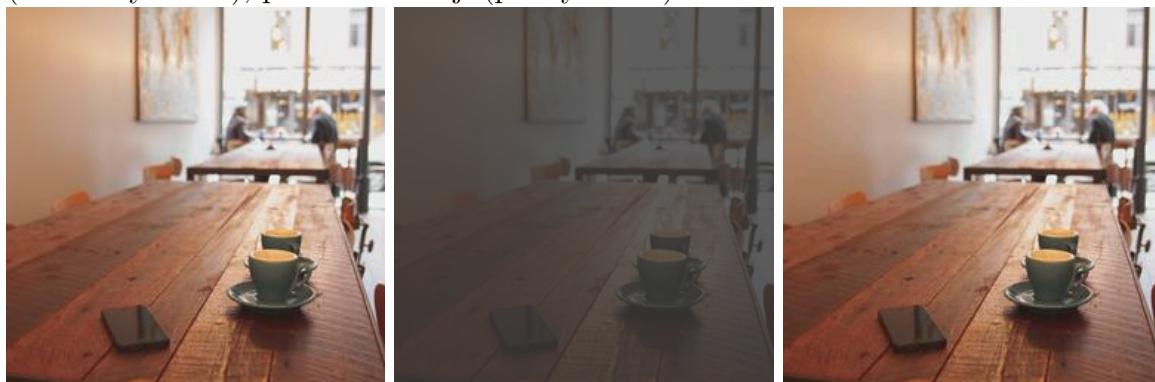
Rysunek 4.1: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



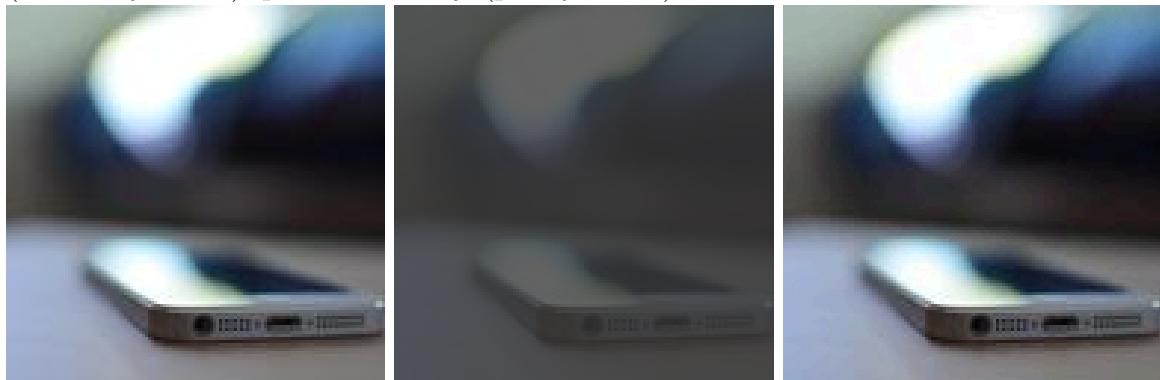
Rysunek 4.2: Przed uruchomieniem algorytmu (lewy obraz), po użyciu dodawania o wartości 30 (środkowy obraz), po normalizacji (prawy obraz)



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)

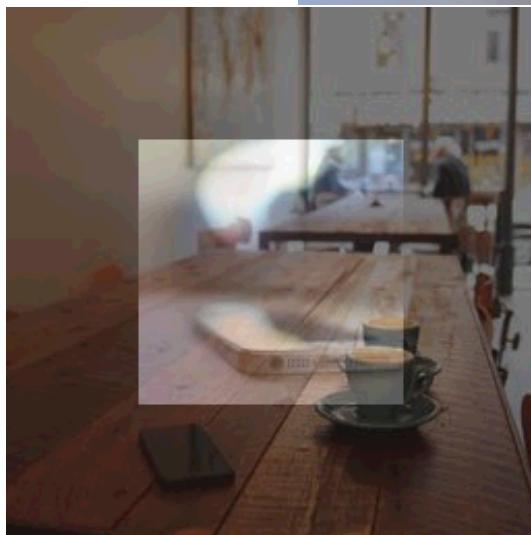
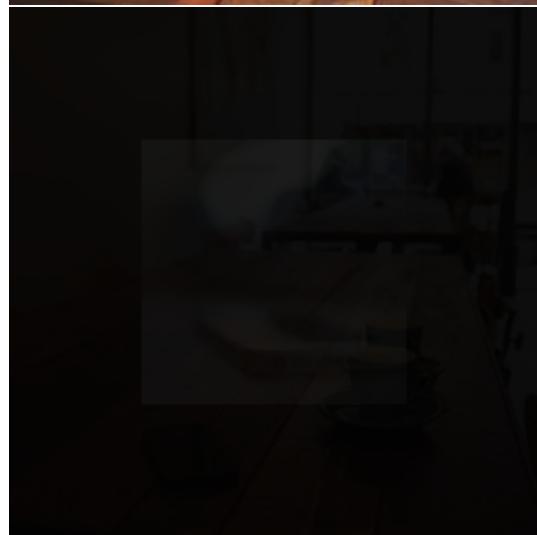
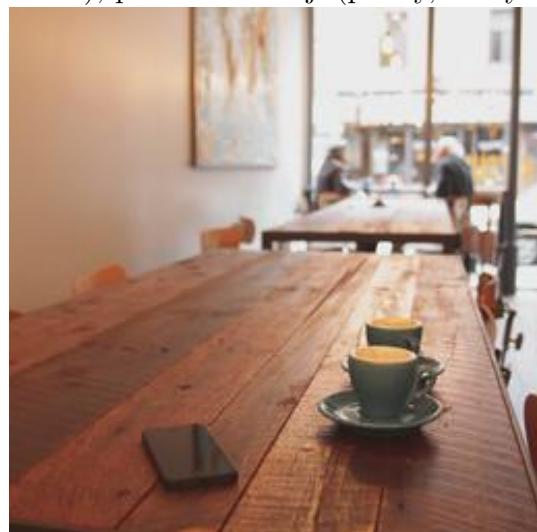
```

## 4.2 Sumowanie dwóch obrazów

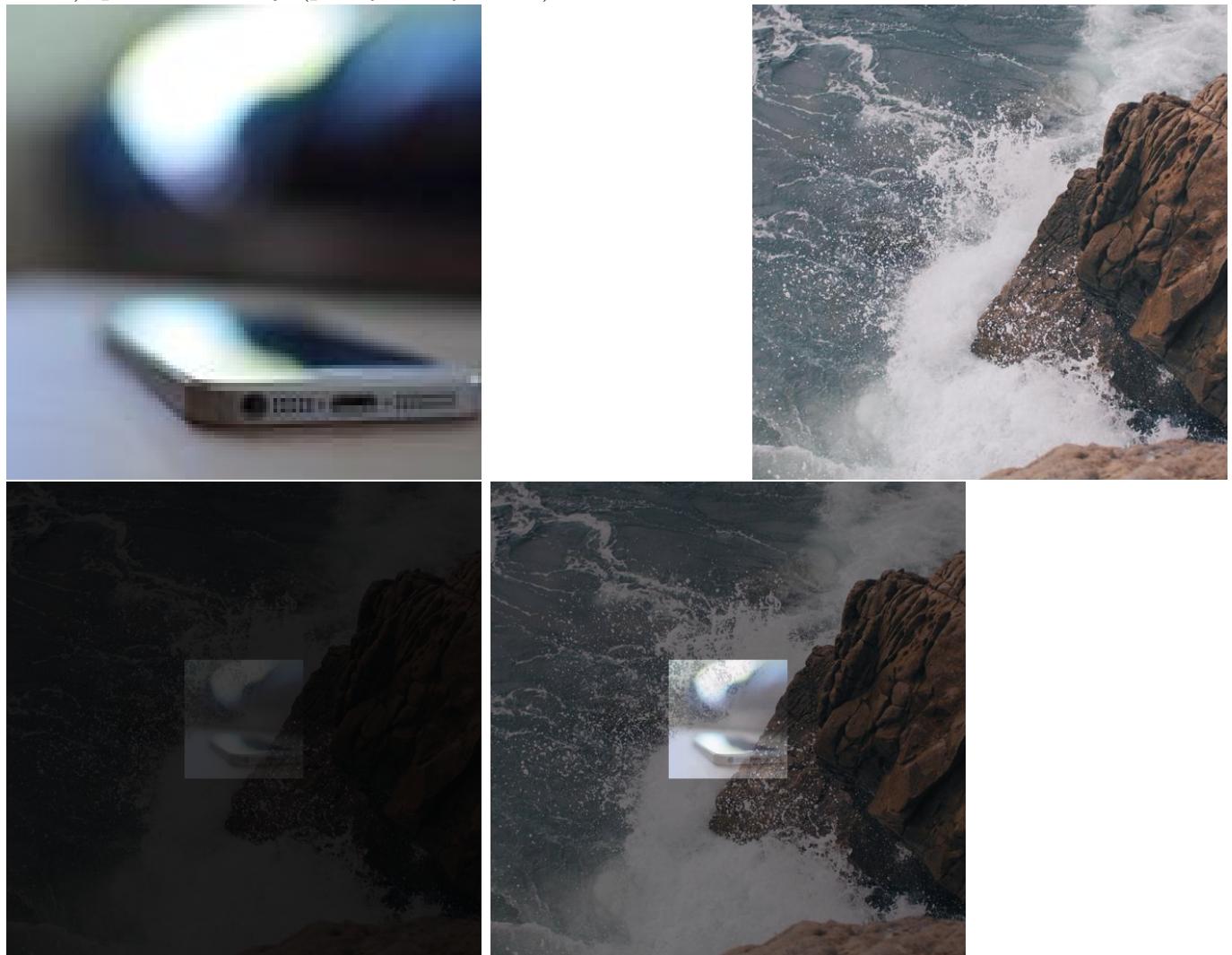
### Algorytm

### Opis

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



Rysunek 4.6: 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 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))

```

## 42 ROZDZIAŁ 4. OPERACJE SUMOWANIA ARYTMETYCZNEGO OBRAZÓW BARWOWYCH

```
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

Kod źródłowy programu

## 4.4 Mnożenie obrazu przez inny obraz

Algorytm

Opis

Kod źródłowy programu

## 4.5 Mieszanie obrazów z określonym współczynnikiem

Algorytm

Opis

Kod źródłowy programu

## 4.6 Potęgowanie obrazu (z zadana potęgą)

Algorytm

Opis

Kod źródłowy programu

#### 44 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

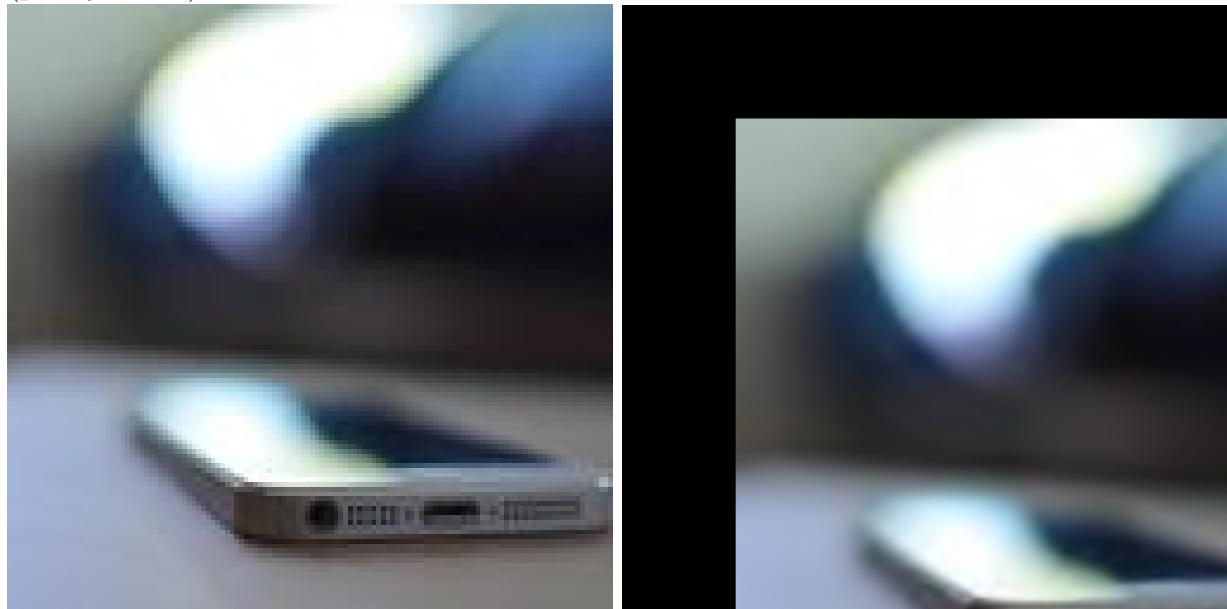
Operacja translacji wykonuje transformację geometryczną polegającą na przeniesieniu każdego z punktów obrazu wejściowego w nowe miejsce na obrazie wynikowym. Pod wpływem translacji element obrazu zlokalizowany na  $(x_1, y_1)$  zostanie przesunięty na nową pozycję  $(x_2, y_2)$ . Różnicą pomiędzy  $(x_1,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

Operacja przemieszczenia przybiera postać:

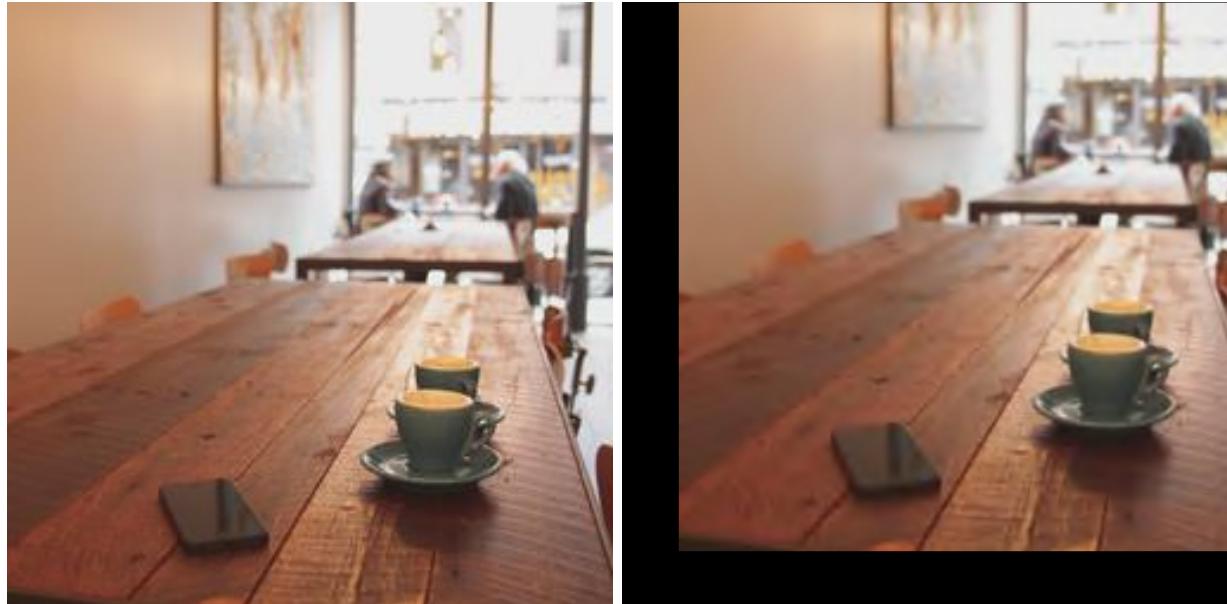
$$x_2 = x_1 + b_x \quad (5.1)$$

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

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 start')
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels24Bits()
    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]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/tGeometric.png')
    print('translation done')
```

## 5.2 Jednorodne skalowanie obrazu

### Opis

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

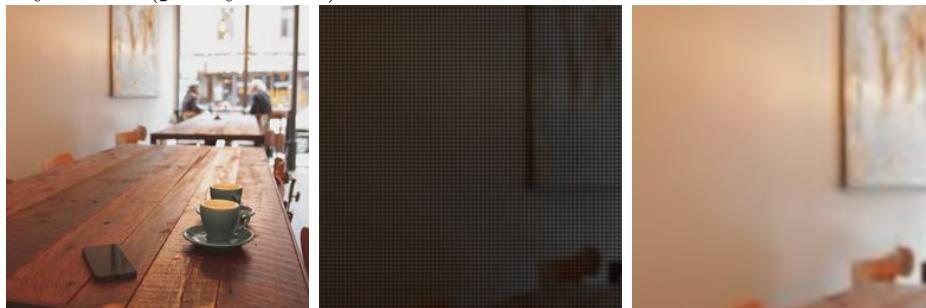
$$x_2 = S_x * x_1 \quad (5.3)$$

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

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)



### Kod źródłowy algorytmu

```

def homogeneousScaling(self, scale = 1.0):
    print('homogeneous scaling start')
    image = self.decoder.getPixels24Bits()

    print('scaling')
    result = self._scaleXY(image, scale)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/hsGeometric.png')
    print('homogeneous scaling done')

def _scaleXY(self, matrix, scale):
    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 scale * y < height and scale * x < width:
                result[int(scale * y)][int(scale * x)] = matrix[y][x]
    return result

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            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) > (height - 2)) | ((h + hOff) < 0) else (h +
                hOff)
            wSafe = w if ((w + wOff) > (width - 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)
```

### 5.3 Niejednorodne skalowanie obrazu

#### Opis

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

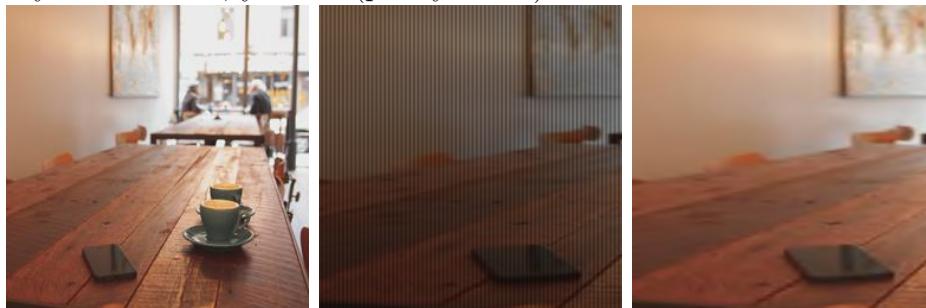
$$x_2 = S_x * x_1 \quad (5.6)$$

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

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 5.4: 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 start')
    image = self.decoder.getPixels24Bits()

    print('scaling')
    result = self._scale(image, scaleX, scaleY)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/nusGeometric.png')
    print('non-uniform scaling done')

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

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            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) > (height - 2)) | ((h + hOff) < 0) else (h + hOff)
            wSafe = w if ((w + wOff) > (width - 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)
```

## 5.4 Obracanie obrazu o dowolny kąt

### Opis

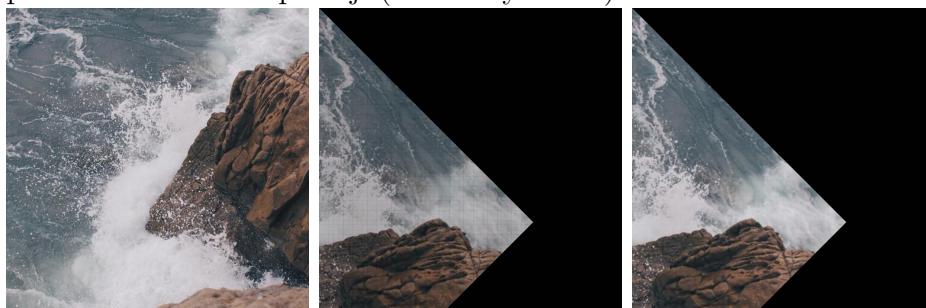
Operacja obrotu wykonywana jest wokół początku układu współrzędnych o kąt  $\varphi$  w taki sposób aby odległość od początku układu do punktu pozostała bez zmian, oraz aby pomiędzy odcinkami danych punków był kąt  $\varphi$ . Właściwości te można pozyskać dzięki wzorom:

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

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

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

Rysunek 5.5: Przed uruchomieniem algorytmu (lewy obraz), po obróceniu o kąt  $45^\circ$  (prawy obraz), po obrocie bez interpolacji (środkowy obraz)



### Kod źródłowy algorytmu

```

def rotation(self, phi):
    print('rotation start')
    image = self.decoder.getPixels24Bits()

    print('rotating')
    result = self._rotate(image, phi)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/rGeometric.png')
    print('rotation done')

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

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            r, g, b = 0, 0, 0
            n = 0

```

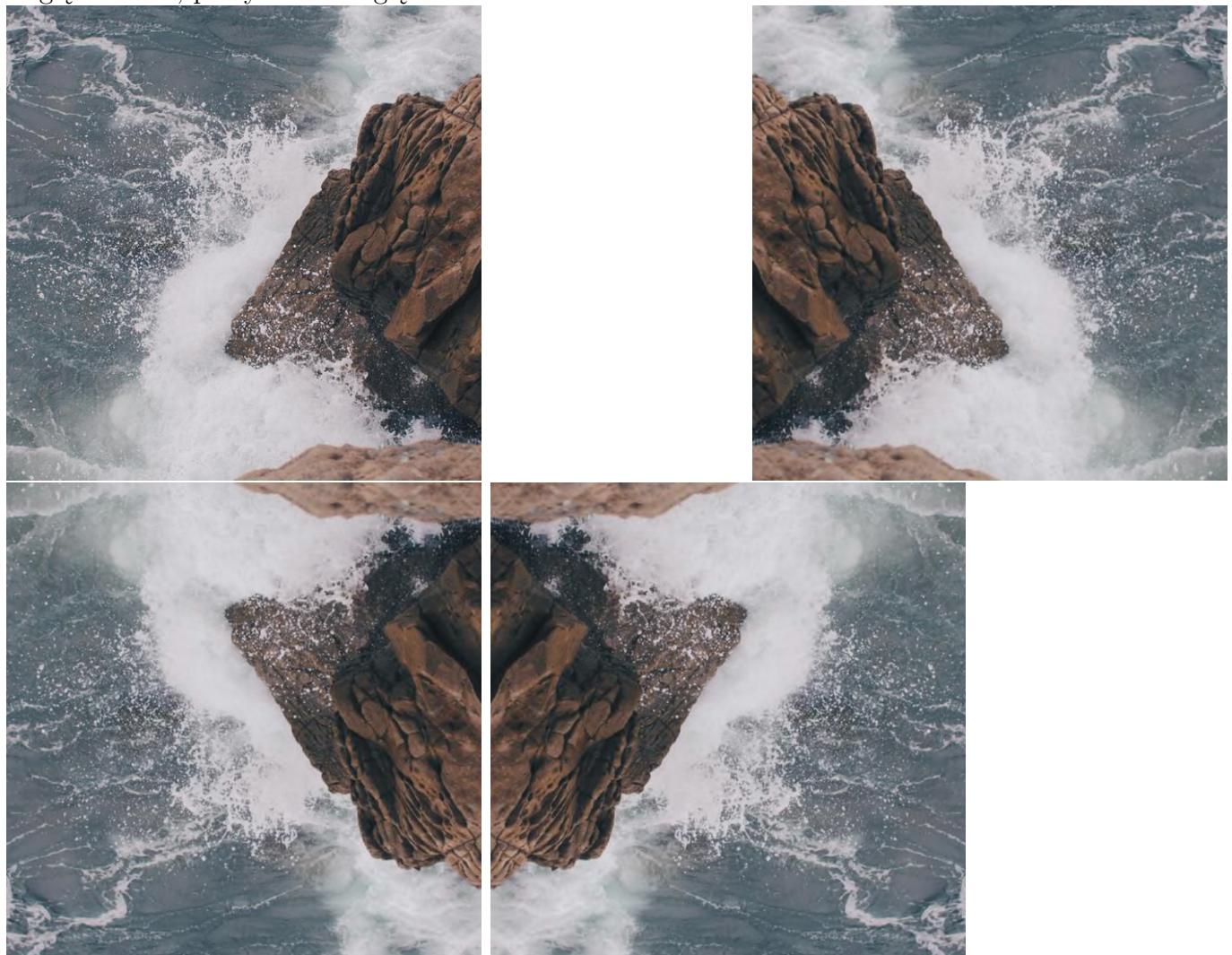
```
if (result[h, w][0] == 1) & (result[h, w][1] == 1) & (result[h, w][2] == 1):
    :
    for hOff in range(-1, 2):
        for wOff in range(-1, 2):
            hSafe = h if ((h + hOff) > (height - 2)) | ((h + hOff) < 0) else (h +
                hOff)
            wSafe = w if ((w + wOff) > (width - 2)) | ((w + wOff) < 0) else (w +
                wOff)
            if (result[hSafe, wSafe][0] > 0) | (result[hSafe, wSafe][1] > 0) | (
                result[hSafe, wSafe][2] > 0):
                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)
```

## 5.5 Symetrie względem osi układu

### Opis

Symetria osiowa 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 naszej pracy przyjmujemy, że lewa dolna krawędź obrazu znajduje się w punkcie  $(0, 0)$ .

Rysunek 5.6: 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 start')
    image = self.decoder.getPixels24Bits()

    print('symmetry operation')
    result = self._symmetryOXorOY(image, ox, oy)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-AxisSymmetry.png')
    print('axis symmetry done')

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 przebiegała oś symetrii.

Rysunek 5.7: Przed uruchomieniem algorytmu (lewy), po symetrii względem prostej  $x=356$  (środkowy), po symetrii względem prostej  $y=356$  (prawy)



### Kod źródłowy algorytmu

```

def customSymmetryX(self, ox):
    print('custom axis symmetry X start')
    if not self._validateSymmetryAxisX(ox):
        return

    print('symmetry operation X')
    image = self.decoder.getPixels24Bits()
    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]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-CustomSymmetryX.png')
    print('custom axis symmetry X done')

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 start')
    if not self._validateSymmetryAxisY(oy):
        return

    print('symmetry operation Y')
    image = self.decoder.getPixels24Bits()
    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]

img = Image.fromarray(result, mode='RGB')
img.save('Resources/Geometric-CustomSymmetryY.png')
print('custom axis symmetry Y done')

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 kopiowania pikseli do obrazu pomocniczego. Skopiowane zostają tylko te piksele, które znajdują się wewnątrz podanego zakresu.

Rysunek 5.8: Przed uruchomieniem algorytmu (lewy), po wycięciu kwadratu od piksela  $(50, 50)$  do  $(300, 300)$  (prawy)



### Kod źródłowy algorytmu

```
def crop(self, (x1, y1), (x2, y2)):
    print('cropping image start')
    image = self.decoder.getPixels24Bits()

    print('cropping')
    for x in range(x1, x2+1):
        for y in range(y1, y2+1):
            image[y, x] = (0, 0, 0)

    img = Image.fromarray(image, mode='RGB')
    img.save('Resources/Geometric-Crop.png')
    print('cropping image done')
```

## 5.8 Kopiowanie fragmentów obrazów

### Opis

Rysunek 5.9: Przed uruchomieniem algorytmu (lewy), po skopiowaniu kwadratu od piksela  $(50, 50)$  do  $(300, 300)$  (prawy)



### Kod źródłowy algorytmu

```
def copy(self, (x1, y1), (x2, y2)):
    print('copying image start')
    image = self.decoder.getPixels24Bits()
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width, 3), numpy.uint8)

    print('copying')
    for x in range(x1, x2+1):
        for y in range(y1, y2+1):
            result[y, x] = image[y, x]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-Copy.png')
    print('copying image done')
```



# Rozdział 6

## Operacje na histogramie obrazu szarego

6.1 Obliczanie histogramu

6.2 Przemieszczanie histogramu

6.3 Rozciąganie histogramu

6.4 Progowanie lokalne

6.5 Progowanie globalne



# Rozdział 7

## Operacje na histogramie obrazu barwowego

- 7.1 Obliczanie histogramu
- 7.2 Przemieszczanie histogramu
- 7.3 Rozciąganie histogramu
- 7.4 Progowanie 1-progowe lokalne
- 7.5 Progowanie wielo-progowe lokalne
- 7.6 Progowanie 1-progowe globalne
- 7.7 Progowanie wielo-progowe globalne