

Wprowadzenie do Przetwarzanie obrazów

Sprawozdanie z laboratorium

Małgorzata Wiśniewska
107418

Warszawa, 2020

Spis treści

1 Wstęp	3
1.1 Format obrazów	3
1.2 Instrukcja obsługi programu	5
2 Operacje ujednolicania obrazów	6
2.1 Ujednolicanie obrazów szarych geometryczne	7
2.2 Ujednolicanie obrazów szarych rozdzielczościowe	10
2.3 Ujednolicanie obrazów RGB geometryczne	13
2.4 Ujednolicanie obrazów RGB rozdzielczościowe	16
3 Operacje sumowania arytmetycznego obrazów szarych	19
3.1 Sumowanie obrazów szarych z określona stałą	20
3.2 Sumowanie dwóch obrazów szarych	22
3.3 Mnożenie obrazów szarych przez określona stałą	27
3.4 Mnożenie obrazu przez inny obraz	29
3.5 Mieszanie obrazów z określonym współczynnikiem	34
3.6 Potęgowanie obrazu z zadana potęgą	39
3.7 Dzielenie obrazów szarych przez zadana stałą	41
3.8 Dzielenie obrazu przez inny obraz	43
3.9 Pierwiastkowanie obrazu	47
3.10 Logarytmowanie obrazu	49
4 Operacje sumowania arytmetycznego obrazów barwowych	51
4.1 Sumowanie obrazów barwowych z określona stałą	52
4.2 Sumowanie dwóch obrazów barwowych	55
4.3 Mnożenie obrazów barwowych przez określona stałą	60
4.4 Mnożenie obrazu przez inny obraz	63
4.5 Mieszanie obrazów z określonym współczynnikiem	67
4.6 Potęgowanie obrazu z zadana potęgą	72
4.7 Dzielenie obrazów barwowych przez zadana stałą	75
4.8 Dzielenie obrazu przez inny obraz	78
4.9 Pierwiastkowanie obrazu	82
4.10 Logarytmowanie obrazu	85
5 Operacje geometryczne na obrazie	88
5.1 Przemieszczanie obrazu o zadany wektor	89
5.2 Skalowanie jednorodne obrazu	91
5.3 Skalowanie niejednorodne obrazu	93
5.4 Obracanie obrazu o dowolny kąt	95
5.5 Symetrie obrazu względem osi układu	97
5.5.1 Symetra względem osi OX	97

5.5.2	Symetria względem osi OY	98
5.5.3	Symetria względem zadanej prostej	99
5.6	Wycinanie fragmentów obrazów	102
5.7	Kopiowanie fragmentów obrazów	103
6	Operacje na histogramie obrazu szarego	105
6.1	Obliczanie histogramu	106
6.2	Przemieszczanie histogramu	108
6.3	Rozciąganie histogramu	111
6.4	Progowanie lokalne	114
6.5	Progowanie globalne	117
7	Operacje na histogramie obrazu barwowego	120
7.1	Obliczanie histogramu	121
7.2	Przemieszczanie histogramu	123
7.3	Rozciąganie histogramu	126
7.4	Progowanie 1-progowe lokalne	130
7.5	Progowanie 1 progowe globalne	134
7.6	Progowanie wieloprogowe lokalne	138
7.7	Progowanie wieloprogowe globalne	142

Rozdział 1

Wstęp

1.1 Format obrazów

Portable Network Graphics (PNG) jest formatem pliku komputerowego do przechowywania obrazów grafiki rastrowej, stworzony jako odpowiedź na roszczenia patentowe dotyczące kompresji LZW, wykorzystywanej w formatach GIF i TIFF.

PNG jest jedynym wieloplatformowym formatem bitmapowym, umożliwiającym zdefiniowanie stopnia przezroczystości (kanał α). Obsługuje 48-bitową głębię kolorów czyli 16 bitów na kanał koloru. Dzięki tym dwóm cechom możliwe jest praktycznie bezstratne zapisanie dowolnej grafiki RGB oraz RGBA. Obsługuje także osadzone profile kolorów ICC, ICM i dane EXIF. Format PNG zapisuje tylko pojedyncze pliki graficzne - nie ma animacji.

Struktura pliku PNG

Plik PNG podzielony jest na dwie części:

1. nagłówek pliku obrazowego

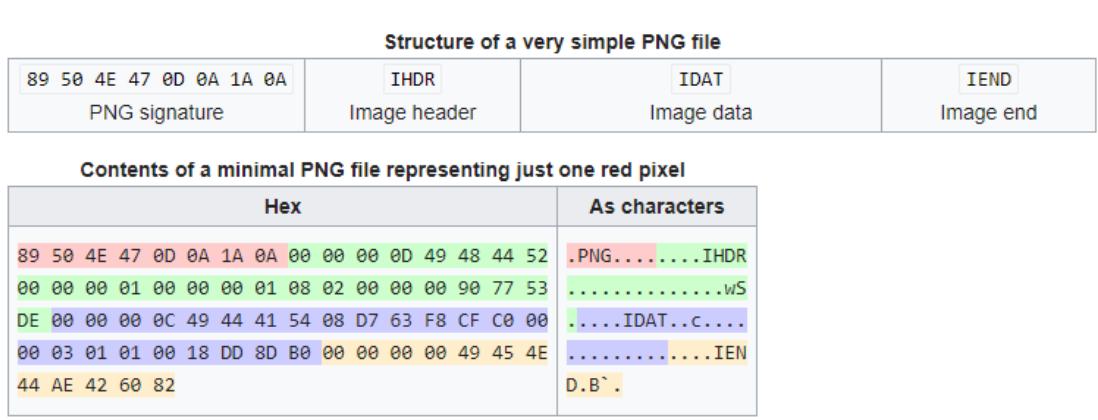
- zaczyna się 8 bajtowych podpisem
- **89** - wysoki bit ustawiony w celu rozpoznania systemów, które nie wspierają 8 bitowych danych i odróżnienia plików tekstowych od plików PNG
- **50 4E 47** - kod ASCII odpowiadający literom PNG pozwala na rozpoznanie pliku PNG ludziom
- **0D 0A** - znak końca linii w stylu DOS, aby rozpoznać DOSowo-Unixowe zakończenia linii w konwersji danych
- **1A** - bit kończący plik w DOS
- **0A** - bit kończący plik w Unix

2. katalog pliku obrazowego

- składa się z kawałków (ang. *chunks*)
- każdy kawałek składa się z czterech części: długości (4 bajty), typu kawałka (4 bajty), danych (tyle bajtów ile zadeklarowano w długości) oraz sumy kontrolnej (4 bajty, ang. *CRC, checksum*)
- najważniejsze, krytyczne dla poprawnego działania kawałki:

- **IHDR** - zawiera informacje o szerokości, wysokości, głębokości, barwie, metodzie kompresji, metodzie filtracji i metodzie przeplotu w obrazie.
- **PLTE** - zawiera listę kolorów
- **IDAT** - zawiera obraz, który może być podzielony między wiele kawałków IDAT
- **IEND** - oznacza koniec obrazu

Przykładowa struktura prostego pliku PNG



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

1.2 Instrukcja obsługi programu

Implementacja zadań została napisana w języku Python3. Wykorzystane biblioteki to numpy, matplotlib oraz Pillow. Przykład instalacji biblioteki:

```
pip install numpy  
pip install matplotlib  
pip install pillow
```

Aby wystartować program i rozpoczęć otrzymywanie wyników przetwarzania poczególnych obrazów należy uruchomić plik app.py Przykład uruchomienia programu:

```
python app.py
```

Rozdział 2

Operacje ujednolicania obrazów

Operacje ujednolicania obrazów dzieli się na dwa etapy. Pierwszym etapem jest ujednolicanie geometryczne, drugim jest ujednolicenie rozdzielczościowe. W prezentowanym programie ujednolicane są dwa obrazy, w taki sposób, że mniejszy z nich jest doprowadzany do takiego samego rozmiaru jak większy. Skutkuje to wygenerowaniem nowego obrazu o zwiększonej ilości piksli niż początkowa wartość. Dzięki zastosowaniu tego typu ujednolicania w efekcie nie następuje widoczny spadek jakości.

2.1 Ujednolicanie obrazów szarych geometryczne

Opis algorytmu

Operacje geometrycznego ujednolicania polega na wyrównaniu liczby pikseli w kolumnach i wierszach w obu obrazach, poprzez zwiększenie liczby pikseli w kolumnach i wierszach mniejszego z obrazów.

1. Wybierz największą wysokość i największą szerokość spośród obu obrazów.
2. Jeśli dany obraz ma mniejszą wysokość lub szerokość, wypełnij różnicę pikslami o wartości 1, tak, żeby wysokość i szerokość obu obrazów była równa.

Efekty wykorzystania algorytmu

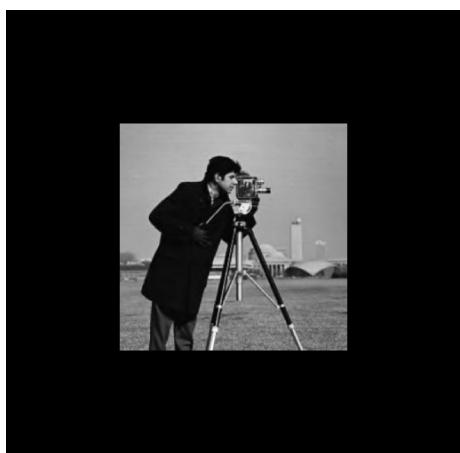


(a) Obraz 1: 256x256



(b) Obraz 2: 512x512

Rysunek 2.1: Obrazy wejściowe



(a) Obraz 1: 512x512



(b) Obraz 2: 512x512

Rysunek 2.2: Obrazy wyjściowe



(a) Obraz 1: 369x480



(b) Obraz 2: 623x640

Rysunek 2.3: Obrazy wejściowe



(a) Obraz 1: 623x640



(b) Obraz 2: 623x640

Rysunek 2.4: Obrazy wyjściowe

Kod źródłowy algorytmu

```
def geoUnificationGrey(self):
    # porównaj wielkość obrazów, jeżeli są tego samego rozmiaru
    # → nie rob nic
    if self.biggerPicture == 0 and self.smallerPicture == 0:
        print('Both_pictures_have_the_same_size')
        return 0
    # stwórz tablice zer do zapisu efektu algorytmu
    result = np.zeros((self.maxLength, self.maxWidth), np.uint8)
    startWidthIndex = int(round((self.maxWidth - self.minLength) /
    # → 2))
```

```
startLengthIndex = int(round((self.maxLength - self.minLength  
    ↪ ) / 2))  
for w in range(0, self.minLength):  
    for l in range(0, self.minLength):  
        result[l + startLengthIndex, w +  
            ↪ startWidthIndex] = self.matrix[l, w]  
#zapisz zunifikowany obraz  
path = self.ex + self.smallerPictureName + '_' + self.  
    ↪ biggerPictureName + '.png'  
self.saver.savePictureFromArray(result, 'L', path)
```

2.2 Ujednolicanie obrazów szarych rozdzielczościowe

Opis algorytmu

Operacja rozdzielczościowego ujednolicania obrazów następuje po ujednoliceniu geometrycznym obrazów wejściowych. Polega na wypełnieniu obrazu pikslami. Brakujące piksele powinny zostać zinterpolowane.

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

Efekty wykorzystania algorytmu



(a) Obraz 1: 512x512



(b) Obraz 2: 512x512

Rysunek 2.5: Obrazy wejściowe po ujednoliceniu geometrycznym



(a) Obraz 1: 512x512



(b) Obraz 2: 512x512

Rysunek 2.6: Obrazy wyjściowe bez interpolacji



(a) Obraz 1: 512x512



(b) Obraz 2: 512x512

Rysunek 2.7: Obrazy wyjściowe po interpolacji

Kod źródłowy algorytmu

```
def resolutionUnificationGrey(self):
    print('Beginning of resolution unification for two grey pictures.
          ')
    if self.biggerPicture == 0 and self.smallerPicture == 0:
        print('Both pictures have the same size')
        return 0
    scaleFactorLength = float(self.maxLength / self.minLength)
    scaleFactorWidth = float(self.maxLength / self.minLength)
    result = np.zeros((self.maxLength, self.maxLength), np.uint8)
    for l in range(self.minLength):
        for w in range(self.minLength):
            if w % 2 == 0:
                pomL = int(scaleFactorLength * l)
                pomW = int(round(scaleFactorWidth * w))
                result[pomL, pomW] = self.matrix[l, w]
            elif w % 2 == 1:
                pomL = int(round(scaleFactorLength * l))
                pomW = int(round(scaleFactorWidth * w))
                result[pomL, pomW] = self.matrix[l, w]
    # zapisz obraz bez interpolacji
    path = self.ex + self.smallerPictureName + '_' + self.
          biggerPictureName + '_withoutInterpolation.png'
    self.saver.savePictureFromArray(result, 'L', path)
    # interpolacja
    for l in range(self.maxLength):
        for w in range(self.maxLength):
            value = 0
            count = 0
            if result[l, w] == 0:
                for lOff in range(-1, 2):
                    for wOff in range(-1, 2):
```

```

lSave = l if ((l + lOff) > (self.maxLength - 2)
    ↪ ) | ((l + lOff) < 0) else (l + lOff)
wSave = w if ((w + wOff) > (self.maxWidth - 2))
    ↪ | ((w + wOff) < 0) else (w + wOff)
if result[lSave, wSave] != 0:
    value += result[lSave, wSave]
    count += 1
result[l, w] = value / count
# zapisz obraz po interpolacji
path = self.ex + self.smallerPictureName + '_' + self.
    ↪ biggerPictureName + '_withInterpolation.png'
self.saver.savePictureFromArray(result, 'L', path)
print('Finished_resolution_unification.')

```

2.3 Ujednolicanie obrazów RGB geometryczne

Opis algorytmu

Operacje geometrycznego ujednolicania polega na wyrównaniu liczby pikseli w kolumnach i wierszach w obu obrazach, poprzez zwiększenie liczby pikseli w kolumnach i wierszach mniejszego z obrazów.

1. Wybierz największą wysokość i największą szerokość spośród obu obrazów.
2. Jeśli dany obraz ma mniejszą wysokość lub szerokość, wypełnij różnicę pikslami o wartości 1 dla każdego z kanałów (R,G,B), tak, żeby wysokość i szerokość obu obrazów była równa.

Efekty wykorzystania algorytmu

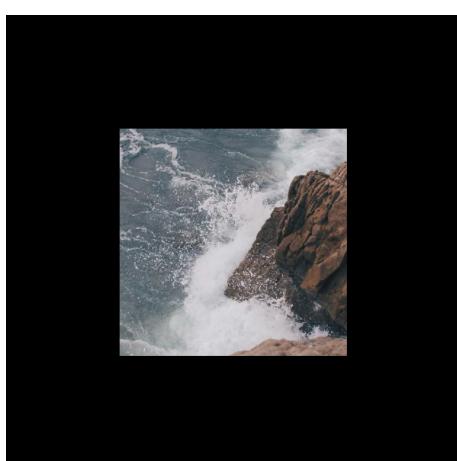


(a) Obraz 1: 512x512



(b) Obraz 2: 1025x1025

Rysunek 2.8: Obrazy wejściowe



(a) Obraz 1: 1025x1025

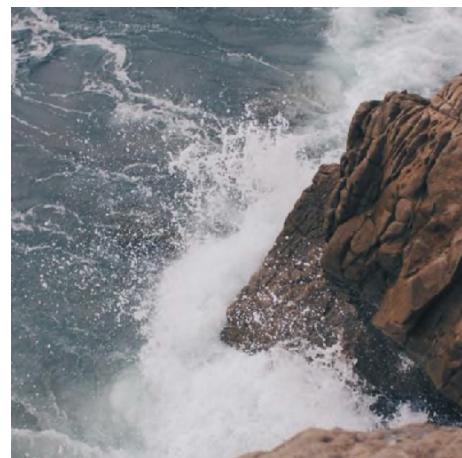


(b) Obraz 2: 1025x1025

Rysunek 2.9: Obrazy wyjściowe

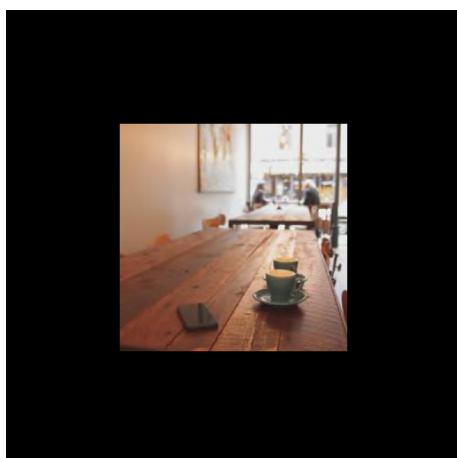


(a) Obraz 1: 256x256

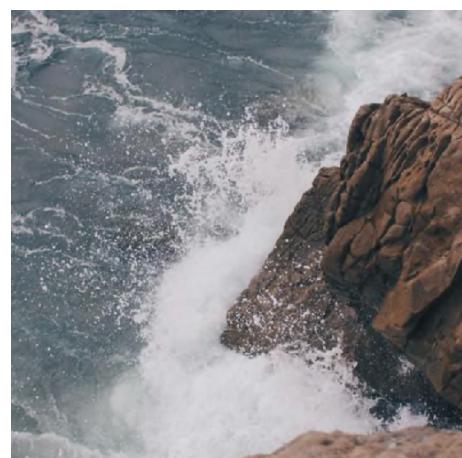


(b) Obraz 2: 512x512

Rysunek 2.10: Obrazy wejściowe



(a) Obraz 1: 512x512



(b) Obraz 2: 512x512

Rysunek 2.11: Obrazy wyjściowe

Kod źródłowy algorytmu

```
def geoUnificationRGB(self):
    if self.biggerPicture == 0 and self.smallerPicture == 0:
        print('Both\u0142pictures\u0142have\u0142the\u0142same\u0142size')
        return 0
    # stworz tablice z zerami jako odstawe dla unifikacji
    result = np.full((self.maxLength, self.maxLength, 3), 0, np.uint8)
    startWidthIndex = int(round((self.maxLength - self.minLength) / 2))
    startLengthIndex = int(round((self.maxLength - self.minLength) /
        ↪ 2))
    for w in range(0, self.minLength):
        for l in range(0, self.minLength):
            result[l + startLengthIndex, w + startWidthIndex] = self.
                ↪ matrix[w, l]
    # zapisz zunifikowany obraz
```

```
path = self.ex + self.smallerPictureName + '_' + self.  
    ↪ biggerPictureName + '.png'  
self.saver.savePictureFromArray(result, 'RGB', path)
```

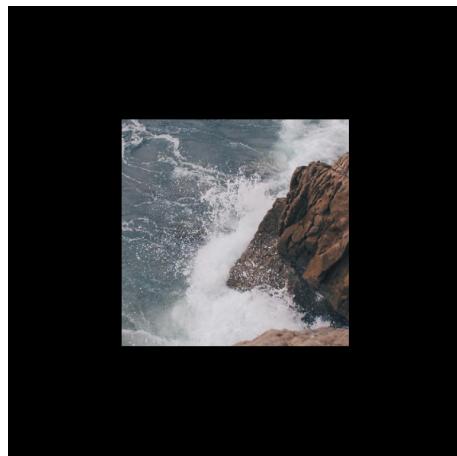
2.4 Ujednolicanie obrazów RGB rozdzielczościowe

Opis algorytmu

Operacja rozdzielczościowego ujednolicania obrazów następuje po ujednoliceniu geometrycznym obrazów wejściowych. Polega na wypełnieniu obrazu pikslami. Brakujące piksele powinny zostać zinterpolowane.

1. Wypełnij cały obraz pikslami o znanej wartości zachowując pewien odstęp między nimi, gdzie odstępem będą piksele o wartości 0.
2. Każdemu pikselowi (ze wszystkich kanałów - R, G, B) o nieznanej wartości przypisz średnią wartość znanych ($\neq 0$) pikseli z jego bezpośredniego otoczenia.

Efekty wykorzystania algorytmu

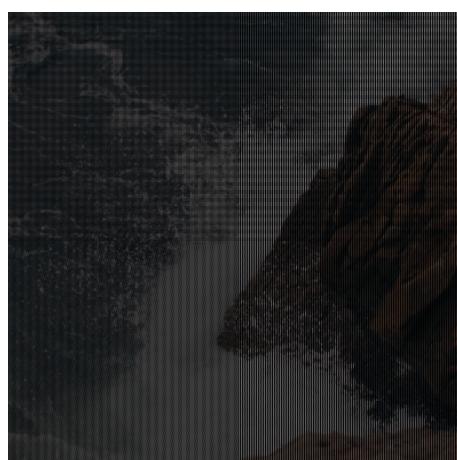


(a) Obraz 1: 1025x1025



(b) Obraz 2: 1025x1025

Rysunek 2.12: Obrazy wejściowe po ujednoliceniu geometrycznym



(a) Obraz 1: 1025x1025



(b) Obraz 2: 1025x1025

Rysunek 2.13: Obrazy wyjściowe bez interpolacji



(a) Obraz 1: 1025x1025



(b) Obraz 2: 1025x1025

Rysunek 2.14: Obrazy wyjściowe po interpolacji

Kod źródłowy algorytmu

```
def resolutionUnificationGrey(self):
    print('Beginning of resolution unification for two grey pictures.
          ')
    if self.biggerPicture == 0 and self.smallerPicture == 0:
        print('Both pictures have the same size')
        return 0
    scaleFactorLength = float(self.maxLength / self.minLength)
    scaleFactorWidth = float(self.maxLength / self.minLength)
    result = np.zeros((self.maxLength, self.maxLength), np.uint8)
    for l in range(self.minLength):
        for w in range(self.minLength):
            if w % 2 == 0:
                pomL = int(scaleFactorLength * l)
                pomW = int(round(scaleFactorWidth * w))
                result[pomL, pomW] = self.matrix[l, w]
            elif w % 2 == 1:
                pomL = int(round(scaleFactorLength * l))
                pomW = int(round(scaleFactorWidth * w))
                result[pomL, pomW] = self.matrix[l, w]
    # zapisz obraz bez interpolacji
    path = self.ex + self.smallerPictureName + '_' + self.
          biggerPictureName + '_withoutInterpolation.png'
    self.saver.savePictureFromArray(result, 'L', path)
    # interpolacja
    for l in range(self.maxLength):
        for w in range(self.maxLength):
            value = 0
            count = 0
            if result[l, w] == 0:
                for lOff in range(-1, 2):
                    for wOff in range(-1, 2):
```

```

lSave = l if ((l + lOff) > (self.maxLength - 2)
    ↪ ) | ((l + lOff) < 0) else (l + lOff)
wSave = w if ((w + wOff) > (self.maxWidth - 2))
    ↪ | ((w + wOff) < 0) else (w + wOff)
if result[lSave, wSave] != 0:
    value += result[lSave, wSave]
    count += 1
result[l, w] = value / count
# zapisz obraz po interpolacji
path = self.ex + self.smallerPictureName + '_' + self.
    ↪ biggerPictureName + '_withInterpolation.png'
self.saver.savePictureFromArray(result, 'L', path)
print('Finished_resolution_unification.')

```

Rozdział 3

Operacje sumowania arytmetycznego obrazów szarych

Operacje arytmetyczne między pikslami dwóch obrazów są wykorzystywane w wielu działach przetwarzania obrazów. Przeprowadza się je wykonując operacje na pojedynczych pikslach. Po operacjach arytmetycznych zwykle konieczne jest normalizowanie obrazu wynikowego. W zadaniach do normalizacji wykorzystano wzór:

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

3.1 Sumowanie obrazów szarych z określona stałą

Opis algorytmu

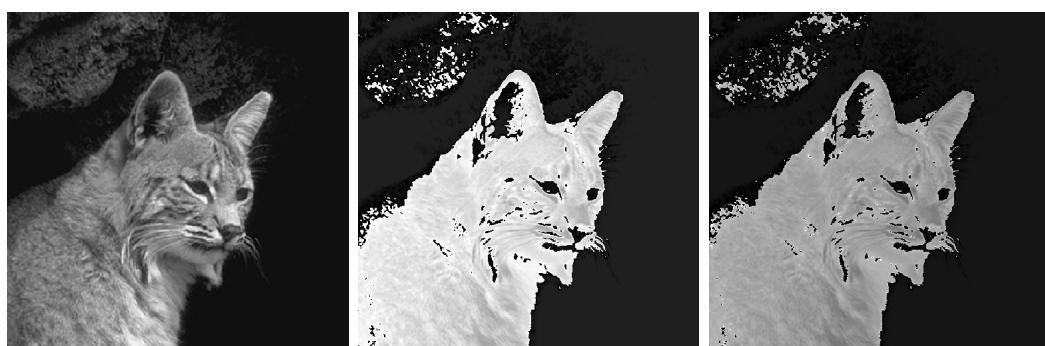
Algorytm sumowania obrazu szarego z określona stałą polega na dodaniu do każdej wartości pojedynczego piksla określonej stałej. Po operacji sumowania następuje normalizacja obrazu.

1. Policz sumy wartości każdego piksla ze stałą. Jeżeli suma przekracza 255 to konieczne jest
 - Wybranie największej sumę piksla ze stałą - Q_{max}
 - Obliczenie D_{max} ze wzoru: $D_{max}[l, w] = (Q_{max}[l, w] - 255)$
 - Obliczenie $X = D_{max}/255$
2. Policz sumę ze wzoru: $Q[l, w] = P[l, w] - (P[l, w] * X) + const - (const * X)$

Efekty wykorzystania algorytmu



Rysunek 3.1: [Od lewej] Szary obraz wejściowy, obraz po sumowaniu ze stałą 70, obraz po normalizacji



Rysunek 3.2: [Od lewej] Szary obraz wejściowy, obraz po sumowaniu ze stałą 400, obraz po normalizacji

Kod źródłowy algorytmu

```
def addConstGrey(self, constant):
    maxBitsColor = self.checkPictureBits(self.pic1)
    length, width, pictureName = self.pic1.getPictureParameters()
    matrix = self.pic1.getGreyMatrix()
    result = np.ones((length, width), np.uint8)
    sumMax = 0
    x = 0
    fmin = maxBitsColor
    fmax = 0
    for l in range(length):
        for w in range(width):
            added = matrix[l, w] + constant
            if sumMax < added:
                sumMax = added
    if sumMax > maxBitsColor:
        x = (sumMax - maxBitsColor) / maxBitsColor
    for l in range(length):
        for w in range(width):
            # Rounded up and assignment of value to the result matrix
            pom = (matrix[l, w] - (matrix[l, w] * x)) + (constant - (
                ↪ constant * x))
            result[l, w] = np.ceil(pom)
            # Search for maximum and minimum
            if fmin > pom:
                fmin = pom
            if fmax < pom:
                fmax = pom
    # save picture with added constant to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
    for l in range(length):
        for w in range(width):
            result[l, w] = maxBitsColor*((result[l, w] - fmin) / (fmax
                ↪ - fmin))
    # save picture with added constant to png file (with
    ↪ normalization)
    path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '_normalized.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
```

3.2 Sumowanie dwóch obrazów szarych

Opis algorytmu

Algorytm sumowania obrazu szarego z drugim obrazem szarym jest określone tylko o tych samych wymiarach $M \times N$ i strukturze ich macierzy. Algorytm sumowania obrazu z obrazem polega na dodaniu do wartości piksla z pierwszego obrazu, wartości odpowiadającego piksla z drugiego obrazu. Po operacji sumowania następuje normalizacja obrazu. Operacja dodawania obrazów są użyteczne przy uśrednianiu obrazów, w celu zredukowania na nich szumu.

1. Policz sumy wartości każdego piksla obrazu pierwszego $P1[l, w]$ z odpowiadającym piksem drugiego obrazu $P2[l, w]$. Jeżeli suma przekracza 255 to konieczne jest
 - Wybranie największej sumy Q_{max}
 - Obliczenie D_{max} ze wzoru: $D_{max}[l, w] = (Q_{max}[l, w] - 255)$
 - Obliczenie $X = D_{max}/255$
2. Policz sumę ze wzoru:

$$Q[l, w] = P1[l, w] - (P1[l, w] * X) + P2[l, w] - (P2[l, w] * X)$$

Efekty wykorzystania algorytmu



Rysunek 3.3: [Od lewej, rząd 1] Szary obraz wejściowy 1, szary obraz wejściowy 2
[Od lewej, rząd 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji



Rysunek 3.4: [Od lewej, rząd 1] Szary obraz wejściowy 1, szary obraz wejściowy 2
[Od lewej, rząd 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```
def addPictureGrey(self):
    if self.checkPictureBits(self.pic1) == self.checkPictureBits(self
        ↪ .pic2):
        maxBitsColor = self.checkPictureBits(self.pic2)
    # check if pictures have same sizes, if not unify them
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.pic1
        ↪ , self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, pictureName1 = self.pic1.getPictureParameters()
    matrix1 = self.pic1.getGreyMatrix()
    length2, width2, pictureName2 = self.pic2.getPictureParameters()
    matrix2 = self.pic2.getGreyMatrix()
    pictureName1 = tempName

    sumMax = 0
    x = 0
    fmax = 0
    fmin = maxBitsColor

    result = np.zeros((length1, width1), np.uint8)

    for l in range(length1):
        for w in range(width1):
            added = int(matrix1[l, w]) + int(matrix2[l, w])
            if sumMax < added:
                sumMax = added

    if sumMax > maxBitsColor:
        x = (sumMax - maxBitsColor) / maxBitsColor

    for l in range(length1):
        for w in range(width1):
            # Rounded up and assignment of value to the result matrix
            pom = int(matrix1[l, w] - (matrix1[l, w] * x)) + int(
                ↪ matrix2[l, w] - (matrix2[l, w] * x))
            result[l, w] = np.ceil(int(pom))
            # Search for maximum and minimum
            if fmin > pom:
                fmin = pom
            if fmax < pom:
                fmax = pom

    # save picture with added constant to png file (without
        ↪ normalization)
```

```

path = self.ex + str(pictureName1) + '_added_' + str(pictureName2
    ↪ ) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

normalized = np.zeros((length1, width1), np.uint8)
for l in range(length1):
    for w in range(width1):
        normalized[l, w] = maxBitsColor*((result[l, w] - fmin) / (
            ↪ fmax - fmin))

# save picture with added constant to png file (with
    ↪ normalization)
path = self.ex + str(pictureName1) + '_added_' + str(pictureName2
    ↪ ) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

3.3 Mnożenie obrazów szarych przez określoną stałą

Opis algorytmu

Algorytm mnożenia obrazu szarego przez określoną stałą polega na przemnożeniu każdego elementu obrazu (piksla) przez określoną stałą (skalar). Dla wszystkich piksli w obrazie wykonaj:

1. Jeżeli wartość piksla jest równa 255 to składowa wynikowa otrzymuje wartość stałej.
2. Jeżeli wartość piksla jest równa 0 to składowa wynikowa otrzymuje wartość 0.
3. Jeżeli wartość piksla jest inną niż 255 lub 0 to składowa wynikowa otrzymuje wartość poprzez pomnożenie wartości piksla przez skalar, podzielenie przez 255 i zaokrąglenie do liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 3.5: [Od lewej] Szary obraz wejściowy, obraz po sumowaniu ze stałą 100, obraz po normalizacji



Rysunek 3.6: [Od lewej] Szary obraz wejściowy, obraz po sumowaniu ze stałą 100, obraz po normalizacji

Kod źródłowy algorytmu

```

def multiplyConstGrey(self, constant):
    maxBitsColor = self.checkPictureBits(self.pic1)
    length, width, matrix, pictureName = self.getPictureParameters(
        ↪ self.pic1)
    result = np.ones((length, width), np.uint8)
    fmin = maxBitsColor
    fmax = 0
    for l in range(length):
        for w in range(width):
            pom = matrix[l, w]
            if pom == maxBitsColor:
                result[l, w] = maxBitsColor
            elif pom == 0:
                result[l, w] = 0
            else:
                result[l, w] = np.ceil(((matrix[l, w] * constant) /
                    ↪ maxBitsColor))
            # Search for maximum and minimum
            if fmin > result[l, w]:
                fmin = result[l, w]
            if fmax < result[l, w]:
                fmax = result[l, w]
    # save picture with added constant to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
    for l in range(length):
        for w in range(width):
            result[l, w] = maxBitsColor*((result[l, w] - fmin) / (fmax
                ↪ - fmin))
    # save picture with added constant to png file (with
    ↪ normalization)
    path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '_normalized.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)

```

3.4 Mnożenie obrazu przez inny obraz

Opis algorytmu

Algorytm mnożenia obrazu szarego przez drugi szary obraz o tych samych wymiarach $M \times N$ i strukturze ich macierzy polega na przemnożeniu wartości piksla z pierwszego obrazu przez wartość odpowiadającego piksla z drugiego obrazu. Po operacji mnożenia następuje normalizacja obrazu. Dla każdego piksla pierwszego obrazu wykonaj następujące czynności:

1. Jeżeli wartość piksla $P1[l, w]$ jest równa 255 to składowa wynikowa otrzymuje wartość odpowiadającego piksla drugiego obrazu $P2[l, w]$.
2. Jeżeli wartość piksla $P1[l, w]$ jest równa 0 to składowa wynikowa otrzymuje wartość 0.
3. Jeżeli wartość piksla $P1[l, w]$ jest inna niż 255 lub 0 to składowa wynikowa otrzymuje wartość poprzez pomnożenie wartości piksla $P1[l, w]$ przez odpowiadający piksel $P2[l, w]$, podzielenie przez 255 i zaokrąglenie do liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 3.7: [Od lewej, rząd 1] Szary obraz wejściowy 1, szary obraz wejściowy 2
[Od lewej, rząd 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji



Rysunek 3.8: [Od lewej, rząd 1] Szary obraz wejściowy 1, szary obraz wejściowy 2
[Od lewej, rząd 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```
def multiplyPicturesGrey(self):
    if self.checkPictureBits(self.pic1) == self.checkPictureBits(self
        ↪ .pic2):
        maxBitsColor = self.checkPictureBits(self.pic2)
    # check if pictures have same sizes, if not unify them
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.pic1
        ↪ , self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, matrix1, pictureName1 = self.
        ↪ getPictureParameters(self.pic1)
    length2, width2, matrix2, pictureName2 = self.
        ↪ getPictureParameters(self.pic2)
    pictureName1 = tempName

    result = np.ones((length1, width1), np.uint8)

    fmin = maxBitsColor
    fmax = 0

    for l in range(length1):
        for w in range(width1):
            if matrix1[l, w] == maxBitsColor:
                result[l, w] = matrix2[l, w]
            elif matrix1[l, w] == 0:
                result[l, w] = 0
            else:
                result[l, w] = np.ceil(((int(matrix1[l, w]) * int(
                    ↪ matrix2[l, w])) / maxBitsColor))
            # Search for maximum and minimum
            if fmin > result[l, w]:
                fmin = result[l, w]

            if fmax < result[l, w]:
                fmax = result[l, w]

    # save picture multiplied by picture to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName1) + '_multiplied_' + str(
        ↪ pictureName2) + '.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)

    for l in range(length1):
        for w in range(width1):
```

```
    result[l, w] = maxBitsColor*((result[l, w] - fmin) / (fmax  
    ↪ - fmin))  
  
# save picture multiplied by picture to png file (with  
    ↪ normalization)  
path = self.ex + str(pictureName1) + '_multiplied_' + str(  
    ↪ pictureName2) + '_normalized.png'  
self.saver.savePictureFromArray(result, self.pictureType, path)
```

3.5 Mieszanie obrazów z określonym współczynnikiem

Opis algorytmu

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami α i $1 - \alpha$ według wzoru:

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

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

1. Weź dwa obrazy szare o takim samym rozmiarze (po ujednoliceniu rozdzielczościowym) P_1 i P_2 .
2. Określ współczynnik mieszania obrazów α wyrażony jako liczba rzeczywista z przedziału $[0, 1]$, gdzie 0 reprezentuje przezroczystość, a 1 reprezentuje nieprzejroczystość.
3. Dla wszystkich piksli w obrazach wejściowych wykonaj:

$$Q[l, w] = \alpha * P_1[l, w] + (1 - \alpha) * P_2[l, w]$$

Efekty wykorzystania algorytmu



Rysunek 3.9: [Od lewej, rząd 1] Szary obraz wejściowy 1, szary obraz wejściowy 2 [Od lewej, rząd 2] Obraz po mieszaniu ze współczynnikiem $\alpha = 0.8$, obraz po normalizacji



Rysunek 3.10: [Od lewej, rzad 1] Szary obraz wejściowy 1, szary obraz wejściowy 2 [Od lewej, rzad 2] Obraz po mieszaniu ze współczynnikiem $\alpha = 0.3$, obraz po normalizacji

Kod źródłowy algorytmu

```
def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationGrey(self.name1, self.
                                                ↪ name2)
    resolutionUni.resolutionUnificationGrey()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2
def blendPictures(self, alfa):
    if self.checkPictureBits(self.pic1) == self.checkPictureBits(
        ↪ self.pic2):
        maxBitsColor = self.checkPictureBits(self.pic2)
    # check if pictures have same sizes, if not unify them
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.
                                                               ↪ pic1, self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, matrix1, pictureName1 = self.
        ↪ getPictureParameters(self.pic1)
    length2, width2, matrix2, pictureName2 = self.
        ↪ getPictureParameters(self.pic2)
    pictureName1 = tempName

    result = np.ones((length1, width1), np.uint8)

    fmin = maxBitsColor
    fmax = 0

    for l in range(length1):
        for w in range(width1):
            pom = float(matrix1[l, w]) * alfa + float(matrix2[l, w
                ↪ ]) * (1 - alfa)
            result[l, w] = np.ceil(pom)

            # Search for maximum and minimum
            if fmin > pom:
                fmin = pom
            if fmax < pom:
                fmax = pom

    # save picture multiplied by picture to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName1) + '_blended_' + str(alfa)
    ↪ + '_' + str(pictureName2) + '.png'
```

```
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)

for l in range(length1):
    for w in range(width1):
        result[l, w] = maxBitsColor*((result[l, w] - fmin) / (
            ↪ fmax - fmin))

# save picture multiplied by picture to png file (with
    ↪ normalization)
path = self.ex + str(pictureName1) + '_blended_' + str(alfa)
    ↪ + '_' + str(pictureName2) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)
```

3.6 Potęgowanie obrazu z zadaną potęgą

Opis algorytmu

Algorytm potęgowania obrazu szarego do określonej stałej jest szczególnym przypadkiem mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru:

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

Efekty wykorzystania algorytmu



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



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

Kod źródłowy algorytmu

```
def raiseToPower(self, power):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    result = np.zeros((length, width), np.uint8)

    maxPicture = 0
```

```

fmin = maxBitsColor
fmax = 0

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if maxPicture < pom:
            maxPicture = pom

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if pom == maxBitsColor:
            pom = maxBitsColor
        elif pom == 0:
            pom = 0
        else:
            pom = np.power(int(pom) / maxPicture, power) *
                  ↪ maxBitsColor
        result[l, w] = np.ceil(pom)
    # Search for maximum and minimum
    if fmin > pom:
        fmin = pom

    if fmax < pom:
        fmax = pom

# save picture raised to constant power to png file (without
    ↪ normalization)
path = self.ex + str(pictureName) + '_power_' + str(power) +
      ↪ '.png'
self.saver.savePictureFromArray(result, self.pictureType,
                                ↪ path)

for l in range(length):
    for w in range(width):
        result[l, w] = maxBitsColor * ((result[l, w] - fmin) /
                                       ↪ (fmax - fmin))

# save picture raised to constant power to png file (with
    ↪ normalization)
path = self.ex + str(pictureName) + '_power_' + str(power) +
      ↪ '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType,
                                ↪ path)

```

3.7 Dzielenie obrazów szarych przez zadaną stałą

Opis algorytmu

Algorytm dzielenia obrazu szarego przez określoną stałą służy korekcji cieniowania między poziomami szarości obrazu. Aby zastosować algorytm dla każdego piksla obrazu wykonaj:

1. Policz sumę piksla ze stałą.
2. Spośród obliczonych sum wybierz Q_{max} - największą sumę.
3. Wartość wynikową policz z następującego wzoru:

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

Wynik zaokrągluj w góre do najbliższej liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 3.13: [Od lewej] Szary obraz wejściowy, obraz po podzieleniu przez stałą $\alpha = 3$, obraz po normalizacji



Rysunek 3.14: [Od lewej] Szary obraz wejściowy, obraz po podzieleniu przez stałą $\alpha = 15$, obraz po normalizacji

Kod źródłowy algorytmu

```
def divideConstGrey(self, constant):
    length, width, matrix, pictureName = self.
        ↪ getPictureParameters(self.pic1)
    result = np.zeros((length, width), np.uint8)

    fmin = maxBitsColor
    fmax = 0
    sumMax = 0

    for l in range(length):
        for w in range(width):
            added = int(matrix[l, w]) + int(constant)
            if sumMax < added:
                sumMax = added

    for l in range(length):
        for w in range(width):
            added = int(matrix[l, w]) + int(constant)
            pom = (added * maxBitsColor) / sumMax
            result[l, w] = np.ceil(pom)
            if fmin > pom:
                fmin = pom
            if fmax < pom:
                fmax = pom

    # save picture with added constant to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName) + '_dividedBy_' + str(
        ↪ constant) + '.png'
    self.saver.savePictureFromArray(result, self.pictureType,
        ↪ path)

    for l in range(length):
        for w in range(width):
            result[l, w] = maxBitsColor*((result[l, w] - fmin) / (
                ↪ fmax - fmin))

    # save picture with added constant to png file (with
    ↪ normalization)
    path = self.ex + str(pictureName) + '_dividedBy_' + str(
        ↪ constant) + '_normalized.png'
    self.saver.savePictureFromArray(result, self.pictureType,
        ↪ path)
```

3.8 Dzielenie obrazu przez inny obraz

Opis algorytmu

1. Policz sumę piksla obrazu P_1 z odpowiadającym pikslem obrazu P_2 .
2. Spośród obliczonych sum wybierz Q_{max} - największą sumę.
3. Wartość wynikową policz z następującego wzoru:

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

Wynik zaokrąglaj w góre do najbliższej liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 3.15: [Od lewej, rzad 1] Szary obraz wejściowy 1, szary obraz wejściowy 2 [Od lewej, rzad 2] Obraz powstały w wyniku dzielenia obrazów 1 i 2, obraz po normalizacji



Rysunek 3.16: [Od lewej, rzad 1] Szary obraz wejściowy 1, szary obraz wejściowy 2 [Od lewej, rzad 2] Obraz powstały w wyniku dzielenia obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```
def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationGrey(self.name1, self.
                                                ↪ name2)
    resolutionUni.resolutionUnificationGrey()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2

def dividePicturesGrey(self):
    if self.checkPictureBits(self.pic1) == self.checkPictureBits(
        ↪ self.pic2):
        maxBitsColor = self.checkPictureBits(self.pic2)
    # check if pictures have same sizes, if not unify them
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.
                                                               ↪ pic1, self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, matrix1, pictureName1 = self.
        ↪ getPictureParameters(self.pic1)
    length2, width2, matrix2, pictureName2 = self.
        ↪ getPictureParameters(self.pic2)
    pictureName1 = tempName

    result = np.ones((length1, width1), np.uint8)

    fmin = maxBitsColor
    fmax = 0
    sumMax = 0

    for l in range(length1):
        for w in range(width1):
            added = int(matrix1[l, w]) + int(matrix2[l, w])
            if sumMax < added:
                sumMax = added

    for l in range(length1):
        for w in range(width1):
            added = int(matrix1[l, w]) + int(matrix2[l, w])
            pom = (added * maxBitsColor) / sumMax
            result[l, w] = np.ceil(pom)
            if fmin > pom:
                fmin = pom
            if fmax < pom:
                fmax = pom
```

```

# save picture multiplied by picture to png file (without
    ↳ normalization)
path = self.ex + str(pictureName1) + '_dividedBy_' + str(
    ↳ pictureName2) + '.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↳ path)

for l in range(length1):
    for w in range(width1):
        result[l, w] = maxBitsColor*((result[l, w] - fmin) / (
            ↳ fmax - fmin))

# save picture multiplied by picture to png file (with
    ↳ normalization)
path = self.ex + str(pictureName1) + '_dividedBy_' + str(
    ↳ pictureName2) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↳ path)

```

3.9 Pierwiastkowanie obrazu

Opis algorytmu

Algorytm pierwiastkowania obrazu szarego jest szczególnym przypadkiem wykorzystania algorytmu potęgowania obrazu przez określoną stałą, która jest ułamkiem. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru:

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

Efekty wykorzystania algorytmu



Rysunek 3.17: [Od lewej] Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 1/3$, obraz po normalizacji



Rysunek 3.18: [Od lewej] Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 1/2$, obraz po normalizacji

Kod źródłowy algorytmu

```
def rootGrey(self, power):
    factorial = 1 / power
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    result = np.zeros((length, width), np.uint8)
```

```

maxPicture = 0
fmin = maxBitsColor
fmax = 0

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if maxPicture < pom:
            maxPicture = pom

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if pom == maxBitsColor:
            pom = maxBitsColor
        elif pom == 0:
            pom = 0
        else:
            pom = np.power(int(pom) / maxPicture, factorial) *
            ↪ maxBitsColor
        result[l, w] = np.ceil(pom)
    # Search for maximum and minimum
    if fmin > pom:
        fmin = pom

    if fmax < pom:
        fmax = pom

# save picture raised to constant power to png file (without
    ↪ normalization)
path = self.ex + str(pictureName) + '_root_' + str(factorial)
    ↪ + '.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)

for l in range(length):
    for w in range(width):
        result[l, w] = maxBitsColor * ((result[l, w] - fmin) /
            ↪ (fmax - fmin))

# save picture raised to constant power to png file (with
    ↪ normalization)
path = self.ex + str(pictureName) + '_root_' + str(factorial)
    ↪ + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)

```

3.10 Logarytmowanie obrazu

Opis algorytmu

Algorytm logarytmowania obrazu szarego powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru:

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

Przsunięcie funkcji obrazu o 1 w góre wynika z nieokreśloności logarytmu w zerze.

Efekty wykorzystania algorytmu



Rysunek 3.19: [Od lewej] Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji



Rysunek 3.20: [Od lewej] Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji

Kod źródłowy algorytmu

```
def logharitmGrey(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    result = np.zeros((length, width), np.uint8)
```

```

maxPicture = 0
fmin = maxBitsColor
fmax = 0

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if maxPicture < pom:
            maxPicture = pom

for l in range(length):
    for w in range(width):
        pom = matrix[l, w]
        if pom == 0:
            pom = 0
        else:
            pom = (np.log(1 + pom) / np.log(1 + maxPicture)) *
                ↪ maxBitsColor
        result[l, w] = np.ceil(pom)
        # Search for maximum and minimum
        if fmin > pom:
            fmin = pom

        if fmax < pom:
            fmax = pom

# save picture raised to constant power to png file (without
    ↪ normalization)
path = self.ex + str(pictureName) + '_log.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)

for l in range(length):
    for w in range(width):
        result[l, w] = maxBitsColor * ((result[l, w] - fmin) /
            ↪ (fmax - fmin))

# save picture raised to constant power to png file (with
    ↪ normalization)
path = self.ex + str(pictureName) + '_log_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)

```

Rozdział 4

Operacje sumowania arytmetycznego obrazów barwowych

Operacje arytmetyczne między pikslami dwóch obrazów są wykorzystywane w wielu działach przetwarzania obrazów. Przeprowadza się je wykonując operacje na pojedynczych pikslach. Po operacjach arytmetycznych zwykle konieczne jest normalizowanie obrazu wynikowego. W zadaniach poruszamy się po przestrzeni barw RGB, a do normalizacji wykorzystano wzór:

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

4.1 Sumowanie obrazów barwowych z określona stałą

Opis algorytmu

Algorytm sumowania obrazu barwowego z określona stałą polega na dodaniu do 3 kanałów (RGB) każdego pojedynczego piksla określonej stałej. Po operacji sumowania następuje normalizacja obrazu.

1. Policz sumy wartości każdego piksla w każdym z kanałów ze stałą. Jeżeli suma przekracza 255 to konieczne jest

- Wybranie największej sumę piksla ze stałą - Q_{max}
- Obliczenie D_{max} ze wzoru: $D_{max}[l, w] = (Q_{max}[l, w] - 255)$
- Obliczenie $X = D_{max}/255$

2. Policz sumę ze wzoru:

$$Q_R[l, w] = P_R[l, w] - (P_R[l, w] * X) + const - (const * X) - 1$$

$$Q_G[l, w] = P_G[l, w] - (P_G[l, w] * X) + const - (const * X) - 1$$

$$Q_B[l, w] = P_B[l, w] - (P_B[l, w] * X) + const - (const * X) - 1$$

Efekty wykorzystania algorytmu



Rysunek 4.1: [Od lewej] Barwowy obraz wejściowy, obraz po sumowaniu ze stałą 20, obraz po normalizacji



Rysunek 4.2: [Od lewej] Szary obraz wejściowy, obraz po sumowaniu ze stałą 200, obraz po normalizacji

Kod źródłowy algorytmu

```

def addConstRGB(self, constant):
    length, width, pictureName = self.pic1.getPictureParameters()
    matrix = self.pic1.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)

    sumMax = 0
    x = 0
    fmin = 255
    fmax = 0

    for l in range(length):
        for w in range(width):
            R = int(matrix[l, w][0]) + int(constant)
            G = int(matrix[l, w][1]) + int(constant)
            B = int(matrix[l, w][2]) + int(constant)
            if sumMax < max([R, G, B]):
                sumMax = max([R, G, B])

    if sumMax > 255:
        x = (sumMax - 255) / 255

    for l in range(length):
        for w in range(width):
            R = (int(matrix[l, w][0]) - (int(matrix[l, w][0]) * x)) +
                ↪ (int(constant) - (int(constant) * x))
            G = (int(matrix[l, w][1]) - (int(matrix[l, w][1]) * x)) +
                ↪ (int(constant) - (int(constant) * x))
            B = (int(matrix[l, w][2]) - (int(matrix[l, w][2]) * x)) +
                ↪ (int(constant) - (int(constant) * x))
            result[w, l][0] = np.ceil(R)
            result[w, l][1] = np.ceil(G)
            result[w, l][2] = np.ceil(B)
            # Search for maximum and minimum
            if fmin > min([R, G, B]):
                fmin = min([R, G, B])

```

```

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

# save picture with added constant to png file (without
    → normalization)
path = self.ex + str(pictureName) + '_constant_' + str(constant)
    → + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length):
    for w in range(width):
        result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
            → - fmin))
        result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
            → - fmin))
        result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
            → - fmin))

# save picture with added constant to png file (with
    → normalization)
path = self.ex + str(pictureName) + '_constant_' + str(constant)
    → + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

4.2 Sumowanie dwóch obrazów barwowych

Opis algorytmu

Algorytm sumowania obrazu barwowego z drugim obrazem barwowy jest określone tylko dla obrazów o tych samych wymiarach $M \times N$ i strukturze ich macierzy. Algorytm sumowania obrazu z obrazem polega na dodaniu do wartości piksla z pierwszego obrazu, wartości odpowiadającego piksla z drugiego obrazu. Po operacji sumowania następuje normalizacja obrazu. Operacja dodawania obrazów są użyteczne przy uśrednianiu obrazów, w celu zredukowania na nich szumu.

1. Policz sumy wartości składowych barwowych każdego piksla obrazu pierwszego $P1[l, w]$ z odpowiadającym piksem drugiego obrazu $P2[l, w]$
2. Wybierz największą sumę $Q_{max} = \max(R_S, G_S, B_S)$
3. Policz sumę ze wzoru:

$$Q_R[l, w] = (R_S * 255) / Q_{max}$$

$$Q_G[l, w] = (G_S * 255) / Q_{max}$$

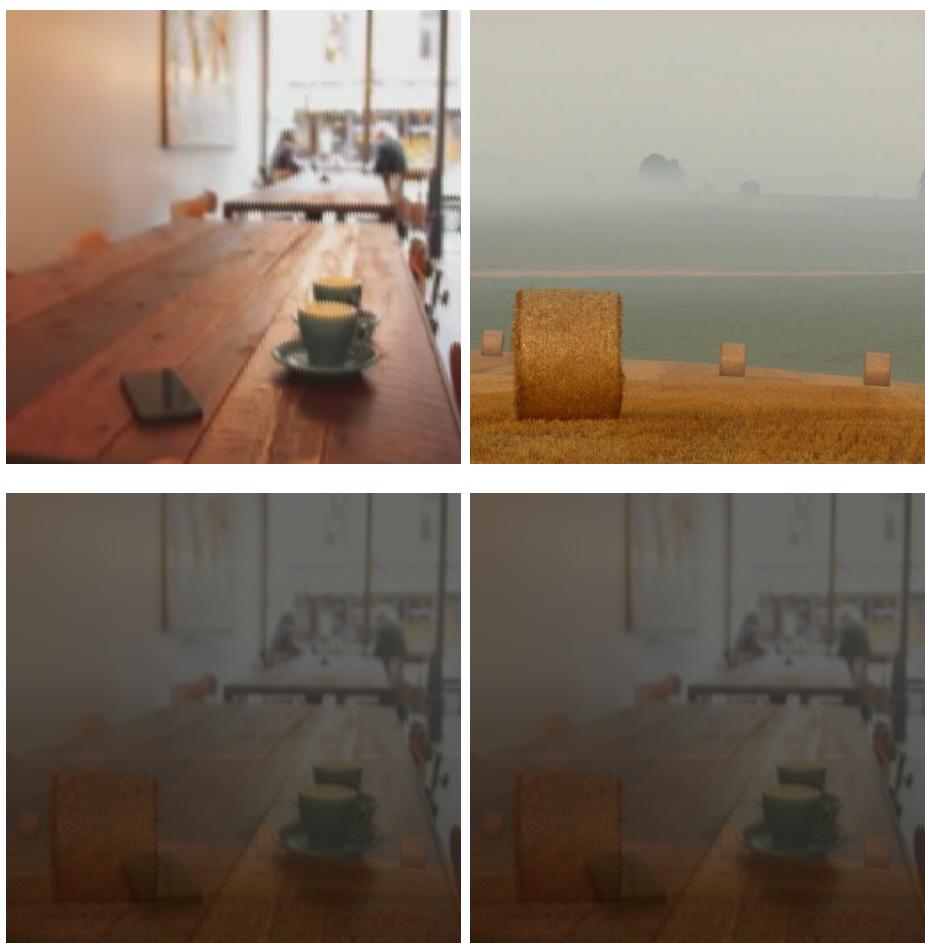
$$Q_B[l, w] = (B_S * 255) / Q_{max}$$

Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 4.3: [Od lewej, rzad 1] Barwowy obraz wejściowy 1, Barwowy obraz wejściowy 2 [Od lewej, rzad 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji



Rysunek 4.4: [Od lewej, rząd 1] Barwowy obraz wejściowy 1, Barwowy obraz wejściowy 2 [Od lewej, rząd 2] Obraz po sumowaniu obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```
def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationRGB(self.name1, self.name2)
    resolutionUni.resolutionUnificationRGB()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2

def addPictureRGB(self):
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.pic1
    ↪ , self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, pictureName1 = self.pic1.getPictureParameters()
    matrix1 = self.pic1.getRGBMatrix()
    length2, width2, pictureName2 = self.pic2.getPictureParameters()
    matrix2 = self.pic2.getRGBMatrix()
    pictureName1 = tempName

    sumMax = 0
    x = 0
    fmax = 0
    fmin = 255

    result = np.zeros((length1, width1, 3), np.uint8)

    for l in range(length1):
        for w in range(width1):
            R = int(matrix1[l, w][0]) + int(matrix2[l, w][0])
            G = int(matrix1[l, w][1]) + int(matrix2[l, w][1])
            B = int(matrix1[l, w][2]) + int(matrix2[l, w][2])
            if sumMax < max([R, G, B]):
                sumMax = max([R, G, B])

    if sumMax > 255:
        x = (sumMax - 255) / 255

    for l in range(length1):
        for w in range(width1):
            R = (int(matrix1[l, w][0]) - (int(matrix1[l, w][0]) * x))
            ↪ + (int(matrix2[l, w][0]) - (int(matrix2[l, w][0]) *
            ↪ x))
            G = (int(matrix1[l, w][1]) - (int(matrix1[l, w][1]) * x))
            ↪ + (int(matrix2[l, w][1]) - (int(matrix2[l, w][1]) *
            ↪ x))
```

```

B = (int(matrix1[1, w][2]) - (int(matrix1[1, w][2]) * x))
    ↪ + (int(matrix2[1, w][2]) - (int(matrix2[1, w][2]) *
    ↪ x))
result[w, 1][0] = np.ceil(R)
result[w, 1][1] = np.ceil(G)
result[w, 1][2] = np.ceil(B)
# Search for maximum and minimum
if fmin > min([R, G, B]):
    fmin = min([R, G, B])
if fmax < max([R, G, B]):
    fmax = max([R, G, B])

# save picture with added picture to png file (without
    ↪ normalization)
path = self.ex + str(pictureName1) + '_added_' + str(pictureName2
    ↪ ) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

normalized = np.zeros((length1, width1, 3), np.uint8)
for l in range(length1):
    for w in range(width1):
        normalized[l, w][0] = 255 * ((result[l, w][0] - fmin) / (
            ↪ fmax - fmin))
        normalized[l, w][0] = 255 * ((result[l, w][1] - fmin) / (
            ↪ fmax - fmin))
        normalized[l, w][0] = 255 * ((result[l, w][2] - fmin) / (
            ↪ fmax - fmin))

# save picture with added picture to png file (with
    ↪ normalization)
path = self.ex + str(pictureName1) + '_added_' + str(pictureName2
    ↪ ) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

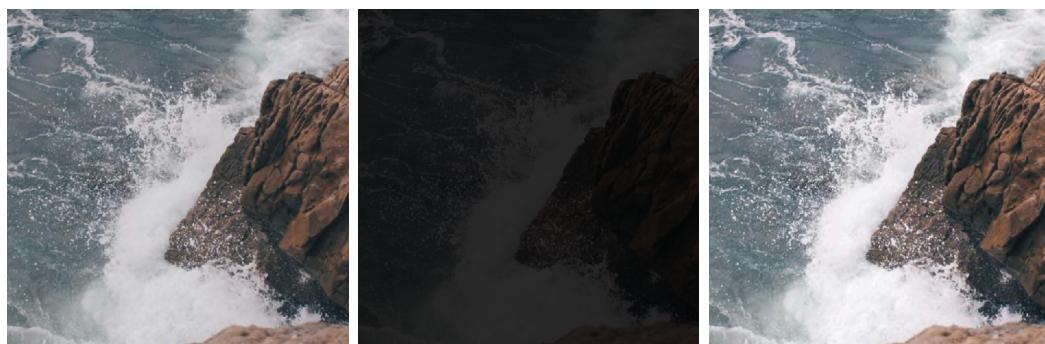
4.3 Mnożenie obrazów barwowych przez określoną stałą

Opis algorytmu

Algorytm mnożenia obrazu barwowego przez określoną stałą polega na przemnożeniu każdego elementu obrazu (piksla) przez określoną stałą (skalar). Dla wszystkich piksli w obrazie wykonaj na każdym z trzech kanałów RGB:

1. Jeżeli wartość składowa piksla jest równa 255 to składowa barwowa wynikowa otrzymuje wartość stałej.
2. Jeżeli wartość składowa piksla jest równa 0 to składowa barwowa wynikowa otrzymuje wartość 0.
3. Jeżeli wartość piksla jest inną niż 255 lub 0 to składowa wynikowa otrzymuje wartość poprzez pomnożenie wartości składowej piksla przez skalar, podzielenie przez 255 i zaokrąglenie do liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 4.5: [Od lewej] Barwowy obraz wejściowy, obraz po przmnożeniu ze stałą 50, obraz po normalizacji



Rysunek 4.6: [Od lewej] Barwowy obraz wejściowy, obraz po przmnożeniu ze stałą 200, obraz po normalizacji

Kod źródłowy algorytmu

```
def assignRGBvalue(self, value, factor):
    if value == 255:
        value = factor
    elif value == 0:
        value = 0
    else:
        value = (int(value) * int(factor)) / 255
    return value

def multiplyConstRGB(self, constant):
    length, width, matrix, pictureName = self.getPictureParameters(
        ↪ self.pic1)
    result = np.ones((length, width, 3), np.uint8)

    fmin = 255
    fmax = 0

    for l in range(length):
        for w in range(width):
            R = self.assignRGBvalue(int(matrix[l, w][0]), constant)
            G = self.assignRGBvalue(int(matrix[l, w][1]), constant)
            B = self.assignRGBvalue(int(matrix[l, w][2]), constant)

            result[w, l][0] = np.ceil(R)
            result[w, l][1] = np.ceil(G)
            result[w, l][2] = np.ceil(B)

            # Search for maximum and minimum
            if fmin > min([R, G, B]):
                fmin = min([R, G, B])
            if fmax < max([R, G, B]):
                fmax = max([R, G, B])

    # save picture with added constant to png file (without
    ↪ normalization)
    path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)

    for l in range(length):
        for w in range(width):
            result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
                ↪ - fmin))
            result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
                ↪ - fmin))
            result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
                ↪ - fmin))
```

```
# save picture with added constant to png file (with
    ↪ normalization)
path = self.ex + str(pictureName) + '_constant_' + str(constant)
    ↪ + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)
```

4.4 Mnożenie obrazu przez inny obraz

Opis algorytmu

Algorytm mnożenia obrazu barwowego przez drugi barwowy obraz o tych samych wymiarach $M \times N$ i strukturze ich macierzy polega na przemnożeniu wartości składowych barwowych piksla z pierwszego obrazu przez odpowiadającą wartość składową barwoną piksla z drugiego obrazu. Po operacji mnożenia następuje normalizacja obrazu. Dla każdej wartości składowej barwowej piksla pierwszego obrazu wykonaj następujące czynności:

1. Jeżeli wartość składowa barwowa piksla $P1[l, w]$ jest równa 255 to składowa wynikowa otrzymuje wartość odpowiadającego piksla drugiego obrazu $P2[l, w]$.
2. Jeżeli wartość składowa barwowa piksla $P1[l, w]$ jest równa 0 to składowa wynikowa otrzymuje wartość 0.
3. Jeżeli wartość składowa barwowa piksla $P1[l, w]$ jest inna niż 255 lub 0 to składowa wynikowa otrzymuje wartość poprzez pomnożenie wartości barwowej piksla $P1[l, w]$ przez odpowiadającą barwoną piksla $P2[l, w]$, podzielenie przez 255 i zaokrąglenie do liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 4.7: [Od lewej, rzad 1] Barwowy obraz wejściowy 1, barwowy obraz wejściowy 2 [Od lewej, rzad 2] Obraz po mnożeniu obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```

def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationRGB(self.name1, self.name2)
    resolutionUni.resolutionUnificationRGB()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2

def assignRGBvalue(self, value, factor):
    if value == 255:
        value = factor
    elif value == 0:
        value = 0
    else:
        value = (int(value) * int(factor)) / 255
    return value

def multiplyPicturesRGB(self):
    compare = Comparer()

```

```

biggerPicture, smallerPicture = compare.comparePictures(self.pic1
    ↪ , self.pic2)
if biggerPicture != 0 and smallerPicture != 0:
    self.pic1, self.pic2 = self.getUnifiedPictures()
# get the values
tempName = smallerPicture.getPictureName()
length1, width1, matrix1, pictureName1 = self.
    ↪ getPictureParameters(self.pic1)
length2, width2, matrix2, pictureName2 = self.
    ↪ getPictureParameters(self.pic2)
pictureName1 = tempName

result = np.ones((length1, width1, 3), np.uint8)

fmin = 255
fmax = 0

for l in range(length1):
    for w in range(width1):
        R = self.assignRGBvalue(int(matrix1[l, w][0]), int(matrix2
            ↪ [l, w][0]))
        G = self.assignRGBvalue(int(matrix1[l, w][1]), int(matrix2
            ↪ [l, w][1]))
        B = self.assignRGBvalue(int(matrix1[l, w][2]), int(matrix2
            ↪ [l, w][2]))

        result[w, l][0] = np.ceil(R)
        result[w, l][1] = np.ceil(G)
        result[w, l][2] = np.ceil(B)

        # Search for maximum and minimum
        if fmin > min([R, G, B]):
            fmin = min([R, G, B])
        if fmax < max([R, G, B]):
            fmax = max([R, G, B])

# save picture multiplied by picture to png file (without
    ↪ normalization)
path = self.ex + str(pictureName1) + '_multiplied_' + str(
    ↪ pictureName2) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length1):
    for w in range(width1):
        result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
            ↪ - fmin))
        result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
            ↪ - fmin))
        result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
            ↪ - fmin))

```

```
# save picture multiplied by picture to png file (with
    ↪ normalization)
path = self.ex + str(pictureName1) + '_multiplied_' + str(
    ↪ pictureName2) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)
```

4.5 Mieszanie obrazów z określonym współczynnikiem

Opis algorytmu

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami α i $1 - \alpha$ według wzoru:

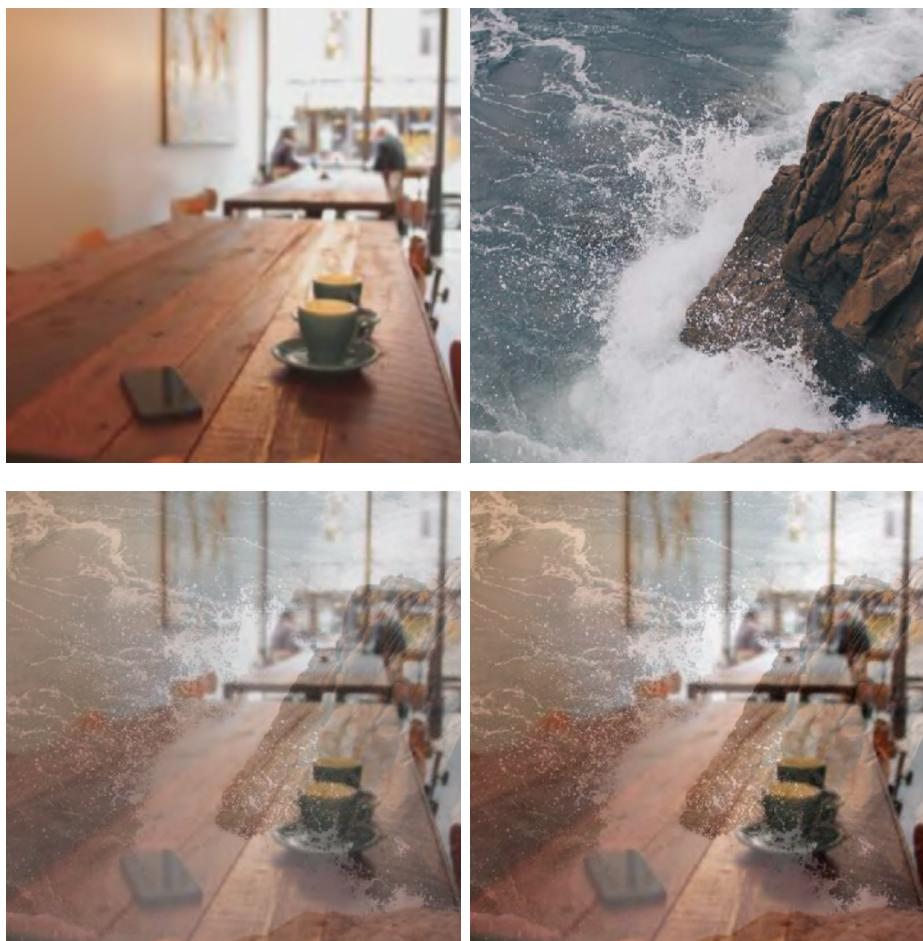
$$f_m = f\alpha + f^I(1 - \alpha)$$

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

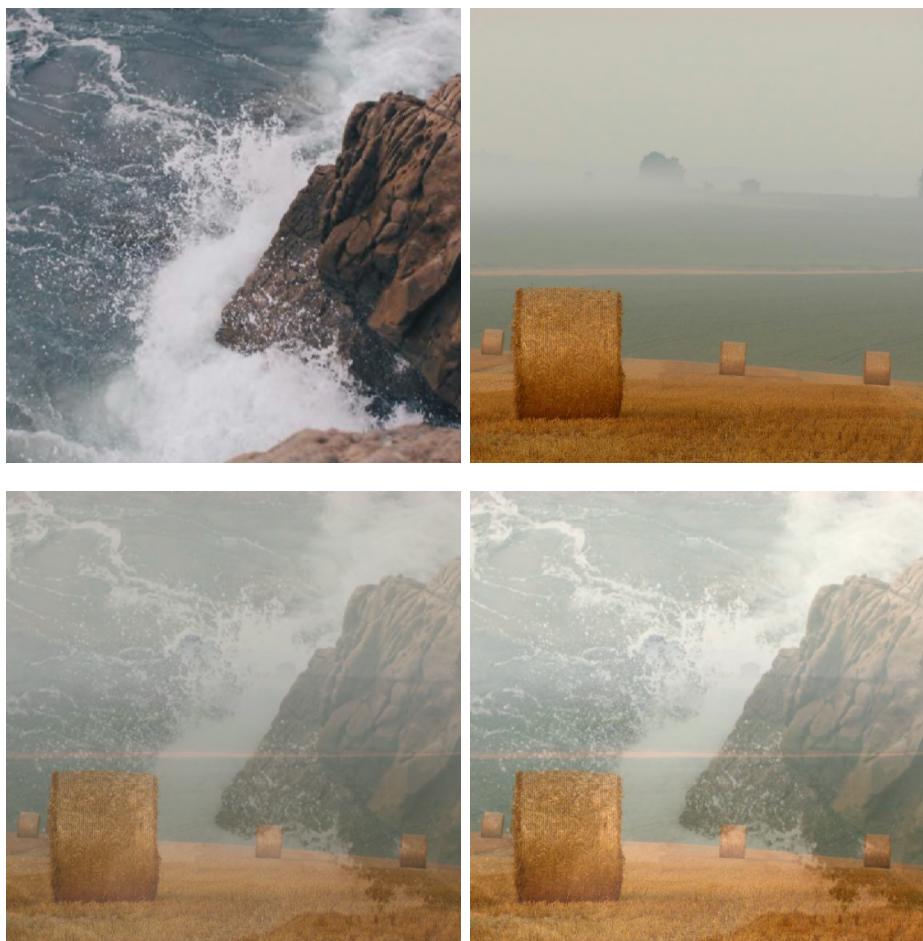
1. Weź dwa obrazy barwowe o takim samym rozmiarze (po ujednoliceniu rozdzielczościowym) P_1 i P_2 .
2. Określ współczynnik mieszania obrazów α wyrażony jako liczba rzeczywista z przedziału $[0, 1]$, gdzie 0 reprezentuje przezroczystość, a 1 reprezentuje nieprzejroczystość.
3. Dla wszystkich pikseli w obrazach wejściowych wykonaj:

$$Q[l, w] = \alpha * P_1[l, w] + (1 - \alpha) * P_2[l, w]$$

Efekty wykorzystania algorytmu



Rysunek 4.8: [Od lewej, rzad 1] Barwowy obraz wejściowy 1, barwowy obraz wejściowy 2 [Od lewej, rzad 2] Obraz po mieszananiu ze współczynnikiem $\alpha = 0.6$, obraz po normalizacji



Rysunek 4.9: [Od lewej, rzad 1] Barwowy obraz wejściowy 1, barwowy obraz wejściowy 2 [Od lewej, rzad 2] Obraz po mieszananiu ze współczynnikiem $\alpha = 0.3$, obraz po normalizacji

Kod źródłowy algorytmu

```
def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationRGB(self.name1, self.name2)
    resolutionUni.resolutionUnificationRGB()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2

def blendPictures(self, alfa):
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.pic1
    ↪ , self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, matrix1, pictureName1 = self.
        ↪ getPictureParameters(self.pic1)
    length2, width2, matrix2, pictureName2 = self.
        ↪ getPictureParameters(self.pic2)
    pictureName1 = tempName

    result = np.ones((length1, width1, 3), np.uint8)

    fmin = 255
    fmax = 0

    for l in range(length1):
        for w in range(width1):
            R = (float(matrix1[l, w][0]) * alfa) + (float(matrix2[l, w
                ↪ ][0]) * (1 - alfa))
            G = (float(matrix1[l, w][1]) * alfa) + (float(matrix2[l, w
                ↪ ][1]) * (1 - alfa))
            B = (float(matrix1[l, w][2]) * alfa) + (float(matrix2[l, w
                ↪ ][2]) * (1 - alfa))

            result[w, l][0] = np.ceil(R)
            result[w, l][1] = np.ceil(G)
            result[w, l][2] = np.ceil(B)

    # Search for maximum and minimum
    if fmin > min([R, G, B]):
        fmin = min([R, G, B])
    if fmax < max([R, G, B]):
        fmax = max([R, G, B])

    # save picture multiplied by picture to png file (without
    ↪ normalization)
```

```

path = self.ex + str(pictureName1) + '_blended_' + str(alfa) + '_'
      + str(pictureName2) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length1):
    for w in range(width1):
        result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
          ↪ - fmin))
        result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
          ↪ - fmin))
        result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
          ↪ - fmin))

# save picture multiplied by picture to png file (with
  ↪ normalization)
path = self.ex + str(pictureName1) + '_blended_' + str(alfa) + '_'
      + str(pictureName2) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

4.6 Potęgowanie obrazu z zadaną potęgą

Opis algorytmu

Algorytm potęgowania obrazu barwowego do określonej stałej jest szczególnym przypadkiem mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano z normalizowanego wzoru:

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

Efekty wykorzystania algorytmu



Rysunek 4.10: [Od lewej] Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 2$, obraz po normalizacji



Rysunek 4.11: [Od lewej] Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 5$, obraz po normalizacji

Kod źródłowy algorytmu

```
def raiseToPower(self, power):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)

    maxPicture = 0
```

```

fmin = 255
fmax = 0

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])
        if maxPicture < max([R, G, B]):
            maxPicture = max([R, G, B])

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])

        if R == 0:
            R = 0
        else:
            R = np.power(int(R) / maxPicture, power) * 255

        if G == 0:
            G = 0
        else:
            G = np.power(int(G) / maxPicture, power) * 255

        if B == 0:
            B = 0
        else:
            B = np.power(int(B) / maxPicture, power) * 255

        result[w, l][0] = np.ceil(R)
        result[w, l][1] = np.ceil(G)
        result[w, l][2] = np.ceil(B)

# Search for maximum and minimum
if fmin > min([R, G, B]):
    fmin = min([R, G, B])
if fmax < max([R, G, B]):
    fmax = max([R, G, B])

# save picture raised to constant power to png file (without
# normalization)
path = self.ex + str(pictureName) + '_power_' + str(power) + '.'
# png
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length):
    for w in range(width):

```

```

    result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
        ↪ - fmin))
    result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
        ↪ - fmin))
    result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
        ↪ - fmin))

# save picture raised to constant power to png file (with
↪ normalization)
path = self.ex + str(pictureName) + '_power_' + str(power) +
    ↪ _normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

4.7 Dzielenie obrazów barwowych przez zadaną stałą

Opis algorytmu

Algorytm dzielenia obrazu barwowego przez określoną stałą służy korekcji cieniowania między poziomami szarości obrazu. Aby zastosować algorytm dla każdego piksla obrazu wykonaj:

1. Policz sumy warytości składowych barwowych piksla ze stałą.
2. Spośród obliczonych sum wybierz $Q_{max} = \max(R_S, G_S, B_S)$ - największą sumę.
3. Wartość wynikową policz z następującego wzoru:

$$Q_R[l, w] = (R_S * 255) / Q_{max}$$

$$Q_G[l, w] = (G_S * 255) / Q_{max}$$

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

Wynik zaokrąglaj w góre do najbliższej liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 4.12: [Od lewej] Barwowy obraz wejściowy, obraz po podzieleniu przez stałą $\alpha = 7$, obraz po normalizacji



Rysunek 4.13: [Od lewej] Barwowy obraz wejściowy,obraz po podzieleniu przez stałą $\alpha = 3$, obraz po normalizacji

Kod źródłowy algorytmu

```

def divideConstRGB(self, constant):
    length, width, matrix, pictureName = self.getPictureParameters(
        ↪ self.pic1)
    result = np.zeros((length, width, 3), np.uint8)

    fmin = 255
    fmax = 0
    sumMax = 0

    for l in range(length):
        for w in range(width):
            R = int(matrix[l, w][0]) + int(constant)
            G = int(matrix[l, w][1]) + int(constant)
            B = int(matrix[l, w][2]) + int(constant)
            if sumMax < max([R, G, B]):
                sumMax = max([R, G, B])

    for l in range(length):
        for w in range(width):
            R_pom = int(matrix[l, w][0]) + int(constant)
            G_pom = int(matrix[l, w][1]) + int(constant)
            B_pom = int(matrix[l, w][2]) + int(constant)

            R = (R_pom * 255) / sumMax
            G = (G_pom * 255) / sumMax
            B = (B_pom * 255) / sumMax

            result[w, l][0] = np.ceil(R)
            result[w, l][1] = np.ceil(G)
            result[w, l][2] = np.ceil(B)
            # Search for maximum and minimum
            if fmin > min([R, G, B]):
                fmin = min([R, G, B])
            if fmax < max([R, G, B]):
                fmax = max([R, G, B])

```

```

path = self.ex + str(pictureName) + '_dividedBy_' + str(constant)
    ↪ + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length):
    for w in range(width):
        result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
            ↪ - fmin))
        result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
            ↪ - fmin))
        result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
            ↪ - fmin))

path = self.ex + str(pictureName) + '_dividedBy_' + str(constant)
    ↪ + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

4.8 Dzielenie obrazu przez inny obraz

Opis algorytmu

1. Policz sumę wartości składowej barwowej piksla obrazu P_1 z odpowiadającym wartością składową piksla obrazu P_2 .
2. Spośród obliczonych sum wybierz $Q_{max} = \max(R_S, G_S, B_S)$ - największą sumę.
3. Wartość wynikową policz z następującego wzoru:

$$Q_R[l, w] = (R_S * 255) / Q_{max}$$

$$Q_G[l, w] = (G_S * 255) / Q_{max}$$

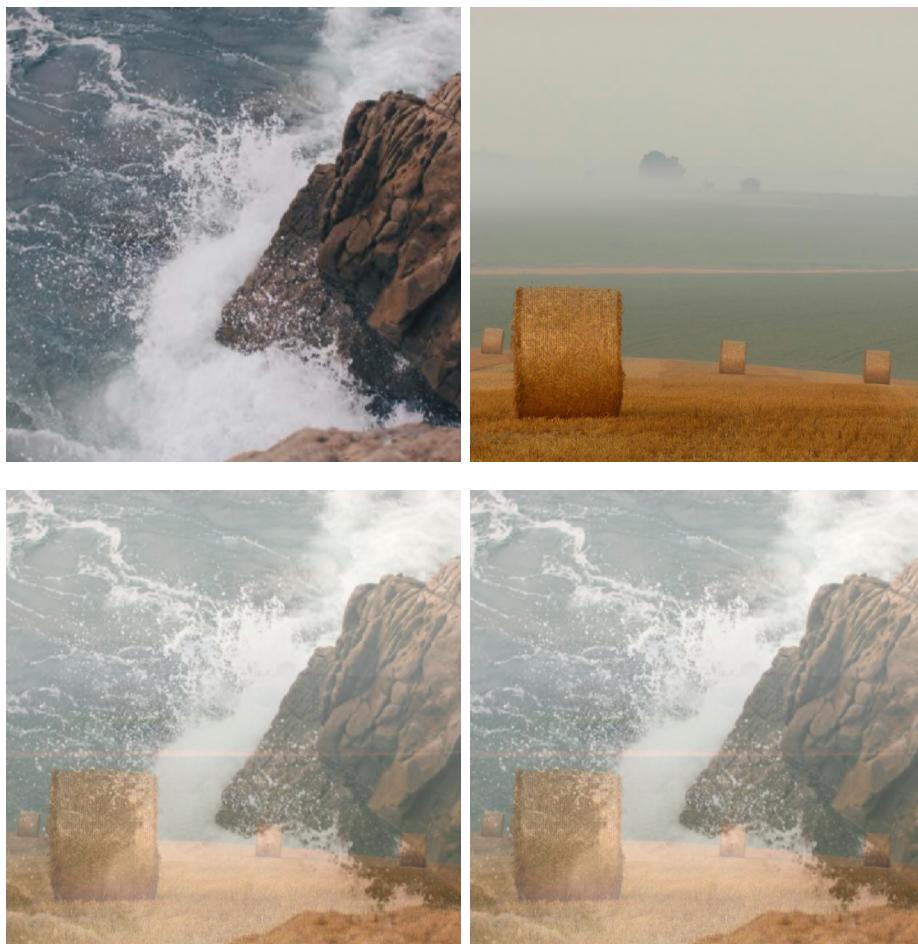
$$Q_B[l, w] = (B_S * 255) / Q_{max}$$

Wynik zaokrąglaj w górę do najbliższej liczby całkowitej.

Efekty wykorzystania algorytmu



Rysunek 4.14: [Od lewej, rząd 1] Barwowy obraz wejściowy 1, barwowy obraz wejściowy 2 [Od lewej, rząd 2] Obraz powstały w wyniku dzielenia obrazów 1 i 2, obraz po normalizacji



Rysunek 4.15: [Od lewej, rzad 1] Barwowy obraz wejściowy 1, barwowy obraz wejściowy 2 [Od lewej, rzad 2] Obraz powstały w wyniku dzielenia obrazów 1 i 2, obraz po normalizacji

Kod źródłowy algorytmu

```

def getUnifiedPictures(self):
    resolutionUni = ResolutionUnificationRGB(self.name1, self.name2)
    resolutionUni.resolutionUnificationRGB()
    pic1Path, pic2Path = resolutionUni.getOutputPaths()
    pic1 = ImageHelper(pic1Path, self.pictureType)
    pic2 = ImageHelper(pic2Path, self.pictureType)
    return pic1, pic2

def dividePicturesRGB(self):
    compare = Comparer()
    biggerPicture, smallerPicture = compare.comparePictures(self.pic1
        ↪ , self.pic2)
    if biggerPicture != 0 and smallerPicture != 0:
        self.pic1, self.pic2 = self.getUnifiedPictures()
    # get the values
    tempName = smallerPicture.getPictureName()
    length1, width1, matrix1, pictureName1 = self.
        ↪ getPictureParameters(self.pic1)

```

```

length2, width2, matrix2, pictureName2 = self.
    ↪ getPictureParameters(self.pic2)
pictureName1 = tempName

result = np.ones((length1, width1, 3), np.uint8)

fmin = 255
fmax = 0
sumMax = 0

for l in range(length1):
    for w in range(width1):
        R = int(matrix1[l, w][0]) + int(matrix2[l, w][0])
        G = int(matrix1[l, w][1]) + int(matrix2[l, w][1])
        B = int(matrix1[l, w][2]) + int(matrix2[l, w][2])
        if sumMax < max([R, G, B]):
            sumMax = max([R, G, B])

for l in range(length1):
    for w in range(width1):
        R_pom = int(matrix1[l, w][0]) + int(matrix2[l, w][0])
        G_pom = int(matrix1[l, w][1]) + int(matrix2[l, w][1])
        B_pom = int(matrix1[l, w][2]) + int(matrix2[l, w][2])

        R = (R_pom * 255) / sumMax
        G = (G_pom * 255) / sumMax
        B = (B_pom * 255) / sumMax

        result[w, l][0] = np.ceil(R)
        result[w, l][1] = np.ceil(G)
        result[w, l][2] = np.ceil(B)
        # Search for maximum and minimum
        if fmin > min([R, G, B]):
            fmin = min([R, G, B])
        if fmax < max([R, G, B]):
            fmax = max([R, G, B])

# save picture divided by picture to png file (without
    ↪ normalization)
path = self.ex + str(pictureName1) + '_dividedBy_' + str(
    ↪ pictureName2) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

normalized = np.zeros((length1, width1, 3), np.uint8)
for l in range(length1):
    for w in range(width1):
        normalized[l, w][0] = 255 * ((result[l, w][0] - fmin) / (
            ↪ fmax - fmin))
        normalized[l, w][0] = 255 * ((result[l, w][1] - fmin) / (
            ↪ fmax - fmin))

```

```
normalized[l, w][0] = 255 * ((result[l, w][2] - fmin) / (
    ↪ fmax - fmin))

# save picture divided by picture to png file (with
    ↪ normalization)
path = self.ex + str(pictureName1) + '_dividedBy_' + str(
    ↪ pictureName2) + '_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)
```

4.9 Pierwiastkowanie obrazu

Opis algorytmu

Algorytm pierwiastkowania obrazu barwowego jest szczególnym przypadkiem wykorzystania algorytmu potęgowania obrazu przez określoną stałą, która jest ułamkiem. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru:

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

Efekty wykorzystania algorytmu



Rysunek 4.16: [Od lewej] Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 1/2$, obraz po normalizacji



Rysunek 4.17: [Od lewej] Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha = 1/6$, obraz po normalizacji

Kod źródłowy algorytmu

```
def rootRGB(self, power):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)
    factorial = 1 / power
```

```

maxPicture = 0
fmin = 255
fmax = 0

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])
        if maxPicture < max([R, G, B]):
            maxPicture = max([R, G, B])

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])

        if R == 0:
            R = 0
        else:
            R = np.power(int(R) / maxPicture, factorial) * 255

        if G == 0:
            G = 0
        else:
            G = np.power(int(G) / maxPicture, factorial) * 255

        if B == 0:
            B = 0
        else:
            B = np.power(int(B) / maxPicture, factorial) * 255

        result[w, l][0] = np.ceil(R)
        result[w, l][1] = np.ceil(G)
        result[w, l][2] = np.ceil(B)

# Search for maximum and minimum
if fmin > min([R, G, B]):
    fmin = min([R, G, B])
if fmax < max([R, G, B]):
    fmax = max([R, G, B])

# save picture raised to constant factorial to png file (without
# normalization)
path = self.ex + str(pictureName) + '_root_' + str(power) + '.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length):

```

```

for w in range(width):
    result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
        ↪ - fmin))
    result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
        ↪ - fmin))
    result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
        ↪ - fmin))

# save picture raised to constant factorial to png file (with
↪ normalization)
path = self.ex + str(pictureName) + '_root_' + str(power) + '
    ↪ _normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

4.10 Logarytmowanie obrazu

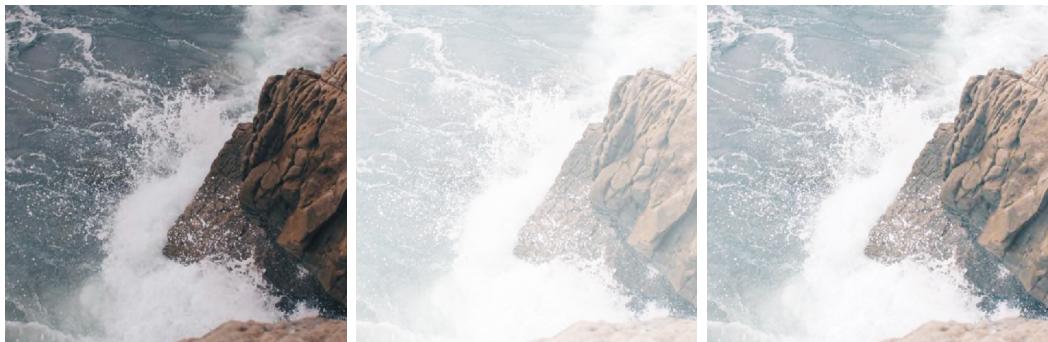
Opis algorytmu

Algorytm logarytmowania obrazu barwowego powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Aby uniknąć wykroczenia poza zakres, skorzystano ze znalezionego wzoru:

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

Przesunięcie funkcji obrazu o 1 w górę wynika z nieokreśloności logarytmu w zerze.

Efekty wykorzystania algorytmu



Rysunek 4.18: [Od lewej] Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji



Rysunek 4.19: [Od lewej] Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji

Kod źródłowy algorytmu

```
def logharitmRGB(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)
```

```

maxPicture = 0
fmin = 255
fmax = 0

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])
        if maxPicture < max([R, G, B]):
            maxPicture = max([R, G, B])

for l in range(length):
    for w in range(width):
        R = int(matrix[l, w][0])
        G = int(matrix[l, w][1])
        B = int(matrix[l, w][2])

        if R == 0:
            R = 0
        else:
            R = (np.log(1 + R) / np.log(1 + maxPicture)) * 255

        if G == 0:
            G = 0
        else:
            G = (np.log(1 + G) / np.log(1 + maxPicture)) * 255

        if B == 0:
            B = 0
        else:
            B = (np.log(1 + B) / np.log(1 + maxPicture)) * 255

        result[w, l][0] = np.ceil(R)
        result[w, l][1] = np.ceil(G)
        result[w, l][2] = np.ceil(B)
        # Search for maximum and minimum
        if fmin > min([R, G, B]):
            fmin = min([R, G, B])
        if fmax < max([R, G, B]):
            fmax = max([R, G, B])

# save picture raised to constant power to png file (without
# normalization)
path = self.ex + str(pictureName) + '_log.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

for l in range(length):
    for w in range(width):

```

```

    result[l, w][0] = 255 * ((result[l, w][0] - fmin) / (fmax
    ↪ - fmin))
    result[l, w][1] = 255 * ((result[l, w][1] - fmin) / (fmax
    ↪ - fmin))
    result[l, w][2] = 255 * ((result[l, w][2] - fmin) / (fmax
    ↪ - fmin))

# save picture raised to constant power to png file (with
    ↪ normalization)
path = self.ex + str(pictureName) + '_log_normalized.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

Rozdział 5

Operacje geometryczne na obrazie

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

5.1 Przemieszczanie obrazu o zadany wektor

Opis algorytmu

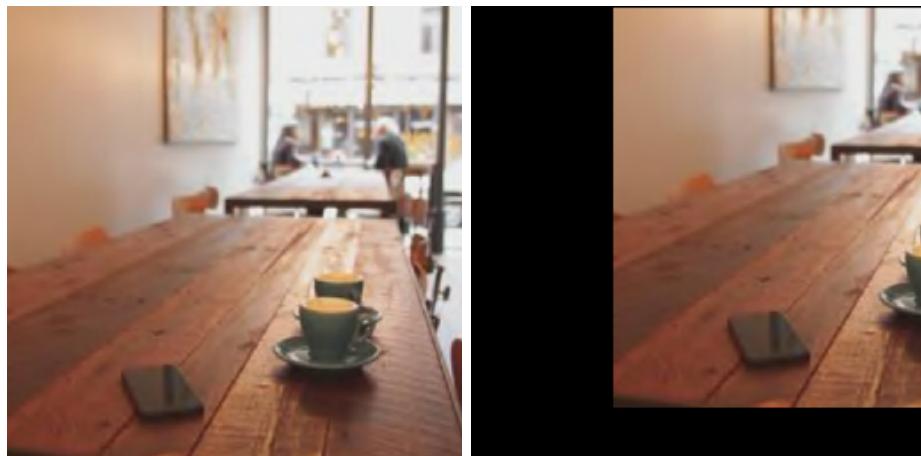
Przemieszczanie obrazu o dany wektor polega na przesunięciu każdego piksla obrazu o określoną wartość zgodnie z zależnościami:

$$x^I = x_o + \Delta x$$

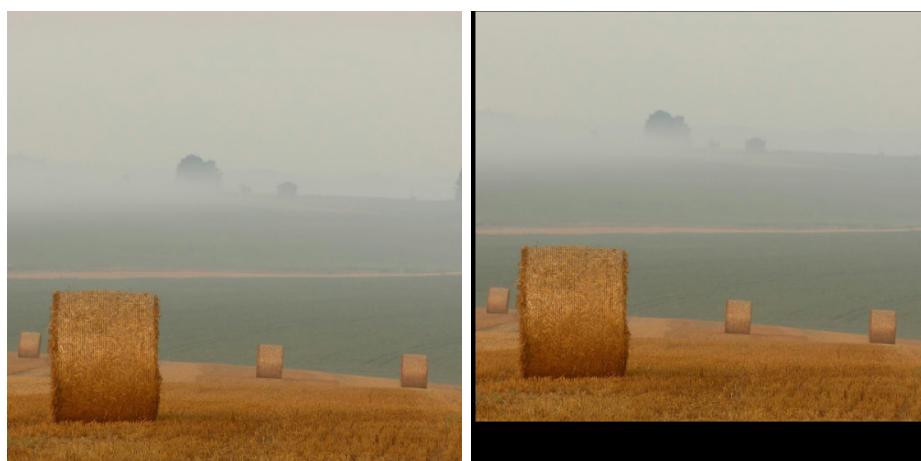
$$y^I = y_o + \Delta y$$

gdzie (x_o, y_o) są współrzędnymi początkowymi piksla, $(\Delta x, \Delta y)$ są wartościami przesunięcia, a (x^I, y^I) są współrzędnymi piksla po przesunięciu.

Efekty wykorzystania algorytmu



Rysunek 5.1: [Od lewej] Barwowy obraz wejściowy, obraz po przesunięciu o wektor $[30, 80]$



Rysunek 5.2: [Od lewej] Barwowy obraz wejściowy, obraz po przesunięciu o wektor $[100, 10]$

Kod źródłowy algorytmu

```
def relocateByVectorRGB(self, x, y):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    deltaX = 0 - x
    result = np.zeros((length, width, 3), np.uint8)

    for l in range(length):
        for w in range(width):
            if 0 < l + y < length and 0 < w + deltaX < width:
                result[w + deltaX, l + y] = matrix[l, w]

    path = self.ex + str(pictureName) + '_moved_[ ' + str(x) + ', '
    ↪ + str(y) + '].png'
    self.saver.savePictureFromArray(result, self.pictureType,
    ↪ path)
```

5.2 Skalowanie jednorodne obrazu

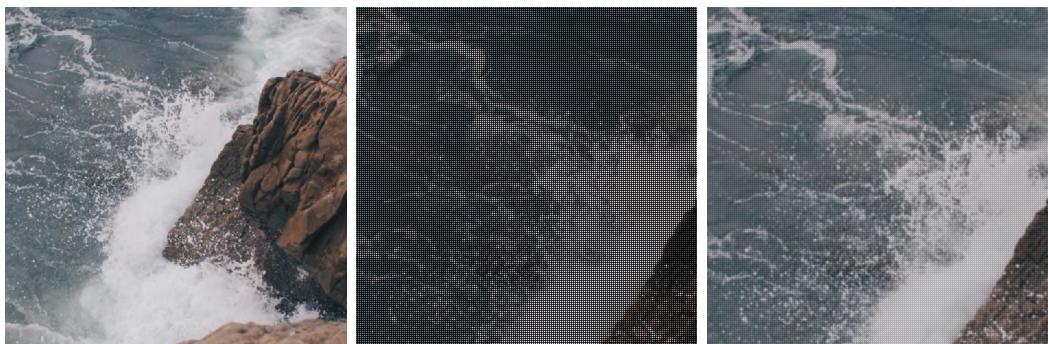
Opis algorytmu

Skalowanie jednorodne obrazu polega na pomnożeniu każdego piksla obrazu przez współczynnik skalowania S według zależności:

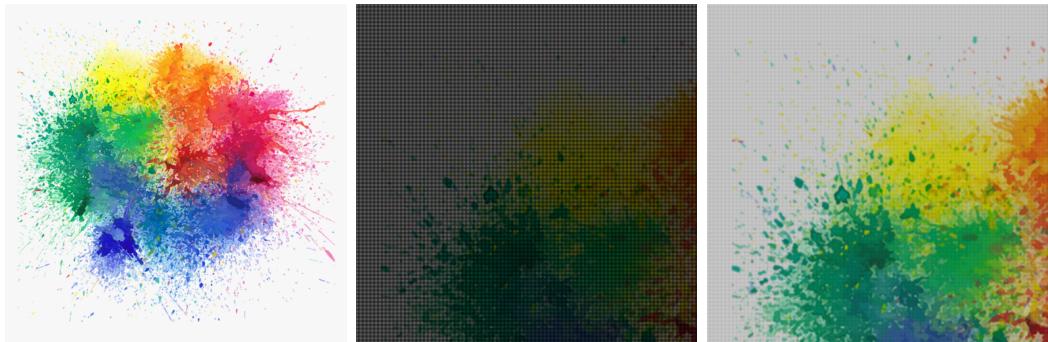
$$x^I = x_o * S$$

$$y^I = y_o * S$$

Efekty wykorzystania algorytmu



Rysunek 5.3: [Od lewej] Barwowy obraz wejściowy, obraz po skalowaniu jednorodnym ze współczynnikiem $S = 1.5$, obraz po interpolacji



Rysunek 5.4: [Od lewej] Barwowy obraz wejściowy, obraz po skalowaniu jednorodnym ze współczynnikiem $S = 2$, obraz po interpolacji

Kod źródłowy algorytmu

```
def homogeneousScalingRGB(self, scale):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)
    for l in range(length):
        for w in range(width):
```

```

        if scale * l < length and scale * w < width:
            result[int(scale * w), int(scale * l)] = matrix[l, w]
path = self.ex + str(pictureName) + '_homogeneous_' + str(scale)
    ↪ + '_withoutInterpolation.png'
self.saver.savePictureFromArray(result, self.pictureType, path)
for l in range(length):
    for w in range(width):
        r, g, b = 0, 0, 0
        n = 1
        if result[l, w][0] < 1 and result[l, w][1] < 1 and result[
            ↪ l, w][2] < 1:
            for lOff in range(-1, 2):
                for wOff in range(-1, 2):
                    lSave = l if ((l + lOff) > (length - 2)) | ((l
                        ↪ + lOff) < 0) else (l + lOff)
                    wSave = w if ((w + wOff) > (width - 2)) | ((w +
                        ↪ wOff) < 0) else (w + wOff)
                    if result[lSave, wSave][0] > 0 or result[lSave,
                        ↪ wSave][1] > 0 or result[lSave, wSave][2]
                        ↪ > 0:
                        r += result[lSave, wSave][0]
                        g += result[lSave, wSave][1]
                        b += result[lSave, wSave][2]
                        n += 1
                    result[l, w] = (r / n, g / n, b / n)
path = self.ex + str(pictureName) + '_homogeneous_' + str(scale)
    ↪ + '_withInterpolation.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

5.3 Skalowanie niejednorodne obrazu

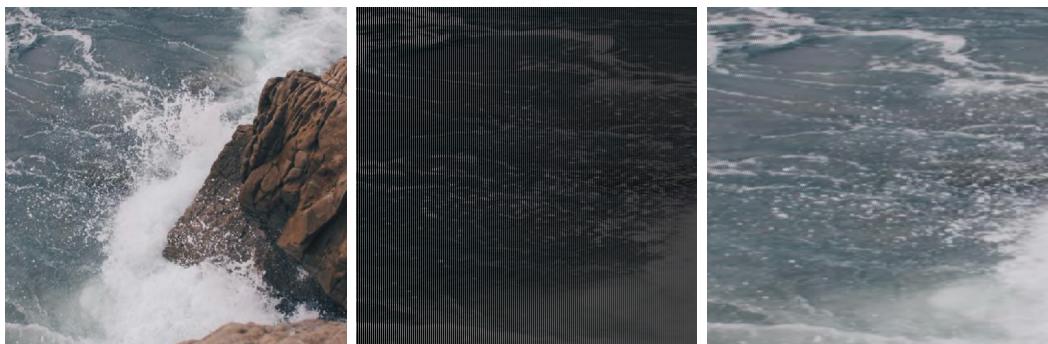
Opis algorytmu

Skalowanie niejednorodne obrazu polega na pomnożeniu każdego piksla obrazu przez współczynnik skalowania S_x, S_y według zależności:

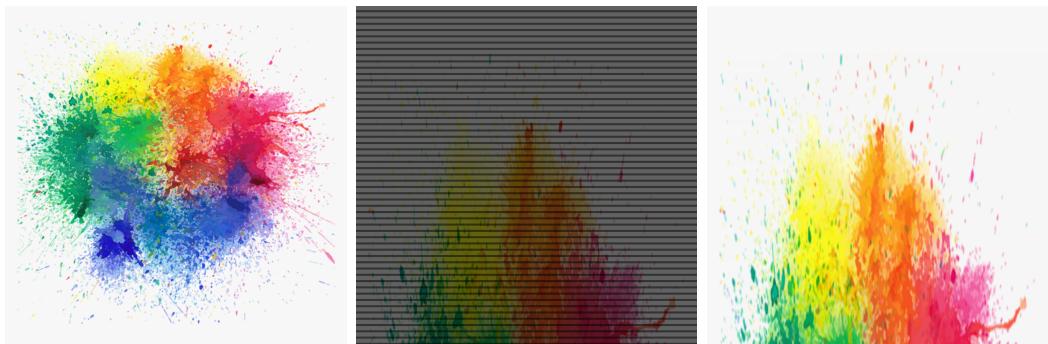
$$x^I = x_o * S_x$$

$$y^I = y_o * S_y$$

Efekty wykorzystania algorytmu



Rysunek 5.5: [Od lewej] Barwowy obraz wejściowy, obraz po skalowaniu niejednorodnym ze współczynnikiem $S_x = 3$ i $S_y = 1$, obraz po interpolacji



Rysunek 5.6: [Od lewej] Barwowy obraz wejściowy, obraz po skalowaniu niejednorodnym ze współczynnikiem $S_x = 1$ i $S_y = 3$, obraz po interpolacji

Kod źródłowy algorytmu

```
def heterogeneousScalingRGB(self, scaleY, scaleX):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.zeros((length, width, 3), np.uint8)
    for l in range(length):
        for w in range(width):
```

```

        if scaleY * l < length and scaleX * w < width:
            result[int(scaleX * w), int(scaleY * l)] = matrix[l, w
                ↪ ]
path = self.ex + str(pictureName) + '_heterogeneous_[ ' + str(
    ↪ scaleY) + ', ' + str(
    scaleX) + ']_withoutInterpolation.png'
self.saver.savePictureFromArray(result, self.pictureType, path)
for l in range(length):
    for w in range(width):
        r, g, b = 0, 0, 0
        n = 0
        if result[l, w][0] < 1 and result[l, w][1] < 1 and result[
            ↪ l, w][2] < 1:
            for lOff in range(-1, 2):
                for wOff in range(-1, 2):
                    lSave = l if ((l + lOff) > (length - 2)) | ((l
                        ↪ + lOff) < 0) else (l + lOff)
                    wSave = w if ((w + wOff) > (width - 2)) | ((w +
                        ↪ wOff) < 0) else (w + wOff)
                    if result[lSave, wSave][0] > 0 or result[lSave,
                        ↪ wSave][1] > 0 or result[lSave, wSave][
                            2] > 0:
                        r += result[lSave, wSave][0]
                        g += result[lSave, wSave][1]
                        b += result[lSave, wSave][2]
                        n += 1
                    result[l, w] = (r / n, g / n, b / n)
path = self.ex + str(pictureName) + '_heterogeneous_[ ' + str(
    ↪ scaleY) + ', ' + str(
    scaleX) + ']_withInterpolation.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

5.4 Obracanie obrazu o dowolny kąt

Opis algorytmu

Obracanie obrazu o dowolny kąt dookoła początku układu współrzędnych wykonuje się zgodnie ze wzorem:

$$x^I = x_o * \cos\alpha - y_o * \sin\alpha$$

$$y^I = x_o * \sin\alpha + y_o * \cos\alpha$$

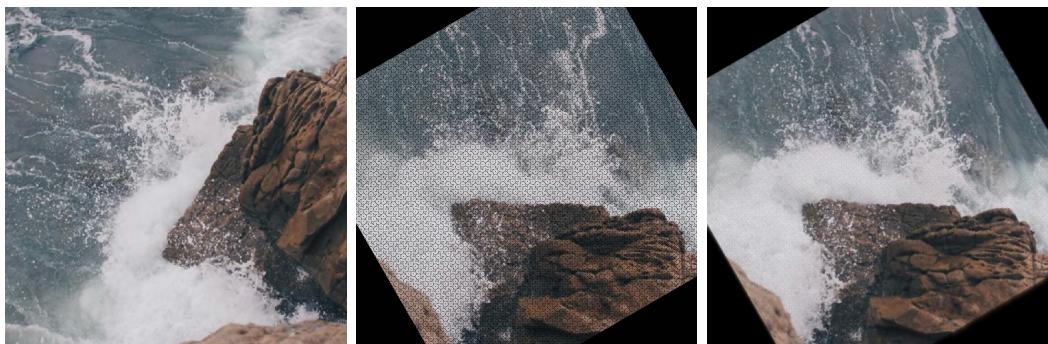
gdzie (x_o, y_o) - współrzędne początkowe piksla, α to kąt obrotu, (x^I, y^I) to współrzędne piksla po obrocie.

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

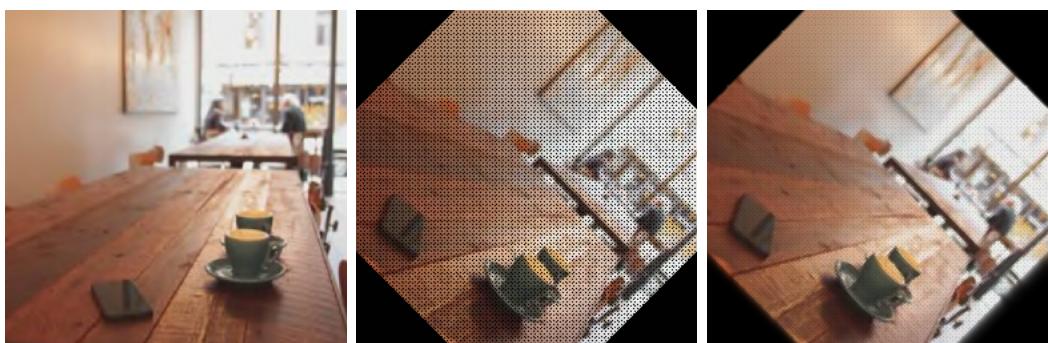
$$\text{new}_x = (x - \text{width}/2) * \cos(\alpha_r) - (y - \text{height}/2) * \sin(\alpha_r) + (\text{width}/2)$$

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

Efekty wykorzystania algorytmu



Rysunek 5.7: [Od lewej] Barwowy obraz wejściowy, obraz po obrocie o kąt $\alpha = 60^\circ$, obraz po interpolacji



Rysunek 5.8: [Od lewej] Barwowy obraz wejściowy, obraz po obrocie o kąt $\alpha = 45^\circ$, obraz po interpolacji

Kod źródłowy algorytmu

```
def angleRGB(self, angle):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.ones((length, width, 3), np.uint8)
    angleRadians = np.radians(angle)
    for l in range(length):
        for w in range(width):
            newW = (w - width / 2) * np.cos(angleRadians) - (l -
                ↪ length / 2) * np.sin(angleRadians) + (width / 2)
            newL = (w - width / 2) * np.sin(angleRadians) + (l -
                ↪ length / 2) * np.cos(angleRadians) + (length / 2)
            if 0 <= newL < length and 0 <= newW < width:
                result[int(newL), int(newW)] = matrix[l, w]
    path = self.ex + str(pictureName) + '_angle_' + str(angle) + '
        ↪ _withoutInterpolation.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
    for l in range(length):
        for w in range(width):
            r, g, b = 0, 0, 0
            n = 1
            if result[l, w][0] == 1 and result[l, w][1] == 1 and
                ↪ result[l, w][2] == 1:
                for lOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        lSave = l if ((l + lOff) > (length - 2)) | ((l
                            ↪ + lOff) < 0) else (l + lOff)
                        wSave = w if ((w + wOff) > (width - 2)) | ((w +
                            ↪ wOff) < 0) else (w + wOff)
                        if result[lSave, wSave][0] > 1 or result[lSave,
                            ↪ wSave][1] > 1 or result[lSave, wSave][
                                2] > 1:
                            r += result[lSave, wSave][0]
                            g += result[lSave, wSave][1]
                            b += result[lSave, wSave][2]
                            n += 1
            result[l, w] = (r / n, g / n, b / n)
    path = self.ex + str(pictureName) + '_angle_' + str(angle) + '
        ↪ _withInterpolation.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
```

5.5 Symetrie obrazu względem osi układu

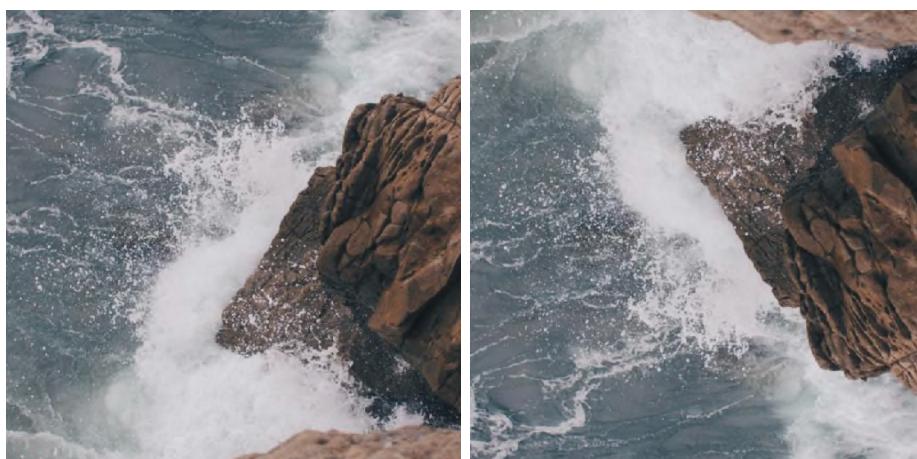
5.5.1 Symetria względem osi OX

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

Efekty wykorzystania algorytmu



Rysunek 5.9: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem osi OX



Rysunek 5.10: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem osi OX

Kod źródłowy algorytmu

```
def symmetryX(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.ones((length, width, 3), np.uint8)
    pomLength = length - 1
```

```

for l in range(length):
    for w in range(width):
        result[w, l] = matrix[pomLength - l, w]

path = self.ex + str(pictureName) + '_X.png'
self.saver.savePictureFromArray(result, self.pictureType, path)

```

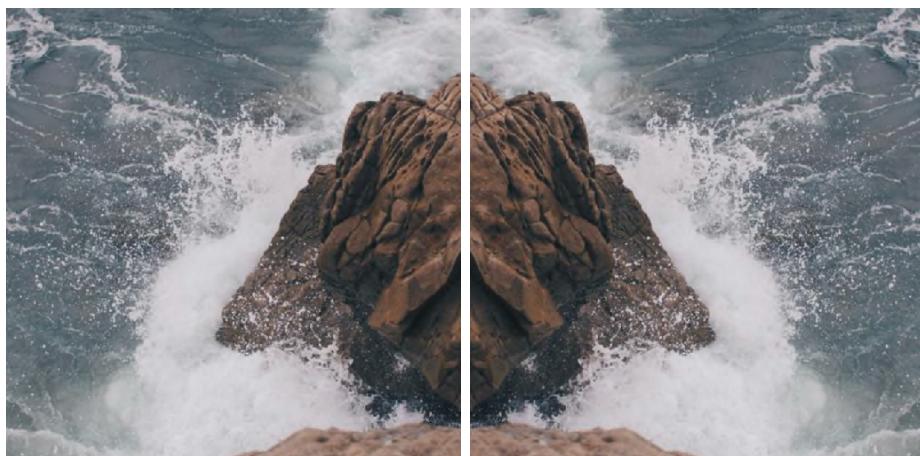
5.5.2 Symetria względem osi OY

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

Efekty wykorzystania algorytmu



Rysunek 5.11: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem osi OY



Rysunek 5.12: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem osi OY

Kod źródłowy algorytmu

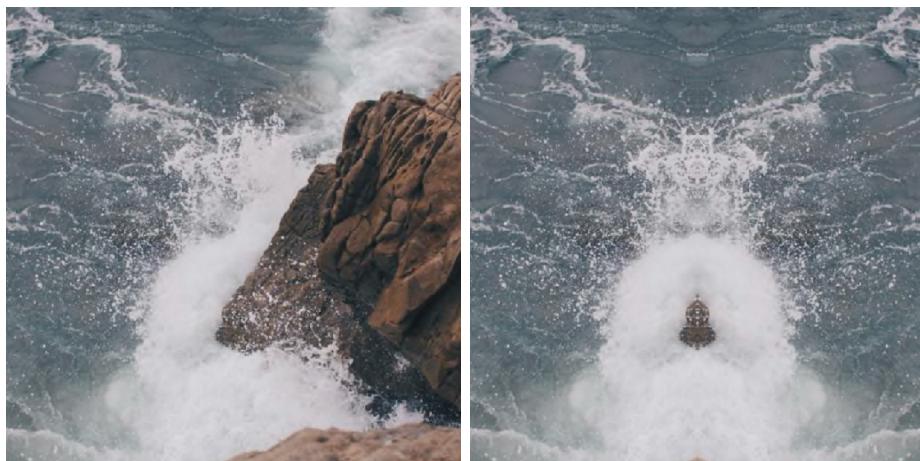
```
def symmetryY(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.ones((length, width, 3), np.uint8)
    pomLength = length - 1
    for l in range(length):
        for w in range(width):
            result[w, l] = matrix[pomLength - l, w]
    path = self.ex + str(pictureName) + '_Y.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
```

5.5.3 Symetria względem zadanej prostej

Efekty wykorzystania algorytmu



Rysunek 5.13: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem pionowej prostej poprowadzonej przez środek obrazu



Rysunek 5.14: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem pionowej prostej poprowadzonej przez środek obrazu

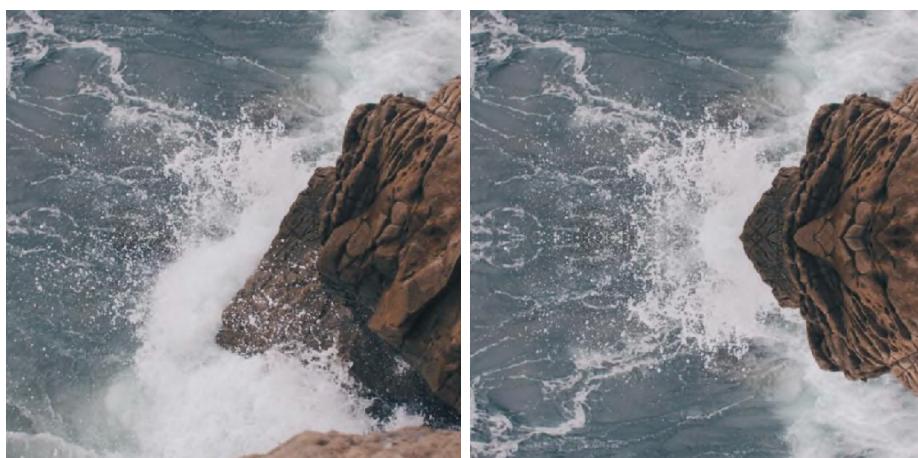
Kod źródłowy algorytmu

```
def symmetryLineVertical(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.ones((length, width, 3), np.uint8)
    pomLength = length - 1
    paramL = length / 2
    for l in range(length):
        for w in range(width):
            if l < paramL:
                result[w, l] = matrix[l, w]
            else:
                result[w, l] = matrix[pomLength - l, w]
    path = self.ex + str(pictureName) + '_vertical.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
```

Efekty wykorzystania algorytmu



Rysunek 5.15: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem poziomej prostej poprowadzonej przez środek obrazu



Rysunek 5.16: [Od lewej] Barwowy obraz wejściowy, obraz po obróceniu symetrycznym względem poziomej prostej poprowadzonej przez środek obrazu

Kod źródłowy algorytmu

```
def symmetryLineHorizontal(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getRGBMatrix()
    result = np.ones((length, width, 3), np.uint8)
    pomWidth = width - 1
    paramW = width / 2
    for l in range(length):
        for w in range(width):
            if w < paramW:
                result[w, l] = matrix[l, w]
            else:
                result[w, l] = matrix[l, pomWidth - w]
    path = self.ex + str(pictureName) + '_horizontal.png'
    self.saver.savePictureFromArray(result, self.pictureType, path)
```

5.6 Wycinanie fragmentów obrazów

Efekty wykorzystania algorytmu



Rysunek 5.17: [Od lewej] Barwowy obraz wejściowy 256x256, obraz po wycięciu fragmentu o współrzędnych $x_{min} = 70$, $x_{max} = 140$, $y_{min} = 160$, $y_{max} = 200$



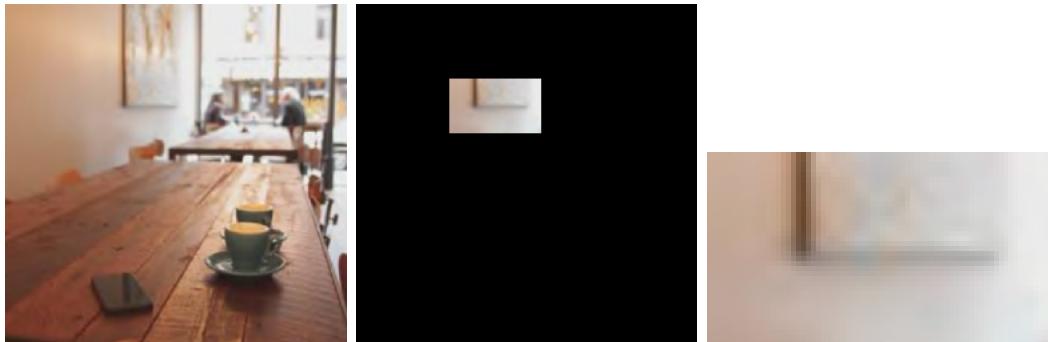
Rysunek 5.18: [Od lewej] Barwowy obraz wejściowy 1025x1025, obraz po wycięciu fragmentu o współrzędnych $x_{min} = 600$, $x_{max} = 1000$, $y_{min} = 160$, $y_{max} = 400$

Kod źródłowy algorytmu

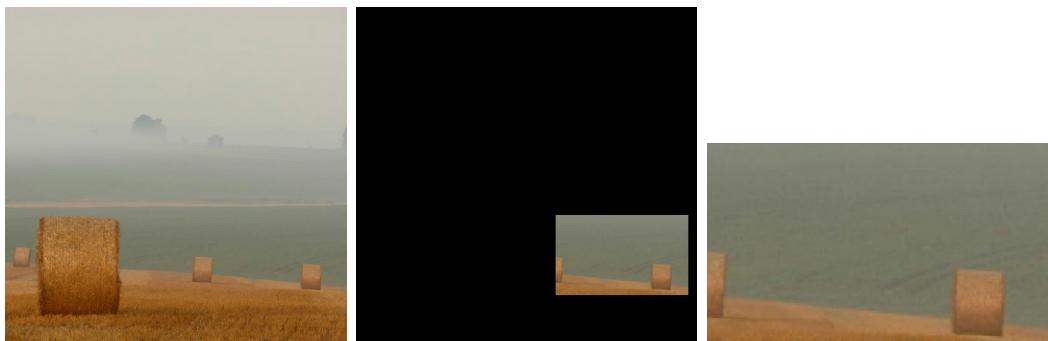
```
def cutPiece(self, yMin, yMax, xMin, xMax):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    for x in range(length - xMax, length - xMin + 1):
        for y in range(yMin, yMax+1):
            matrix[x, y] = (0, 0, 0)
    path = self.ex + str(pictureName) + '_' + str(yMin) + ',' + str(
        ↳ yMax) + ') , (' + str(xMin) + ',' + str(xMax) + ').png'
    self.saver.savePictureFromArray(matrix, self.pictureType, path)
```

5.7 Kopiowanie fragmentów obrazów

Efekty wykorzystania algorytmu



Rysunek 5.19: [Od lewej] Barwowy obraz wejściowy 256x256, wycięty fragment o współrzędnych $x_{min} = 70$, $x_{max} = 140$, $y_{min} = 160$, $y_{max} = 200$ na czarnym tle, wycięty fragment bez czarnego tła



Rysunek 5.20: [Od lewej] Barwowy obraz wejściowy 1025x1025, wycięty fragment o współrzędnych $x_{min} = 600$, $x_{max} = 1000$, $y_{min} = 160$, $y_{max} = 400$ na czarnym tle, wycięty fragment bez czarnego tła

Kod źródłowy algorytmu

```
def copyPiece(self, yMin, yMax, xMin, xMax):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    frame = np.zeros((length, width, 3), np.uint8)
    for x in range(length - xMax, length - xMin + 1):
        for y in range(yMin, yMax - 1):
            frame[x, y] = matrix[x, y]
    path = self.ex + str(pictureName) + '_withBlackFrame.png'
    self.saver.savePictureFromArray(frame, self.pictureType, path)
    copiedPiece = np.zeros((xMax - xMin - 1, yMax - yMin - 1, 3), np.
        ↪ uint8)
    xPom = 0
    for y in range(yMin, yMax - 1):
```

```

yPom = 0
print('x:\u2022' + str(x))
for x in range(length - xMax, length - xMin - 1):
    print('y:\u2022' + str(y))
    print('yMax:\u2022' + str(yMax))
    print('yPom:\u2022' + str(yPom))
    copiedPiece[yPom, xPom] = frame[x, y]
    yPom += 1
    xPom += 1
path = self.ex + str(pictureName) + '_(' + str(yMin) + ',', ' + str(
    ↪ yMax) + ') , (' + str(xMin) + ',', ' + str(xMax) + ') .png'
self.saver.savePictureFromArray(copiedPiece, self.pictureType,
    ↪ path)

```

Rozdział 6

Operacje na histogramie obrazu szarego

Histogram to najprostszy opis całościowy obrazu. Jego obliczanie polega na odczytaniu poziomu szarości każdego piksla i rejestraniu jego wystąpienia. Wyniki obliczeń w przedstawianych zadaniach są opisane za pomocą wykresu słupkowego, gdzie os OX reprezentuje poziomy szarości, a os OY reprezentuje liczbę ich wystąpień. Na podstawie histogramu obrazu możemy odczytać dane o jasności i intensywności obrazu.

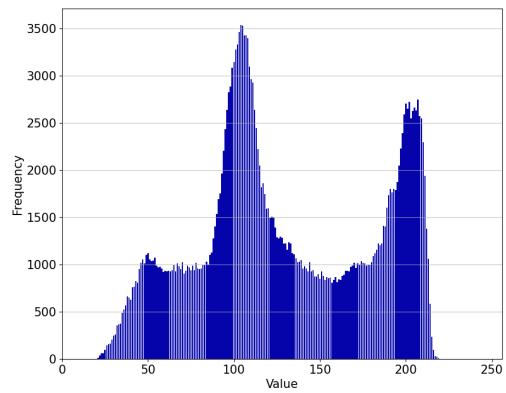
6.1 Obliczanie histogramu

Opis algorytmu

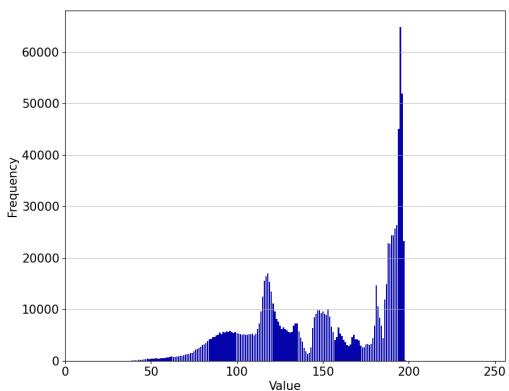
Histogram obrazu szarego jest wykresem częstości występowania wartości szarości piksli w obrazie tj. przyporządkowuje liczbę pikseli do danego poziomu szarości.

1. Zaalokuj tablicę 256 elementową, wypełnioną zerami, gdzie każdy element odpowiada ilości wystąpień danego poziomu szarości.
2. Dla każdego piksela zwięksź element tablicy o indeksie równym poziomowi szarości danego piksela.

Efekty wykorzystania algorytmu



Rysunek 6.1: [Od lewej] Szary obraz wejściowy, histogram obrazu wejściowego



Rysunek 6.2: [Od lewej] Szary obraz wejściowy, histogram obrazu wejściowego

Kod źródłowy algorytmu

```
def plotHistogram(self, bins, hist, path):
    fig = plt.figure(figsize=[10, 8])
    plt.bar(bins[:-1], hist, width=0.8, color='#0504aa')
    plt.xlim(min(bins), max(bins))
    plt.grid(axis='y', alpha=0.75)
    plt.xlabel('Value', fontsize=15)
    plt.ylabel('Frequency', fontsize=15)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.ylabel('Frequency', fontsize=15)
    plt.savefig(path)

def calculateHistogram(self, matrix, path):
    length, width, pictureName = self.pic.getPictureParameters()
    histogram = np.zeros(256, np.int64)
    for l in range(length):
        for w in range(width):
            bin = matrix[l, w]
            histogram[bin] += 1
    bins = np.arange(257)
    self.plotHistogram(bins, histogram, path)
```

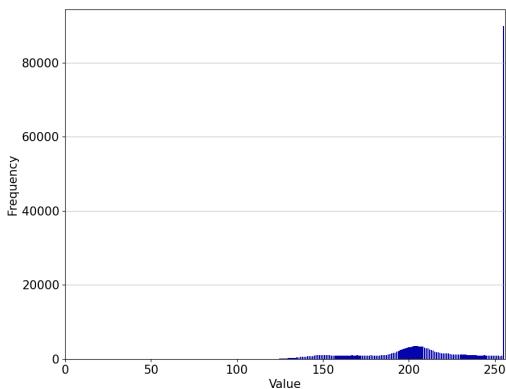
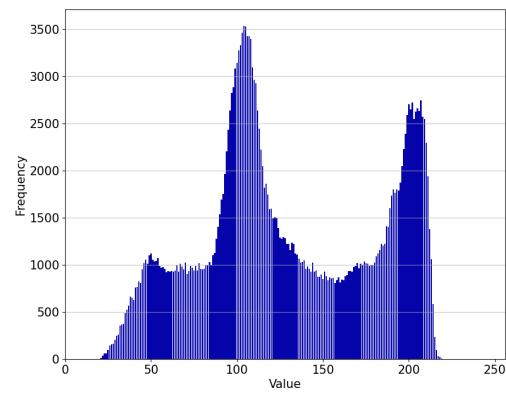
6.2 Przemieszczanie histogramu

Opis algorytmu

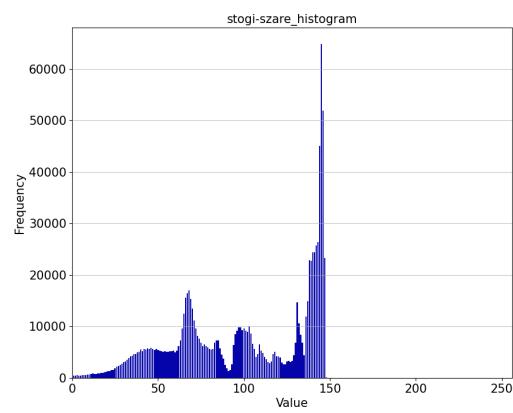
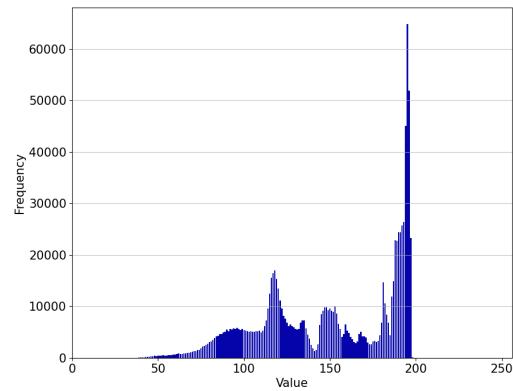
Przemieszczanie histogramu polega na dodaniu tej samej wartości od poziomu szarości każdego piksla w obrazie. W rezultacie obraz jest równomiernie rozjaśniony bądź przciemniony. Dla każdej wartości piksla wykonaj:

1. Jeśli wartość piksla po operacji dodawania jest większa niż 255, przypisz mu wartość 255.
2. Jeśli wartość piksla po operacji dodawania jest mniejsza niż 0, przypisz mu wartość 0.

Efekty wykorzystania algorytmu



Rysunek 6.3: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po przesunięciu o 100, histogram obrazu po przesunięciu



Rysunek 6.4: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po przesunięciu o -50, histogram obrazu po przesunięciu

Kod źródłowy algorytmu

```
def moveHistogram(self, const):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    ex = './Documentation/images/ExEffects/5/52/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width), np.uint8)

    for l in range(length):
        for w in range(width):
            pom = int(matrix[l, w]) + const
            if pom > 255:
                pom = 255
            elif pom < 0:
                pom = 0
            result[l, w] = pom

    path = str(ex) + str(pictureName) + '_moved_' + str(const) + '
    ↘ _histogram.png'
```

```
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_moved_' + str(const) + '.'
        ↪ png'
    self.savePicture(result, path)
```

6.3 Rozciąganie histogramu

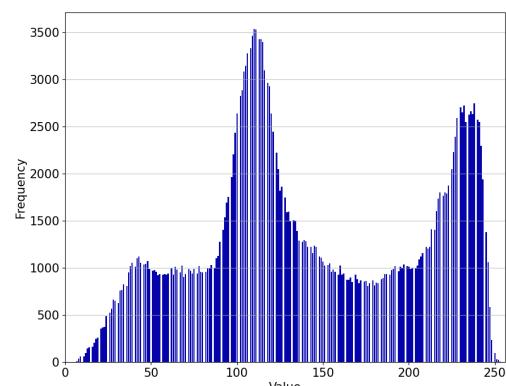
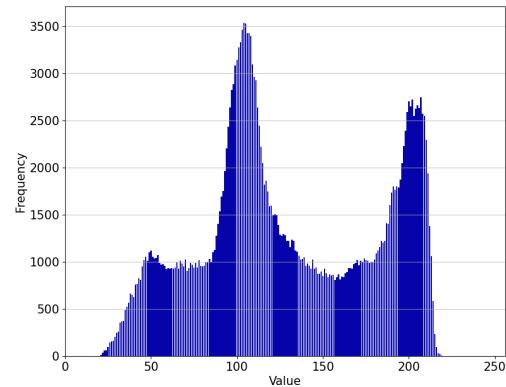
Opis algorytmu

Rozciąganie histogramu dokonuje się na obrazie, którego poziomy szarości nie są rozpięte na cały możliwy zakres. W efekcie rozciągnięcia histogramu otrzymujemy obraz o poziomach szarości rozpiętych na cały możliwy zakres np. [0, 255]

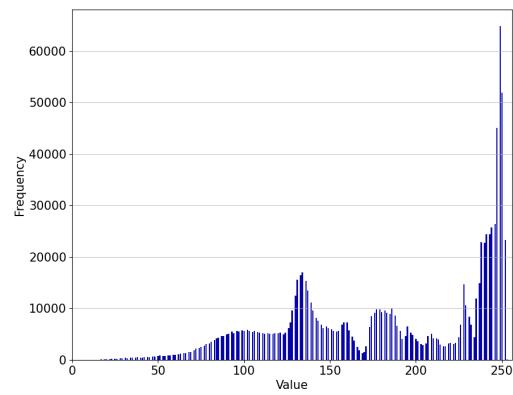
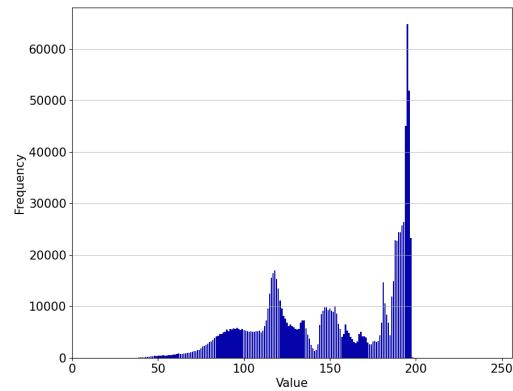
1. Znajdź w obrazie najmniejszy i największy poziom szarości (wartość piksla).
2. Dla każdego piksla w obrazie oblicz jego nową wartość według wzoru:

$$P_n = \frac{255}{(max - min) * (P_o - min)}$$

Efekty wykorzystania algorytmu



Rysunek 6.5: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po rozciągnięciu histogramu, histogram obrazu po przesunięciu



Rysunek 6.6: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po rozciągnięciu histogramu, histogram obrazu po przesunięciu

Kod źródłowy algorytmu

```
def extendHistogram(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/5/53/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width), np.uint8)
    vMax = 0
    vMin = 255
    for l in range(length):
        for w in range(width):
            pom = matrix[l, w]
            if pom > vMax and pom != 255:
                vMax = pom
            if pom < vMin:
                vMin = pom
    for l in range(length):
```

```
for w in range(width):
    pom = ((int(matrix[l, w]) - vMin) * 255) / (vMax - vMin)
    if pom > 255:
        pom = 255
    elif pom < 0:
        pom = 0
    result[l, w] = pom
path = str(ex) + str(pictureName) + '_extended_histogram.png'
self.calculateHistogram(result, path)
path = str(ex) + str(pictureName) + '_extended.png'
self.savePicture(result, path)
```

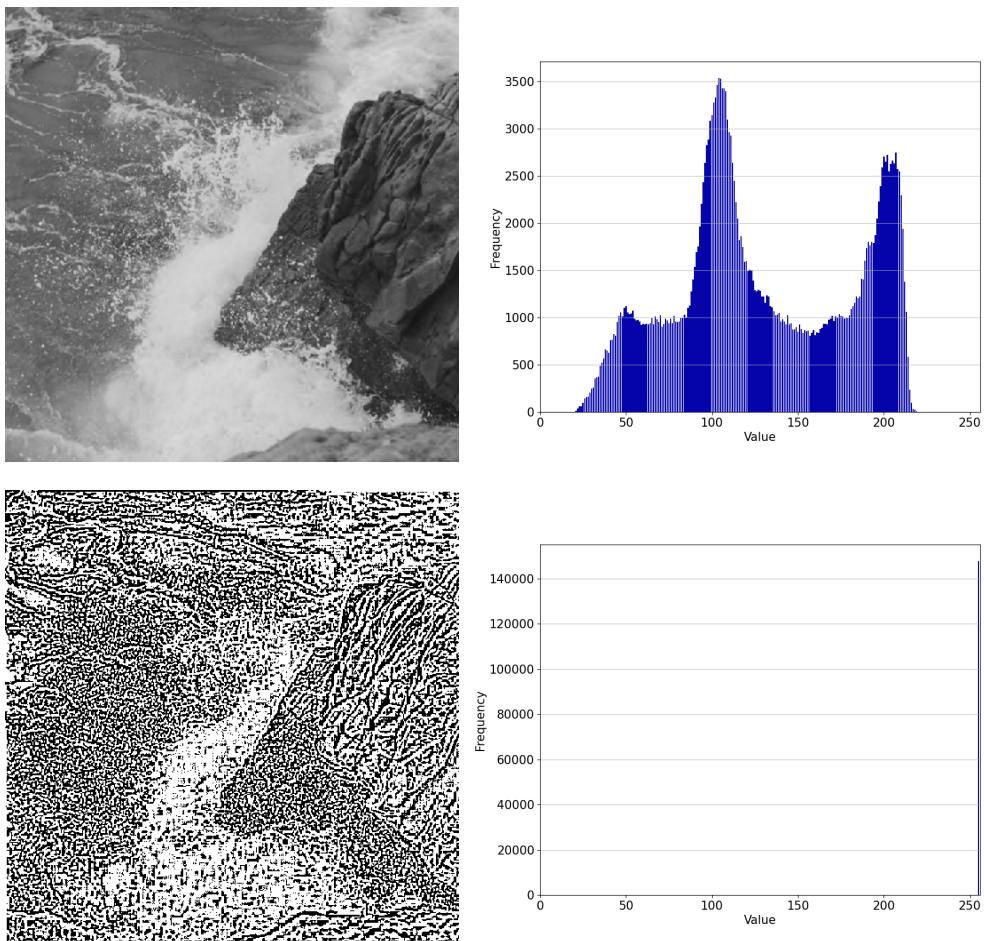
6.4 Progowanie lokalne

Opis algorytmu

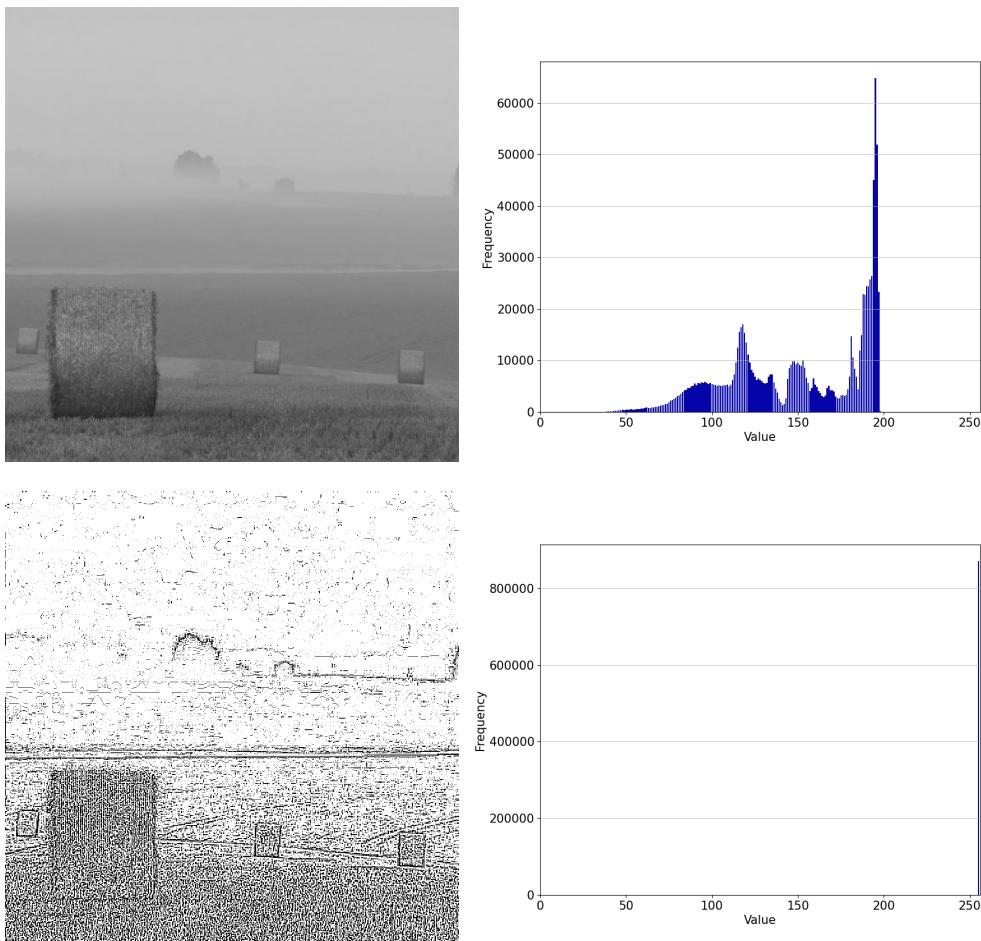
Progowanie lokalne oblicza wartość progową dla każdego piksla z osobna. Jest to jedna z metod binaryzacji obrazu, która w wyniku dokładniej odwzorowuje kształt obiektu na obrazie.

1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla w obrazie oblicz wartość progową jako średnią wartość piksli w otoczeniu danego piksla:
 - jeżeli wartość piksla środkowego jest mniejsza niż 0 to przypisz mu wartość 0.
 - jeżeli wartość piksla środkowego jest większa niż 0 to przypisz mu wartość 255.

Efekty wykorzystania algorytmu



Rysunek 6.7: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu lokalnym, histogram obrazu po progowaniu lokalnym



Rysunek 6.8: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu lokalnym, histogram obrazu po progowaniu lokalnym

Kod źródłowy algorytmu

```
def localHistogram(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/5/54/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width), np.uint8)
    left, right = -(int(round(3 / 2))), int(round(3 / 2) + 1)
    for l in range(length):
        for w in range(width):
            n = 0
            threshold = 0
            pom = matrix[l, w]
            for lOff in range(left, right):
                for wOff in range(left, right):
                    lSafe = l if ((l + lOff) > (left + length)) else (
                        ↪ l + lOff)
                    if pom <= result[lSafe, wSafe]:
                        n += 1
                    if n >= threshold:
                        result[l, w] = 255
                    else:
                        result[l, w] = 0
            if n >= threshold:
                result[l, w] = 255
            else:
                result[l, w] = 0
```

```
wSafe = w if ((w + wOff) > (left + width)) else (w  
    ↪ + wOff)  
threshold += matrix[lSafe, wSafe]  
n += 1  
threshold = int(round(threshold / n))  
result[1, w] = 0 if (pom < threshold) else 255  
path = str(ex) + str(pictureName) + '_local_histogram.png'  
self.calculateHistogram(result, path)  
path = str(ex) + str(pictureName) + '_local.png'  
self.savePicture(result, path)
```

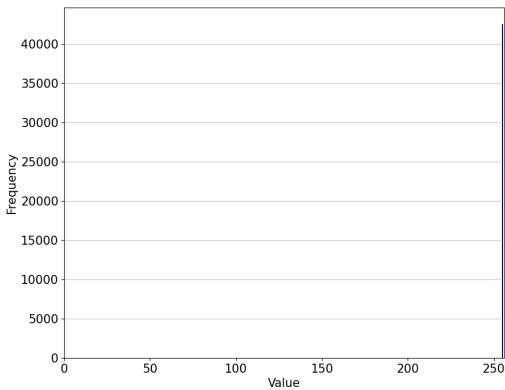
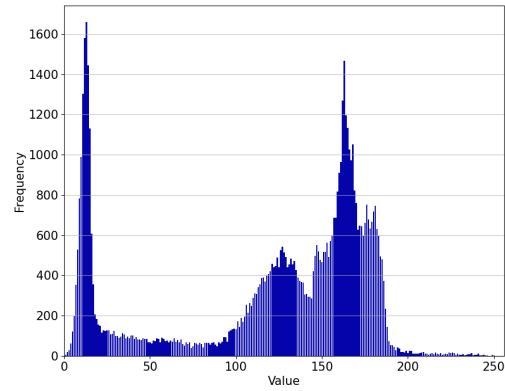
6.5 Progowanie globalne

Opis algorytmu

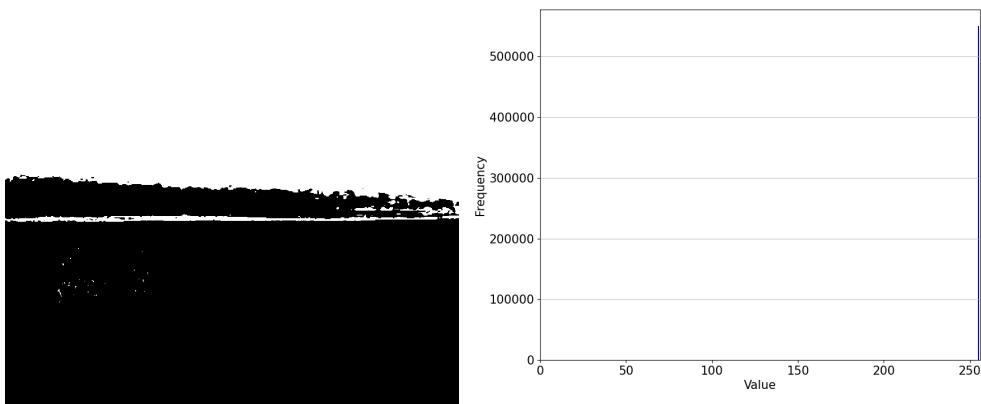
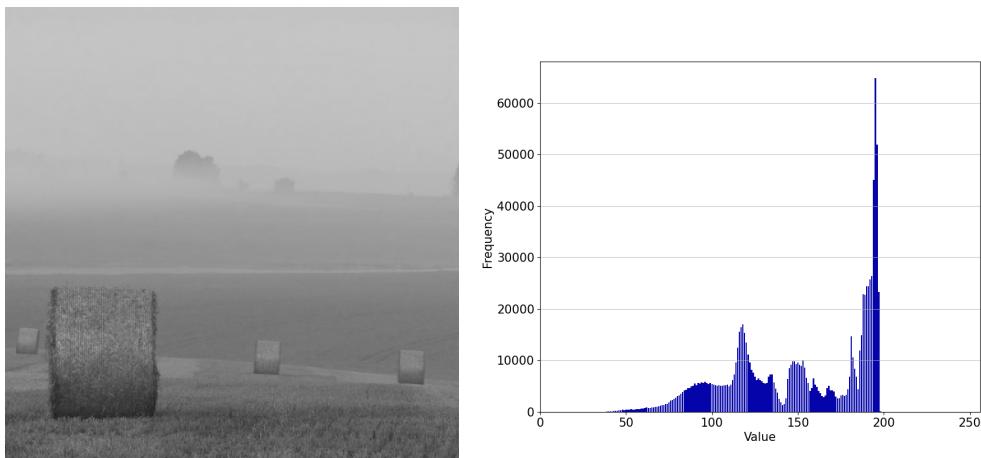
Progowanie globalne jest jedną z metod binaryzacji obrazu, która w wyniku daje obraz binarny. Wartość progowa jest ustalana globalnie biorąc pod uwagę wartość każdego piksla w obrazie, po czym stosuje wyliczony próg, aby nadać nową wartość każdemu piksowi.

1. Oblicz wartość progową T , jako wartość średnią wartość z wszystkich piksli w obrazie.
2. Dla każdego piksła w obrazie:
 - jeżeli wartość piksła środkowego jest mniejsza niż T to przypisz mu wartość 0.
 - jeżeli wartość piksła środkowego jest większa niż T to przypisz mu wartość 255.

Efekty wykorzystania algorytmu



Rysunek 6.9: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu globalnym, histogram obrazu po progowaniu globalnym



Rysunek 6.10: [Od lewej, rząd 1] Szary obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu globalnym, histogram obrazu po progowaniu globalnym

Kod źródłowy algorytmu

```

def globalHistogram(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/5/55/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width), np.uint8)
    sum = 0
    for l in range(length):
        for w in range(width):
            sum += matrix[l, w]
    mean = int(round(sum / (length * width)))
    for l in range(length):
        for w in range(width):
            pom = matrix[l, w]
            result[l, w] = 0 if (pom < mean) else 255
    path = str(ex) + str(pictureName) + '_global_histogram.png'

```

```
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_global.png'
    self.savePicture(result, path)
```

Rozdział 7

Operacje na histogramie obrazu barwowego

Histogram to najprostszy opis całościowy obrazu. Jego obliczanie polega na odczytaniu poziomu barwy każdego piksla i rejestrówaniu jego wystąpienia. Wyniki obliczeń w przedstawianych zadaniach są opisane za pomocą wykresu słupkowego, gdzie os OX reprezentuje poziomy barwy, a os OY reprezentuje liczbę ich wystąpień. Na podstawie histogramu obrazu możemy odczytać dane o jasności i intensywności obrazu.

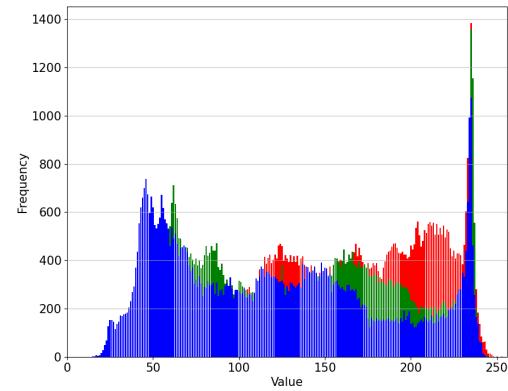
7.1 Obliczanie histogramu

Opis algorytmu

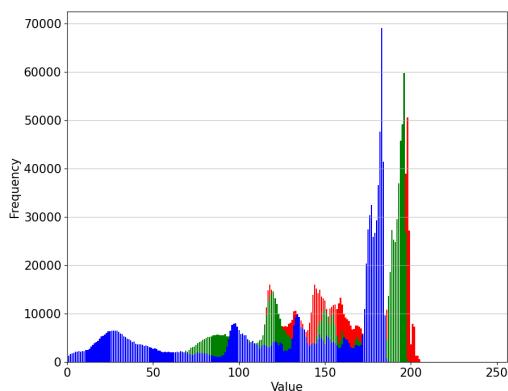
Histogram obrazu barwowego jest wykresem częstości występowania wartości barwowej piksla w obrazie tj. przyporządkowuje liczbę pikseli do danego poziomu natężenia barwy.

1. Zaalokuj tablicę 256 elementową (każdy element to 3 elementowa tablica wypełniona zerami), gdzie każdy element odpowiada poziomom natężenia barw piksla.
2. Dla każdego piksla zwięksź element tablicy o indeksie równym poziomowi barwy danego piksla.

Efekty wykorzystania algorytmu



Rysunek 7.1: [Od lewej] Barwy obraz wejściowy, histogram obrazu wejściowego



Rysunek 7.2: [Od lewej] Barwy obraz wejściowy, histogram obrazu wejściowego

Kod źródłowy algorytmu

```
def calculateHistogram(self, matrix, path):
    length, width, pictureName = self.pic.getPictureParameters()
    histogram = np.zeros((3, 256), np.int64)
    for l in range(length):
        for w in range(width):
            r = matrix[l, w][0]
            g = matrix[l, w][1]
            b = matrix[l, w][2]
            histogram[0][r] += 1
            histogram[1][g] += 1
            histogram[2][b] += 1
    bins = np.arange(257)
    self.plotHistogram(bins, histogram, path)

def plotHistogram(self, bins, hist, path):
    plt.figure(figsize=[10, 8])
    plt.bar(bins[:-1], hist[0], width=0.8, color='red')
    plt.bar(bins[:-1], hist[1], width=0.8, color='green')
    plt.bar(bins[:-1], hist[2], width=0.8, color='blue')
    plt.xlim(min(bins), max(bins))
    plt.grid(axis='y', alpha=0.75)
    plt.xlabel('Value', fontsize=15)
    plt.ylabel('Frequency', fontsize=15)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.ylabel('Frequency', fontsize=15)
    plt.savefig(path)
```

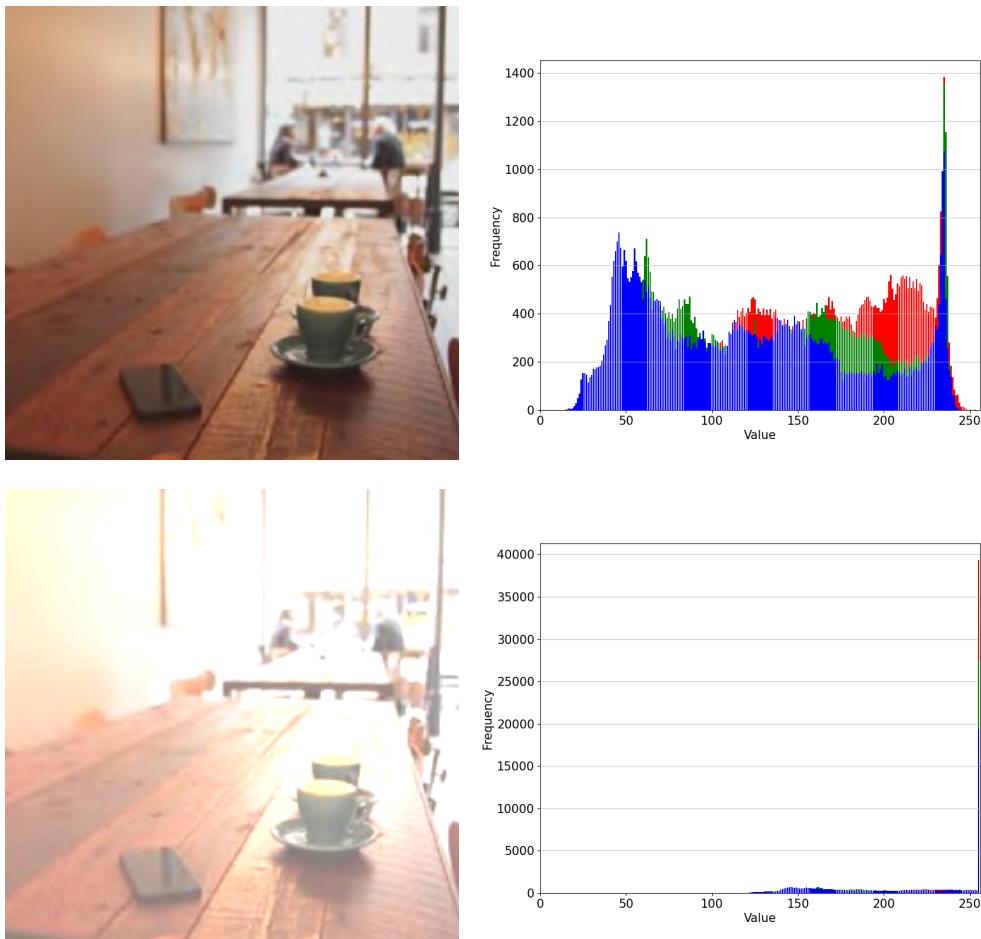
7.2 Przemieszczanie histogramu

Opis algorytmu

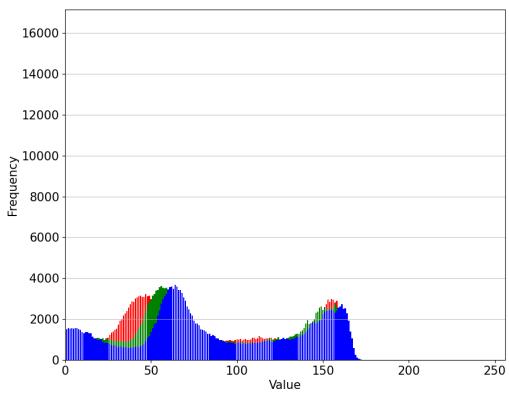
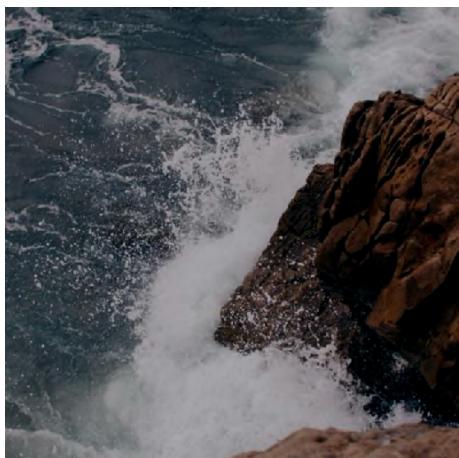
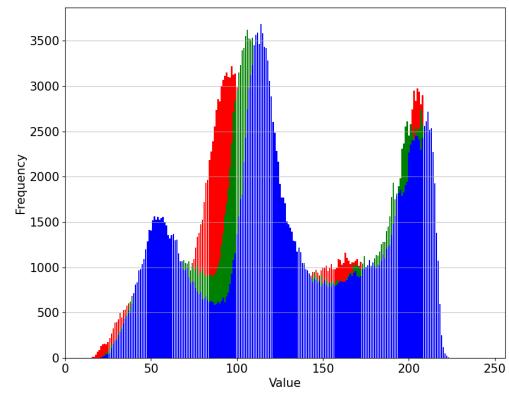
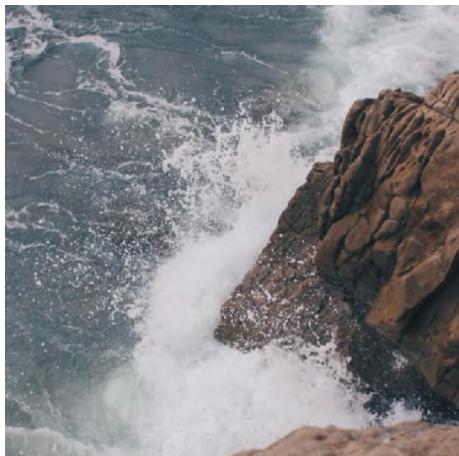
Przemieszczanie histogramu polega na dodaniu tej samej wartości od poziomu nateżenia każdej barwy każdego piksla w obrazie. W wyniku obraz jest równomiernie rozjaśniony bądź przesunięty. Dla każdej wartości barwy każdego piksla wykonaj:

1. Jeśli wartość barwy po operacji dodawania jest większa niż 255, przypisz jej wartość 255.
2. Jeśli wartość barwy po operacji dodawania jest mniejsza niż 0, przypisz jej wartość 0.

Efekty wykorzystania algorytmu



Rysunek 7.3: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po przesunięciu o 100, histogram obrazu po przesunięciu



Rysunek 7.4: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po przesunięciu o -50, histogram obrazu po przesunięciu

Kod źródłowy algorytmu

```
def moveHistogram(self, const):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    ex = './Documentation/images/ExEffects/6/62/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    for l in range(length):
        for w in range(width):
            r = int(matrix[l, w][0]) + const
            if r > 255:
                r = 255
            elif r < 0:
                r = 0
            result[l, w][0] = r
            g = int(matrix[l, w][1]) + const
            if g > 255:
                g = 255
            result[l, w][1] = g
            b = int(matrix[l, w][2]) + const
            if b > 255:
                b = 255
            result[l, w][2] = b
```

```
    elif g < 0:
        g = 0
    result[l, w][1] = g
    b = int(matrix[l, w][2]) + const
    if b > 255:
        b = 255
    elif b < 0:
        b = 0
    result[l, w][2] = b
path = str(ex) + str(pictureName) + '_moved_' + str(const) + '
    ↳ _histogram.png'
self.calculateHistogram(result, path)
path = str(ex) + str(pictureName) + '_moved_' + str(const) + '.
    ↳ png'
self.savePicture(result, path)
```

7.3 Rozciąganie histogramu

Opis algorytmu

Rozciąganie histogramu dokonuje się na obrazie, którego poziomy natężeń barw nie są rozpięte na cały możliwy zakres. W efekcie rozciągnięcia histogramu otrzymujemy obraz o poziomach natężeń barw rozpiętych na cały możliwy zakres np. [0, 255]

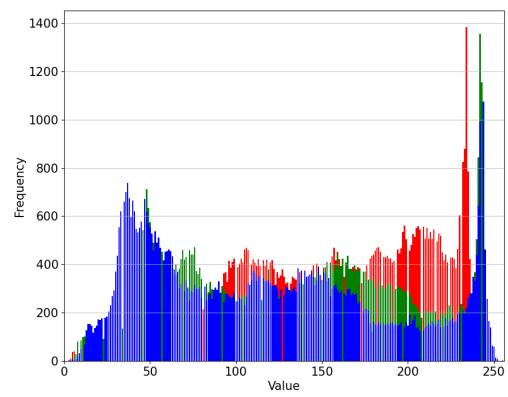
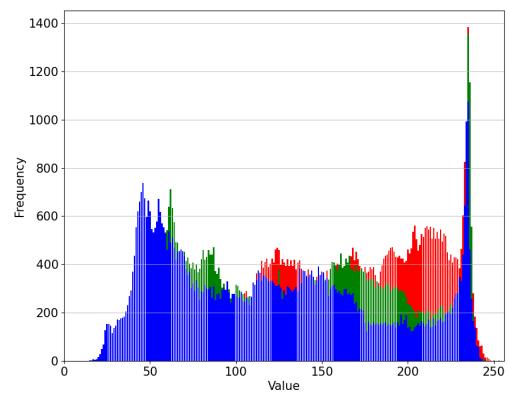
1. Znajdź w obrazie najmniejszy i największy poziom natężeń barwy (wartość piksla odpowiadająca natężeniu barwy).
2. Dla każdego piksla w obrazie oblicz jego nową wartość według wzoru:

$$P_{Rn} = \frac{255}{(max_R - min_R) * (P_{Ro} - min_R)}$$

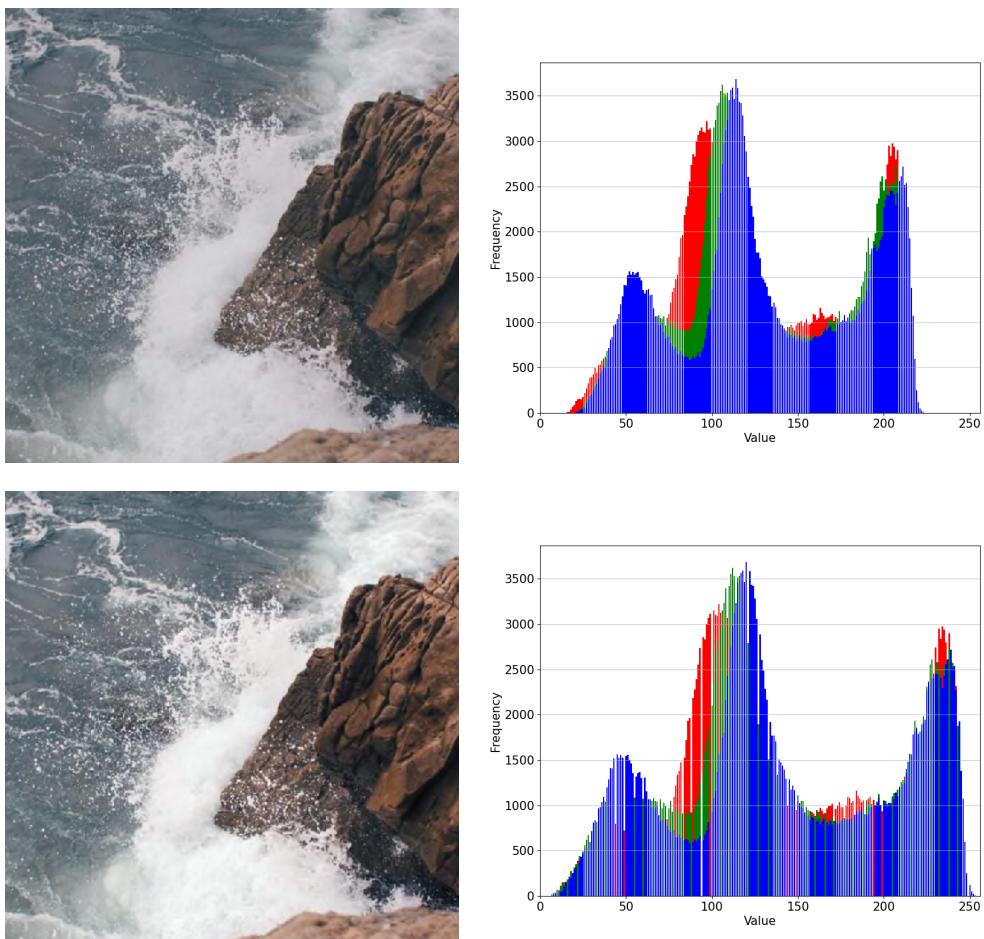
$$P_{Bn} = \frac{255}{(max_B - min_B) * (P_{Bo} - min_B)}$$

$$P_{Gn} = \frac{255}{(max_G - min_G) * (P_{Go} - min_G)}$$

Efekty wykorzystania algorytmu



Rysunek 7.5: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po rozcięgnięciu histogramu, histogram obrazu po przesunięciu



Rysunek 7.6: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po rozciągnięciu histogramu, histogram obrazu po przesunięciu

Kod źródłowy algorytmu

```
def extendHistogram(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/6/63/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    vMax = [0] * 3
    vMin = [255] * 3
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = matrix[l, w][color]
                if pom > vMax[color] and pom != 255:
                    vMax[color] = pom
                if pom < vMin[color]:
                    vMin[color] = pom
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = ((int(matrix[l, w][color]) - vMin[color]) * 255)
                ↪ / (vMax[color] - vMin[color])
                if pom > 255:
                    pom = 255
                elif pom < 0:
                    pom = 0
                result[l, w][color] = pom
    path = str(ex) + str(pictureName) + '_extended_histogram.png'
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_extended.png'
    self.savePicture(result, path)
```

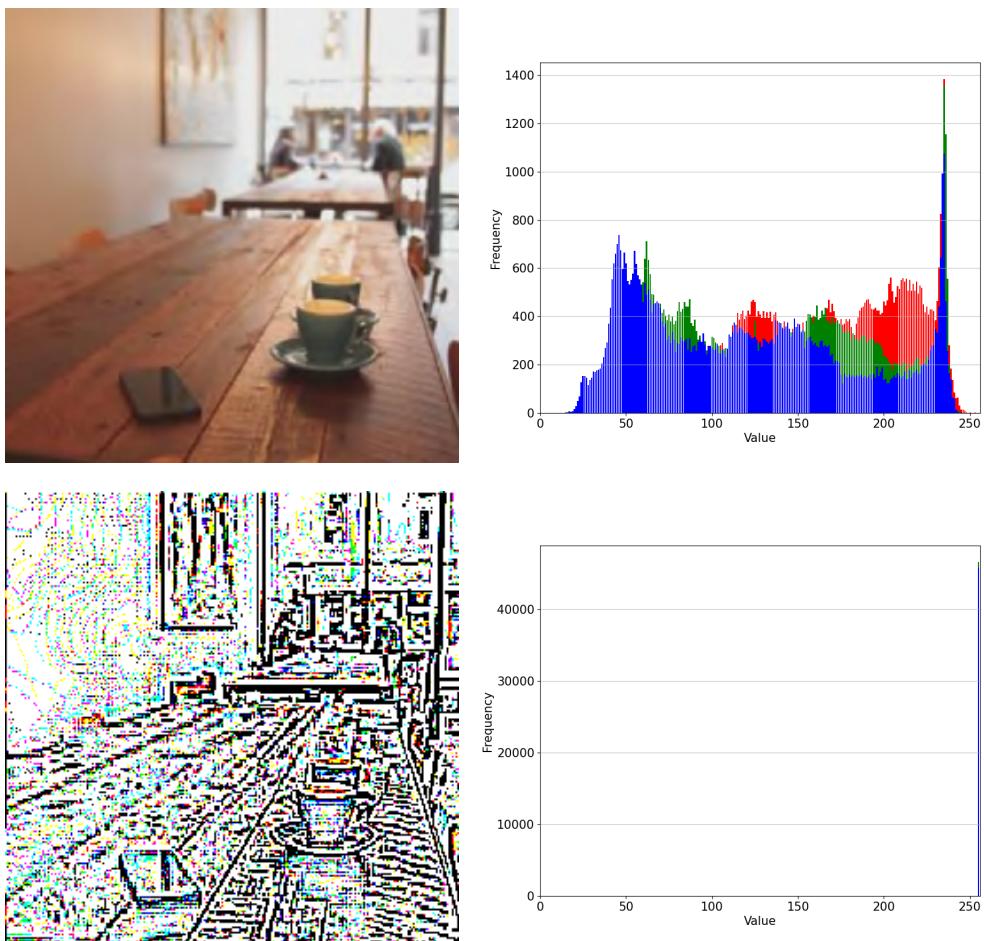
7.4 Progowanie 1-progowe lokalne

Opis algorytmu

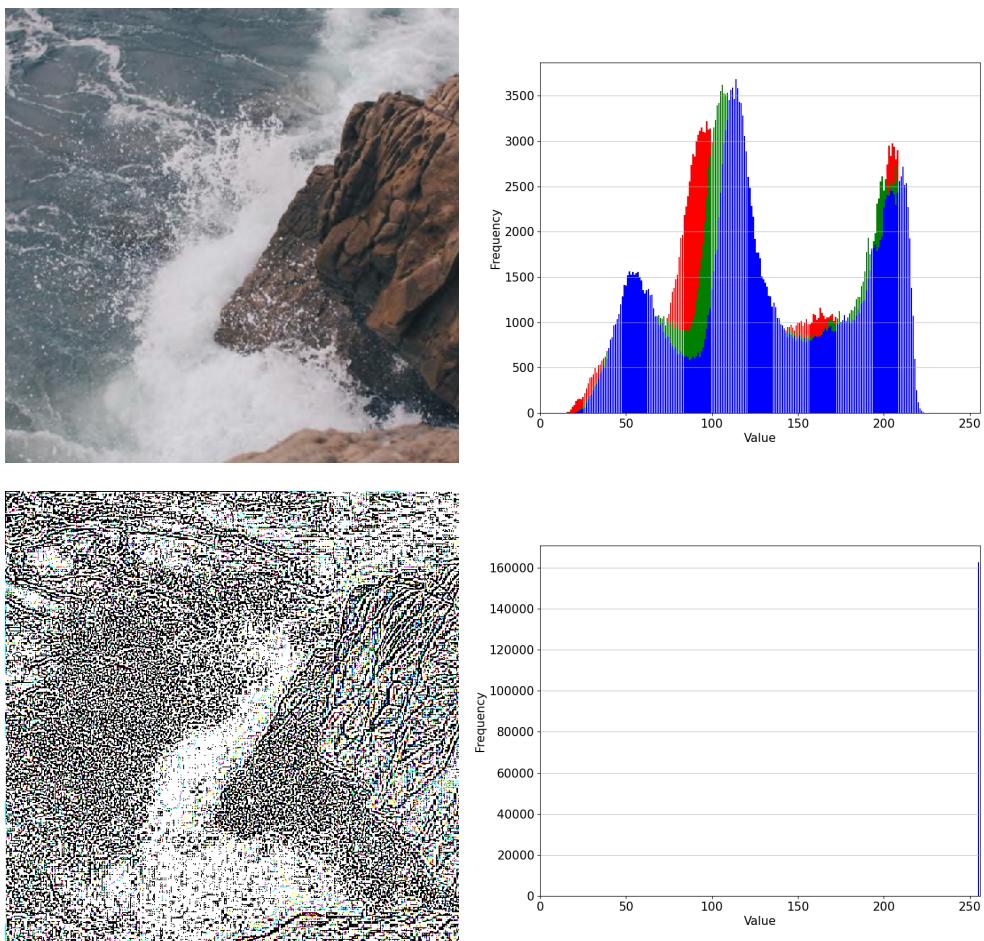
Progowanie 1-progowe lokalne oblicza wartość progową dla każdego piksla z osobna. W wyniku takiego progowania obraz dokładniej odzworzorowuje kształt obiektu. Próg obliczany jest jako średnia wartość piksli w obrazie, dla każdego piksla z osobna.

1. Zdefiniuj wielkość otoczenia piksla - wartość musi być nieparzysta, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla (P):
 - Dla każdego kanału (C):
 - Oblicz wartość progową T_c jako średnią wartość piksli P_c w otoczeniu danego piksla P .
 - Jeśli wartość piksla P_c jest mniejsza niż T_c to przypisz mu wartość 0.
 - Jeśli wartość piksla P_c jest większa niż T_c to przypisz mu wartość 255.

Efekty wykorzystania algorytmu



Rysunek 7.7: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu 1-progowym lokalnym histogramu, histogram obrazu



Rysunek 7.8: [Od lewej, rzad 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rzad 2] Obraz po progowaniu 1-progowym lokalnym histogramu, histogram obrazu

Kod źródłowy algorytmu

```
def oneThresholdLocalHistogram(self, dim=3):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/6/64/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    low, up = -(int(dim / 2)), (int(dim / 2) + 1)
    for color in range(3):
        for l in range(length):
            for w in range(width):
                n = 0
                threshold = 0
                pom = matrix[l, w][color]
                for lOff in range(low, up):
                    for wOff in range(low, up):
                        lSafe = l if ((l + lOff) > (low + length)) else
                            ↪ (l + lOff)
                        wSafe = w if ((w + wOff) > (low + width)) else
                            ↪ (w + wOff)
                        threshold += matrix[lSafe, wSafe][color]
                        n += 1
                threshold = int(round(threshold / n))
                result[l, w][color] = 0 if (pom < threshold) else 255
    path = str(ex) + str(pictureName) + '_oneThresholdLocal_histogram
        ↪ .png'
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_oneThresholdLocal.png'
    self.savePicture(result, path)
```

7.5 Progowanie 1 progowe globalne

Opis algorytmu

Progowanie 1-progowe globalne oblicza wartość progową dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa jest stosowana do nadania każdemu pikselowi nowej wartości.

1. Zdefiniuj wielkość otoczenia piksla - wartość musi być nieparzysta, aby mógł istnieć piksel środkowy).
2. Zdefiniuj ilość progów (T).
3. Dla każdego piksla (P):
 - Dla każdego kanału (C):
 - Znajdź $\text{MAX}(P_c)$ i $\text{MIN}(P_c)$.
 - Oblicz skalę S_c według wzoru:
4. Wylicz nową wartość piksla (P_{c_n}) wg. wzoru:

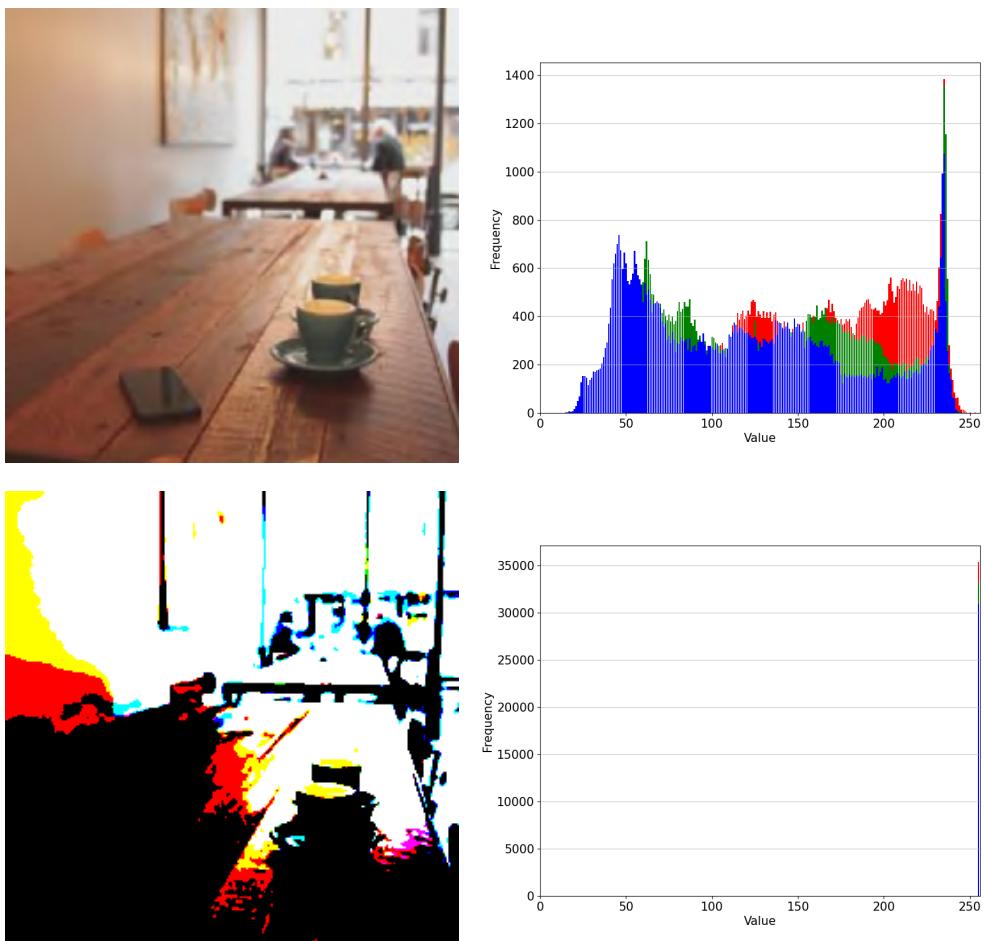
$$S_c = \frac{\text{MAX}(P_c)}{(T - 1)}$$

- Jeśli skala S_c jest 0 przypisz jej wartość 1, aby uniknąć dzielenia przez 0.

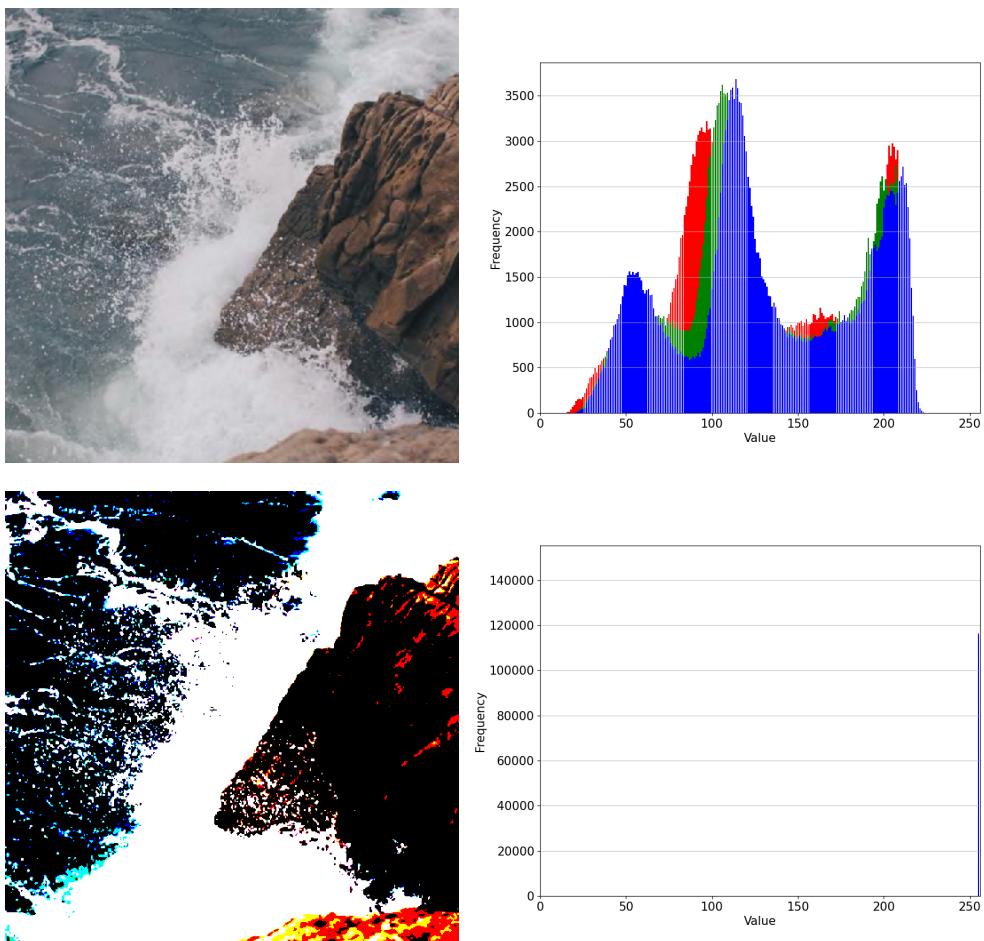
4. Wylicz nową wartość piksla (P_{c_n}) wg. wzoru:

$$P_{c_n} = \left\lceil \frac{P_c}{S_c} \right\rceil * S_c$$

Efekty wykorzystania algorytmu



Rysunek 7.9: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu 1-progowym globalnym histogramu, histogram obrazu



Rysunek 7.10: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu 1-progowym globalnym histogramu, histogram obrazu

Kod źródłowy algorytmu

```
def oneThresholdGlobalHistogram(self):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/6/66/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    added = [0] * 3
    mean = [0] * 3
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = matrix[l, w][color]
                added[color] += pom
        mean[color] = int(round(added[color] / (length * width)))
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = matrix[l, w][color]
                result[l, w][color] = 0 if (pom < mean[color]) else
                    ↪ 255
    path = str(ex) + str(pictureName) +
        ↪ _oneThresholdGlobal_histogram.png'
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_oneThresholdGlobal.png'
    self.savePicture(result, path)
```

7.6 Progowanie wieloprogowe lokalne

Opis algorytmu

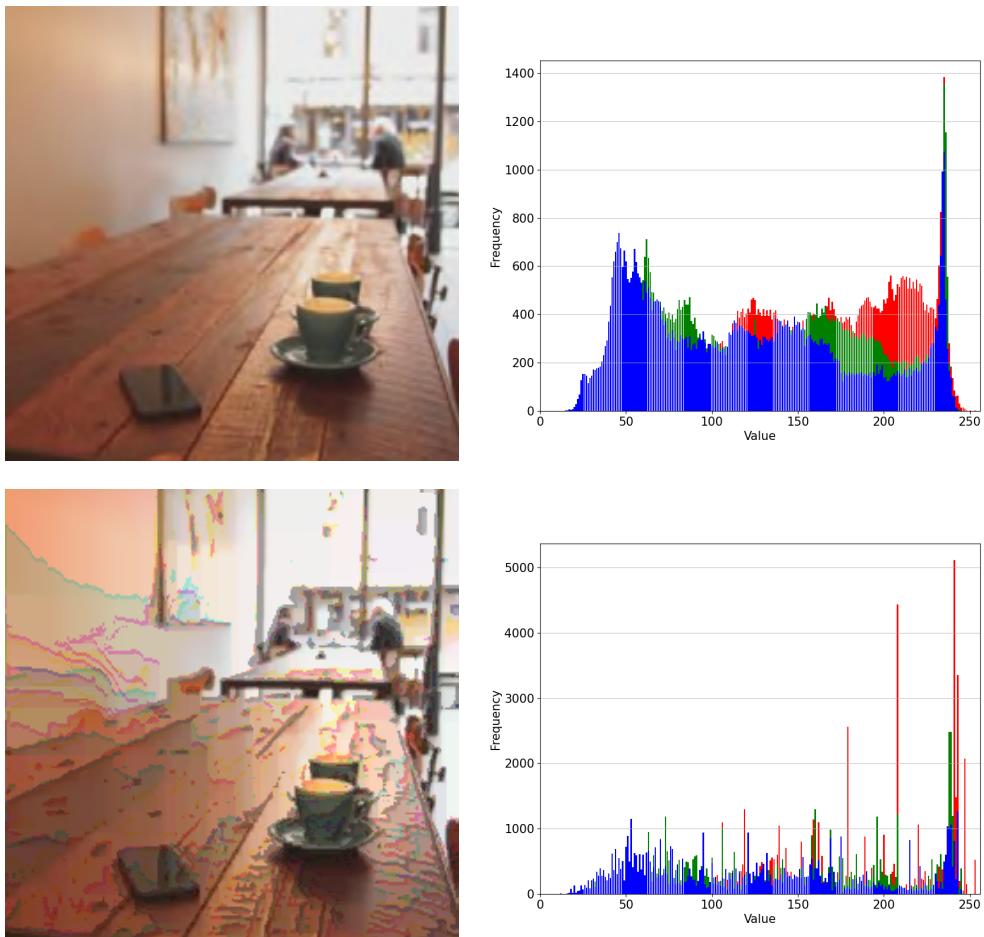
Progowanie wieloprogowe globalne oblicza wartość progową dla każdego piksla z osobna. W wyniku takiego progowania obraz ma mniejszą ilość kolorów w obrazie. Progi obliczane są dla każdego kanału z osobna.

1. Zdefiniuj wielkość otoczenia piksla - wartość musi być nieparzysta, aby mógł istnieć piksel środkowy).
2. Zdefiniuj ilość progów (T).
3. Dla każdego piksla (P):
 - Dla każdego kanału (C):
 - Znajdź $MAX(P_c)$ i $MIN(P_c)$.
 - Oblicz skalę S_c według wzoru:
4. Wylicz nową wartość piksla (P_{c_n}) wg. wzoru:

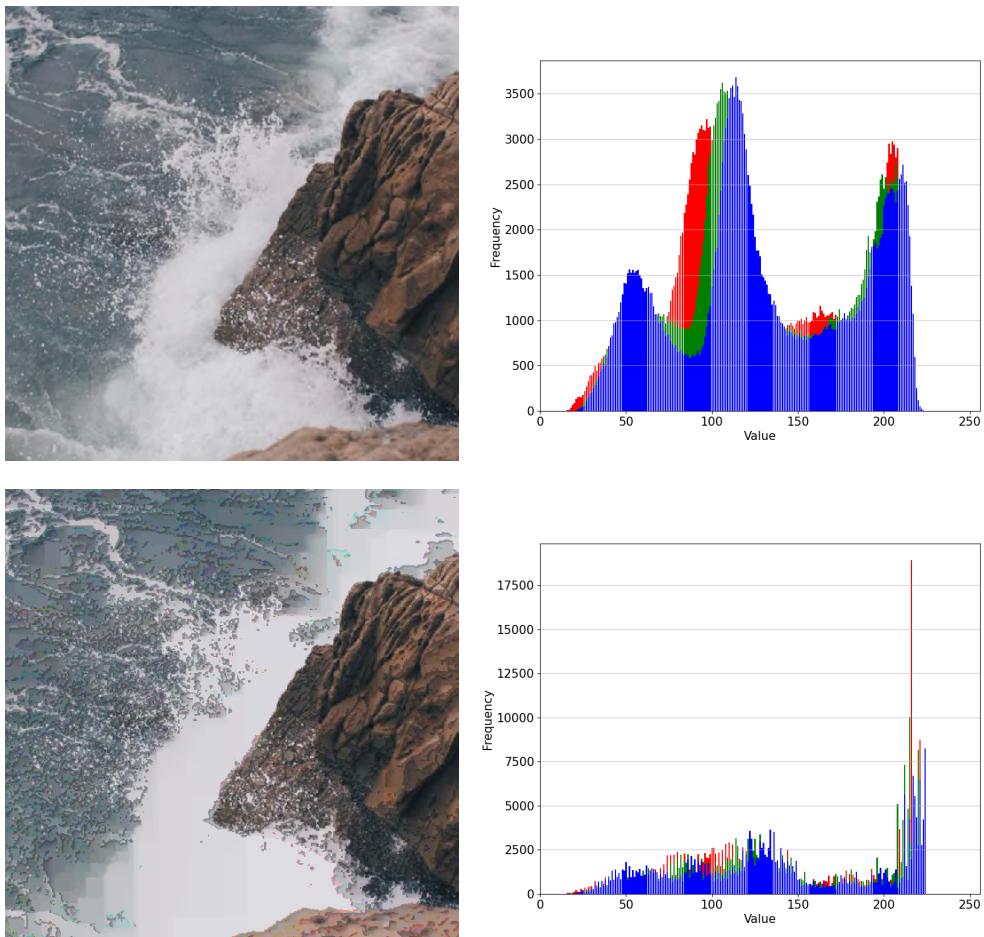
$$S_c = \frac{MAX(P_c)}{(T - 1)}$$

- Jeśli skala S_c jest 0 przypisz jej wartość 1, aby uniknąć dzielenia przez 0.

Efekty wykorzystania algorytmu



Rysunek 7.11: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu wieloprogowym lokalnym histogramu, histogram obrazu



Rysunek 7.12: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu wieloprogowym lokalnym histogramu, histogram obrazu

Kod źródłowy algorytmu

```
def multiThresholdLocalHistogram(self, dim=3, bins=4):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/6/65/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    low, up = -(int(dim / 2)), (int(dim / 2) + 1)
    for l in range(length):
        for w in range(width):
            pom = matrix[l, w]
            vMax = [0] * 3
            vMin = [255] * 3
            for lOff in range(low, up):
                for wOff in range(low, up):
                    lSafe = l if ((l + lOff) > (low + length)) else (l
                        ↪ + lOff)
                    wSafe = w if ((w + wOff) > (low + width)) else (w
                        ↪ + wOff)
                    pom = matrix[lSafe, wSafe]
                    for k in range(3):
                        vMax[k] = max(vMax[k], pom[k])
                        vMin[k] = min(vMin[k], pom[k])
            scale = [0] * 3
            for k in range(3):
                scale[k] = vMax[k] / (bins - 1)
                if scale[k] == 0:
                    scale[k] = 1
            for k in range(3):
                pom[k] = int(round(pom[k] / scale[k])) * scale[k]
            result[l, w] = pom
    path = str(ex) + str(pictureName) + '_' + str(bins) + '_'
    ↪ ThresholdLocal_histogram.png'
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_' + str(bins) + '_'
    ↪ ThresholdLocal.png'
    self.savePicture(result, path)
```

7.7 Progowanie wieloprogowe globalne

Opis algorytmu

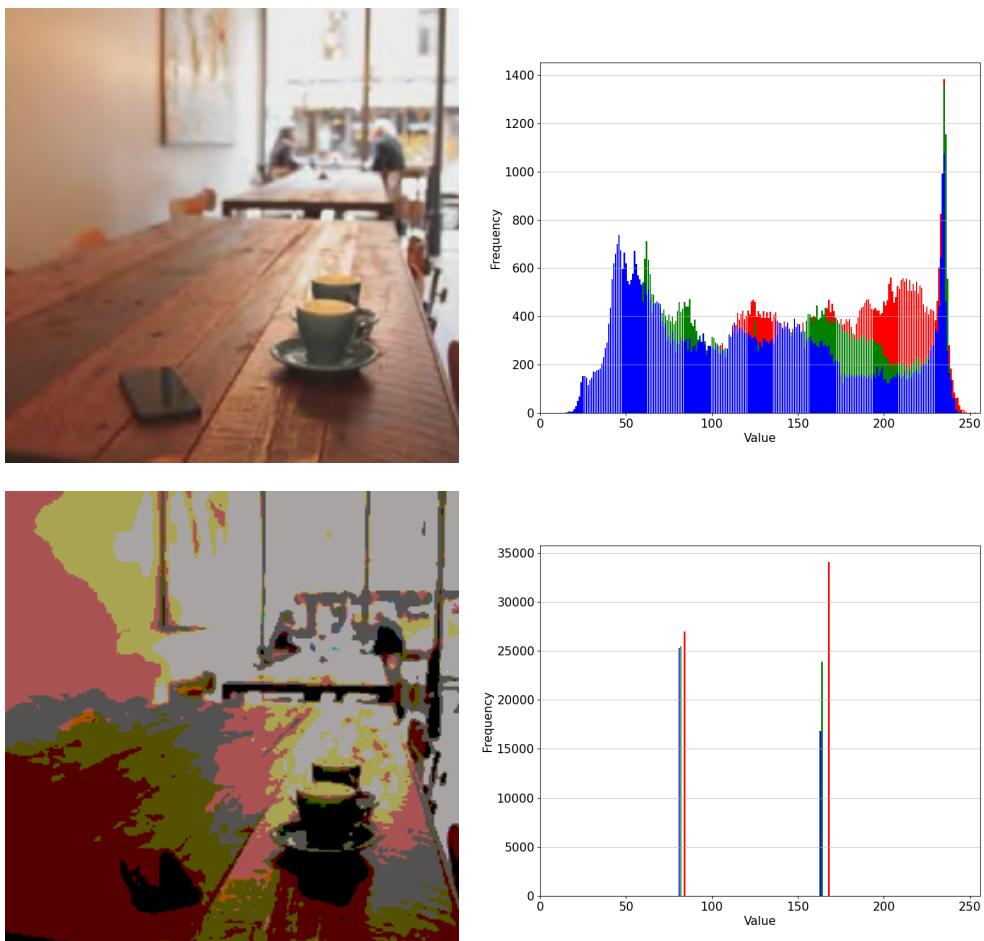
Progowanie wieloprogowe globalne oblicza wartość progową dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa jest stosowana do nadania każdemu pikselowi nowej wartości.

1. Zdefiniuj ilość progów (T).
2. Dla każdego piksla (P):
 - Dla każdego kanału (C):
 - Znajdź $MAX(P_c)$ i $MIN(P_c)$.
 - Oblicz skalę S_c według wzoru:
3. Wylicz nową wartość piksla (P_{c_n}) wg. wzoru:

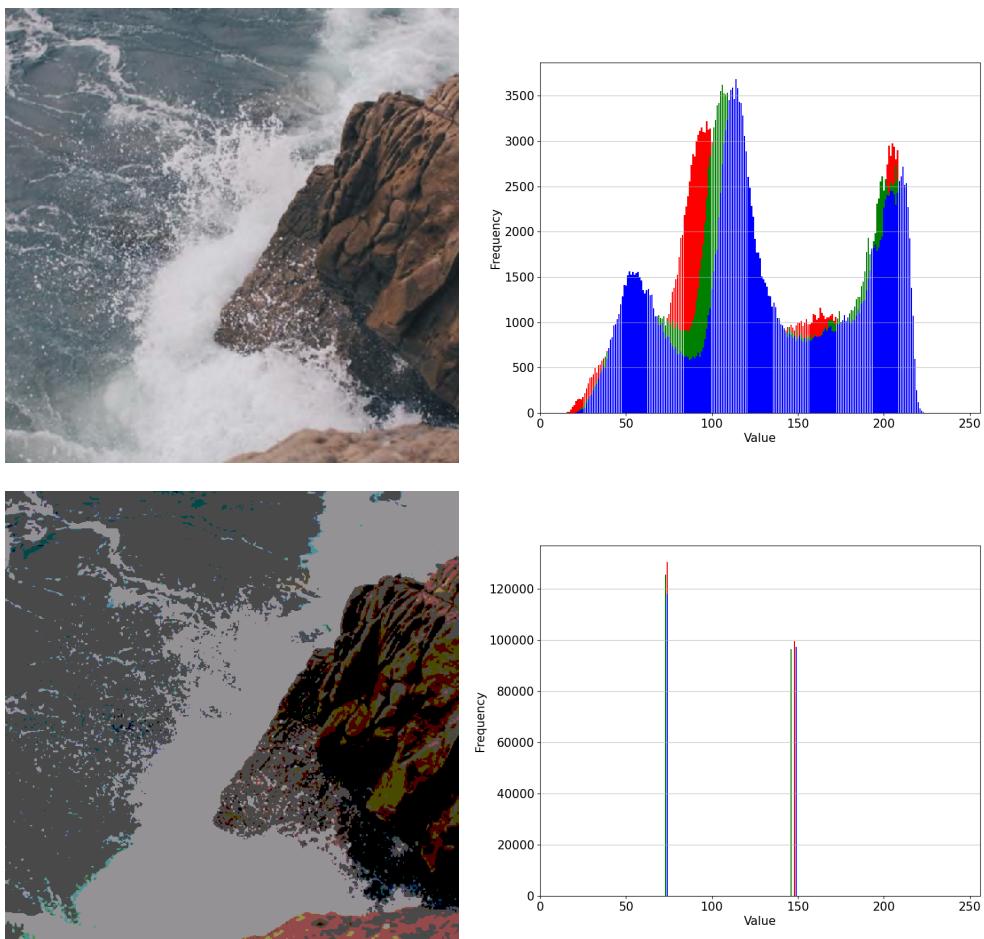
$$S_c = \frac{MAX(P_c)}{(T - 1)}$$

- Jeśli skala S_c jest 0 przypisz jej wartość 1, aby uniknąć dzielenia przez 0.

Efekty wykorzystania algorytmu



Rysunek 7.13: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu wieloprogowym globalnym histogramu, histogram obrazu



Rysunek 7.14: [Od lewej, rząd 1] Barwowy obraz wejściowy, histogram obrazu wejściowego [Od lewej, rząd 2] Obraz po progowaniu wieloprogowym globalnym histogramu, histogram obrazu

Kod źródłowy algorytmu

```
def multiThresholdGlobalHistogram(self, bins=4):
    length, width, pictureName = self.pic.getPictureParameters()
    matrix = self.pic.getGreyMatrix()
    # save basic histogram of modified picture
    ex = './Documentation/images/ExEffects/6/67/'
    path = str(ex) + str(pictureName) + '_histogram.png'
    self.calculateHistogram(matrix, path)
    result = np.zeros((length, width, 3), np.uint8)
    vMax = [0] * 3
    vMin = [255] * 3
    scale = [0] * 3
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = matrix[l, w][color]
                if pom > vMax[color]:
                    vMax[color] = pom
                if pom > vMin[color]:
                    vMin[color] = pom
    for color in range(3):
        scale[color] = vMax[color] / (bins - 1)
    for color in range(3):
        for l in range(length):
            for w in range(width):
                pom = matrix[l, w][color]
                result[l, w][color] = int(round(pom) / scale[color]) *
                    scale[color]
    path = str(ex) + str(pictureName) +
        _multiThresholdGlobal_histogram.png'
    self.calculateHistogram(result, path)
    path = str(ex) + str(pictureName) + '_multiThresholdGlobal.png'
    self.savePicture(result, path)
```