

Przetwarzanie obrazów

Sprawozdanie z laboratorium

Małgorzata Wiśniewska

Warszawa, 2020

Spis treści

Rozdział 1

Wstęp

1.1 Format obrazów

1.2 Instrukcja obsługi programu

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 piksli w kolumnach i wierszach w obu obrazach, poprzez zwiększenie liczby piksli 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):
    # porownaj wielkosc obrazow, jezeli sa tego samego rozmiaru
    # → nie rob nic
    if self.biggerPicture == 0 and self.smallerPicture == 0:
        print('Both_pictures_have_the_same_size')
        return 0
    # stworz 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 (\bar{x}) piksli 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(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.maxLength - 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.')

```

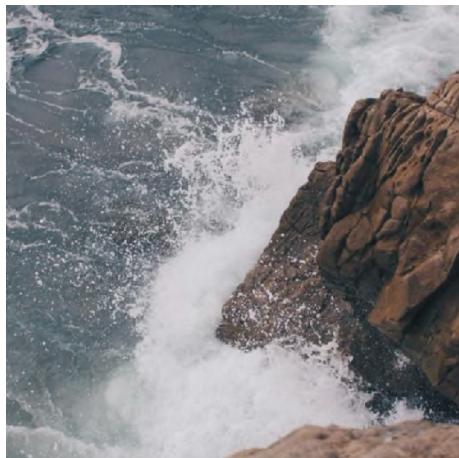
2.3 Ujednolicanie obrazów RGB geometryczne

Opis algorytmu

Operacje geometrycznego ujednolicania polega na wyrównaniu liczby piksli w kolumnach i wierszach w obu obrazach, poprzez zwiększenie liczby piksli 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

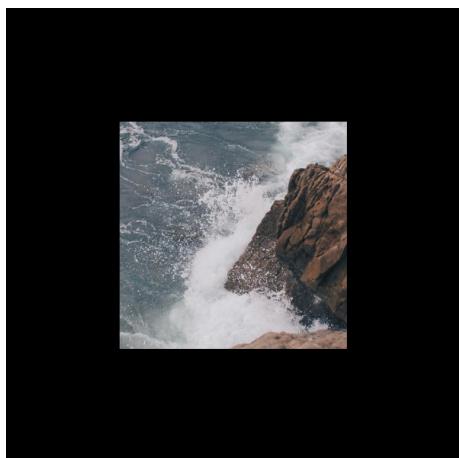


(a) Obraz 1: 512x512



(b) Obraz 2: 1025x1025

Rysunek 2.8: Obrazy wejściowe



(a) Obraz 1: 1025x1025

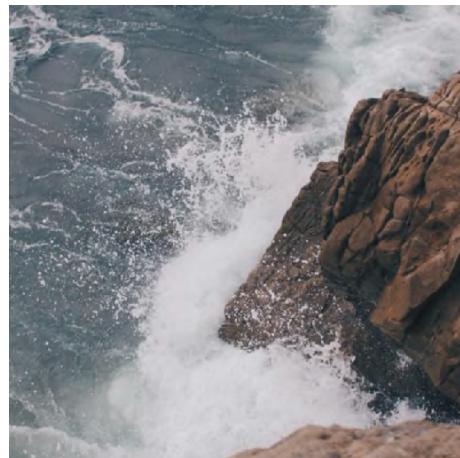


(b) Obraz 2: 1025x1025

Rysunek 2.9: Obrazy wyjściowe

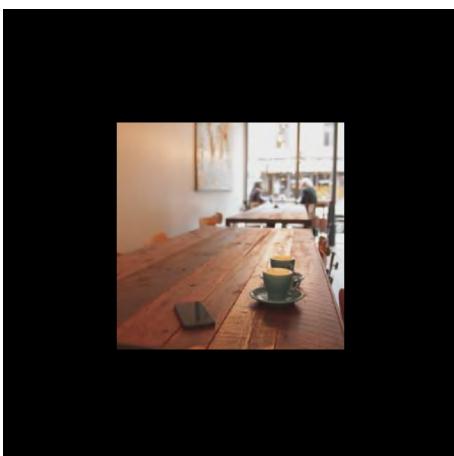


(a) Obraz 1: 256x256



(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
    # stwórz tablice z zerami jako odstawę 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)
```

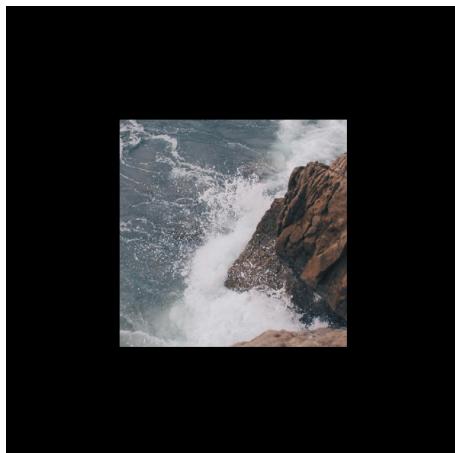
2.4 Ujednolicanie obrazów RGB rozdzielczościowe

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 (≥ 0) pikseli z jego bezpośredniego otoczenia.

Efekty wykorzystania algorytmu

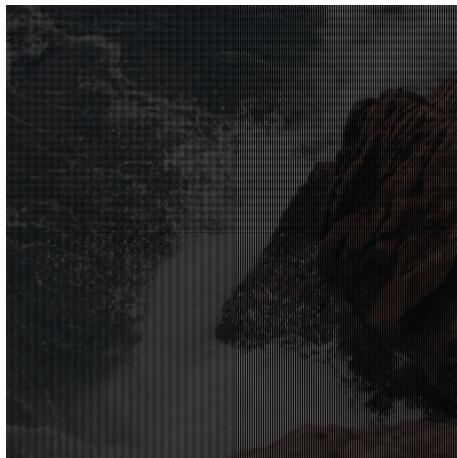


(a) Obraz 1: 1025x1025



(b) Obraz 2: 1025x1025

Rysunek 2.12: Obrazy wejściowe po ujednoliceniu geometrycznym

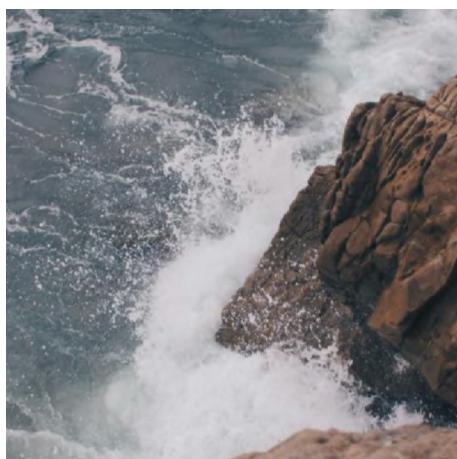


(a) Obraz 1: 1025x1025



(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(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.maxLength - 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

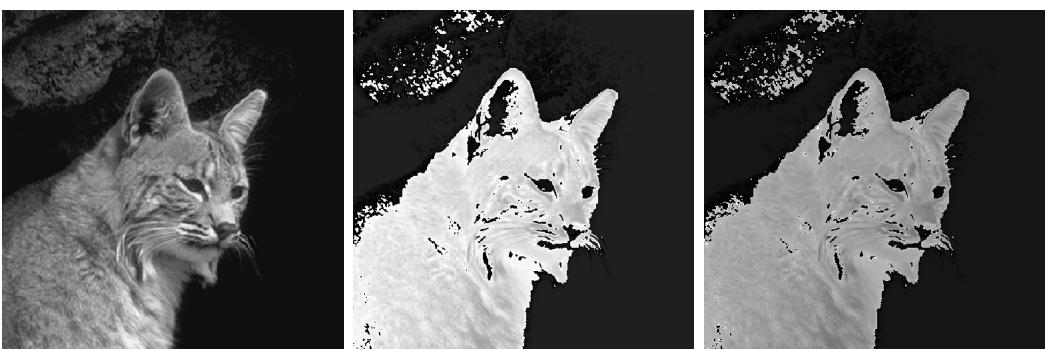
Algorytm sumowania obrazu szarego z określona stałą polega na dodaniu do każdej wartości pojedynczego piksla określonej stałej. Po operacji sumowania następuje normalizacja obrazu.

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. Operacje 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 odpowiadających piksli dwóch obrazów - 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, rzad 1] Szary obraz wejściowy 1, szary obraz wejściowy 2
[Od lewej, rzad 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 stałą.
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
- 3.5 Mieszanie obrazów z określonym współczynnikiem
- 3.6 Potęgowanie obrazu z zadaną potęgą
- 3.7 Dzielenie obrazów szarych przez zadaną stałą
- 3.8 Dzielenie obrazu przez inny obraz
- 3.9 Pierwiastkowanie obrazu
- 3.10 Logarytmowanie obrazu

Rozdział 4

Operacje sumowania arytmetycznego obrazów barwowych

4.1 Sumowanie obrazów barwowych

4.1.1 Sumowanie obrazu z określona stałą

4.1.2 Sumowanie dwóch obrazów

4.2 Mnożenie obrazów barwowych

4.2.1 Mnożenie obrazu przez określoną stałą

4.2.2 Mnożenie obrazu przez inny obraz

4.3 Mieszanie obrazów z określonym współczynnikiem

4.4 Potęgowanie obrazu z zadana potęgą

4.5 Dzielenie obrazów barwowych

4.5.1 Dzielenie obrazu przez zadana stałą

4.5.2 Dzielenie obrazu przez inny obraz

4.6 Pierwiastkowanie obrazu

4.7 Logarytmowanie obrazu

Rozdział 5

Operacje geometryczne na obrazie

5.1 Przemieszczanie obrazu o zadany wektor

5.2 Skalowanie obrazu

5.2.1 Skalowanie jednorodne

5.2.2 Skalowanie niejednorodne

5.3 Obracanie obrazu o dowolny kąt

5.4 Symetrie obrazu

5.4.1 Symetria względem osi OX

5.4.2 Symetria względem osi OY

5.4.3 Symetria względem zadanej prostej

5.5 Wycinanie fragmentów obrazów

5.6 Kopiowanie fragmentów obrazów

Rozdział 6

Operacje na histogramie obrazu szarego

- 6.1 Obliczanie histogramu
- 6.2 Przemieszczanie histogramu
- 6.3 Rozciąganie histogramu
- 6.4 Progowanie lokalne
- 6.5 Progowanie globalne

Rozdział 7

Operacje na histogramie obrazu barwowego

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