

Przetwarzanie Obrazów: Sprawozdanie

Damian Ubowski Maciej Tarach

Warszawa, 2019

Spis treści

1 Wstęp	7
1.1 Format obrazu	7
1.1.1 Struktura formatu	7
1.1.2 Przykładowa struktura IFF	7
1.1.3 Instrukcja obsługi programu	8
2 Operacje ujednoliciania obrazów	9
2.1 Ujednolicenie obrazów szarych geometryczne	9
2.2 Ujednolicenie obrazów szarych rozdzielczościowe	11
2.3 Ujednolicenie obrazów RGB geometryczne	14
2.4 Ujednolicenie obrazów RGB rozdzielczościowe	17
3 Operacje sumowania arytmetycznego obrazów szarych	21
3.1 Sumowanie (określonej) stałej z obrazem	21
3.2 Sumowanie dwóch obrazów	21
3.3 Mnożenie obrazu przez zadaną liczbę	21
3.4 Mnożenie obrazu przez inny obraz	21
3.5 Mieszanie obrazów z określonym współczynnikiem	21
3.6 Potęgowanie obrazu (z zadaną potegą)	21
3.7 Dzielenie obrazu przez (zadaną) liczbę	21
3.8 Dzielenie obrazu przez przez inny obraz	21
3.9 Pierwiastkowanie obrazu	21
3.10 Logarytmowanie obrazu	21
4 Operacje sumowania arytmetycznego obrazów barwowych	23
4.1 Sumowanie (określonej) stałej z obrazem	23
4.2 Sumowanie dwóch obrazów	23
4.3 Mnożenie obrazu przez zadaną liczbę	23
4.4 Mnożenie obrazu przez inny obraz	23
4.5 Mieszanie obrazów z określonym współczynnikiem	23
4.6 Potęgowanie obrazu (z zadaną potegą)	23
4.7 Dzielenie obrazu przez (zadaną) liczbę	23
4.8 Dzielenie obrazu przez przez inny obraz	23
4.9 Pierwiastkowanie obrazu	23
4.10 Logarytmowanie obrazu	23
5 Operacje geometryczne na obrazie	25
5.1 Przesunięcie obrazu o zadany wektor	25
5.2 Jednorodne skalowanie obrazu	26
5.3 Niejednorodne skalowanie obrazu	28
5.4 Obracanie obrazu o dowolny kąt	29
5.5 Symetrie względem osi układu	30
5.6 Symetrie względem zadanej prostej	32

5.7	Wycinanie fragmentów obrazu	33
5.8	Kopiowanie fragmentów obrazów	34
6	Operacje na histogramie obrazu szarego	35
6.1	Obliczanie histogramu	35
6.2	Przemieszczanie histogramu	35
6.3	Rozciąganie histogramu	35
6.4	Progowanie lokalne	35
6.5	Progowanie globalne	35
7	Operacje na histogramie obrazu barwowego	37
7.1	Obliczanie histogramu	37
7.2	Przemieszczanie histogramu	37
7.3	Rozciąganie histogramu	37
7.4	Progowanie 1-progowe lokalne	37
7.5	Progowanie wielo-progowe lokalne	37
7.6	Progowanie 1-progowe globalne	37
7.7	Progowanie wielo-progowe globalne	37
8	Operacje morfologiczne na obrazach binarnych	39
8.1	Erozja	39
8.1.1	Opis	39
8.1.2	Kod źródłowy algorytmu	39
8.2	Dylatacja	40
8.2.1	Opis	40
8.2.2	Kod źródłowy algorytmu	40
8.3	Otwarcie	41
8.3.1	Opis	41
8.3.2	Kod źródłowy algorytmu	41
8.4	Zamknięcie	42
8.4.1	Opis	42
8.4.2	Kod źródłowy algorytmu	43
9	Operacje morfologiczne na obrazach szarych	45
9.1	Erozja	45
9.1.1	Opis	45
9.1.2	Kod źródłowy algorytmu	47
9.2	Dylatacja	47
9.2.1	Opis	47
9.2.2	Kod źródłowy algorytmu	49
9.3	Otwarcie	50
9.3.1	Opis	50
9.3.2	Kod źródłowy algorytmu	51
9.4	Zamknięcie	52
9.4.1	Kod źródłowy algorytmu	53
10	Filtrowanie wygładzające liniowe i nieliniowe	55
10.1	Filtr dolnoprzepustowy uśredniający	55
10.1.1	Opis	55
10.1.2	Kod źródłowy algorytmu	56
10.2	Filtr dolnoprzepustowy Gaussowski	57
10.2.1	Opis	57
10.2.2	Kod źródłowy algorytmu	58

10.3 Filtr medianowy	59
10.3.1 Opis	59
10.3.2 Kod źródłowy algorytmu	60
10.4 Filtr modalny	60
10.4.1 Opis	60
10.4.2 Kod źródłowy algorytmu	61
10.5 Filtr Kuwahary	62
10.5.1 Opis	62
10.5.2 Kod źródłowy algorytmu	64
10.6 Filtr minimalny	66
10.7 Filtr maksymalny	66

Rozdział 1

Wstęp

1.1 Format obrazu

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

1.1.1 Struktura formatu

Pliki DjVu rozpoczynają się od swojej “Magic number” potwierdzającej rodzaj pliku i mającej wartość *0x41 0x54 0x26 0x54*. Następnie czerpiąc inspirację ze struktury IFF (**Interchange File Format**) plik dzieli się na kawałki (*ang. chunks*) zawierające interesujące nas cenne dane. Takie jak szerokość lub wysokość obrazu, dpi, informacje o kolorach, rozmieszczeniu pikseli, etc. Każdy kawałek składający się z ID typu, długości zawartości i samej zawartości tworzy zwarty format. Identyfikator typu określa rolę w jakiej przyjdzie służyć kawałkowi. Do dyspozycji ma ich całkiem sporo, ale uwzględniając najbardziej przydatne w naszym kontekście to ograniczymy liczbę do:

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

1.1.2 Przykładowa struktura IFF

FORM:DJVU [14260]

INFO [10]

Sjbz [13133]

FG44 [181]

BG44 [935]

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

1.1.3 Instrukcja obsługi programu

W celu uruchomienia kodu źródłowego będzie niezbędny:

- * [DjVuLibre](#) ($\geq 3.5.21$)
- * [Python](#) (≥ 2.6 lub $3.X$)
- * [Cython](#) (≥ 0.19 , lub ≥ 0.20 dla Python 3)
- * [pkg-config](#) (POSIX)

Rozdział 2

Operacje ujednolicania obrazów

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

2.1 Ujednolicenie obrazów szarych geometryczne

Algorytm

Opis

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

Kroki

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

Kod źródłowy algorytmu

```
def geometricGray(self):  
    print('geometric gray unification start')  
    width, height = self.firstDecoder.width, self.firstDecoder.height  
    if width < self.maxWidth or height < self.maxHeight:  
        # Create black background  
        firstResult = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)  
        # Copy smaller image to bigger  
        startWidthIndex = int(round((self.maxWidth - width) / 2))  
        startHeightIndex = int(round((self.maxHeight - height) / 2))  
        pixelsBuffer = self.firstDecoder.getPixels()  
        for h in range(0, height):  
            for w in range(0, width):  
                firstResult[h + startHeightIndex, w + startWidthIndex] = pixelsBuffer[h, w]  
        img = Image.fromarray(firstResult, mode='L')
```

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



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



```



```

2.2 Ujednolicenie obrazów szarych rozdzielczościowe

Algorytm

Opis

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

Kroki

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

Kod źródłowy algorytmu

```

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

def _scaleUpGray(self, decoder, outputPath):
    width, height = decoder.width, decoder.height
    scaleFactoryW = float(self.maxWidth) / width
    scaleFactoryH = float(self.maxHeight) / height

```

Rysunek 2.3: Skutki braku interpolacji



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



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



```

if width < self.maxWidth or height < self.maxHeight:
    pixelsBuffer = decoder.getPixels()
    result = numpy.zeros((self.maxHeight, self.maxWidth), numpy.uint8)
    # Fill values
    for h in range(height):
        for w in range(width):
            if w%2 == 0:
                result[int(scaleFactoryH * h), int(round(scaleFactoryW * w)) + 1] =
                    pixelsBuffer[h, w]
            if w%2 == 1:
                result[int(round(scaleFactoryH * h)) + 1, int(round(scaleFactoryW * w))] =
                    pixelsBuffer[h, w]
    # Interpolate
    self._interpolateGray(result)
    img = Image.fromarray(result, mode='L')
    img.save(outputPath)

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

```

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



2.3 Ujednolicenie obrazów RGB geometryczne

Algorytm

Opis

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

Kroki

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

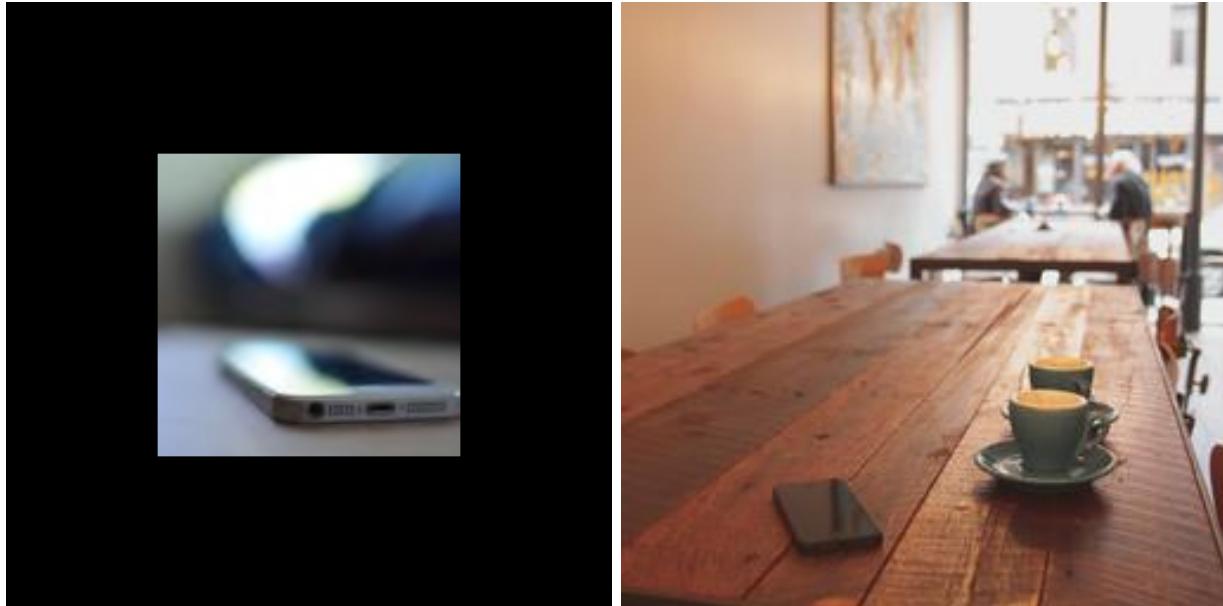
Rysunek 2.7: Po uruchomieniu algorytmu (od lewej): obraz 1 (1024x1024, 300dpi), obraz 2 (1024x1024, 300dpi)



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



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



Kod źródłowy algorytmu

```

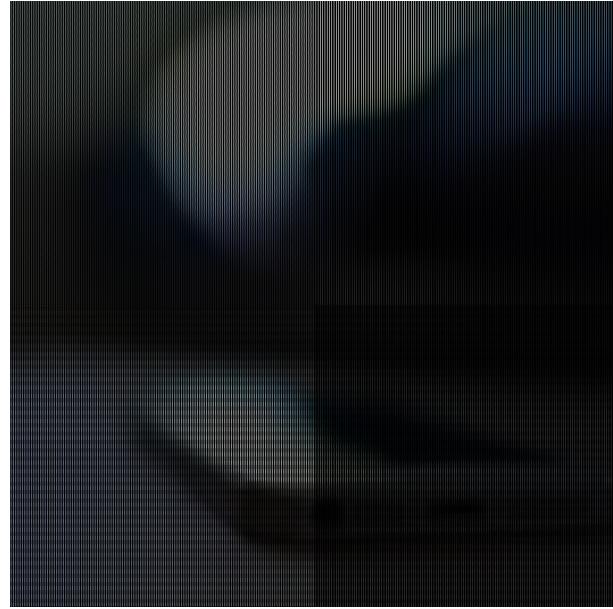
def geometricColor(self):
    print('geometric color unification start')
    self.firstDecoder.setColor()
    width, height = self.firstDecoder.width, self.firstDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        result = self._paintInMiddleColor(self.firstDecoder)
      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

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

def _paintInMiddleColor(self, decoder):
    # Create black background
    result = numpy.full((self.maxHeight, self.maxWidth, 3), 0, numpy.uint8)
    # Copy smaller image to bigger
    width, height = decoder.width, decoder.height
    startWidthIndex = int(round((self.maxWidth - width) / 2))
    startHeightIndex = int(round((self.maxHeight - height) / 2))
    pixelsBuffer = decoder.getPixels24Bits()
    for h in range(0, height):
        for w in range(0, width):
            result[h + startHeightIndex, w + startWidthIndex] = pixelsBuffer[h, w]
    return result

```

Rysunek 2.10: Skutki braku interpolacji



2.4 Ujednolicenie obrazów RGB rozdzielczościowe

Algorytm

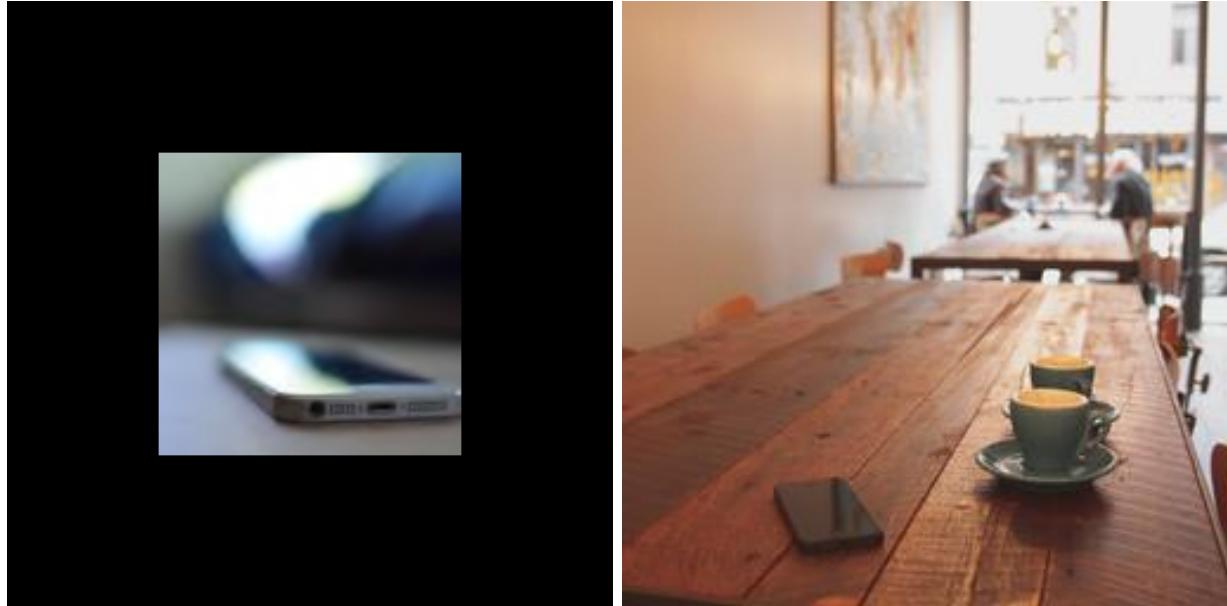
Opis

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

Kroki

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

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



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



Kod źródłowy algorytmu

```

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

def _scaleUpColor(self, decoder, outputPath):
    width, height = decoder.width, decoder.height
    scaleFactoryW = float(self.maxWidth) / width
    scaleFactoryH = float(self.maxHeight) / height
    if width < self.maxWidth or height < self.maxHeight:
        pixelsBuffer = decoder.getPixels24Bits()
        result = numpy.full((self.maxHeight, self.maxWidth, 3), 1, numpy.uint8)
        # Fill values
        for h in range(height):
            for w in range(width):
                if w%2 == 0:
                    result[int(scaleFactoryH * h), int(round(scaleFactoryW * w)) + 1] =
                        pixelsBuffer[h, w]
                if w%2 == 1:
                    result[int(round(scaleFactoryH * h)) + 1, int(scaleFactoryW * w)] =
                        pixelsBuffer[h, w]
        # Interpolate
        self._interpolateColor(result)
        img = Image.fromarray(result, mode='RGB')
        img.save(outputPath)

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

```


Rozdział 3

Operacje sumowania arytmetycznego obrazów szarych

- 3.1 Sumowanie (określonej) stałej z obrazem
- 3.2 Sumowanie dwóch obrazów
- 3.3 Mnożenie obrazu przez zadaną liczbę
- 3.4 Mnożenie obrazu przez inny obraz
- 3.5 Mieszanie obrazów z określonym współczynnikiem
- 3.6 Potęgowanie obrazu (zadaną potęgą)
- 3.7 Dzielenie obrazu przez (zadaną) liczbę
- 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 (określonej) stałej z obrazem
- 4.2 Sumowanie dwóch obrazów
- 4.3 Mnożenie obrazu przez zadaną liczbę
- 4.4 Mnożenie obrazu przez inny obraz
- 4.5 Mieszanie obrazów z określonym współczynnikiem
- 4.6 Potęgowanie obrazu (zadaną potęgą)
- 4.7 Dzielenie obrazu przez (zadaną) liczbę
- 4.8 Dzielenie obrazu przez inny obraz
- 4.9 Pierwiastkowanie obrazu
- 4.10 Logarytmowanie obrazu

24 ROZDZIAŁ 4. OPERACJE SUMOWANIA ARYTMETYCZNEGO OBRAZÓW BARWOWYCH

Rozdział 5

Operacje geometryczne na obrazie

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

5.1 Przemieszczenie obrazu o zadany wektor

Opis

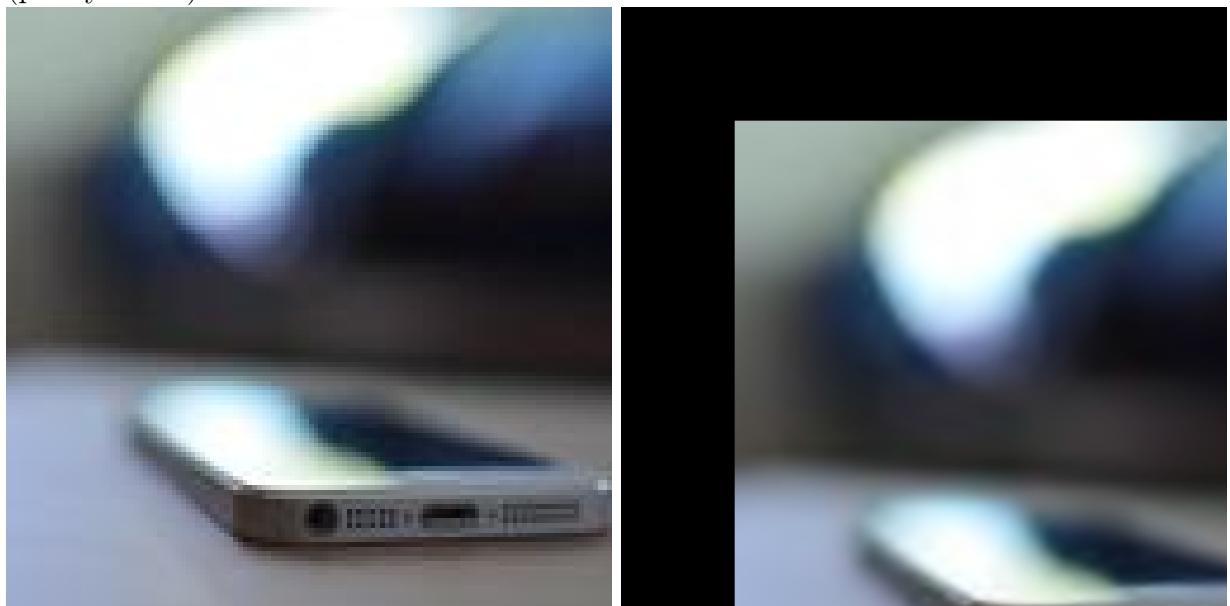
Operacja translacji wykonuje transformację geometryczną polegającą na przeniesieniu każdego z punktów obrazu wejściowego w nowe miejsce na obrazie wynikowym. Pod wpływem translacji element obrazu zlokalizowany na (x_1, y_1) zostanie przesunięty na nową pozycję (x_2, y_2) . Różnicą pomiędzy (x_1, y_1)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

Operacja przemieszczenia przybiera postać:

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

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

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



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



Kod źródłowy algorytmu

```
def translate(self, deltaX = 0, deltaY = 0):
    print('translation start')
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels24Bits()
    result = numpy.zeros((height, width, 3), numpy.uint8)

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

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

5.2 Jednorodne skalowanie obrazu

Opis

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

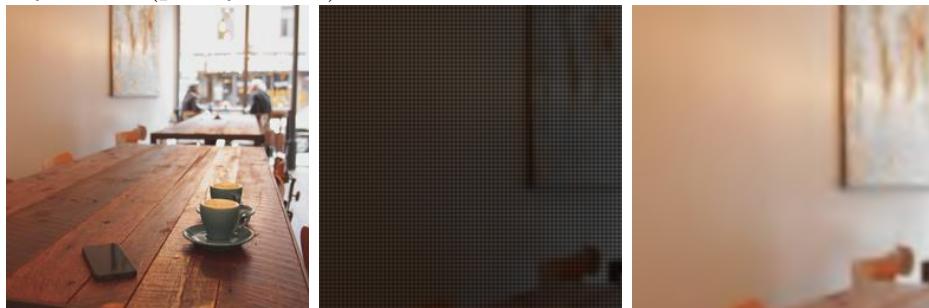
$$x_2 = S_x * x_1 \quad (5.3)$$

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

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

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

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



Kod źródłowy algorytmu

```

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

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

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

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

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

```

5.3 Niejednorodne skalowanie obrazu

Opis

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

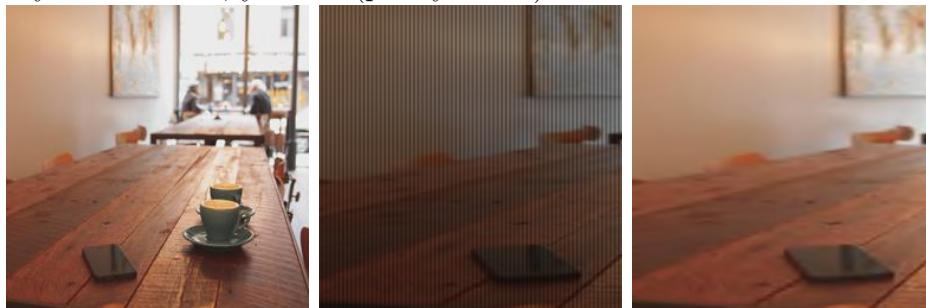
$$x_2 = S_x * x_1 \quad (5.6)$$

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

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

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

Rysunek 5.4: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik $x = 2.0$, $y = 1.0$ (prawy obraz)



Kod źródłowy algorytmu

```
def nonUniformScaling(self, scaleX = 1.0, scaleY = 1.0):
    print('non-uniform scaling start')
    image = self.decoder.getPixels24Bits()

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

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

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

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

```

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

```

5.4 Obracanie obrazu o dowolny kąt

Opis

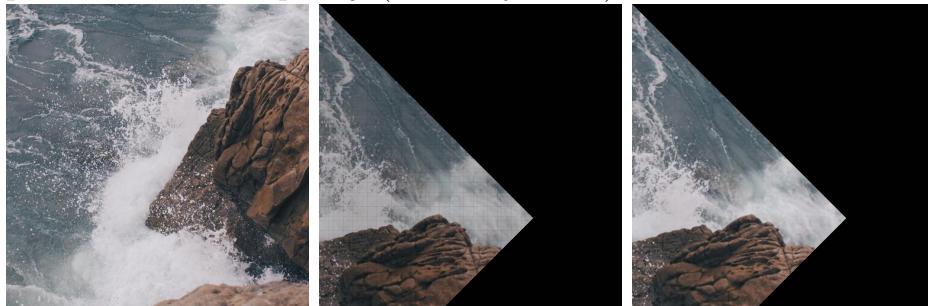
Operacja obrotu wykonywana jest wokół początku układu współrzędnych o kąt φ w taki sposób aby odległość od początku układu do punktu pozostała bez zmian, oraz aby pomiędzy odcinkami danych punków był kąt φ . Właściwości te można pozyskać dzięki wzorom:

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

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

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

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



Kod źródłowy algorytmu

```

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

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

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

```

```

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

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

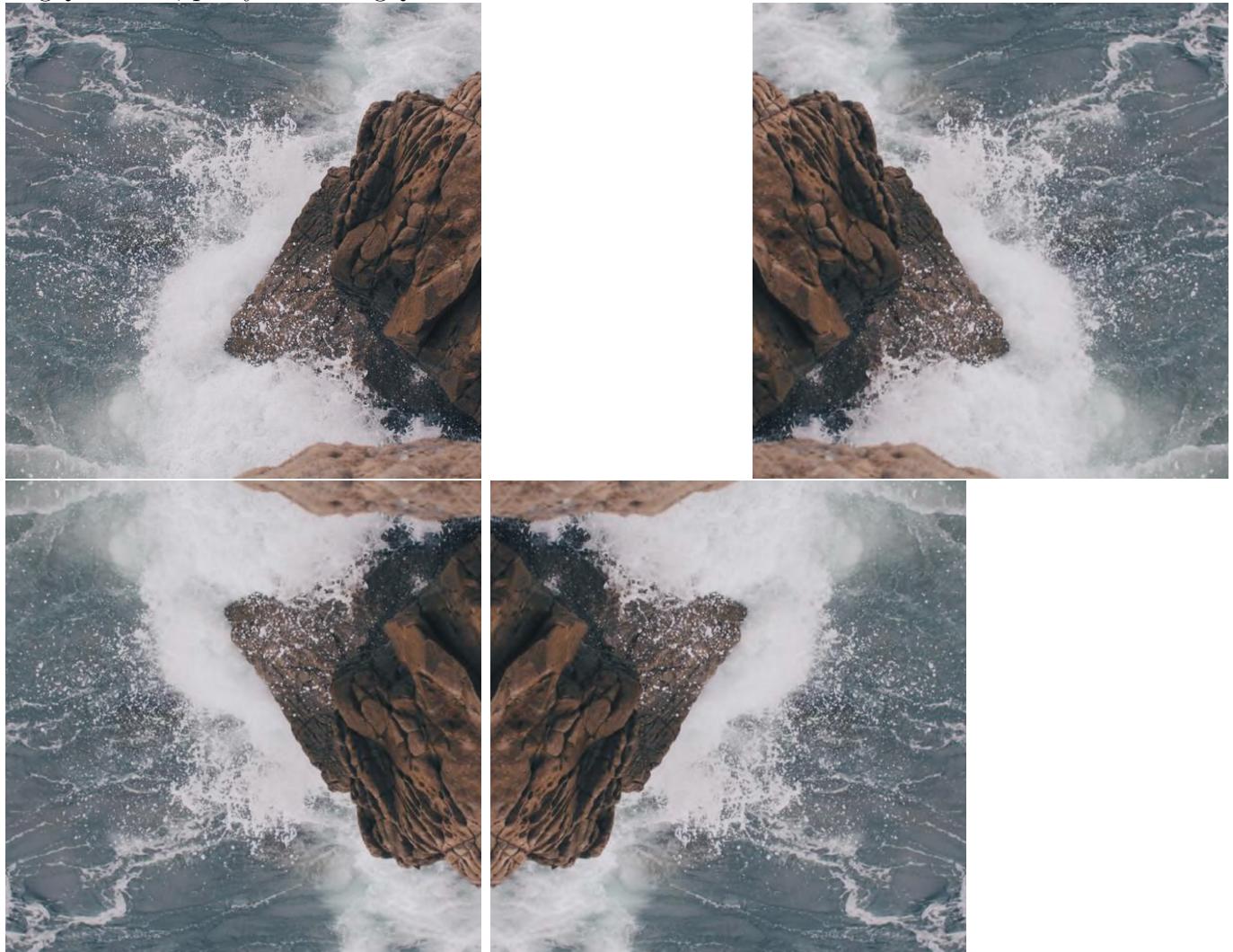
```

5.5 Symetrie względem osi układu

Opis

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

Rysunek 5.6: Od lewej: przed uruchomieniem algorytmu, po symetrii względem OX, po symetrii względem OY, po symetrii względem OX oraz OY



Kod źródłowy algorytmu

```

def axisSymmetry(self, ox, oy):
    print('axis symmetry start')
    image = self.decoder.getPixels24Bits()

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

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

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

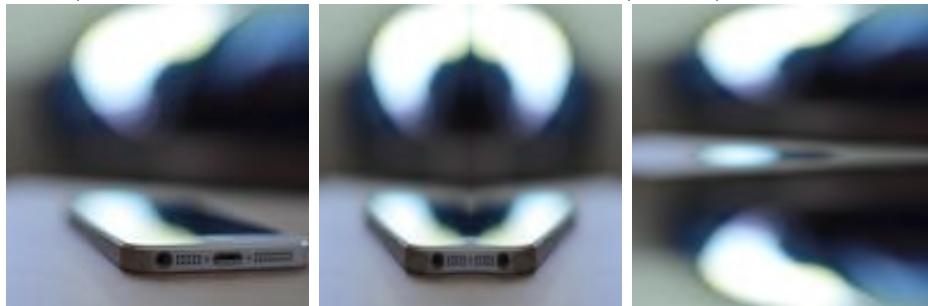
```

5.6 Symetrie względem zadanej prostej

Opis

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

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



Kod źródłowy algorytmu

```

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

    print('symmetry operation X')
    image = self.decoder.getPixels24Bits()
    height, width = self.decoder.height, self.decoder.width
    resultWidth = ox*2
    result = numpy.zeros((height, resultWidth, 3), numpy.uint8)
    for y in range(height):
        for x in range(ox):
            result[y][x] = image[y][x]
            result[y][resultWidth-1-x] = image[y][x]

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

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

def customSymmetryY(self, oy):
    print('custom axis symmetry Y start')
    if not self._validateSymmetryAxisY(oy):
        return

    print('symmetry operation Y')
    image = self.decoder.getPixels24Bits()
    height, width = self.decoder.height, self.decoder.width
    resultHeight = oy*2
    result = numpy.zeros((resultHeight, width, 3), numpy.uint8)

```

```

for y in range(oy):
    for x in range(width):
        result[y][x] = image[y][x]
        result[resultHeight-1-y][x] = image[y][x]

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

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

```

5.7 Wycinanie fragmentów obrazu

Opis

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

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



Kod źródłowy algorytmu

```

def crop(self, (x1, y1), (x2, y2)):
    print('cropping image start')
    image = self.decoder.getBytes24Bits()

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

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

```

5.8 Kopiowanie fragmentów obrazów

Opis

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



Kod źródłowy algorytmu

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

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

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

Rozdział 6

Operacje na histogramie obrazu szarego

6.1 Obliczanie histogramu

6.2 Przemieszczanie histogramu

6.3 Rozciąganie histogramu

6.4 Progowanie lokalne

6.5 Progowanie globalne

Rozdział 7

Operacje na histogramie obrazu barwowego

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

Rozdział 8

Operacje morfologiczne na obrazach binarnych

Obrazy binarne mogą zawierać wiele niedoskonałości. W szczególności podczas tworzeniu obrazu binarnego z obrazu kolorowego (*ang. thresholding*) zdarza się, że na obrazie wynikowym zostanie jakiś szum lub niechciany piksel. Operacje morfologiczne powstały aby umożliwić usuwanie tych niedoskonałości.

8.1 Erozja

8.1.1 Opis

Okrawanie (*ang. erosion*) oznacza **kurczanie** się zbioru czarnych połączonych pikseli co może prowadzić do kurczenia się elementów na obrazie jak i również usuwaniu połączeń pomiędzy elementami czy też wystających części elementu. W tym przykładzie do erozji używamy małego elementu strukturyzującego (*ang. structuring element*) o wielkości 2x2. Użycie większego formatu skutkowałoby wycięciem większej połaci czarnych pikseli.

Rysunek 8.1: Przed uruchomieniem algorytmu (lewy), po uruchomieniu erozji (prawy)



8.1.2 Kod źródłowy algorytmu

```
# Ex7.1
def erosion(self):
    print('erosion start')
    height, width = self.decoder.height, self.decoder.width
```

```

image = self.decoder.getPixels()
result = numpy.zeros((height, width), numpy.uint8)

minY, minX = 1, 1
maxY, maxX = height-1, width-1
for y in range(minY, maxY):
    for x in range(minX, maxX):
        neighbourPixels = [255, 255, 255, 255]

        neighbourPixels[0]=(image[y][x-1])
        neighbourPixels[1]=(image[y-1][x])
        neighbourPixels[2]=(image[y][x+1])
        neighbourPixels[3]=(image[y+1][x])

        if 255 in neighbourPixels:
            result[y][x] = 255
        else:
            result[y][x] = 0

img = Image.fromarray(result, mode='L')
img.save('Resources/morph-erosion.png')
print('erosion done')

```

8.2 Dylatacja

8.2.1 Opis

Nakładanie (*ang. dilation*) oznacza **rozszerzanie** zbioru czarnych połączonych pikseli. Świecznie nadaje się do powiększaniu elementów obrazu jak i przy wypełnianiu luk w grupie białych pikseli.

Rysunek 8.2: Przed uruchomieniem algorytmu (lewy), po dylatacji o element strukturyzujący 2x2 (prawy)



8.2.2 Kod źródłowy algorytmu

```

# Ex7.2
def dilation(self):
    print('dilation start')
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()
    result = numpy.zeros((height, width), numpy.uint8)

```

```

minY, minX = 1, 1
maxY, maxX = height-1, width-1
for y in range(minY, maxY):
    for x in range(minX, maxX):
        neighbourPixels = [255, 255, 255, 255]

        neighbourPixels[0]=(image[y][x-1])
        neighbourPixels[1]=(image[y-1][x])
        neighbourPixels[2]=(image[y][x+1])
        neighbourPixels[3]=(image[y+1][x])

        if 0 in neighbourPixels:
            result[y][x] = 0
        else:
            result[y][x] = 255

img = Image.fromarray(result, mode='L')
img.save('Resources/morph-dilation.png')
print('dilation done')

```

8.3 Otwarcie

8.3.1 Opis

Wiele operacji morfologicznych składa się z kilku prostszych operacji takich jak: erozja, okrawanie lub nawet dopełnienie. Otwarcie jest jedną z takich operacji i jest złożeniem erozji poprzedzającej okrawanie. Nazwa tej operacji wzięła się od następstw jej użycia. Potrafi ona “otwierać” przerwy pomiędzy grupami pikseli, jeśli są one połączone cienkim ”mostem”. Zwiększenie elementu strukturalnego tej operacji spowoduje zwiększenie przerwy pomiędzy obiektami.

Rysunek 8.3: Przed uruchomieniem algorytmu (lewy), po otwarciu”(prawy)



8.3.2 Kod źródłowy algorytmu

```

#Ex7.3
def opening(self):
    print('opening start')
    image = self.decoder.getPixels()

    result = self._erosionOperation(image)

```

```

result = self._dilationOperation(result)

img = Image.fromarray(result, mode='L')
img.save('Resources/morph-opening.png')
print('opening done')

def _dilationOperation(self, image):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width), numpy.uint8)

    minY, minX = 1, 1
    maxY, maxX = height-1, width-1
    for y in range(minY, maxY):
        for x in range(minX, maxX):
            neighbourPixels = [255, 255, 255, 255]

            neighbourPixels[0]=(image[y][x-1])
            neighbourPixels[1]=(image[y-1][x])
            neighbourPixels[2]=(image[y][x+1])
            neighbourPixels[3]=(image[y+1][x])

            if 0 in neighbourPixels:
                result[y][x] = 0
            else:
                result[y][x] = 255
    return result

def _erosionOperation(self, image):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width), numpy.uint8)

    minY, minX = 1, 1
    maxY, maxX = height-1, width-1
    for y in range(minY, maxY):
        for x in range(minX, maxX):
            neighbourPixels = [255, 255, 255, 255]

            neighbourPixels[0]=(image[y][x-1])
            neighbourPixels[1]=(image[y-1][x])
            neighbourPixels[2]=(image[y][x+1])
            neighbourPixels[3]=(image[y+1][x])

            if 255 in neighbourPixels:
                result[y][x] = 255
            else:
                result[y][x] = 0
    return result

```

8.4 Zamknięcie

8.4.1 Opis

Zamknięcie jest również złożeniem kilku operacji - tak jak otwarcie. Lecz w tym wypadku zamieniamy operacje wykorzystywane w otwarciu kolejnością. W ten sposób otrzymamy funkcję, która zamiast dzielić piksele łączy je bez zmiany ogólnego kształtu obiektu. Skuteczne przy "zamykaniu" dziur w obiektach.

Rysunek 8.4: Przed uruchomieniem algorytmu (lewy), po zamknięciu (prawy)



8.4.2 Kod źródłowy algorytmu

```
#Ex7.4
def closing(self):
    print('closing start')
    image = self.decoder.getPixels()

    result = self._dilationOperation(image)
    result = self._erosionOperation(result)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/morph-closing.png')
    print('closing done')

def _dilationOperation(self, image):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width), numpy.uint8)

    minY, minX = 1, 1
    maxY, maxX = height-1, width-1
    for y in range(minY, maxY):
        for x in range(minX, maxX):
            neighbourPixels = [255, 255, 255, 255]

            neighbourPixels[0]=(image[y][x-1])
            neighbourPixels[1]=(image[y-1][x])
            neighbourPixels[2]=(image[y][x+1])
            neighbourPixels[3]=(image[y+1][x])

            if 0 in neighbourPixels:
                result[y][x] = 0
            else:
                result[y][x] = 255
    return result

def _erosionOperation(self, image):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width), numpy.uint8)

    minY, minX = 1, 1
    maxY, maxX = height-1, width-1
    for y in range(minY, maxY):
```

```
for x in range(minX, maxX):
    neighbourPixels = [255, 255, 255, 255]

    neighbourPixels[0]=(image[y][x-1])
    neighbourPixels[1]=(image[y-1][x])
    neighbourPixels[2]=(image[y][x+1])
    neighbourPixels[3]=(image[y+1][x])

    if 255 in neighbourPixels:
        result[y][x] = 255
    else:
        result[y][x] = 0
return result
```

Rozdział 9

Operacje morfologiczne na obrazach szarych

Operacje morfologiczne obrazów szarych to uogólnienie idei operacji morfologicznych na obrazach binarnych. Z zbioru wartości $0, 255$ przechodzimy do zbioru $0, \dots, 255$. Operacje OR i AND następujemy funkcjami Min i Max. Element strukturalny w tym rozdziale został rozbudowany o dwa elementy: o dynamiczną wielkość tablicy (z punktem centralnym w jej środku) pozwalającą na określenie zasięgu naszej operacji, oraz o wartości w elemencie pozwalające na określeniu głębi zmian koloru w obrazie.

9.1 Erozja

9.1.1 Opis

Erozja w obrazach szarych sprowadza się do zwiększenia udziału ciemniejszych kolorów. Na zdjęciach po erozji można zaobserwować dwie zależności. Pierwsza z nich dotyczy wielkości elementu strukturalnego - im większy element tym większy efekt operacji, co można zobaczyć po pogrubieniu czarnych elementów. Natomiast nadanie elementowi wartości sprawi, że średnia wartość pikseli się obniży.

Rysunek 9.1: Obraz niezmieniony (lewy), po erozji z użyciem elementu strukturalnego 3x3 o wartościach 50 (prawy)



Rysunek 9.2: Obraz niezmieniony (lewy), po erozji z użyciem elementu strukturalnego 3×3 o wartościach 100 (prawy)



Rysunek 9.3: Obraz niezmieniony (lewy), po erozji z użyciem elementu strukturalnego 10×10 o wartościach 0 (prawy)



Rysunek 9.4: Obraz niezmieniony (lewy), po erozji z użyciem elementu strukturalnego 10×10 o wartościach 50 (prawy)



9.1.2 Kod źródłowy algorytmu

```
# Ex8.1
def erosion(self, seHeight, seWidth, seDepth):
    print('erosion start')
    image = self.decoder.getPixels()

    structuralElement = numpy.full((seHeight, seWidth), seDepth, numpy.uint8)
    result = self._erosionOperation(image, structuralElement, (4, 4))

    img = Image.fromarray(result, mode='L')
    img.save('Resources/morph-gray-erosion-strel{}x{}-{}.png'.format(str(seHeight),
                                                                     str(seWidth), str(seDepth)))
    print('erosion done')

def _erosionOperation(self, image, structuralElement, elementCenterIndices):
    image32 = image.copy().astype('int32')
    height, width = self.decoder.height, self.decoder.width
    result32 = numpy.zeros((height, width), numpy.int32)
    structuralElement32 = structuralElement.astype(numpy.int32)

    seHeight, seWidth = structuralElement32.shape
    seHalfY, seHalfX = seHeight - 1 - elementCenterIndices[0], seWidth - 1 -
                        elementCenterIndices[1]
    minY, minX = seHalfY, seHalfX
    maxY, maxX = height - (seHeight - minY), width - (seWidth - minX)
    for y in range(minY, maxY):
        for x in range(minX, maxX):
            neighbourPixels = structuralElement32.copy()
            for seY in range(-seHalfY, seHalfY):
                for seX in range(-seHalfX, seHalfX):
                    neighbourPixels[seY][seX] = image32[y+seY][x+seX] - structuralElement32
                                                [seY][seX]
            result32[y][x] = numpy.amin(neighbourPixels)
    result8 = numpy.clip(result32, 0, 255).astype('uint8')
    return result8
```

9.2 Dylatacja

9.2.1 Opis

Okrawanie w wersji obrazów szarych sprowadza się do zmniejszenia udziału czerni. Po dylatacji można zaobserwować jak czarne obiekty ulegają zmniejszeniu i deformacji, oraz jak ogólna aparycja obrazu staje się jaśniejsza. Dzieje się tak z powodu funkcji *max()*, która wybiera przy okrawaniu tylko najjaśniejsze sąsiedztwo piksela, oraz przez wartość elementu strukturalnego, która jest dodawana do wartości pikseli. Wadą większego elementu jest jego skłonność do zatracania szczegółów podczas dylatacji.

Rysunek 9.5: Obraz niezmieniony (lewy), po dylatacji z użyciem elementu strukturalnego 5×5 o wartościach 50 (prawy)



Rysunek 9.6: Obraz niezmieniony (lewy), po dylatacji z użyciem elementu strukturalnego 5×5 o wartościach 100 (prawy)



Rysunek 9.7: Obraz niezmieniony (lewy), po dylatacji z użyciem elementu strukturalnego 10×10 o wartościach 0 (prawy)



Rysunek 9.8: Obraz niezmieniony (lewy), po dylatacji z użyciem elementu strukturalnego 10x10 o wartościach 50 (prawy)



9.2.2 Kod źródłowy algorytmu

```
# Ex8.2
def dilation(self, seHeight, seWidth, seDepth, seCenter=(0, 0)):
    print('dilation start')
    image = self.decoder.getPixels()

    structuralElement = numpy.full((seHeight, seWidth), seDepth, numpy.uint8)
    result = self._dilationOperation(image, structuralElement, seCenter)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/morph-gray-dilation-strel{}x{}-{}.png'.format(str(seHeight),
                                                                     str(seWidth), str(seDepth)))
    print('dilation done')

def _dilationOperation(self, image, structuralElement, elementCenterIndices=(0, 0)):
    image32 = image.copy().astype('int32')
    height, width = self.decoder.height, self.decoder.width
    result32 = numpy.zeros((height, width), numpy.int32)
    structuralElement32 = structuralElement.astype(numpy.int32)

    seHeight, seWidth = structuralElement32.shape
    seHalfY, seHalfX = seHeight-1-elementCenterIndices[0], seWidth-1-
                        elementCenterIndices[1]

    minY, minX = seHalfY, seHalfX
    maxY, maxX = height-(seHeight-minY), width-(seWidth-minX)
    for y in range(minY, maxY):
        for x in range(minX, maxX):
            neighbourPixels = structuralElement32.copy()
            for seY in range(-seHalfY, seHalfY):
                for seX in range(-seHalfX, seHalfX):
                    neighbourPixels[seY][seX] = image32[y+seY][x+seX] + structuralElement32
                                                [seY][seX]
            result32[y][x] = numpy.amax(neighbourPixels)
    result8 = numpy.clip(result32, 0, 255).astype('uint8')
    return result8
```

9.3 Otwarcie

9.3.1 Opis

W obrazie binarnym otwarcie przyczyniało się do tworzenia przerw pomiędzy obiektymi czarni. W wypadku obrazów szarych ta właściwość nie przepada, ale dochodzi jeszcze jedna wynikająca z wartością elementu strukturalnego. Im mniejsza jego wartość tym większy kontrast pomiędzy obiektymi czerni i bieli. Na obrazach wynikowych widać również zatracenie szczegółów w oddali jak i również większą dominację jasno białego koloru.

Rysunek 9.9: Obraz niezmieniony (lewy), po otwarciu z użyciem elementu strukturalnego 5x5 o wartościach 50 (prawy)



Rysunek 9.10: Obraz niezmieniony (lewy), po otwarciu z użyciem elementu strukturalnego 5x5 o wartościach 100 (prawy)



Rysunek 9.11: Obraz niezmieniony (lewy), po otwarciu z użyciem elementu strukturalnego 10x10 o wartościach 50 (prawy)



Rysunek 9.12: Obraz niezmieniony (lewy), po otwarciu z użyciem elementu strukturalnego 10x10 o wartościach 100 (prawy)



9.3.2 Kod źródłowy algorytmu

Funkcje erozji i okrawania są takie same jak w poprzednich sekcjach.

```
#Ex8.3
def opening(self, seHeight, seWidth, seDepth, seCenter=(0, 0)):
    print('opening start')
    image = self.decoder.getPixels()

    structuralElement = numpy.full((seHeight, seWidth), seDepth, numpy.uint8)
    result = self._erosionOperation(image, structuralElement, seCenter)
    result = self._dilationOperation(result, structuralElement, seCenter)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/morph-gray-opening-strel{}x{}.png'.format(str(seHeight),
                                                               str(seWidth), str(seDepth)))
    print('opening done')
```

9.4 Zamknięcie

Rysunek 9.13: Obraz niezmieniony (lewy), po zamknięciu z użyciem elementu strukturalnego 5x5 o wartościach 50 (prawy)



Rysunek 9.14: Obraz niezmieniony (lewy), po zamknięciu z użyciem elementu strukturalnego 5x5 o wartościach 100 (prawy)



Rysunek 9.15: Obraz niezmieniony (lewy), po zamknięciu z użyciem elementu strukturalnego 10x10 o wartościach 50 (prawy)



Rysunek 9.16: Obraz niezmieniony (lewy), po zamknięciu z użyciem elementu strukturalnego 10x10 o wartościach 100 (prawy)



9.4.1 Kod źródłowy algorytmu

Funkcje erozji i okrawania są takie same jak w poprzednich sekcjach.

```
#Ex8.4
def closing(self, seHeight, seWidth, seDepth, seCenter=(0, 0)):
    print('closing start')
    image = self.decoder.getPixels()

    structuralElement = numpy.full((seHeight, seWidth), seDepth, numpy.uint8)
    result = self._dilationOperation(image, structuralElement, seCenter)
    result = self._erosionOperation(result, structuralElement, seCenter)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/morph-gray-closing-strel{}x{}-{}.png'.format(str(seHeight),
                                                                     str(seWidth), str(seDepth)))
    print('closing done')
```


Rozdział 10

Filtrowanie wygładzające liniowe i nieliniowe

Według definicji filtrowanie to technika modyfikująca lub upiększająca obraz. Dla przykładu, można nałożyć filtr na obraz w celu uwypuklenia jego krawędzi w taki sposób w jaki robi to filtr Robertsa. Przy czym filtr Robertsa należy do grupy filtrów liniowych co można poznać po operacjach jakie wykonuje. Głównie będą to proste operacje jak suma liniowa i będą się one wywodzić ze splotu różnych funkcji. W kategoriach filtrów znajdują się jeszcze filtry nieliniowe oparte na bazie kombinacji liniowych i heurystyk. Przykładem takiego filtru może być filtr medianowy.

10.1 Filtr dolnoprzepustowy uśredniający

10.1.1 Opis

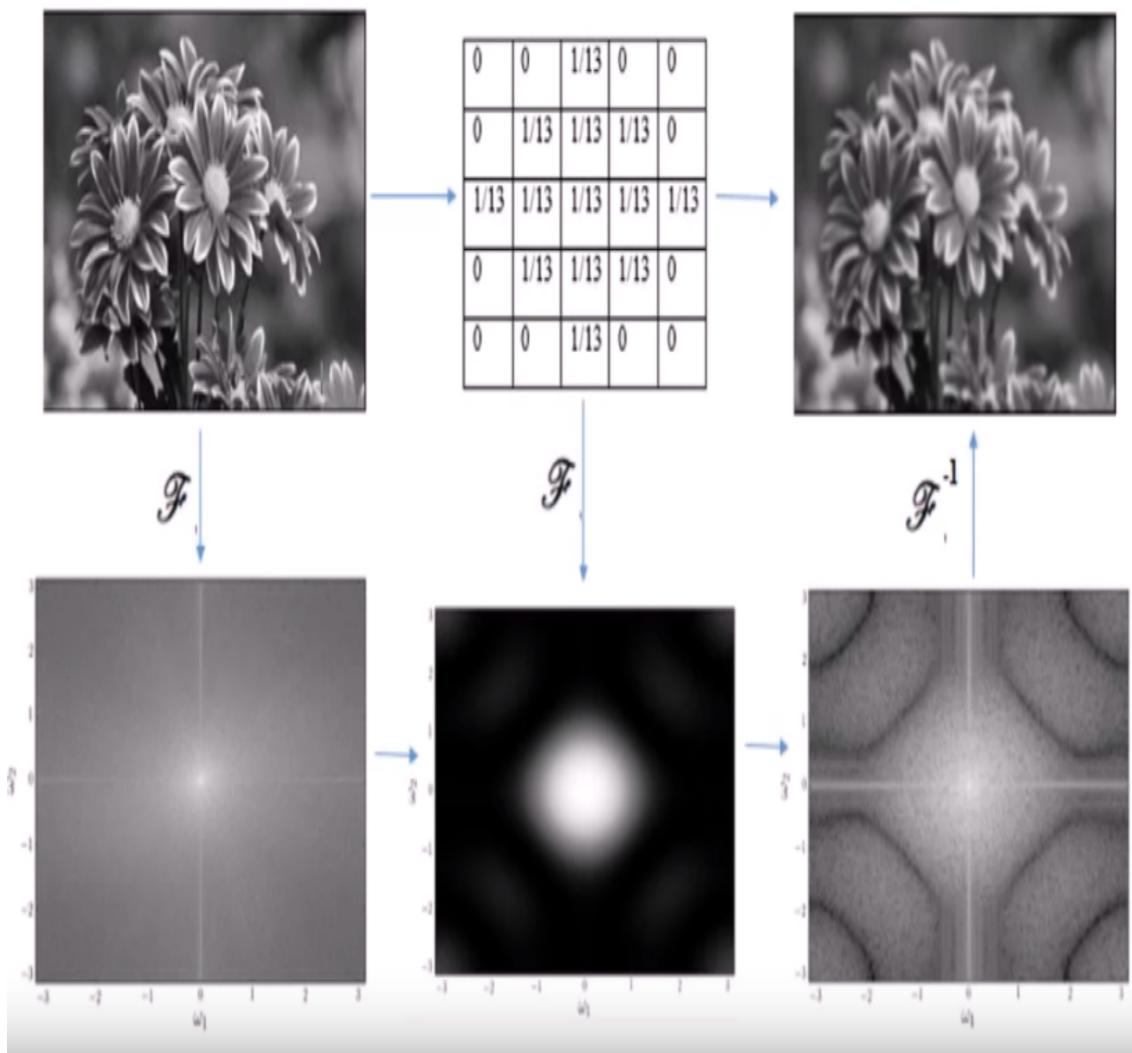
Podstawowy filtr dający efekt rozmazania lub wygładzenia. Filtr oblicza średnią pikseli w elemencie strukturalnym (maska) o wielkości dziewięciu pikseli i przypisuje do piksela w środku. W swojej prostocie filtr ten ma słaby punkt w postaci szumu, który przejawia się w postaci białych, losowych pikseli. Po zastosowaniu filtru biały piksel może być bardziej widoczny niż wcześniej.

Maska tej operacji przedstawia się następująco:

$$\frac{1}{9} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Dzięki niej jesteśmy w stanie symulować zachowanie oryginalnego filtru dolnoprzepustowego z elektroniki, który polegał na odcięciu górnych pasm częstotliwości w sygnale. Jednakże mając obraz w formacie macierzy trudno byłoby nam zgadnąć jaki piksel skrywa jaką wartość częstotliwości. Dlatego w tym temacie pomocny jest algorytm **Szybkiej Transformaty Fouriera** (*ang. Fast Fourier Transform - FFT*). Pozwala on na przeniesienie macierzy pikseli zawierającą wartości kolorów (domena czasu) w domenę częstotliwości.

Rysunek 10.1: Zmiana w częstotliwości obrazu przed i po zastosowaniu pokazanej maski



Rysunek 10.2: Obraz niezmieniony (lewy), po zastosowaniu filtru z maską 9x9 (środkowy), z maską 27x27 (prawy)



10.1.2 Kod źródłowy algorytmu

```
# Ex9.1 - Box Blur
def boxBlur(self, kernelSize=(9,9)):
    print('box blur with size {}x{} start'.format(kernelSize[0], kernelSize[1]))
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()
```

```

result = numpy.empty((height, width), numpy.uint8)
kernel = numpy.ones(kernelSize)

kernelHeight, kernelWidth = kernel.shape
overlapHeight, overlapWidth = int(math.ceil(kernelHeight/2)), int(math.
                                         ceil(kernelWidth/2))

for y in range(height):
    for x in range(width):
        average, hitsCount = 0, 0
        for yOff in range(-overlapHeight, overlapHeight):
            for xOff in range(-overlapWidth, overlapWidth):
                ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0)
                           else (y + yOff)
                xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0)
                           else (x + xOff)
                average += image[ySafe, xSafe] * kernel[yOff + 1, xOff +
                                              1]
                hitsCount += kernel[yOff + 1, xOff + 1]
        average = int(round(average/hitsCount))
        result[y, x] = average

img = Image.fromarray(result, mode='L')
img.save('Resources/filter-boxblur{}x{}.png'.format(kernelSize[0],
                                                       kernelSize[1]))
print('box blur with size {}x{} done'.format(kernelSize[0], kernelSize[1]))
)

```

10.2 Filtr dolnoprzepustowy Gaussowski

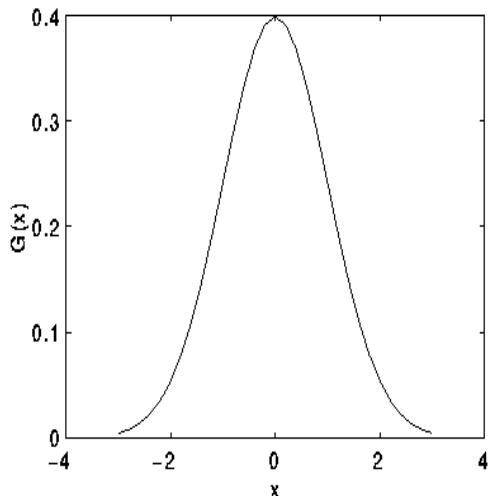
10.2.1 Opis

Kolejnym filtrem dolnoprzepustowym, wygładzającym obraz i usuwającym detale i szum jest filtr oparty o rozkład Gaussa. Ma ona zastosowanie przy produkcji maski dla tego filtra. Wartości maski, w swej idei, układają się w kształcie dzwonu Gaussa (Rys. 10.3), tzn. największa wartość jest pośrodku macierzy, a wartości wokół niej tworzą pierścień składający się z mniejszych wartości. Jeśli wielkość maski na to pozwala kolejny pierścień mniejszych wartości otula poprzedni.

Przykładowe maski stworzone za pomocą funkcji Gaussa:

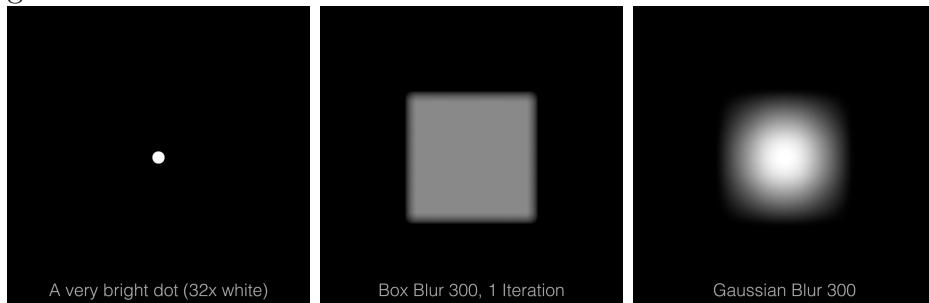
$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \frac{1}{256} * \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Rysunek 10.3: Dzwon Gaussa



Zaletą użycia rozkładu normalnego jest efekt jaki uzyskujemy w porównaniu do filtru uśredniającego. Rozkład normalny daje promieniście rozchodzązącą się maskę, która z kolei nadaje wagę naszym pikselom. Używając wag do obliczania sumy w algorytmie filtra, uzyskujemy PSF (*Point-spread function*) o okrągłym, kolistym kształcie. W porównaniu do poprzedniego rodzaju filtra, który zawsze daje nam rozmycie oparte o kształt prostokąta.

Rysunek 10.4: (Od lewej) Kropka bez rozmycia, rozmycie filtrem uśredniającym, rozmycie filtrem gaussowskim



Rysunek 10.5: Obraz niezmieniony (lewy), po zastosowaniu filtru z maską 9x9 i wartościami sumującymi się do 546 (środkowy), z maską 18x18 i wartościami sumującymi się do 1092 (prawy)



10.2.2 Kod źródłowy algorytmu

```
# Ex9.1 - Gaussian Blur
def gaussianBlur(self, kernelSize=16, kernelFactory=256):
    print('gaussian blur with size {}x{} start'.format(kernelSize, kernelSize))
    height, width = self.decoder.height, self.decoder.width
```

```

image = self.decoder.getPixels()

result = numpy.empty((height, width), numpy.uint8)
kernel = (self.generateGaussianKernel(kernelSize, 1)*kernelFactory).
          astype(numpy.int32)

overlap = int(math.ceil(kernelSize/2))
for y in range(height):
    for x in range(width):
        average, kernelHitsValue = 0, 0
        for yOff in range(-overlap, overlap):
            for xOff in range(-overlap, overlap):
                ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0)
                           else (y + yOff)
                xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0)
                           else (x + xOff)
                average += image[ySafe, xSafe] * kernel[yOff + 1, xOff +
                                                1]
                kernelHitsValue += kernel[yOff + 1, xOff + 1]
        average = int(round(average/kernelHitsValue))
        result[y, x] = average

img = Image.fromarray(result, mode='L')
img.save('Resources/filter-gaussianblur{}x{}.png'.format(kernelSize,
                                                          kernelSize, kernelFactory))
print('gaussian blur with size {}x{} done'.format(kernelSize, kernelSize))
)

def generateGaussianKernel(self, size=3, sigma=1):
    lim = size//2 + (size % 2)/2
    x = numpy.linspace(-lim, lim, size+1)
    kern1d = numpy.diff(st.norm.cdf(x))
    kern2d = numpy.outer(kern1d, kern1d)
    return kern2d/kern2d.sum()

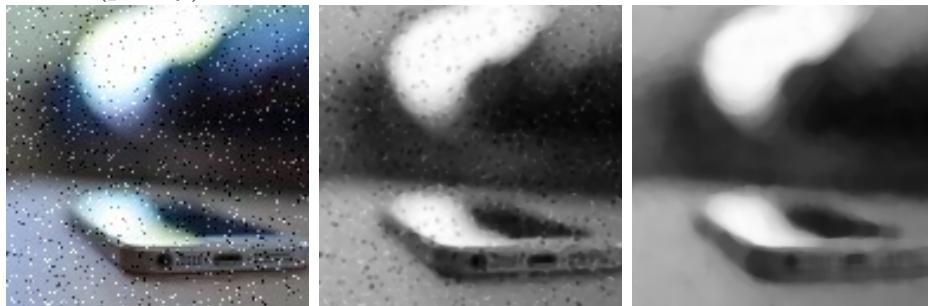
```

10.3 Filtr medianowy

10.3.1 Opis

Filtr medianowy należy do rodziny filtrów nieliniowych, co oznacza że składa się on z kombinacji liniowych bądź opiera się na heurystyce. Po zastosowaniu go na zaszumiony obraz zdumiewamy się jego skutecznością w wale z białymi i/lub czarnymi pikselami. Natomiast po przyjrzeniu się możemy jeszcze dostrzec wyraźne linie krawędzi obiektów na obrazie co wyróżnia go na tle innych filtrów mogących usuwać szумy. Swoje niezwykłe, użyteczne właściwości filtr ten zawdzięcza kwartylowi rzędu 1/2, czyli medianie, która ignoruje wszystkie skrajne wartości w aktualnym oknie podczas wykonywania się algorytmu

Rysunek 10.6: Obraz niezmieniony (lewy), po zastosowaniu filtru z maską 9x9 (środkowy), z maską 27x27 (prawy)



10.3.2 Kod źródłowy algorytmu

```
# Ex9.2
def median(self, squareKernelSize=9):
    print('median filter with size {}x{} start'.format(squareKernelSize,
                                                       squareKernelSize))
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()

    result = numpy.empty((height, width), numpy.uint8)
    kernel = numpy.zeros((squareKernelSize, squareKernelSize), numpy.uint8)

    overlap = int(math.ceil(squareKernelSize/2))
    for y in range(height):
        for x in range(width):
            median = kernel.copy()
            for yOff in range(-overlap, overlap):
                for xOff in range(-overlap, overlap):
                    ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0)
                           else (y + yOff)
                    xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0)
                           else (x + xOff)
                    median[yOff+overlap][xOff+overlap] = image[ySafe][xSafe]
            result[y, x] = numpy.median(median)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/filter-median{}x{}.png'.format(squareKernelSize,
                                                       squareKernelSize))
    print('median filter with size {}x{} done'.format(squareKernelSize,
                                                       squareKernelSize))
```

10.4 Filtr modalny

10.4.1 Opis

Filtr modalny bazuje na prostej statystyce. Podczas wykonywania algorytmu w oknie filtru jest obliczana ilość wystąpień wartości pikseli i na rezultat jest wybierana wartość występująca najczęściej. Jeśli wszystkie wartości występują tą samą ilość razy wtedy rezultatem jest aktualnie rozważany piksel. Filtr modalny produkuje gładkie, bezszumne obrazy. Czasem zdarza się, że obraz wyjściowy zawierał będzie obiekty z nieregularnymi kształtami, których nie było wcześniej.

Rysunek 10.7: Obraz niezmieniony (lewy, górnny), po zastosowaniu filtru z maską 9x9 (prawy, górnny), z maską 18x18 (lewy, dolny), z maską 27x27 (prawy, dolny)



Dobranie odpowiedniej maski do obrazu jest kluczowe, aby filtr spełnił swoje zadanie. Siła tego algorytmu - statystyka - to równocześnie jego słabość. Mając obraz na którym króluje jeden lub kilka kolorów stajemy przed obliczem bezkształtnej jednokolorowej masy tego koloru po wykonaniu algorytmu jeśli źle dobierzemy wielkość maski.

10.4.2 Kod źródłowy algorytmu

```
# Ex9.3
def modalGray(self, squareKernelSize=9):
    print('modal filter with size {}x{} start'.format(squareKernelSize,
                                                       squareKernelSize))
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()

    result = numpy.empty((height, width), numpy.uint8)

    overlap = int(math.ceil(squareKernelSize/2))
    for y in range(height):
        for x in range(width):
            kernel = numpy.zeros((squareKernelSize, squareKernelSize), numpy.
                                  uint8)
            for yOff in range(-overlap, overlap):
                for xOff in range(-overlap, overlap):
```

```

        ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0)
                    ) else (y + yOff)
        )
        xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0)
                    else (x + xOff)
    kernel[yOff+overlap][xOff+overlap] = image[ySafe][xSafe]
result[y, x] = self.mostFrequent(kernel, kernel[overlap][overlap]
)

img = Image.fromarray(result, mode='L')
img.save('Resources/filter-modal{}x{}.png'.format(squareKernelSize,
                                                    squareKernelSize))
print('modal filter with size {}x{} done'.format(squareKernelSize,
                                                    squareKernelSize))

def mostFrequent(self, matrix, defaultValue):
    array1d = numpy.reshape(matrix, matrix.size)
    (values, counts) = numpy.unique(matrix, return_counts=True)
    if self.allEquals(counts):
        return defaultValue
    mostFrequentIndex = numpy.argmax(counts)
    return values[mostFrequentIndex]

def allEquals(self, iterator):
    iterator = iter(iterator)
    try:
        first = next(iterator)
    except StopIteration:
        return True
    return all(first == rest for rest in iterator)

```

10.5 Filtr Kuwahary

10.5.1 Opis

Filtr wygładzający Kuwahary polega na swojej idei na wybraniu jednego z czterech regionów, na które jest podzielona maska. Wybór odbywa się poprzez obliczenie odchylenia standardowego każdego regionu i wybranie tego który ma najmniejsze odchylenie. Każdy region nachodzi na krawędź dwóch sąsiednich. Im większa maska tym mniejsze jest rozmazanie krawędzi powstałe w wyniku usunięcia szumu.

Rysunek 10.8: Przykład maski z zaznaczonymi regionami

<u>X₁</u>	X ₂	X ₃	X ₄	X ₅
X ₆	<u>X₇</u>	X ₈	X ₉	X ₁₀
X ₁₁	<u>X₁₂</u>	<u>X₁₃</u>	X ₁₄	X ₁₅
X ₁₆	X ₁₇	X ₁₈	X ₁₉	X ₂₀
X ₂₁	X ₂₂	<u>X₂₃</u>	X ₂₄	X ₂₅

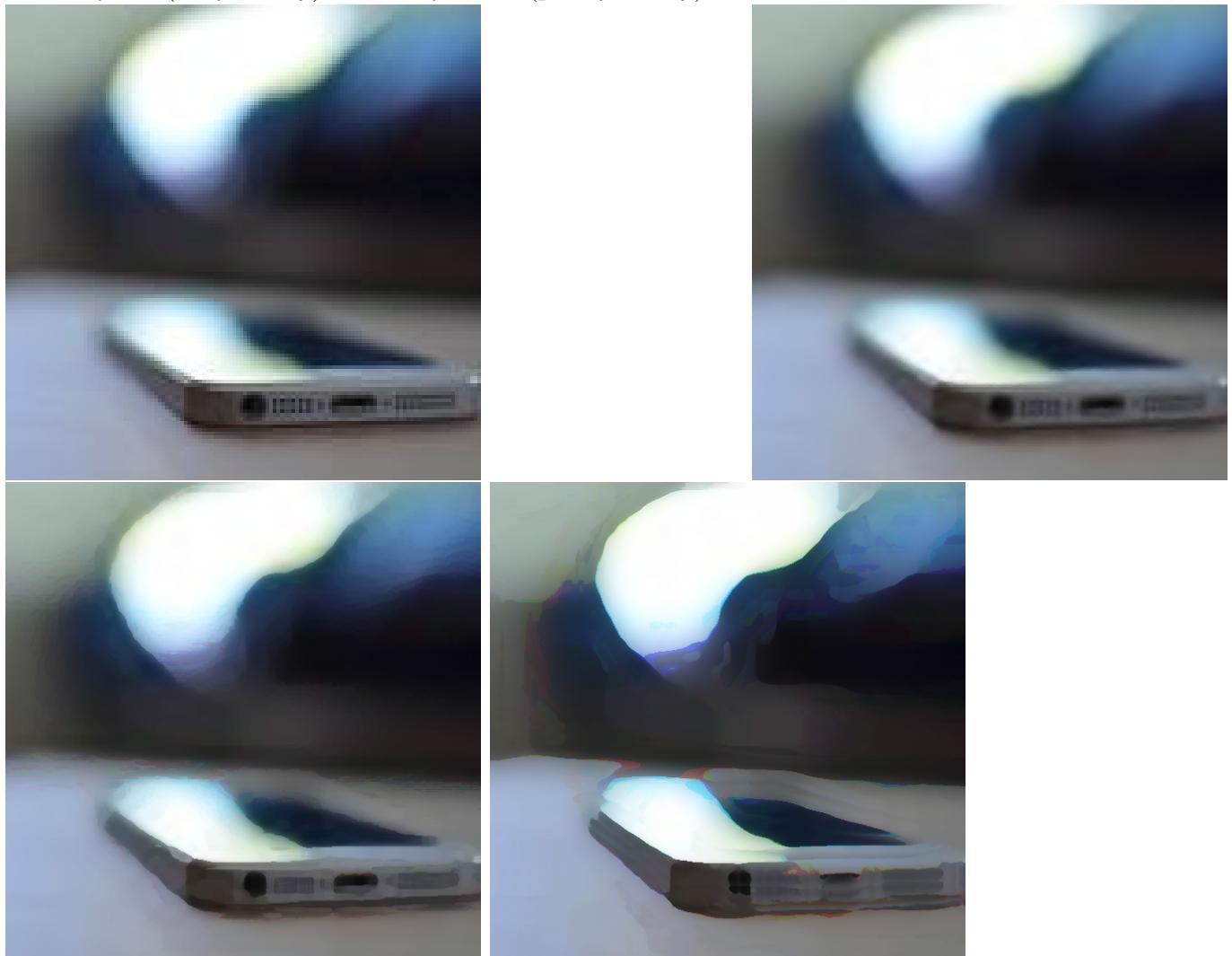
<u>X₁</u>	X ₂	<u>X₃</u>	<u>X₄</u>	X ₅
X ₆	X ₇	<u>X₈</u>	<u>X₉</u>	<u>X₁₀</u>
X ₁₁	X ₁₂	<u>X₁₃</u>	<u>X₁₄</u>	<u>X₁₅</u>
X ₁₆	X ₁₇	X ₁₈	<u>X₁₉</u>	X ₂₀
X ₂₁	X ₂₂	<u>X₂₃</u>	X ₂₄	X ₂₅

X ₁	X ₂	X ₃	X ₄	X ₅
X ₆	X ₇	X ₈	X ₉	X ₁₀
X ₁₁	X ₁₂	<u>X₁₃</u>	<u>X₁₄</u>	X ₁₅
X ₁₆	<u>X₁₇</u>	<u>X₁₈</u>	X ₁₉	X ₂₀
X ₂₁	X ₂₂	<u>X₂₃</u>	X ₂₄	X ₂₅

Rysunek 10.9: Obraz niezmieniony (lewy, górnny), po zastosowaniu filtru z maską 3x3 (prawy, górnny), z maską 9x9 (lewy, dolny), z maską 27x27 (prawy, dolny)



Rysunek 10.10: Obraz niezmieniony (lewy, górnny), po zastosowaniu filtru z maską 3x3 (prawy, górnny), z maską 9x9 (lewy, dolny), z maską 27x27 (prawy, dolny)



10.5.2 Kod źródłowy algorytmu

```
# Ex9.4 - Gray
def kuwaharaGray(self, squareKernelSize=3):
    print('kuwahara gray filtering with size {}x{} start'.format(
        squareKernelSize,
        squareKernelSize))
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels()
    result = numpy.empty((height, width), numpy.uint8)

    commonAxis = int(math.ceil(squareKernelSize/2))
    for y in range(height):
        for x in range(width):
            kernel = numpy.zeros((squareKernelSize, squareKernelSize), numpy.
                uint8)
            for yOff in range(-commonAxis, commonAxis+1):
                for xOff in range(-commonAxis, commonAxis+1):
                    ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0
                        ) else (y + yOff)
                    xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0
                        ) else (x + xOff)
                    kernel[yOff+commonAxis][xOff+commonAxis] = image[ySafe][
                        xSafe]
```

```

        result[y, x] = self.findLowestSDRegion(kernel, commonAxis)

    img = Image.fromarray(result, mode='L')
    img.save('Resources/filter-kuwahara{}x{}.png'.format(squareKernelSize,
                                                          squareKernelSize))
    print('kuwahara gray filtering with size {}x{} done'.format(
          squareKernelSize,
          squareKernelSize))

# Ex9.4 - Color
def kuwaharaColor(self, squareKernelSize=3):
    print('kuwahara color filtering with size {}x{} start'.format(
          squareKernelSize,
          squareKernelSize))

    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels24Bits()
    result = numpy.empty((height, width, 3), numpy.uint8)

    commonAxis = int(math.ceil(squareKernelSize/2))
    for y in range(height):
        for x in range(width):
            kernel = numpy.zeros((squareKernelSize, squareKernelSize, 3),
                                 numpy.uint8)
            for yOff in range(-commonAxis, commonAxis+1):
                for xOff in range(-commonAxis, commonAxis+1):
                    ySafe = y if ((y + yOff) > (height - 1) or (y + yOff) < 0)
                           else (y + yOff)
                    xSafe = x if ((x + xOff) > (width - 1) or (x + xOff) < 0)
                           else (x + xOff)
                    kernel[yOff+commonAxis][xOff+commonAxis] = image[ySafe][
                        xSafe]
            result[y, x, 0] = self.findLowestSDRegion(kernel[:, :, 0],
                                                       commonAxis)
            result[y, x, 1] = self.findLowestSDRegion(kernel[:, :, 1],
                                                       commonAxis)
            result[y, x, 2] = self.findLowestSDRegion(kernel[:, :, 2],
                                                       commonAxis)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/filter-kuwahara-color{}x{}.png'.format(
          squareKernelSize,
          squareKernelSize))
    print('kuwahara color filtering with size {}x{} done'.format(
          squareKernelSize,
          squareKernelSize))

def findLowestSDRegion(self, kernel, commonAxis):
    length = kernel.shape[0]
    upperLeftRegion = kernel[0:commonAxis+1, 0:commonAxis+1]
    upperRightRegion = kernel[commonAxis:length, 0:commonAxis+1]
    lowerLeftRegion = kernel[0:commonAxis+1, commonAxis:length]
    lowerRightRegion = kernel[commonAxis:length, commonAxis:length]
    standardDeviation = [numpy.std(upperLeftRegion),
                          numpy.std(upperRightRegion),
                          numpy.std(lowerLeftRegion),
                          numpy.std(lowerRightRegion)]
    minSD = numpy.min(standardDeviation)
    if minSD == standardDeviation[0]:
        return upperLeftRegion.mean()
    elif minSD == standardDeviation[1]:
        return upperRightRegion.mean()
    elif minSD == standardDeviation[2]:

```

```
        return lowerLeftRegion.mean()
else:
    return lowerRightRegion.mean()
```

10.6 Filtr minimalny

10.7 Filtr maksymalny