

UNIWERSYTET KARDYNAŁA STEFANA WYSZYŃSKIEGO
W WARSZAWIE

WYDZIAŁ MATEMATYCZNO-PRZYRODNICZY
SZKOŁA NAUK ŚCISŁYCH

Katarzyna Mitrus

Michał Słotwiński

Wprowadzenie do Przetwarzania Obrazów

Sprawozdanie z laboratorium

Prowadzący:
prof. Wojciech Mokrzycki

Warszawa, 2018

Spis treści

Spis rysunków	4
Rozdział 1. Wstęp	10
1.1 Specyfikacja wykorzystanego fortformatu obrazu	10
1.2 Instrukcja obsługi programu	11
Rozdział 2. Operacje ujednoliciania obrazów	12
2.1 Ujednolicenie obrazów szarych geometryczne	13
2.2 Ujednolicenie obrazów szarych rozdzielczościowe	16
2.3 Ujednolicenie obrazów RGB geometryczne	20
2.4 Ujednolicenie obrazów RGB rozdzielczościowe	23
Rozdział 3. Operacje sumowania arytmetycznego obrazów szarych	27
3.1 Sumowanie (określonej) stałej z obrazem	27
3.2 Sumowanie dwóch obrazów	30
3.3 Mnożenie obrazu przez zadaną liczbę	33
3.4 Mnożenie obrazu przez inny obraz	35
3.5 Mieszanie obrazów z określonym współczynnikiem	38
3.6 Potęgowanie obrazu (z zadaną potęgą)	41
3.7 Dzielenie obrazu przez (zadaną) liczbę	43
3.8 Dzielenie obrazu przez przez inny obraz	45
3.9 Pierwiastkowanie obrazu	48
3.10 Logarytmowanie obrazu	51
Rozdział 4. Operacje sumowania arytmetycznego obrazów barwowych	53
4.1 Sumowanie (określonej) stałej z obrazem	53
4.2 Sumowanie dwóch obrazów	56
4.3 Mnożenie obrazu przez zadaną liczbę	60
4.4 Mnożenie obrazu przez inny obraz	63
4.5 Mieszanie obrazów z określonym współczynnikiem	67
4.6 Potęgowanie obrazu	70
4.7 Dzielenie obrazu przez (zadaną) liczbę	73
4.8 Dzielenie obrazu przez inny obraz	76
4.9 Pierwiastkowanie obrazu	80
4.10 Logarytmowanie obrazu	83
Rozdział 5. Operacje geometryczne na obrazie	86
5.1 Przesunięcie obrazu o zadany wektor	86
5.2 Jednorodne skalowanie obrazu	88

5.3 Niejednorodne skalowanie obrazu	90
5.4 Obracanie obrazu o dowolny kąt	92
5.5 Symetrie względem osi układu	95
5.6 Symetrie względem zadanej prostej	97
5.7 Wycinanie fragmentów obrazu	100
5.8 Kopiowanie fragmentów obrazów	102
Rozdział 6. Operacje na histogramie obrazu szarego	104
6.1 Obliczanie histogramu	105
6.2 Przemieszczanie histogramu	107
6.3 Rozciąganie histogramu	110
6.4 Progowanie lokalne	113
6.5 Progowanie globalne	116
Rozdział 7. Operacje na histogramie obrazu barwowego	119
7.1 Obliczanie histogramu	120
7.2 Przemieszczanie histogramu	122
7.3 Rozciąganie histogramu	125
7.4 Progowanie 1-progowe lokalne	128
7.5 Progowanie wielo-progowe lokalne	131
7.6 Progowanie 1-progowe globalne	135
7.7 Progowanie wielo-progowe globalne	138
Rozdział 8. Operacje morfologiczne na obrazach binarnych	141
8.1 Okrawanie (erozja)	141
8.2 Nakładanie (dylatacja)	144
8.3 Otwarcie	146
8.4 Zamknięcie	148
Rozdział 9. Operacje morfologiczne na obrazach szarych	151
9.1 Okrawanie (erozja)	151
9.2 Nakładanie (dylatacja)	153
9.3 Otwarcie	155
9.4 Zamknięcie	157
Rozdział 10. Filtrowanie liniowe i nieliniowe	159
10.1 Filtr dolnoprzepustowy uśredniający	160
10.2 Filtr dolnoprzepustowy Gaussowski	164
10.3 Operator Roberts'a	168
10.4 Operator Prewitt'a	172
10.5 Operator Sobel'a	176
10.6 Filtr kompasowy	181
10.7 Gradient wektora kierunkowego	189
10.8 Filtr medianowy	195
10.9 Filtr maksymalny	199
10.10 Filtr minimalny	203
10.11 Filtr płaskorzeźbowy	207
Bibliografia	212

Spis rysunków

2.1	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)	13
2.2	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	13
2.3	Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)	14
2.4	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	14
2.5	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	16
2.6	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	16
2.7	Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	17
2.8	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	17
2.9	Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)	20
2.10	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	20
2.11	Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)	21
2.12	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	21
2.13	Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	23
2.14	Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)	23
2.15	Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	24
2.16	Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)	24
3.1	(Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji	27
3.2	(Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji	28
3.3	(Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji	30
3.4	(Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji	31
3.5	(Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=50, obraz po normalizacji	33
3.6	(Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=100, obraz po normalizacji	33
3.7	(Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji	35
3.8	(Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstał w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji	36
3.9	(Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.5$, poniżej obraz wynikowy po normalizacji	38
3.10	(Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$, poniżej obraz wynikowy po normalizacji	39

3.11 (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=2$, obraz po normalizacji	41
3.12 (Od lewej) Szary obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=3$, obraz po normalizacji	41
3.13 (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji	43
3.14 (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji	43
3.15 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji	45
3.16 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji	46
3.17 (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ($\alpha=1/2$), obraz po normalizacji	48
3.18 (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ($\alpha=1/3$), obraz po normalizacji	48
3.19 (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji	51
3.20 (Od lewej) Szary obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji	51
 4.1 (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji	53
4.2 (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji	54
4.3 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji	56
4.4 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji	57
4.5 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 50, obraz po normalizacji	60
4.6 (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 100, obraz po normalizacji	60
4.7 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji	63
4.8 (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji	64
4.9 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.5$, poniżej obraz wynikowy po normalizacji	67
4.10 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$, poniżej obraz wynikowy po normalizacji	68
4.11 (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=2$, obraz po normalizacji	70
4.12 (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=3$, obraz po normalizacji	70

4.13 (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji	73
4.14 (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji	73
4.15 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji	76
4.16 (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji	77
4.17 (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ($\alpha=1/2$), obraz po normalizacji	80
4.18 (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ($\alpha=1/3$), obraz po normalizacji	80
4.19 (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji	83
4.20 (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji	83
 5.1 (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [40, 70]	86
5.2 (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [200, 100]	87
5.3 (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem $S=1.5$, obraz po interpolacji	88
5.4 (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem $S=2$, obraz po interpolacji	88
5.5 (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem $S_x=2$ oraz współczynnikiem $S_y=1$, obraz po interpolacji	90
5.6 (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem $S_x=1$ oraz współczynnikiem $S_y=2$, obraz po interpolacji	90
5.7 (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 40 stopni, obraz po interpolacji	92
5.8 (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 110 stopni, obraz po interpolacji	92
5.9 (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X	95
5.10 (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X	95
5.11 (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y	96
5.12 (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y	96
5.13 (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu	97
5.14 (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu	97
5.15 (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu	97
5.16 (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu	98
5.17 (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$	100

5.18 (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$	100
5.19 (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$, Skopiowany fragment (151x426)	102
5.20 (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$, Skopiowany fragment (201x201)	102
6.1 Obraz szary, histogram szarości tego obrazu	105
6.2 Obraz szary, histogram szarości tego obrazu	105
6.3 Obraz szary wejściowy, histogram szarości tego obrazu	107
6.5 Obraz szary wejściowy, histogram szarości tego obrazu	108
6.4 Obraz szary przesunięty o 100, histogram szarości tego obrazu	108
6.6 Obraz szary przesunięty o -100, histogram szarości tego obrazu	109
6.7 Obraz wejściowy, histogram szarości tego obrazu	110
6.8 Obraz po rozciągnięciu, histogram szarości tego obrazu	110
6.9 Obraz wejściowy, histogram szarości tego obrazu	111
6.10 Obraz po rozciągnięciu, histogram szarości tego obrazu	111
6.11 Obraz wejściowy, histogram szarości tego obrazu	113
6.12 Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu	113
6.13 Obraz wejściowy, histogram szarości tego obrazu	114
6.14 Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu	114
6.15 Obraz wejściowy, histogram szarości tego obrazu	116
6.16 Obraz po progowaniu globalnym, histogram szarości tego obrazu	117
6.17 Obraz wejściowy, histogram szarości tego obrazu	117
6.18 Obraz po progowaniu globalnym, histogram szarości tego obrazu	118
7.1 Obraz barwny, histogram barw tego obrazu	120
7.2 Obraz barwny, histogram barw tego obrazu	120
7.3 Obraz wejściowy, histogram barw tego obrazu	122
7.4 Obraz wyjściowy przesunięty o 50, histogram barw tego obrazu	122
7.5 Obraz wejściowy, histogram barw tego obrazu	123
7.6 Obraz wyjściowy przesunięty o -50, histogram barw tego obrazu	123
7.7 Obraz wejściowy, histogram barw tego obrazu	125
7.8 Obraz po rozciągnięciu, histogram barw tego obrazu	126
7.9 Obraz wejściowy, histogram barw tego obrazu	126
7.10 Obraz po rozciągnięciu, histogram barw tego obrazu	126
7.11 Obraz wejściowy, histogram szarości tego obrazu	128
7.12 Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu	129
7.13 Obraz wejściowy, histogram szarości tego obrazu	129
7.14 Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu	129
7.15 Obraz wejściowy, histogram barw tego obrazu	131
7.16 Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu	132
7.17 Obraz wejściowy, histogram barw tego obrazu	132
7.18 Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu	132
7.19 Obraz wejściowy, histogram barw tego obrazu	135

7.20	Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu	135
7.21	Obraz wejściowy, histogram barw tego obrazu	136
7.22	Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu	136
7.23	Obraz wejściowy, histogram barw tego obrazu	138
7.24	Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu	139
7.25	Obraz wejściowy, histogram barw tego obrazu	139
7.26	Obraz po progowaniu wielo-progowym globalnym (progi 4), histogram barw tego obrazu	139
8.1	(Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)	142
8.2	(Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)	142
8.3	(Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)	144
8.4	(Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)	144
8.5	(Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)	146
8.6	(Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)	146
8.7	(Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)	148
8.8	(Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)	148
9.1	(Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji okrawania (erozji)	151
9.2	(Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji okrawania (erozji)	152
9.3	(Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji nakładania (dylatacji)	153
9.4	(Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji nakładania (dylatacji)	153
9.5	(Od lewej) Obraz wejściowy (256x256), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)	155
9.6	(Od lewej) Obraz wejściowy (512x512), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)	155
9.7	(Od lewej) Obraz wejściowy (256x256), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)	157
9.8	(Od lewej) Obraz wejściowy (512x512), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)	157
10.1	Obraz wejściowy, obraz uśredniony	160
10.2	Obraz wejściowy, obraz uśredniony	160
10.3	Obraz wejściowy, obraz uśredniony	161
10.4	Obraz wejściowy, obraz uśredniony	161
10.5	Obraz wejściowy, obraz po filtracji filtrem Gaussa	164
10.6	Obraz wejściowy, obraz po filtracji filtrem Gaussa	165
10.7	Obraz wejściowy, obraz po filtracji filtrem Gaussa	165

10.8 Obraz wejściowy, obraz po filtracji filtrem Gaussa	165
10.9 Obraz wejściowy, obraz po filtracji filtrem Roberts'a	168
10.10 Obraz wejściowy, obraz po filtracji filtrem Roberts'a	169
10.11 Obraz wejściowy, obraz po filtracji filtrem Roberts'a	169
10.12 Obraz wejściowy, obraz po filtracji filtrem Roberts'a	169
10.13 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a	172
10.14 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a	173
10.15 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a	173
10.16 Obraz wejściowy, obraz po filtracji filtrem Prewitt'a	173
10.17 Obraz wejściowy, obraz po filtracji filtrem Sobel'a	176
10.18 Obraz wejściowy, obraz po filtracji filtrem Sobel'a	177
10.19 Obraz wejściowy, obraz po filtracji filtrem Sobel'a	177
10.20 Obraz wejściowy, obraz po filtracji filtrem Sobel'a	177
10.21 Obraz wejściowy, obraz po filtracji filtrem kompasowym	182
10.22 Obraz wejściowy, obraz po filtracji filtrem kompasowym	182
10.23 Obraz wejściowy, obraz po filtracji filtrem kompasowym	182
10.24 Obraz wejściowy, obraz po filtracji filtrem kompasowym	183
10.25 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego	190
10.26 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego	190
10.27 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego	190
10.28 Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego	191
10.29 Obraz wejściowy, obraz po filtracji filtrem medianowym	195
10.30 Obraz wejściowy, obraz po filtracji filtrem medianowym	195
10.31 Obraz wejściowy, obraz po filtracji filtrem medianowym	196
10.32 Obraz wejściowy, obraz po filtracji filtrem medianowym	196
10.33 Obraz wejściowy, obraz po filtracji filtrem maksymalnym	199
10.34 Obraz wejściowy, obraz po filtracji filtrem maksymalnym	199
10.35 Obraz wejściowy, obraz po filtracji filtrem maksymalnym	200
10.36 Obraz wejściowy, obraz po filtracji filtrem maksymalnym	200
10.37 Obraz wejściowy, obraz po filtracji filtrem minimalnym	203
10.38 Obraz wejściowy, obraz po filtracji filtrem minimalnym	203
10.39 Obraz wejściowy, obraz po filtracji filtrem minimalnym	204
10.40 Obraz wejściowy, obraz po filtracji filtrem minimalnym	204
10.41 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym	207
10.42 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym	208
10.43 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym	208
10.44 Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym	208

Rozdział 1

Wstęp

1.1 Specyfikacja wykorzystanego formatu obrazu

Tagged Image File Format (TIFF or TIF) jest formatem pliku komputerowego do przechowywania obrazów grafiki rastrowej (oraz osadzania elementów grafiki wektorowej). Jest rastrowym formatem uniwersalnym, tzn. może być zapisany w trybie kolorów:

- CMYK
- YCbCr
- CIELab
- RGB
- skala szarości
- kolor oparty na indeksowaniu
- kolor oparty na bitmapie z dowolną głębokością bitową

Również można dobrać dowolną rozdzielczość z opcją kanału przezroczystości alfa lub bez. Format TIFF wiele algorytmów kompresji bezstratnej:

- PackBits
- LZW (Lempel-Ziv-Welch)
- CCITT Fax group 3 & 4

Plik TIFF podzielony jest na trzy części:

1. nagłówek pliku obrazowego (Image File Header, IFH)
2. katalog pliku obrazowego (Image File Directory, IFD)
3. część danych obrazu

Plik TIFF może zawierać wiele obrazów. Nagłówek pliku składa się z ośmiu bajtów. Jak sama nazwa wskazuje, format TIFF używa struktur danych noszących nazwę Tag, definiujących cechy zawartych w nim obrazów. Np. dla obrazu 320x240 piksli, szerokość była by oznaczona tagiem "width", a wysokość "height", po którym następuje liczba 320 lub 240. Poniżej znajdują się trzy możliwe formy wewnętrznej struktury danych pliku TIFF (zawierającej trzy obrazy).

header	header	header
IFD 0	IFD 0	Image 0
IFD 1	Image 0	Image 1
IFD n	IFD 1	Image 2
Image 0	Image 1	IFD 0
Image 1	IFD 2	IFD 1
Image n	Image 2	IFD 2

W każdym przykładzie nagłówka pojawia się na początku. W pierwszym przykładzie katalog(IFD) jest zapisany na początku jeden za drugim, taki sposób pozwala na szybki odczyt danych.

W drugim przykładzie po każdym katalogu(IFD) są dane bitmapowe, co jest najczęściej spotykanym rozwiązaniem przy plikach z wieloma obrazami.

W ostatnim przykładzie najpierw są zapisane dane bitmapowe, a następnie katalogi(IFD). Aby sprawdzić, czy obraz jest zapisany w formacie TIFF wystarczy odczytać z nagłówka pierwsze 4 bajty, jeśli mają wartość 49h 49h 00h 2Ah lub 4Dh 4Dh 00h 2Ah, to jest to plik TIFF.

1.2 Intstrukcja obsługi programu

Implementacja zadań została napisana w języku Python3. Wykorzystane biblioteki to numpy, matplotlib oraz Pillow. Aby wystartować program i rozpoczęć otrzymywanie wyników przetwarzania poszczególnych obrazów należy uruchomić plik main.py Przykład uruchomienia programu:

python main.py

Rozdział 2

Operacje ujednolicania obrazów

Operacja ujednolicania obrazów dzieli się na dwa etapy. Pierwszym jest ujednolicenie geometryczne, drugim zaś ujednolicenie rozdzielczościowe. W tym programie ujednolicane są dwa obrazy rastrowe w taki sposób, że mniejszy doprowadzany jest do większego, przez co generowany jest nowy obraz o większe liczbie pikseli niż początkowo. Taki sposób ujednolicania nie powoduje widocznego spadku jakości.

2.1 Ujednolicenie obrazów szarych geometryczne

Opis algorytmu

Operacja geometrycznego ujednolicenia obrazów polega na doprowadzeniu obydwu obrazów do takiej samej liczby wierszy piksli w każdym obrazie i takiej samej liczby kolumn piksli w każdym obrazie.

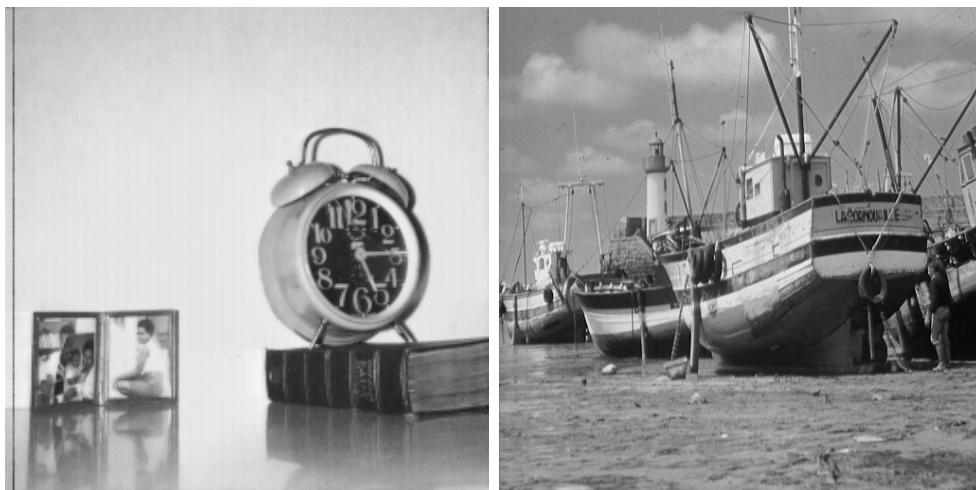
1. Wybierz największą wysokość i największą szerokość z dwóch obrazów.
2. Jeśli dany obraz ma mniejszą szerokość albo wysokość, wypełnij różnicę pikslami o wartości 1 (aby uniknąć dzielenia przez 0).



Rysunek 2.1: Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)



Rysunek 2.2: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.3: Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.4: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.1: Geometryczne ujednolicanie obrazów szarych

```
def geometricGray( self , show = False ) :
    # najwieksza szerokosc sposrod dwuch obrazow
    width1 = self .im1 .shape [1]
    width2 = self .im2 .shape [1]
    maxWidth = width1 if width1 > width2 else width2

    # najwieksza wysokosc sposrod dwuch obrazow
    height1 = self .im1 .shape [0]
    height2 = self .im2 .shape [0]
    maxHeight = height1 if height1 > height2 else height2
```

```

# alokacja pamięci na obrazy wynikowe
resultImage1 = np.empty((maxHeight, maxWidth), dtype = np.
    uint8)
resultImage2 = np.empty((maxHeight, maxWidth), dtype = np.
    uint8)

# współrzędne początku rysowania obrazu 1 w środku
startWidthCoord = int(round((maxWidth - width1) / 2))
startHeightCoord = int(round((maxHeight - height1) / 2))

# wypełnienie obrazu czarny kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage1[i, j] = 1

# narysowanie wysrodkowanego obrazu
for i in range(0, height1):
    for j in range(0, width1):
        resultImage1[i + startHeightCoord, j + startWidthCoord] =
            self.im1[i, j]

# współrzędne początku rysowania obrazu 1 w środku
startWidthCoord = int(round((maxWidth - width2) / 2))
startHeightCoord = int(round((maxHeight - height2) / 2))

# wypełnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage2[i, j] = 1

# narysowanie wysrodkowanego obrazu
for i in range(0, height2):
    for j in range(0, width2):
        resultImage2[i + startHeightCoord, j + startWidthCoord] =
            self.im2[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "L"), Image.
        fromarray(resultImage2, "L"))
    self.save(resultImage1, self.im1Name, "unificationGeo")
    self.save(resultImage2, self.im2Name, "unificationGeo")

```

2.2 Ujednolicenie obrazów szarych rozdzielczościowe

Opis algorytmu

Operacja rozdzielczościowego ujednolicenia obrazów następuje po ujednoliceniu geometrycznym i polega na wypełnieniu obrazu pikslami, a brakujące piksele powinny być 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.



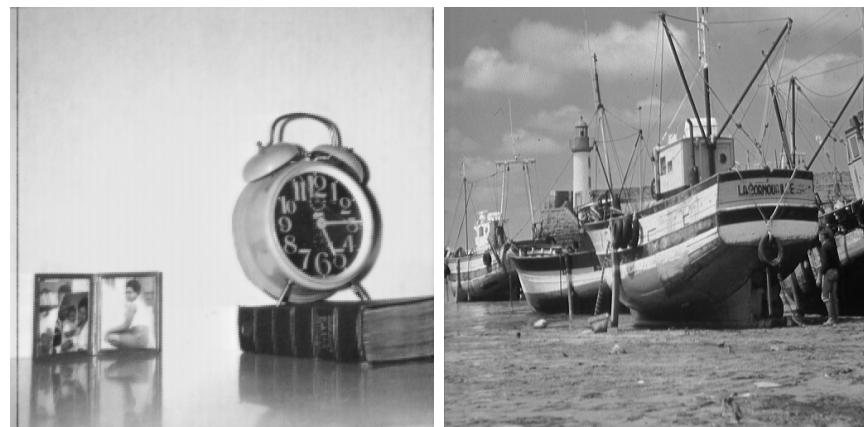
Rysunek 2.5: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.6: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.7: Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)



Rysunek 2.8: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.2: Rastrowe ujednolicianie obrazów szarych

```

def rasterGray( self , show = False ) :
    # pierwszy jest ZAWSZE wiekszy
    width1 = self .im1 .shape [1]
    width2 = self .im2 .shape [1]

    height1 = self .im1 .shape [0]
    height2 = self .im2 .shape [0]

    scaleW = width1 / width2
    scaleH = height1 / height2

    # alokacja pamieci na obrazy wynikowe
    resultImage1 = np .zeros ((height1 , width1) , dtype = np .uint8)

```

```

resultImage2 = np.zeros((height1, width1), dtype = np.uint8)
tmp = np.zeros((height1, width1), dtype = np.uint8)

for i in range(height1):
    for j in range(width1):
        resultImage1[i, j] = self.im1[i, j]

# wypelnianie
count = 0
for i in range(height2):
    for j in range(width2):
        if count == 0:
            resultImage2[int(round(scaleH*i)), int(round(scaleW*j)) + 1] =
                self.im2[i, j]
            count += 1
        if count == 1:
            resultImage2[int(round(scaleH*i)) + 1, int(round(scaleW*j))] =
                self.im2[i, j]
            count = 0

# interpolacja
for i in range(height1):
    for j in range(width1):
        value = 0
        n = 0
        tmp[i, j] = resultImage2[i, j]
        if resultImage2[i, j] < 1:
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height1 - 2)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width1 - 2)) | ((j +
                        jOff) < 0) else (j + jOff)
                    if resultImage2[iSafe, jSafe] > 0:
                        value += resultImage2[iSafe, jSafe]
                        n += 1
                    tmp[i, j] = value / n
                    resultImage2[i, j] = tmp[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "L"), Image.
              fromarray(resultImage2, "L"))
self.save(resultImage1, self.im1Name, "unificationRas")

```

```
self.save(resultImage2, self.im2Name, "unificationRas")
```

2.3 Ujednolicenie obrazów RGB geometryczne

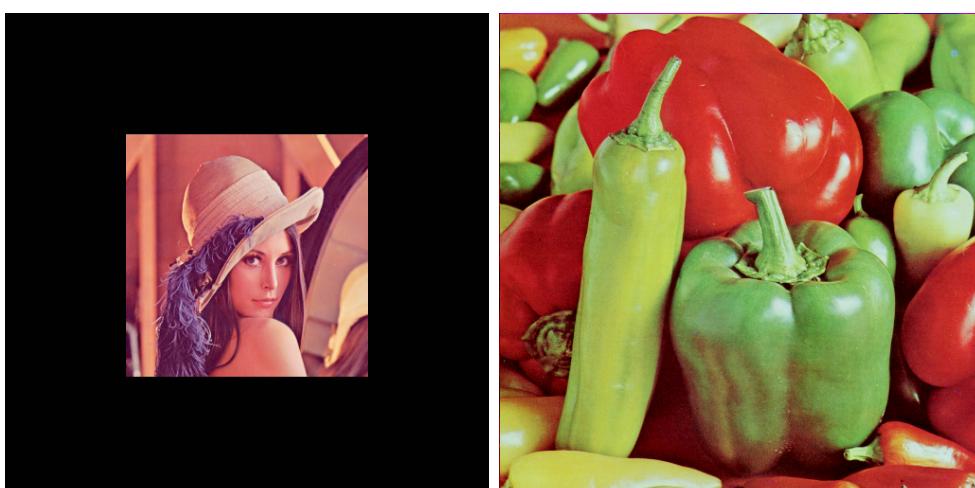
Opis algorytmu

Operacja geometrycznego ujednolicenia obrazów polega na doprowadzeniu obydwu obrazów do takiej samej liczby wierszy pikseli w każdym obrazie i takiej samej liczby kolumn pikseli w każdym obrazie.

1. Wybierz największą wysokość i największą szerokość z dwóch obrazów.
2. Jeśli dany obraz ma mniejszą szerokość albo wysokość, wypełnij różnicę pikslami o wartości 1 dla każdego z kanałów R, G i B (aby uniknąć dzielenia przez 0).



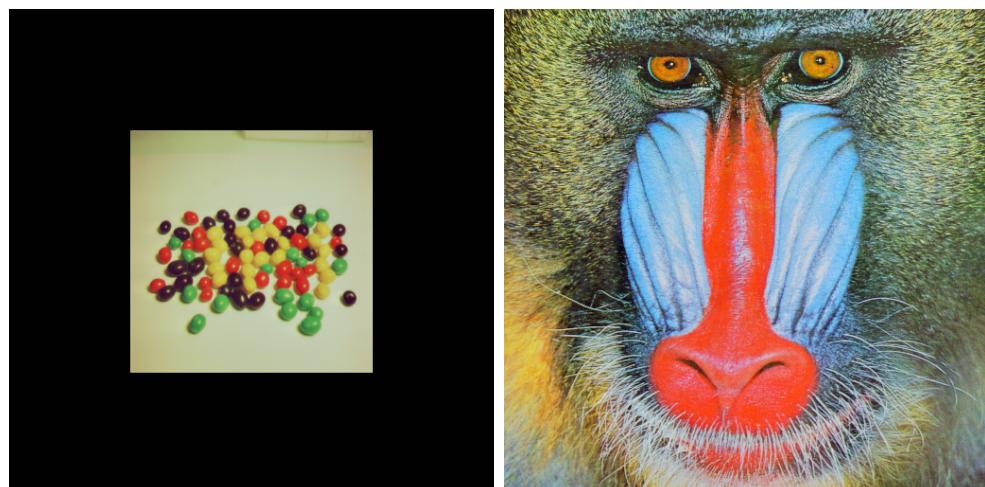
Rysunek 2.9: Obrazy wejściowe (od lewej): obraz 1 (256x256), obraz 2 (512x512)



Rysunek 2.10: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.11: Obrazy wejściowe (od lewej): obraz 3 (256x256), obraz 4 (512x512)



Rysunek 2.12: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.3: Geometryczne ujednolicianie obrazów barwnych

```
def geometricColor(self , show = False):
    # najwieksza szerokosc sposrod dwuch obrazow
    width1 = self.im1.shape[1]
    width2 = self.im2.shape[1]
    maxWidth = width1 if width1 > width2 else width2

    # najwieksza wysokosc sposrod dwuch obrazow
    height1 = self.im1.shape[0]
    height2 = self.im2.shape[0]
    maxHeight = height1 if height1 > height2 else height2
```

```

# alokacja pamieci na obrazy wynikowe
resultImage1 = np.empty((maxHeight, maxWidth, 3), dtype = np.
    uint8)
resultImage2 = np.empty((maxHeight, maxWidth, 3), dtype = np.
    uint8)

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width1) / 2))
startHeightCoord = int(round((maxHeight - height1) / 2))

# wypelnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage1[i, j] = (1, 1, 1)

# narysowanie wysrodkowanego obrazu
for i in range(0, height1):
    for j in range(0, width1):
        resultImage1[i + startHeightCoord, j + startWidthCoord] =
            self.im1[i, j]

# wspolrzedne poczatku rysowania obrazu 1 w srodku
startWidthCoord = int(round((maxWidth - width2) / 2))
startHeightCoord = int(round((maxHeight - height2) / 2))

# wypelnienie obrazu czarnym kolorem
for i in range(0, maxHeight):
    for j in range(0, maxWidth):
        resultImage2[i, j] = (1, 1, 1)

# narysowanie wysrodkowanego obrazu
for i in range(0, height2):
    for j in range(0, width2):
        resultImage2[i + startHeightCoord, j + startWidthCoord] =
            self.im2[i, j]

if show:
    self.show(Image.fromarray(resultImage1, "RGB"), Image.
        fromarray(resultImage2, "RGB"))
    self.save(resultImage1, self.im1Name, "unificationGeo")
    self.save(resultImage2, self.im2Name, "unificationGeo")

```

2.4 Ujednolicenie obrazów RGB rozdzielczościowe

Opis algorytmu

Operacja rozdzielczościowego ujednolicenia obrazów następuje po ujednoliceniu geometrycznym i polega na wypełnieniu obrazu pikslami, a brakujące piksle powinny być zinterpolowane.

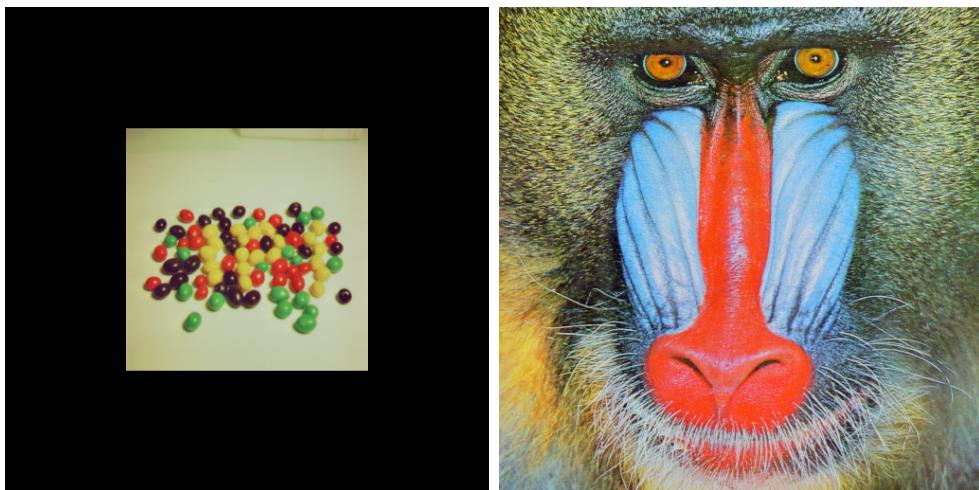
1. Wypełnij cały obraz pikslami o znanej wartości zachowując pewien odstęp między nimi.
2. Każdemu pikselowi o nieznanej wartości przypisz zinterpolowaną wartość (dla każdego z kanałów R, G, B) znanych piksli z jego bezpośredniego otoczenia.



Rysunek 2.13: Obrazy wejściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.14: Obrazy wyjściowe (od lewej): obraz 1 (512x512), obraz 2 (512x512)



Rysunek 2.15: Obrazy wejściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)



Rysunek 2.16: Obrazy wyjściowe (od lewej): obraz 3 (512x512), obraz 4 (512x512)

Listing 2.4: Rastrowe ujednolicanie obrazów barwnych

```
def rasterColor(self, show = False):  
    # pierwszy jest ZAWSZE wiekszy  
    width1 = self.im1.shape[1]  
    width2 = self.im2.shape[1]  
  
    height1 = self.im1.shape[0]  
    height2 = self.im2.shape[0]  
  
    scaleW = width1 / width2  
    scaleH = height1 / height2
```

```

# alokacja pamieci na obrazy wynikowe
resultImage1 = np.zeros((height1, width1, 3), dtype = np.
    uint8)
resultImage2 = np.zeros((height1, width1, 3), dtype = np.
    uint8)
tmp = np.zeros((height1, width1, 3), dtype = np.uint8)

for i in range(height1):
    for j in range(width1):
        resultImage1[i, j] = self.im1[i, j]

# wypelnianie
count = 0
for i in range(height2):
    for j in range(width2):
        if count == 0:
            resultImage2[int(scaleH*i), int(round(scaleW*j)) + 1] =
                self.im2[i, j]
            count += 1
        if count == 1:
            resultImage2[int(round(scaleH*i)) + 1, int(scaleW*j)] =
                self.im2[i, j]
            count = 0

# interpolacja
for i in range(height1):
    for j in range(width1):
        r, g, b = 0, 0, 0
        n = 0
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] <
            1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height1 - 2)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width1 - 2)) | ((j +
                        jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (
                        resultImage2[iSafe, jSafe][1] > 0) | (
                        resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        n += 1
        resultImage1[i, j] = r/n

```

```
g += resultImage2[ iSafe , jSafe ][ 1 ]
b += resultImage2[ iSafe , jSafe ][ 2 ]
n += 1
tmp[ i , j ] = ( r/n , g/n , b/n )
resultImage2[ i , j ] = tmp[ i , j ]

if show:
    self.show( Image.fromarray( resultImage1 , "RGB" ) , Image.
        fromarray( resultImage2 , "RGB" ) )
self.save( resultImage1 , self.im1Name , "unificationRas" )
self.save( resultImage2 , self.im2Name , "unificationRas" )
```

Rozdział 3

Operacje sumowania arytmetycznego obrazów szarych

Arytmetyczne operacje między pikslami p i q dwóch obrazów są używane w wielu dzia-łach przetwarzania obrazów. Przeprowadzane się je wykonując działania na pojedynczych pikslach i są uwarunkowane wymaganiami zależnymi od typu operacji. Po operacjach arytmetycznych zwykle niezbędna jest normalizacja. W przedstawionych zadaniach do normalizacji wykorzystano wzór:

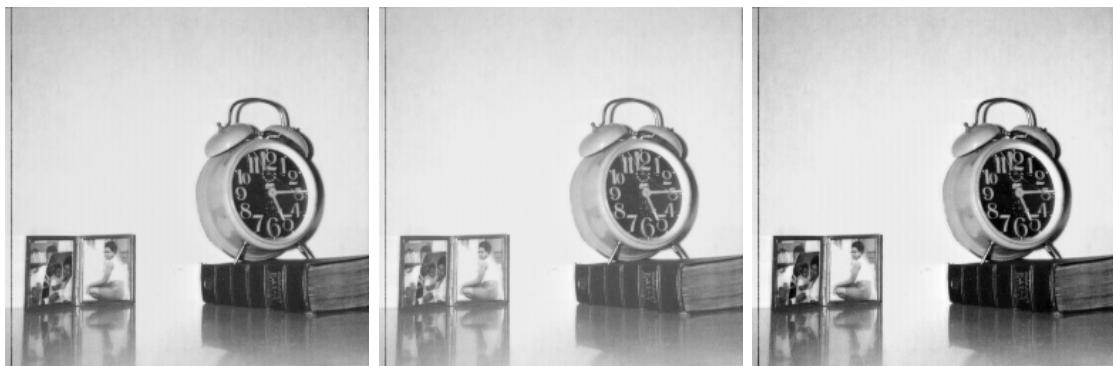
$$f_{norm} = Z_{rep}[(f - f_{min}) / (f_{max} - f_{min})]$$

3.1 Sumowanie (określonej) stałej z obrazem

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

1. Policz sumy wartości kazdego piksla ze stałą (*const*).
2. Jeżeli jedna z tych sum jest większa niż 255 to:
3. Wybierz największą sumę Q_{max} i policz D_{max} ze wzoru: $D_{max}[i, j] = (Q_{max}[i, j] - 255)$
4. Oblicz $X = D_{max}/255$
5. Policz sumy ze wzoru

$$Q[i, j] = P[i, j] - (P[i, j] * X) + const - (const * X)$$



Rysunek 3.1: (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji



Rysunek 3.2: (Od lewej) Szary obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji

Listing 3.1: Sumowanie obrazu szarego ze stałą

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):
        # Obliczanie sumy
        L = int(image_matrix[x][y]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < L:
            Q_max = L

# Sprawdzenie czy przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255)
  
```

```
# Obliczenie sumy z uwzględnieniem zakresu
for y in range(height):
    for x in range(width):
        L = (image_matrix[x][y] - (image_matrix[x][y] * X)) + (
            const - (const * X))

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y] = math.ceil(L)

        # Poszukiwanie minimum i maksimum
        if f_min > L:
            f_min = L
        if f_max < L:
            f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) /
            (f_max - f_min))
```

3.2 Sumowanie dwóch obrazów

Algebraiczne sumowanie obrazów f i f' jest określone jedynie dla obrazów o tych samych wymiarach $M \times N$ i strukturze ich macierzy. Dodawanie obrazów jest użyteczne w uśrednianiu obrazów, wykonywanym w celu zredukowania na nich szumu. 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.

1. Policz sumy wartości kazdego piksla obrazu pierwszego $P1[i,j]$ z piksem obrazu drugiego $P2[i,j]$.
2. Jeżeli jedna z tych sum jest większa niż 255 to:
3. Wybierz największą sumę Q_{max} i policz D_{max} ze wzoru: $D_{max}[i, j] = (Q_{max}[i, j] - 255)$
4. Oblicz $X = D_{max}/255$
5. Policz sumy ze wzoru

$$Q[i, j] = P1[i, j] - (P1[i, j] * X) + P2[i, j] - (P2[i, j] * X)$$



Rysunek 3.3: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.4: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.2: Sumowanie obrazów szarych

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])

```

```

# Poszukiwanie maksimum
if Q_max < L:
    Q_max = L

# Sprawdzenie czy przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255)

# Obliczenie sumy z uwzglednieniem zakresu
for y in range(height):
    for x in range(width):
        L = (image1_matrix [x][y] - (image1_matrix [x][y] * X)) +
            (image2_matrix [x][y] - (image2_matrix [x][y] * X))

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix [x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix [x][y] = 255 * ((result_matrix [x][y] - f_min) /
            (f_max - f_min))

```

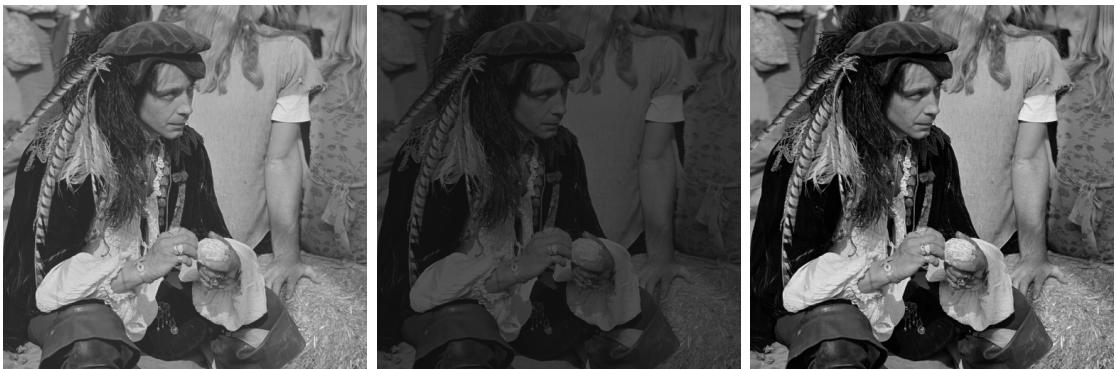
3.3 Mnożenie obrazu przez zadaną liczbę

Mnożenie obrazu f przez skalar wykonuje się mnożąc każdy element obrazu $f_{i,j}$ (wartość funkcji obrazowej piksla) przez ten skalar.

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa piksela $P_1[i, j]$ ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości stalej.
3. W przeciwnym przypadku, jeżeli składowa barwy piksła $P_1[i, j]$ ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 3.5: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=50, obraz po normalizacji



Rysunek 3.6: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę=100, obraz po normalizacji

Listing 3.3: Mnożenie obrazu szarego przez zadaną liczbę

```
iimage1_matrix = self .im1
```

```

height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

# Mnozenie
for y in range(height):
    for x in range(width):

        L = int(image1_matrix[x][y])
        if L == 255:
            L = const
        elif L == 0:
            L = 0
        else:
            L = (int(image1_matrix[x][y]) * const) / 255

# Zaokroglenie do najblizszej wartosci calkowitej z gory
# i przypisanie wartosci
result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

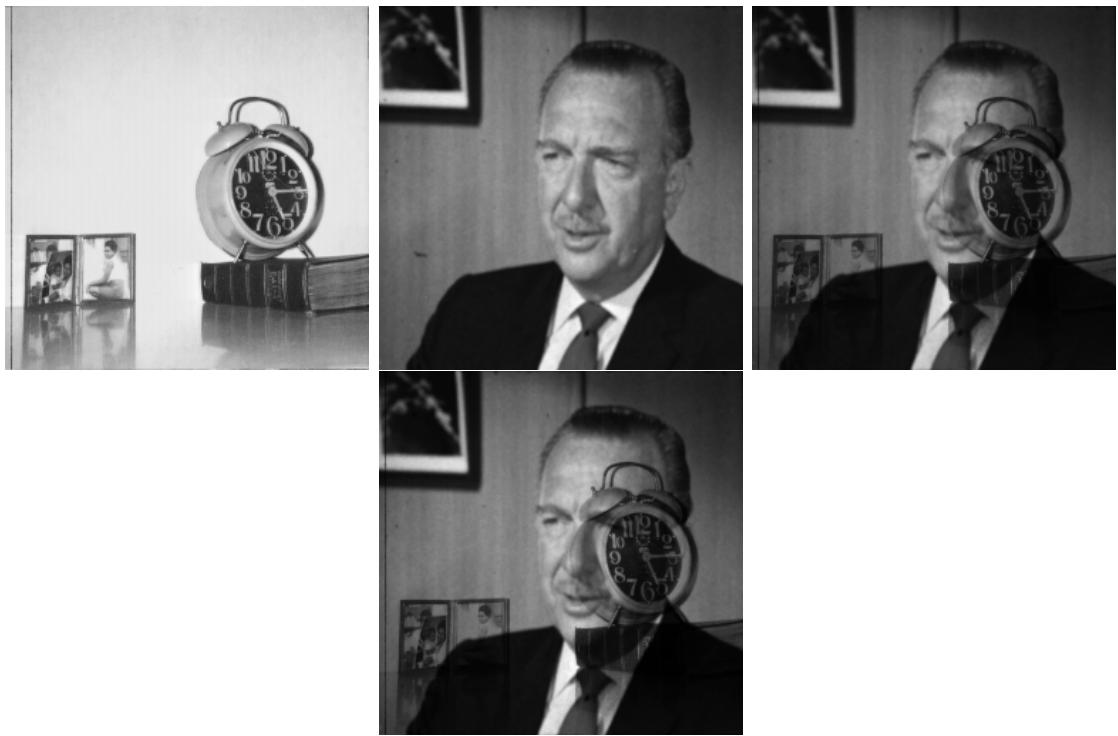
# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))

```

3.4 Mnożenie obrazu przez inny obraz

Mnożenie obrazu f przez inny obraz wykonuje się mnożąc każdy element obrazu $P1_{i,j}$ (wartość funkcji obrazowej piksla) przez odpowiadającego piksla drugiego obrazu $P2_{i,j}$

1. Weź dwa identycznych rozmiarów obrazy P_1 i P_2 .
2. Dla wszystkich pikseli w obrazie wykonaj:
3. Jeżeli składowa piksela $P_1[i, j]$ ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości składowej $P_2[i, j]$.
4. W przeciwnym przypadku, jeśli składowa piksela $P_1[i, j]$ ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
5. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 3.7: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.8: (Od lewej) Pierwsze dwa to szare obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.4: Mnożenie obrazu szarego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        L = int(image1_matrix[x][y])
        if L == 255:
            L = image2_matrix[x][y]
        elif L == 0:
            L = 0

```

```
else:  
    L = (int(image1_matrix[x][y]) * int(image2_matrix[x]  
        ][y]))/255  
  
    # Zaokrąglenie do najbliższej wartości całkowitej z  
    # gory  
    # i przypisanie wartości  
    result_matrix[x][y] = math.ceil(L)  
  
    # Poszukiwanie minimum i maksimum  
    if f_min > L:  
        f_min = L  
    if f_max < L:  
        f_max = L  
  
# Normalizacja  
norm_matrix = np.zeros((width, height), dtype=np.uint8)  
for y in range(height):  
    for x in range(width):  
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min  
            ) / (f_max - f_min))
```

3.5 Mieszanie obrazów z określonym współczynnikiem

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami α i $(1 - \alpha)$, odpowiednio, wg wzoru:

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

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

1. Weź dwa identycznych rozmiarów obrazy P_1 i P_2 .
2. Określ współczynnik mieszania α wyrażony jako liczba rzeczywista z zakresu $<0, 1>$; 0 reprezentuje pewną przezroczystość, 1 - nieprzezroczystości.
3. Dla wszystkich pikseli w obrazach wejściowych wykonuj $Q(i, j) = \alpha * P_1(i, j) + (1 - \alpha) * P_2(i, j)$



Rysunek 3.9: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mie- szania obrazów ze współczynnikiem $\alpha=0.5$, poniżej obraz wynikowy po normalizacji



Rysunek 3.10: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$, poniżej obraz wynikowy po normalizacji

Listing 3.5: Mieszanie obrazów szarych z określonym współczynnikiem

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        L = float(image1_matrix[x][y]) * alfa + (1-alfa) *
            float(image2_matrix[x][y])

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory

```

```
# i przypisanie wartosci
result_matrix[x][y] = math.ceil(L)

# Poszukiwanie minimum i maksimum
if f_min > L:
    f_min = L
if f_max < L:
    f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min) / (f_max - f_min))
```

3.6 Potęgowanie obrazu (z zadana potęgą)

Potęgowanie obrazu jest szczególnym przypadkiem operacji mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

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



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

Listing 3.6: Potęgowanie obrazu szarego (z zadana potęgą)

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
```


3.7 Dzielenie obrazu przez (zadaną) liczbę

Dzielenie obrazów stosuje się w celu korekcji cieniowania między poziomami szarości.

1. Dla wszystkich pikseli w tych obrazach wykonaj:
2. Policz sumy piksli ze stałą.
3. Wybierz największą sumę Q_{max} i policz równania:

$$Q[i, j] = (S * 255) / Q_{max},$$
 Wynik zaokrągluj do najbliższej górnej liczby całkowitej.



Rysunek 3.13: (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji



Rysunek 3.14: (Od lewej) Szary obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji

Listing 3.7: Dzielenie obrazu szarego przez (zadaną) liczbę

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)
```

```

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):
        L = int(image_matrix[x][y]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < L:
            Q_max = L

for y in range(height):
    for x in range(width):
        L = int(image_matrix[x][y]) + int(const)
        Q_L = (L * 255) / Q_max

        # Zaokroglenie do najblizszej wartosci całkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(Q_L)

        # Poszukiwanie minimum i maksimum
        if f_min > Q_L:
            f_min = Q_L
        if f_max < Q_L:
            f_max = Q_L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)
            / (f_max - f_min))

```

3.8 Dzielenie obrazu przez przez inny obraz

1. Weź dwa identycznych rozmiarów obrazy $P1$ i $P2$
2. Dla wszystkich pikseli w tych obrazach wykonaj:
3. Policz sumy piksli ze stałą.
4. Wybierz największą sumę Q_{max} i policz równania:
$$Q[i, j] = (S * 255) / Q_{max},$$

Wynik zaokrągluj do najbliższej górnej liczby całkowitej.



Rysunek 3.15: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 3.16: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku podzielenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 3.8: Dzielenie obrazu szarego przez przeszywany obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):
        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])

        # Poszukiwanie maksimum
        if Q_max < L:

```

```

Q_max = L

for y in range(height):
    for x in range(width):

        # Obliczanie sumy
        L = int(image1_matrix[x][y]) + int(image2_matrix[x][y])
        Q_L = (L * 255) / Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(Q_L)

        # Poszukiwanie minimum i maksimum
        if f_min > Q_L:
            f_min = Q_L
        if f_max < Q_L:
            f_max = Q_L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)
            / (f_max - f_min))

```

3.9 Pierwiastkowanie obrazu

Pierwiastkowanie obrazu jest szczególnym przypadkiem operacji potęgowania obrazów, gdzie wykładnikiem jest ułamek. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

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



Rysunek 3.17: (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ($\alpha=1/2$), obraz po normalizacji



Rysunek 3.18: (Od lewej) Szary obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ($\alpha=1/3$), obraz po normalizacji

Listing 3.9: Pierwiastkowanie obrazu szarego

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)
```

```

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

alfa = 1/deg # Zamiana stopnia pierwiastka na ulamek

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])

        # Poszukiwanie maksimum
        if f_img_max < L:
            f_img_max = L

for y in range(height):
    for x in range(width):

        L = int(image_matrix[x][y])
        if L == 255:
            L = 255
        elif L == 0:
            L = 0
        else:
            L = math.pow(int(image_matrix[x][y]) / f_img_max,
                        alfa) * 255

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(L)

        # Poszukiwanie minimum i maksimum
        if f_min > L:
            f_min = L
        if f_max < L:
            f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):

```

```
norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min  
) / (f_max - f_min))
```

3.10 Logarytmowanie obrazu

Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Wykorzystano wzór z normalizacją /citeWykRat:

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

Przesunięcie funkcji obrazowej f do góry o 1 przed jej logarytmowaniem wynika z nieokreśloności logarytmu w zerze. Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. .



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

Listing 3.10: Logarytmowanie obrazu szarego

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height), dtype=np.uint8)
```

```

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):
        L = int(image_matrix[x][y])

        # Poszukiwanie maksimum
        if f_img_max < L:
            f_img_max = L

for y in range(height):
    for x in range(width):
        L = int(image_matrix[x][y])

        if L == 0:
            L = 0
        else:
            L = (math.log(1 + L) / math.log(1 + f_img_max)) *
                255

        # Zaokroglenie do najblzszej wartosci calkowitej z
        gory
        # i przypisanie wartosci
        result_matrix[x][y] = math.ceil(L)

        # Poszukiwanie minimum i maksimum
        if f_min > L:
            f_min = L
        if f_max < L:
            f_max = L

# Normalizacja
norm_matrix = np.zeros((width, height), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y] = 255 * ((result_matrix[x][y] - f_min)
            / (f_max - f_min))

```

Rozdział 4

Operacje sumowania arytmetycznego obrazów barwowych

Arytmetyczne operacje na obrazach barwowych przeprowadza się wykonując działania na pojedynczych pikslach i są uwarunkowane wymaganiami zależnymi od typu operacji. W przedstawionych zadaniach poruszamy się w przestrzeni barw RGB, a do normalizacji wykorzystano wzór:

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

4.1 Sumowanie (określonej) stałej z obrazem

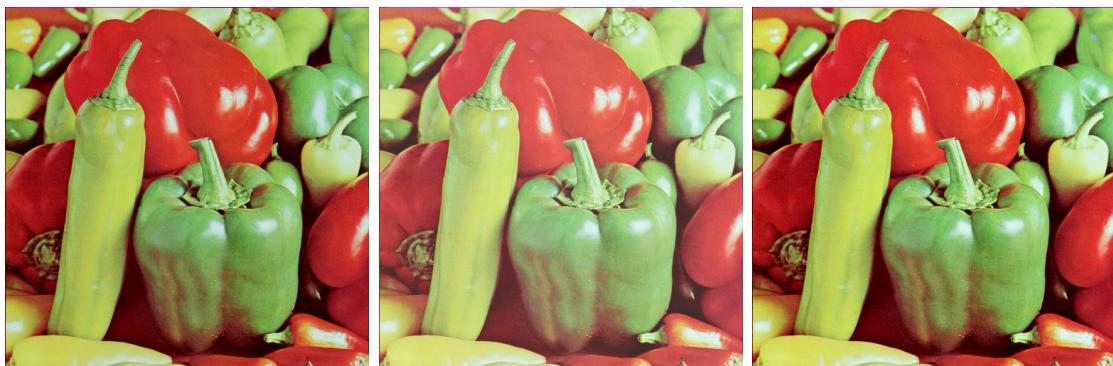
Algorytm sumowania obrazu barwowego z określona stałą polega na dodaniu do każdej składowej barwowej pojedynczego piksla stałej liczby. Po operacji sumowania obraz poddawany jest normalizacji.

1. Policz sumy wartości kazdego piksla ze stałą (*const*).
2. Jeżeli jedna z tych sum jest większa niż 255 to:
 3. Wybierz największą sumę Q_{max} i policz D_{max} ze wzoru: $D_{max}[i, j] = (Q_{max}[i, j] - 255)$
 4. Oblicz $X = D_{max}/255$
 5. Policz sumy ze wzoru

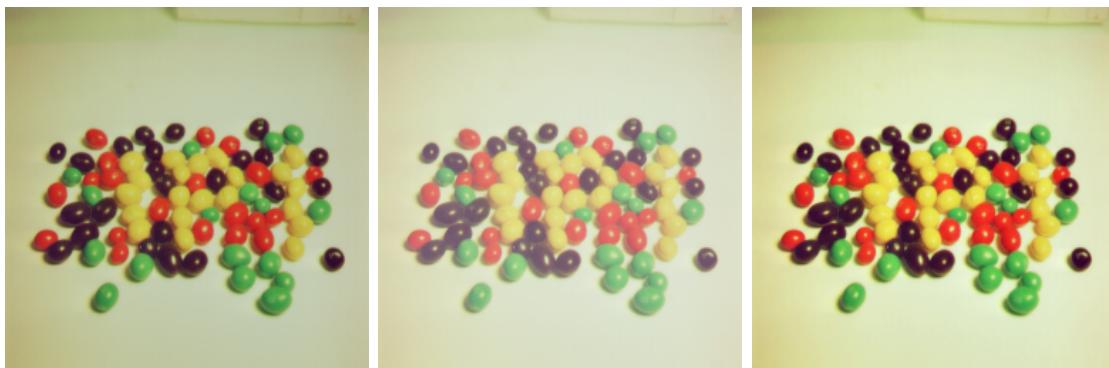
$$Q_R[i, j] = P_R[i, j] - (P_R * X) + const - (const * X) - 1$$

$$Q_G[i, j] = P_G[i, j] - (P_G * X) + const - (const * X) - 1$$

$$Q_B[i, j] = P_B[i, j] - (P_B * X) + const - (const * X) - 1$$



Rysunek 4.1: (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 50, obraz po normalizacji



Rysunek 4.2: (Od lewej) Barwowy obraz wejściowy, obraz po sumowaniu ze stałą = 100, obraz po normalizacji

Listing 4.1: Sumowanie obrazu barwowego ze stałą

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R = int(image_matrix[x][y][0]) + int(const)
        G = int(image_matrix[x][y][1]) + int(const)
        B = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R, G, B]):
            Q_max = max([R, G, B])

# Sprawdzenie czy maksimum przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255

```

```

X = (D_max/255) # Obliczenie proporcji

# Obliczenie sum z uwzglednieniem zakresu
for y in range(height):
    for x in range(width):
        R = (image_matrix[x][y][0] - (image_matrix[x][y][0] * X
            )) + (const - (const * X))
        G = (image_matrix[x][y][1] - (image_matrix[x][y][1] * X
            )) + (const - (const * X))
        B = (image_matrix[x][y][2] - (image_matrix[x][y][2] * X
            )) + (const - (const * X))

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(R)
        result_matrix[x][y][1] = math.ceil(G)
        result_matrix[x][y][2] = math.ceil(B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([R, G, B]):
            f_min = min([R, G, B])
        if f_max < max([R, G, B]):
            f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -
            f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -
            f_min) / (f_max - f_min))

```

4.2 Sumowanie dwóch obrazów

Algebraiczne sumowanie obrazów f i f' jest określone jedynie dla obrazów o tych samych wymiarach $M \times N$ i strukturze ich macierzy. Dodawanie obrazów jest użyteczne w uśrednianiu obrazów, wykonywanym w celu zredukowania na nich szumu. 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.

1. Weź dwa identycznych rozmiarów obrazy $P1$ i $P2$
2. Dla wszystkich pikseli w tych obrazach wykonaj:
3. Policz sumy odpowiadających składowych barwy.
4. Wybierz największą sumę $Q_{max} = \max(R_s, G_s, B_s)$ i policz równania:

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

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

$$Q_B[i, j] = (B_S * 255) / Q_{max}.$$

Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 4.3: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.4: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstaje w wyniku sumowania obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.2: Sumowanie obrazów barwowych

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
Q_max = 0
D_max = 0
X = 0
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum

```

```

R = int(image1_matrix[x][y][0]) + int(image2_matrix[x][
    y][0])
G = int(image1_matrix[x][y][1]) + int(image2_matrix[x][
    y][1])
B = int(image1_matrix[x][y][2]) + int(image2_matrix[x][
    y][2])

# Poszukiwanie maksimum
if Q_max < max([R, G, B]):
    Q_max = max([R, G, B])

# Sprawdzenie czy maximum przekracza zakres
if Q_max > 255:
    D_max = Q_max - 255
    X = (D_max/255) # Obliczenie proporcji

# Obliczenie sum z uwzglednieniem zakresu
for y in range(height):
    for x in range(width):
        R = (image1_matrix[x][y][0] - (image1_matrix[x][y][0] *
            X)) + (image2_matrix[x][y][0] - (image2_matrix[x][
            y][0] * X))
        G = (image1_matrix[x][y][1] - (image1_matrix[x][y][1] *
            X)) + (image2_matrix[x][y][1] - (image2_matrix[x][
            y][1] * X))
        B = (image1_matrix[x][y][2] - (image1_matrix[x][y][2] *
            X)) + (image2_matrix[x][y][2] - (image2_matrix[x][
            y][2] * X))

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(R)
        result_matrix[x][y][1] = math.ceil(G)
        result_matrix[x][y][2] = math.ceil(B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([R, G, B]):
            f_min = min([R, G, B])
        if f_max < max([R, G, B]):
            f_max = max([R, G, B])

# Normalizacja

```

```

norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] - f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] - f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] - f_min) / (f_max - f_min))

```

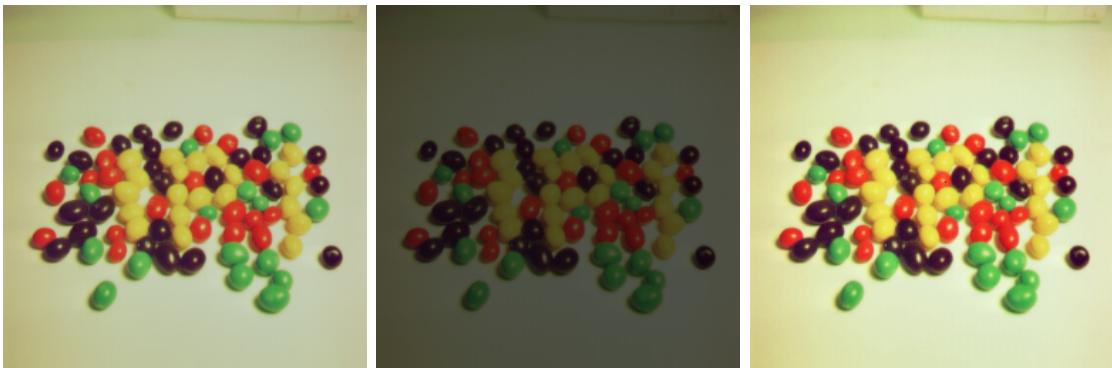
4.3 Mnożenie obrazu przez zadaną liczbę

Mnożenie obrazu f przez skalar wykonuje się mnożąc każdy element obrazu $f_{i,j}$ (wartość funkcji obrazowej piksla) przez ten skalar. Barwa wynikowa jest zawsze barwą ciemniejszą.

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa barwy piksela $P_1[i, j]$ ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości stalej.
3. W przeciwnym przypadku, jeżeli składowa barwy piksła $P_1[i, j]$ ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 4.5: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 50, obraz po normalizacji



Rysunek 4.6: (Od lewej) Szary obraz wejściowy, obraz po przemnożeniu przez liczbę 100, obraz po normalizacji

Listing 4.3: Mnożenie obrazu barwowego przez zadaną liczbę

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
```

```

width = image_matrix.shape[1]      # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = int(image_matrix[x][y][0])
        G = int(image_matrix[x][y][1])
        B = int(image_matrix[x][y][2])

        if R == 255:
            R = const
        elif R == 0:
            R = 0
        else:
            R = (int(image_matrix[x][y][0]) * int(const)) / 255

        if G == 255:
            G = const
        elif G == 0:
            G = 0
        else:
            G = (int(image_matrix[x][y][1]) * int(const)) / 255

        if B == 255:
            B = const
        elif B == 0:
            B = 0
        else:
            B = (int(image_matrix[x][y][2]) * int(const)) / 255

        # Zaokrąglenie do najbliższej wartości całkowitej z
        # gory
        # i przypisanie wartości
        result_matrix[x][y][0] = math.ceil(R)
        result_matrix[x][y][1] = math.ceil(G)
        result_matrix[x][y][2] = math.ceil(B)

```


4.4 Mnożenie obrazu przez inny obraz

Mnożenie obrazu f przez inny obraz wykonuje się mnożąc każdy element obrazu $P1_{i,j}$ (wartość funkcji obrazowej piksla) przez odpowiadającego piksla drugiego obrazu $P2_{i,j}$

1. Dla wszystkich pikseli w obrazie wykonaj:
2. Jeżeli składowa piksela $P1[i, j]$ ma wartość 255 to składowa wynikowa otrzymuje wartość odpowiadającą wartości składowej $P2[i, j]$.
3. W przeciwnym przypadku, jeżeli składowa piksela $P1[i, j]$ ma wartość 0 to składowa wynikowa otrzymuje wartość 0.
4. W przeciwnym wypadku mnóż odpowiednie składowe, a wynik dziel przez 255 zaokrąglając do najbliższej liczby całkowitej.



Rysunek 4.7: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstający w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.8: (Od lewej) Pierwsze dwa to barwowe obrazy wejściowe, następnie obraz powstały w wyniku przemnożenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.4: Mnożenie obrazu barwowego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        if R == 255:
            result_matrix[x][y][0] = 0
            result_matrix[x][y][1] = 0
            result_matrix[x][y][2] = 0
        else:
            result_matrix[x][y][0] = (f_max - f_min) * R / 255 + f_min
            result_matrix[x][y][1] = (f_max - f_min) * G / 255 + f_min
            result_matrix[x][y][2] = (f_max - f_min) * B / 255 + f_min
    
```

```

R = image2_matrix[x][y][0]
elif R == 0:
    R = 0
else:
    R = (int(image1_matrix[x][y][0]) * int(
        image2_matrix[x][y][0]))/255

if G == 255:
    G = image2_matrix[x][y][1]
elif G == 0:
    G = 0
else:
    G = (int(image1_matrix[x][y][1]) * int(
        image2_matrix[x][y][1]))/255

if B == 255:
    B = image2_matrix[x][y][2]
elif B == 0:
    B = 0
else:
    B = (int(image1_matrix[x][y][2]) * int(
        image2_matrix[x][y][2]))/255

# Zaokrąglenie do najbliższej wartości całkowitej z
# gory
# i przypisanie wartości
result_matrix[x][y][0] = math.ceil(R)
result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))

```

```
norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -  
    f_min) / (f_max - f_min))  
norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -  
    f_min) / (f_max - f_min))
```

4.5 Mieszanie obrazów z określonym współczynnikiem

Mieszanie dwóch obrazów polega na sumowaniu ich z wagami α i $(1-\alpha)$, odpowiednio, wg 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^I w obraz f . W mieszaniu obrazów nie ma problemu normalizacji, a jedynie jednolitości struktur i typów obrazowych.

1. Weź dwa identycznych rozmiarów obrazy P_1 i P_2 .
2. Określ współczynnik mieszania α wyrażony jako liczba rzeczywista z zakresu $<0, 1>$; 0 reprezentuje pewną przezroczystość, 1 - nieprzezroczystości.
3. Dla wszystkich pikseli w obrazach wejściowych wykonuj $Q(i, j) = \alpha * P_1(i, j) + (1 - \alpha) * P_2(i, j)$



Rysunek 4.9: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.5$, poniżej obraz wynikowy po normalizacji



Rysunek 4.10: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku mieszania obrazów ze współczynnikiem $\alpha=0.8$, poniżej obraz wynikowy po normalizacji

Listing 4.5: Mieszanie obrazów barwowych z określonym współczynnikiem

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0

for y in range(height):
    for x in range(width):

        R = float(image1_matrix[x][y][0]) * alfa + (1-alfa) *
            float(image2_matrix[x][y][0])
        G = float(image1_matrix[x][y][1]) * alfa + (1-alfa) *
            float(image2_matrix[x][y][1])

```


4.6 Potęgowanie obrazu

Potęgowanie obrazu jest szczególnym przypadkiem operacji mnożenia obrazów. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

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



Rysunek 4.11: (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=2$, obraz po normalizacji



Rysunek 4.12: (Od lewej) Barwowy obraz wejściowy, obraz po podniesieniu do potęgi $\alpha=3$, obraz po normalizacji

Listing 4.6: Potęgowanie obrazu barwowego

```
image1_matrix = self.im1
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
```

```

f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

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

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        if R == 0:
            R = 0
        else:
            R = 255 * (math.pow(int(image1_matrix[x][y][0]), /
                                f_img_max, alfa))

        if G == 0:
            G = 0
        else:
            G = 255 * (math.pow(int(image1_matrix[x][y][1]), /
                                f_img_max, alfa))

        if B == 0:
            B = 0
        else:
            B = 255 * (math.pow(int(image1_matrix[x][y][2]), /
                                f_img_max, alfa))

# Zaokrąglenie do najbliższej wartości całkowitej z
# gory
# i przypisanie wartości
result_matrix[x][y][0] = math.ceil(R)

```

```

result_matrix[x][y][1] = math.ceil(G)
result_matrix[x][y][2] = math.ceil(B)

# Poszukiwanie minimum i maksimum
if f_min > min([R, G, B]):
    f_min = min([R, G, B])
if f_max < max([R, G, B]):
    f_max = max([R, G, B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] - f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] - f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] - f_min) / (f_max - f_min))

```

4.7 Dzielenie obrazu przez (zadaną) liczbę

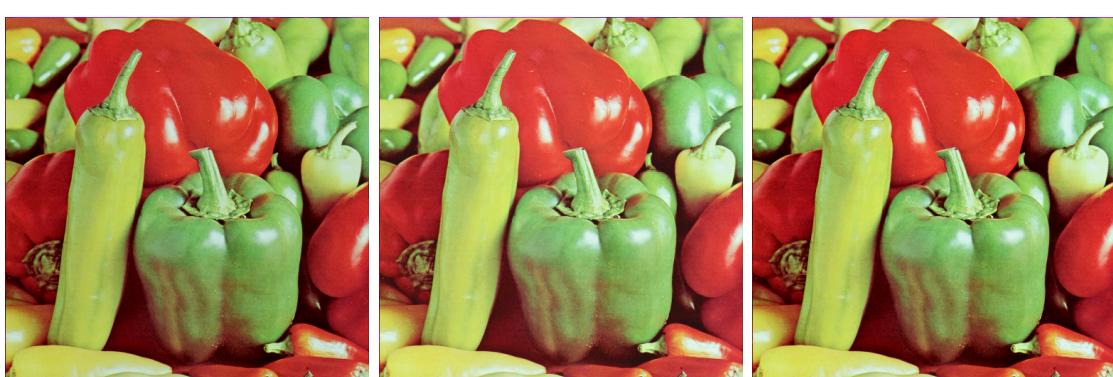
Dzielenie obrazów stosuje się w celu korekcji cieniowania między poziomami szarości.

1. Dla wszystkich pikseli w tych obrazach wykonaj:
 2. Policz sumy pikseli ze stałą.
 3. Wybierz największą sumę $Q_{max} = \max(R_s, G_s, B_s)$ i policz równania:

$$Q_R[i, j] = (R_s * 255)/Q_{max},$$

$$Q_G[i, j] = (G_s * 255)/Q_{max},$$

$$Q_B[i, j] = (B_s * 255)/Q_{max}.$$
- Wynik zaokrąglaj do najbliższej górnej liczby całkowitej.



Rysunek 4.13: (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=15, obraz po normalizacji



Rysunek 4.14: (Od lewej) Barwowy obraz wejściowy, obraz po podzieleniu przez liczbę=3, obraz po normalizacji

Listing 4.7: Dzielenie obrazu barwowego przez (zadaną) liczbę

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
```

```

width = image_matrix.shape[1]      # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R_S, G_S, B_S]):
            Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        Q_R = (R_S * 255)/Q_max
        Q_G = (G_S * 255)/Q_max
        Q_B = (B_S * 255)/Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(Q_R)
        result_matrix[x][y][1] = math.ceil(Q_G)
        result_matrix[x][y][2] = math.ceil(Q_B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([Q_R, Q_G, Q_B]):
            f_min = min([Q_R, Q_G, Q_B])

```


4.8 Dzielenie obrazu przez inny obraz

1. Dla wszystkich pikseli w tych obrazach wykonaj:
 2. Weź dwa identycznych rozmiarów obrazy $P1$ i $P2$
 3. Policz sumy pikseli ze stałą.
 4. Wybierz największą sumę $Q_{max} = \max(R_s, G_s, B_s)$ i policz równania:

$$Q_R[i, j] = (R_s * 255) / Q_{max}$$

$$Q_G[i, j] = (G_s * 255) / Q_{max}$$

$$Q_B[i, j] = (B_s * 255) / Q_{max}$$
- Wynik zaokrągluj do najbliższej górnej liczby całkowitej.



Rysunek 4.15: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstał w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji



Rysunek 4.16: (Od lewej) Dwa obrazy wejściowe, następnie obraz powstały w wyniku dzielenia obrazów, poniżej obraz wynikowy po normalizacji

Listing 4.8: Dzielenie obrazu barwowego przez inny obraz

```

image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image1_matrix[x][y][0]) + int(image2_matrix[x][y][0])

```

```

G_S = int(image1_matrix[x][y][1]) + int(image2_matrix[x]
    ][y][1])
B_S = int(image1_matrix[x][y][2]) + int(image2_matrix[x]
    ][y][2])

# Poszukiwanie maksimum
if Q_max < max([R_S, G_S, B_S]):
    Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image1_matrix[x][y][0]) + int(image2_matrix[x]
            ][y][0])
        G_S = int(image1_matrix[x][y][1]) + int(image2_matrix[x]
            ][y][1])
        B_S = int(image1_matrix[x][y][2]) + int(image2_matrix[x]
            ][y][2])

        Q_R = (R_S * 255)/Q_max
        Q_G = (G_S * 255)/Q_max
        Q_B = (B_S * 255)/Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(Q_R)
        result_matrix[x][y][1] = math.ceil(Q_G)
        result_matrix[x][y][2] = math.ceil(Q_B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([Q_R, Q_G, Q_B]):
            f_min = min([Q_R, Q_G, Q_B])
        if f_max < max([Q_R, Q_G, Q_B]):
            f_max = max([Q_R, Q_G, Q_B])

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] -
            f_min) / (f_max - f_min))

```

```
norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] -  
    f_min) / (f_max - f_min))  
norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] -  
    f_min) / (f_max - f_min))
```

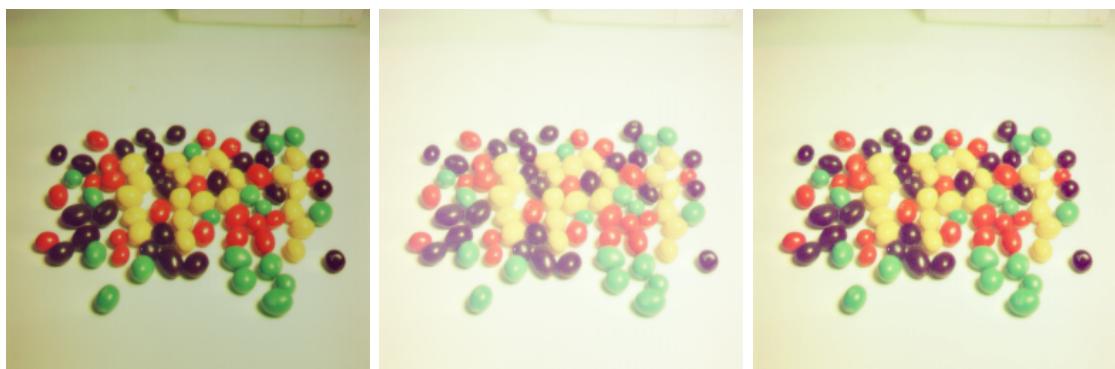
4.9 Pierwiastkowanie obrazu

Pierwiastkowanie obrazu jest szczególnym przypadkiem operacji potęgowania obrazów, gdzie wykładnikiem jest ułamek. Aby uniknąć wykroczenia poza zakres, skorzystano ze znormalizowanego wzoru [2]:

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



Rysunek 4.17: (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem kwadratowym ($\alpha=1/2$), obraz po normalizacji



Rysunek 4.18: (Od lewej) Barwowy obraz wejściowy, obraz po spierwiastkowaniu pierwiastkiem stopnia trzeciego ($\alpha=1/3$), obraz po normalizacji

Listing 4.9: Pierwiastkowanie obrazu barwowego

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)
```

```

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
Q_max = 0

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        # Poszukiwanie maksimum
        if Q_max < max([R_S, G_S, B_S]):
            Q_max = max([R_S, G_S, B_S])

for y in range(height):
    for x in range(width):

        # Obliczanie sum
        R_S = int(image_matrix[x][y][0]) + int(const)
        G_S = int(image_matrix[x][y][1]) + int(const)
        B_S = int(image_matrix[x][y][2]) + int(const)

        Q_R = (R_S * 255) / Q_max
        Q_G = (G_S * 255) / Q_max
        Q_B = (B_S * 255) / Q_max

        # Zaokroglenie do najblizszej wartosci calkowitej z
        # gory
        # i przypisanie wartosci
        result_matrix[x][y][0] = math.ceil(Q_R)
        result_matrix[x][y][1] = math.ceil(Q_G)
        result_matrix[x][y][2] = math.ceil(Q_B)

        # Poszukiwanie minimum i maksimum
        if f_min > min([Q_R, Q_G, Q_B]):
            f_min = min([Q_R, Q_G, Q_B])
        if f_max < max([Q_R, Q_G, Q_B]):
            f_max = max([Q_R, Q_G, Q_B])

```

```

# Normalizacja
norm_matrix = np.zeros((width, height, 3), dtype=np.uint8)
for y in range(height):
    for x in range(width):
        norm_matrix[x][y][0] = 255 * ((result_matrix[x][y][0] - f_min) / (f_max - f_min))
        norm_matrix[x][y][1] = 255 * ((result_matrix[x][y][1] - f_min) / (f_max - f_min))
        norm_matrix[x][y][2] = 255 * ((result_matrix[x][y][2] - f_min) / (f_max - f_min))

```

4.10 Logarytmowanie obrazu

Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu. Wykorzystano wzór z normalizacją /citeWykRat:

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

Przesunięcie funkcji obrazowej f do góry o 1 przed jej logarytmowaniem wynika z nieokreśloności logarytmu w zerze. Logarytmowanie obrazu powoduje rozjaśnienie i zróżnicowanie najciemniejszych obszarów obrazu.



Rysunek 4.19: (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji



Rysunek 4.20: (Od lewej) Barwowy obraz wejściowy, obraz po logarytmowaniu logarytmem naturalnym, obraz po normalizacji

Listing 4.10: Logarytmowanie obrazu barwowego

```
image1_matrix = self.im1
image2_matrix = self.im2
height = image1_matrix.shape[0]      # wysokosc
width = image1_matrix.shape[1]       # szereoksc
```

```

result_matrix = np.zeros((width, height, 3), dtype=np.uint8)

# Inicjalizacja zmiennych
f_min = 255
f_max = 0
f_img_max = 0

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        # Poszukiwanie maksimum
        if f_img_max < max([R, G, B]):
            f_img_max = max([R, G, B])

for y in range(height):
    for x in range(width):

        R = int(image1_matrix[x][y][0])
        G = int(image1_matrix[x][y][1])
        B = int(image1_matrix[x][y][2])

        if R == 0:
            R = 0
        else:
            R = math.log(1 + int(image1_matrix[x][y][0])) /
                math.log(1 + int(f_img_max)) * 255

        if G == 0:
            G = 0
        else:
            G = math.log(1 + int(image1_matrix[x][y][1])) /
                math.log(1 + int(f_img_max)) * 255

        if B == 0:
            B = 0
        else:
            B = math.log(1 + int(image1_matrix[x][y][2])) /
                math.log(1 + int(f_img_max)) * 255

```


Rozdział 5

Operacje geometryczne na obrazie

Transformacje geometryczne są szczególnie wykorzystywane w przypadku dopasowywania obrazu do układu współrzędnych oraz w przypadku eliminowania zniekształceń geometrycznych obrazu. W przedstawionych operacjach obrazy umieszczane są w pierwszej ćwiartce układu współrzędnych.

5.1 Przemieszczenie obrazu o zadany wektor

Przesuwanie obrazu polega na zmianie współrzędnych każdego piksela obrazu o określoną wartość zgodnie z zależnościami:

$$\begin{aligned}x^I &= x_o + \Delta x \\y^I &= y_o + \Delta y\end{aligned}$$

gdzie (x_o, y_o) - współrzędnych początkowe piksla; $(\Delta x, \Delta y)$ - wartości przesunięcia; (x^I, y^I) - współrzędne piksla po przesunięciu;



Rysunek 5.1: (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [40, 70]



Rysunek 5.2: (Od lewej) Obraz wejściowy, obraz po przesunięciu o wektor [200, 100]

Listing 5.1: Przesunięcie obrazu o zadany wektor

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

delta_y = 0 - delta_y # Poruszamy sie w pierwszej cwartce
                      # ukladu wspolrzednych

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

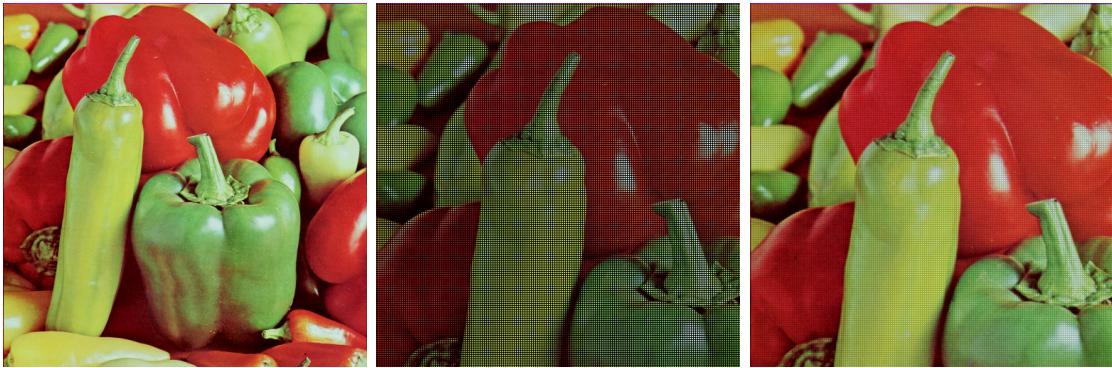
for y in range(height):
    for x in range(width):
        if 0 < y+delta_y < height and 0 < x+delta_x < width:
            result_matrix[y+delta_y][x+delta_x] = image_matrix[
                y][x]

```

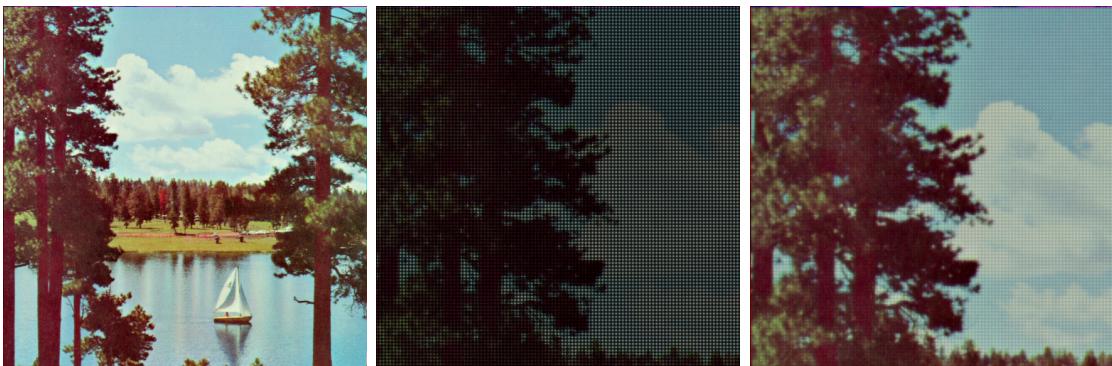
5.2 Jednorodne skalowanie obrazu

Skalowanie jednorodne obrazu polega na pomnożeniu współrzędnych każdego piksła obrazu przez współczynnik skalowania S wg. wzoru:

$$\begin{aligned}x^I &= x_o * S \\y^I &= y_o * S\end{aligned}$$



Rysunek 5.3: (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem $S=1.5$, obraz po interpolacji



Rysunek 5.4: (Od lewej) Obraz wejściowy, obraz skalowaniu jednorodnym ze współczynnikiem $S=2$, obraz po interpolacji

Listing 5.2: Jednorodne skalowanie obrazu

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

# result_matrix = np.empty((math.ceil(height/scale_y), math.
#                           ceil(width/scale_x), 3), dtype=np.uint8)
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
```

```

for y in range(height):
    for x in range(width):
        if scale*y < height and scale*x < width:
            # result_matrix[y][x] = image_matrix[scale*y][scale
            # *x]
            result_matrix[int(scale*y)][int(scale*x)] =
                image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype = np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) |
                        ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) |
                        ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (
                        resultImage2[iSafe, jSafe][1] > 0) | (
                        resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        g += resultImage2[iSafe, jSafe][1]
                        b += resultImage2[iSafe, jSafe][2]
                    n += 1
        tmp[i, j] = (r/n, g/n, b/n)
        resultImage2[i, j] = tmp[i, j]

```

5.3 Niejednorodne skalowanie obrazu

Skalowanie niejednorodne obrazu polega na pomnożeniu współrzędnych każdego piksła obrazu przez współczynniki skalowania S_x, S_y wg. wzoru:

$$\begin{aligned}x^I &= x_o * S_x \\y^I &= y_o * S_y\end{aligned}$$

gdzie (x_o, y_o) - współrzędne początkowe piksela; (S_x, S_y) - wartości współczynników skalowania; (x^I, y^I) - współrzędne piksła po skalowaniu;



Rysunek 5.5: (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem $S_x=2$ oraz współczynnikiem $S_y=1$, obraz po interpolacji



Rysunek 5.6: (Od lewej) Obraz wejściowy, obraz skalowaniu niejednorodnym ze współczynnikiem $S_x=1$ oraz współczynnikiem $S_y=2$, obraz po interpolacji

Listing 5.3: Niejednorodne skalowanie obrazu

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc
```

```

# result_matrix = np.empty((math.ceil(height/scale_y), math.
# ceil(width/scale_x), 3), dtype=np.uint8)
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        if scale_y*y < height and scale_x*x < width:
            # result_matrix[y][x] = image_matrix[scale_y*y][
            # scale_x*x]
            result_matrix[int(scale_y*y)][int(scale_x*x)] =
                image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype = np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) |
                        ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) |
                        ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (
                        resultImage2[iSafe, jSafe][1] > 0) | (
                        resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
                        g += resultImage2[iSafe, jSafe][1]
                        b += resultImage2[iSafe, jSafe][2]
                        n += 1
        tmp[i, j] = (r/n, g/n, b/n)
        resultImage2[i, j] = tmp[i, j]

```

5.4 Obracanie obrazu o dowolny kąt

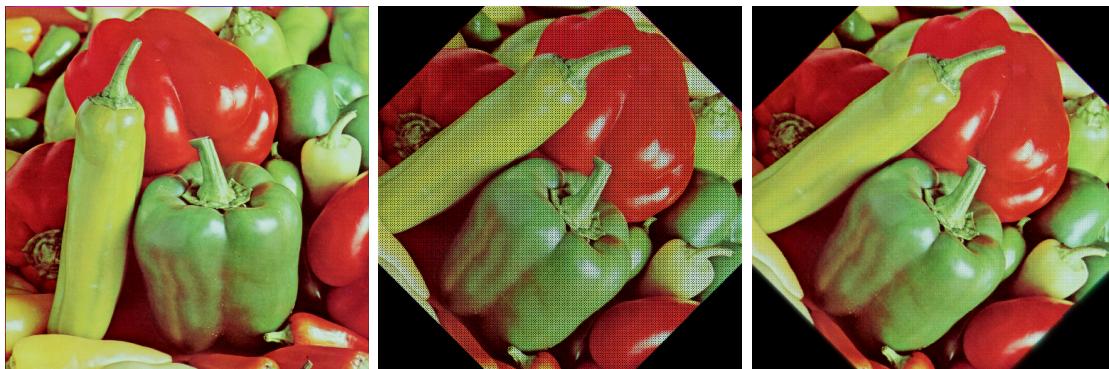
Operację obrotu dookoła początku układu współrzędnych wykonuje się zgodnie ze wzorem:

$$\begin{aligned}x^I &= x_o * \cos\alpha - y_o * \sin\alpha \\y^I &= x_o * \sin\alpha + y_o * \cos\alpha\end{aligned}$$

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

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

$$\begin{aligned}new_x &= (x - width/2) * \text{math.cos}(alfa_r) - (y - height/2) * \text{math.sin}(alfa_r) + (width/2) \\new_y &= (x - width/2) * \text{math.sin}(alfa_r) + (y - height/2) * \text{math.cos}(alfa_r) + (height/2)\end{aligned}$$



Rysunek 5.7: (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 40 stopni, obraz po interpolacji



Rysunek 5.8: (Od lewej) Obraz wejściowy, obraz po obróceniu wokół środka obrazu o kąt 110 stopni, obraz po interpolacji

Listing 5.4: Obracanie obrazu o dowolny kąt

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

# Przekształcenie na radiany
alfa_r = math.radians(alfa)

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        new_x = (x - width/2) * math.cos(alfa_r) - (y - height/2) * math.sin(alfa_r) + (width/2)
        new_y = (x - width/2) * math.sin(alfa_r) + (y - height/2) * math.cos(alfa_r) + (height/2)
        if new_y < height and new_y >= 0 and new_x >= 0 and new_x < width:
            result_matrix[int(new_y)][int(new_x)] =
                image_matrix[y][x]

resultImage2 = np.copy(result_matrix)
tmp = np.ones((height, width, 3), dtype = np.uint8)

# Interpolacja
for i in range(height):
    for j in range(width):
        r, g, b = 0, 0, 0
        n = 1
        tmp[i, j] = resultImage2[i, j]
        if (resultImage2[i, j][0] < 1) & (resultImage2[i, j][1] < 1) & (resultImage2[i, j][2] < 1):
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 2)) | ((i + iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) | ((j + jOff) < 0) else (j + jOff)
                    if (resultImage2[iSafe, jSafe][0] > 0) | (resultImage2[iSafe, jSafe][1] > 0) | (resultImage2[iSafe, jSafe][2] > 0):
                        r += resultImage2[iSafe, jSafe][0]
n = n - 1
r = r / n
tmp[i, j] = r
resultImage2[i, j] = tmp[i, j]

```

```
g += resultImage2[ iSafe , jSafe ][ 1 ]
b += resultImage2[ iSafe , jSafe ][ 2 ]
n += 1
tmp[ i , j ] = ( r/n , g/n , b/n )
resultImage2[ i , j ] = tmp[ i , j ]
```

5.5 Symetrie względem osi układu

Symetria względem osi X

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



Rysunek 5.9: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X



Rysunek 5.10: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi X

Listing 5.5: Symetrie względem osi X

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_height = height-1 #array height last index

for y in range(height):
    for x in range(width):
        result_matrix[y][x] = image_matrix[_height-y][x]
```

Symetria względem osi Y

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



Rysunek 5.11: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y



Rysunek 5.12: (Od lewej) Obraz wejściowy, obraz symetryczny względem osi Y

Listing 5.6: Symetrie względem osi Y

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_width = width - 1 #array width last index

for y in range(height):
    for x in range(width):
        result_matrix[y][x] = image_matrix[y][_width - x]
    
```

5.6 Symetrie względem zadanej prostej



Rysunek 5.13: (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu



Rysunek 5.14: (Od lewej) Obraz wejściowy, obraz symetryczny względem pionowej prostej poprowadzonej przez środek obrazu



Rysunek 5.15: (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu



Rysunek 5.16: (Od lewej) Obraz wejściowy, obraz symetryczny względem poziomej prostej poprowadzonej przez środek obrazu

Listing 5.7: Symetrie względem zadanej prostej

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

#Symetria wzgledem poziomej prostej poprowadzonej przez srodek
#obrazu
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_height = height - 1 #array height last index

param_y = height/2

for y in range(height):
    for x in range(width):
        if y < param_y:
            result_matrix[y][x] = image_matrix[y][x]
        else:
            result_matrix[y][x] = image_matrix[_height - y][x]

#Symetria wzgledem pionowej prostej poprowadzonej przez srodek
#obrazu
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)
_width = width - 1 #array width last index

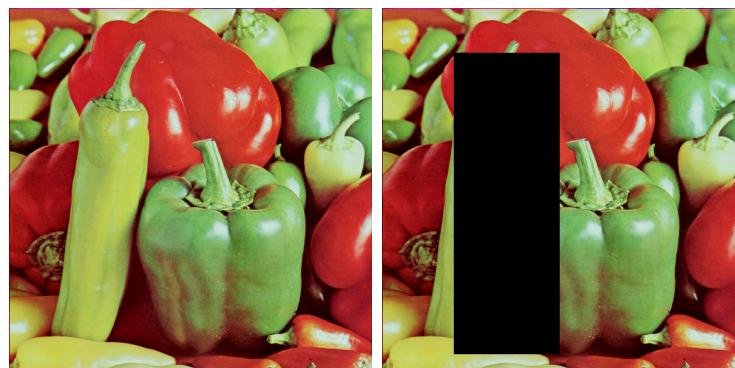
param_x = width/2

for y in range(height):

```

```
for x in range(width):
    if x < param_x:
        result_matrix[y][x] = image_matrix[y][x]
    else:
        result_matrix[y][x] = image_matrix[y][-width - x]
```

5.7 Wycinanie fragmentów obrazu



Rysunek 5.17: (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$



Rysunek 5.18: (Od lewej) Obraz wejściowy (512x512), obraz po wycięciu fragmentu o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$

Listing 5.8: Wycinanie fragmentów obrazu

```

image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

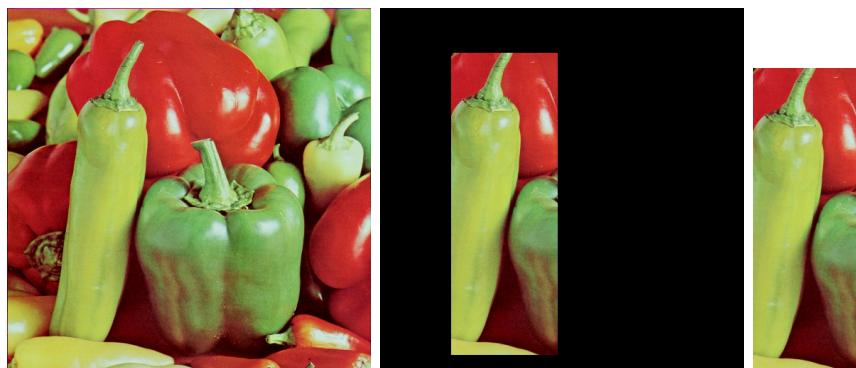
result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        # Poruszamy sie w pierwszej cwiartce osi układu
        # wspolrzednych

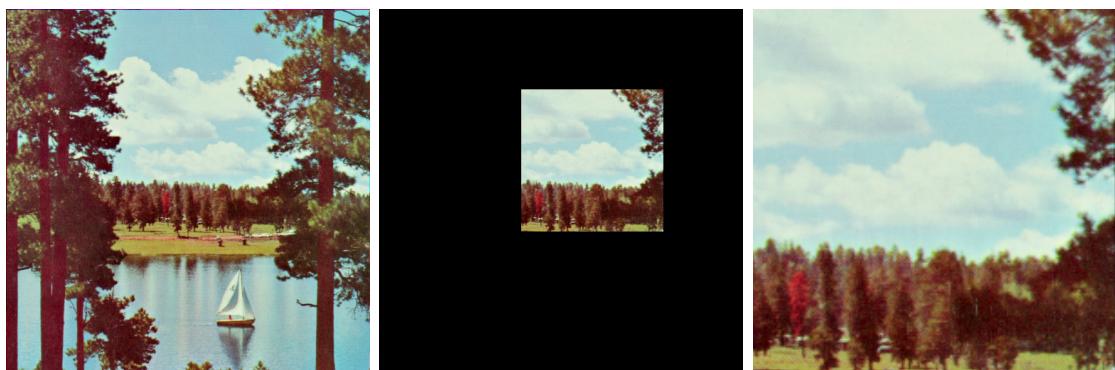
```

```
if x > x_min and x < x_max and y < height-y_min and y >
    height-y_max:
    result_matrix[y][x] = 0
else:
    result_matrix[y][x] = image_matrix[y][x]
```

5.8 Kopiowanie fragmentów obrazów



Rysunek 5.19: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 100, x_{max} = 250, y_{min} = 25, y_{max} = 450$, Skopiowany fragment (151x426)



Rysunek 5.20: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) ze skopiowanym fragmentem o współrzędnych $x_{min} = 200, x_{max} = 400, y_{min} = 200, y_{max} = 400$, Skopiowany fragment (201x201)

Listing 5.9: Kopiowanie fragmentów obrazów

```
image_matrix = self.im1
height = image_matrix.shape[0]      # wysokosc
width = image_matrix.shape[1]       # szereoksc

result_matrix = np.zeros((height, width, 3), dtype=np.uint8)

# Macierz o wymiarach wycinanego fragmentu
```

```
cut_matrix = np.zeros((y_max-y_min + 1, x_max-x_min + 1, 3),  
                      dtype=np.uint8)  
  
cut_y = 0  
for y in range(height):  
    cut_x = 0  
    for x in range(width):  
        # Poruszamy sie w pierwszej cwiartce osi układu  
        # współrzędnych  
        if x >= x_min and x <= x_max and y <= height-y_min and  
            y >= height-y_max:  
            result_matrix[y][x] = image_matrix[y][x]  
            cut_matrix[cut_y][cut_x] = image_matrix[y][x]  
            cut_x+=1  
        if cut_x > 0:  
            cut_y+=1
```

Rozdział 6

Operacje na histogramie obrazu szarego

Histogram to najprostszy opis całościowy obrazu. Dlatego stosuje się go, by rozpoznać, jakie dalsze metody i operacje należy zastosować na przetwarzanym obrazie, by osiągnąć założony cel. Jego obliczenie polega na odczytaniu szarości każdego piksla i rejestraniu jej wystąpienia o danym poziomie.

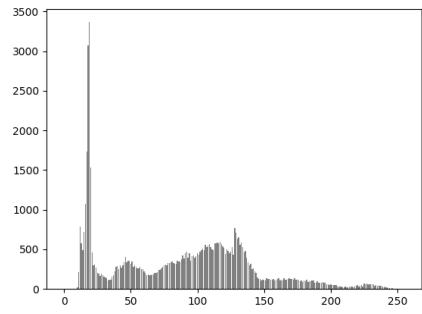
Histogram to funkcja przypisująca każdemu poziomowi szarości obrazu, liczbę piksli z danym poziomem szarości, czyli jest to wykres częstości występowania wartości piksli w obrazie.

6.1 Obliczanie histogramu

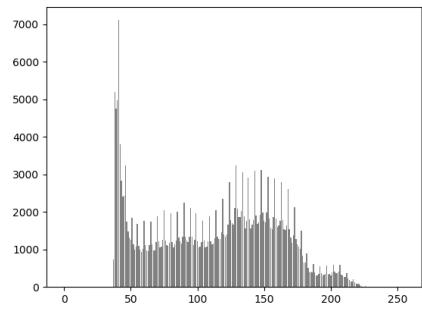
Opis algorytmu

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 (tyle, ile poziomów szarości w obrazie)
2. Dla każdego piksela:
3. Zwięksź element tablicy o indeksie równym pozycji szarości danego piksela



Rysunek 6.1: Obraz szary, histogram szarości tego obrazu



Rysunek 6.2: Obraz szary, histogram szarości tego obrazu

Listing 6.1: Obliczanie histogramu

```
def calculate(self, plot = False, image = None):
    if image is None:
        image = self.im

    width = image.shape[1]      # szereoksc
    height = image.shape[0]     # wysokosc
    hist = [0] * 256            # histogram szarosci

    for i in range(height):
        for j in range(width):
            bin = image[i, j]    # odcien szarosci
            hist[bin] += 1

    if plot:
        # tablica [0, 1, ..., 254, 255]
        bins = np.arange(256)
        self.plotHistogram(bins, hist)

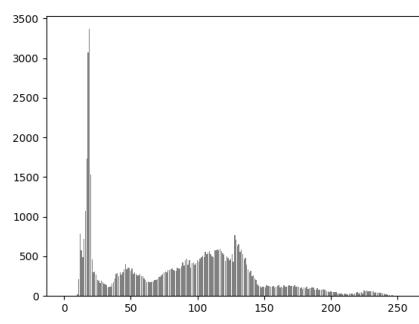
    return hist
```

6.2 Przemieszczanie histogramu

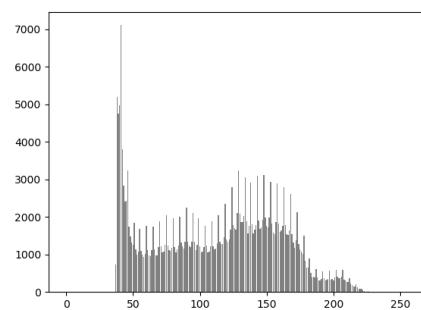
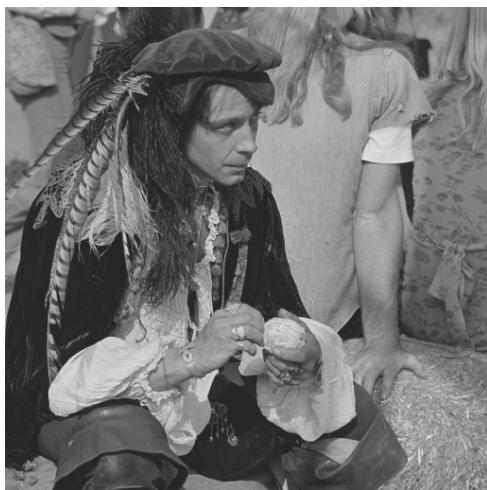
Opis algorytmu

Przemieszczenie histogramu polega na dodaniu lub odjęciu tej samej wartości od poziomu szarości każdego piksla w obrazie. W rezultacie obraz jest odpowiednio równomiernie rozjaśniony bądź przyciemniony. Nie można przekroczyć przyjętego zakresu poziomów szarości.

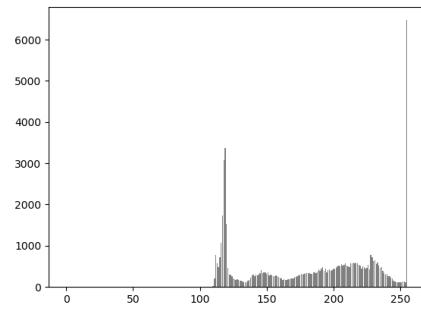
1. Do każdej wartości piksla dodaj podaną stałą o którą chcesz przemieścić histogram.
2. Jeśli wartość piksla po operacji dodawania wykracza poza zakres 255:
 3. Przypisz jej wartość 255.
 4. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0:
 5. Przypisz jej wartość 0.



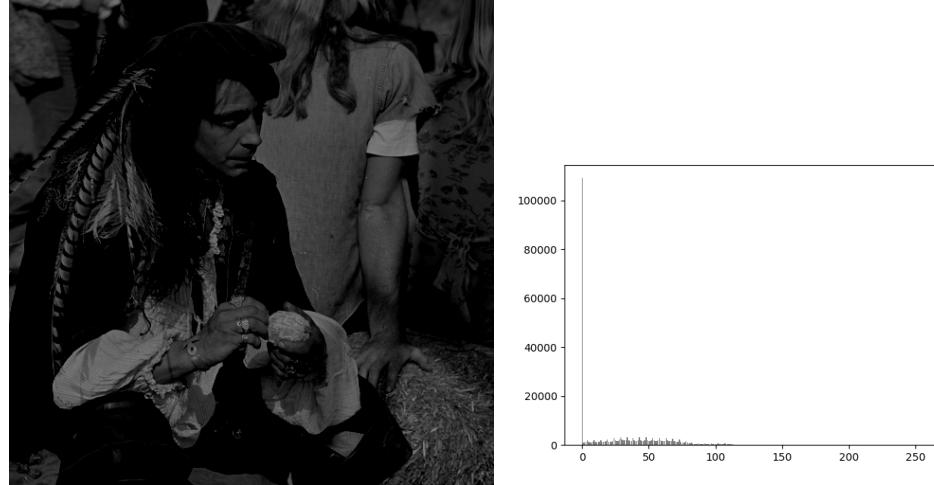
Rysunek 6.3: Obraz szary wejściowy, histogram szarości tego obrazu



Rysunek 6.5: Obraz szary wejściowy, histogram szarości tego obrazu



Rysunek 6.4: Obraz szary przesunięty o 100, histogram szarości tego obrazu



Rysunek 6.6: Obraz szary pryesunięty o -100, histogram szarości tego obrazu

Listing 6.2: Przemieszczenie histogramu

```
def move( self , const = 0 , show = False , plot = False ):
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    # przemieszczenie
    for i in range( height ):
        for j in range( width ):
            value = int( self .im [i , j ]) + const
            if value < 0:
                value = 0
            elif value > 255:
                value = 255
            resultImage [i , j ] = value

    if show:
        self .show( Image .fromarray( resultImage , "L" ))
    self .calculate( plot , resultImage )
    self .save( resultImage , self .imName , "moveHist" )
```

6.3 Rozciąganie histogramu

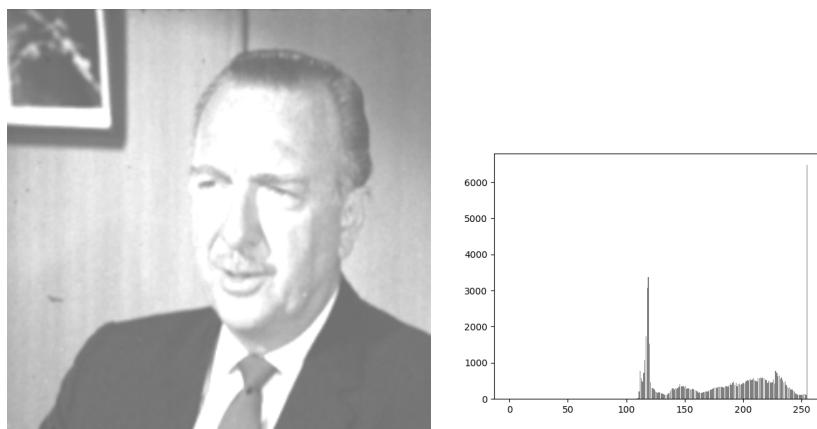
Opis algorytmu

Rozciągania histogramu dokonuje się na obrazie, którego poziomy szarości nie są rozpięte na cały możliwy zakres np. [51, 233]. Operacja rozciągnięcia histogramu rozciągnie histogram tak, aby był rozpięty na cały możliwy zakres poziomów szarości np. [0, 255].

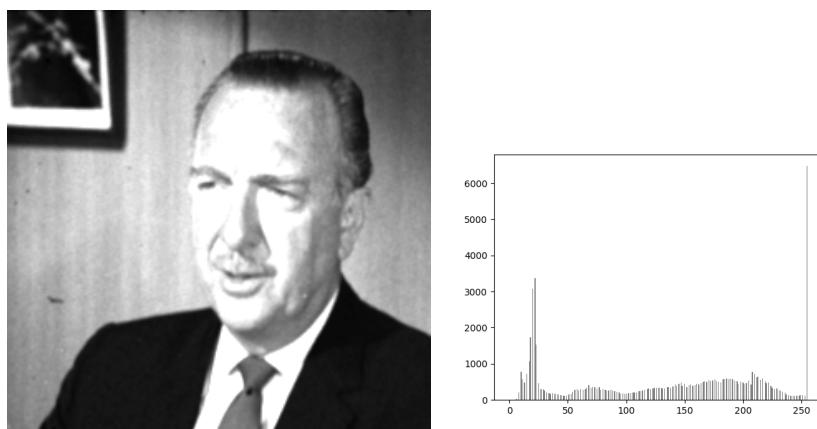
1. Znajdź w obrazie największą(\max) i najmniejszą(\min) wartość piksla
2. Dla każdego piksla:
3. Oblicz nową wartość piksla stosując wzór:

$$P_n = 255 // (\max - \min) * (P_o - \min).$$

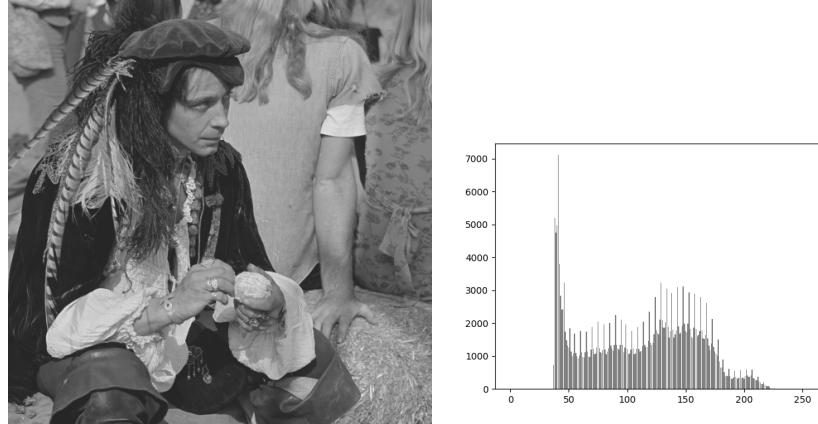
W taki sposób, jeśli odcienie szarości obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, odcienie szarości będą w zakresie [0, 255]



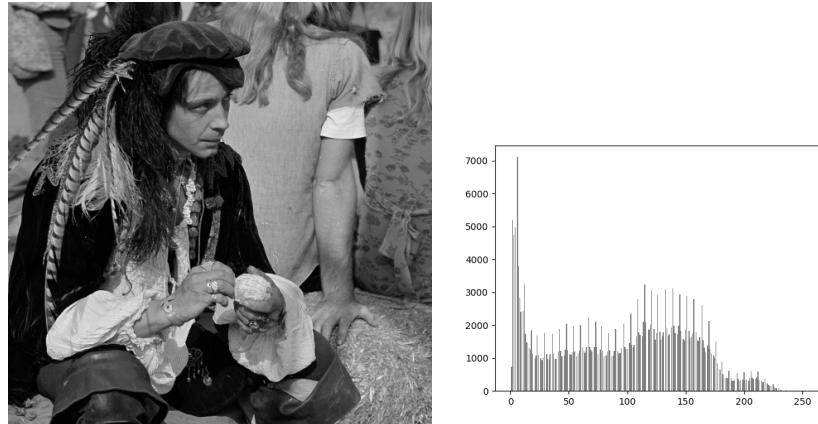
Rysunek 6.7: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.8: Obraz po rozciagnięciu, histogram szarości tego obrazu



Rysunek 6.9: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.10: Obraz po rozciągnięciu, histogram szarości tego obrazu

Listing 6.3: Rozciąganie histogramu

```
def stretch(self , show = False , plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width) , dtype=np.uint8)
    for i in range(height):
        for j in range(width):
            resultImage[i , j] = self.im[i , j]

# wartosc min i max w obrazie
    maxValue = 0
    minValue = 255
    while maxValue != 255:
```

```
# wartosci max i min w obrazie
for i in range(height):
    for j in range(width):
        currValue = resultImage[i, j]
        maxValue = max(maxValue, currValue)
        minValue = min(minValue, currValue)

# rozciaganie
for i in range(height):
    for j in range(width):
        pix = resultImage[i, j]
        resultImage[i, j] = ((255 / (maxValue - minValue)) * (pix
            - minValue))

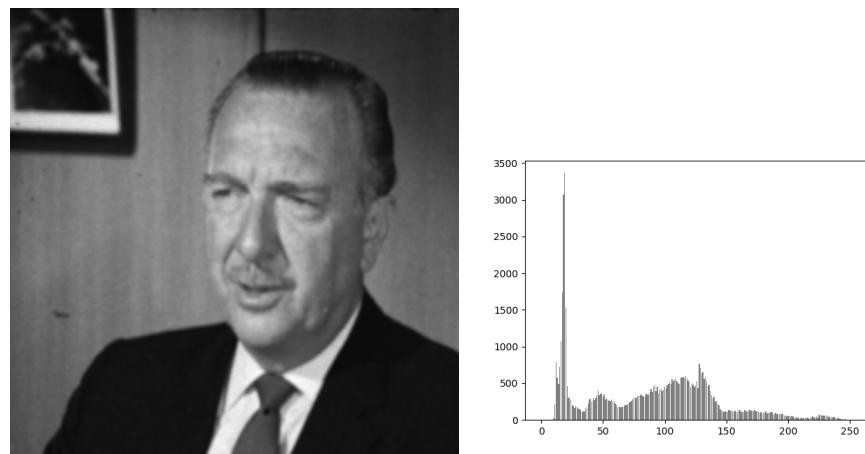
if show:
    self.show(Image.fromarray(resultImage, "L"))
self.calculate(plot, resultImage)
self.save(resultImage, self.imName, "stretchHist")
```

6.4 Progowanie lokalne

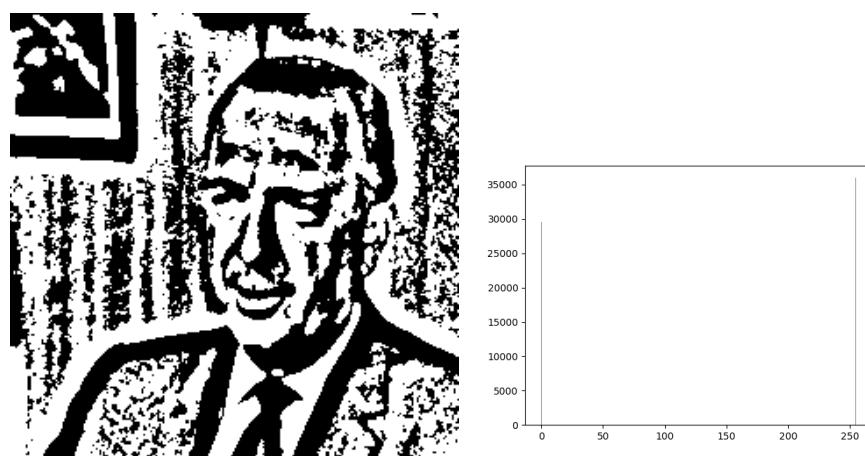
Opis algorytmu

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.

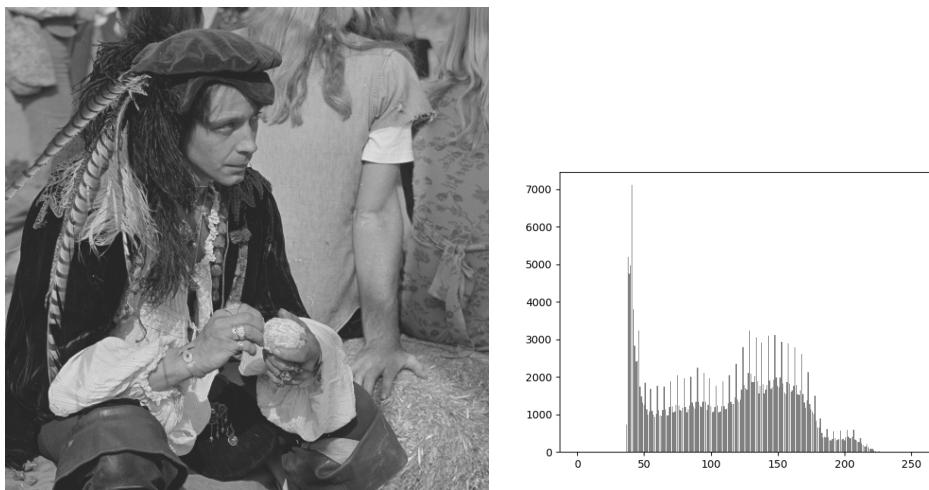
1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksla:
 3. Oblicz wartość progową jako średnią wartość piksli w otoczeniu danego piksla.
 4. Jeśli wartość piksla środkowego jest < 0 :
 5. Przypisz mu wartość 0.
 6. W przeciwnym przypadku przypisz mu wartość 255.



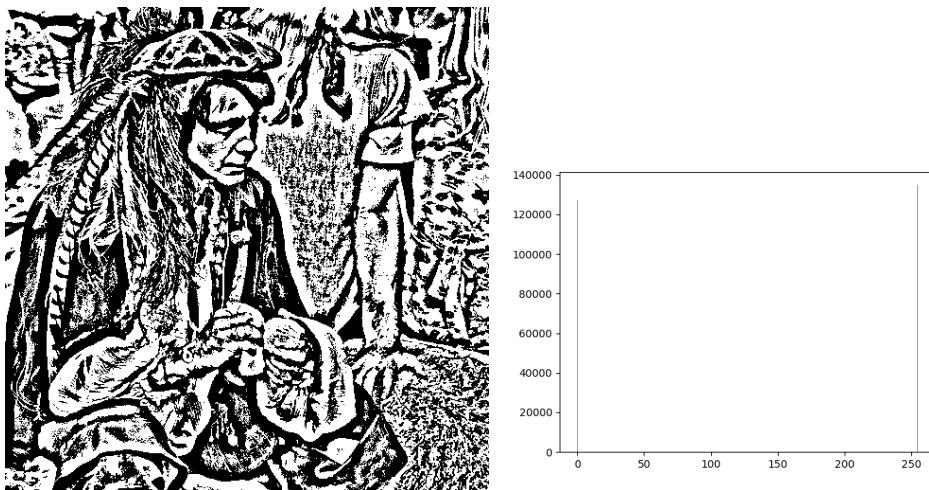
Rysunek 6.11: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.12: Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu



Rysunek 6.13: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.14: Obraz po progowaniu z parametrem 21, histogram szarości tego obrazu

Listing 6.4: Progowanie lokalne

```

def localThreshold(self, dim = 3, show = False, plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc
    l, r = -(int(round(dim / 2))), int(round(dim / 2) + 1)  # wsp
    . sasiadow

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width), dtype=np.uint8)

    # progowanie lokalne
    for i in range(height):

```

```
for j in range(width):
    n = 0
    threshold = 0
    currPix = self.im[i, j]
    for iOff in range(l, r):
        for jOff in range(l, r):
            iSafe = i if ((i + iOff) > (height + 1)) else (i + iOff)
            jSafe = j if ((j + jOff) > (width + 1)) else (j + jOff)
            threshold += self.im[iSafe, jSafe]
            n += 1
    threshold = int(round(threshold / n))
    resultImage[i, j] = 0 if (currPix < threshold) else 255

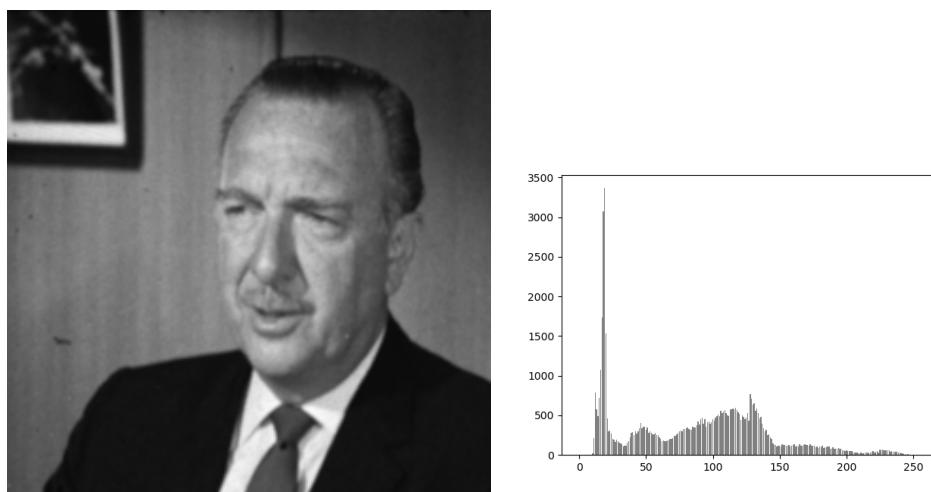
if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "locThreshold")
```

6.5 Progowanie globalne

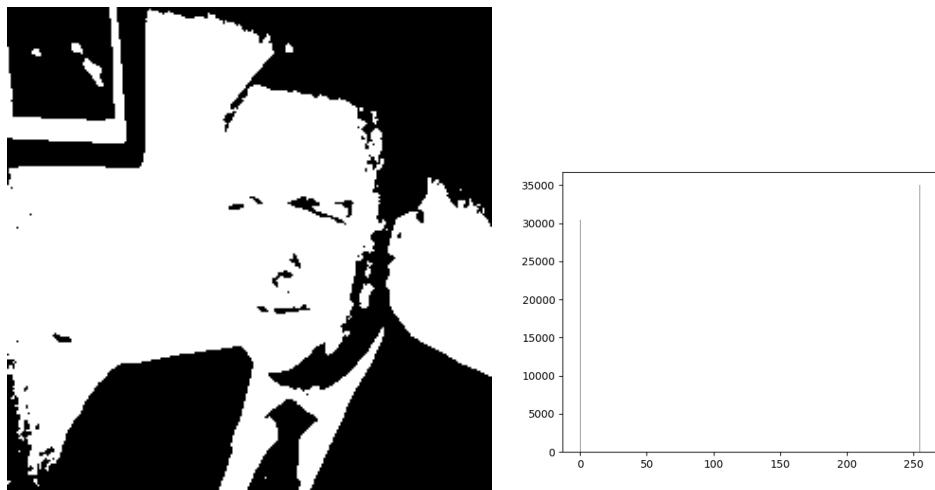
Opis algorytmu

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

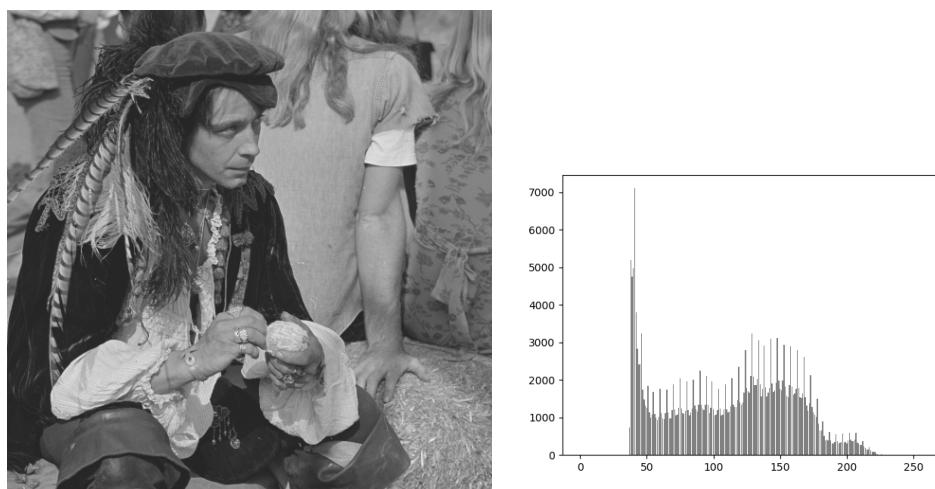
1. Oblicz wartość progową T , jako średnią wartość z wszystkich pikseli w obrazie
2. Dla każdego piksla:
 3. Jeśli wartość danego piksla jest $< T$:
 4. przypisz mu wartość 0.
 5. W przeciwnym przypadku przypisz mu wartość 255.



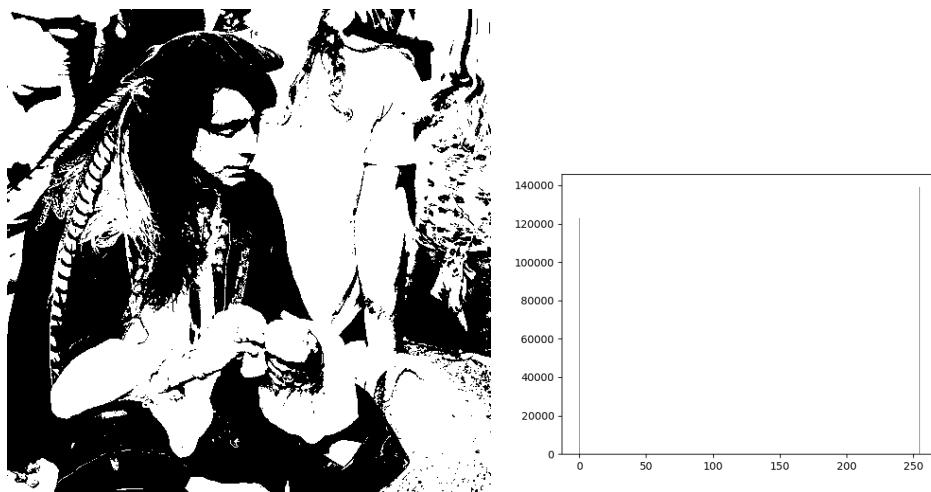
Rysunek 6.15: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.16: Obraz po progowaniu globalnym, histogram szarości tego obrazu



Rysunek 6.17: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 6.18: Obraz po progowaniu globalnym, histogram szarości tego obrazu

Listing 6.5: Progowanie globalne

```

def globalThreshold(self, show = False, plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width), dtype=np.uint8)

    # prog globalny
    threshold = 0
    n = 0
    for i in range(height):
        for j in range(width):
            threshold += self.im[i, j]
            n += 1
        threshold = int(round(threshold / n))

    # binaryzacja
    for i in range(height):
        for j in range(width):
            resultImage[i, j] = 0 if (self.im[i, j] < threshold) else
                255

    if show:
        self.show(Image.fromarray(resultImage, "L"))
        self.calculate(plot, resultImage)
        self.save(resultImage, self.imName, "globThreshold")

```

Rozdział 7

Operacje na histogramie obrazu barwowego

Histogram to najprostszy opis całościowy obrazu. Dlatego stosuje się go, by rozpoznać, jakie dalsze metody i operacje należy zastosować na przetwarzanym obrazie, by osiągnąć założony cel. Jego obliczenie polega na odczytaniu barwy każdego piksla i rejestraniu jej wystąpienia o danym poziomie.

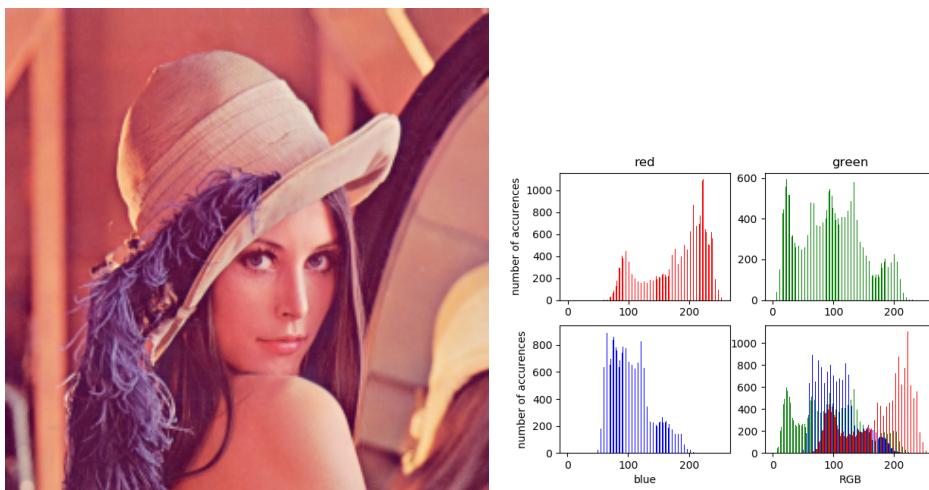
Histogram to funkcja przypisująca każdemu poziomowi barwy obrazu, liczbę piksli z danym poziomem barwy, czyli jest to wykres częstości występowania wartości piksli w obrazie.

7.1 Obliczanie histogramu

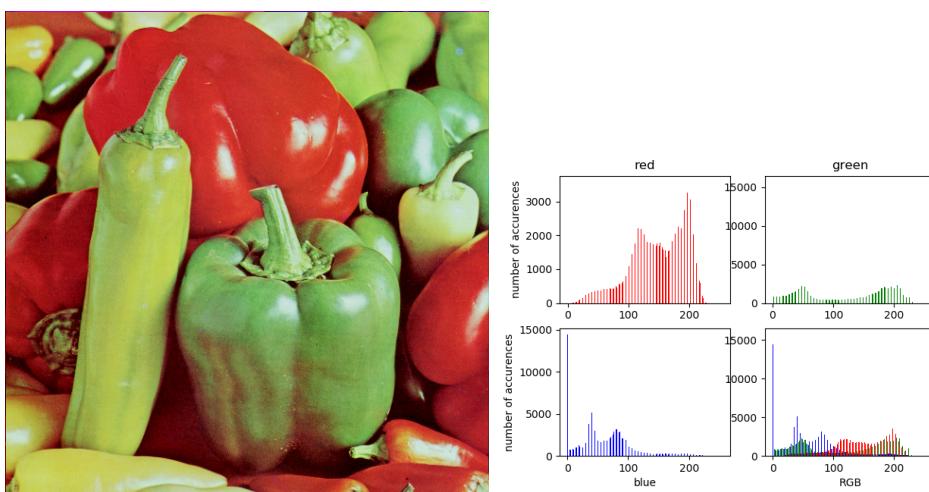
Opis algorytmu

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

1. Zaalokuj 3 tablice 256 elementowe (tyle, ile poziomów barw w obrazie)
2. Dla każdego piksela:
3. Dla każdej barwy:
4. Zwięksź element tablicy danej barwy o indeksie równym poziomie tej barwy danego piksela



Rysunek 7.1: Obraz barwny, histogram barw tego obrazu



Rysunek 7.2: Obraz barwny, histogram barw tego obrazu

Listing 7.1: Obliczanie histogramu

```
def calculate(self, plot = False, image = None):
    if image is None:
        image = self.im

    width = image.shape[1]           # szerokosc
    height = image.shape[0]          # wysokosc
    hist = [0] * 3                  # histogram RGB
    hist[0] = [0] * 256              # histogram R
    hist[1] = [0] * 256              # histogram G
    hist[2] = [0] * 256              # histogram B

    for i in range(height):
        for j in range(width):
            bin = image[i, j]
            for k in range(3):
                hist[k][bin[k]] += 1

    if plot:
        # tablica [0, 1, ..., 254, 255]
        bins = np.arange(256)
        self.plotHistogram(bins, hist)

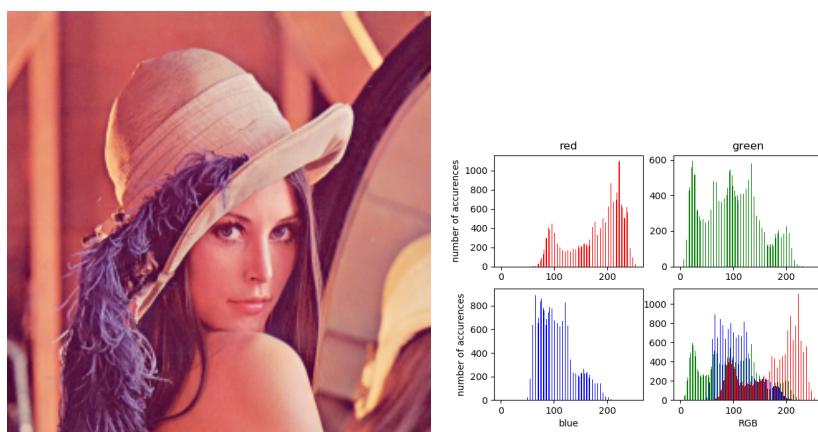
    return hist                      # [0] - hist R, [1] - hist G, [2]
                                    - hist B
```

7.2 Przemieszczanie histogramu

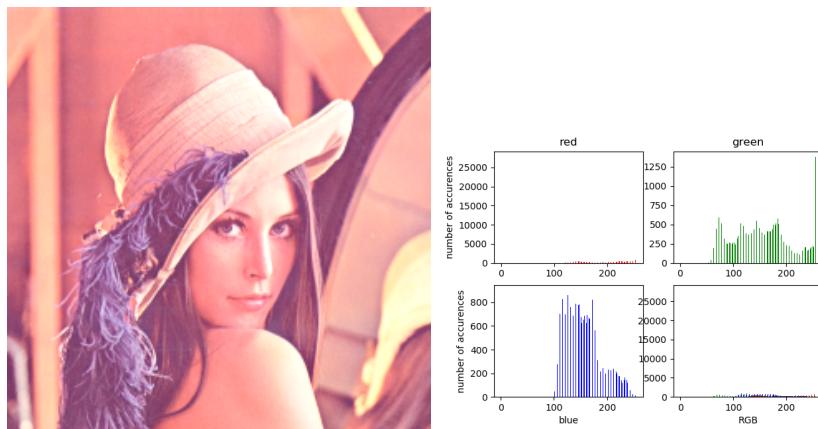
Opis algorytmu

Przemieszczanie histogramu polega na dodaniu lub odjęciu tej samej wartości od poziomu każdej z barw każdego piksla w obrazie. W rezultacie obraz jest odpowiednio równomiernie rozjaśniony bądź przyciemniony. Nie można przekroczyć przyjętego zakresu poziomu barwy.

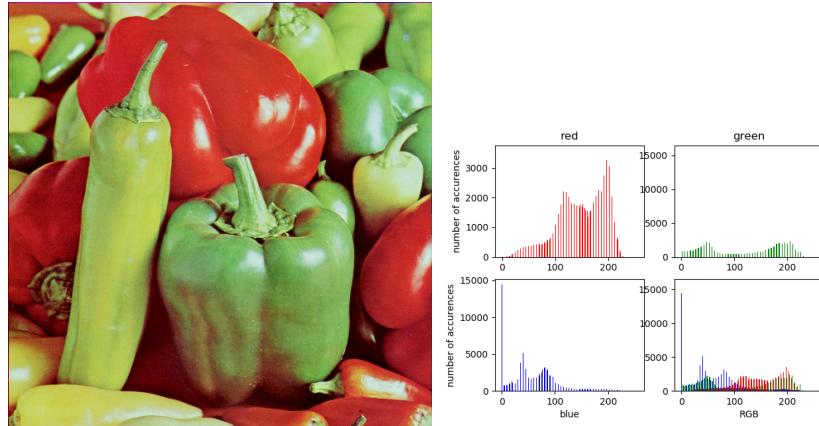
1. Do każdej wartości barwy piksla dodaj podaną stałą, o której chcesz przemieścić histogram.
2. Jeśli wartość barwy piksla po operacji dodawania wykracza poza zakres 255: Przypisz jej wartość 255.
3. Jeśli wartość piksla po operacji dodawania jest mniejsza od 0: Przypisz jej wartość 0.



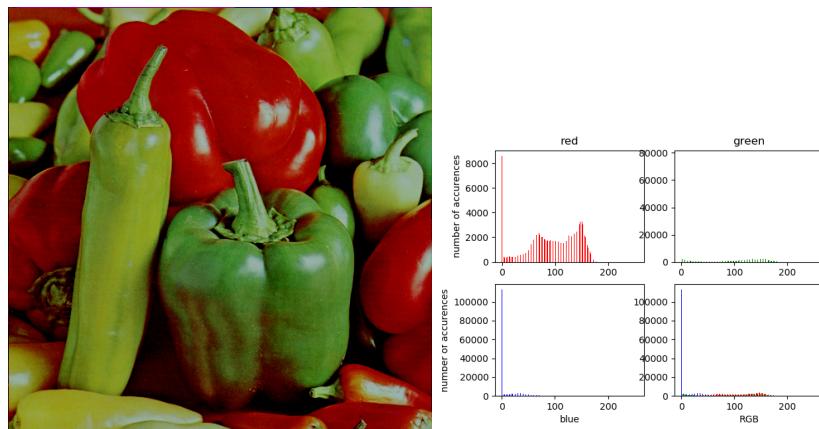
Rysunek 7.3: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.4: Obraz wyjściowy przesunięty o 50, histogram barw tego obrazu



Rysunek 7.5: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.6: Obraz wyjściowy przesunięty o -50, histogram barw tego obrazu

Listing 7.2: Przemieszczanie histogramu

```

def move( self , const = 0 , show = False , plot = False ) :
    width = self .im .shape [1]      # szereoksc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    # przemieszczanie
    for i in range( height ) :
        for j in range( width ) :
            value = self .im [i , j]
            for k in range( len( value )) :
                v = value [k]
                v += const

```

```
if v < 0:  
    v = 0  
elif v > 255:  
    v = 255  
value[k] = v  
resultImage[i, j] = value  
  
if show:  
    self.show(Image.fromarray(resultImage, "RGB"))  
    self.calculate(plot, resultImage)  
    self.save(resultImage, self.imName, "moveHist")
```

7.3 Rozciąganie histogramu

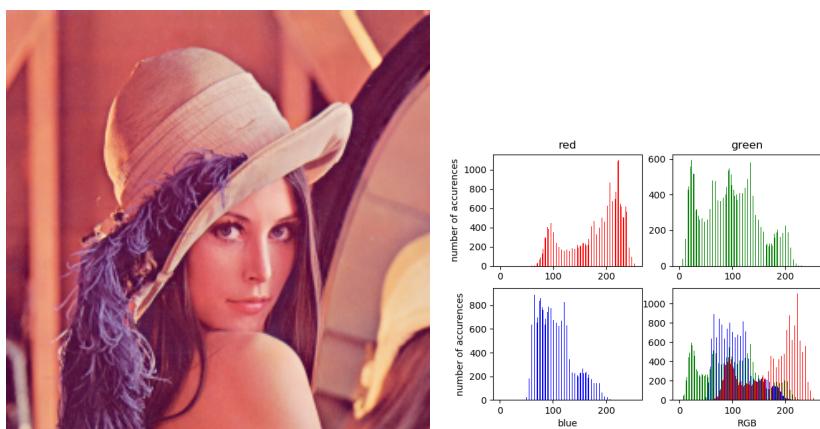
Opis algorytmu

Rozciągania histogramu dokonuje się na obrazie, którego poziomy barw nie są rozpięte na cały możliwy zakres np. [51, 233]. Operacja rozciągnięcia histogramu rozciągnie histogram tak, aby był rozpięty na cały możliwy zakres poziomów barw np. [0, 255].

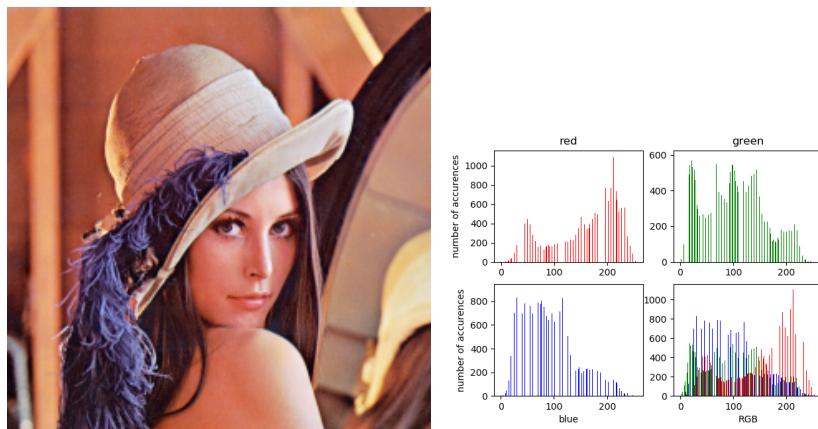
1. Znajdź w obrazie największą(\max_c) i najmniejszą(\min_c) wartość piksla dla każdej z barw(c)
2. Dla każdego piksla(P_o):
3. Dla każdej z barw(c):
4. Oblicz nową wartość piksla(P_n) stosując wzór:

$$P_n = 255 // (\max_c - \min_c) * (P_o - \min_c).$$

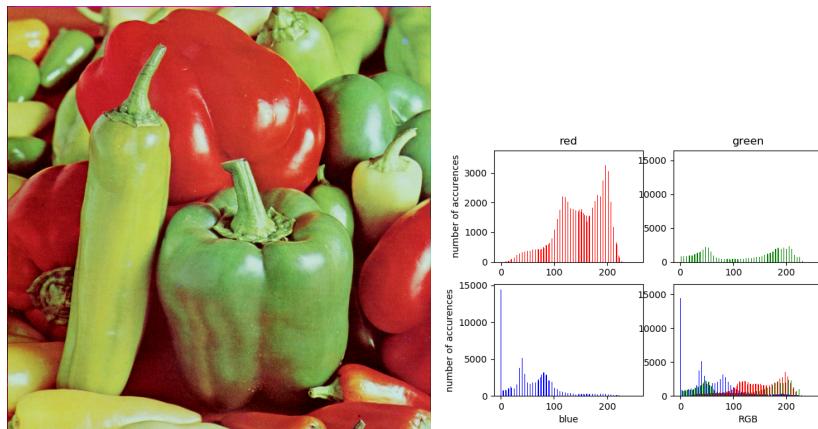
W taki sposób, jeśli barwy obrazu wejściowego były w zakresie np. [12, 239], po operacji rozciągania histogramu, będą w zakresie [0, 255].



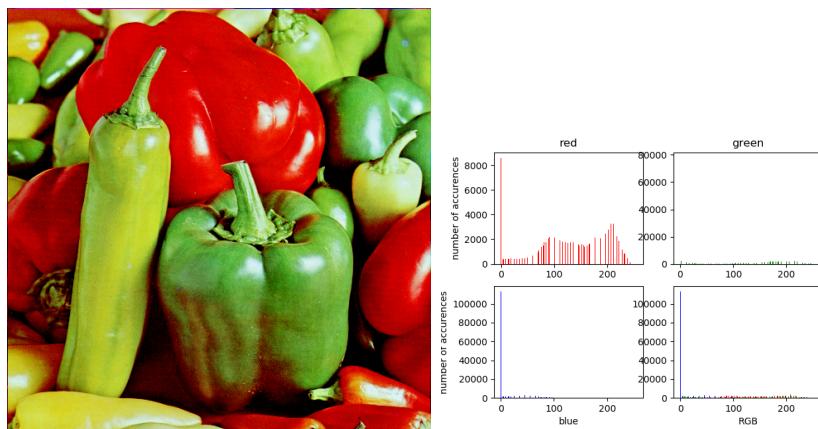
Rysunek 7.7: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.8: Obraz po rozciagnięciu, histogram barw tego obrazu



Rysunek 7.9: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.10: Obraz po rozciagnięciu, histogram barw tego obrazu

Listing 7.3: Rozciąganie histogramu

```
def stretch(self, show = False, plot = False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    for i in range(height):
        for j in range(width):
            resultImage[i, j] = self.im[i, j]

    # wartosci min i max w obrazie
    maxValue = [0] * 3
    minValue = [255] * 3
    while (maxValue[0] != 255) & (maxValue[1] != 255) & (maxValue[2] != 255):
        # wartosci max i min w obrazie
        for i in range(height):
            for j in range(width):
                currValue = resultImage[i, j]
                for k in range(3):
                    maxValue[k] = max(maxValue[k], currValue[k])
                    minValue[k] = min(minValue[k], currValue[k])

    # rozciaganie
    for i in range(height):
        for j in range(width):
            pix = resultImage[i, j]
            for k in range(3):
                pix[k] = ((255 / (maxValue[k] - minValue[k])) * (pix[k] - minValue[k]))
            resultImage[i, j] = pix

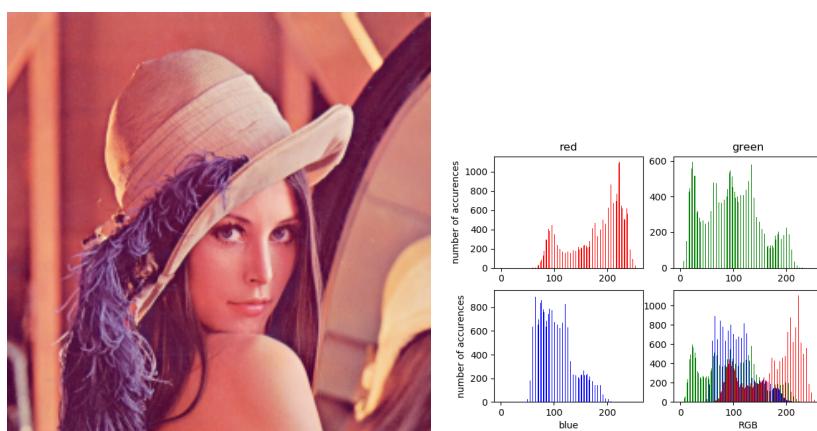
    if show:
        self.show(Image.fromarray(resultImage, "RGB"))
    self.calculate(plot, resultImage)
    self.save(resultImage, self.imName, "stretchHist")
```

7.4 Progowanie 1-progowe lokalne

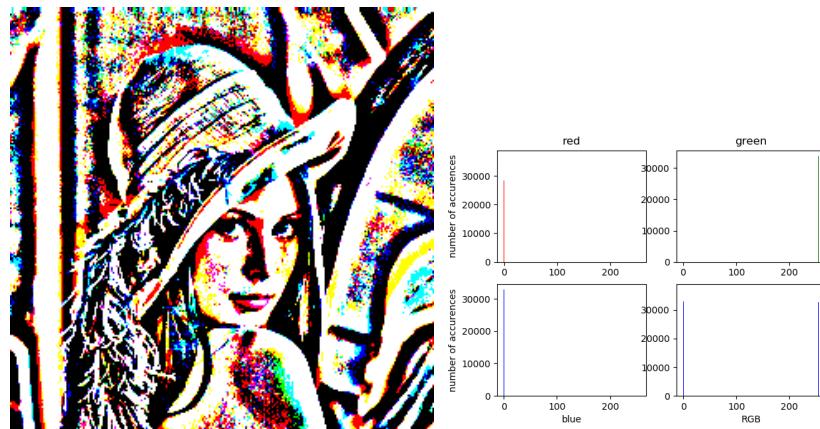
Opis algorytmu

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

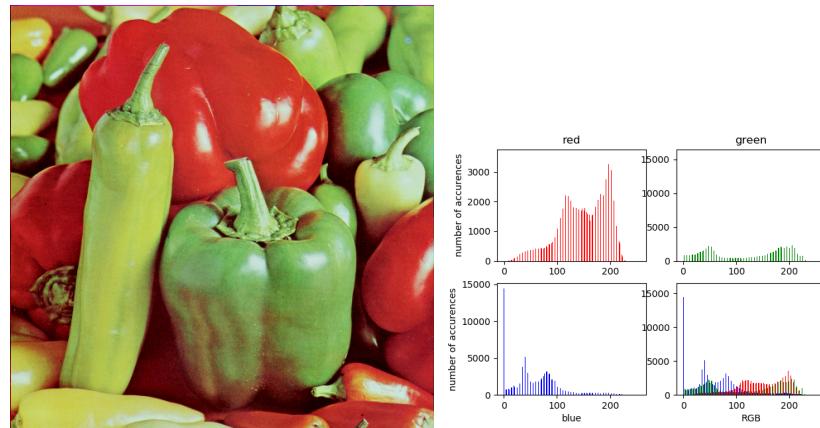
1. Zdefiniuj wielkość otoczenia piksła (musi być nieparzysta, po to, aby mógł istnieć piksel środkowy).
2. Dla każdego piksła(P):
3. Dla każdego kanału(C):
4. Oblicz wartość progową T_C jako średnią wartość piksli P_C w otoczeniu danego piksła(P).
5. Jeśli wartość piksła P_C jest $< T_C$:
6. Przypisz mu wartość 0.
7. W przeciwnym przypadku przypisz mu wartość 255.



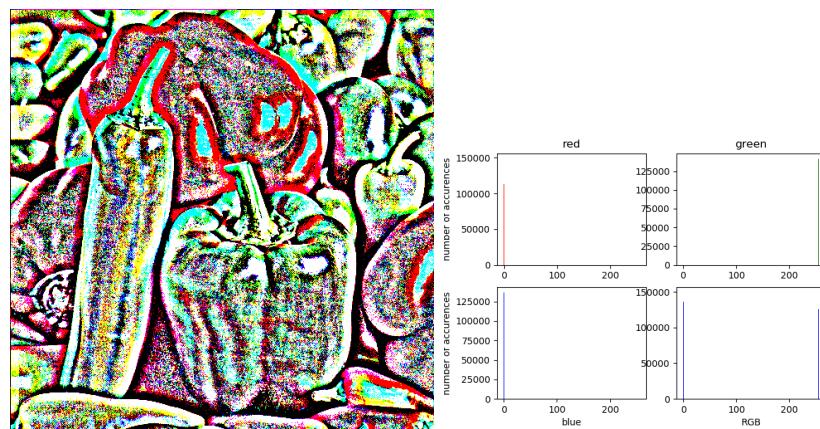
Rysunek 7.11: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 7.12: Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu



Rysunek 7.13: Obraz wejściowy, histogram szarości tego obrazu



Rysunek 7.14: Obraz po progowaniu z otoczeniem piksla 21x21, histogram szarości tego obrazu

Listing 7.4: Progowanie 1-progowe lokalne

```

def localSingleThreshold(self, dim = 3, show = False, plot =
    False):
    width = self.im.shape[1]      # szereokosc
    height = self.im.shape[0]     # wysokosc
    low, up = -(int(dim / 2)), (int(dim / 2) + 1) # wsp.
        sasiadow

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width, 3), dtype=np.uint8)
    tmp = np.empty((height, width, 3))
    tmp2 = np.empty((height, width, 3))

    # progowanie lokalne
    for i in range(height):
        for j in range(width):
            n = 0
            r = 0
            g = 0
            b = 0
            currPix = self.im[i, j]
            for iOff in range(low, up):
                for jOff in range(low, up):
                    iSafe = i if ((i + iOff) > (height + low)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width + low)) | ((j +
                        jOff) < 0) else (j + jOff)
                    r += int(self.im[iSafe, jSafe][0])
                    g += int(self.im[iSafe, jSafe][1])
                    b += int(self.im[iSafe, jSafe][2])
                    n += 1
            r = int(round(r / n))
            g = int(round(g / n))
            b = int(round(b / n))
            resultImage[i, j] = (0 if (currPix[0] < r) else 255, 0 if
                (currPix[1] < g) else 255, 0 if (currPix[2] < b)
                else 255)

    if show:
        self.show(Image.fromarray(resultImage, "RGB"))
        self.calculate(plot, resultImage)
        self.save(resultImage, self.imName, "localSingleThreshold")

```

7.5 Progowanie wielo-progowe lokalne

Opis algorytmu

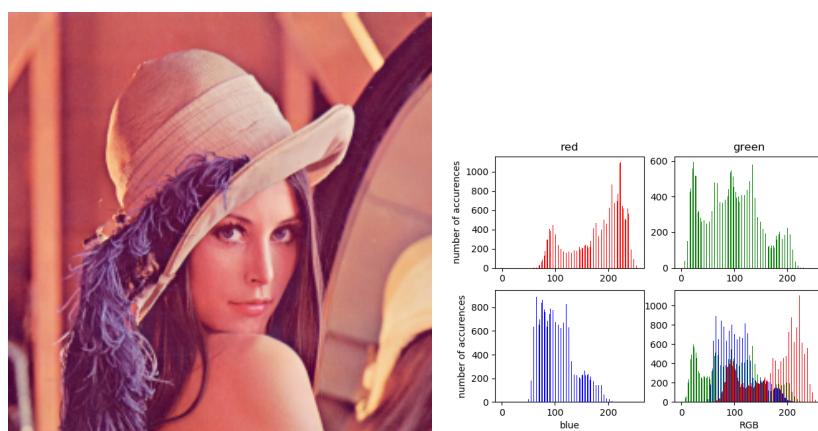
Progowanie wielo-progowe lokalne oblicza wartości progowe dla każdego piksla z osobna. W wyniku takiego progowania obraz ma mniejszą ilość kolorów w obrazie. Progi obliczane są dla każdego kanału z osobna.

1. Zdefiniuj wielkość otoczenia piksla (musi być nieparzysta, po to aby mógł istnieć piksel środkowy).
2. Zdefiniuj ilość progów(T).
3. Dla każdego piksla(P):
4. Dla każdego kanału(C):
5. Znajdź $MAX(P_C)$ i $MIN(P_C)$.
6. Oblicz skalę(S_C) wg. wzoru:

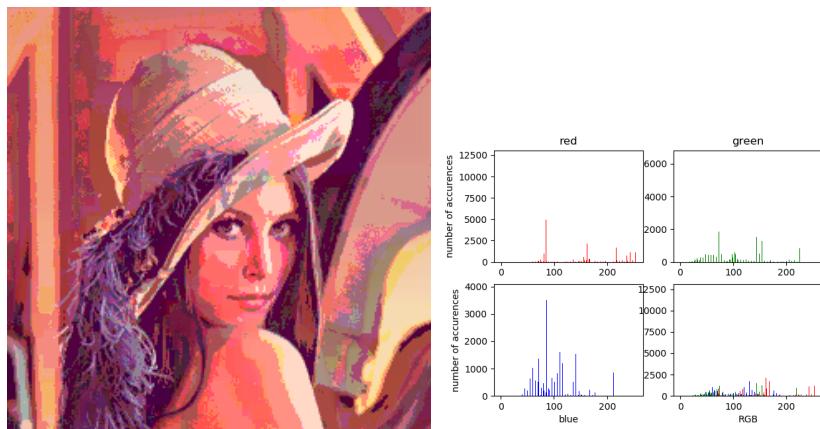
$$S_C = \frac{MAX(P_C)}{(T-1)}.$$

7. Jeśli $S_C = 0$:
8. Przypisz S_C wartość 1 (aby uniknąć dzielenia przez 0).
9. Wylicz nową wartość piksla(P_{C_n}) wg. wzoru:

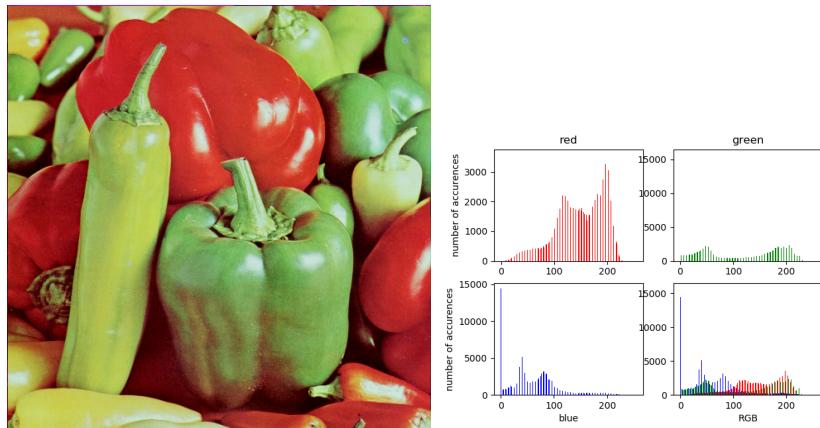
$$P_{C_n} = \lceil \frac{P_C}{S_C} \rceil * S_C.$$



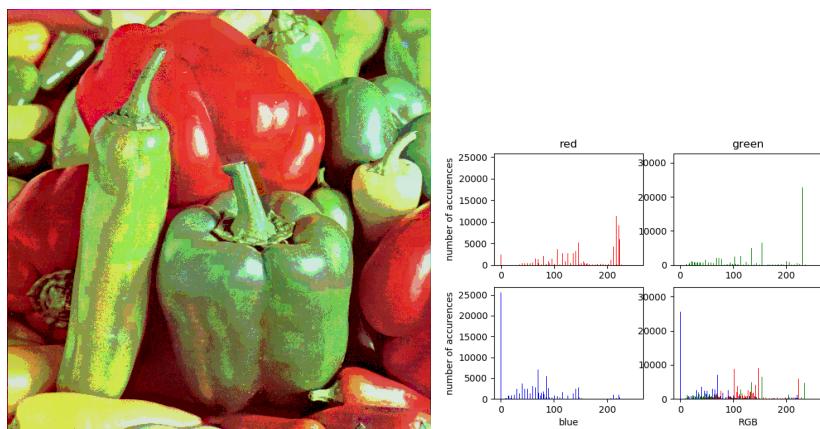
Rysunek 7.15: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.16: Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu



Rysunek 7.17: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.18: Obraz po progowaniu lokalnym (okno 21x21, progi 4), histogram barw tego obrazu

Listing 7.5: Progowanie wielo-progowe lokalne

```

def localMultiThreshold( self , dim=3, bins=4, show=False , plot=
    False ):

    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc
    low , up = -(int(dim / 2)) , (int(dim / 2) + 1)  # wsp.
        # sasiadow

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)

    # progowanie lokalne
    for i in range(height):
        for j in range(width):
            n = 0
            r = 0
            g = 0
            b = 0
            currPix = self .im [i , j]
            maxValue = [0] * 3
            minValue = [255] * 3
            for iOff in range(low , up):
                for jOff in range(low , up):
                    iSafe = i if ((i + iOff) > (height + low)) | ((i +
                        iOff) < 0) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width + low)) | ((j +
                        jOff) < 0) else (j + jOff)
                    currValue = self .im [iSafe , jSafe]
                    for k in range(3):
                        maxValue [k] = max(maxValue [k] , currValue [k])
                        minValue [k] = min(minValue [k] , currValue [k])
                    scale = [0] * 3
                    for k in range(3):
                        scale [k] = maxValue [k] / (bins - 1)
                        if scale [k] == 0:
                            scale [k] = 1
                    for k in range(3):
                        v = int(round(currPix [k] / scale [k])) * scale [k]
                        currPix [k] = v
                    resultImage [i , j] = currPix

    if show:
        self .show(Image .fromarray(resultImage , "RGB"))

```

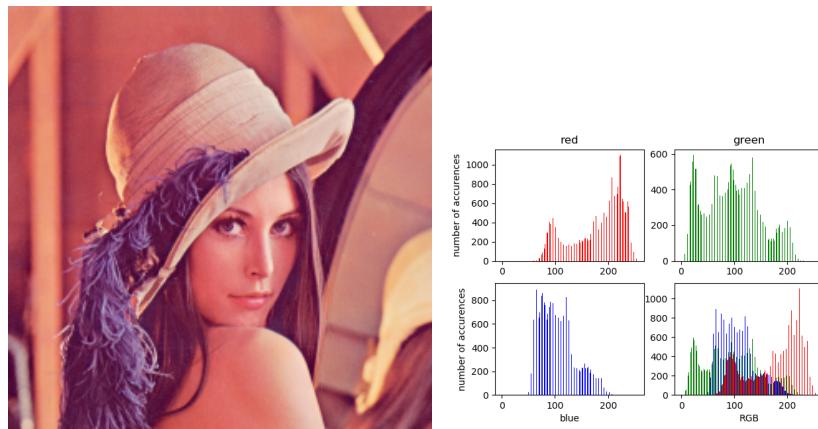
```
self.calculate(plot, resultImage)
self.save(resultImage, self.imName, "localMultiThreshold")
```

7.6 Progowanie 1-progowe globalne

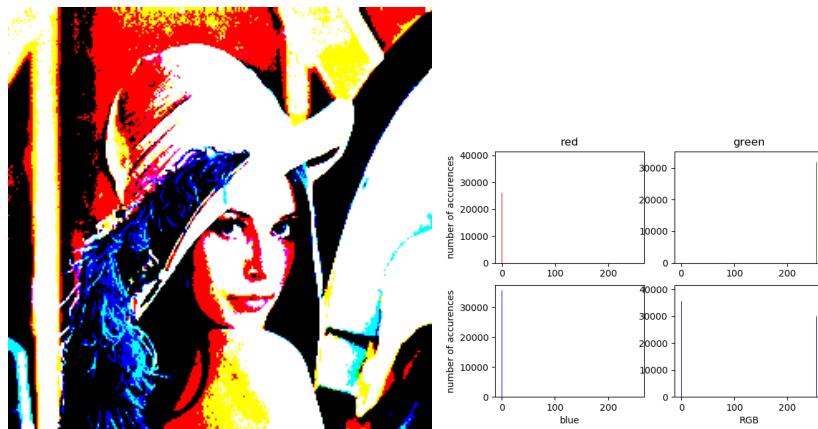
Opis algorytmu

W progowaniu 1-progowym globalnym wartość progowa jest ustalana globalnie dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa, jest stosowana do nadania każdemu pikslowi nową wartość.

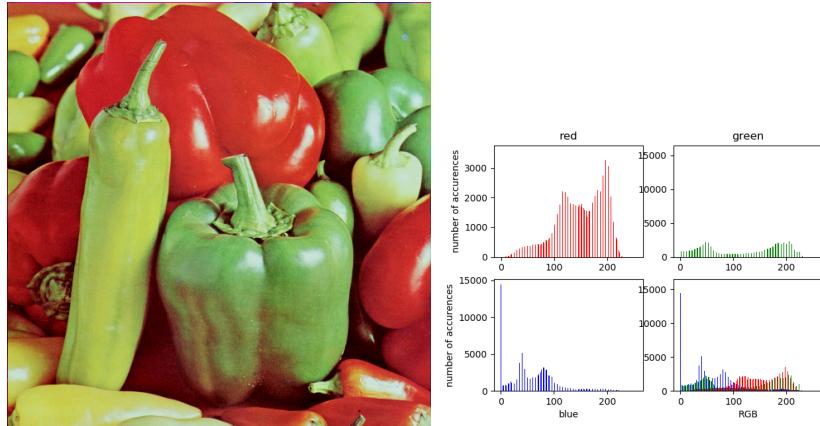
1. Dla każdego kanału(C):
2. Oblicz wartość progową T_C , jako średnią wartość z wszystkich pikseli P_C w obrazie.
3. Dla każdego piksela(P):
4. Dla każdego kanału(C):
5. Jeśli wartość P_C danego piksela jest $< T_C$:
6. przypisz mu wartość 0.
7. W przeciwnym przypadku przypisz mu wartość 255.



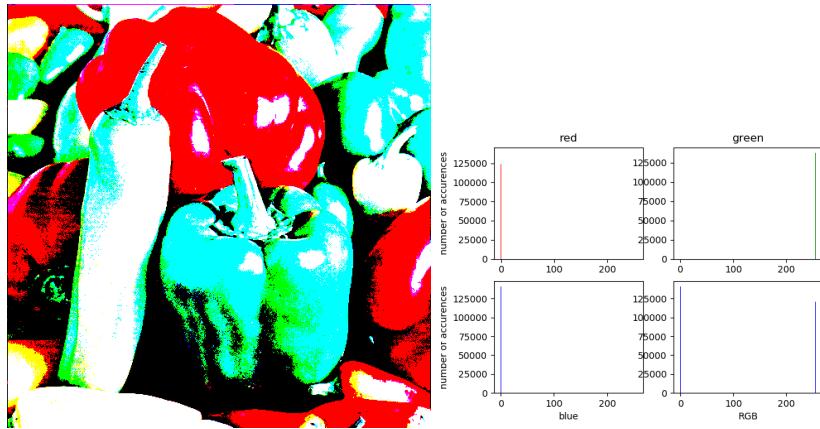
Rysunek 7.19: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.20: Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu



Rysunek 7.21: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.22: Obraz po progowaniu 1-progowym globalnym, histogram barw tego obrazu

Listing 7.6: Progowanie 1-progowe globalne

```

def globalSingleThreshold(self , show = False , plot = False) :
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)
tmp = np.empty((height , width , 3))
tmp2 = np.empty((height , width , 3))

# prog globalny
globalR = 0
globalG = 0
globalB = 0
nR = 0

```

```
nG = 0
nB = 0
for i in range(height):
    for j in range(width):
        globalR += self.im[i, j][0]
        nR += 1
        globalG += self.im[i, j][1]
        nG += 1
        globalB += self.im[i, j][2]
        nB += 1
    globalR = int(round(globalR / nR))
    globalG = int(round(globalG / nG))
    globalB = int(round(globalB / nB))

# kwantzyacja
for i in range(height):
    for j in range(width):
        resultImage[i, j] = (0 if (self.im[i, j][0] < globalR)
                           else 255, 0 if (self.im[i, j][1] < globalG) else 255,
                           0 if (self.im[i, j][2] < globalB) else 255)

if show:
    self.show(Image.fromarray(resultImage, "RGB"))
self.calculate(plot, resultImage)
self.save(resultImage, self.imName, "globalSingleThreshold")
```

7.7 Progowanie wielo-progowe globalne

Opis algorytmu

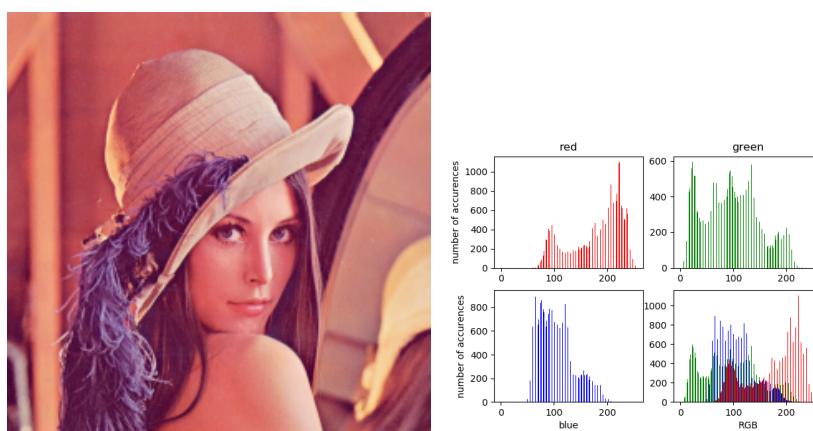
W progowaniu wielo-progowym globalnym wartości progowe są ustalane dla każdego kanału z osobna, biorąc pod uwagę wartość każdego piksla w obrazie. Następnie, tak wyliczona wartość progowa, jest stosowana do nadania każdemu pikselowi nowej wartości.

1. Zdefiniuj ilość progów(T).
2. Dla każdego piksla(P):
3. Dla każdego kanału(C):
4. Znajdź $MAX(P_C)$ i $MIN(P_C)$.
5. Oblicz skalę(S_C) wg. wzoru:

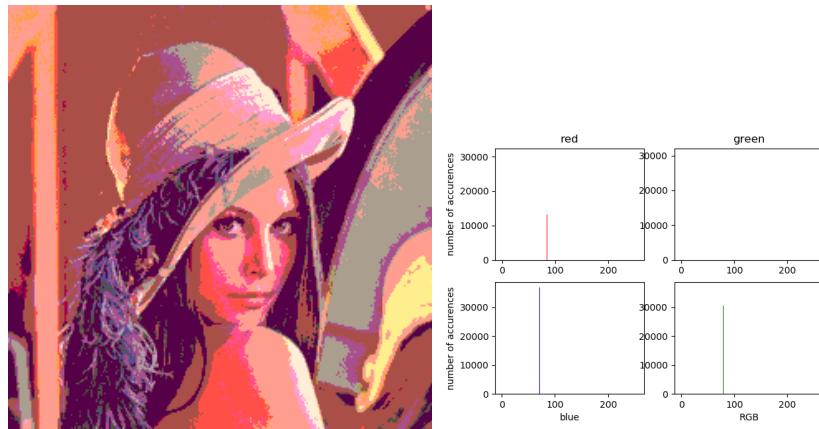
$$S_C = \frac{MAX(P_C)}{(T-1)}.$$

6. Dla każdego piksla(P):
7. Dla każdego kanału(C):
8. Wylicz nową wartość piksla(P_{C_n}) wg. wzoru:

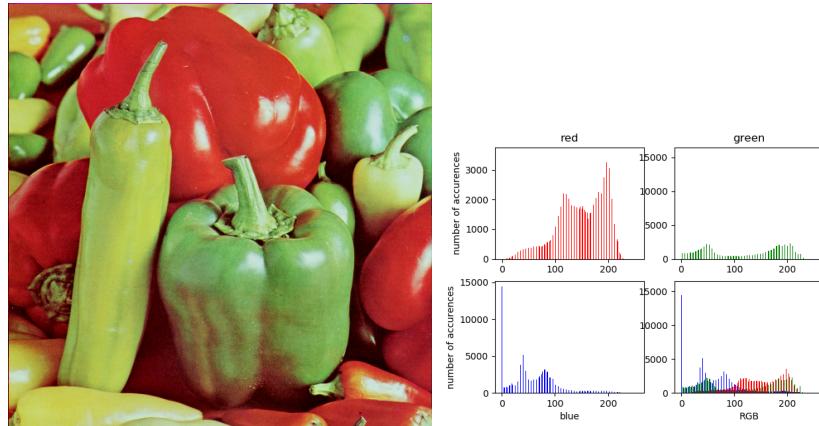
$$P_{C_n} = \lceil \frac{P_C}{S_C} \rceil * S_C.$$



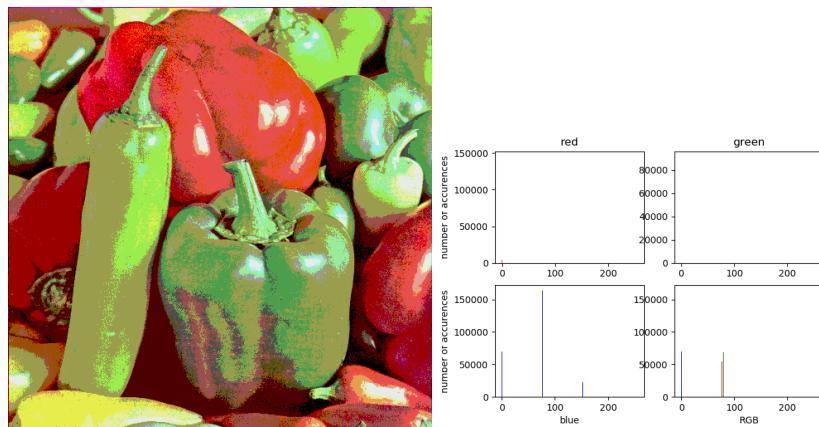
Rysunek 7.23: Obraz wejściowy, histogram barw tego obrazu



Rysunek 7.24: Obraz po progowaniu wielo-progowym globalnym (progi 4), histogramy barw tego obrazu



Rysunek 7.25: Obraz wejściowy, histogramy barw tego obrazu



Rysunek 7.26: Obraz po progowaniu wielo-progowym globalnym (progi 4), histogramy barw tego obrazu

Listing 7.7: Progowanie wielo-progowe globalne

```

def globalMultiThreshold(self , bins = 4, show = False , plot =
False):
    width = self.im.shape[1]      # szereoksc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)

maxValue = [0] * 3
minValue = [255] * 3
# wartosci max i min w obrazie
for i in range(height):
    for j in range(width):
        currValue = self.im[i , j]
        for k in range(3):
            maxValue[k] = max(maxValue[k] , currValue[k])
            minValue[k] = min(minValue[k] , currValue[k])

scale = [0] * 3
for k in range(3):
    scale[k] = maxValue[k] / (bins - 1)

for i in range(height):
    for j in range(width):
        pix = self.im[i , j]
        for k in range(3):
            pix[k] = int(round(pix[k] / scale[k])) * scale[k]
        resultImage[i , j] = pix

if show:
    self.show(Image.fromarray(resultImage , "RGB"))
    self.calculate(plot , resultImage)
    self.save(resultImage , self.imName, "globalMultiThreshold")

```

Rozdział 8

Operacje morfologiczne na obrazach binarnych

W obrazie binarnym piksele mogą przybierać tylko dwie wartości. Zazwyczaj kodowane są za pomocą pojedynczego bitu i przyjmują wartość 0 lub 1. Spotyka się także reprezentacje wykorzystujące inne pary wartości: (0, 255), (-1, 1), (True, False).

W przedstawionych przykładach została użyta reprezentacja (0,255), gdzie 0 oznacza czerń a 255 biały.

W morfologicznym przetwarzaniu obrazów ważne jest określenie, kiedy dwa piksele sąsiadują ze sobą. W tym celu definiuje się dla każdego piksla jego sąsiedztwo. Przy implementacji wykorzystano sąsiedztwo czterospójne:

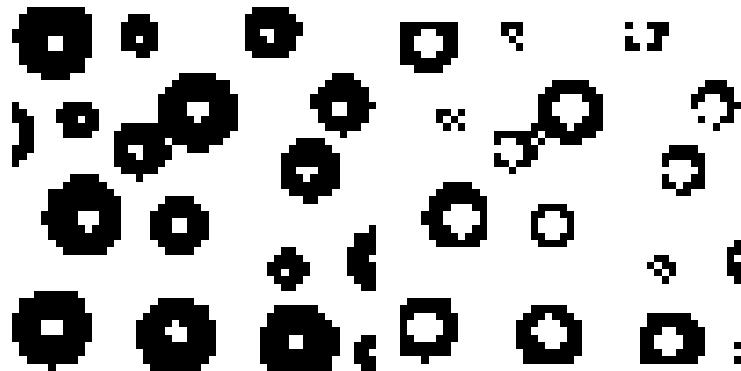
Sąsiedztwo czterospójne (von Neumanna) - obejmuje cztery piksele przyległe do danego z góry, dołu i po bokach

$$N_4(p) = ((x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y))$$

8.1 Okrawanie (erozja)

Przyjeto, że wartości wykraczające poza granice (wysokość/szerokość) obrazu są białe (mają wartość 255)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$ od piksla (x, y) .
3. Jeżeli którykolwiek z sąsiadów ma wartość równą 255 to środkowy piksel (x, y) ma przyjąć wartość 255.
4. Jeżeli wszyscy sąsiedzi mają wartość równą 0 to środkowy piksel (x, y) ma przyjąć wartość 0.



Rysunek 8.1: (Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)



Rysunek 8.2: (Od lewej) Obraz wejściowy (50x50), obraz po operacji okrawania (erozji)

Listing 8.1: Operacja okrawania (erozji) na obrazie binarnym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        # Przyjeto, ze wartosci wykraczajace poza granice
        # obrazu sa biale (maja wartosc 255)
        neighbour_pix = [255, 255, 255, 255]

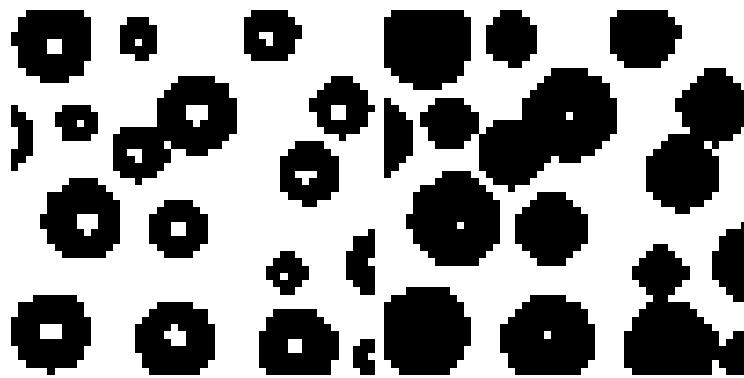
        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x][0])
    
```

```
if x + 1 < width:  
    neighbour_pix[2]=(image_matrix[y][x+1][0])  
if y + 1 < height:  
    neighbour_pix[3]=(image_matrix[y+1][x][0])  
  
if 255 in neighbour_pix:  
    result_matrix[y][x] = 255 #biały  
else:  
    result_matrix[y][x] = 0 #czarny
```

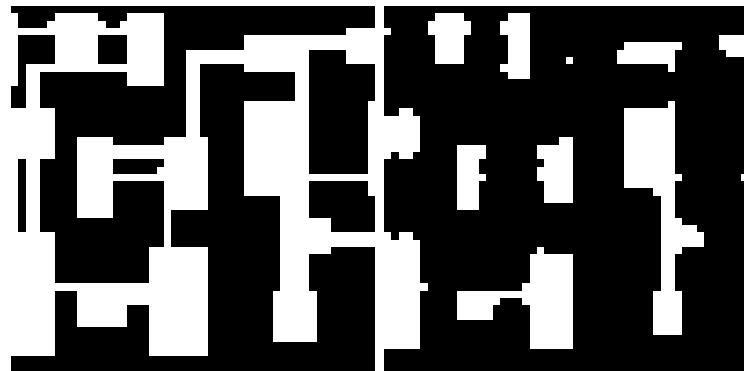
8.2 Nakładanie (dylatacja)

Przyjeto, że wartości wykraczające poza granice (wysokość/szerokość) obrazu są białe (maja wartość 255)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$ od piksla (x, y) .
3. Jeżeli którykolwiek z sąsiadów ma wartość równą 0 to środkowy piksel (x, y) ma przyjąć wartość 0.
4. Jeżeli wszyscy sąsiedzi mają wartość równą 255 to środkowy piksel (x, y) ma przyjąć wartość 255.



Rysunek 8.3: (Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)



Rysunek 8.4: (Od lewej) Obraz wejściowy (50x50), obraz po operacji nakładania (dylatacji)

Listing 8.2: Operacja nakładania (dylatacji) na obrazie binarnym

```
image_matrix = self.im1
```

```
width = image_matrix.shape[1]      # szereokosc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

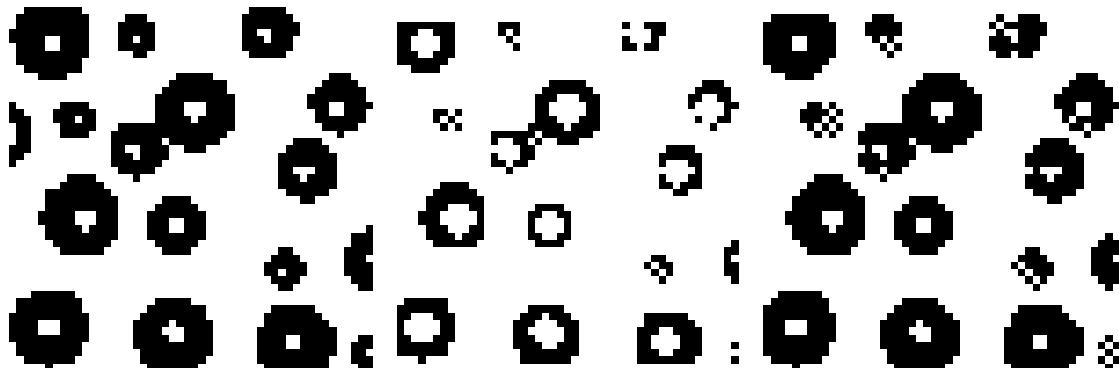
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1][0])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x][0])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1][0])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x][0])

        if 0 in neighbour_pix:
            result_matrix[y][x] = 0
        else:
            result_matrix[y][x] = 255
```

8.3 Otwarcie

Otwarcie morfologiczne jest równoważne nałożeniu operacji dylatacji na wynik erozji obrazu pierwotnego.



Rysunek 8.5: (Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)



Rysunek 8.6: (Od lewej) Obraz wejściowy (50x50), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)

Listing 8.3: Operacja otwarcia na obrazie binarnym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#erozja
for y in range(height):
    for x in range(width):
        if image_matrix[y][x] == 1:
            if image_matrix[y-1][x-1] == 1 and image_matrix[y-1][x] == 1 and image_matrix[y-1][x+1] == 1 and image_matrix[y][x-1] == 1 and image_matrix[y][x+1] == 1 and image_matrix[y+1][x-1] == 1 and image_matrix[y+1][x] == 1 and image_matrix[y+1][x+1] == 1:
                e_result_matrix[y][x] = 0
            else:
                e_result_matrix[y][x] = 1
        else:
            e_result_matrix[y][x] = 0

#dylatacja
for y in range(height):
    for x in range(width):
        if e_result_matrix[y][x] == 1:
            if e_result_matrix[y-1][x-1] == 1 and e_result_matrix[y-1][x] == 1 and e_result_matrix[y-1][x+1] == 1 and e_result_matrix[y][x-1] == 1 and e_result_matrix[y][x+1] == 1 and e_result_matrix[y+1][x-1] == 1 and e_result_matrix[y+1][x] == 1 and e_result_matrix[y+1][x+1] == 1:
                d_result_matrix[y][x] = 1
            else:
                d_result_matrix[y][x] = 0
        else:
            d_result_matrix[y][x] = 0
    
```

```

neighbour_pix = [255, 255, 255, 255]

if x - 1 > 0:
    neighbour_pix[0]=(image_matrix[y][x-1][0])
if y - 1 > 0:
    neighbour_pix[1]=(image_matrix[y-1][x-1][0])
if x + 1 < width:
    neighbour_pix[2]=(image_matrix[y][x+1][0])
if y + 1 < height:
    neighbour_pix[3]=(image_matrix[y+1][x+1][0])

if 255 in neighbour_pix:
    e_result_matrix[y][x] = 255
else:
    e_result_matrix[y][x] = 0

Image.fromarray(e_result_matrix).show()
#dylatacja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

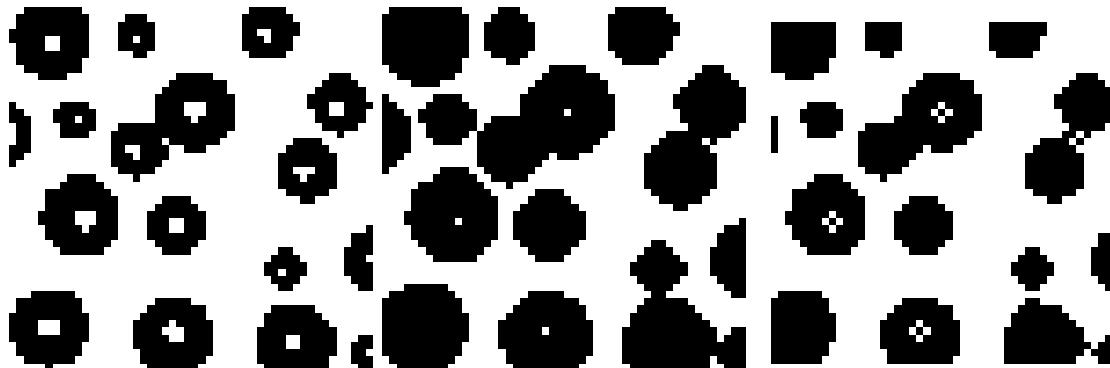
        if x - 1 > 0:
            neighbour_pix[0]=(e_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(e_result_matrix[y-1][x-1])
        if x + 1 < width:
            neighbour_pix[2]=(e_result_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(e_result_matrix[y+1][x+1])

        if 0 in neighbour_pix:
            d_result_matrix[y][x] = 0
        else:
            d_result_matrix[y][x] = 255

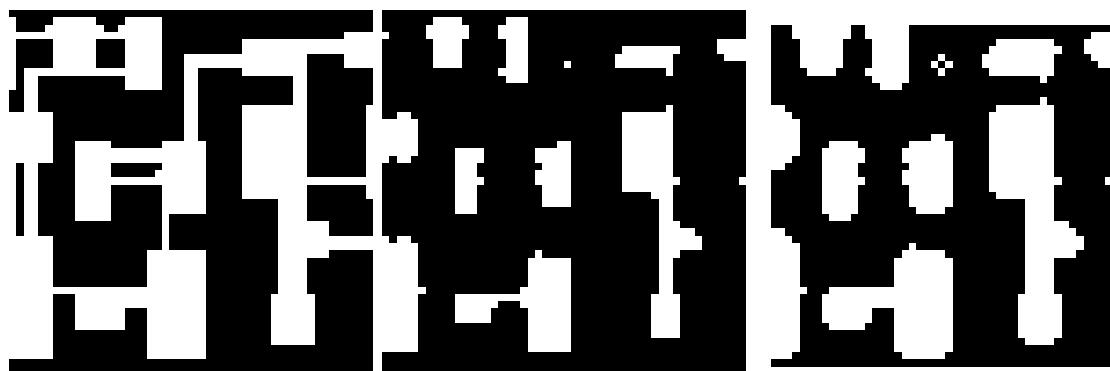
```

8.4 Zamknięcie .

Zamknięcie morfologiczne jest równoważne nałożeniu operacji erozji na wynik dylatacji obrazu pierwotnego.



Rysunek 8.7: (Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)



Rysunek 8.8: (Od lewej) Obraz wejściowy (50x50), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)

Kod źródłowy .

Listing 8.4: Operacja zamknięcia na obrazie binarnym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#dylatacja
for y in range(height):
    for x in range(width):
        if image_matrix[y][x] == 1:
            e_result_matrix[y][x] = 0
            for i in range(-1, 2):
                for j in range(-1, 2):
                    if (y + i) >= 0 and (y + i) < height and (x + j) >= 0 and (x + j) < width:
                        if image_matrix[y + i][x + j] == 1:
                            e_result_matrix[y][x] = 1
                            break
            if e_result_matrix[y][x] == 1:
                d_result_matrix[y][x] = 1
        else:
            d_result_matrix[y][x] = 0
```

```

for x in range(width):
    neighbour_pix = [255, 255, 255, 255]

    if x - 1 > 0:
        neighbour_pix[0]=(image_matrix[y][x
            -1][0])
    if y - 1 > 0:
        neighbour_pix[1]=(image_matrix[y-1][x
            ][0])
    if x + 1 < width:
        neighbour_pix[2]=(image_matrix[y][x
            +1][0])
    if y + 1 < height:
        neighbour_pix[3]=(image_matrix[y+1][x
            ][0])

    if 0 in neighbour_pix:
        d_result_matrix[y][x] = 0
    else:
        d_result_matrix[y][x] = 255

```

Image.fromarray(e_result_matrix).show()

```

#erozja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(d_result_matrix[y][x
                -1])
        if y - 1 > 0:
            neighbour_pix[1]=(d_result_matrix[y-1][
                x])
        if x + 1 < width:
            neighbour_pix[2]=(d_result_matrix[y][x
                +1])
        if y + 1 < height:
            neighbour_pix[3]=(d_result_matrix[y+1][
                x])

        if 255 in neighbour_pix:
            e_result_matrix[y][x] = 255

```

```
else:  
    e_result_matrix [y] [x] = 0
```

Rozdział 9

Operacje morfologiczne na obrazach szarych

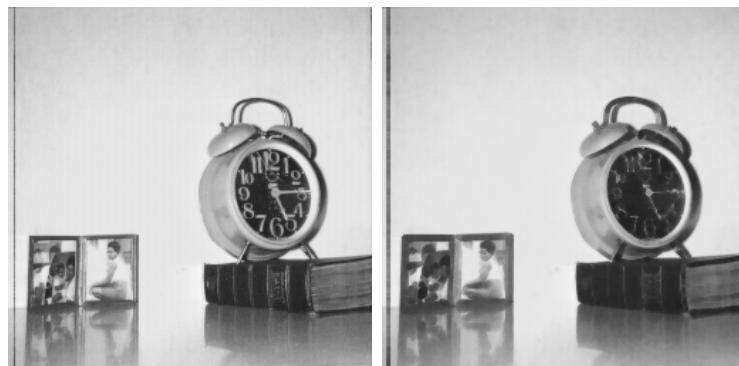
W morfologicznym przetwarzaniu obrazów ważne jest określenie, kiedy dwa piksele sąsiadują ze sobą. W tym celu definiuje się dla każdego piksla jego sąsiedztwo. Przy implementacji wykorzystano sąsiedztwo czterospójne:

Sąsiedztwo czterospójne (von Neumanna) - obejmuje cztery piksele przyległe do danego z góry, dołu i po bokach

$$N_4(p) = ((x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y))$$

9.1 Okrawanie (erozja)

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli do wektora $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$ od piksla (x, y) .
3. Posortuj wektor.
4. Przypisz wartość piksla o najniższej jasności do piksla (x, y) .



Rysunek 9.1: (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji okrawania (erozji)



Rysunek 9.2: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji okrawania (erozji)

Listing 9.1: Operacja okrawania (erozji) na obrazie szarym

```

image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)

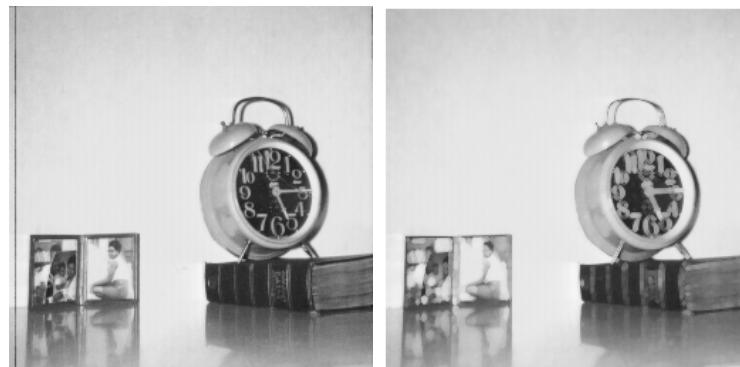
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        min_pix = min(neighbour_pix)
        result_matrix[y][x] = min_pix
    
```

9.2 Nakładanie (dylatacja) .

1. Dla wszystkich pikseli wykonaj:
2. Wczytaj wartości sąsiadujących pikseli do wektora $(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)$ od piksela (x, y) .
3. Posortuj wektor.
4. Przypisz wartość piksela o najwyższej jasności do piksela (x, y) .



Rysunek 9.3: (Od lewej) Obraz wejściowy (256x256), obraz (256x256) po operacji nakładania (dylatacji)



Rysunek 9.4: (Od lewej) Obraz wejściowy (512x512), obraz (512x512) po operacji nakładania (dylatacji)

Listing 9.2: Operacja nakładania (dylatacji) na obrazie szarym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

result_matrix = np.zeros((height, width), dtype=np.uint8)
```

```
print(image_matrix)
print(result_matrix)

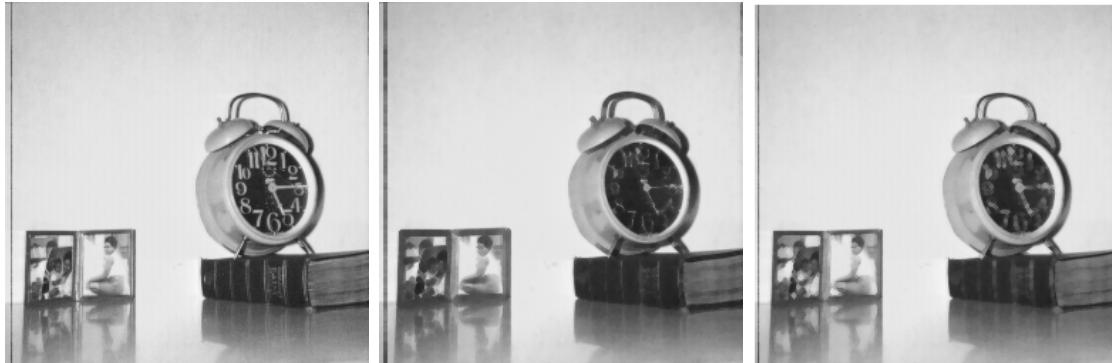
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(image_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(image_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(image_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(image_matrix[y+1][x])

        max_pix = max(neighbour_pix)
        result_matrix[y][x] = max_pix
```

9.3 Otwarcie .

Otwarcie morfologiczne jest równoważne nałożeniu operacji dylatacji na wynik erozji obrazu pierwotnego.



Rysunek 9.5: (Od lewej) Obraz wejściowy (256x256), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)



Rysunek 9.6: (Od lewej) Obraz wejściowy (512x512), obraz po dylacji, obraz po operacji otwarcia (erozja → dylatacja)

Listing 9.3: Operacja otwarcia na obrazie szarym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#erozja
for y in range(height):
    for x in range(width):
```

```

neighbour_pix = [255, 255, 255, 255]

if x - 1 > 0:
    neighbour_pix[0]=(image_matrix[y][x-1])
if y - 1 > 0:
    neighbour_pix[1]=(image_matrix[y-1][x])
if x + 1 < width:
    neighbour_pix[2]=(image_matrix[y][x+1])
if y + 1 < height:
    neighbour_pix[3]=(image_matrix[y+1][x])

min_pix = min(neighbour_pix)
e_result_matrix[y][x] = min_pix

Image.fromarray(e_result_matrix).show()
#dylatacja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

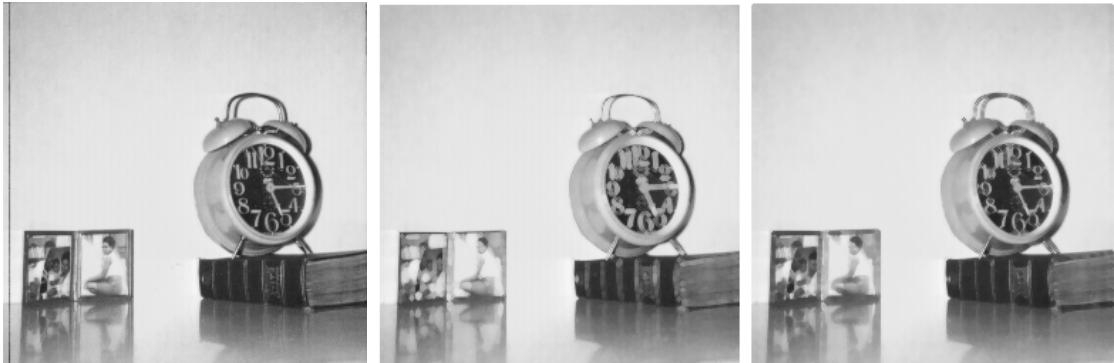
        if x - 1 > 0:
            neighbour_pix[0]=(e_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(e_result_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(e_result_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(e_result_matrix[y+1][x])

        max_pix = max(neighbour_pix)
        d_result_matrix[y][x] = max_pix

```

9.4 Zamknięcie .

Zamknięcie morfologiczne jest równoważne nałożeniu operacji erozji na wynik dylatacji obrazu pierwotnego.



Rysunek 9.7: (Od lewej) Obraz wejściowy (256x256), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)



Rysunek 9.8: (Od lewej) Obraz wejściowy (512x512), obraz po erozji, obraz po operacji zamknięcia (dylatacja → erozja)

Kod źródłowy .

Listing 9.4: Operacja zamknięcia na obrazie szarym

```
image_matrix = self.im1
width = image_matrix.shape[1]      # szereoksc
height = image_matrix.shape[0]     # wysokosc

e_result_matrix = np.zeros((height, width), dtype=np.uint8)
d_result_matrix = np.zeros((height, width), dtype=np.uint8)

#dylatacja
for y in range(height):
    for x in range(width):
        if image_matrix[y][x] > 0:
```

```

for x in range(width):
    neighbour_pix = [255, 255, 255, 255]

    if x - 1 > 0:
        neighbour_pix[0]=(image_matrix[y][x-1])
    if y - 1 > 0:
        neighbour_pix[1]=(image_matrix[y-1][x])
    if x + 1 < width:
        neighbour_pix[2]=(image_matrix[y][x+1])
    if y + 1 < height:
        neighbour_pix[3]=(image_matrix[y+1][x])

    max_pix = max(neighbour_pix)
    d_result_matrix[y][x] = max_pix

#erozja
for y in range(height):
    for x in range(width):
        neighbour_pix = [255, 255, 255, 255]

        if x - 1 > 0:
            neighbour_pix[0]=(d_result_matrix[y][x-1])
        if y - 1 > 0:
            neighbour_pix[1]=(d_result_matrix[y-1][x])
        if x + 1 < width:
            neighbour_pix[2]=(d_result_matrix[y][x+1])
        if y + 1 < height:
            neighbour_pix[3]=(d_result_matrix[y+1][x])

        min_pix = min(neighbour_pix)
        e_result_matrix[y][x] = min_pix

```

Rozdział 10

Filtrowanie liniowe i nieliniowe

Filtrowanie obrazów oznacza, że do obliczenia nowej wartości danego piksela brane są pod uwagę wartości pikseli z jego otoczenia. Każdy punkt z otoczenia wnosi swój wkład (wagę) podczas przeprowadzania końcowych obliczeń. Wagi te są zapisywane w postaci maski. Najczęściej spotykane rozmiary filtrów to maski o rozmiarach 3x3, 5x5. Ale filtry o rozmiarach 7x7, 9x9 również są nierzadko używane. Filtrowanie odbywa się w ten sposób, że maskę przemieszcza się w obrębie obrazu z krokiem równym odległości międzypiksowej. Poniżej przykład dla maski o wymiarach 3x3:

$$\begin{matrix} f_{-1,-1} & f_{0,-1} & f_{1,-1} \\ f_{-1,0} & f_{0,0} & f_{1,0} \\ f_{-1,1} & f_{0,1} & f_{1,1} \end{matrix}$$

By obliczyć nową wartość piksela należy wpierw obliczyć sumę ważoną składowych punktu oraz wszystkich sąsiadujących punktów zgodnie z wagami wskazanymi przez maskę.

$$s = f_{-1,-1} * a_{i-1,j-1} + f_{0,-1} * a_{i,j-1} + f_{1,-1} * a_{i+1,j-1} + f_{-1,0} * a_{i-1,j} + f_{0,0} * a_{i,j} + f_{1,0} * a_{i+1,j} + f_{-1,1} * a_{i-1,j+1} + f_{0,1} * a_{i,j+1} + f_{1,1} * a_{i+1,j+1}$$

Otrzymaną sumę dzielimy przez sumę wszystkich wag maski, jeżeli jest ona różna od 0.

$$a'_{i,j} = \frac{s}{f_{-1,-1} + f_{0,-1} + f_{1,-1} + f_{-1,0} + f_{0,0} + f_{1,0} + f_{-1,1} + f_{0,1} + f_{1,1}}$$

10.1 Filtr dolnoprzepustowy uśredniający

Opis algorytmu

Filtr uśredniający jest podstawowym filtrem dolnoprzepustowym, jego wynikiem jest uśrednienie każdego piksla razem ze swoimi sąsiadami. Maska:

$$\frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

1. Dla każdego piksla(P):
2. Dla każdej barwy:
3. Zsumuj wartości barwy piksli, otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Sumę barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikselowi P .



Rysunek 10.1: Obraz wejściowy, obraz uśredniony



Rysunek 10.2: Obraz wejściowy, obraz uśredniony



Rysunek 10.3: Obraz wejściowy, obraz uśredniony



Rysunek 10.4: Obraz wejściowy, obraz uśredniony

Listing 10.1: Filtr dolnoprzepustowy uśredniający (obraz szary)

```

def averageGray(self , show = False):
    width = self.im.shape[1]      # szereokosc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width) , dtype=np.uint8)

mask = np.ones((3 , 3))

# wygladzanie
for i in range(height):
    for j in range(width):
        avg = 0
        n = 0
        for iOff in range(-1, 1):
            for jOff in range(-1, 1):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                avg += self.im[iSafe , jSafe] * mask[iOff + 1 , jOff +
                    1]
                n += mask[iOff + 1 , jOff + 1]
        avg = int(round(avg / n))
        resultImage[i , j] = avg

    if show:
        self.show(Image.fromarray(resultImage , "L"))
    self.save(resultImage , self.imName, "lowpassAvg")

```

Listing 10.2: Filtr dolnoprzepustowy uśredniający (obraz barwny)

```

def averageColor ( self , show = False ) :
    width = self .im .shape [1]      # szereokosc
    height = self .im .shape [0]     # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    mask = np .ones ((3 , 3))

    # wygladzanie
    for i in range (height):
        for j in range (width):
            avgr = 0
            avgg = 0
            avgb = 0
            n = 0
            for iOff in range (-1, 1):
                for jOff in range (-1, 1):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    avgr += self .im [iSafe , jSafe ] [0] * mask [iOff + 1 ,
                        jOff + 1]
                    avgg += self .im [iSafe , jSafe ] [1] * mask [iOff + 1 ,
                        jOff + 1]
                    avgb += self .im [iSafe , jSafe ] [2] * mask [iOff + 1 ,
                        jOff + 1]
                    n += mask [iOff + 1 , jOff + 1]
            avgr = int (round (avgr / n))
            avgg = int (round (avgg / n))
            avgb = int (round (avgb / n))
            resultImage [i , j] = (avgr , avgg , avgb)

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
    self .save (resultImage , self .imName , "lowpassAvg")

```

10.2 Filtr dolnoprzepustowy Gaussowski

Opis algorytmu

Filtr Gaussa jest filtrem uśredniającym. Jego maska aproksymuje 2-wymiarową krzywą Gaussa. W odrużnieniu od filtru uśredniającego efekt rozmycia przez ten filtr jest mniejszy. Maska:

$\frac{1}{47}$	1	1	1	1	1
	1	4	6	4	1
	1	1	1	1	1
	1	4	6	4	1
	1	1	1	1	1

1. Dla każdego piksla(P):
2. Dla każdej barwy:
3. Zsumuj wartości barwy piksli, otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Sumę wartości barwy podziel przez sumę wag maski.
5. Przypisz nową wartość barwy pikselowi P .



Rysunek 10.5: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.6: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.7: Obraz wejściowy, obraz po filtracji filtrem Gaussa



Rysunek 10.8: Obraz wejściowy, obraz po filtracji filtrem Gaussa

Listing 10.3: Filtr dolnoprzepustowy Gaussowski (obraz szary)

```

def gaussGray( self , show = False ):
    width = self .im .shape [1]  # szerokość
    height = self .im .shape [0]  # wysokość

    # alokacja pamięci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    mask = np .ones((5 , 5))
    mask [1 , 1] = mask [3 , 3] = mask [1 , 3] = mask [3 , 1] = 4
    mask [1 , 2] = mask [3 , 2] = 6

    # filtracja
    for i in range (height):
        for j in range (width):
            n = 0
            value = 0
            for iOff in range (-2, 3):
                for jOff in range (-2, 3):
                    iSafe = i if ((i + iOff) > (height - 2)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 2)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 2 , jOff
                        + 2]
                    n += mask [iOff + 2 , jOff + 2]
            value = int (round (value / n))
            resultImage [i , j] = value

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "lowpassGauss")

```

Listing 10.4: Filtr dolnoprzepustowy Gaussowski (obraz barwny)

```

def gaussColor(self , show = False):
    width = self.im.shape[1]      # szereokosc
    height = self.im.shape[0]     # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)

mask = np.ones((5 , 5))
mask[1 , 1] = mask[3 , 3] = mask[1 , 3] = mask[3 , 1] = 4
mask[1 , 2] = mask[3 , 2] = 6

# filtracja
for i in range(height):
    for j in range(width):
        n = 0
        r , g , b = 0 , 0 , 0
        for iOff in range(-2, 3):
            for jOff in range(-2, 3):
                iSafe = i if ((i + iOff) > (height - 2)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 2)) else (j +
                    jOff)
                r += self.im[iSafe , jSafe][0] * mask[iOff + 2 , jOff +
                    2]
                g += self.im[iSafe , jSafe][1] * mask[iOff + 2 , jOff +
                    2]
                b += self.im[iSafe , jSafe][2] * mask[iOff + 2 , jOff +
                    2]
                n += mask[iOff + 2 , jOff + 2]
        r = int(round(r / n))
        g = int(round(g / n))
        b = int(round(b / n))
        resultImage[i , j] = (r , g , b)

if show:
    self.show(Image.fromarray(resultImage , "RGB"))
    self.save(resultImage , self.imName , "lowpassGauss")

```

10.3 Operator Roberts'a

Opis algorytmu

Filtr Roberts'a jest jednym z najbardziej znanych filtrów do wykrywania krawędzi w obrazie. Wynikowa wartość składowej po zastosowaniu owego filtra może wyjść ujemna, aby temu zapobiec należy użyć wartości bezwzględnej. Filtr Roberts'a jest bardzo wrażliwy na szum i ma niski poziom reakcji na krawędź obrazu. Maska:

0	0	0
0	0	-1
0	1	0

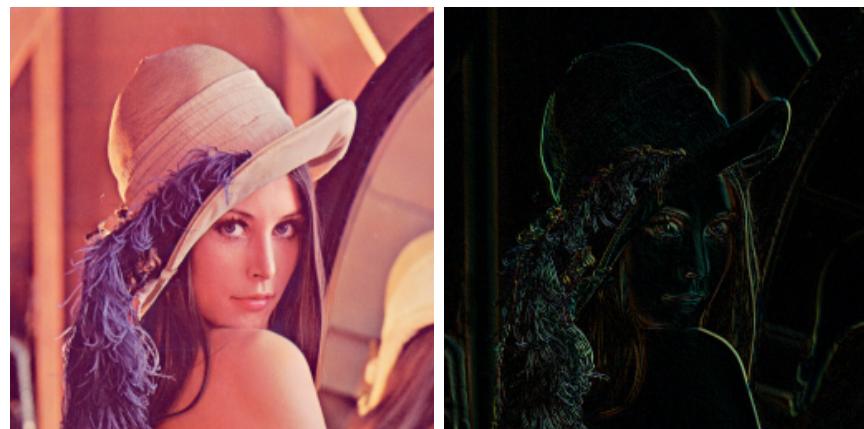
1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Zsumuj wartości pikseli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Przypisz nową wartość barwy pikselowi P .



Rysunek 10.9: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.10: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.11: Obraz wejściowy, obraz po filtracji filtrem Roberts'a



Rysunek 10.12: Obraz wejściowy, obraz po filtracji filtrem Roberts'a

Listing 10.5: Operator Roberts'a (obraz szary)

```

def robertsGray(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width) , dtype=np.uint8)

mask = np.zeros((3 , 3))
mask[2 , 1] = 1
mask[1 , 2] = -1

# filtracja
for i in range(height):
    for j in range(width):
        value = 0
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                value += self.im[ iSafe , jSafe ] * mask[iOff + 1 , jOff
                    + 1]
        resultImage[i , j] = abs(value)

if show:
    self.show(Image.fromarray(resultImage , "L"))
    self.save(resultImage , self.imName , "highpassRoberts")

```

Listing 10.6: Operator Roberts'a (obraz barwny)

```

def robertsColor (self , show = False):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty ((height , width , 3) , dtype=np .uint8)
    tmp = np .empty ((height , width , 3))
    tmp2 = np .empty ((height , width , 3))

    mask = np .zeros ((3 , 3))
    mask [2 , 1] = 1
    mask [1 , 2] = -1

    # filtracja
    for i in range (height):
        for j in range (width):
            r , g , b = 0 , 0 , 0
            for iOff in range (-1 , 2):
                for jOff in range (-1 , 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ] [0] * mask [iOff + 1 , jOff +
                        1]
                    g += self .im [iSafe , jSafe ] [1] * mask [iOff + 1 , jOff +
                        1]
                    b += self .im [iSafe , jSafe ] [2] * mask [iOff + 1 , jOff +
                        1]
            resultImage [i , j] = (abs(r) , abs(g) , abs(b))

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "highpassRoberts")

```

10.4 Operator Prewitt'a

Opis algorytmu

Filtr Prewitt'a, podobnie jak filtr Roberts'a, służy do wykrywania krawędzi i może w wyniku wygenerować wartość ujemną, aby temu zapobiec należy użyć wartości bezwzględnej. Maska Prewitt'a jest rozszerzeniem maski Roberts'a i nie jest tak wrażliwa na szum. Maska:

$$\frac{1}{2} \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

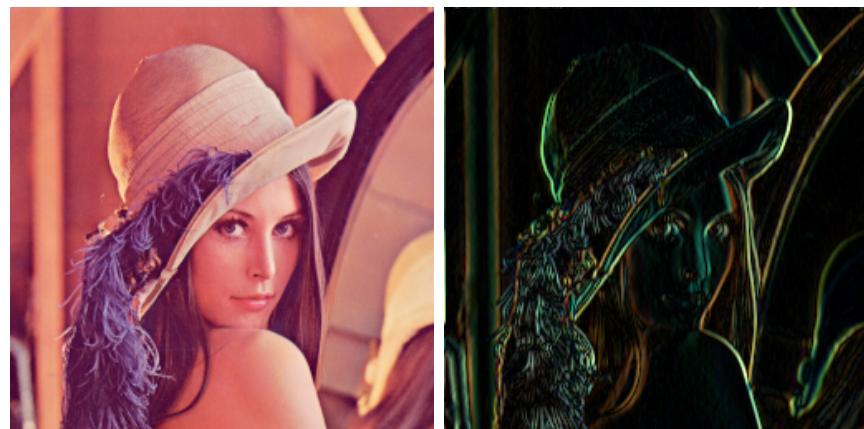
1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Zsumuj wartości pikseli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 2.
6. Jeśli otrzymany wynik jest > 255 .
7. Przypisz mu wartość 255.
8. Przypisz nową wartość barwy pikseli P .



Rysunek 10.13: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.14: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.15: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a



Rysunek 10.16: Obraz wejściowy, obraz po filtracji filtrem Prewitt'a

Listing 10.7: Operator Prewitt'a (obraz szary)

```

def prewittGray(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width) , dtype=np.uint8)

mask = np.zeros((3 , 3))
mask[0 , 0] = mask[1 , 0] = mask[2 , 0] = -1
mask[0 , 2] = mask[1 , 2] = mask[2 , 2] = 1

# filtracja
for i in range(height):
    for j in range(width):
        value = 0
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                value += self.im[iSafe , jSafe] * mask[iOff + 1 , jOff
                    + 1]
        resultImage[i , j] = abs(value / 2)

if show:
    self.show(Image.fromarray(resultImage , "L"))
    self.save(resultImage , self.imName , "highpassPrewitt")

```

Listing 10.8: Operator Prewitt'a (obraz barwny)

```

def prewittColor ( self , show = False):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty ((height , width , 3) , dtype=np .uint8)
    tmp = np .empty ((height , width , 3))
    tmp2 = np .empty ((height , width , 3))

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [1 , 0] = mask [2 , 0] = -1
    mask [0 , 2] = mask [1 , 2] = mask [2 , 2] = 1

    # filtracja
    for i in range (height):
        for j in range (width):
            r , g , b = 0 , 0 , 0
            for iOff in range (-1 , 2):
                for jOff in range (-1 , 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self .im [iSafe , jSafe ] [0] * mask [iOff + 1 , jOff +
                        1]
                    g += self .im [iSafe , jSafe ] [1] * mask [iOff + 1 , jOff +
                        1]
                    b += self .im [iSafe , jSafe ] [2] * mask [iOff + 1 , jOff +
                        1]
            r , g , b = (abs (r )/2 , abs (g )/2 , abs (b )/2)
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage [i , j] = (r , g , b)

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "highpassPrewitt")

```

10.5 Operator Sobel'a

Opis algorytmu

Filtr Prewitt'a, podobnie jak filtr Roberts'a, służy do wykrywania krawędzi i może w wyniku wygenerować wartość ujemną, aby temu zapobiec należy użyć wartości bezwzględnej. Maska Prewitt'a jest rozszerzeniem maski Roberts'a i nie jest tak wrażliwa na szum. Maska:

	1	2	1
$\frac{1}{4}$	0	0	0
	-1	-2	-1

1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Zsumuj wartości piksli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 4.
6. Jeśli otrzymana wartość jest > 255 .
7. Przypisz jej wartość 255.
8. Przypisz nową wartość barwy pikslowi P .



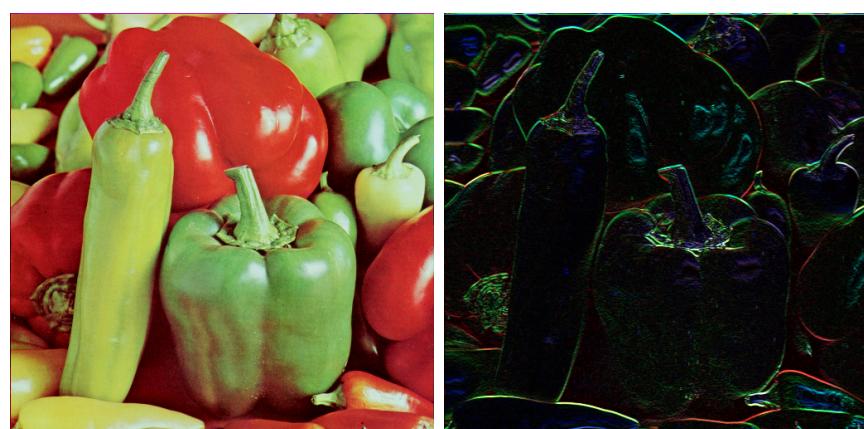
Rysunek 10.17: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.18: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.19: Obraz wejściowy, obraz po filtracji filtrem Sobel'a



Rysunek 10.20: Obraz wejściowy, obraz po filtracji filtrem Sobel'a

Listing 10.9: Operator Sobel'a (obraz szary)

```

def sobolGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [0 , 2] = 1
    mask [2 , 0] = mask [2 , 2] = -1
    mask [0 , 1] = 2
    mask [2 , 1] = -2

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff
                        + 1]
            resultImage [i , j] = abs (value / 4)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "highpassSobel")

```

Listing 10.10: Operator Sobel'a (obraz barwny)

```

def sobolColor(self , show = False):
    width = self.im.shape[1]  # szereoksc
    height = self.im.shape[0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width , 3) , dtype=np.uint8)
    tmp = np.empty((height , width , 3))
    tmp2 = np.empty((height , width , 3))

    mask = np.zeros((3 , 3))
    mask[0 , 0] = mask[0 , 2] = 1
    mask[2 , 0] = mask[2 , 2] = -1
    mask[0 , 1] = 2
    mask[2 , 1] = -2

    # filtracja
    for i in range(height):
        for j in range(width):
            r , g , b = 0 , 0 , 0
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r += self.im[iSafe , jSafe][0] * mask[iOff + 1, jOff +
                        1]
                    g += self.im[iSafe , jSafe][1] * mask[iOff + 1, jOff +
                        1]
                    b += self.im[iSafe , jSafe][2] * mask[iOff + 1, jOff +
                        1]
            r = abs(r) / 2
            g = abs(g) / 2
            b = abs(b) / 2
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage[i , j] = (r , g , b)

```

```
if show:  
    self.show( Image.fromarray(resultImage, "RGB") )  
    self.save(resultImage, self.imName, "highpassSobol")
```

10.6 Filtr kompasowy

Opis algorytmu

Filtr kompasowy polega na splecieniu zbioru 8 masek wzornikowych, gdzie każda z nich jest czuła w innym kierunku. Dla każdego piksla wybierana jest maska o maksymalnej reakcji. Maski Sobel'a:

$\frac{1}{4}$	-1	0	1
	-2	0	2
	-1	0	1

$\frac{1}{4}$	0	1	2
	-1	0	1
	-2	-1	0

$\frac{1}{4}$	1	2	1
	0	0	0
	-1	-2	-1

$\frac{1}{4}$	2	1	0
	1	0	-1
	0	-1	-2

$\frac{1}{4}$	1	0	-1
	2	0	-2
	1	0	-1

$\frac{1}{4}$	0	-1	-2
	1	0	-1
	2	1	0

$\frac{1}{4}$	-1	-2	-1
	0	0	0
	1	2	1

$\frac{1}{4}$	-2	-1	0
	-1	0	1
	0	1	2

1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Dla każdej z masek(M):
4. Zsumuj wartości piksli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski M .
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Następnie podziel ją przez 4.

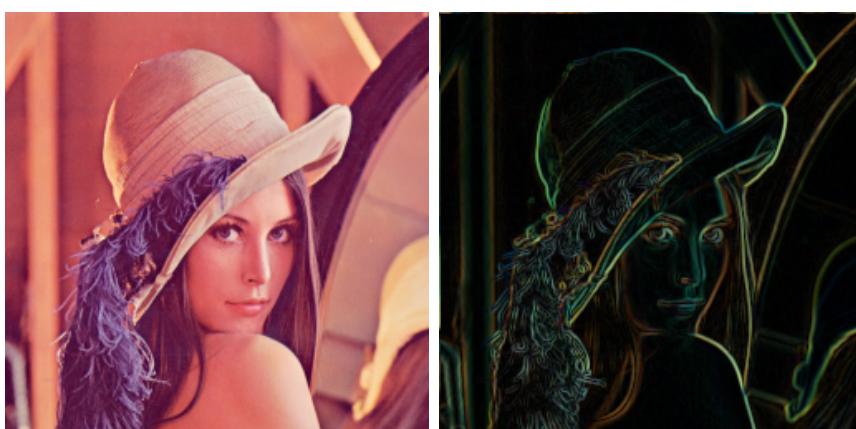
7. Wybierz największą z wartości, która jest maksymalną reakcją gradientu.
8. Przypisz nową wartość barwy pikselowi P .



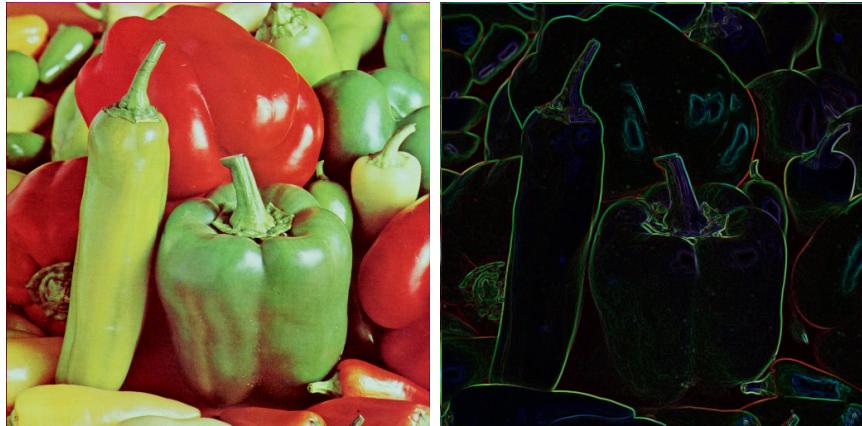
Rysunek 10.21: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.22: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.23: Obraz wejściowy, obraz po filtracji filtrem kompasowym



Rysunek 10.24: Obraz wejściowy, obraz po filtracji filtrem kompasowym

Listing 10.11: Filtr kompasowy (obraz szary)

```

def compassGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    #maska
    mask = [0] * 8

    mask [0] = np .zeros ((3 , 3))
    mask [0][0 , 2] = mask [0][2 , 2] = 1
    mask [0][0 , 0] = mask [0][2 , 0] = -1
    mask [0][1 , 2] = 2
    mask [0][1 , 0] = -2

    mask [1] = np .zeros ((3 , 3))
    mask [1][0 , 1] = mask [1][1 , 2] = 1
    mask [1][1 , 0] = mask [1][2 , 1] = -1
    mask [1][0 , 2] = 2
    mask [1][2 , 0] = -2

    mask [2] = np .zeros ((3 , 3))
    mask [2][0 , 0] = mask [2][0 , 2] = 1
    mask [2][2 , 0] = mask [2][2 , 2] = -1
    mask [2][0 , 1] = 2
    mask [2][2 , 1] = -2

```

```

mask[3] = np.zeros((3, 3))
mask[3][0, 1] = mask[3][1, 0] = 1
mask[3][1, 2] = mask[3][2, 1] = -1
mask[3][0, 0] = 2
mask[3][2, 2] = -2

mask[4] = np.zeros((3, 3))
mask[4][0, 0] = mask[4][2, 0] = 1
mask[4][0, 2] = mask[4][2, 2] = -1
mask[4][1, 0] = 2
mask[4][1, 2] = -2

mask[5] = np.zeros((3, 3))
mask[5][1, 0] = mask[5][2, 1] = 1
mask[5][0, 1] = mask[5][1, 2] = -1
mask[5][2, 0] = 2
mask[5][0, 2] = -2

mask[6] = np.zeros((3, 3))
mask[6][2, 0] = mask[6][2, 2] = 1
mask[6][0, 0] = mask[6][0, 2] = -1
mask[6][2, 1] = 2
mask[6][0, 1] = -2

mask[7] = np.zeros((3, 3))
mask[7][1, 2] = mask[7][2, 1] = 1
mask[7][0, 1] = mask[7][1, 0] = -1
mask[7][2, 2] = 2
mask[7][0, 0] = -2

# filtracja
for i in range(height):
    for j in range(width):
        value = [0] * 8
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                for k in range(8):

```

```
    value[k] += self.im[iSafe, jSafe] * mask[k][iOff +
        1, jOff + 1]

    resultImage[i, j] = max(map(abs, value)) / 4

if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "compassSobol")
```

Listing 10.12: Filtr kompasowy (obraz barwny)

```

def compassColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    #maska
    mask = [0] * 8

    mask [0] = np .zeros ((3 , 3))
    mask [0][0 , 2] = mask [0][2 , 2] = 1
    mask [0][0 , 0] = mask [0][2 , 0] = -1
    mask [0][1 , 2] = 2
    mask [0][1 , 0] = -2

    mask [1] = np .zeros ((3 , 3))
    mask [1][0 , 1] = mask [1][1 , 2] = 1
    mask [1][1 , 0] = mask [1][2 , 1] = -1
    mask [1][0 , 2] = 2
    mask [1][2 , 0] = -2

    mask [2] = np .zeros ((3 , 3))
    mask [2][0 , 0] = mask [2][0 , 2] = 1
    mask [2][2 , 0] = mask [2][2 , 2] = -1
    mask [2][0 , 1] = 2
    mask [2][2 , 1] = -2

    mask [3] = np .zeros ((3 , 3))
    mask [3][0 , 1] = mask [3][1 , 0] = 1
    mask [3][1 , 2] = mask [3][2 , 1] = -1
    mask [3][0 , 0] = 2
    mask [3][2 , 2] = -2

    mask [4] = np .zeros ((3 , 3))
    mask [4][0 , 0] = mask [4][2 , 0] = 1
    mask [4][0 , 2] = mask [4][2 , 2] = -1
    mask [4][1 , 0] = 2
    mask [4][1 , 2] = -2

```

```

mask[5] = np.zeros((3, 3))
mask[5][1, 0] = mask[5][2, 1] = 1
mask[5][0, 1] = mask[5][1, 2] = -1
mask[5][2, 0] = 2
mask[5][0, 2] = -2

mask[6] = np.zeros((3, 3))
mask[6][2, 0] = mask[6][2, 2] = 1
mask[6][0, 0] = mask[6][0, 2] = -1
mask[6][2, 1] = 2
mask[6][0, 1] = -2

mask[7] = np.zeros((3, 3))
mask[7][1, 2] = mask[7][2, 1] = 1
mask[7][0, 1] = mask[7][1, 0] = -1
mask[7][2, 2] = 2
mask[7][0, 0] = -2

# filtracja
for i in range(height):
    for j in range(width):
        r = [0] * 8
        g = [0] * 8
        b = [0] * 8
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                for k in range(8):
                    r[k] += self.im[iSafe, jSafe][0] * mask[k][iOff +
                        1, jOff + 1]
                    g[k] += self.im[iSafe, jSafe][1] * mask[k][iOff +
                        1, jOff + 1]
                    b[k] += self.im[iSafe, jSafe][2] * mask[k][iOff +
                        1, jOff + 1]

            resultImage[i, j] = (max(map(abs, r)) / 4, max(map(abs, g
            )) / 4, max(map(abs, b)) / 4)

if show:

```

```
self.show(Image.fromarray(resultImage, "RGB"))
self.save(resultImage, self.imName, "compassSobel")
```

10.7 Gradient wektora kierunkowego

Opis algorytmu

Filtr ten służy do wykrywania krawędzi w obrazie. Podobnie jak dla filtru kompasowego, filtr wektora kierunkowego polega na splecieniu 4 masek wzornikowych, gdzie każda z nich jest czuła w innym kierunku, a dla każdego piksla wybierana jest maska o maksymalnej reakcji. Maski Prewitt'a:

$\frac{1}{2}$	-1	0	1
	-1	0	1
	-1	0	1

$\frac{1}{2}$	1	1	1
	0	0	0
	-1	-1	-1

$\frac{1}{2}$	1	0	-1
	1	0	-1
	1	0	-1

$\frac{1}{2}$	-1	-1	-1
	0	0	0
	1	1	1

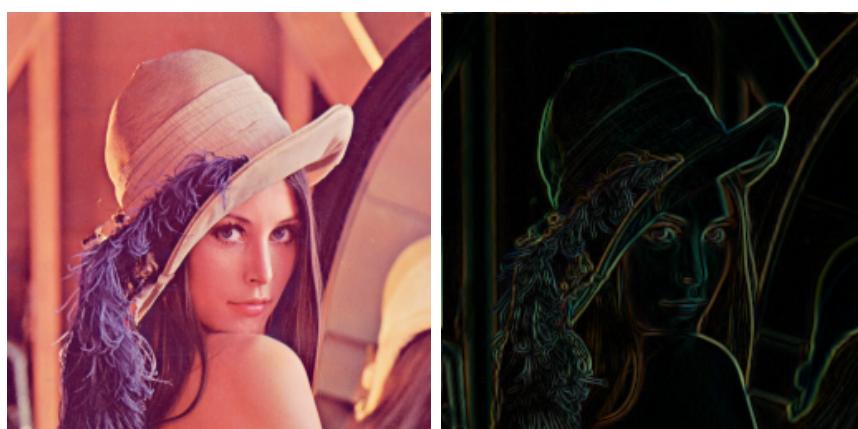
1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Dla każdej z masek(M):
4. Zsumuj wartości pikseli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski M .
5. Zastosuj wartość bezwzględną na otrzymanej sumie.
6. Następnie podziel ją przez 4.
7. Wybierz największą z wartości, która jest maksymalną reakcją gradientu.
8. Przypisz nową wartość barwy pikslowi P .



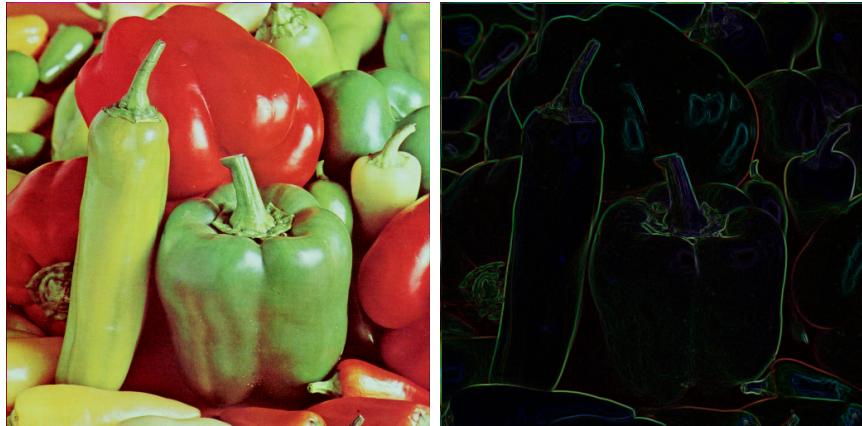
Rysunek 10.25: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.26: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.27: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego



Rysunek 10.28: Obraz wejściowy, obraz po filtracji filtrem wekora kierunkowego

Listing 10.13: Gradient wektora kierunkowego (obraz szary)

```
def VDGGray(self , show = False) :
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height , width) , dtype=np.uint8)

    #maska
    mask = [0] * 4

    mask[0] = np.zeros((3 , 3))
    mask[0][0 , 2] = mask[0][2 , 2] = 1
    mask[0][0 , 0] = mask[0][2 , 0] = -1
    mask[0][1 , 2] = 1
    mask[0][1 , 0] = -1

    mask[1] = np.zeros((3 , 3))
    mask[1][0 , 0] = mask[1][0 , 2] = 1
    mask[1][2 , 0] = mask[1][2 , 2] = -1
    mask[1][0 , 1] = 1
    mask[1][2 , 1] = -1

    mask[2] = np.zeros((3 , 3))
    mask[2][0 , 0] = mask[2][2 , 0] = 1
    mask[2][0 , 2] = mask[2][2 , 2] = -1
    mask[2][1 , 0] = 1
    mask[2][1 , 2] = -1
```

```
mask[3] = np.zeros((3, 3))
mask[3][2, 0] = mask[3][2, 2] = 1
mask[3][0, 0] = mask[3][0, 2] = -1
mask[3][2, 1] = 1
mask[3][0, 1] = -1

# filtracja
for i in range(height):
    for j in range(width):
        value = [0] * 4
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                for k in range(4):
                    value[k] += self.im[iSafe, jSafe] * mask[k][iOff + 1,
                        jOff + 1]

        resultImage[i, j] = max(map(abs, value)) / 2

if show:
    self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "vdgSobol")
```

Listing 10.14: Gradient wektora kierunkowego (obraz barwny)

```

def VDGColor( self , show = False ) :
    width = self .im .shape [1]   # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)
    tmp = np .empty(( height , width , 3))
    tmp2 = np .empty(( height , width , 3))

    #maska
    mask = [0] * 4

    mask [0] = np .zeros ((3 , 3))
    mask [0][0 , 2] = mask [0][2 , 2] = 1
    mask [0][0 , 0] = mask [0][2 , 0] = -1
    mask [0][1 , 2] = 1
    mask [0][1 , 0] = -1

    mask [1] = np .zeros ((3 , 3))
    mask [1][0 , 0] = mask [1][0 , 2] = 1
    mask [1][2 , 0] = mask [1][2 , 2] = -1
    mask [1][0 , 1] = 1
    mask [1][2 , 1] = -1

    mask [2] = np .zeros ((3 , 3))
    mask [2][0 , 0] = mask [2][2 , 0] = 1
    mask [2][0 , 2] = mask [2][2 , 2] = -1
    mask [2][1 , 0] = 1
    mask [2][1 , 2] = -1

    mask [3] = np .zeros ((3 , 3))
    mask [3][2 , 0] = mask [3][2 , 2] = 1
    mask [3][0 , 0] = mask [3][0 , 2] = -1
    mask [3][2 , 1] = 1
    mask [3][0 , 1] = -1

    # filtracja
    for i in range (height):
        for j in range (width):
            r = [0] * 4
            g = [0] * 4

```

```

b = [0] * 4
for iOff in range(-1, 2):
    for jOff in range(-1, 2):
        iSafe = i if ((i + iOff) > (height - 1)) else (i +
            iOff)
        jSafe = j if ((j + jOff) > (width - 1)) else (j +
            jOff)
    for k in range(4):
        r[k] += self.im[iSafe, jSafe][0] * mask[k][iOff +
            1, jOff + 1]
        g[k] += self.im[iSafe, jSafe][1] * mask[k][iOff +
            1, jOff + 1]
        b[k] += self.im[iSafe, jSafe][2] * mask[k][iOff +
            1, jOff + 1]

    resultImage[i, j] = (max(map(abs, r)) / 4, max(map(abs, g
        )) / 4, max(map(abs, b)) / 4)

if show:
    self.show(Image.fromarray(resultImage, "RGB"))
    self.save(resultImage, self.imName, "vdgSobol")

```

10.8 Filtr medianowy

Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Filtr medianowy (środkowy) opiera się na medianie, czyli wartości środkowej spośród uporządkowanych wartości piksli z otoczenia badanego piksla. Filtr ten stosuje się do redukcji szumu w obrazie.

1. Dla każdego piksla (P):
2. Dla każdej z barw (C):
3. Umieść wartości barwy C piksli z otoczenia piksla P w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy C , a następnie wybierz medianę.
5. Przypisz znalezioną medianę jako nową wartość barwy piksla P .



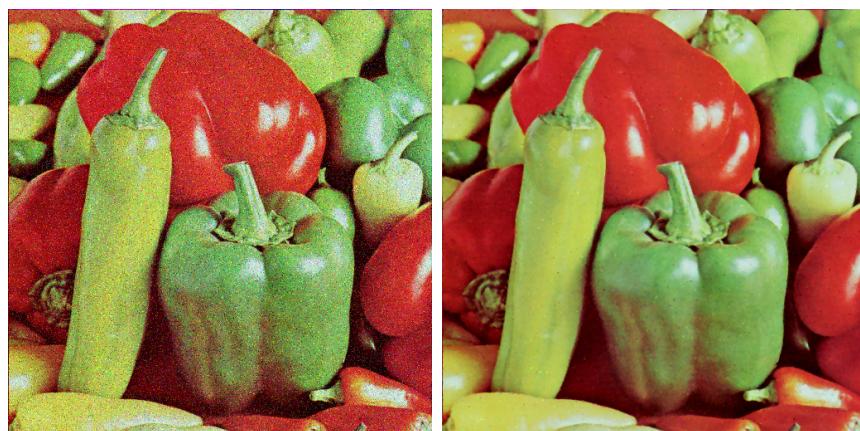
Rysunek 10.29: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.30: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.31: Obraz wejściowy, obraz po filtracji filtrem medianowym



Rysunek 10.32: Obraz wejściowy, obraz po filtracji filtrem medianowym

Listing 10.15: Filtr medianowy (obraz szary)

```

def medianGray( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width ) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            median = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    median[3*(1 + iOff) + jOff + 1] = self .im [iSafe ,
                        jSafe]
            median .sort ()
            u = int(round(len(median)/2))
            resultImage [i , j] = median[u] if ((u*2) % 2 == 0) else ((
                median[u - 1] + median[u])/2)

    if show:
        self .show (Image .fromarray (resultImage , "L"))
    self .save (resultImage , self .imName , "median")

```

Listing 10.16: Filtr medianowy (obraz brawny)

```

def medianColor(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)

for i in range(height):
    for j in range(width):
        r = [0] * 9
        g = [0] * 9
        b = [0] * 9
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                r[3 * (1 + iOff) + jOff + 1] = self.im[iSafe , jSafe
                    ][0]
                g[3 * (1 + iOff) + jOff + 1] = self.im[iSafe , jSafe
                    ][1]
                b[3 * (1 + iOff) + jOff + 1] = self.im[iSafe , jSafe
                    ][2]
        r.sort()
        g.sort()
        b.sort()
        ur = int(round(len(r) / 2))
        ug = int(round(len(g) / 2))
        ub = int(round(len(b) / 2))
        resultImage[i , j] = (r[ur] if ((ur*2) % 2 == 0) else ((r[
            ur - 1] + r[ur])/2) , g[ug] if ((ug*2) % 2 == 0) else
            ((g[ug - 1] + g[ug])/2) , b[ub] if ((ub*2) % 2 == 0)
            else ((b[ub - 1] + b[ub])/2))

if show:
    self.show(Image.fromarray(resultImage , "RGB"))
    self.save(resultImage , self.imName, "median")

```

10.9 Filtr maksymalny

Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Zwany jest także filtrem dekompresującym albo ekspansywnym. Jego działanie polega na wybraniu z pod maski punktu o wartości największej. Jego działanie powoduje zwiększenie jasności obrazu, daje to efekt powiększania się obiektów.

1. Dla każdego piksla (P):
2. Dla każdej z barw (C):
3. Umieść wartości barwy C piksli z otoczenia piksla P w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy C , a następnie wybierz ostatni(największy) element.
5. Przypisz znalezioną wartość jako nową wartość barwy piksla P .



Rysunek 10.33: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.34: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.35: Obraz wejściowy, obraz po filtracji filtrem maksymalnym



Rysunek 10.36: Obraz wejściowy, obraz po filtracji filtrem maksymalnym

Listing 10.17: Filtr maksymalny (obraz szary)

```
def maxGray(self, show=False):
    width = self.im.shape[1]  # szerokosc
    height = self.im.shape[0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width), dtype=np.uint8)

    for i in range(height):
        for j in range(width):
            median = [0] * 9
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                    median[3 * (1 + iOff) + jOff + 1] = self.im[iSafe, jSafe]
            median.sort()
            resultImage[i, j] = median[len(median) - 1]

    if show:
        self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "max")
```

Listing 10.18: Filtr maksymalny (obraz barwny)

```

def maxColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            r = [0] * 9
            g = [0] * 9
            b = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][0]
                    g[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][1]
                    b[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][2]
            r .sort ()
            g .sort ()
            b .sort ()
            resultImage [i , j] = (r [len (r) - 1] , g [len (g) - 1] , b [len (
                b) - 1])

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "max")

```

10.10 Filtr minimalny

Opis algorytmu

Jeden z filtrów statystycznych, którego efekt opiera się na wyborze odpowiedniego piksla pod maską. Zwany jest także filtrem kompresującym albo erozyjnym. Jego działanie polega na wybraniu z pod maski punktu o wartości najmniejszej. Jego działanie powoduje zmniejszenie jasności obrazu, daje to efekt erozji obiektów.

1. Dla każdego piksla (P):
2. Dla każdej z barw (C):
3. Umieść wartości barwy C piksli z otoczenia piksla P w tablicy jednowymiarowej.
4. Posortuj rosnąco wartości barwy C , a następnie wybierz pierwszy(najmniejszy) element.
5. Przypisz znalezioną wartość jako nową wartość barwy piksla P .



Rysunek 10.37: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.38: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.39: Obraz wejściowy, obraz po filtracji filtrem minimalnym



Rysunek 10.40: Obraz wejściowy, obraz po filtracji filtrem minimalnym

Listing 10.19: Filtr minimalny (obraz szary)

```
def minGray(self, show=False):
    width = self.im.shape[1]  # szerokosc
    height = self.im.shape[0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np.empty((height, width), dtype=np.uint8)

    for i in range(height):
        for j in range(width):
            median = [0] * 9
            for iOff in range(-1, 2):
                for jOff in range(-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                    median[3 * (1 + iOff) + jOff + 1] = self.im[iSafe, jSafe]
            median.sort()
            resultImage[i, j] = median[0]

    if show:
        self.show(Image.fromarray(resultImage, "L"))
    self.save(resultImage, self.imName, "min")
```

Listing 10.20: Filtr minimalny (obraz barwny)

```

def minColor( self , show = False ):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty(( height , width , 3) , dtype=np .uint8)

    for i in range (height):
        for j in range (width):
            r = [0] * 9
            g = [0] * 9
            b = [0] * 9
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i +
                        iOff)
                    jSafe = j if ((j + jOff) > (width - 1)) else (j +
                        jOff)
                    r[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][0]
                    g[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][1]
                    b[3 * (1 + iOff) + jOff + 1] = self .im [ iSafe , jSafe
                        ][2]
            r .sort ()
            g .sort ()
            b .sort ()
            resultImage [i , j] = (r [0] , g [0] , b [0])

    if show:
        self .show (Image .fromarray (resultImage , "RGB"))
        self .save (resultImage , self .imName , "min")

```

10.11 Filtr płaskorzeźbowy

Opis algorytmu

Filtr płaskorzeźbowy swoją nazwę zdobył od efektu jaki generuje jego maska, obrazy przepuszczane przez ten filtr, w efekcie przypominają płaskorzeźbę. Sposób definiowania kierunków jest podobny jak w przypadku kierunkowych filtrów gradientowych. Różnicą jest jedynie wartość środkowa mająca wartość 1. W tym programie została użyta maska Prewitt'a. Maska:

	-1	0	1
$\frac{1}{2}$	-1	1	1
	-1	0	1

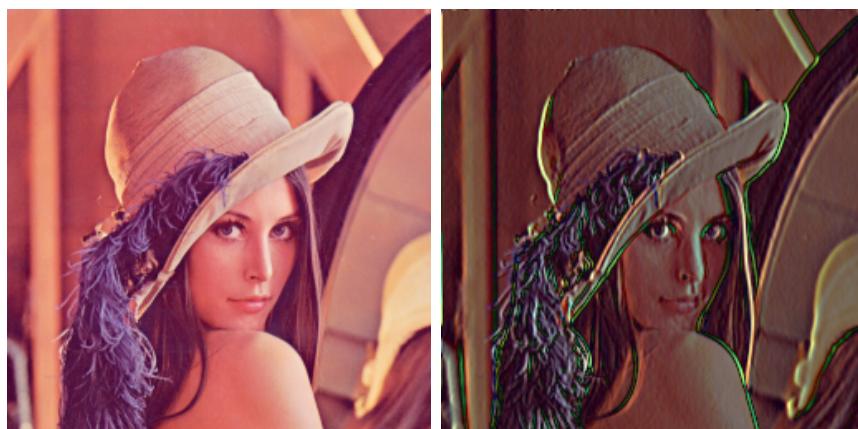
1. Dla każdego piksla(P):
2. Dla każdego kanału(C):
3. Zsumuj wartości pikseli P_C , otaczających piksel P pomnożonych przez odpowiednią wagę maski.
4. Zastosuj wartość bezwzględną na otrzymanej sumie.
5. Następnie podziel ją przez 2.
6. Jeśli otrzymana wartość jest > 255 .
7. Przypisz jej wartość 255.
8. Przypisz nową wartość barwy pikselowi P .



Rysunek 10.41: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.42: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.43: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym



Rysunek 10.44: Obraz wejściowy, obraz po filtracji filtrem płaskorzeźbowym

Listing 10.21: Filtr płaskorzeźbowy (obraz szary)

```

def reliefGray (self , show = False):
    width = self .im .shape [1]  # szereoksc
    height = self .im .shape [0]  # wysokosc

    # alokacja pamieci na obraz wynikowy
    resultImage = np .empty ((height , width) , dtype=np .uint8)

    mask = np .zeros ((3 , 3))
    mask [0 , 0] = mask [1 , 0] = mask [2 , 0] = -1
    mask [0 , 2] = mask [1 , 2] = mask [2 , 2] = 1
    mask [1 , 1] = 1

    # filtracja
    for i in range (height):
        for j in range (width):
            value = 0
            for iOff in range (-1, 2):
                for jOff in range (-1, 2):
                    iSafe = i if ((i + iOff) > (height - 1)) else (i + iOff
                        )
                    jSafe = j if ((j + jOff) > (width - 1)) else (j + jOff)
                    value += self .im [iSafe , jSafe] * mask [iOff + 1 , jOff +
                        1]
            value = abs (value) / 2
            if value > 255:
                value = 255
            resultImage [i , j] = value

    if show:
        self .show (Image .fromarray (resultImage , "L"))
        self .save (resultImage , self .imName , "reliefPrewitt")

```

Listing 10.22: Filtr płaskorzeźbowy (obraz barwny)

```

def reliefColor(self , show = False):
    width = self.im.shape[1] # szereoksc
    height = self.im.shape[0] # wysokosc

# alokacja pamieci na obraz wynikowy
resultImage = np.empty((height , width , 3) , dtype=np.uint8)
tmp = np.empty((height , width , 3))
tmp2 = np.empty((height , width , 3))

mask = np.zeros((3 , 3))
mask[0 , 0] = mask[1 , 0] = mask[2 , 0] = -1
mask[0 , 2] = mask[1 , 2] = mask[2 , 2] = 1
mask[1 , 1] = 1

# filtracja
for i in range(height):
    for j in range(width):
        r , g , b = 0 , 0 , 0
        for iOff in range(-1, 2):
            for jOff in range(-1, 2):
                iSafe = i if ((i + iOff) > (height - 1)) else (i +
                    iOff)
                jSafe = j if ((j + jOff) > (width - 1)) else (j +
                    jOff)
                r += self.im[iSafe , jSafe][0] * mask[iOff + 1 , jOff +
                    1]
                g += self.im[iSafe , jSafe][1] * mask[iOff + 1 , jOff +
                    1]
                b += self.im[iSafe , jSafe][2] * mask[iOff + 1 , jOff +
                    1]
            r = abs(r) / 2
            g = abs(g) / 2
            b = abs(b) / 2
            if r > 255:
                r = 255
            if g > 255:
                g = 255
            if b > 255:
                b = 255
            resultImage[i , j] = (r , g , b)

```

```
if show:  
    self.show(Image.fromarray(resultImage, "RGB"))  
    self.save(resultImage, self.imName, "reliefPrewitt")  
L [1]
```

Bibliografia

- [1] Wojciech S. Mokrzycki. *Wprowadzenie do przetwarzania informacji wizualnej Tom II*. Akademicka Oficyna Wydawnicza EXIT, 2012.
- [2] J. Ratajczaki. *Cyfrowe przetwarzanie obrazów i sygnałów - wykład 3*. Politechnika Wrocławskiego, 2015.