

Demozaikowanie

ArtsyKimiko

23 stycznia 2024

1 Demozaikowanie

Demozaikowanie to proces przekształcania danych obrazu pozyskanych przez aparaty fotograficzne w pełny, kolorowy obraz. W fotografii cyfrowej, każdy piksel na obrazie pierwotnie reprezentuje tylko jeden kolor, a różne kolory są zapisane w mozaikowej strukturze. Demozaikowanie ma na celu odtworzenie pełnej gamy kolorów na podstawie tych pierwotnych danych mozaikowych.

2 Filtry kolorów

Opisane poniżej filtry kolorów (Bayera i Fuji X-Trans) są używane w matrycach CFA (Color Filter Array), które decydują, jakie informacje kolorystyczne są rejestrowane przez poszczególne piksele na sensorze aparatu

2.1 Maska Bayera

Maska Bayera jest jednym z najbardziej powszechnych układów CFA w cyfrowych sensorach obrazu. Oznacza, że co cztery piksele, dwa to zielone, jeden czerwony, a jeden niebieski. Jest stosunkowo prosty, ale efektywny w praktyce. Wygląd układu pikseli prezentuje się następująco:

$$\begin{bmatrix} G & R \\ B & G \end{bmatrix}$$

2.2 Maska Fuji X-Trans

Maska Fuji X-Trans to zaawansowany układ używany w aparatach firmy Fujifilm. Jest on inny niż tradycyjna maska Bayera i został zaprojektowany w taki sposób, aby zmniejszyć efekt moiré i poprawić ogólną jakość obrazu. Piksele czerwone, zielone i niebieskie są rozmiieszczone w bardziej nieregularny sposób niż w przypadku maski Bayera. Schemat jej pikseli prezentuje się następująco:

$$\begin{bmatrix} G & B & R & G & B & R \\ R & G & G & B & G & G \\ B & G & G & R & G & G \\ G & B & R & G & B & R \\ R & G & G & B & G & G \\ B & G & G & R & G & G \end{bmatrix}$$

3 Metody Demozaikowania

3.1 Interpolacja

W celu zrealizowania demozaikowania za pomocą metody interpolacji przy użyciu maski Bayera, została utworzona funkcja `demosaic_Bayer_interpolation`. Funkcja ta przyjmuje dwuwymiarową macierz `image`, reprezentującą obraz, oraz opcjonalny parametr `type` (domyślnie ustawiony na `average`), który decyduje o rodzaju interpolacji.

W kolejnym kroku korzystamy z funkcji `Mosaicing_Bayer`, która przeprowadza proces mozaikowania na podanym obrazie. Wynik tego procesu, zapisany w zmiennej `mosaiking`, jest trójwymiarową macierzą, gdzie każdy kanał reprezentowany jest zgodnie z układem mozaiki Bayera. W związku z tym, możemy przypisać zmiennym `red_channel`, `green_channel` i `blue_channel`, reprezentującym odpowiednio czerwony, zielony i niebieski kanał, wartości pochodzące z odpowiednich kanałów macierzy `mosaiking`, tj. `mosaiking[:, :, 0]`, `mosaiking[:, :, 1]` oraz `mosaiking[:, :, 2]`.

Następnie funkcja przechodzi do interpolacji kolorów czerwonego, zielonego i niebieskiego. Proces interpolacji jest realizowany osobno dla różnych przypadków, biorąc pod uwagę parzystość/nieparzystość wierszy i kolumn. Poniżej przedstawiono jedynie fragmenty kodu odpowiedzialne za właściwy wybór wierszy i kolumn, niezbędny do poprawnej realizacji interpolacji.

```
# Interpolacja koloru czerwonego
for i in range(1,height,2):
    for j in range(1,width,2):
        ...
for i in range(height):
    for j in range(0,width,2):
        ...

# Interpolacja koloru zielonego
for i in range(0, height, 2):
    for j in range(1, width, 2):
        ...
for i in range(1, height, 2):
    for j in range(0, width, 2):
        ...

# Interpolacja koloru niebieskiego
for i in range(0,height,2):
    for j in range(0,width,2):
        ...
for i in range(height):
    for j in range(1,width,2):
        ...
```

Przy tej interpolacji można wybierać między dwoma rodzajami: `average`, gdzie używana jest średnia arytmetyczna, oraz `max`, gdzie wybierana jest maksymalna wartość spośród sąsiadów. Poniżej przedstawiono odpowiednie fragmenty kodu (w tym wypadku dla czerwonego koloru):

```
if type == "average":
    red_channel[i, j] = np.mean([red_channel[x, y] for x, y in valid_neighbors])
elif type == "max":
    red_channel[i, j] = np.max([red_channel[x, y] for x, y in valid_neighbors])
```

Ostatecznie, na podstawie uzyskanych kanałów czerwonego, zielonego i niebieskiego, tworzona jest macierz wynikowa za pomocą funkcji `np.vstack`. Dzięki tej operacji funkcja jest w stanie zwrócić zdemozaikowany obraz jako rezultat.

3.2 Konwolucja

Demozaikowanie przy użyciu metody konwolucji opiera się na wykorzystaniu przesunięć jądra konwolucji, gdzie maska jest dostosowana do przekazywania lub uśredniania kolorów z otoczenia piksela. Proces ten obejmuje analizę całego obrazu, z krokiem ustawnionym na jeden piksel, a wartość wzmacnienia jest uzależniona od mozaiki i proporcjonalna do liczby pikseli w danym kolorze.

Aby zobrazować działanie konwolucji w kontekście demozaikowania, zaimplementowano funkcje wykorzystujące maski Bayera i Fuji. Poniżej znajduje się fragment kodu odpowiedzialny za te operacje.

```
def demosaic_Bayer_convolution(image):
    bayer_mask = np.array([[1, 1], [1, 1]], [[1/2, 1/2], [1/2, 1/2]], [[1, 1], [1, 1]])
    result = np.dstack([cv2.filter2D(image[:, :, i], -1, bayer_mask[i]) for i in range(3)])
    return result

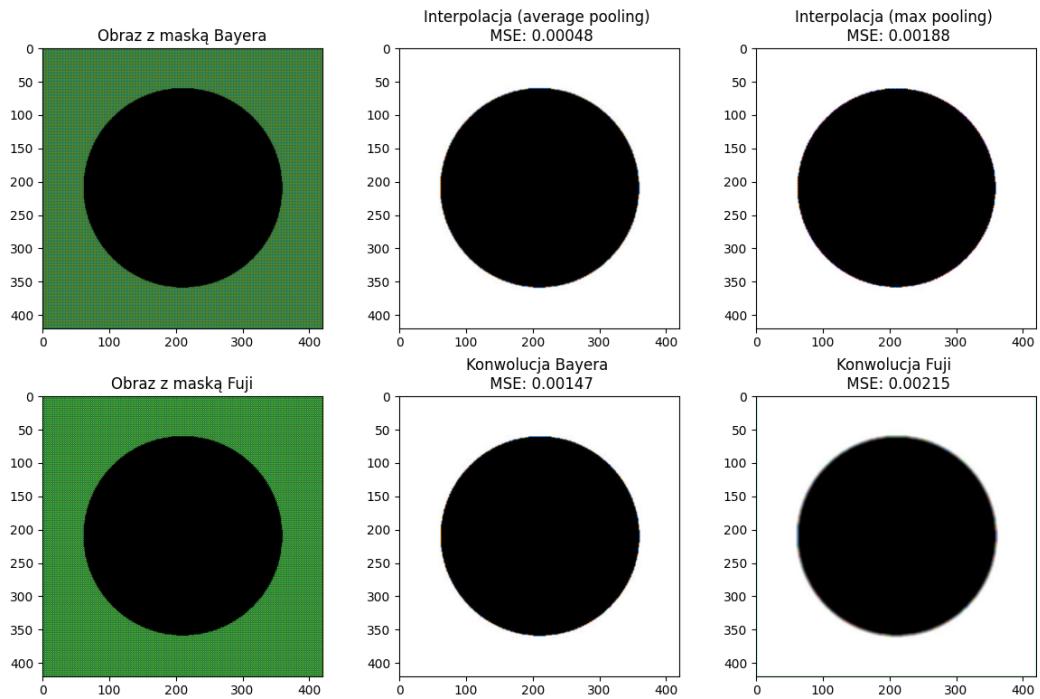
def demosaic_Fuji_convolution(image):
    fuji_mask = np.array([[1/8]*6]*6, [[1/20]*6]*6, [[1/8]*6]*6)
    result = np.dstack([cv2.filter2D(image[:, :, i], -1, fuji_mask[i]) for i in range(3)])
    return result
```

Funkcja `demosaic_Bayer_convolution` wykorzystuje maskę Bayera, gdzie trzy warstwy są zdefiniowane jako macierze 2x2 o różnych wagach dla poszczególnych kanałów koloru (czerwonego, zielonego, niebieskiego). Każda warstwa tej maski przypisuje odpowiednie wagę pikselom otoczenia, co wpływa na proces konwolucji dla każdego kanału.

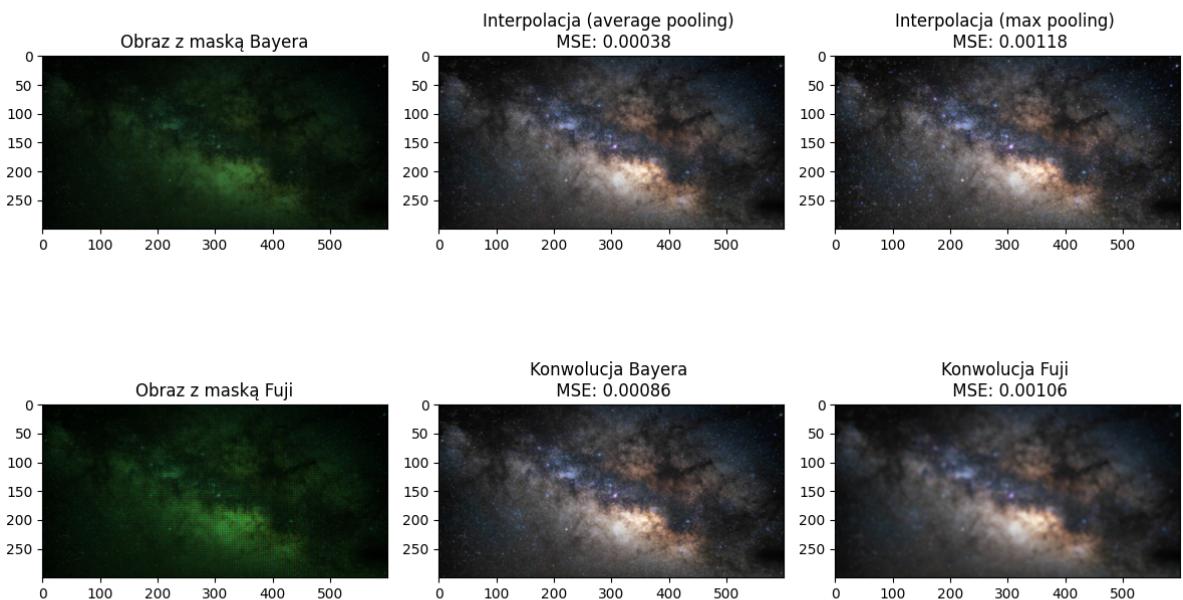
Natomiast druga funkcja, `demosaic_Fuji_convolution`, używa maski Fuji, również zdefiniowanej jako macierze z różnymi wagami dla poszczególnych kanałów koloru, jednak o wymiarach 6x6. Waga każdej warstwy w masce Fuji różni się od wag w masce Bayera, wprowadzając różnice w procesie przekształcania pikseli.

Obie funkcje wspólnie korzystają z konwolucji 2D, stosując ją dla każdego kanału obrazu wejściowego niezależnie. Wyniki konwolucji są następnie łączone w trójwymiarową macierz, reprezentującą zdemozaikowany obraz. Wspólnie stosują również konwolucję 2D i proces tworzenia zdemozaikowanego obrazu poprzez połączenie kanałów. Wartości wag w maskach decydują o tym, w jaki sposób piksele są przekształcane, a różnice w maskach wpływają na ostateczny rezultat demozaikowania.

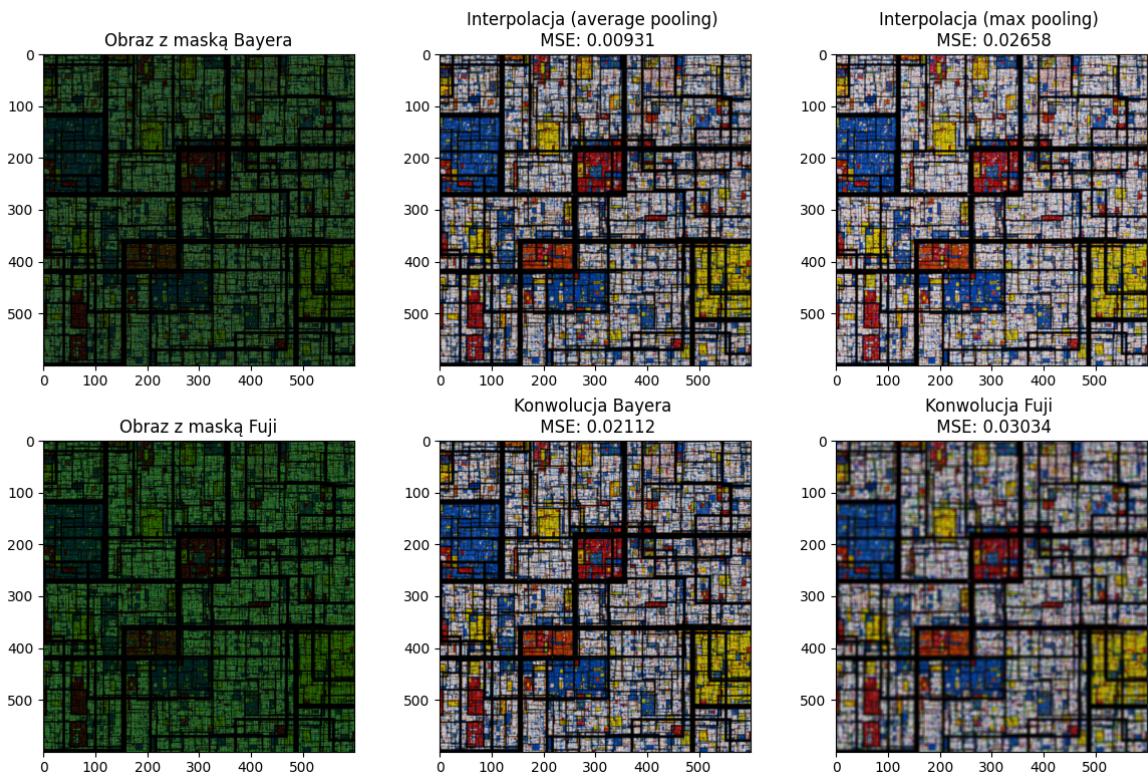
3.3 Porównanie jakości demozaikowania



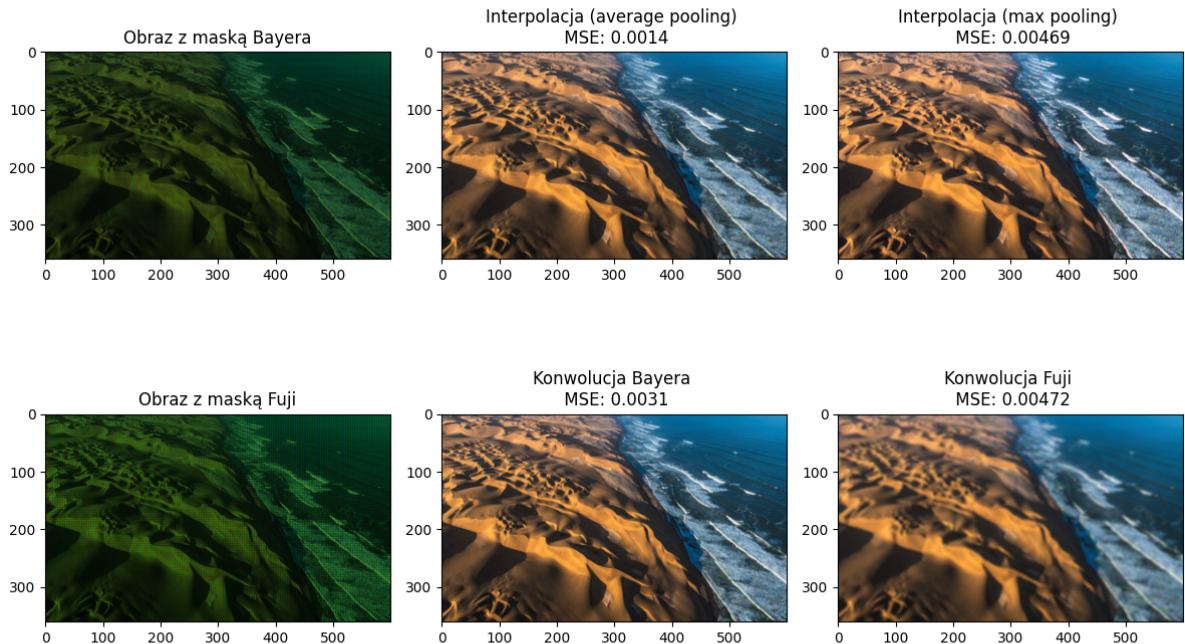
Rysunek 1: Porównanie demozaikowania różnymi metodami z oryginalnym obrazem przy użyciu wskaźnika MSE dla pliku "circle.npy".



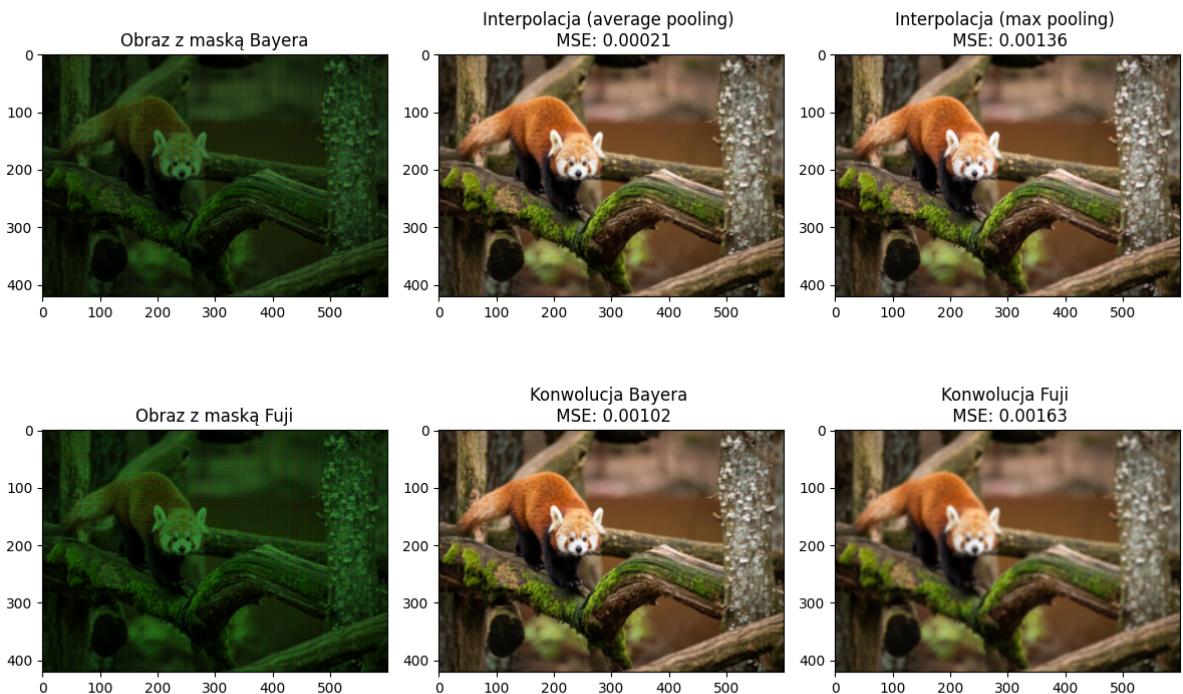
Rysunek 2: Porównanie demozaikowania różnymi metodami z oryginalnym obrazem przy użyciu wskaźnika MSE dla pliku "milky-way.npy".



Rysunek 3: Porównanie demozaikowania różnymi metodami z oryginalnym obrazem przy użyciu wskaźnika MSE dla pliku "mond.npy".



Rysunek 4: Porównanie demozaikowania różnymi metodami z oryginalnym obrazem przy użyciu wskaźnika MSE dla pliku "namib.npy".



Rysunek 5: Porównanie demozaikowania różnymi metodami z oryginalnym obrazem przy użyciu wskaźnika MSE dla pliku "panda.npy".

W przypadku metody interpolacji (average pooling), wartości MSE są generalnie niskie, co sugeruje stosunkowo dobrą dokładność tej metody. Niemniej jednak, obserwuje się pewne zróżnicowanie skuteczności w zależności od rodzaju obrazu. Interpolacja (max pooling), chociaż podobna do average pooling, charakteryzuje się nieco wyższymi wartościami MSE, co może sugerować utratę informacji szczegółowych i ograniczenia dokładności. Obydwie te metody są stosunkowo proste i szybkie do implementacji, ale ich skuteczność może zależeć od charakterystyki konkretnego obrazu.

Natomiast metoda konwolucji Bayera wydaje się działać umiarkowanie dobrze, osiągając wartości MSE na poziomie zbliżonym do interpolacji, ale oferując potencjalnie lepszą jakość obrazu. Konwolucja ta jest bardziej zaawansowaną techniką, korzystającą z maski Bayera dostosowanej do pikseli otoczenia. Może być szczególnie skuteczna dla obrazów zawierających struktury o niskiej częstotliwości.

Natomiast konwolucja z maską Fuji, choć podobna do konwolucji Bayera, osiąga nieco wyższe wartości MSE, co może wynikać z różnic w masce Fuji. Różnice te wprowadzają pewne zmiany do procesu przekształcania pikseli, co może wpływać na skuteczność metody w zależności od charakterystyki obrazu.

Ostateczny wybór metody demozaikowania powinien zależeć od konkretnego przypadku, biorąc pod uwagę specyfikę obrazu i pożądanego poziomu dokładności. Interpolacje są szybkie i proste, ale mogą być mniej dokładne, zwłaszcza dla bardziej złożonych obrazów. Z kolei konwolucje oferują większą precyzję, jednak mogą wymagać więcej zasobów obliczeniowych. Eksperymentowanie z różnymi metodami jest kluczowe w celu doboru optymalnej strategii w zależności od konkretnego scenariusza.

4 Zastosowania Konwolucji

4.1 Wykrywanie krawędzi

Do wykrywania krawędzi na obrazach używa się gradientów jasności, gdzie duże zmiany jasności wskazują na krawędzie. W tym celu często stosuje się operator Laplace (L) dla operatory Sobela (S_x , S_y) osi X i Y. Operator Laplace stanowi przybliżenie gradientu, natomiast operatory Sobela są przybliżeniami pochodnych obrazu w osiach X i Y. Poniżej znajdują się maski Laplace'a i Sobela, które zostały wykorzystane w implementacji, a także odpowiadające im fragmenty kodu:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

```
def Laplace(image):
    Laplace = np.array([[0,1,0], [1,-4,1], [0,1,0]])
    result = np.dstack([cv2.filter2D(image[:, :, i], -1, Laplace) for i in range(3)])
    return result

def Sobel(image):
    Sobel_X= np.array([[1,0,-1], [2,0,-2], [1,0,-1]])
    Sobel_Y= np.array([[1,2,1], [0,0,0], [-1,-2,-1]])
    result_X = np.dstack([cv2.filter2D(image[:, :, i], -1, Sobel_X) for i in range(3)])
    result_Y = np.dstack([cv2.filter2D(image[:, :, i], -1, Sobel_Y) for i in range(3)])
    result_XY = result_X + result_Y
    return result_XY
```

Funkcja `Laplace(image)` operuje przy użyciu maski konwolucyjnej, która akcentuje różnice pikseli w otaczającym obszarze, kierując uwagę na zmiany jasności w różnych kierunkach. To podejście umożliwia wykrywanie obszarów charakteryzujących się szybkimi zmianami jasności, co często odpowiada lokalizacji krawędzi na obrazie.

Z kolei funkcja `Sobel(image)` korzysta z masek konwolucyjnych, które wykorzystują różnice w jasności pikseli w danym obszarze obrazu do identyfikacji miejsc, gdzie występują krawędzie. Pracuje ona w dwóch kierunkach – poziomym i pionowym – stosując dwa oddzielne filtry. Dzięki temu możliwe jest wykrywanie krawędzi zarówno w kierunku poziomym, jak i pionowym, co zapewnia kompleksową identyfikację struktur krawędziowych na obrazie.

Oprócz operatorów Laplace'a i Sobela, możemy także wyróżnić operator Scharr (Sc_x , Sc_y) oraz operator Prewitt (P_x , P_y). Poniżej zostały przedstawione ich maski oraz fragmenty implementacji:

$$Sc_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad Sc_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad P_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

```
def Scharr(image):
    Scharr_X = np.array([[-3,0,3], [-10,0,10], [-3,0,3]])
    Scharr_Y = np.array([[-3,-10,-3], [0,0,0], [3,10,3]])
    result_X = np.dstack([cv2.filter2D(image[:, :, i], -1, Scharr_X) for i in range(3)])
    result_Y = np.dstack([cv2.filter2D(image[:, :, i], -1, Scharr_Y) for i in range(3)])
    result_XY = result_X + result_Y
    return result_XY

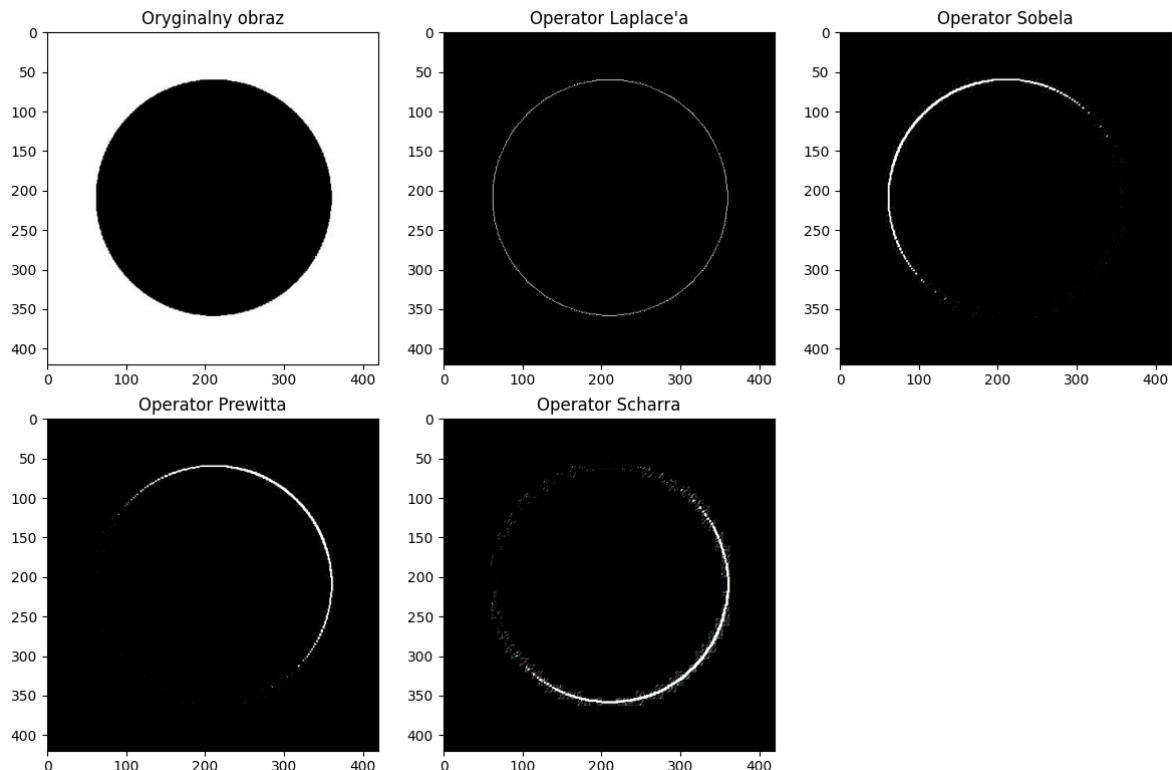
def Prewitt(image):
    Prewitt_X = np.array([[-1,0,1], [-1,0,1], [-1,0,1]])
    Prewitt_Y = np.array([[1,1,1], [0,0,0], [-1,-1,-1]])
    ...
    return result_XY
```

W rzeczywistości, operatory Scharr i Prewitt posiadają bardzo zbliżone implementacje do operatora Sobela. Funkcje korzystają z tej samej metody `cv2.filter2D` do przeprowadzenia operacji konwolucji, a wyniki są następnie łączone w trójwymiarową tablicę przy użyciu np. `dstack`. W każdym przypadku kanały obrazu są przetwarzane niezależnie, a wyniki konwolucji są sumowane. Kluczową różnicą między funkcjami `Sobel(image)`, `Scharr(image)` i `Prewitt(image)` leży w zastosowanych filtrach krawędziowych, które decydują o sposobie detekcji krawędzi na obrazie.

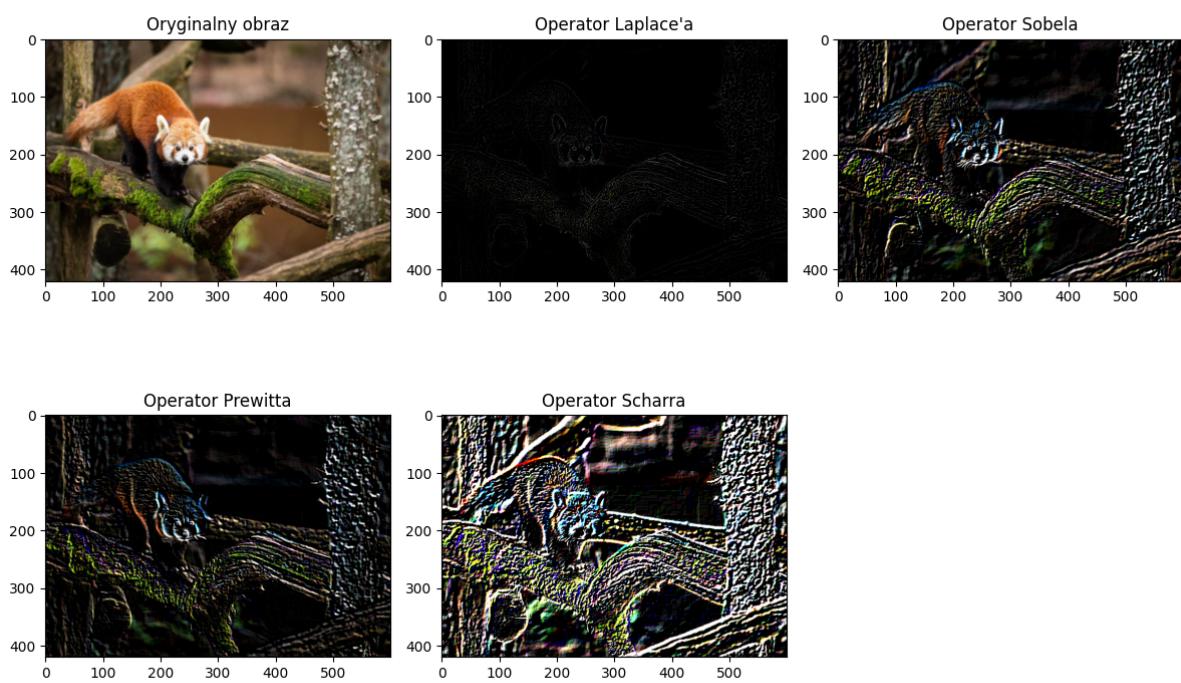
Filtры Scharr'a to wyważone wersje filtrów Sobela, charakteryzujące się równomiernym wzmacnieniem krawędzi. Ze względu na bardziej złożone wagie w maskach konwolucyjnych, oferują one lepszą wrażliwość w detekcji krawędzi, zwłaszcza w sytuacjach, gdzie zachowanie proporcji jest istotne.

Z kolei filtry Prewitta, choć podobne do Sobela, różnią się wzorcem wag w dwóch maskach konwolucyjnych dla osi X i Y. Są stosunkowo proste, a zastosowanie różnych wag może prowadzić do innych rezultatów w detekcji krawędzi w porównaniu do operatora Sobela.

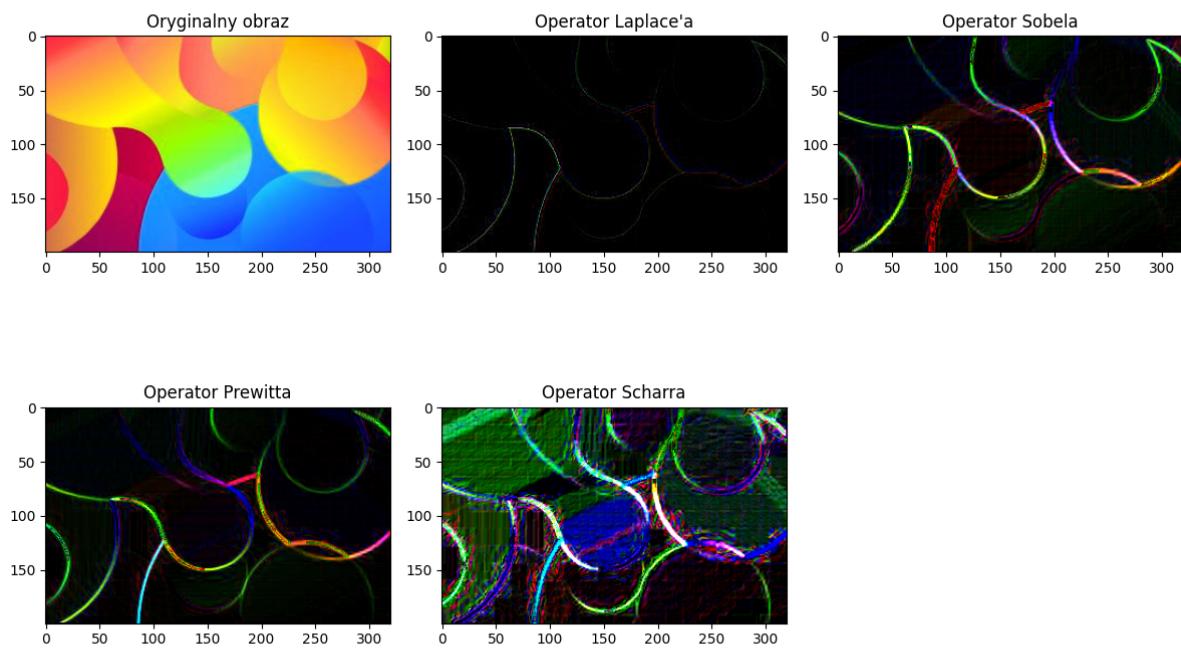
Poniżej przedstawiono rezultaty zastosowania tych filtrów na plikach.



Rysunek 6: Porównanie operatorów wykrywania krawędzi dla pliku "circle.npy".



Rysunek 7: Porównanie operatorów wykrywania krawędzi dla pliku "panda.npy".



Rysunek 8: Porównanie operatorów wykrywania krawędzi dla pliku "Colors.jpg".

Analizując obrazy wynikowe, można zauważyc, że operator Laplacea skutecznie identyfikuje krawędzie charakteryzujące się wyraźnymi zmianami jasności, lecz jednocześnie generuje pewne zakłócenia w postaci szumów. W kontekście pliku "circle.npy" operator efektywnie identyfikuje krawędzie obiektu, choć może również prowadzić do fałszywych detekcji w obszarach o mniejszych zmianach jasności.

Operator Sobela wykazuje się efektywnością w detekcji krawędzi o różnych orientacjach. W plikach "panda.npy" i "Colors.jpg" skutecznie identyfikuje krawędzie obiektów oraz różnice w jasności na tle, co przyczynia się do jego powszechnego stosowania w praktyce.

Operator Prewitta osiąga zbliżoną skutecznosć do Sobela w detekcji krawędzi, jednak subtelne różnice w wagach mogą prowadzić do odmiennych rezultatów. W przypadku plików "circle.npy" i "Colors.jpg" również efektywnie identyfikuje krawędzie.

Operator Scharr, charakteryzujący się wyższą wrażliwością na krawędzie o mniejszych zmianach gradientu, skutecznie identyfikuje krawędzie w plikach "circle.npy" i "panda.npy", lecz może być bardziej podatny na szumy.

Ostateczny wybór konkretnego operatora zależy od charakterystyki obrazu oraz oczekiwanych rezultatów, biorąc pod uwagę różnice wrażliwości na specyficzne cechy krawędziowe oraz wpływ szumów.

4.2 Rozmywanie

Aby uzyskać efekt rozmycia obrazu, stosuje się jądro uśredniające, podobne do tego używanego przy skalowaniu, przy czym krok konwolucji jest równy 1. Alternatywnie, można zastosować jądro ważone rozkładem normalnym, co nazywane jest rozmyciem Gaussowskim. Przykładowe jądro Gaussowskie, oznaczane symbolem G , jest zdefiniowane jako:

$$G = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Implementacja tego jądra znajduje się w funkcji `Gaussian(image)`. Funkcja ta używa znormalizowanego jądra Gaussowskiego do przeprowadzenia operacji rozmycia na każdym kanale obrazu, co prowadzi do wygładzenia i redukcji szumów. Ostateczny wynik to obraz poddany operacji rozmycia Gaussowskiego. Poniżej przedstawiono implementację tego kodu.

```
def Gaussian(image):
    kernel = (1/16) * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])

    result = np.dstack([cv2.filter2D(image[:, :, i], -1, kernel) for i in range(3)])
    return result
```

Istnieje jednak możliwość rozbudowy funkcji Gaussa, umożliwiając zainicjalizowanie rozmycia z większym rozmiarem jądra. W odróżnieniu od wcześniejszego podejścia, gdzie jądro było stałe, w tej implementacji można elastycznie dostosować rozmiar jądra za pomocą parametru `kernel_size`. Poniżej zamieszczono fragment kodu, który jest odpowiedzialny za implementację tej funkcji.

```
def Gaussian_blur(image, kernel_size):
    kernel = np.fromfunction(
        lambda x, y: (1/(2*np.pi*(kernel_size/2)**2)) * np.exp(-((x-(kernel_size-1)/2)
        **2 + (y-(kernel_size-1)/2)**2) / (2.0*(kernel_size/2)**2)),
        (kernel_size, kernel_size)
    )

    kernel = kernel / np.sum(kernel)

    result = np.dstack([cv2.filter2D(image[:, :, i], -1, kernel) for i in range(3)])
    return result
```

W pierwszym kroku generowane jest jądro dynamicznie przy użyciu funkcji Gaussa dwuwymiarowego. Funkcja ta jest zdefiniowana jako wyrażenie lambda, a jej wzory służą do obliczania wartości jądra na podstawie odległości od środka. Rozmiar generowanego jądra kontrolowany jest poprzez parametr `kernel_size`, co umożliwia dostosowanie operacji rozmycia do konkretnych potrzeb.

Następnie jądro jest normalizowane poprzez podzielenie go przez sumę wszystkich jego elementów. Normalizacja jest kluczowa, aby zapewnić, że suma wartości jądra wynosi 1. Dzięki temu można utrzymać jasność obrazu, nawet po zastosowaniu operacji rozmycia.

Sam proces przetwarzania obrazu odbywa się przy użyciu filtra 2D z biblioteki OpenCV (`cv2`). Pętla iteruje po trzech kanałach (RGB) obrazu, a każdy kanał poddawany jest konwolucji z wcześniej zdefiniowanym jądrem Gaussowskim.

Wyniki konwolucji dla poszczególnych kanałów są następnie łączone w jedną trójwymiarową tablicę za pomocą funkcji `np.dstack`. Finalnym rezultatem jest obraz poddany operacji rozmycia Gaussowskiego, a elastyczność funkcji pozwala na dostosowanie rozmiaru jądra, co wpływa na stopień rozmycia obrazu. To podejście umożliwia bardziej precyzyjną kontrolę nad procesem rozmycia, zgodnie z konkretnymi oczekiwaniami użytkownika.

Oprócz klasycznego rozmycia Gaussowskiego istnieje alternatywna metoda wykorzystująca rozmycie uśredniające. Poniżej przedstawiono funkcję `Average_Blurr`, która implementuje to rozwiązanie:

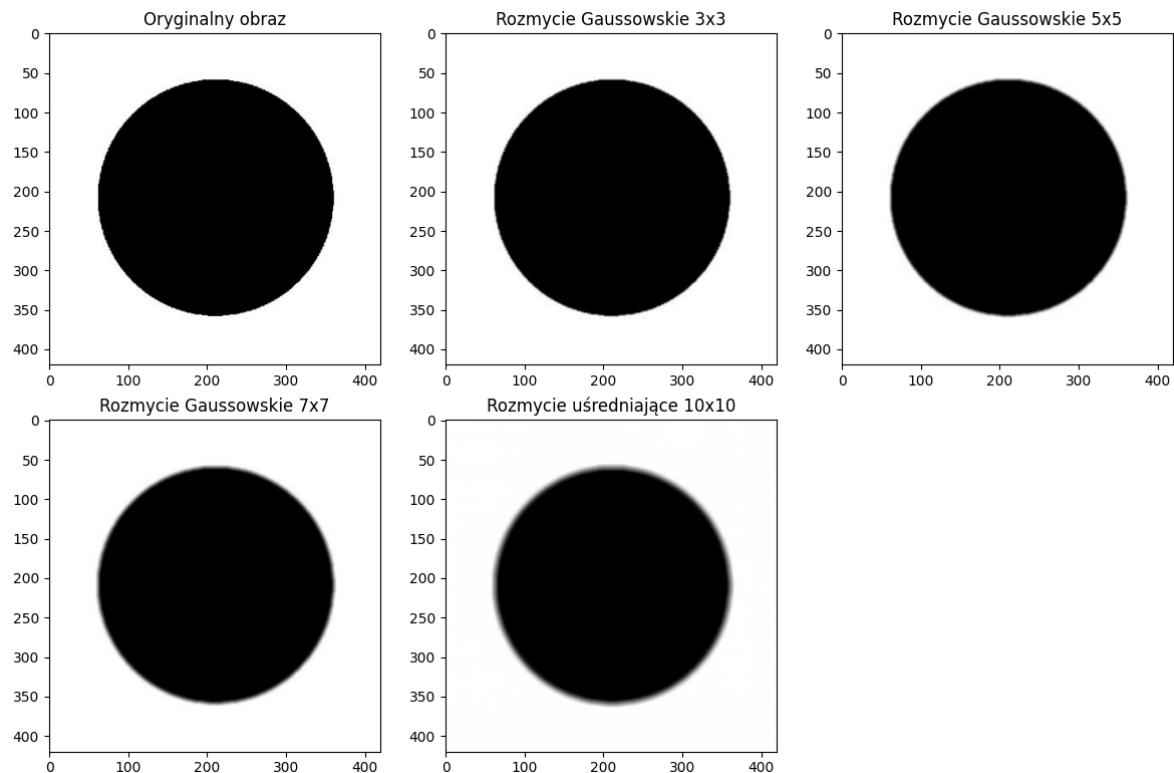
```
def Average_Blurr(image, size):
    filter = np.ones((size, size)) / size ** 2

    result = np.dstack([cv2.filter2D(image[:, :, i], -1, filter) for i in range(3)])
    return result
```

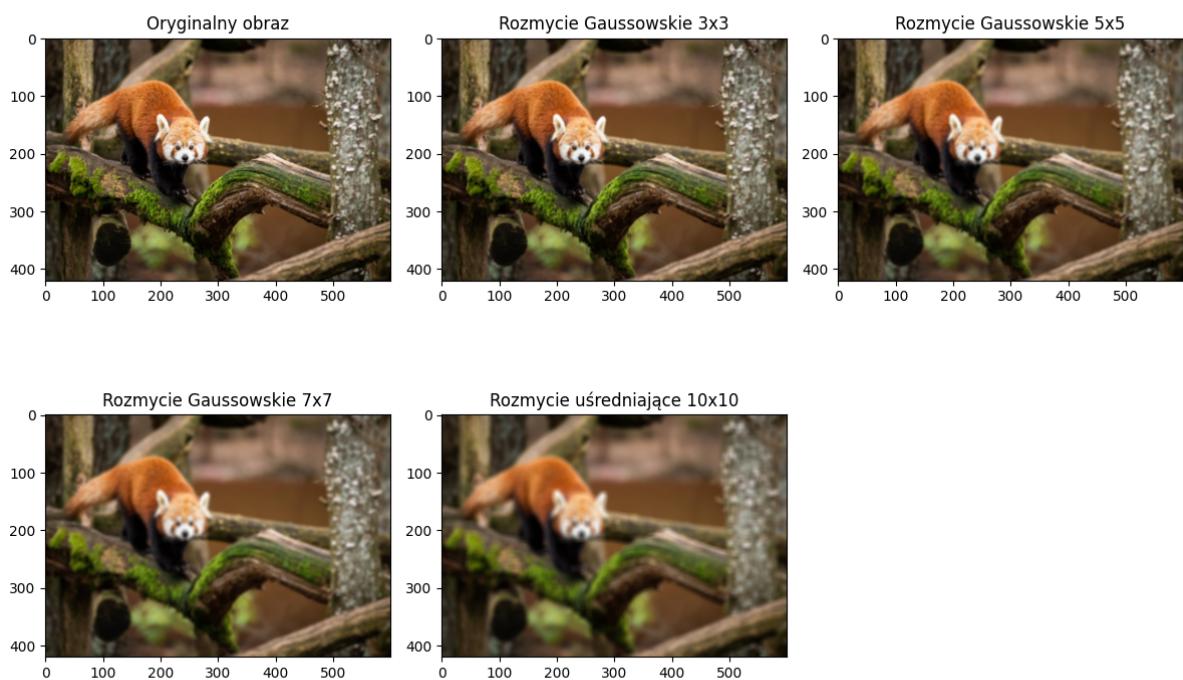
W tej funkcji używane jest jądro uśredniające, reprezentowane przez macierz wypełnioną jedynkami, podzieloną przez kwadrat rozmiaru filtra. Proces przetwarzania obrazu polega na zastosowaniu konwolucji zdefiniowanego filtra na każdym kanale obrazu RGB. Ostateczny wynik to obraz poddany rozmyciu uśredniającemu, co prowadzi do efektu wygładzenia.

Warto zaznaczyć, że rozmycie uśredniające polega na przybliżeniu każdego piksela poprzez średnią wartość pikseli w jego otoczeniu. Jest to prostsza operacja niż rozmycie Gaussowskie, ale może być skuteczna w przypadku pewnych zastosowań, zwłaszcza gdy ważne jest szybkie i łatwe rozmycie obrazu.

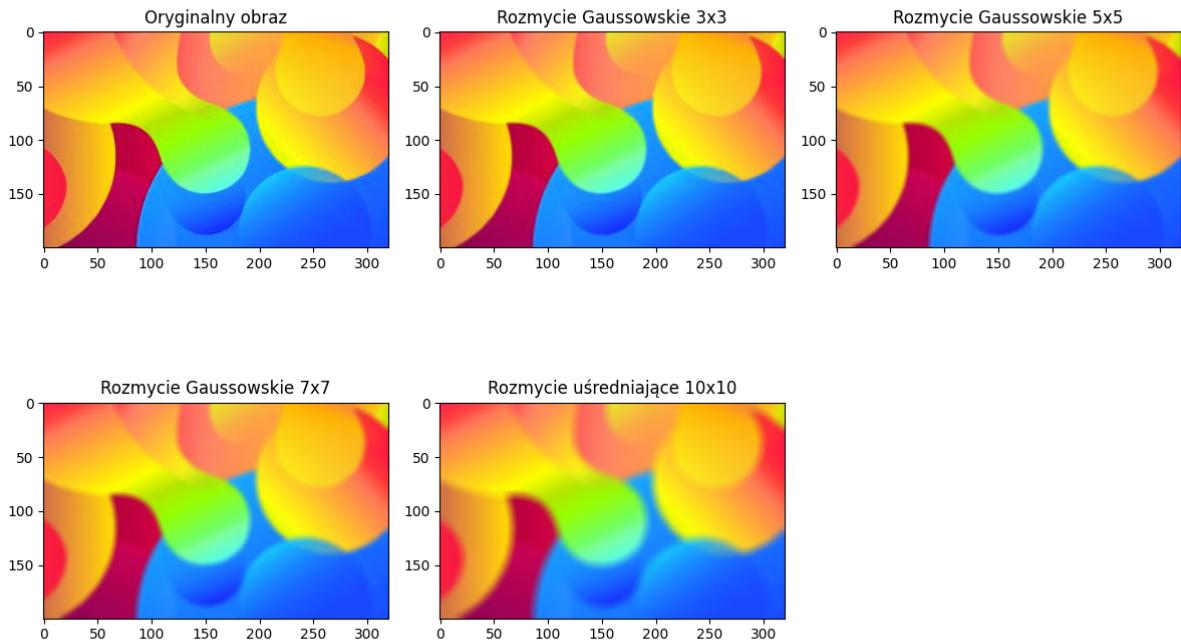
Poniżej przedstawiono rezultaty zastosowania opisanych funkcji przetwarzania obrazu na wybranych plikach graficznych, ukazujące różnice pomiędzy operacjami rozmycia Gaussowskiego, uśredniającego oraz ich różnymi wariantami.



Rysunek 9: Porównanie metod rozmywania obrazów dla pliku "circle.npy".



Rysunek 10: Porównanie metod rozmywania obrazów dla pliku "panda.npy".



Rysunek 11: Porównanie metod rozmywania obrazów dla pliku "Colors.jpg".

Otrzymane wyniki po zastosowaniu różnych metod rozmywania obrazów pozwalają stwierdzić, że funkcja `Gaussian(image)`, wykorzystująca stałe jądro Gaussowskie, skutecznie realizuje zadanie wygładzania obrazu, zwłaszcza w obszarach o dużej zmienności pikseli. Rezultaty są widoczne we wszystkich przetworzonych plikach. Funkcja ta skutecznie redukuje detale i eliminuje drobne szумy, co jest korzystne w kontekście uzyskiwania estetycznych efektów wizualnych.

Rozbudowa funkcji do `Gaussian_blur(image, kernel_size)` pozwala na dynamiczne dostosowywanie rozmiaru jądra Gaussowskiego. To podejście wprowadza większą elastyczność. W zależności od ustawionego parametru `kernel_size`, uzyskujemy różne stopnie rozmycia, co pozwala dostosować operację do konkretnych wymagań i charakterystyki obrazu.

Z kolei funkcja `Average_Blur(image, size)` wykorzystująca rozmycie uśredniające prezentuje prostsze podejście. Jest efektywna w szybkim rozmyciu obrazu, co jest szczególnie zauważalne na plikach. Jednakże, w porównaniu do rozmycia Gaussowskiego, może skutkować większym zachowaniem szczegółów, co może być zarówno zaletą, jak i wadą w zależności od kontekstu zastosowania.

Warto również zauważać, że różnice między metodami są bardziej widoczne w plikach o zróżnicowanej zawartości, gdzie istnieją obszary o różnych poziomach szczegółowości.

4.3 Wyostrzanie

Wyostrzanie obrazu to proces, który wykorzystuje kombinację rozmycia i detekcji krawędzi. Rozmywanie lokalnie uśrednia obraz, eliminując wysokie częstotliwości, podczas gdy detekcja krawędzi usuwa niskie częstotliwości. W efekcie wyostrzania podnosi się wysokie częstotliwości, pozostawiając niskie nie-naruszone. Poniżej przedstawiono przykładowe jądro wyostrzające, oznaczone jako W :

$$W = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Implementację tej operacji znajdujemy w funkcji `Sharpen(image)`:

```
def Sharpen(image):
    Sharp = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])

    result = np.dstack([cv2.filter2D(image[:, :, i], -1, Sharp) for i in range(3)])
    return result
```

Funkcja ta korzysta z zdefiniowanego jądra wyostrzającego, `Sharp`, i przeprowadza konwolucję na każdym kanale obrazu RGB za pomocą funkcji `cv2.filter2D`. Ostateczny rezultat to obraz poddany operacji wyostrzania, co może prowadzić do bardziej zaznaczonych krawędzi i wyraźniejszych szczegółów.

W ramach poszerzania naszych możliwości wyostrzania obrazu, warto wspomnieć o funkcji `Unsharp_masking(image, kernel_size, strength=1.5)`. To narzędzie stosuje technikę maskowania nieostrym (ang. unsharp masking), gdzie korzysta z rozmycia Gaussowskiego do generowania maski wyostrzającej. Poniżej zamieszczono implementację tej funkcji.

```
def Unsharp_masking(image, kernel_size, strength=1.5):
    blurred = Gaussian_blur(image, kernel_size)

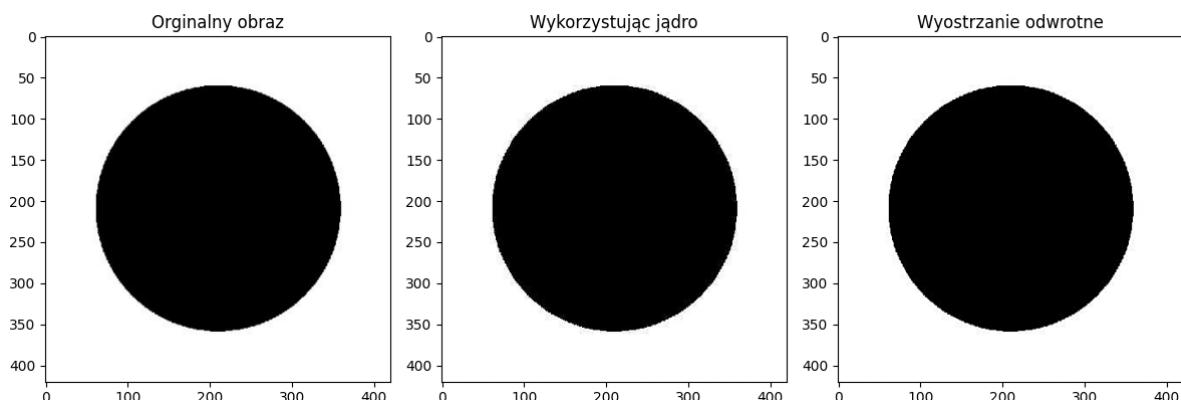
    result = image + strength * (image - blurred)
    return result
```

Najpierw, obraz jest rozmywany przy użyciu funkcji `Gaussian_blur(image, kernel_size)`, gdzie `kernel_size` kontroluje stopień rozmycia. Następnie, na podstawie oryginalnego obrazu i jego rozmytej wersji, generowana jest maska wyostrzająca. Parametr `strength` dostosowuje siłę wyostrzania.

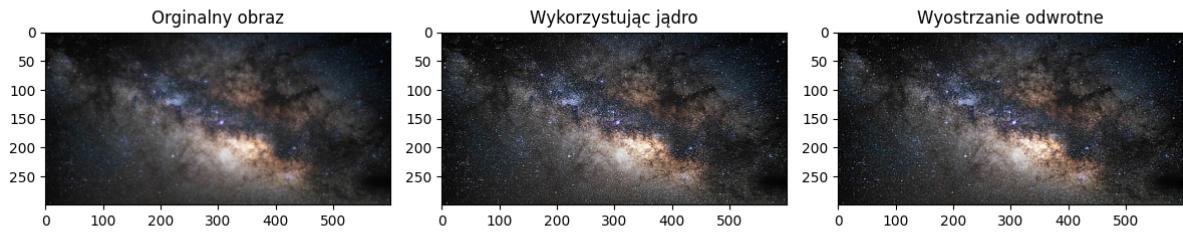
Ostateczny wynik to suma oryginalnego obrazu i iloczynu różnicy między oryginałem a jego wersją rozmytą, pomnożoną przez siłę wyostrzania.

Ta metoda pozwala na subtelne wyostrzenie krawędzi i detali w obrazie, oferując dodatkową elastyczność w dostosowywaniu efektu w zależności od konkretnych potrzeb.

W celu bardziej czytelnego zobrazowania efektów działania tych programów, poniżej znajduje się porównanie ich wyników.



Rysunek 12: Porównanie metod wyostrzania obrazów dla pliku "circle.npy".



Rysunek 13: Porównanie metod wyostrzania obrazów dla pliku "milky-way.npy".



Rysunek 14: Porównanie metod wyostrzania obrazów dla pliku "panda.npy".

Funkcja `Sharpen` intensywnie zaznacza krawędzie, co może być korzystne, gdy istotne są wyraziste kontury obiektów. Natomiast `Unsharp_masking` zapewnia bardziej subtelne wyostrzenie, skupiając się na detaliach i zachowując naturalność obrazu.

Obie metody umożliwiają dostosowanie siły wyostrzania, jednak wybór zależy od konkretnego zastosowania. `Sharpen` nadaje się do sytuacji, gdzie priorytetem są wyraziste krawędzie, podczas gdy `Unsharp_masking` może być bardziej odpowiednia do subtelnej poprawy detali.

5 Wnioski

W kontekście metod demozaikowania, obserwowane wyniki wskazują, że interpolacja, zwłaszcza po-przez zastosowanie metody `average pooling`, cechuje się niskimi wartościami MSE, co sugeruje jej relatywnie wysoką dokładność. Niemniej jednak, zaobserwowano pewne zróżnicowanie skuteczności w zależności od charakterystyki obrazu. W przypadku konwolucji Bayera, metoda ta wydaje się działać umiarkowanie dobrze, osiągając wyniki porównywalne do interpolacji, przy potencjalnie lepszej jakości obrazu. Warto zauważyć, że konwolucja Bayera może szczególnie efektywnie działać dla obrazów zawierających struktury o niskiej częstotliwości. Z kolei konwolucja z maską Fuji, choć podobna do konwolucji Bayera, uzyskuje nieco wyższe wartości MSE, co może być wynikiem różnic w zastosowanej masce, wprowadzając pewne zmienne do procesu transformacji pikseli.

W przypadku operatorów detekcji krawędzi, operator Laplacea efektywnie identyfikuje krawędzie o wyraźnych zmianach jasności, lecz generuje pewne zakłócenia w postaci szumów. Operator Sobela wykazuje się efektywnością w detekcji krawędzi o różnych orientacjach. Operator Prewitta osiąga zbliżoną skuteczność, ale subtelne różnice w wagach mogą wpływać na rezultaty. Z kolei operator Scharra, charakteryzujący się wyższą wrażliwością na krawędzie o mniejszych zmianach gradientu, skutecznie identyfikuje krawędzie, ale może być bardziej podatny na szумy.

Przeanalizowanie wyników różnych metod rozmywania obrazów pozwala stwierdzić, że funkcja `Gaussian(image)` zastosowana z użyciem stałego jądra Gaussowskiego skutecznie realizuje zadanie wygładzania obrazu, eliminując detale i drobne szumy. Rozbudowa funkcji do `Gaussian_blur(image, kernel_size)` pozwala na dynamiczne dostosowywanie rozmiaru jądra, co wprowadza większą elastyczność w procesie. Z kolei funkcja `Average Blur(image, size)`, choć prostsza, jest efektywna w szybkim rozmyciu obrazu, zachowując więcej szczegółów niż rozmycie Gaussowskie. Różnice między tymi metodami są bardziej zauważalne w plikach o zróżnicowanej zawartości, gdzie istnieją obszary o różnych poziomach szczegółowości.

Podsumowując kwestie wyostrzania obrazu, funkcja Sharpen intensywnie zaznacza krawędzie, co może być korzystne, gdy istotne są wyraziste kontury obiektów. Z kolei `Unsharp_masking` zapewnia bardziej subtelne wyostrzenie, skupiając się na detalach i zachowując naturalność obrazu. Obydwie metody pozwalają na dostosowanie siły wyostrzania, z wyborem zależnym od konkretnego zastosowania. Sharpen nadaje się do sytuacji, gdzie priorytetem są wyraziste krawędzie, podczas gdy `Unsharp_masking` może być bardziej odpowiedni do subtelnej poprawy detali.