

# Biometria - Projekt 1

## Aplikacja do przetwarzania i analizy obrazów

Michał Pytel, Jakub Półtorak

March 2025

### Spis treści

<b>1 Wstęp</b>	<b>3</b>
<b>2 Opis aplikacji</b>	<b>3</b>
2.1 Implementacja i środowisko . . . . .	3
2.1.1 Wykorzystane biblioteki . . . . .	3
2.2 Struktura aplikacji . . . . .	4
2.3 Interfejs użytkownika . . . . .	4
2.3.1 Główne elementy interfejsu . . . . .	4
2.3.2 Interakcja z użytkownikiem . . . . .	5
<b>3 Implementacja i analiza metod przetwarzania obrazów</b>	<b>5</b>
3.1 Reprezentacja obrazów w aplikacji . . . . .	5
3.2 Metody konwersji do skali szarości . . . . .	5
3.2.1 Metoda luminancji (luminance) . . . . .	5
3.2.2 Metoda jasności (lightness) . . . . .	6
3.2.3 Metoda średniej (average) . . . . .	6
3.2.4 Porównanie metod konwersji do skali szarości . . . . .	7
3.3 Regulacja jasności i kontrastu . . . . .	8
3.3.1 Regulacja jasności . . . . .	8
3.3.2 Regulacja kontrastu . . . . .	8
3.3.3 Wyniki regulacji jasności i kontrastu . . . . .	9
3.4 Inwersja kolorów . . . . .	9
3.5 Binaryzacja . . . . .	10
3.5.1 Wpływ progu binaryzacji na wynik . . . . .	10
3.6 Filtry splotowe . . . . .	11
3.6.1 Funkcja bazowa do aplikacji filtrów . . . . .	11
3.6.2 Wystrzeganie obrazu . . . . .	11
3.6.3 Filtr uśredniający (Box Blur) . . . . .	12
3.6.4 Filtr Gaussa . . . . .	13
3.6.5 Filtr medianowy . . . . .	13
3.6.6 Detekcja krawędzi . . . . .	14

3.6.7	Porównanie filtrów splotowych . . . . .	15
3.7	Filtr niestandardowy . . . . .	15
3.8	Analiza obrazu . . . . .	16
3.8.1	Histogram . . . . .	16
3.8.2	Projekcje . . . . .	16
<b>4</b>	<b>Optymalizacja wydajności</b>	<b>17</b>
4.1	Wektoryzacja obliczeń . . . . .	17
4.1.1	Implementacja niewektoryzowana (wolna) . . . . .	17
4.1.2	Implementacja wektoryzowana (szybka) . . . . .	17
4.2	Efektywne zarządzanie pamięcią . . . . .	18
4.3	Optymalizacja filtrów . . . . .	18
<b>5</b>	<b>Wnioski</b>	<b>18</b>
5.1	Porównanie metod przetwarzania obrazów . . . . .	18
5.2	Napotkane problemy . . . . .	18
5.3	Ograniczenia zaimplementowanych metod . . . . .	18
5.4	Możliwe rozszerzenia . . . . .	19

# 1 Wstęp

Tematem pierwszego projektu było przetwarzanie i analiza obrazów. W ramach projektu zaimplementowaliśmy aplikację, która umożliwia zastosowanie różnych metod przetwarzania obrazów - zarówno operacji na pojedynczych pikselach, jak i algorytmów splotowych wykorzystujących maski.

Celem projektu było:

- Praktyczne poznanie różnych technik przetwarzania obrazów
- Implementacja graficznego interfejsu użytkownika pozwalającego na interaktywne zastosowanie tych technik
- Analiza efektywności i ograniczeń różnych metod na przykładzie rzeczywistych obrazów

# 2 Opis aplikacji

## 2.1 Implementacja i środowisko

Aplikacja została zaimplementowana w języku Python, który został wybrany ze względu na:

- Bogaty ekosystem bibliotek do przetwarzania obrazów i tworzenia interfejsów graficznych
- Czytelną i zwięzłą składnię, która pozwala na szybki rozwój aplikacji
- Wsparcie dla operacji wektorowych dzięki bibliotece NumPy, co umożliwia efektywne przetwarzanie dużych zbiorów danych

### 2.1.1 Wykorzystane biblioteki

- **NumPy** - wykorzystana do efektywnych operacji macierzowych i wektorowych, co jest kluczowe dla przetwarzania obrazów. Wybrana ze względu na znaczącą przewagę wydajnościową nad standardowym podejściem iteracyjnym w Pythonie.
- **PyQt5** - użyta do implementacji interfejsu graficznego. Wybrana ze względu na:
  - Rozbudowane komponenty GUI dostosowane do różnych systemów operacyjnych
  - Wsparcie dla zaawansowanych funkcji takich jak obsługa gestów i zdarzeń dotykowych
  - Możliwość dostosowania wyglądu aplikacji
- **Matplotlib** - wykorzystana do wizualizacji histogramów i projekcji obrazów. Wybrana ze względu na elastyczność w tworzeniu wykresów i grafik.

- **OpenCV (cv2)** - uzyta głównie do wczytywania i zapisywania obrazów w różnych formatach oraz konwersji między przestrzeniami barw. OpenCV zapewnia efektywne funkcje niskopoziomowe, które uzupełniają funkcjonalność zaimplementowaną samodzielnie.

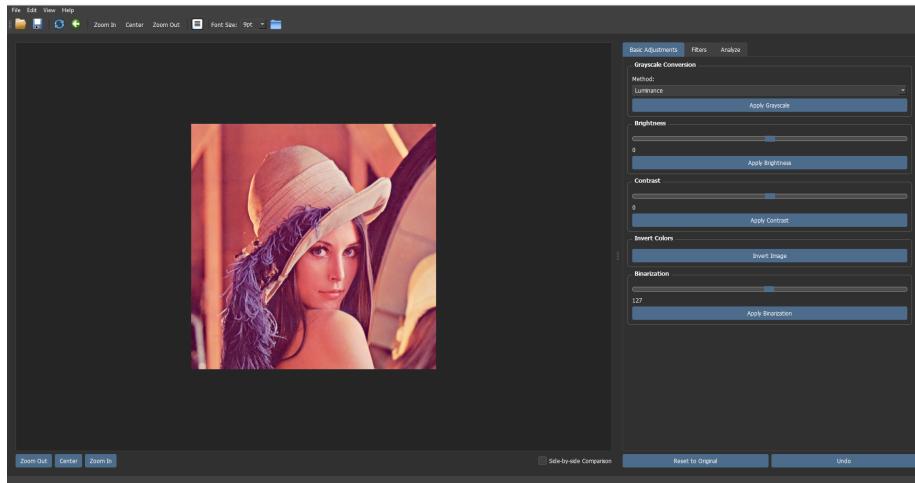
## 2.2 Struktura aplikacji

Aplikacja została podzielona na dwa główne pliki:

- **photo\_editor.py** - zawiera implementację interfejsu graficznego i logikę aplikacji
- **functions.py** - zawiera algorytmy przetwarzania obrazów

Taki podział pozwala na oddzielenie logiki biznesowej od interfejsu użytkownika, co ułatwia testowanie i rozwijanie aplikacji.

## 2.3 Interfejs użytkownika



Rysunek 1: Interfejs aplikacji do przetwarzania obrazów

### 2.3.1 Główne elementy interfejsu

Interfejs aplikacji został zaprojektowany z myślą o intuicyjnej obsłudze i składa się z:

- **Głównego obszaru wyświetletania obrazu** - z funkcjami powiększania, przewijania i porównywania z oryginałem
- **Panelu sterowania** - podzielonego na zakładki tematyczne:

- Podstawowe operacje (konwersja do skali szarości, jasność, kontrast, inwersja, binaryzacja)
- Filtry (wystrzanie, rozmycie, detekcja krawędzi, filtry niestandardowe)
- Analiza (histogram, projekcje)
- **Menu i paska narzędzi** - zapewniających szybki dostęp do funkcji i ustawień
- **Paska statusu** - informującego o aktualnych operacjach i parametrach

### 2.3.2 Interakcja z użytkownikiem

Aplikacja została zaprojektowana z uwzględnieniem różnych form interakcji:

- **Obsługa klawiaturą** - skróty klawiszowe dla najczęściej używanych funkcji
- **Obsługa myszą** - standardowe operacje interfejsu
- **Obsługa gestów** - powiększanie za pomocą gestu pinch-to-zoom na touchpadzie, przesuwanie obrazu

## 3 Implementacja i analiza metod przetwarzania obrazów

### 3.1 Reprezentacja obrazów w aplikacji

W aplikacji obrazy są reprezentowane jako wielowymiarowe tablice NumPy:

- Obrazy RGB: tablice 3D o wymiarach (wysokość, szerokość, 3), gdzie ostatni wymiar reprezentuje kanały R, G, B
- Obrazy w skali szarości: tablice 2D o wymiarach (wysokość, szerokość)
- Wartości pikseli są przechowywane jako liczby całkowite 8-bitowe (0-255)

### 3.2 Metody konwersji do skali szarości

Zaimplementowano trzy metody konwersji obrazów kolorowych do skali szarości:

#### 3.2.1 Metoda luminancji (luminance)

- **Wzór:**  $Y = 0.299R + 0.587G + 0.114B$
- **Implementacja:**

```

1 def grayscale_luminance(img):
2     """Convert RGB to grayscale using luminance method (
3         vectorized)"""
4     return np.dot(img[... , :3] , [0.299 , 0.587 , 0.114]).astype(np.uint8)

```

- **Uzasadnienie:** Ta metoda wykorzystuje ważoną sumę kanałów RGB, która lepiej oddaje percepcję jasności przez ludzkie oko. Współczynniki odzwierciedlają różną wrażliwość oka na poszczególne kolory.
- **Zalety:** Daje wyniki najbardziej zbliżone do percepji ludzkiego oka, zachowuje kontrast między kolorami.
- **Wady:** Wymaga nieco więcej obliczeń niż metoda średniej.

### 3.2.2 Metoda jasności (lightness)

- **Wzór:**  $Y = \frac{\max(R,G,B)+\min(R,G,B)}{2}$
- **Implementacja wektoryzowana:**

```

1 def grayscale_lightness(img):
2     """Convert RGB to grayscale using lightness method (
3         vectorized)"""
4     img_float = img.astype(float)
5     min_values = np.min(img_float , axis=2)
6     max_values = np.max(img_float , axis=2)
7     img_gray = (min_values + max_values) / 2
8     return np.clip(img_gray , 0 , 255).astype(np.uint8)

```

- **Uzasadnienie:** Ta metoda opiera się na średniej z najjaśniejszego i najciemniejszego kanału dla każdego piksela.
- **Zalety:** Prosta koncepcyjnie, zachowuje ekstremalne wartości.
- **Wady:** Może prowadzić do utraty szczegółów w obszarach o średniej jasności, ponieważ uwzględnia tylko skrajne wartości kanałów.
- **Optymalizacja:** Początkowo metoda była zaimplementowana przy użyciu pętli, co skutkowało niską wydajnością dla dużych obrazów. Wektoryzacja znacząco przyspieszyła działanie funkcji.

### 3.2.3 Metoda średniej (average)

- **Wzór:**  $Y = \frac{R+G+B}{3}$
- **Implementacja:**

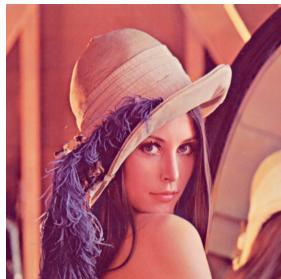
```

1 def grayscale_average(img):
2     """Convert RGB to grayscale using average method (
3         vectorized)"""
4     return np.mean(img, axis=2).astype(np.uint8)

```

- **Uzasadnienie:** Najprostsza metoda konwersji, traktująca wszystkie kanały równoważnie.
- **Zalety:** Bardzo prosta obliczeniowo, łatwa w implementacji.
- **Wady:** Nie uwzględnia różnej wrażliwości ludzkiego oka na poszczególne kolory, co może prowadzić do nieoczekiwanych rezultatów.

### 3.2.4 Porównanie metod konwersji do skali szarości



(a) Obraz oryginalny



(b) Metoda luminancji



(c) Metoda jasności



(d) Metoda średniej

Rysunek 2: Porównanie metod konwersji obrazu do skali szarości

#### Wnioski:

- Metoda luminancji daje najbardziej naturalny wygląd i najlepiej zachowuje kontrast między różnymi kolorami
- Metoda jasności może nadmiernie wzmacniać kontrast w niektórych obszarach

- Metoda średniej może powodować spłaszczenie kontrastu i ujednolicenie obszarów o różnych kolorach, ale podobnej jasności

### 3.3 Regulacja jasności i kontrastu

#### 3.3.1 Regulacja jasności

- **Wzór:**  $I_{out} = I_{in} + value$
- **Implementacja:**

```

1 def brightness(img, value):
2     """Adjust brightness by adding value to all pixels"""
3     return np.clip(img.astype(float) + value, 0, 255).astype
    (np.uint8)

```

- **Zasada działania:** Dodanie stałej wartości do wszystkich pikseli obrazu.
- **Zakres parametru:** Od -255 (całkowite przyśiemnienie) do 255 (całkowite rozjaśnienie).
- **Ograniczenia:** Dla skrajnych wartości może dojść do utraty informacji przez obcięcie wartości do zakresu 0-255.

#### 3.3.2 Regulacja kontrastu

- **Wzór:**  $I_{out} = factor \cdot (I_{in} - 128) + 128$ , gdzie  $factor = \frac{259 \cdot (param+255)}{255 \cdot (259 - param)}$
- **Implementacja:**

```

1 def contrast(img, param):
2     """Adjust contrast using the formula with factor
    calculation"""
3     factor = (259 * (param + 255)) / (255 * (259 - param))
4     return np.clip(factor * (img.astype(float) - 128) + 128,
    0, 255).astype(np.uint8)

```

- **Zasada działania:** Zmiana rozpiętości wartości pikseli wokół środkowej wartości (128).
- **Zakres parametru:** Od -127 (znaczne zmniejszenie kontrastu) do 127 (znaczne zwiększenie kontrastu).
- **Ograniczenia:** Dla wysokich wartości może prowadzić do utraty szczegółów w bardzo jasnych i bardzo ciemnych obszarach.

### 3.3.3 Wyniki regulacji jasności i kontrastu



Rysunek 3: Efekty regulacji jasności i kontrastu

#### Wnioski:

- Regulacja jasności jest prostą operacją, ale może prowadzić do utraty szczegółów przy skrajnych wartościach
- Regulacja kontrastu jest bardziej złożona i może dawać lepsze rezultaty dla obrazów o słabym kontraście
- Kombinacja obu operacji pozwala na uzyskanie optymalnych wyników dla różnych typów obrazów
- Dla obrazów o nierównomiernym oświetleniu te globalne metody mogą nie dawać zadowalających rezultatów - w takich przypadkach lepsze mogą być metody lokalne, jak wyrównanie histogramu

### 3.4 Inwersja kolorów

- **Wzór:**  $I_{out} = 255 - I_{in}$
- **Implementacja:**

```
1 def inverse(img):  
2     """Invert image colors"""  
3     return 255 - img
```

- **Zasada działania:** Zamiana wartości każdego piksela na wartość dopełniającą do 255.
  - **Zastosowania:** Tworzenie efektu negatywu, podkreślanie szczegółów, przygotowanie do dalszego przetwarzania.

### 3.5 Binaryzacja

- **Wzór:**  $I_{out} = \begin{cases} 255 & I_{in} > threshold \\ 0 & I_{in} \leq threshold \end{cases}$
  - **Implementacja:**

```
1 def binarize(img, thresh):
2     """Binarize an image (works on both grayscale and RGB)
3     """
4
5     if len(img.shape) == 3: # RGB image
6         gray = grayscale_luminance(img)
7     else: # Already grayscale
8         gray = img
9
10    return (gray > thresh).astype(np.uint8) * 255
```

- **Zasada działania:** Konwersja obrazu do postaci czarno-białej na podstawie progu.
  - **Zakres parametru:** Od 0 (wszystkie piksele białe) do 255 (wszystkie piksele czarne).
  - **Ograniczenia:** Globalna binaryzacja z pojedynczym progiem może nie dawać dobrych wyników dla obrazów o nierównomiernym oświetleniu lub zróżnicowanym tle.
  - **Zastosowania:** Segmentacja obiektów, przygotowanie do analizy kształtów, rozpoznawanie tekstu.

### 3.5.1 Wpływ progu binaryzacji na wynik

(a) Obraz oryginalny

(b) Próg = 50

(c) Próg = 127

(d) Próg = 200

Rysunek 4: Wpływ progu na wynik binaryzacji

## Wnioski:

- Dobór odpowiedniego progu jest kluczowy dla skutecznej binaryzacji
- Dla obrazów z dobrym kontrastem między obiektem a tłem, binaryzacja daje dobre rezultaty
- Dla obrazów o złożonym tle lub nierównomiernym oświetleniu lepsze wyniki może dać adaptacyjna binaryzacja (z progiem lokalnym)

## 3.6 Filtry splotowe

W aplikacji zaimplementowano kilka rodzajów filtrów splotowych, które działają na zasadzie:

$$I_{out}(x, y) = \sum_{i=-r}^r \sum_{j=-r}^r K(i, j) \cdot I_{in}(x + i, y + j) \quad (1)$$

gdzie:

- $K$  - jądro (maska) filtru
- $r$  - promień maski (dla maski  $3 \times 3$ ,  $r = 1$ )

### 3.6.1 Funkcja bazowa do aplikacji filtrów

Wszystkie filtry splotowe wykorzystują wspólną funkcję bazową:

```
1 def apply_kernel(img, kernel):
2     """Apply a kernel to an image with proper padding"""
3     # Implementacja funkcji aplikuj cęj mask do obrazu
4     # z odpowiednim paddingiem, obs ug typ w danych i
5     # ...
```

### 3.6.2 Wystrzanie obrazu

- **Jądro filtru:**  $K = \begin{bmatrix} 0 & -s & 0 \\ -s & 1+4s & -s \\ 0 & -s & 0 \end{bmatrix}$ , gdzie  $s$  to parametr siły wystrzania

- **Implementacja:**

```
1 def sharpness(img, strength=1):
2     # Center value increases with strength
3     center_value = 1 + 4 * strength
4     # Edge values get more negative with strength
5     edge_value = -1 * strength
```

```

6     kernel = np.array([
7         [0, edge_value, 0],
8         [edge_value, center_value, edge_value],
9         [0, edge_value, 0]
10    ])
11
12
13    new_img = apply_kernel(img, kernel)
14    return new_img

```

- **Zasada działania:** Podkreślenie różnic między pikselem centralnym a jego sąsiadami.
- **Zakres parametru:** Od 0.1 (minimalne wyostrzenie) do 5.0 (bardzo silne wyostrzenie).
- **Ograniczenia:** Przy wysokich wartościach może wzmacniać szum i powodować artefakty.

### 3.6.3 Filtr uśredniający (Box Blur)

- **Jądro filtru:**  $K = \frac{1}{k^2} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$ , gdzie  $k$  to rozmiar jądra

- **Implementacja:**

```

1 def mean_filter(img, k):
2     if k%2 ==0:
3         raise ValueError("k must be an odd number")
4
5     kernel = np.ones((k,k)) / k**2 #element-wise
6
7     new_img = apply_kernel(img, kernel)
8     return new_img

```

- **Zasada działania:** Zastąpienie wartości piksela średnią wartością z jego sąsiedztwa.
- **Zakres parametru:** Nieparzyste wartości od 3 (minimalne rozmycie) do 25 (bardzo silne rozmycie).
- **Ograniczenia:** Rozmywa zarówno szum, jak i krawędzie, powodując utratę szczegółów.
- **Zastosowania:** Redukcja szumu, wygładzanie obrazu, zmniejszanie drobnych detali.

### 3.6.4 Filtr Gaussa

- **Jądro filtru:**  $K(i, j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$ , gdzie  $\sigma$  to odchylenie standardowe
- **Implementacja:**

```
1 def gaussian_kernel(size, sigma, size_y=None):
2     """Create a Gaussian kernel with specified parameters"""
3     # Implementacja generowania jądra Gaussa
4     # ...
5
6 def gaussian_blur(img, k, sigma):
7     kernel = gaussian_kernel(k, sigma=sigma)
8     new_img = apply_kernel(img, kernel)
9     return new_img
```

- **Zasada działania:** Zastosowanie filtru z wagami opartymi na rozkładzie Gaussa.
- **Parametry:**
  - $k$  - rozmiar jądra (od 1 do 25)
  - $\sigma$  - odchylenie standardowe (od 0.1 do 10.0)
- **Zalety:** Bardziej naturalne rozmycie niż filtr uśredniający, lepsze zachowanie krawędzi.
- **Zastosowania:** Redukcja szumu, wygładzanie obrazu, przygotowanie do detekcji krawędzi.

### 3.6.5 Filtr medianowy

- **Zasada działania:** Zastąpienie wartości piksela medianą wartości z jego sąsiedztwa.
- **Implementacja:**

```
1 def median_filter(img, k=3):
2     """Apply a median filter to an image"""
3     # Implementacja filtru medianowego
4     # ...
```

- **Parametr:**  $k$  - rozmiar okna (nieparzyste wartości od 3 do 25).
- **Zalety:** Bardzo skuteczny w usuwaniu szumu impulsowego (typu "sól i pieprz"), lepiej zachowuje krawędzie niż filtry liniowe.
- **Wady:** Może usuwać cienkie linie i drobne detale, wyższa złożoność obliczeniowa niż filtry liniowe.

### 3.6.6 Detekcja krawędzi

W aplikacji zaimplementowano kilka metod detekcji krawędzi:

#### Operator Robertsa

- **Zasada działania:** Obliczenie gradientu obrazu przy użyciu różnic sąsiednich pikseli.
- **Wzór:**  $|\nabla f| = |f(x+1, y) - f(x, y)| + |f(x, y+1) - f(x, y)|$

#### Operator Prewitta

- **Zasada działania:** Detekcja krawędzi przy użyciu masek w 8 kierunkach.
- **Przykładowe maski:**

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad (2)$$

$$G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (3)$$

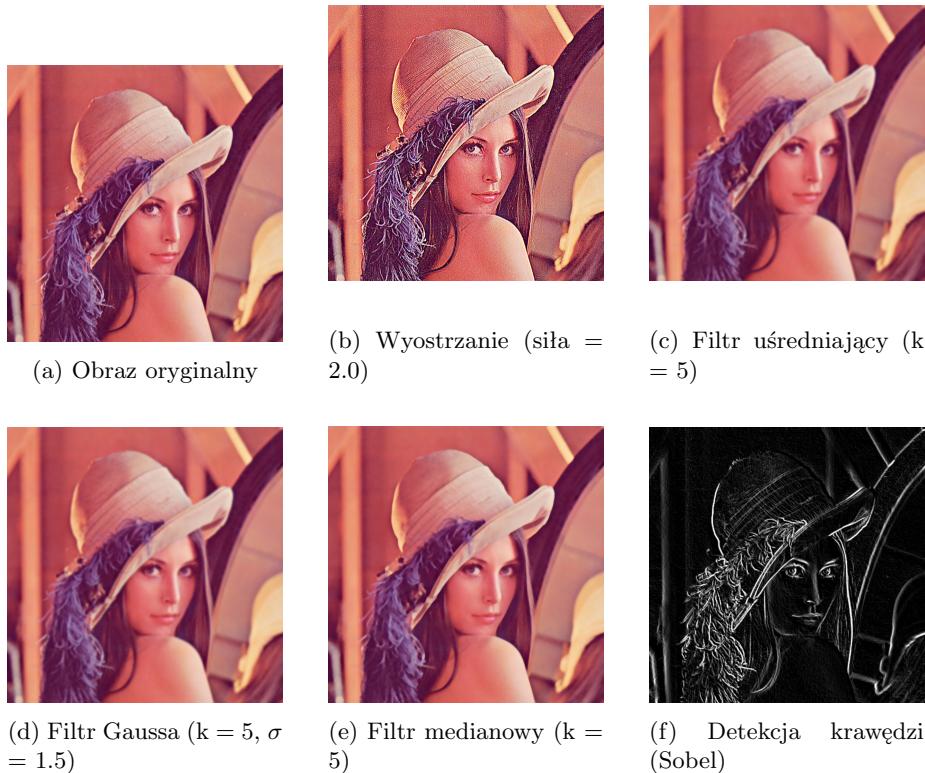
#### Operator Sobela

- **Zasada działania:** Podobny do operatora Prewitta, ale z większymi wagami dla środkowych pikseli.
- **Przykładowe maski:**

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (4)$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (5)$$

### 3.6.7 Porównanie filtrów splotowych



Rysunek 5: Porównanie różnych filtrów splotowych

#### Wnioski:

- Filtr wyostrzający jest skuteczny w podkreślaniu detali, ale może wzmacniać szum
- Filtr uśredniający dobrze redukuje szum, ale znacznie rozmywa krawędzie
- Filtr Gaussa zapewnia bardziej naturalne rozmycie z lepszym zachowaniem krawędzi
- Filtr medianowy jest najskuteczniejszy w usuwaniu szumu impulsowego
- Operatory detekcji krawędzi różnią się czułością i odpornością na szum

## 3.7 Filtr niestandardowy

Aplikacja umożliwia tworzenie własnych masek filtru o różnych rozmiarach ( $3 \times 3$ ,  $5 \times 5$ ,  $7 \times 7$ ) oraz wybór spośród predefiniowanych masek.

- **Implementacja:**

```

1 def apply_custom_kernel(img, kernel):
2     """Apply a custom kernel to an image"""
3     # Check kernel size
4     if kernel.shape not in [(3, 3), (5, 5), (7, 7)]:
5         raise ValueError("Kernel must be 3x3, 5x5, or 7x7")
6
7     # Apply the kernel
8     return apply_kernel(img, kernel)

```

- **Predefiniowane maski:**

- Rozmycie
- Wystrzenie
- Detekcja krawędzi
- Uwypuklenie (emboss)

- **Opcje:** Możliwość normalizacji maski (suma elementów równa 1)

## 3.8 Analiza obrazu

### 3.8.1 Histogram

- **Zasada działania:** Wizualizacja rozkładu intensywności pikseli w obrazie.

- **Implementacja:**

```

1 def plot_histogram(img):
2     # Implementacja wizualizacji histogramu
3     # Obszary zarówno obraz w RGB, jak i w skali
4     szaro_ci
5     # ...

```

- **Zastosowania:** Analiza kontrastu, jasności, ocena efektów przetwarzania obrazu.

### 3.8.2 Projekcje

- **Zasada działania:** Wizualizacja sum intensywności pikseli wzduż osi poziomej i pionowej.

- **Implementacja:**

```

1 def plot_image_with_projections(img):
2     # Compute projections
3     horizontal_projection = np.sum(img, axis=1)    # Sum
4     across rows
5     vertical_projection = np.sum(img, axis=0)      # Sum
6     across columns
7
# Implementacja wizualizacji obrazu z projekcjami
# ...

```

- **Zastosowania:** Analiza rozkładu obiektów na obrazie, wykrywanie linii tekstu, segmentacja znaków.

## 4 Optymalizacja wydajności

### 4.1 Wektoryzacja obliczeń

W aplikacji zastosowano wektoryzację obliczeń przy użyciu NumPy, co pozwoliło na znaczne przyspieszenie przetwarzania obrazów. Przykładem może być optymalizacja funkcji konwersji do skali szarości:

#### 4.1.1 Implementacja niewektoryzowana (wolna)

```

1 def grayscale_lightness_slow(img):
2     height = len(img)
3     width = len(img[0])
4
5     img = img.astype(float)
6     img_gray = np.zeros((height, width), dtype=np.float16)
7
8     for i in range(height):
9         for j in range(width):
10             r, g, b = img[i,j]
11             img_gray[i,j] = (min(r,g,b) + max(r,g,b))/2
12
13     img_gray = np.clip(img_gray, 0, 255).astype(np.uint8)
14     return img_gray

```

#### 4.1.2 Implementacja wektoryzowana (szybka)

```

1 def grayscale_lightness(img):
2     img_float = img.astype(float)
3     min_values = np.min(img_float, axis=2)
4     max_values = np.max(img_float, axis=2)
5     img_gray = (min_values + max_values) / 2
6     return np.clip(img_gray, 0, 255).astype(np.uint8)

```

## 4.2 Efektywne zarządzanie pamięcią

W aplikacji zastosowano techniki efektywnego zarządzania pamięcią, takie jak:

- Używanie odpowiednich typów danych (np. np.uint8 dla obrazów)
- Przeprowadzanie obliczeń w wyższej precyzyji (float), a następnie konwersja wyników do oryginalnego typu
- Unikanie niepotrzebnego kopiowania danych

## 4.3 Optymalizacja filtrów

Operacje splotowe są mało zoptymalizowane poprzez potrzebę liczenia piksel po pikselu, dającą złożoność obliczeniową na poziomie  $O(n^2)$

# 5 Wnioski

## 5.1 Porównanie metod przetwarzania obrazów

- Operacje punktowe (jasność, kontrast, inwersja) są proste obliczeniowo, ale ich skuteczność jest ograniczona do globalnych zmian w obrazie.
- Filtry splotowe (wystrzanie, rozmycie) umożliwiają bardziej zaawansowane przetwarzanie, ale wymagają więcej obliczeń.
- Filtry nieliniowe (medianowy) mogą być bardziej skuteczne w pewnych zastosowaniach, ale są bardziej wymagające obliczeniowo.

## 5.2 Napotkane problemy

- Obsługa krawędzi obrazu przy operacjach splotowych - rozwiązano poprzez zastosowanie paddingu typu "reflect"
- Wydajność dla dużych obrazów - rozwiązano poprzez wektoryzację obliczeń
- Zapewnienie spójnej obsługi obrazów RGB i w skali szarości - rozwiązano poprzez odpowiednie sprawdzanie wymiarów i konwersje z powodu braku możliwości

## 5.3 Ograniczenia zaimplementowanych metod

- Złożoność dla filtrów
- Prosta regulacja jasności i kontrastu może powodować utratę szczegółów
- Filtry wystrzajające mogą wzmacniać szum obecny w obrazie

## 5.4 Możliwe rozszerzenia

- Implementacja adaptacyjnej binaryzacji (z lokalnym progiem)
- Dodanie operacji morfologicznych (dylacja, erozja)
- Implementacja wyrównania histogramu dla poprawy kontrastu
- Dodanie bardziej zaawansowanych metod redukcji szumu

## Literatura

- [1] dr inż. Janusz Rafałko "Biometria - wykłady 1,2,3,4"
- [2] OpenCV Documentation, <https://docs.opencv.org/>
- [3] PyQt Documentation, <https://doc.qt.io/qtforpython/>
- [4] NumPy Documentation, <https://numpy.org/doc/>
- [5] R. C. Gonzalez, R. E. Woods, "Digital Image Processing", 4th Edition, Pearson, 2018
- [6] R. Szeliski, "Computer Vision: Algorithms and Applications", Springer, 2021