

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Matematyczny
specjalność: ogólna

Klaudia Bała

**Opis i implementacja niektórych podstawowych
algorytmów przetwarzania obrazu**

Praca licencjacka
napisana pod kierunkiem
dr hab. Macieja Paluszyńskiego

Wrocław 2024

Spis treści

1 Wstęp	3
2 Operacje punktowe	4
2.1 Zmiana jasności	5
2.2 Manipulacja kontrastem	6
3 Przekształcenia geometryczne	9
3.1 Obrót	9
3.2 Skalowanie	18
4 Filtry, czyli sploty	19
4.1 Rozmycie obrazu	20
4.2 Wyostrzanie obrazu	22
5 Interaktywny program	25
6 Wykrywanie okręgów na obrazie	28
7 Kody implementacji	41
Bibliografia	49

1 Wstęp

Tematem pracy są rozważania nad podstawowymi algorytmami przetwarzania obrazu, a dokładniej ich podstawami matematycznymi. Opracowanie jest próbą rozwikłania działania pewnych operacji, z których często korzystamy w codziennym życiu podczas obróbki fotografii. Niekoniecznie omówione tutaj sposoby patrzenia na każdą z tych operacji są domyślne w programach. W rozważaniach zostaną wprowadzone pewne uproszczenia oraz pominięte pewne treści, które nie pomogłyby w zrozumieniu podstawowego działania tych procesów, a jedynie niepotrzebnie skomplikowałyby zagadnienie. W tekście zostaną poruszone typowe operacje, takie jak zmiana jasności czy kontrastu, obrót, skalowanie, rozmywanie oraz wyostrzanie obrazu. W pracy zawarto także omówienie i implementację algorytmu wykrywającego okręgi na obrazie, używając transformaty Hougha. W rozważaniach pracujemy na obrazach o rozszerzeniu .jpg, które będziemy reprezentować w skali szarości. Językiem, użytym do implementacji i analizy algorytmów, jest język programowania Python. Na końcu pracy został dołączony kod z własną implementacją większości omówionych technik.

Wszystkie omawiane operacje podzielimy na trzy kategorie: operacje punktowe, przekształcenia geometryczne, filtry. W pierwszej znajdują się przekształcenia, które możemy uzyskać dzięki nałożeniu funkcji bezpośrednio na wartość piksela. W drugiej znajdują się te, które działają na indeksach pikseli, czyli ich pozycjach w maticzce. W ostatniej kategorii zaś znajdują się operacje, które działają na wartościach piksela i jego sąsiadach (otoczeniu).

2 Operacje punktowe

Jest to najbardziej podstawowa z możliwych operacji na obrazach. Polega na nałożeniu funkcji jednej zmiennej na wartości pikseli z obrazu oryginalnego, aby uzyskać nową wartość piksela w obrazie wynikowym. Możemy to zapisać w następującej postaci:

$$g(x, y) = h(f(x, y)), \quad (1)$$

gdzie $g(x, y)$ to obraz wynikowy, zaś $f(x, y)$ to obraz oryginalny. $h(z)$ to funkcja jednej zmiennej jaką nakładamy na przeskalowaną wartość piksela. Przypomnijmy, że wartości f i g to liczby całkowite z przedziału $[0, 255]$. Z reguły, przed wykonaniem przekształceń, wartości te są przeskalowane do przedziału $[0, 1]$. W powyższym równaniu przyjmujemy, że wartości te zostały przeskalowane.

Wprowadzimy pewne ograniczenia na funkcję h . Chcemy, aby funkcja h spełniała niżej wymienione warunki:

1. $h : [0, 1] \rightarrow [0, 1]$,
2. $h(0) = 0, h(1) = 1$,
3. h jest bijekcją.

Wszystkie wyżej wymienione warunki będą przez nas wymagane. Warto zwrócić uwagę na fakt, że w ogólnym przypadku istnieją sytuacje, kiedy stosuje się funkcje, które nie spełniają tych warunków z różnych przyczyn. Wszystkie postawione przez nas kryteria mają swoje uzasadnienie. Pierwsze dwa warunki zapewniają zachowanie przedziału wartości na jakim pracujemy, co jest dla nas bardzo istotne. Zaś warunek trzeci zapewnia odwracalność naszego przekształcenia.

Po nałożeniu funkcji na przeskalowane wartości pikseli chcemy, by obraz wynikowy ponownie miał wartości całkowite z przedziału $[0, 255]$. Zatem ponownie musimy dokonać skalowania jak i następnie zaokrąglenia wartości pikseli do najbliższej liczby całkowitej. Są to czynności niezbędne podczas pracy z dyskretnymi wartościami, dlatego z reguły opisując wykonane kroki nie wspomina się o tym, choć są one w nich zawarte.

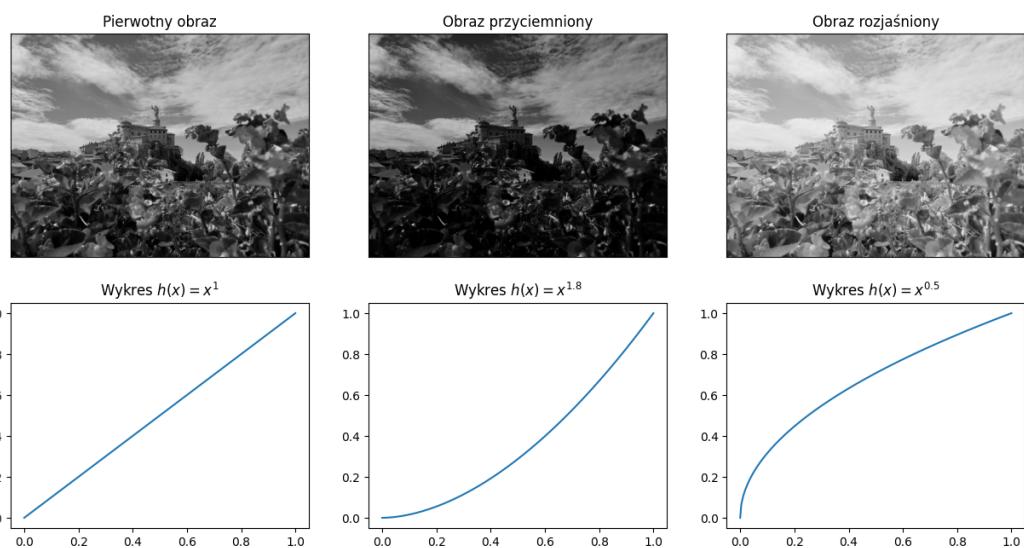
2.1 Zmiana jasności

Pierwszą grupą funkcji, którą będziemy rozważać, będą funkcje potęgowe. Będziemy używać funkcji postaci:

$$h(z) = z^\gamma, \gamma > 0. \quad (2)$$

Funkcje takiej postaci są mocno związane z przetwarzaniem obrazu. Używane są do tak zwanej korekcji gamma, która jest podstawową operacją poprawiającą parametry obrazu. Oko ludzkie jest bardziej wrażliwe na zmianę tonów w ciemnych odcieniach niż w jasnych, dlatego korekcja gamma pozwala na nieliniowy sposób korekty jasności. Dodatkowo urządzenia, na przykład monitory, zniekształcają sygnał (z reguły zgodnie z działaniem funkcji potęgowej z^γ) i aby zniwelować te zniekształcenia, również stosujemy korektę gamma (każde urządzenie ma swój własny współczynnik gamma, dlatego stosujemy funkcję potęgową z odpowiednim wykładnikiem).

Oglądając zdjęcia bez zastosowania korekty gamma, możemy zauważyć, że są zazwyczaj zbyt ciemne, co prowadzi nas do wniosku, że dzięki zastosowaniu funkcji potęgowej możemy manipulować jasnością obrazu. Poniżej został pokazany wynik zastosowania funkcji potęgowych o różnych wartościach wykładnika na obrazie oraz jej wykresy.



Rysunek 1: Wynik zastosowania funkcji potęgowych na obrazie z współczynnikami γ równymi odpowiednio (od lewej) 1, 1.8 i 0.5, w pierwszym wierszu mamy otrzymane po transformacji obrazy, w drugim wykresy funkcji h .

Rezultaty naszej implementacji potwierdzają, że przy użyciu funkcji $h(z) = z^\gamma$ możemy manipulować jasnością obrazu, co mogliśmy już przewidzieć również z kształtu naszych krzywych. Wytlumaczmy teraz dokładnie, jak to interpretować.

Dla współczynnika $\gamma > 1$, wykres naszej funkcji znajduje się pod wykresem funkcji $y = x$. Dla każdego x mamy $h(x) \leq x$, czyli na naszym przekształconym obrazie zmniejszamy wartość jasności, co naturalnie powoduje przyciemnienie obrazu. Jedynie czysty biały i czysty czarny pozostaną bez zmian. Odwrotną sytuację mamy w przypadku, gdy $0 < \gamma < 1$, wtedy wykres funkcji h znajduje się nad krzywą $y = x$, co powoduje, że w przekształconym obrazie do pikseli dodajemy pewną wartość jasności, co w efekcie daje nam rozjaśnienie obrazu.

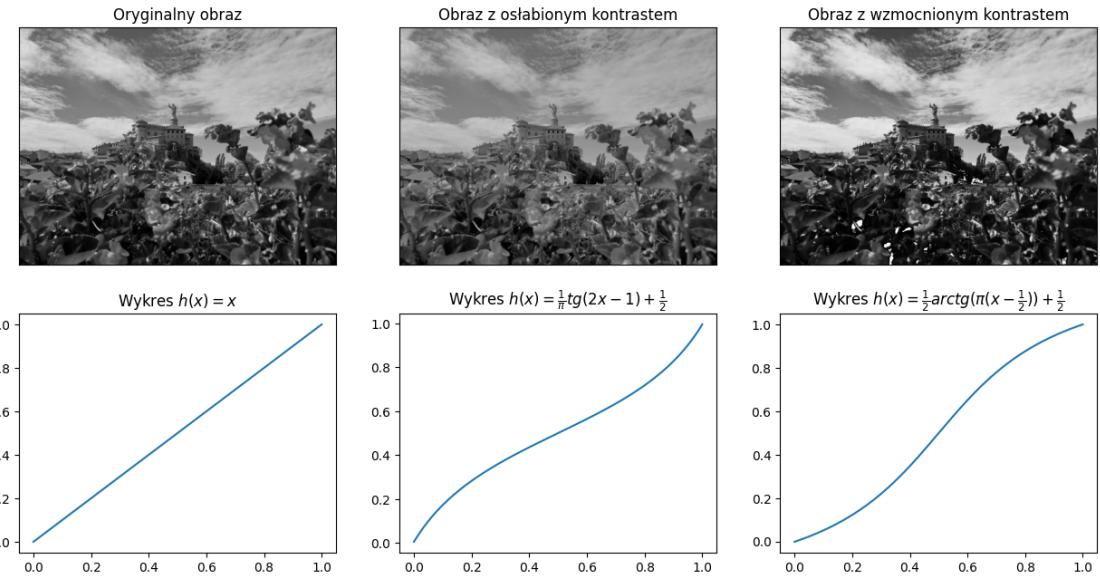
2.2 Manipulacja kontrastem

Kolejną funkcją, którą będziemy rozważać będzie funkcja manipulująca kontrastem. Chcemy, aby funkcja ta „odejmowała” pewną wartość jasności dla ciemnych pikseli i „dodawała” dla jasnych. Umownie ustalmy, że mamy „po równo” ciemnych i jasnych pikseli, czyli wartością graniczną jest $\frac{1}{2}$, która tak samo jak 0 nie jest ani liczbą dodatnią, ani ujemną, tak $\frac{1}{2}$ nie należy do jasnych ani do ciemnych pikseli.

Funkcje, które będziemy stosowali w tym podrozdziale, będą zachowywały dodatkowo punkt $(\frac{1}{2}, \frac{1}{2})$ (to znaczy $h(\frac{1}{2}) = \frac{1}{2}$) oprócz $(0, 0)$ i $(1, 1)$. Chcemy, aby na przedziale $(0, \frac{1}{2})$ wykres funkcji h znajdował się pod prostą o równaniu $y = x$, co spowoduje zmniejszenie jasności w pikselach skategoryzowanych jako ciemne. Zaś na przedziale $(\frac{1}{2}, 1)$ chcemy, by wykres funkcji h leżał nad prostą $y = x$, co wpłynie na zwiększenie jasności w pikselach skategoryzowanych jako jasne. Tak jak wcześniej zostało wspomniane funkcja h będzie zachowywała punkty $(0, 0)$, $(\frac{1}{2}, \frac{1}{2})$ oraz $(1, 1)$.

Aby uzyskać efekt zmniejszenia kontrastu, musimy działać dokładnie odwrotnie w stosunku do konstrukcji poprzedniej krzywej. Zauważmy, że jeśli mamy funkcję zwiększającą kontrast oraz istnieje do niej funkcja odwrotna (skoro w naszym przypadku funkcja jest bijekcją, to istnieje do niej funkcja odwrotna), to ta funkcja odwrotna zmniejsza kontrast. Przykładami funkcji spełniających takie własności są $\frac{1}{2}\arctg(\pi(x - \frac{1}{2})) + \frac{1}{2}$ dla wzmacniania kontrastu oraz $\frac{1}{\pi}\tg(2x - 1) + \frac{1}{2}$ dla osłabienia kontrastu. Poniżej został przedstawiony efekt przekształcenia obrazu przez podane

poprzednio przykładowe funkcje manipulujące kontrastem.



Rysunek 2: Wynik zastosowania podanych przykładowych funkcji manipulujących kontrastem na obrazie. W pierwszym wierszu mamy otrzymane po transformacji obrazy, w drugim wykresy funkcji h .

Mamy już przykład funkcji spełniających nasze założenia, jednakże jak znaleźć całą grupę funkcji spełniających te założenia, ale posiadających zmienny współczynnik umożliwiający regulację głębokości kontrastu? Na początek spróbujemy znaleźć taką funkcję, której funkcję odwrotną potrafimy wyrazić jawnym wzorem oraz ma poszukiwany przez nas kształt przypominający obróconą literę S. Takie kryteria spełnia funkcja $f(x) = \arctg(x)$. Dlatego spróbujemy przekształcić ją w taki sposób, aby spełniała pozostałe kryteria, to znaczy przechodziła przez punkty $(0, 0)$, $(\frac{1}{2}, \frac{1}{2})$ oraz $(1, 1)$. Na początek przesuniemy jej środek symetrii z punktu $(0, 0)$ do punktu $(\frac{1}{2}, \frac{1}{2})$. W ten sposób będziemy mieli zachowany punkt $(\frac{1}{2}, \frac{1}{2})$. Zatem teraz nasza funkcja ma postać $f_1(x) = \arctg(x - \frac{1}{2}) + \frac{1}{2}$. Następnie poprzez skalowanie osi OX zapewnmy przechodzenie przez punkt $(0, 0)$, stąd mamy:

$$\begin{aligned} f_2(x) &= \arctg(a(x - \frac{1}{2})) + \frac{1}{2} \\ 0 &= \arctg(-\frac{a}{2}) + \frac{1}{2} \Rightarrow 0 = -\arctg(\frac{a}{2}) + \frac{1}{2} \Rightarrow \arctg(\frac{a}{2}) = \frac{1}{2} \Rightarrow \tg(\frac{1}{2}) = \frac{a}{2} \\ &\Rightarrow a = 2\tg(\frac{1}{2}) \end{aligned}$$

Z symetrii wiemy, że mamy zapewnione także $f_2(1) = 1$. Stąd mamy już funkcję podstawową, która spełnia wszystkie wymagane założenia. Aby otrzymać cały zbiór takich funkcji, będziemy manipulować skalowaniem zarówno osi OX

jak i OY tak, aby nadal zachowany został punkt $(0, 0)$. Zatem teraz nasza funkcja przyjmie postać:

$$f_3(x) = b \cdot \operatorname{arctg} \left(c \cdot 2 \operatorname{tg} \left(\frac{1}{2} \right) \left(x - \frac{1}{2} \right) \right) + \frac{1}{2}$$

Wyrazimy parametr b za pomocą parametru c .

$$\begin{aligned} 0 &= b \cdot \operatorname{arctg} \left(c \cdot 2 \operatorname{tg} \left(\frac{1}{2} \right) \left(-\frac{1}{2} \right) \right) + \frac{1}{2} \Rightarrow 0 = b \cdot \operatorname{arctg} \left(-c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \right) + \frac{1}{2} \\ &\Rightarrow b \cdot \operatorname{arctg} \left(c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \right) = \frac{1}{2} \Rightarrow b = \frac{1}{2 \operatorname{arctg} \left(c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \right)} \end{aligned}$$

Ostatecznie nasza funkcja ma postać:

$$f_3(x) = \frac{1}{2 \operatorname{arctg} \left(c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \right)} \operatorname{arctg} \left(2c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \left(x - \frac{1}{2} \right) \right) + \frac{1}{2}.$$

Zauważmy, że nadal nasza funkcja jest bijekcją. Nasze operacje zachowywały tę własność funkcji, dzięki czemu możemy w jawnym sposobie napisać wzór na funkcję odwrotną:

$$f_3^{-1}(x) = \frac{1}{2c \cdot \operatorname{tg} \left(\frac{1}{2} \right)} \operatorname{tg} \left(2 \operatorname{arctg} \left(c \cdot \operatorname{tg} \left(\frac{1}{2} \right) \right) \left(x - \frac{1}{2} \right) \right) + \frac{1}{2}.$$

Powyższe obliczenia i metody posłużą nam w późniejszej części pracy do implementacji programu, w którym będziemy mogli wykonywać wszystkie omówione podstawowe operacje na obrazie.

Cały powyższy rozdział powstał w oparciu o [1, podrozdział 11.2: Image Display], [3, podrozdział 3.1: Point operators], [2, podrozdział 3.2: Some Basic Intensity Transformation Functions].

3 Przekształcenia geometryczne

W tym rozdziale zajmiemy się omówieniem niektórych operacji wykonywanych na adresach pikseli, w szczególności omówimy operacje obrotu i skalowania. Rozdział ten powstał w oparciu o [3, podrozdział 3.6: Geometric transformations], [2, podrozdział 2.4: Image Sampling and Quantization] oraz [4].

3.1 Obrót

Obrót o kąt θ wokół punktu P , oznaczamy O_P^θ , to przekształcenie izometryczne płaszczyzny, zachowujące kąty między prostymi oraz odległości między odpowiednimi punktami. W naszych rozważaniach będziemy omawiać tylko liniowe przekształcenia, stąd $P = (0, 0)$. Operację obrotu punktu $X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2$ o kąt θ (wokół punktu $P = (0, 0)$) możemy wyrazić za pomocą wzoru:

$$X' = \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad (3)$$

Zauważmy, że przy takiej postaci macierzy obrotu, nasz obrót będzie wykonywany przeciwnie do ruchu wskazówek zegara (przy tradycyjnie zorientowanych osiach i $\theta > 0$). Aby otrzymać obrót zgodny z ruchem wskazówek zegara, macierz obrotu powinna mieć postać:

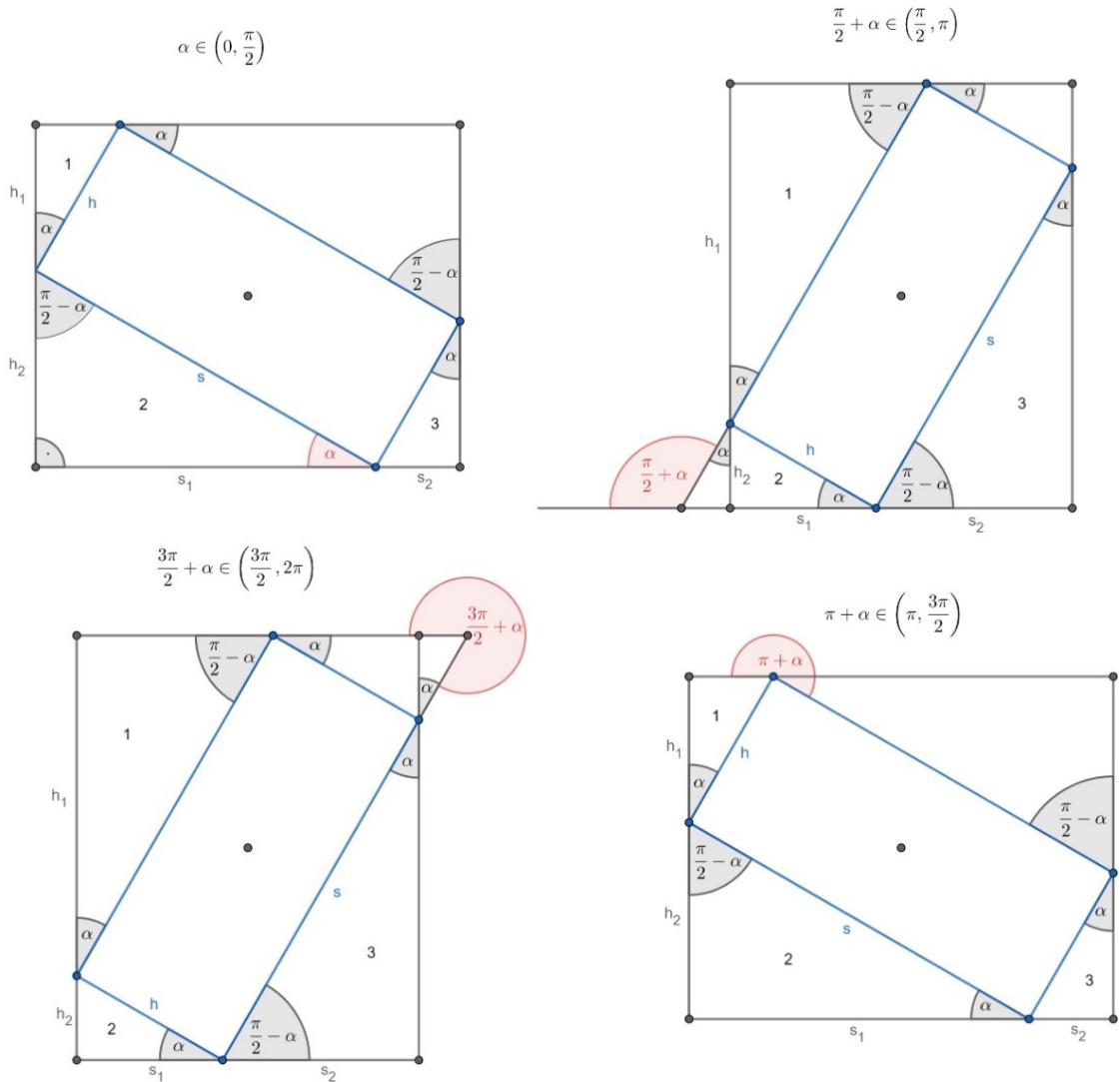
$$\begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix}.$$

Warto zwrócić uwagę na fakt, iż przez reprezentację obrazu za pomocą macierzy, nie mamy tradycyjnie zorientowanych osi układu współrzędnych. Nasza oś OY jest skierowana w dół a nie w górę, co powoduje, że używając wyżej podanych macierzy kierunek obrotu jest przeciwny do wcześniej podanego. Stąd pomimo użycia macierzy z równania (3) nasz obrót będzie wykonany zgodnie z ruchem wskazówek zegara. Nałożenie obrotu na obraz możemy zapisać następująco:

$$g(x, y) = M_O \cdot f(x, y) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$

gdzie $g(x, y)$ oznacza obraz obrócony, M_O macierz obrotu, zaś $f(x, y)$ obraz oryginalny. Pamiętajmy, że przed zastosowaniem wzoru będziemy musieli przesunąć obraz, tak by środek obrazu znalazł się w środku układu współrzędnych, następnie po obrocie ponownie musimy przesunąć obraz o odpowiedni wektor, tak by jego środek znalazł się w odpowiednim miejscu.

Wiemy już, co należy wykonać z adresem każdego piksela. Następnie jego wartość umieszczamy pod nowym adresem. Ale zanim przejdziemy do implementacji potrzebujemy jeszcze znać wymiary obrazu obróconego, dlatego teraz przejdziemy do ich wyznaczenia.



Rysunek 3: Schematyczne ilustracje do wyliczenia wymiarów obrazu po obrocie w zależności od kąta obrotu.

Na rysunku 3 przedstawiono schematy dla 4 przypadków, jak wyznaczyć

szerokość i wysokość obrazu obróconego, w zależności w jakim przedziale znajduje się wartość kąta θ (na razie pomijamy kąty $\frac{1}{2}\pi, \pi, \frac{3}{2}\pi, 2\pi$ gdyż dla nich wyznaczenie nowej wysokości i szerokości jest oczywiste). Korzystamy z oznaczeń z rysunku 3, θ oznacza kąt obrotu, a kąt $\alpha \in (0, \frac{\pi}{2})$.

$$1. \quad \theta \in (0, \frac{\pi}{2})$$

Niech $\theta = \alpha$. Na podstawie trójkątów (oznaczonych numerami 1,2 i 3 na odpowiednim schemacie na rysunku 3) możemy zapisać:

$$\cos(\alpha) = \frac{h_1}{h} \Rightarrow h_1 = h \cdot \cos(\alpha)$$

$$\sin(\alpha) = \frac{h_2}{s} \Rightarrow h_2 = s \cdot \sin(\alpha)$$

$$\cos(\alpha) = \frac{s_1}{s} \Rightarrow s_1 = s \cdot \cos(\alpha)$$

$$\sin(\alpha) = \frac{s_2}{h} \Rightarrow s_2 = h \cdot \sin(\alpha)$$

Skąd otrzymujemy:

$$h_1 + h_2 = h \cdot \cos(\alpha) + s \cdot \sin(\alpha) = h \cdot \cos(\theta) + s \cdot \sin(\theta)$$

$$s_1 + s_2 = s \cdot \cos(\alpha) + h \cdot \sin(\alpha) = s \cdot \cos(\theta) + h \cdot \sin(\theta)$$

$$2. \quad \theta \in (\frac{\pi}{2}, \pi)$$

Niech $\theta = \frac{\pi}{2} + \alpha$, gdzie $\alpha \in (0, \frac{\pi}{2})$. Na podstawie trójkątów (oznaczonych numerami 1,2 i 3 na odpowiednim schemacie na rysunku 3) możemy zapisać:

$$\cos(\alpha) = \frac{h_1}{s} \Rightarrow h_1 = s \cdot \cos(\alpha)$$

$$\sin(\alpha) = \frac{h_2}{h} \Rightarrow h_2 = h \cdot \sin(\alpha)$$

$$\cos(\alpha) = \frac{s_1}{h} \Rightarrow s_1 = h \cdot \cos(\alpha)$$

$$\sin(\alpha) = \frac{s_2}{s} \Rightarrow s_2 = s \cdot \sin(\alpha)$$

Dodatkowo wiemy:

$$\sin(\frac{\pi}{2} + \alpha) = \cos(\alpha) \text{ oraz } \cos(\frac{\pi}{2} + \alpha) = -\sin(\alpha)$$

Skąd otrzymujemy:

$$h_1 + h_2 = s \cdot \cos(\alpha) + h \cdot \sin(\alpha) = s \cdot \sin\left(\frac{\pi}{2} + \alpha\right) + h \cdot \left(-\cos\left(\frac{\pi}{2} + \alpha\right)\right) = \\ s \cdot \sin(\theta) + h \cdot (-\cos(\theta))$$

$$s_1 + s_2 = h \cdot \cos(\alpha) + s \cdot \sin(\alpha) = h \cdot \sin\left(\frac{\pi}{2} + \alpha\right) + s \cdot \left(-\cos\left(\frac{\pi}{2} + \alpha\right)\right) = \\ h \cdot \sin(\theta) + s \cdot (-\cos(\theta))$$

3. $\theta \in \left(\pi, \frac{3\pi}{2}\right)$

Niech $\theta = \pi + \alpha$, gdzie $\alpha \in (0, \frac{\pi}{2})$. Na podstawie trójkątów (oznaczonych numerami 1,2 i 3 na odpowiednim schemacie na rysunku 3) możemy zapisać:

$$\begin{aligned} \cos(\alpha) &= \frac{h_1}{h} \Rightarrow h_1 = h \cdot \cos(\alpha) \\ \sin(\alpha) &= \frac{h_2}{s} \Rightarrow h_2 = s \cdot \sin(\alpha) \\ \cos(\alpha) &= \frac{s_1}{s} \Rightarrow s_1 = s \cdot \cos(\alpha) \\ \sin(\alpha) &= \frac{s_2}{h} \Rightarrow s_2 = h \cdot \sin(\alpha) \end{aligned}$$

Dodatkowo wiemy:

$$\sin(\pi + \alpha) = -\sin(\alpha) \text{ oraz } \cos(\pi + \alpha) = -\cos(\alpha)$$

Skąd otrzymujemy:

$$\begin{aligned} h_1 + h_2 &= h \cdot \cos(\alpha) + s \cdot \sin(\alpha) = h \cdot (-\cos(\pi + \alpha)) + s \cdot (-\sin(\pi + \alpha)) = \\ &\quad h \cdot (-\cos(\theta)) + s \cdot (-\sin(\theta)) \\ s_1 + s_2 &= s \cdot \cos(\alpha) + h \cdot \sin(\alpha) = s \cdot (-\cos(\pi + \alpha)) + h \cdot (-\sin(\pi + \alpha)) = \\ &\quad s \cdot (-\cos(\theta)) + h \cdot (-\sin(\theta)) \end{aligned}$$

4. $\theta \in \left(\frac{3\pi}{2}, 2\pi\right)$

Niech $\theta = \frac{3\pi}{2} + \alpha$, gdzie $\alpha \in (0, \frac{\pi}{2})$. Na podstawie trójkątów (oznaczonych numerami 1,2 i 3 na odpowiednim schemacie na rysunku 3) możemy zapisać:

$$\begin{aligned} \cos(\alpha) &= \frac{h_1}{s} \Rightarrow h_1 = s \cdot \cos(\alpha) \\ \sin(\alpha) &= \frac{h_2}{h} \Rightarrow h_2 = h \cdot \sin(\alpha) \\ \cos(\alpha) &= \frac{s_1}{h} \Rightarrow s_1 = h \cdot \cos(\alpha) \\ \sin(\alpha) &= \frac{s_2}{s} \Rightarrow s_2 = s \cdot \sin(\alpha) \end{aligned}$$

Dodatkowo wiemy:

$$\sin\left(\frac{3\pi}{2} + \alpha\right) = -\cos(\alpha) \text{ oraz } \cos\left(\frac{3\pi}{2} + \alpha\right) = \sin(\alpha)$$

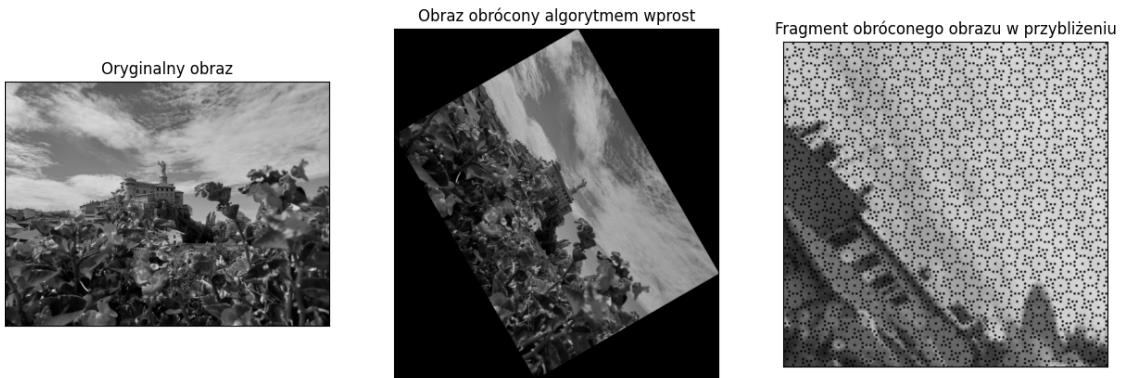
Skąd otrzymujemy:

$$h_1 + h_2 = s \cdot \cos(\alpha) + h \cdot \sin(\alpha) = s \cdot (-\sin(\frac{3\pi}{2} + \alpha)) + h \cdot \cos(\frac{3\pi}{2} + \alpha) = \\ s \cdot (-\sin(\theta)) + h \cdot \cos(\theta)$$

$$s_1 + s_2 = h \cdot \cos(\alpha) + s \cdot \sin(\alpha) = h \cdot (-\sin(\frac{3\pi}{2} + \alpha)) + s \cdot \cos(\frac{3\pi}{2} + \alpha) = \\ h \cdot (-\sin(\theta)) + s \cdot \cos(\theta)$$

Dodatkowo zauważamy, że dla $\theta \in \{0, 2\pi\}$ działa wzór uzyskany dla przedziału $(0, \frac{\pi}{2})$, następnie odpowiednio dla kątów $\frac{\pi}{2}, \pi, \frac{3\pi}{2}$ działają wzory uzyskane dla przedziałów $(\frac{\pi}{2}, \pi), (\pi, \frac{3\pi}{2}), (\frac{3\pi}{2}, 2\pi)$. Zatem możemy zapisać ogólny wzór wyrażający wymiary obrazu wyjściowego:

$$h' = h \cdot |\cos(\theta)| + s \cdot |\sin(\theta)| \\ s' = s \cdot |\cos(\theta)| + h \cdot |\sin(\theta)|$$



Rysunek 4: Efekt implementacji obrotu algorytmem wprost.

Mamy już wszystkie potrzebne informacje do implementacji. Zatem spójrzmy na efekt jej zastosowania na obrazie pokazany na rysunku 4. Jak możemy zauważyc nasze rozwiązanie ma pewne wady. Przez dość spory rozmiar zdjęcia, to jest 1997×1498 pikseli, na nim samym nie dostrzegamy tak wyraźnie pojawiającego się problemu, ale kiedy spojrzymy obok na ilustrację z przybliżonym fragmentem zdjęcia już wyraźnie widzimy pojawiający się problem. Na obróconym obrazie, powstają czarne piksele, które reprezentują „dziury”. Pojawiają się one dlatego, iż nasza pierwotna siatka kwadratów (ułożonych wzdłuż linii równoległych do krawędzi obrazu) po obróceniu nie daje nam już linii równoległych i prostopadłych do odpowiednich krawędzi obrazu (wyjątkiem są kąty $\frac{1}{2}\pi, \pi, \frac{3}{2}\pi, 2\pi$). Z tej przyczyny

mamy problem z przyporządkowaniem części pikseli z pierwotnego zdjęcia do nowego, gdyż po zaokrągleniu adresów, część pikseli się pokrywa, co powoduje brak pikseli w innych miejscach. Z tej przyczyny tradycyjne podejście z wykorzystaniem wzoru wprost zawodzi. Jednak tę kwestię możemy rozwiązać podchodząc do problemu od końca, to znaczy od obrazu obróconego.

Nasz nowy algorytm będzie polegał na szukaniu dla każdego piksela z obrazu obróconego piksela od którego pochodzi. Innymi słowy na adres nowego piksela nakładamy przekształcenie odwrotne, dzięki czemu uzyskujemy adres piksela w pierwotnym obrazie. Oczywiście tutaj także nie otrzymujemy zazwyczaj całkowitych wartości dla adresu, co powoduje, że używamy metody interpolacji z obrazu pierwotnego, którego piksele traktujemy jako próbki. Jest wiele metod interpolacji, jedne są dokładniejsze inne mniej, wśród tych metod znajdziemy metodę najbliższego sąsiada, wykorzystującego piksel (punkt), którego odległość od otrzymanego punktu (na podstawie obrazu wynikowego) jest najmniejsza. Powszechnie stosuje się także interpolacje liniową, dwuliniową, sześcienną (cubic) czy dwusześcienną (bicubic). Jako że nie potrzebujemy bardzo wysokiej jakości naszych obrazów, by pokazać zachodzące zjawiska, w implementacjach została zastosowana metoda najbliższego sąsiada (zaokrąglamy wyliczony adres do najbliższych wartości całkowitych). Aby wykonać naszą implementację potrzebujemy jeszcze wyznaczyć przekształcenie odwrotne.

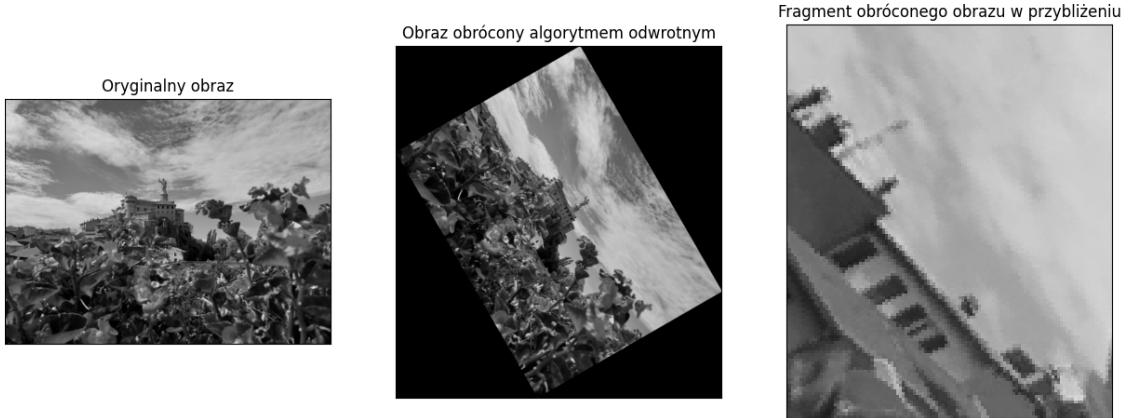
$$\begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}^{-1} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}^{-1} \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Mamy już wszystkie elementy potrzebne do implementacji algorytmu. Pełny kod implementacji jest zawarty jako część listingu 1. Na rysunku 5 przedstawiono efekt działania naszego programu na obrazie (dokładnie tym samym, co poprzednio dla lepszego porównania). Jak możemy zauważyć, w tym rozwiążaniu

nie widać żadnych brakujących pikseli, co pokazuje, że to podejście jest dużo skuteczniejsze.



Rysunek 5: Efekt implementacji obrotu algorytmem odwrotnym.

Istnieje jeszcze jeden ciekawy sposób implementacji obrotu. Ten sposób wykorzystuje przekształcenie afinczne zwane ścięciem (shear), które możemy wyrazić wzorem:

$$Q = \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow Q' = \begin{pmatrix} x + ky \\ y \end{pmatrix}, \text{ gdzie } k \neq 0.$$

Przy tej postaci wzoru przekształcamy oszę OX , zachowując oszę OY , ale równie dobrze możemy przekształcić oszę OY , a zachować niezmienioną oszę OX . Ważnym aspektem tego przekształcenia jest fakt, że przekształcamy w zakresie tylko jednej osi, zachowując drugą w niezmienionym stanie. Macierze tego przekształcenia mają postać:

$$A = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \text{ lub } B = \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix}, \text{ gdzie } a \in \mathbb{R}, b \in \mathbb{R}.$$

Dla ułatwienia rozróżnienia, czy chodzi nam o macierz ścięcia górnego- czy dolnotrójkatną, będziemy nazywać je macierzami typu A lub B . Wykorzystując dokładnie 3 macierze takiej postaci możemy rozłożyć nasz obrót na 3 ścięcia. Dzięki takiemu rozwiązaniu unikniemy problemów ze znieksztalcaniem siatki, ponieważ działając tylko wzdłuż jednej osi, jedynie przesuwamy piksele, co daje nam jednoznaczność przekształcenia, gdzie każdy piksel przejdzie na inny, dzięki czemu unikamy „dziur” i interpolacji (choć skoro zaokrąglamy adresy pikseli, to w pewnym sensie dokonujemy interpolacji, ale w najprostszy możliwy sposób). Poniżej rozważymy, dlaczego

2 macierze nie wystarczą, a dlaczego 3 już tak, i jakiej postaci muszą one być. Na początek wyznaczymy iloczyn dwóch takich macierzy. Rozważymy iloczyn dwóch macierzy tej samej postaci (użyjemy macierzy typu A) i dwóch macierzy różnych typów.

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & a+c \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix} = \begin{pmatrix} 1+ab & a \\ b & 1 \end{pmatrix}$$

Jak możemy zauważyc, iloczyn macierzy tego samego typu daje macierz także tego samego typu. Zatem wykonując kilkukrotnie taką operację, nie przybliża nas to do otrzymania macierzy, w której wszystkie elementy zależą od nieznanych parametrów. Z drugiej strony mnożenie macierzy różnego typu przybliża nas do otrzymania macierzy szukanej postaci. Jednak jak możemy zauważyc, nadal jeden element macierzy jest równy 1, zatem dwie macierze nie wystarczą, by rozłożyć macierz obrotu dla dowolnego kąta. Jednakże iloczyn 3 macierzy, które mają różne typy z sąsiadującymi, daje nam szukaną postać macierzy:

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1+ab & a+c+abc \\ b & bc+1 \end{pmatrix}$$

Jeśli przyrównamy tę macierz do macierzy obrotu i będzie istniało rozwiązanie, oznaczać to będzie, że 3 macierze wystarczą. Zatem przypuszczamy, że szukany rozkład będzie miał postać:

$$M_O = A_1 B_1 A_2 \text{ lub } M_O = B_1 A_1 B_2.$$

Sprawdzmy teraz, czy istnieje rozwiązanie tej równości. Najpierw rozważymy rozkład typu ABA .

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1+ab & a+c+abc \\ b & bc+1 \end{pmatrix}$$

$$\begin{pmatrix} 1+ab & a+c+abc \\ b & bc+1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Zatem mamy układ 4 równań z 3 niewiadomymi, $\theta \in [0, 2\pi]$:

$$\begin{cases} 1 + ab = \cos(\theta) \\ a + c + abc = -\sin(\theta) \\ b = \sin(\theta) \\ bc + 1 = \cos(\theta) \end{cases} = \begin{cases} b = \sin(\theta) \\ a = \frac{\cos(\theta)-1}{\sin(\theta)}, \sin(\theta) \neq 0 \\ c = \frac{\cos(\theta)-1}{\sin(\theta)}, \sin(\theta) \neq 0 \\ a + c + abc = -\sin(\theta) \end{cases} = \begin{cases} b = \sin(\theta) \\ a = \frac{\cos(\theta)-1}{\sin(\theta)}, \sin(\theta) \neq 0 \\ c = \frac{\cos(\theta)-1}{\sin(\theta)}, \sin(\theta) \neq 0 \end{cases}$$

Niestety przy takiej formie a i c możemy mieć problem dla kątów o wartościach bliskich 0, dlatego przekształcimy trochę to wyrażenie:

$$a = c = \frac{\cos(\theta) - 1}{\sin(\theta)} = \frac{(1 - 2\sin^2(\frac{\theta}{2})) - 1}{2\sin(\frac{\theta}{2})\cos(\frac{\theta}{2})} = -\frac{\sin(\frac{\theta}{2})}{\cos(\frac{\theta}{2})} = -\tan\left(\frac{\theta}{2}\right).$$

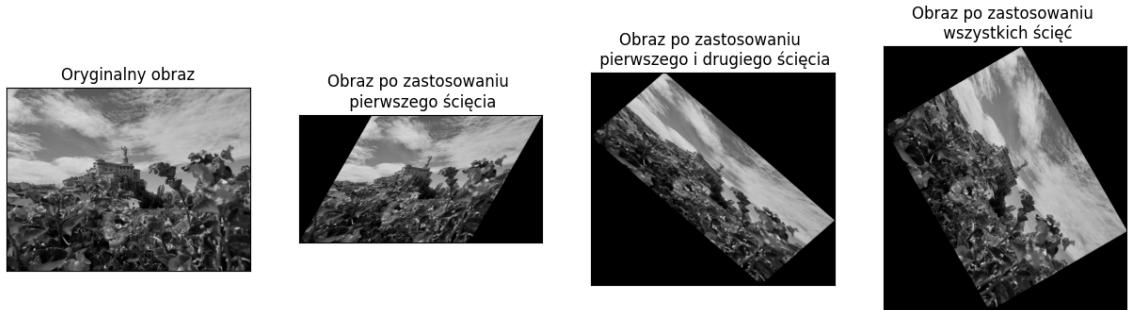
Znaleźliśmy nasze rozwiązanie. Wzór na rozkład macierzy obrotu przyjmuje postać:

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin(\theta) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{pmatrix}.$$

Analogicznie postępując dla rozkładu BAB otrzymujemy:

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \tan\left(\frac{\theta}{2}\right) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\sin(\theta) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \tan\left(\frac{\theta}{2}\right) & 1 \end{pmatrix}.$$

Teraz możemy zaimplementować obrót, używając któregokolwiek z powyżej wyznaczonych rozkładów. W zastosowanym algorytmie użyto rozkładu ABA . Efekt zastosowania obrotu tą metodą został pokazany na rysunku 6.



Rysunek 6: Efekt implementacji obrotu za pomocą 3 ścieć.

3.2 Skalowanie

Inną z podstawowych operacji wykonywanych przez nas na obrazach jest skalowanie. Niekiedy chcemy zwiększyć rozmiar obrazu, a niekiedy go zmniejszyć. Trzeba pamiętać, że pomimo zwiększenia rozmiaru obrazu, tak naprawdę nie użyujemy żadnych nowych informacji niż posiadaliśmy z pierwotnego obrazu. Przy zmniejszaniu rozmiaru zdjęcia jest trochę inaczej, bo wręcz tracimy część informacji. Na pierwszy rzut oka algorytm zmiany skali wydaje się być dość prosty, ale posiada jeden ważny krok, który w zależności od potrzeb użytkownika może być prosty lub bardziej złożony. Mowa tu o interpolacji wartości pikseli. Podobnie jak przy obrocie i tutaj zmieniając ilość pikseli, musimy interpolować i próbować. Jak skomplikowane to będzie zależy tylko od nas. Jako że w tej pracy skalowanie zostanie użyte tylko na potrzeby przyspieszenia działania interaktywnego okna (które opiszymy wkrótce), zostanie tutaj użyty najprostszy sposób interpolacji, czyli metoda najbliższego sąsiada, ale w ogólnym przypadku ta metoda interpolacji daje niezadowalające rezultaty. Nasz algorytm skalowania będzie przebiegał w następujących krokach:

1. Stworzenie nowej macierzy o wymiarach przeskalowanych zgodnie z parametrem skali k .
2. Dla każdego elementu nowej macierzy interpolujemy wartość piksela z oryginalnego obrazu.

Operację tę będziemy stosować przy generacji interaktywnego okna, więc pominimy w tym miejscu praktyczne przykłady.

4 Filtry, czyli sploty

Niewątpliwie zdarzyło się nam usłyszeć o określeniu nałożenia filtra w kontekście zdjęć. W tym rozdziale omówimy filtry związane z rozmyciem (między innymi filtr Gaussa) i wyostrzaniem obrazu (między innymi filtr Laplace'a). Z matematycznego punktu widzenia operację nałożenia filtra na zdjęcie opisuje operacja splotu, tradycyjnie dla jednego wymiaru definiowana jako:

$$(g * f)(x) = \int_{-\infty}^{+\infty} g(y)f(x - y)dy. \quad (4)$$

Jednakże będziemy zajmować się filtrami dwuwymiarowymi. Nasz obraz będzie macierzą rozmiaru $a \times b$, stąd będziemy trochę inaczej definiować operację splotu niż w równaniu (4), gdyż nasz przypadek jest dyskretny, a nie ciągły. W naszej sytuacji operacja splotu dla funkcji dwóch zmiennych jest zdefiniowana wzorem:

$$(g * f)(s, t) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} g(i, j)f(s - i, t - j),$$

gdzie $s \in [0, a - 1] \cap \mathbb{N}, t \in [0, b - 1] \cap \mathbb{N}$.

Widać, że w przypadku dyskretnego, skończonego obrazu pojawia się tak zwany problem brzegowy, związany z faktem, że niekoniecznie $s - i$ oraz $t - j$ wpadają do odpowiedniego zakresu. Poruszmy ten temat poniżej. W naszym przypadku będziemy rozważać tylko jądra o skończonym rozmiarze, stąd założymy że rozmiar macierzy jądra (maski) to $n \times m$. Dodatkowo zazwyczaj chcemy, by jądra spełniały warunek

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} g_{ij} = 1.$$

Oznacza on, że suma wyrazów macierzy ma być równa 1. Warunek ten będzie wymagany dla jąder, których wartości wszystkich elementów są liczbami nieujemnymi, dzięki czemu zakres wartości pikseli pozostanie niezmieniony. W naszym przypadku rozmiar jądra zawsze będzie wynosił 3×3 (ale sama funkcja wykonująca operację splotu, działa dla dowolnego rozmiaru jądra, ale ważne by jego wymiary były

liczbami nieparzystymi), zatem możemy zapisać:

$$(g * f)(s, t) = \sum_{i=-1}^1 \sum_{j=-1}^1 g_{i+1,j+1} f(s - i, t - j),$$

gdzie $s \in [0, a - 1] \cap \mathbb{N}, t \in [0, b - 1] \cap \mathbb{N}$.

Zanim przejdziemy już do konkretnych przykładów filtrów wróćmy do problemu brzegowego. A mianowicie, co robić z pikselami, którym brakuje sąsiadów z którejś strony? Rozwiązaniem jest rozszerzenie obrazka o tak zwaną ramkę. Jest kilka różnych metod wypełniania tej ramki między innymi wypełnienie czarnymi pikselami (zero) lub innym kolorem (constant), inną metodą jest tak zwane lustro (mirror), czyli odbijanie pikseli względem krawędzi obrazu. Oprócz tego wśród metod znajdziemy także ściskanie do krawędzi (clamp) czy kafelkowanie (wrap), które odpowiada określonemu rozszerzeniu obrazu na całą płaszczyznę. Więcej informacji na ten temat można znaleźć w pozycji [3, strony 101-102]. W aplikacji naszego algorytmu zostanie użyta metoda constant.

Ten rozdział powstał w oparciu o [1, rozdział 6: Operations on Images], [2, podrozdział 3.4: Fundamentals of Spatial Filtering], [3, rozdział 3: Intensity Transformations and Spatial Filtering].

4.1 Rozmycie obrazu

Jednym z filtrów mających efekt rozmycia jest filtr Gaussa, zadany wzorem pochodzący od gęstości rozkładu normalnego (dla dwóch wymiarów):

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Jest to wzór dla ciągłej dziedziny. Aby w praktyczny sposób wyznaczyć macierz jądra przekształcenia, posłużymy się filtrami dwumianowymi jednowymiarowymi. Jak wynika z centralnego twierdzenia granicznego, możemy przybliżać rozkład normalny rozkładem dwumianowym. Gęstość rozkładu dwumianowego z parametrami n i p (oznaczamy $\mathcal{B}(n, p)$) jest zadana wzorem:

$$P(X = k) = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}, \text{ gdzie } \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Biorąc $p = \frac{1}{2}$ otrzymujemy wzór przybliżający szukane jądro filtra Gaussa. Wraz ze

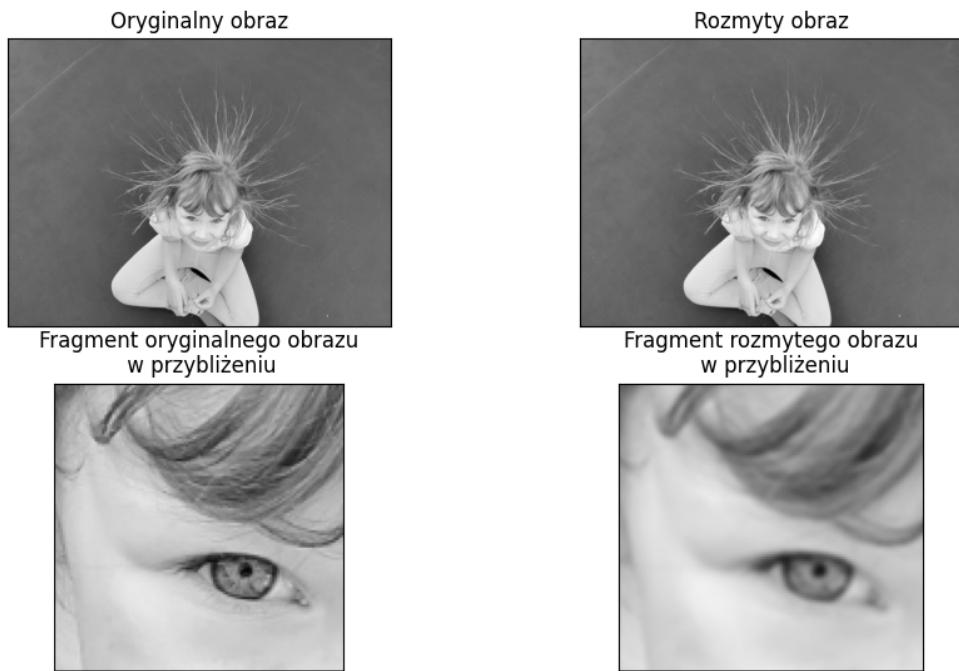
zwiększeniem się wartości n wzór ten coraz lepiej przybliża maskę dla filtra Gaussa. W naszym algorytmie chcemy użyć macierzy rozmiaru 3×3 , stąd $n = 2$. Może budzić obawy mała wartość n , ale znając fakt, że jeżeli na zdjęciu wykonamy dwa razy operację splotu z jądrem rozmiaru 3×3 , jest to równoważne z wykonaniem splotu z jądrem 5×5 , jest to wartość wystarczająca, ponieważ w literaturze (na przykład w [3, strony 102]) już dla takiego rozmiaru macierzy tej postaci używa się określenia jądra filtra Gaussa. Możemy zapisać formułę wzoru przybliżającego:

$$b(s) = \frac{2!}{s!(2-s)!} \cdot \frac{1}{2^2} \text{ dla } s = 0, 1, 2.$$

Skąd otrzymujemy macierz jądra w jednym wymiarze postaci : $\frac{1}{4} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}^T$. Aby wyznaczyć jądro dla dwóch wymiarów, postępujemy w następujący sposób:

$$\frac{1}{4} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot \frac{1}{4} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}^T = \frac{1}{4} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot \frac{1}{4} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

Mamy już macierz potrzebną do użycia w algorytmie, dzięki czemu możemy wykonać rozmywanie obrazu. Poniżej na rysunku 7 został pokazany obraz oryginalny oraz obraz po nałożeniu filtra. Poniżej zostały również zestawione przybliżone fragmenty zdjęć dla lepszej widoczności działania filtra Gaussa.



Rysunek 7: Efekt zastosowania filtra Gaussa na obrazie.

4.2 Wyostrzanie obrazu

Przy rozmywaniu nową wartość pikseli uzyskiwało się dzięki uśrednianiu wartości pikseli w sąsiedztwie. Zauważając, że uśrednianie jest analogiczne do całkowania, możemy przypuszczać, że wykonując operację odwrotną, czyli różniczkowanie, uzyskamy wyostrzanie. Na początek rozważymy definicje pierwszej i drugiej pochodnej dla funkcji jednej zmiennej.

$$\frac{df}{dx} = f(x+1) - f(x), \quad (5)$$

$$\frac{d^2f}{dx^2} = f(x+1) + f(x-1) - 2f(x). \quad (6)$$

Zauważmy, że wzory te stanowią adaptację do sytuacji dyskretnej zwykłej definicji pochodnej. Zwróćmy uwagę, że wzór na pochodną drugiego rzędu nie jest iteracją pochodnej pierwszego rzędu:

$$\frac{d}{dx}(f(x+1) - f(x)) = f(x+2) - f(x+1) - f(x+1) + f(x) = f(x+2) - 2f(x+1) + f(x),$$

ale jest jej zmodyfikowaną wersją. Tak zdefiniowane wzory pomogą nam w późniejszych rozważaniach.

Jednym z popularniejszych filtrów wyostrzających jest filtr Laplace'a. Do konstrukcji jądra tego przekształcenia używany jest laplasjan funkcji dwóch zmiennych wyrażony wzorem:

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y)$$

Aby wyrazić to równanie w dyskretnej dziedzinie, posłużymy się wcześniejszymi zdefiniowanymi wzorami na pochodne, ale pamiętając, że mamy teraz funkcję dwóch zmiennych. Otrzymujemy zatem:

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2}(x, y) &= f(x+1, y) + f(x-1, y) - 2f(x, y) \\ \text{oraz} \\ \frac{\partial^2 f}{\partial y^2}(x, y) &= f(x, y+1) + f(x, y-1) - 2f(x, y). \end{aligned}$$

Skąd dyskretna postać laplasjanu jest wyrażona wzorem:

$$\nabla^2 f(x, y) = f(x + 1, y) + f(x - 1, y) + f(x, y + 1) + f(x, y - 1) - 4f(x, y).$$

Wykorzystując ten wzór, możemy uzyskać macierz jądra przekształcenia, niezbędną do użycia w naszym algorytmie. Jest ona postaci:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}. \quad (7)$$

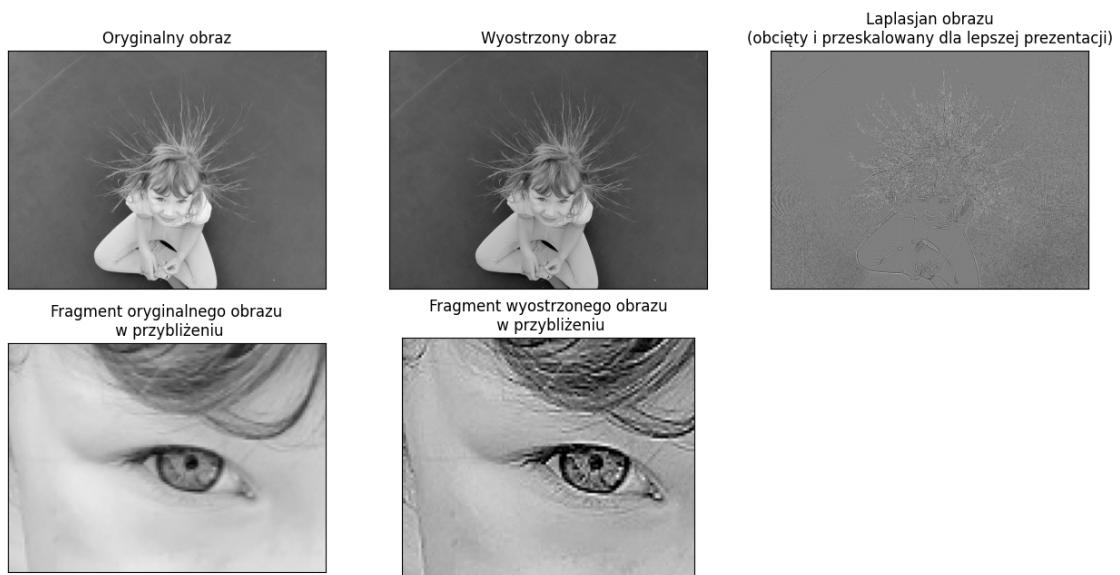
Nakładając filtr o takim jądrze, otrzymamy maskę, w której zostaną podkreślone wszelkie nagłe zmiany intensywności często krawędzie, ale także szum. Niestety przez to, że macierz ma zarówno wyrazy dodatnie jak i ujemne, nie jest ona w żaden sposób normalizowana, co wpływa później na fakt, że otrzymany przez nas laplasjan obrazu ma inny zakres wartości niż obraz oryginalny (w szczególności piksele mogą mieć wartości ujemne). Rozważenie tego jest ważne w kontekście operacji, którą następnie wykonamy, by uzyskać wyostrzony obraz. Kolejny krok algorytmu opisuje wzór:

$$g(x, y) = f(x, y) + c[\nabla^2 f(x, y)],$$

gdzie $g(x, y)$ to wynikowy obraz z wyostrzonymi krawędziami, $f(x, y)$ to obraz oryginalny, $c = -1$ dla macierzy zadanej równaniem (7), zaś $\nabla^2 f(x, y)$ to oczywiście wcześniej omówiony laplasjan obrazu oryginalnego.

Wróćmy do rozważenia zakresu wartości laplasjanu, a następnie jego sumy z obrazem oryginalnym. Jako że trudno jest przewidzieć zakres $\nabla^2 f$ dla dowolnego zdjęcia, najlepiej przed sumą z oryginalnym zdjęciem przeskalać jego zakres wartości na przedział $[-255, 255] \cap \mathbb{N}$. W ten sposób różnica z laplasjanem może zmienić wartość każdego piksela w oryginalnym obrazie na dowolną inną wartość. Niestety później także musimy skorygować zakres wartości najlepiej przez obcięcie, wszystkie wartości ujemne na 0, a te przekraczające wartość 255 na 255. W ten sposób nasz obraz wynikowy nadal będzie miał odpowiedni zakres wartości pikseli. Poniżej na rysunku 8 został pokazany laplasjan obrazu, jak i obraz z wyostrzonymi krawędziami uzyskany z jego użyciem. Dla lepszego porównania zostały także

zestawione w przybliżeniu detale obrazu wyostrzonego z obrazem oryginalnym.



Rysunek 8: Efekt zastosowania filtra Laplace'a na obrazie oraz wyostrzenie obrazu z jego użyciem.

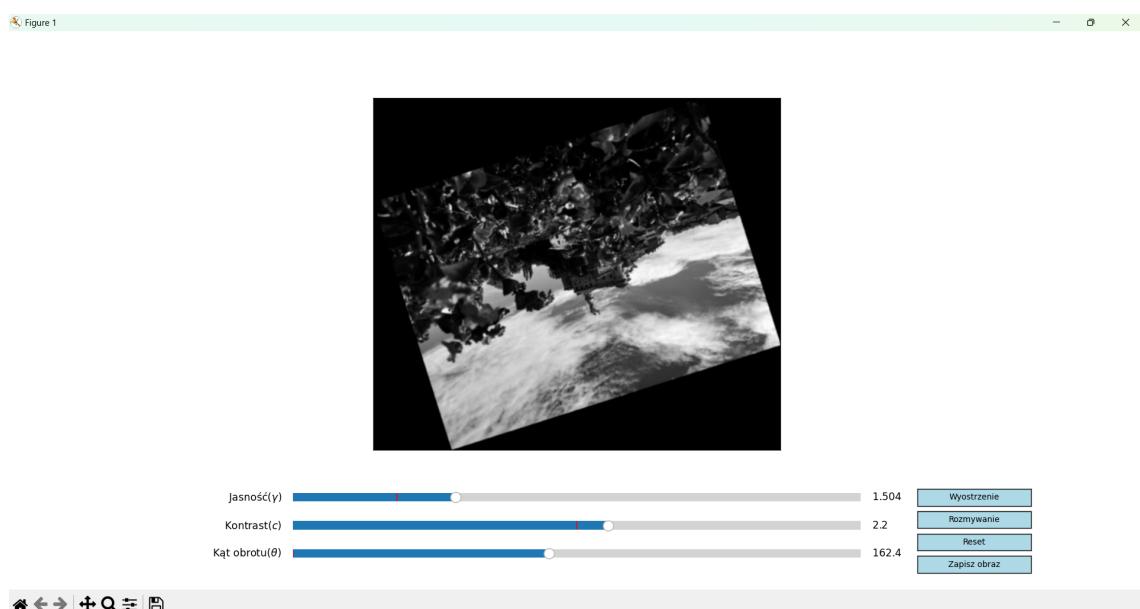
5 Interaktywny program

Używając wszystkich dotychczas omówionych przekształceń, dokonamy implementacji interaktywnego okna, w którym będziemy mogli rozjaśnić/przyciemnić zdjęcie, zwiększyć/zmniejszyć kontrast w obrazie, obrócić zdjęcie o dowolny kąt oraz rozmyć lub wyostrzyć (obraz), a następnie zapisać przekształcony obraz. Skalowanie nie będzie dostępne explicite w dostępnych opcjach funkcjonalności programu, ale zostanie użyte wewnętrznie, by użytkownik ustawiając zdjęcie wejściowe o dość dużym rozmiarze, nie spowalniał bardzo działania programu. Dlatego większy wymiar obrazu, jeśli okaże się większy od 500 pikseli, zostanie zredukowany do tej liczby. Przed uruchomieniem programu użytkownik musi wstawić ścieżkę do zdjęcia, które chce przekształcać, by program mógł zadziałać bez przeszkód (w kodzie jest komentarz na ten temat, w którym miejscu należy to wykonać).

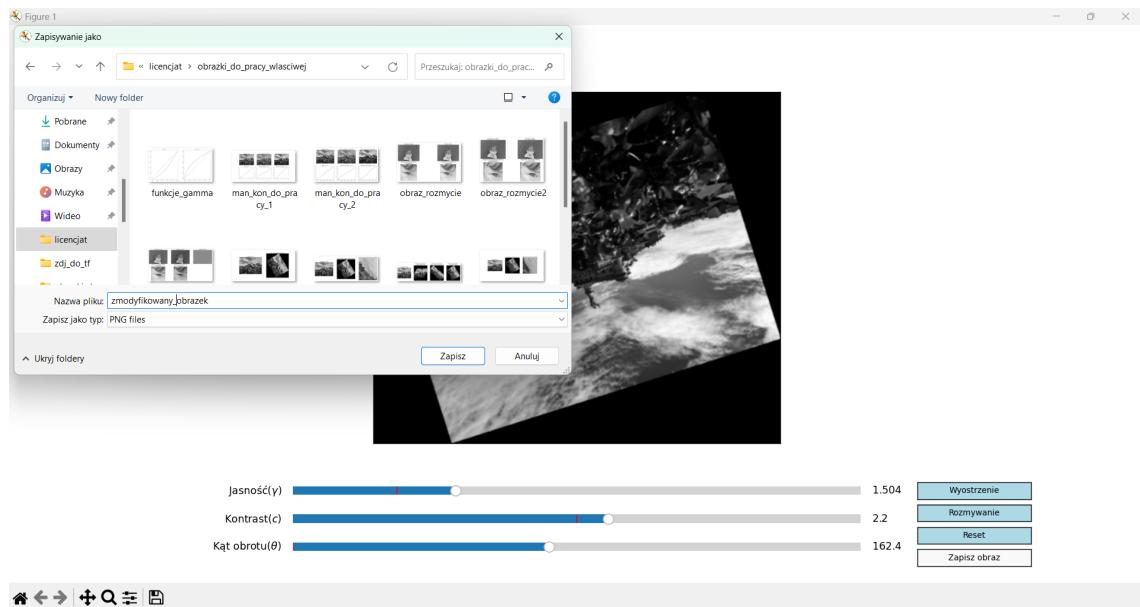
Program (w języku Python) został napisany w taki sposób, że zalecaną kolejnością wykonywania przekształceń jest skorzystanie najpierw z manipulacji jasnością/kontrastem czy kątem obrotu, a następnie skorzystanie z opcji wyostrzania i rozmycia. Jeśli nie skorzysta się z zalecanej kolejności, to program może zapomnieć część wykonanych operacji. Będzie się działało tak wtedy, kiedy po wykonaniu rozmycia/wyostrzenia skorzystamy z opcji zmiany jasności,kontrastu bądź kąta obrotu. Program zapomni o wykonanych wyostrzeniach i rozmyciach. Jest tak dlatego, że operację obrotu zawsze musimy wykonywać na oryginalnym zdjęciu (by wymiary jak i kąt obrotu były odpowiednie), a nie na już przekształconym (gdyż mógłby być on już obrócony, a kąt podany w interfejsie dotyczy kąta obrotu dla oryginalnego zdjęcia). Z racji, że program został napisany tak, iż opcje z suwakami aktualizowane są wspólnie, dlatego podział kolejności wykonywanych czynności dotyczy ich całej grupy. Program zamkamy klikając „X” w prawym górnym rogu ekranu. Pełny kod implementacji programu został zawarty w listingu 1. Przykładowe działanie i wygląd okna interfejsu programu zostały pokazane na ilustracjach poniżej.



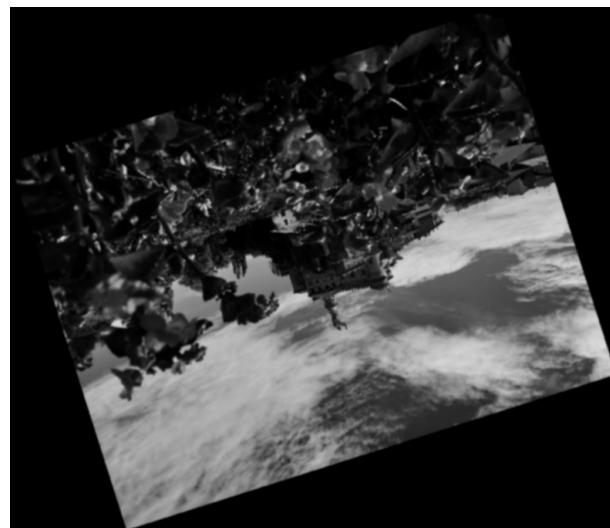
Rysunek 9: Początkowy wygląd okna interfejsu zaraz po uruchomieniu programu.



Rysunek 10: Wygląd okna programu po użyciu różnych dostępnych opcji.



Rysunek 11: Działanie opcji „Zapisz obraz”.



Rysunek 12: Uzyskany za pomocą programu zmodyfikowany obraz.

6 Wykrywanie okręgów na obrazie

Wykrywanie punktów, prostych czy krzywych jest bardzo ważnym etapem w zaawansowanej analizie obrazów. Jeśli chcemy wykryć podobne wzorce występujące na obrazach to najlepiej najpierw zacząć od znalezienia wyrażeń matematycznych, opisujących znane nam kształty czy obiekty. Pierwszym etapem procesu jest zamiana obrazu na binarny poprzez wykrywanie krawędzi. Jest to najbardziej podstawowa czynność, jaką musimy wykonać na zdjęciu, jeśli chcemy je poddać dalszej analizie.

Nasz algorytm wykrywania krawędzi jest adaptacją algorytmu wykrywania krawędzi Canny'ego. Składa się on z kilku następujących kroków:

1. Redukcja szumu.
2. Obliczanie gradientu.
3. Wyznaczenie „maksimów lokalnych”.
4. Wyznaczenie silnych i słabych krawędzi.
5. Szukanie krawędzi z histerezą.

Wyjaśnijmy teraz, na czym polegają kolejne kroki algorytmu. Na początek wykonujemy redukcję szumu, czyli rozmywamy obraz nakładając filtr Gaussa. Wykonujemy tę operację, by zmniejszyć wpływ szumu w dalszym procesie wykrywania krawędzi. W dalszym kroku będziemy korzystać z pochodnej, która jest wrażliwa na występujący szum, dlatego ten krok jest niezbędny dla poprawności całego algorytmu. W drugim kroku wyznaczamy gradient obrazu, czyli wektor pochodnych cząstkowych, w naszym przypadku jest on postaci:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}.$$

Zwykle do wyznaczenia pochodnych cząstkowych w algorytmie Canny'ego używa się filtrów Sobela (których postać wynika wprost z wzoru na pochodną (5)), stąd

zostaną one użyte w naszym algorytmie. Macierz jądra dla parametru x jest postaci:

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix},$$

zaś macierz dla parametru y ma postać:

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}.$$

Po wyznaczeniu gradientu obrazu, liczymy moduł gradientu i kierunek gradientu, według następujących wzorów:

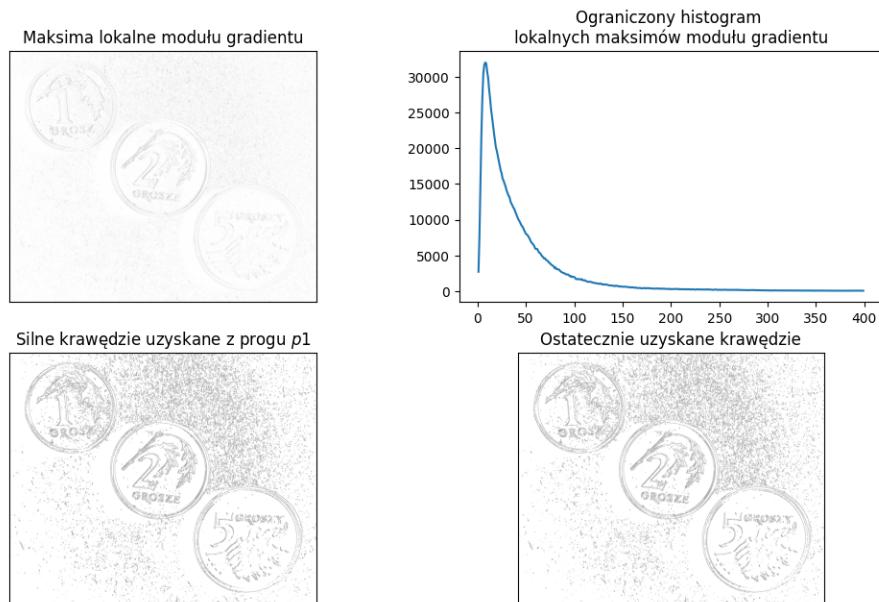
$$|\nabla f| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

oraz

$$\theta = \arctg \left(\frac{\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial y}} \right).$$

W samym algorytmie do wyznaczenia kierunku gradientu używamy funkcji `np.arctan2()`, a nie liczymy wprost według wcześniejszej podanego wzoru. Mając już kierunki gradientu możemy je skwantyzować do 4 kierunków N-S (poziomy), W-E (pionowy), NE-SW oraz NW-SE. Wykonujemy kwantyzację kierunków, bo skoro pracujemy na obrazie, to możemy się przemieszczać zasadniczo tylko wzdłuż tych 4 kierunków, więc rozważanie większej ilości nie miałoby sensu. W tym momencie przechodzimy do kroku 3, czyli szukamy „maksimów lokalnych” modułu gradientu. Polega to na tym, że każdy piksel porównujemy z dwoma pikselami, znajdującymi się w najbliższym sąsiedztwie i leżącymi zgodnie z jego (tego piksela) kierunkiem gradientu. Sprawdzamy, czy moduł jego gradientu jest większy od modułu gradientu pozostałych, jeśli tak jest, to jest to nasze „lokalne maksimum” i jest ono dodane do odpowiedniej macierzy. W ten sposób następuje już wstępna selekcja pikseli, dzięki której uzyskujemy dość „grube krawędzie”. W tym momencie przechodzimy do kroku 4, gdzie wybieramy silne i słabe krawędzie.

W tym etapie ustalamy wartość dwóch progów p_1 i p_2 (oznaczenia progów są zgodne z oznaczeniami występującymi w kodzie implementacji), które pozwala na odpowiednie przyporządkowanie pikseli. Jeśli element z macierzy zawierającej „maksima lokalne” ma wartość wyższą lub równą p_1 jest on zaklasyfikowany jako silna krawędź, natomiast jeśli ma wartość mniejszą od p_1 , ale wyższą lub równą od p_2 , zostaje zakwalifikowany jako słaba krawędź. Każdy zakwalifikowany piksel ląduje w odpowiedniej macierzy, co jest reprezentowane przez zmianę wartości odpowiedniego elementu z 0 na 255. Następnie zostaje wykonany ostatni krok, czyli wyznaczenie krawędzi z histerezą. Polega on na tym, że musimy zdecydować, czy zostają uznane za krawędzie piksele sklasyfikowane jako słabe krawędzie. Nasza metoda będzie akceptowała te piksele, których sąsiadami są piksele sklasyfikowane jako silne krawędzie. Dodatkowo warto zwrócić uwagę na fakt, że po akceptacji słabej krawędzi staje się ona silną krawędzią i taką pełni później rolę. W zasadzie powinniśmy wykonywać procedurę akceptacji słabych krawędzi, aż upewnilibyśmy się, że każda słaba krawędź została poprawnie zaklasyfikowana, czyli te, które mogły zostać zaakceptowane, zostały zaakceptowane. Jednak, aby uzyskać zadowalające rezultaty, wystarczy już kilka powtórzeń tej procedury, stąd ograniczymy się do 4-krotnego powtórzenia procesu akceptacji. W ten sposób otrzymujemy macierz, której elementy mają wartość 0 lub 255, oznaczające brak lub występowanie krawędzi.



Rysunek 13: Obrazy ilustrujące efekt po ważnych etapach algorytmu wykrywania krawędzi oraz pomocniczy histogram.

Ilustracje na rysunku 13, obrazujące kolejne etapy algorytmu, zostały zmodyfikowane dla lepszej prezentacji. Modyfikacja polega na odwróceniu znaczenia kolorów. Z reguły (i tak jest również w implementacji algorytmu) brak krawędzi oznaczany jest kolorem czarnym, a jej występowanie kolorem białym, zaś na ilustracjach znaczenie kolorów zostało odwrócone (również dla macierzy „maksimów lokalnych”, gdzie nowa wartość piksela powstała z różnicy wartości 255 i oryginalnej wartości piksela). Progi, jakie były użyte do wykrycia krawędzi w powyższym przypadku, to 100 i 70. Dodatkowo został umieszczony również ograniczony histogram maksimów lokalnych modułu gradientu (tutaj już dla wartości bez modyfikacji), ponieważ możemy z niego wywnioskować, jakie powinny być wybrane wartości progów. Powinny być one dobierane indywidualnie do każdego zdjęcia i procesu, jakim zostanie później poddany obraz, gdyż mają bardzo duże znaczenie na wynik końcowy algorytmu.

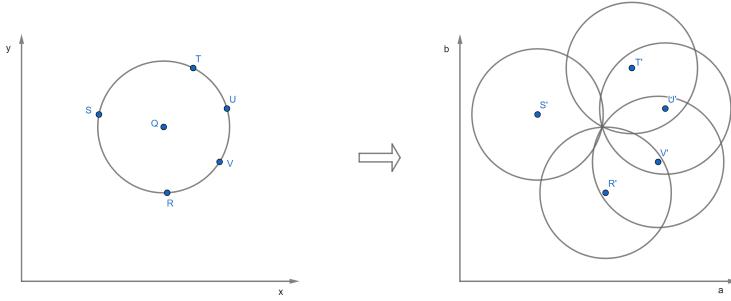
Możemy już przejść do algorytmu wykrywania okręgów. Jedną z najbardziej popularnych metod detekcji okręgów jest algorytm wykorzystujący transformatę Hougha. Jest to metoda pozwalająca na wykrycie regularnych kształtów w obrazie binarnym. Zajmiemy się tylko szczególnym przypadkiem transformaty Hougha, czyli transformacją Hougha dla okręgów. Okrąg o promieniu r i środku w punkcie (a, b) możemy wyrazić wzorem:

$$(x - a)^2 + (y - b)^2 = r^2$$

lub (parametrycznie)

$$\begin{cases} x = a + r\cos(\beta), & \beta \in [0, 2\pi] \\ y = b + r\sin(\beta), & \beta \in [0, 2\pi] \end{cases}$$

Założmy, że znamy wartość r . Wtedy naszą przestrzenią parametrów będzie przestrzeń (a, b) , to znaczy, że mając okrąg w przestrzeni (x, y) , każdy punkt okręgu w przestrzeni (a, b) będzie reprezentowany jako okrąg o środku w tym właśnie punkcie i promieniu r . Poniżej została dodana schematyczna ilustracja wspomagająca zrozumienie tego zagadnienia.



Rysunek 14: Jak działa transformacja Hougha dla okręgów o znanym promieniu.

Jak możemy już dostrzec ze schematycznego rysunku, że wszystkie okręgi w przestrzeni (a, b) , powstałe z punktów okręgu z przestrzeni (x, y) , przecinają się w jednym punkcie, tym punktem jest szukany przez nas środek okręgu oryginalnego i właśnie tę własność będziemy wykorzystywali.

Jednak jeśli wartość promienia r jest nieznana za przestrzeń parametrów przyjmujemy przestrzeń (a, b, r) , czyli 3-wymiarową przestrzeń. W tej sytuacji każdy punkt z przestrzeni (x, y) jest reprezentowany przez powierzchnię stożka w przestrzeni parametrów. Jednak praca na 3-wymiarowej nieograniczonej przestrzeni parametrów wydaje się dość uciążliwa i czasochłonna, stąd w naszym algorytmie ograniczymy zakres wartości promienia r do liczb całkowitych z przedziału podanego przez użytkownika (użytkownik podaje minimalną wartość promienia (**rmin**), to jest lewy kraniec przedziału, oraz maksymalną wartość promienia (**rmax**), to jest prawy kraniec przedziału). Jest to już pierwsza wada naszego algorytmu, ponieważ użytkownik na samym początku musi orientować się, w jakim przedziale znajdują się wartości promieni szukanych okręgów.

Nasz algorytm będzie przebiegał w następujących krokach:

1. Dyskretyzacja przedziału wartości β .
2. Akumulacja „głosów” w przestrzeni (a, b, r) .
3. Ostateczny wybór okręgów.

W pierwszym kroku musimy zdyskretyzować wartości z przedziału $[0, 2\pi]$, ponieważ nie jesteśmy w stanie wyznaczyć naszego algorytmu dla wszystkich wartości z tego przedziału. Stąd warto wziąć wartości postaci $\frac{2\pi i}{n}$, gdzie $i = 0, \dots, n-1$, zaś $n \in \mathbb{N}$. W naszym przypadku przyjmiemy $n = 100$. Następnie dla podanego przez

użytkownika zakresu promieni (promienie też są całkowite) wyznaczymy krótki postaci $(r, r\cos(\beta), r\sin(\beta))$, gdzie dodatkowo zaokrąglamy otrzymane wartości do najbliższych liczb rzeczywistych. W następnym kroku reprezentujemy wszystkie punkty krawędziowe okręgiem w przestrzeni (a, b, r) , dla wszystkich wcześniej wygenerowanych krotek, dodając wartość 1 do odpowiedniego elementu z tej przestrzeni. W ten sposób następuje głosowanie i powstają tak zwane „piki”. W ostatnim kroku następuje selekcja kandydatów na okręgi. Pierwszym kryterium jest przekroczenie progu (**threshold**), ustawionego przez użytkownika, przez liczbę głosów podzieloną przez n , drugim są zadane przez nas warunki logiczne, które porównują parametry zaakceptowanych okręgów z parametrami kandydata. Warunki te mają zapobiec akceptacji nadmiernej ilości okręgów, zwłaszcza takich leżących blisko siebie, które w zasadzie pochodzą od tego samego obiektu z obrazu. Brak tego warunku może spowodować, że zamiast wykrycia jednego okręgu na obiekcie, będzie ich kilka, dodatkowo ich środki będą blisko położone, a promienie będą miały zbliżone wartości. Dzieje się tak ponieważ krawędzie, które otrzymaliśmy, nie są idealne. Są często po-przerywane i lekko zniekształcone, przez co nie otrzymujemy idealnego przecięcia się wszystkich okręgów w jednym punkcie jak było na rysunku 14. Dlatego warto dodać warunek logiczny zapobiegający takim sytuacjom, choć może on także powodować zmniejszenie ilości wykrytych okręgów. W naszym algorytmie będziemy sprawdzali, czy różnica promieni lub odległość środków (dla kandydata z już zaakceptowanymi okręgami) jest większa niż połowa długości minimalnej wartości promienia (podanego przez użytkownika).

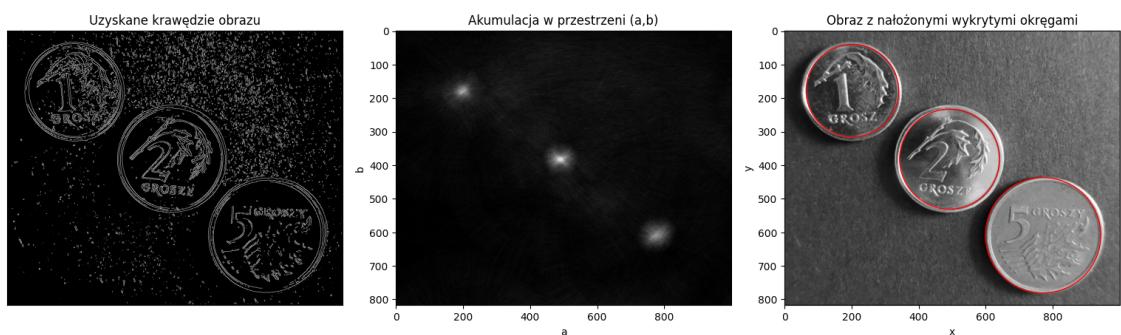
Skuteczność algorytmu zależy od wielu czynników, pierwszym jest już obraz krawędziowy, który dostajemy, kolejnymi są wszystkie ustawiane przez nas parametry. Przykładowo, jeśli źle ustawimy zakres wartości promieni, algorytm może nie wykryć żadnych okręgów lub wręcz przeciwnie wykryć zbyt dużo (tak naprawdę na obrazie nie występujących).

Choć algorytm wydaje się w pełni poprawny, ma jednak pewną wadę - czas jego pracy nie jest zbyt optymistyczny. Istnieje jednak możliwość poprawy działania naszego algorytmu z wykorzystaniem wcześniej wyznaczonych kierunków gradientu. Jak wiemy gradient jest prostopadły do stycznej do krawędzi wyznaczonej w tym punkcie, zatem dla okręgu gradient leży w tym samym kierunku, co promień

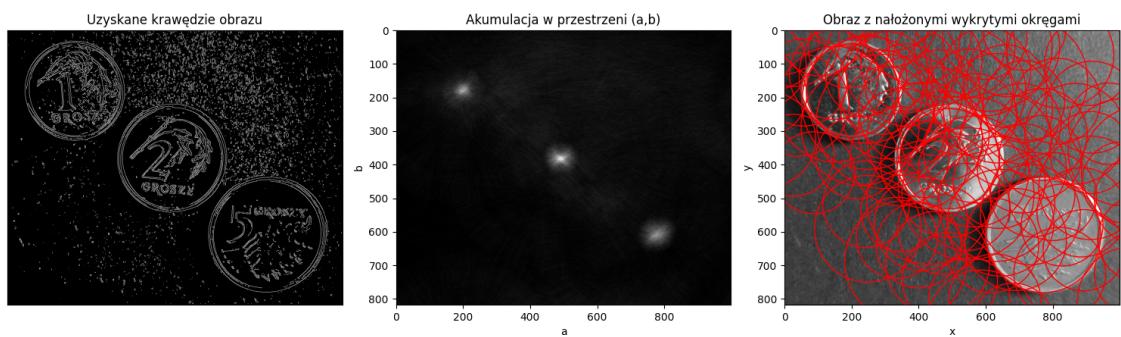
tego okręgu (wyznaczony) w tym punkcie. Dzięki tej obserwacji możemy się ograniczyć do narysowania dwóch punktów, położonych zgodnie z kierunkiem gradientu, a nie jak poprzednio rysowaliśmy ich aż 100. Pozwala to na znaczne przyspieszenie działania algorytmu. Zmieniona została też formuła progowania. Tym razem liczbę głosów dzielimy przez wartość promienia, choć formuła wydaje się mieć dużo sensu, jak okaże się później ma pewne wady. Pozostałe czynności pozostają bez zmian.

Innym sposobem na modyfikację tego algorytmu jest pomysł, by najpierw wspólnie dla wszystkich wartości promieni głosować na kandydatów na środki okręgów jednocześnie tworząc dla każdego środka listę promieni. Następnie po akceptacji środka, promieniem tego okręgu będzie taka wartość, która na liście pojawiła się najczęściej. Ta modyfikacja ma tę zaletę, że będziemy poprawniej wybierać odpowiednie punkty jako środki okręgu. Niestety wadą tego rozwiązania jest fakt, że dla jednego środka wybierzemy tylko jeden promień.

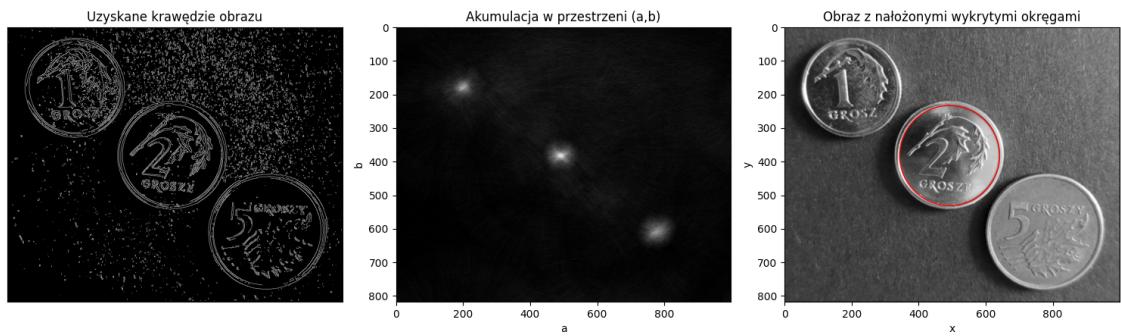
Teraz przejdziemy do praktycznej części, to znaczy przedstawimy działanie algorytmów na przykładach. Najpierw bardziej szczegółowo z użyciem wersji, rysującej tylko punkty w kierunku gradientu, omówimy ogólne problemy pojawiające się podczas pracy z algorytmami, a następnie pokażemy również kilka przykładów dla pozostałych wersji algorytmu. Każdy rysunek przykładu będzie zawierać 3 ilustracje, na pierwszej (licząc od lewej strony) będzie ukazany obraz krawędziowy, uzyskany w wyniku użycia algorytmu wykrywania krawędzi, na drugiej będzie ukazana przestrzeń akumulacji (a, b) wspólna dla wszystkich wartości promieni, na ostatniej ilustracji zaś będzie pokazane zdjęcie z dodanymi na czerwono wykrytymi okręgami. W opisie rysunku zostaną podane parametry zastosowane podczas użycia algorytmu. Będziemy pracować na zdjęciach monet oraz ulic (zawierających znaki drogowe). Poniżej zostanie zaprezentowana seria przykładów, następnie zostaną omówione pojawiające się problemy.



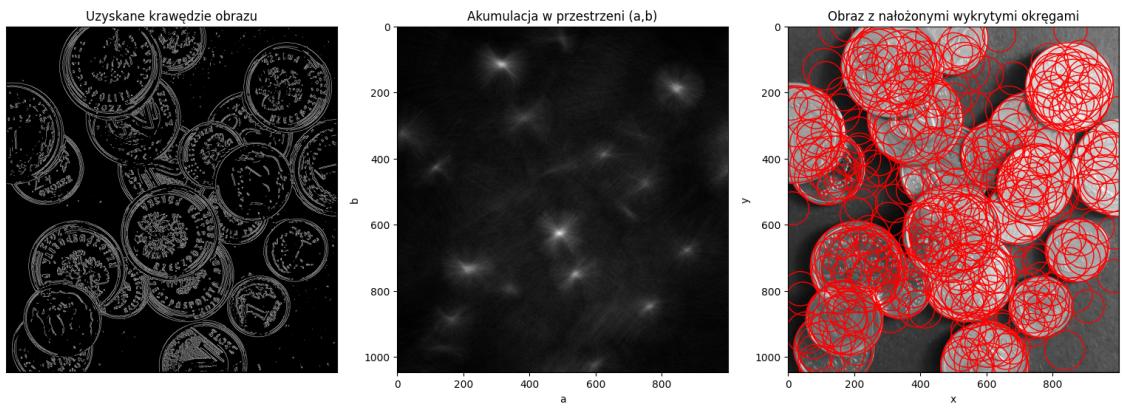
Rysunek 15: p1= 100, p2= 70, rmin= 120, rmax= 200, threshold= 0.1



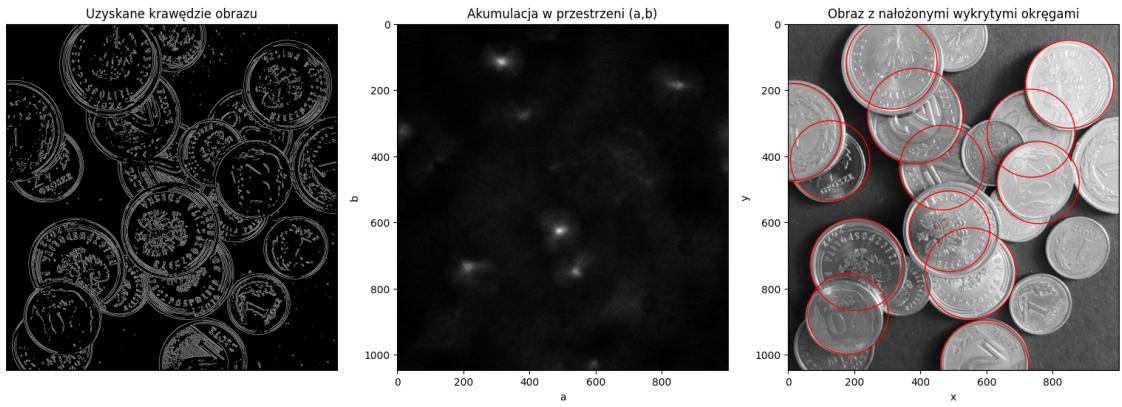
Rysunek 16: p1= 100, p2= 70, rmin= 120, rmax= 200, threshold= 0.03



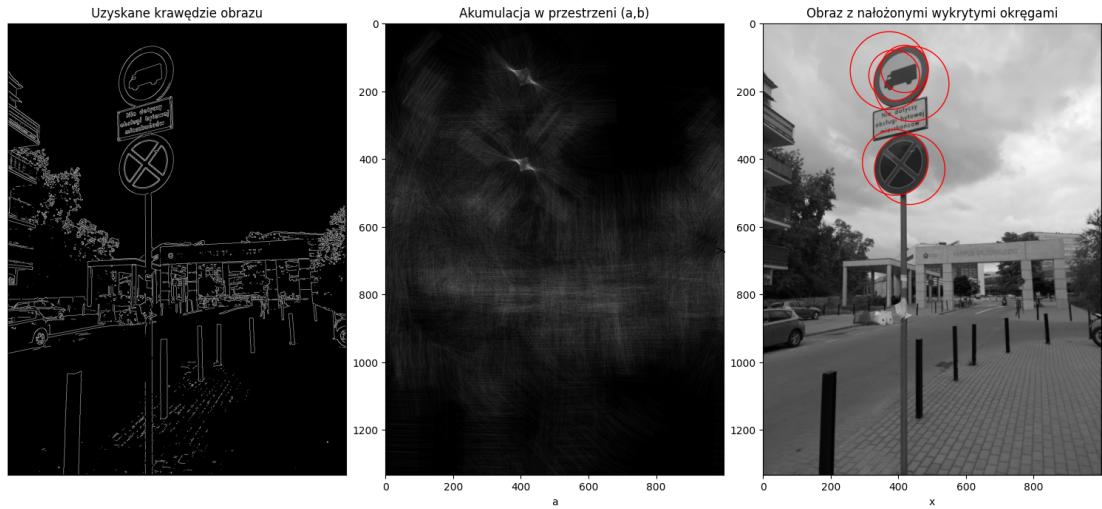
Rysunek 17: p1= 100, p2= 70, rmin= 120, rmax= 200, threshold= 0.2



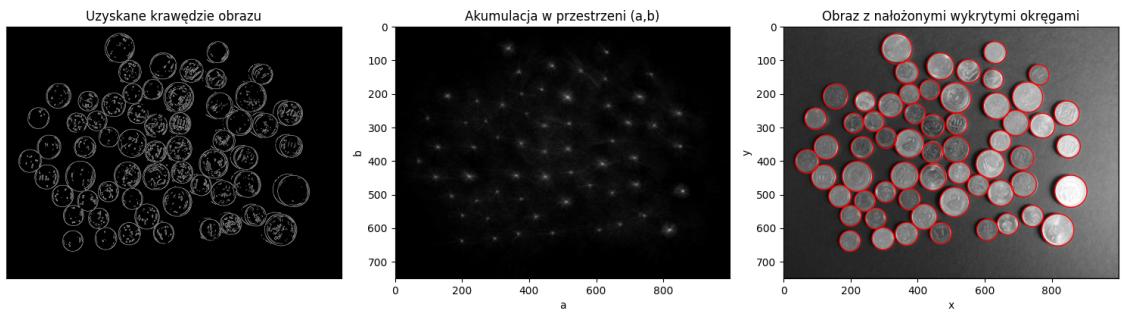
Rysunek 18: $p1=100$, $p2=70$, $rmin=50$, $rmax=200$, $threshold=0.1$



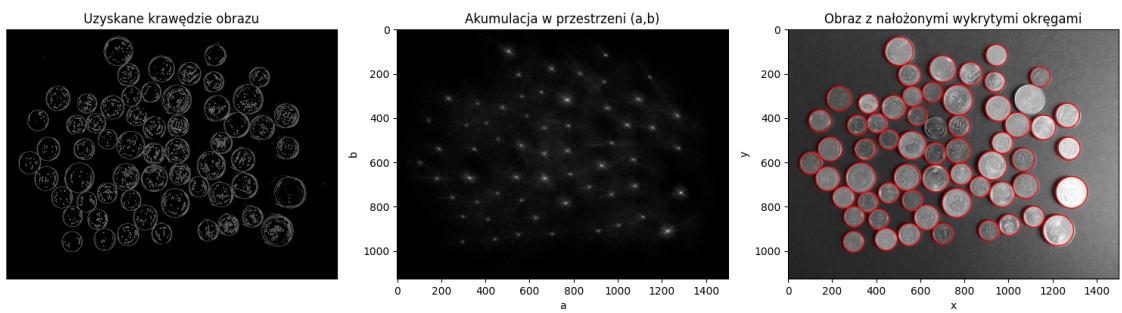
Rysunek 19: $p1=100$, $p2=70$, $rmin=120$, $rmax=200$, $threshold=0.1$



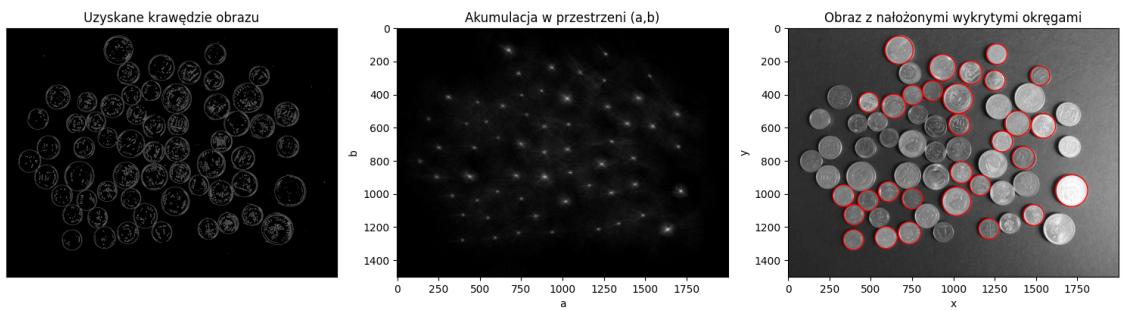
Rysunek 20: $p1=100$, $p2=70$, $rmin=70$, $rmax=200$, $threshold=0.15$



Rysunek 21: $p1 = 100$, $p2 = 70$, $rmin = 25$, $rmax = 70$, $threshold = 0.5$, rozmiar zdjęcia 1000×750



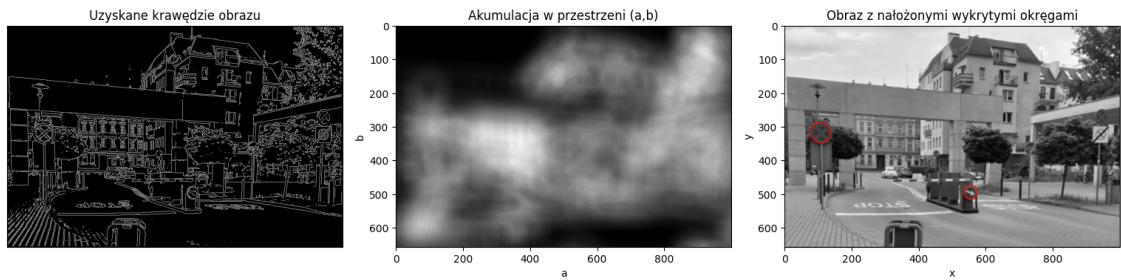
Rysunek 22: $p1 = 100$, $p2 = 70$, $rmin = 35$, $rmax = 110$, $threshold = 0.5$, rozmiar zdjęcia 1500×1125



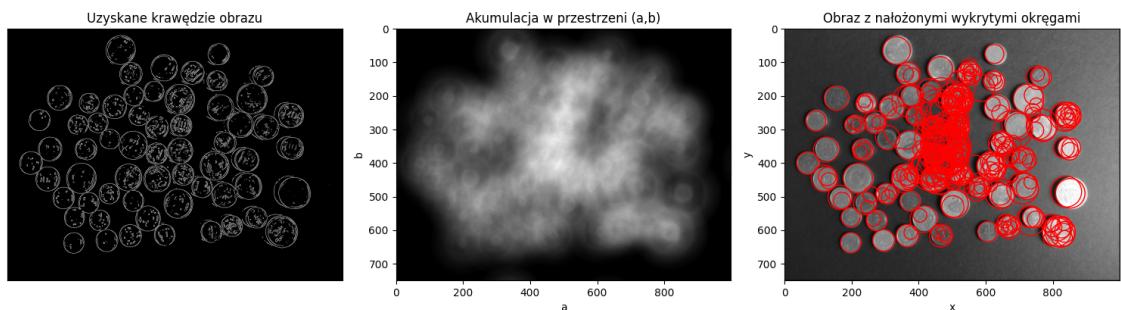
Rysunek 23: $p1 = 100$, $p2 = 70$, $rmin = 50$, $rmax = 140$, $threshold = 0.5$, rozmiar zdjęcia 2000×1500

Po analizie serii powyższych przykładów zauważmy, że jeśli wartość $rmin$ (podana przez użytkownika) będzie dużo mniejsza niż wartość promieni szukanych okręgów, najczęściej w efekcie algorytm wykrywa zbyt wiele okręgów (rysunek 18). Jest tak również w przypadku, kiedy zaniżymy za bardzo wartość $threshold$

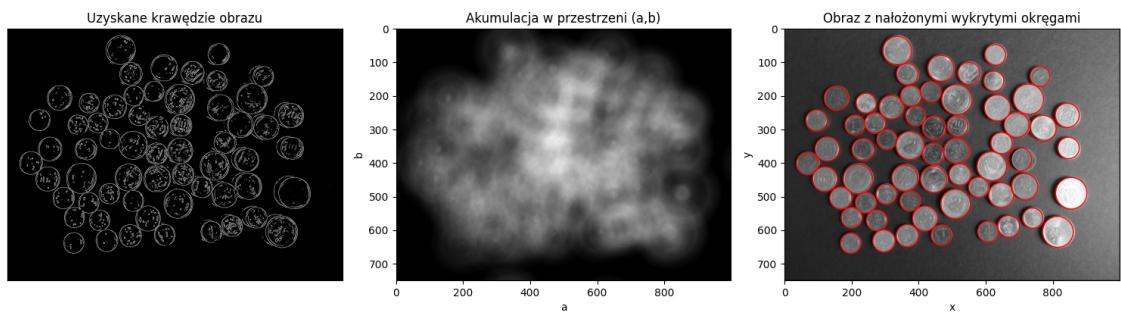
(rysunek 16). Z kolei jeśli zbytnio zawyżymy wartość **rmin** algorytm naturalnie nie wykryje okręgów o małych promieniach (rysunek 19). Jeśli zawyżymy wartość **threshold** to najprawdopodobniej nie zostaną wykryte niepełne okręgi, czy okręgi, których krawędzie miały zniekształcenia (rysunek 17). Warto zwrócić uwagę, że przy odpowiedniej wartości **threshold**, na przykład do kształtów eliptycznych, zostaną dopasowane okręgi pasujące do fragmentów tego kształtu (rysunek 20). Ostatnie 3 przykłady (rysunki 21, 22, 23 ukazują to samo zdjęcie, ale mające inną ilość pikseli) zostały zestawione, by pokazać, że wraz ze wzrostem promienia maleje dokładność „przecinania się okręgów w jednym punkcie” (to znaczy, że podczas akumulacji głosów większe okręgi wbrew pozorom wcale nie dostają więcej głosów niż mniejsze, na rysunku 21 wykryto wszystkie okręgi, na 22 nie wykryto dużej monety znajdującej się w okolicy prawego górnego rogu, zaś na rysunku 23 nie została już wykryta większość większych monet), co pokazuje, że nie warto szukać kształtów na obrazach o zbyt dużym rozmiarze (najlepiej wcześniej go zeskalować z użyciem dość dokładnej metody interpolacji). Teraz pokażemy kilka przykładów z użyciem pierwszej wersji algorytmu.



Rysunek 24: $p1= 100$, $p2= 70$, $rmin= 20$, $rmax= 70$, $threshold= 0.48$

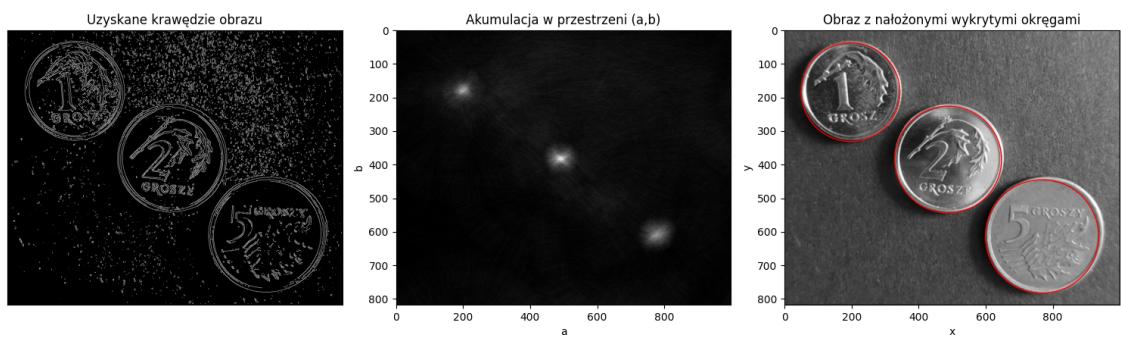


Rysunek 25: $p1= 100$, $p2= 70$, $rmin= 20$, $rmax= 70$, $threshold= 0.3$

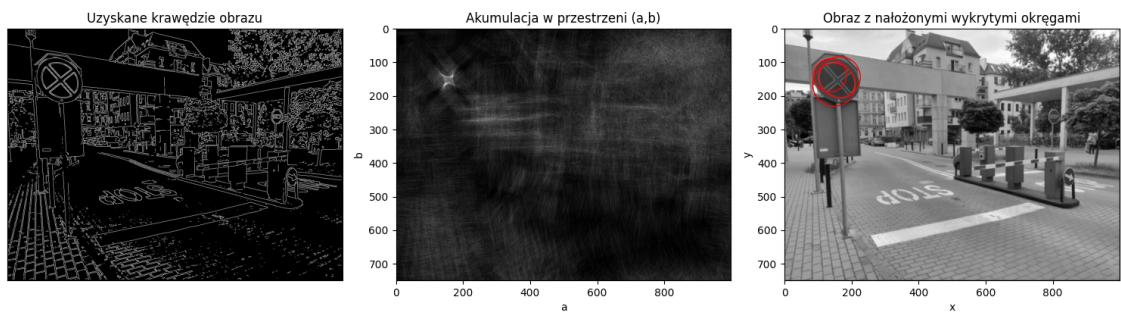


Rysunek 26: $p1= 100$, $p2= 70$, $rmin= 30$, $rmax= 70$, $threshold= 0.4$

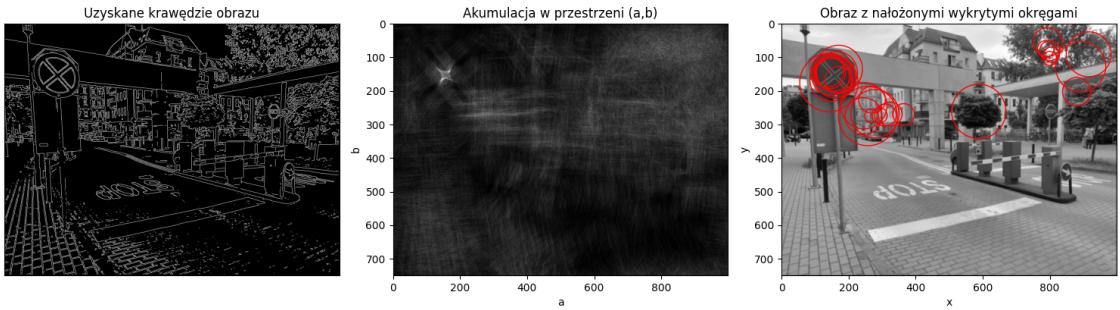
I w końcu również kilka przykładów dla ostatniej wersji algorytmu.



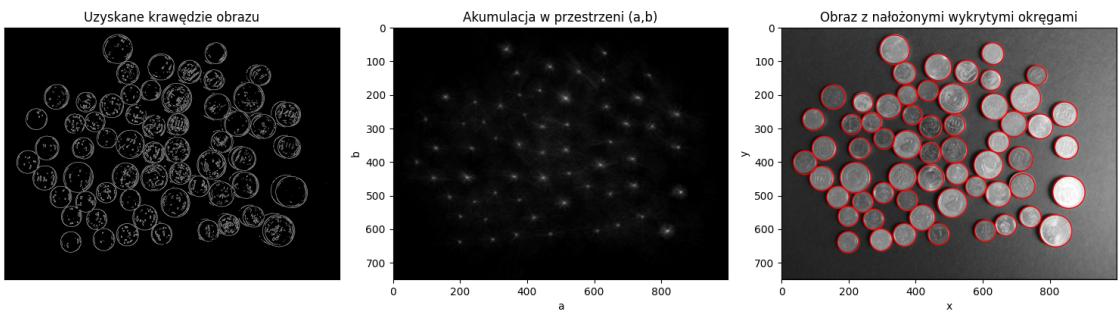
Rysunek 27: $p1= 100$, $p2= 70$, $rmin= 120$, $rmax= 200$, $threshold= 0.5$



Rysunek 28: $p1= 100$, $p2= 70$, $rmin= 20$, $rmax= 100$, $threshold= 0.7$



Rysunek 29: $p_1 = 100$, $p_2 = 70$, $r_{min} = 20$, $r_{max} = 100$, $threshold = 0.5$



Rysunek 30: $p_1 = 100$, $p_2 = 70$, $r_{min} = 25$, $r_{max} = 70$, $threshold = 0.5$

Jak widzimy każdą wersją algorytmu możemy skutecznie wykryć okręgi na obrazie, dzięki odpowiedniemu dobraniu parametrów. Każda z przedstawionych wersji algorytmu ma swoje indywidualne wady i zalety oraz najpewniej każda sprawdziłaby się lepiej w innych sytuacjach (ale to wymagałoby szerszych testów). To jakiego efektu potrzebuje użytkownik nakieruje go na to, jakiej wersji powinien użyć. W listingu 2 został zawarty algorytm wykrywania krawędzi wraz z drugą wersją naszego algorytmu wykrywania okręgów.

Ta część pracy powstała w oparciu o [5], [6], [7], [8], [9] oraz wiedzę i materiały z przedmiotu „Wstęp do zastosowań analizy falkowej” prowadzonym w semestrze letnim roku akademickiego 2022/23 w Instytucie Matematyki Uniwersytetu Wrocławskiego.

7 Kody implementacji

Listing 1: Kod interaktywnego programu wykonującego podstawowe operacje na obrazie

```
from tkinter import filedialog
import cv2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button

def fun_image(image, f_w, c):
    n_row, n_col = image.shape
    image = image.copy()
    for i in range(n_row):
        for j in range(n_col):
            x = image[i][j] / 255
            image[i][j] = np.floor(f_w(x, c=c) * 255)
    return image

def gamma_operation(gamma, image):
    n_row, n_col = image.shape
    image = image.copy()
    for i in range(n_row):
        for j in range(n_col):
            x = image[i][j] / 255
            if x > 0:
                image[i][j] = np.floor(x ** gamma * 255)
            else:
                image[i][j] = 0
    return image

def f1(x, c):
    if c > 0:
        return ((1 / (2 * np.arctan(c * np.tan(1 / 2)))) *
                np.arctan(c * 2 * np.tan(1 / 2) * (x - 1 / 2)) + 1 / 2)
    elif c == 0:
        return x
    else:
        return ((1 / (2 * (-c)) * np.tan(1 / 2))) *
                np.tan(2 * np.arctan((-c) * np.tan(1 / 2)) * (x - 1 / 2)) + 1 / 2

def do_mask_on_image(im, kernel):
    im = np.double(im)
    n_row_kernel = kernel.shape[0]
    extension_image = np.pad(im, n_row_kernel // 2, mode='constant')
    image.blur = im.copy()
```

```

for i in range(im.shape[0]):
    for j in range(im.shape[1]):
        pixel_new_value = 0
        for m in range(n_row_kernel):
            for k in range(n_row_kernel):
                weight = kernel[m][k]
                neighbor = extension_image[i + m][j + k]
                pixel_new_value += weight * neighbor
        image.blur[i][j] = pixel_new_value

return image.blur

def scale_lap(im):
    min_val = np.min(im)
    scale_image = im - min_val
    scale_image = scale_image * (510 / np.max(scale_image))
    return scale_image - 255

def rotation(im, teta_st):
    angle = np.radians(teta_st)
    cosine = np.cos(angle)
    sine = np.sin(angle)
    mat_r_o = np.array([[cosine, sine], [-sine, cosine]])

    new_height = int(abs(sine) * im.shape[1] + abs(cosine) * im.shape[0])
    new_width = int(abs(cosine) * im.shape[1] + abs(sine) * im.shape[0])
    original_centre_height = (im.shape[0] - 1) / 2
    original_centre_width = (im.shape[1] - 1) / 2
    original_center = np.array([[original_centre_width], [original_centre_height]])
    new_centre_height = (new_height - 1) / 2
    new_centre_width = (new_width - 1) / 2
    new_center = np.array([[new_centre_width], [new_centre_height]])
    new_im = np.zeros((new_height, new_width))

    for i in range(new_height):
        for j in range(new_width):
            x = np.array([[j], [i]])
            y = np.round((mat_r_o @ (x - new_center)) + original_center)
            if int(y[1][0]) in range(im.shape[0]) and int(y[0][0]) in range(im.shape[1]):
                new_im[i][j] = im[int(y[1][0])][int(y[0][0])]

    return new_im

if __name__ == '__main__':
    # dla image1 nalezy zmienic sciezke pliku w tym miejscu
    image1 = cv2.imread("C:\\\\Users\\\\klaud\\\\Downloads\\\\zdj_do_tf\\\\wyjazd1.jpg",
                        cv2.IMREAD_GRAYSCALE)
    # aby moc skorzystac z programu, na sciezce do dowolnego zdjecia z komputera uzytkownika
    image1 = np.double(image1)

```

```

# skalowanie, jesli jest wymagane
if (image1.shape[0] > 500) or (image1.shape[1] > 500):
    kk = 500 / max(image1.shape[0], image1.shape[1])
    original_height, original_width = image1.shape
    new_height1, new_width1 = (int(original_height * kk), int(original_width * kk))
    sc_image = np.zeros((new_height1, new_width1))

for ii in range(new_height1):
    for jj in range(new_width1):
        xx = np.round(np.array([[jj], [ii]]) * (1 / kk))
        if (int(xx[1][0]) in range(image1.shape[0])) and
           (int(xx[0][0]) in range(image1.shape[1])):
            sc_image[ii][jj] = image1[int(xx[1][0])][int(xx[0][0])]

image1 = sc_image

# maski
gaussian_kernel_3x3 = 1 / 16 * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
laplacian_kernel_3x3 = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])

# zasadnicza konstrukcja pojawiajacego sie okna
fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25, right=0.75)
axcolor = 'lightblue'

g_0 = 1
c_0 = 0
teta_0 = 0

current_image = image1
ax.imshow(current_image, cmap='gray')
ax.set_xticks([]), ax.set_yticks([])

axs = plt.axes((0.25, 0.1, 0.5, 0.03), facecolor=axcolor)
axgamma = plt.axes((0.25, 0.15, 0.5, 0.03), facecolor=axcolor)
axteta = plt.axes((0.25, 0.05, 0.5, 0.03), facecolor=axcolor)

sc = Slider(axs, 'Kontrast($c$)', -20, 20, valinit=c_0)
sgamma = Slider(axgamma, 'Jasnosc($\gamma$)', 0.1, 5, valinit=g_0)
steta = Slider(axteta, 'Kat\xf3robota($\theta$)', 0, 360, valinit=teta_0)

def update(val):
    global current_image
    g = sgamma.val
    c = sc.val
    teta = steta.val
    current_image = image1.copy()
    current_image = fun_image(current_image, f1, c)
    current_image = rotation(current_image, teta)

```

```

        current_image = gamma_operation(g, current_image)
        ax.imshow(current_image, cmap='gray')
        fig.canvas.draw_idle()

sc.on_changed(update)
sgamma.on_changed(update)
steta.on_changed(update)

sharpax = plt.axes((0.8, 0.15, 0.1, 0.03))
blurax = plt.axes((0.8, 0.11, 0.1, 0.03))
resetax = plt.axes((0.8, 0.07, 0.1, 0.03))
saveax = plt.axes((0.8, 0.03, 0.1, 0.03))

button_sharp = Button(sharpax, 'Wystrzenie', color=axcolor, hovercolor='0.5')
button_blur = Button(blurax, 'Rozmywanie', color=axcolor, hovercolor='0.9')
button_reset = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')
button_save = Button(saveax, 'Zapisz obraz', color=axcolor, hovercolor='0.975')

def apply.blur(event):
    global current_image
    current_image = do_mask_on_image(do_mask_on_image(current_image, gaussian_kernel_3x3),
                                     gaussian_kernel_3x3)
    ax.imshow(current_image, cmap='gray')
    fig.canvas.draw_idle()

def apply.sharp(event):
    global current_image
    lap_mask = do_mask_on_image(current_image, laplacian_kernel_3x3)
    c = -1
    current_image = np.clip(current_image + c * scale_lap(lap_mask), 0, 255)
    ax.imshow(current_image, cmap='gray')
    fig.canvas.draw_idle()

def reset(event):
    sc.reset()
    sgamma.reset()
    steta.reset()
    global current_image
    current_image = image1
    ax.imshow(current_image, cmap='gray')
    fig.canvas.draw_idle()

def save.image(event):
    global current_image
    file_path = filedialog.asksaveasfilename(defaultextension=".png",
                                              filetypes=[("PNG files", "*.png"),
                                                         ("JPEG files", "*.jpg"),
                                                         ("All files", "*.*")])
    if file_path:

```

```

cv2.imwrite(file_path, current_image)

button.blur.on_clicked(apply_blur)
button.sharp.on_clicked(apply_sharp)
button.reset.on_clicked(reset)
button.save.on_clicked(save_image)

def resize_fonts(event=None):
    for button in [button_sharp, button.blur, button.reset, button.save]:
        bbox = button.ax.get_window_extent().transformed(fig.dpi_scale_trans.inverted())
        width, height = bbox.width, bbox.height
        fontsize = min(width, height) * 35
        button.label.set_fontsize(fontsize)
    fig.canvas.draw_idle()

fig.canvas.mpl_connect('resize_event', resize_fonts)
resize_fonts()

plt.show()

```

Listing 2: Kod algorytmu wykrywania krawędzi i okręgów na obrazie (wersja druga)

```

import cv2
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
from interaktywne_okno_new2 import do_mask_on_image # kod funkcji jest w listingu powyżej
import matplotlib.patches as patches

# wczytywanie obrazów
image1 = cv2.imread("C:\\\\Users\\\\klaud\\\\Downloads\\\\zdj_do_tf\\\\monety10.jpg",
                     cv2.IMREAD_GRAYSCALE)
image1 = np.double(image1)

# ustalenie progów
p1, p2 = 100, 70

# redukcja szumu
gaussian_kernel_3x3 = 1/16 * np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
il_g = 2
for i in range(il_g):
    image1 = do_mask_on_image(image1, gaussian_kernel_3x3)

# maski Sobela
v_sobel_mask = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])

h_sobel_mask = np.array([[-1, -2, -1],

```

```

[0, 0, 0],
[1, 2, 1]])

dx = do_mask_on_image(image1, v_sobel_mask)
dy = do_mask_on_image(image1, h_sobel_mask)

modul_of_gradient = np.zeros(dx.shape)
for i in range(modul_of_gradient.shape[0]):
    for j in range(modul_of_gradient.shape[1]):
        modul_of_gradient[i][j] = ((dx[i][j])**2 + (dy[i][j])**2)**(1/2)

# kierunek gradientu
angle2 = np.arctan2(dy, dx)
angle_of_gradient = angle2.copy()

# kwantyzacja kierunkow
for i in range(angle_of_gradient.shape[0]):
    for j in range(angle_of_gradient.shape[1]):
        if angle_of_gradient[i][j] < 0:
            angle_of_gradient[i][j] += np.pi
        if angle_of_gradient[i][j] <= np.pi/8 or angle_of_gradient[i][j] > 7*np.pi/8:
            angle_of_gradient[i][j] = 1 # E-W(poziomy)
        elif angle_of_gradient[i][j] <= 3*np.pi/8:
            angle_of_gradient[i][j] = 2
            # normalnie NE-SW, ale przy zmianie orientacji osi to NW-SE
        elif angle_of_gradient[i][j] <= 5*np.pi/8:
            angle_of_gradient[i][j] = 3 # N-S (pionowy)
        else:
            angle_of_gradient[i][j] = 4
            # normalnie NW-SE, ale przy orientacji osi OY wiodo to NE-SW

# szukanie lokalnych maksimow
max_loc_grad = np.zeros(image1.shape)

for i in range(1, image1.shape[0]-1):
    for j in range(1, image1.shape[1]-1):
        if angle_of_gradient[i][j] == 1:
            u, v = 0, 1
        elif angle_of_gradient[i][j] == 2:
            u, v = 1, 1
        elif angle_of_gradient[i][j] == 3:
            u, v = 1, 0
        else:
            u, v = 1, -1
        if ((modul_of_gradient[i][j] >= modul_of_gradient[i+u][j+v]) and
            (modul_of_gradient[i][j] >= modul_of_gradient[i-u][j-v])):
            max_loc_grad[i][j] = modul_of_gradient[i][j]

# wyznaczenie slabych i silnych krawedzi
strong_edges = np.zeros(max_loc_grad.shape)

```

```

weak_edges = np.zeros(max_loc_grad.shape)
for i in range(0, image1.shape[0]-1):
    for j in range(0, image1.shape[1]-1):
        if max_loc_grad[i][j] >= p1:
            strong_edges[i][j] = 255
        elif max_loc_grad[i][j] >= p2:
            weak_edges[i][j] = 255
        else:
            pass

# szukanie krawedzi z histereza
for k in range(1, 4):
    for i in range(1, image1.shape[0] - 1):
        for j in range(1, image1.shape[1] - 1):
            if weak_edges[i][j] == 255:
                if_any_strong = False
                for u in range(-1, 2):
                    for v in range(-1, 2):
                        if strong_edges[i+u][j+v] == 255:
                            if_any_strong = True
                            break
                if if_any_strong:
                    break
            strong_edges[i][j] = 255
            weak_edges[i][j] = 0
edges = strong_edges

# wykrywanie okregow, wersja z rysowaniem pkt z okrgu tylko w kierunku gradientu

# ustawianie parametrow
rmin = 120
rmax = 200
threshold = 0.2

acc_space = np.zeros(image1.shape)
acc = defaultdict(int)
for x in range(edges.shape[1]):
    for y in range(edges.shape[0]):
        if edges[y][x] != 0:
            teta = angle2[y][x]
            if teta < 0:
                teta += np.pi
            for r in range(rmin, rmax+1):
                for alpha in [teta, teta+np.pi]:
                    a = x - int(r*np.cos(alpha))
                    b = y - int(r*np.sin(alpha))
                    acc[(a, b, r)] += 1
                    if (a < image1.shape[1]) and (b < image1.shape[0]):
                        acc_space[b][a] += 1

```

```

circles = []
for k, v in sorted(acc.items(), key=lambda i: -i[1]):
    x, y, r = k
    if (v / r >= threshold and
        all(((x - xc) ** 2 + (y - yc) ** 2)**(1/2) > rmin/2 or
            abs(rc-r) > rmin/2 for xc, yc, rc in circles)):
        circles.append((x, y, r))
# parametry znalezionejnych okregow znajduja sie w circles

# tworzenie wykresow, do prezentacji wynikow, jak w przykladach

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

axes[0].imshow(edges, cmap='gray')
axes[0].set_title('Uzyskane\u0144 krawedzie\u0144 obrazu')
axes[0].set_xticks([])
axes[0].set_yticks([])

axes[1].imshow(acc_space, cmap='gray')
axes[1].set_title('Akumulacja\u0144 w\u0144 przestrzeni\u0144(a,b)')
axes[1].set_xlabel('a')
axes[1].set_ylabel('b')

axes[2].imshow(image1, cmap="gray")
axes[2].set_title('Obraz\u0144 z\u0144 nalozonymi\u0144 wykrytymi\u0144 okregami')
axes[2].set_xlabel('x')
axes[2].set_ylabel('y')

# Dodawanie wykrytych okregow
for x, y, r in circles:
    circle = patches.Circle((x, y), r, edgecolor="red", facecolor="none", linewidth=1)
    axes[2].add_patch(circle)

axes[2].set_aspect('equal', adjustable='box')

plt.tight_layout()
plt.show()

```

Literatura

- [1] Jonas Gomez, Luiz Velho, *Image Processing for Computer Graphics*, Springer, New York, 1997, ISBN 9780387948546.
- [2] Rafael C. Gonzalez, Richard E. Woods, *Digital Image Processing*, 4Th Edition, Pearson Education Limited, Harlow, England, 2018, ISBN 9781292223049.
- [3] Richard Szeliski, *Computer Vision: Algorithms and Applications*, Springer, Heidelberg, Germany, 2011, ISBN 9781848829343.
- [4] Amir Hossein Ashtari, Md Jan Nordin , Seyed Mostafa Mousavi Kahaki, *Double Line Image Rotation*, IEEE Transactions on Image Processing, 2015, vol. 24, no. 11, pp. 3370-2285.
- [5] Jaroslav Borovička, *Circle Detection Using Hough Transforms Documentation*, COMS30121-Image Processing and Computer Vision, 2003, vol. 48.
- [6] Huashan Ye, Guocan Shang, Lina Wang, Min Zheng, *A new method based on Hough Transform for quick line and circle detection*, 8th International Conference on BioMedical Engineering and Informatics, Shenyang, China, 2015, pp. 52-56.
- [7] Priyanka Mukhopadhyay, Bidyut Baran Chaudhuri *A survey of Hough Transform*, Pattern Recognition, 2014, vol. 48.
- [8] Yun Ou, Honggui Deng, Yang Liu, Zeyu Zhang, Xusheng Ruan, Qiguo Xu, Chengzuo Peng, *A Fast Circle Detection Algorithm Based on Information Compression*, Sensors, 2022, vol. 22, no. 19:7267.
- [9] Ali Rad, Karim Faez, Navid Qaragozlou, *Fast Circle Detection Using Gradient Pair Vectors*, in Proc. VIIth Digital Image Computing: Techniques and Applications, Sydney, Australia, 2003, pp. 879–888.