

# Wykrywanie defektów na płytach PCB z wykorzystaniem klasycznych metod przetwarzania obrazów

Projekt - Diagnostyka procesów  
Jakub Dufke s189016

## 1. Wstęp

Celem niniejszego opracowania była implementacja oraz empiryczna weryfikacja metody wykrywania defektów na płytach PCB (ang. *Printed Circuit Board*) na podstawie podejścia opisanego w pracy naukowej: “Comparative Study of Image Processing and Transfer Learning Techniques for an Automated PCB Fault Detection System”, opublikowanej w czasopiśmie *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, Volume 9 Issue VI, czerwiec 2021.

W ramach pracy zrealizowano i oceniono klasyczną metodę opartą na porównaniu obrazu referencyjnego z obrazem testowym. Celem było odwzorowanie tego podejścia w języku Python z użyciem biblioteki OpenCV oraz ocena skuteczności wykrywania typowych defektów takich jak: braki otworów, zwarcia, nadmiar ścieżek oraz nieciągłości.

## 2. Implementacja algorytmu

### 2.1 Wczytywanie i skalowanie obrazów

Proces rozpoczyna się od wczytania dwóch obrazów: obrazu referencyjnego, przedstawiającego poprawnie wykonaną płytę PCB bez defektów, oraz obrazu testowego, który może zawierać różnego rodzaju uszkodzenia. Oba obrazy są początkowo wczytywane w przestrzeni barw RGB, a następnie konwertowane do skali szarości za pomocą funkcji `cv2.cvtColor()`, co upraszcza dalsze operacje przetwarzania obrazu.

Ze względu na ograniczoną przestrzeń roboczą interfejsu graficznego, obrazy poddawane są skalowaniu. Skalowanie odbywa się proporcjonalnie (z zachowaniem współczynnika proporcji). Istotne jest, że jeśli obraz już mieści się w tych wymiarach, nie jest on powiększany, co zapobiega sztucznemu rozciągnięciu i pogorszeniu jakości. Dzięki temu obrazy pozostają przejrzyste i umożliwiają komfortowy podgląd oraz analizę w GUI aplikacji.

## 2.2 Rozmycie Gaussowskie

Po konwersji do skali szarości obrazy są poddawane operacji rozmycia za pomocą filtra Gaussa o rozmiarze jądra  $3 \times 3$ . Rozmycie to realizowane jest przy pomocy funkcji `cv2.GaussianBlur()`. Celem tego etapu jest redukcja szumów i wygładzenie lokalnych przejść tonalnych, co minimalizuje wpływ drobnych zakłóceń w kolejnych krokach przetwarzania.

Rozmycie Gaussowskie jest powszechnie stosowane w przetwarzaniu obrazów jako etap wstępny przed segmentacją lub wykrywaniem krawędzi, ponieważ poprawia spójność danych wejściowych i zwiększa odporność na zakłócenia.

Listing 1: Rozmycie Gaussowskie obrazu

```

1 blur_template_img = cv2.GaussianBlur(template_img_resize, (3, 3),
2                                     0)
3 images['Template_Blurred'] = cv2_gray_to_image_tk(blur_template_img)

```

## 2.3 Progowanie adaptacyjne

Po rozmyciu, obrazy są binarizowane z wykorzystaniem adaptacyjnego progowania średniego. W odróżnieniu od klasycznych metod progowania globalnego (np. Otsu), progowanie adaptacyjne ustala próg indywidualnie dla każdego piksela, w oparciu o średnią wartość jasności jego otoczenia. Implementacja wykorzystuje funkcję `cv2.adaptiveThreshold()`.

Zaletą tej metody jest wysoka odporność na nierównomierne oświetlenie lub lokalne różnice kontrastu, które często występują przy fotografowaniu fizycznych obiektów (jak płytki PCB) bez idealnych warunków laboratoryjnych. W rezultacie uzyskiwane są wyraźniejsze i bardziej stabilne kontury przewodów i padów na płytce.

Listing 2: Progowanie adaptacyjne obrazu

```

1 template_adap_thresh = cv2.adaptiveThreshold(
2     blur_template_img, 255,
3     cv2.ADAPTIVE_THRESH_MEAN_C,
4     cv2.THRESH_BINARY, 15, 5
5 )
6 images['Template_Adaptive_Threshold'] = cv2_gray_to_image_tk(
    template_adap_thresh)

```

## 2.4 Odejmowanie obrazów

Kolejnym krokiem jest odejmowanie progowanych obrazów: testowego od referencyjnego. Operacja ta realizowana jest z użyciem funkcji `cv2.subtract()`, która wykonuje piksel po pikselu różnicę wartości jasności pomiędzy dwoma obrazami.

Celem tej operacji jest wyizolowanie fragmentów, które różnią się pomiędzy obiema wersjami płytka - potencjalnych defektów takich jak: zwarcia, braki ścieżek, dodatkowe elementy czy nieciągłości. Otrzymany obraz różnicowy zawiera jedynie te obszary, w których zaszła zmiana - białe piksele wskazują miejsca, gdzie obraz testowy różni się od referencyjnego.

## 2.5 Filtracja medianowa

Ponieważ odejmowanie może wprowadzać zakłócenia (np. pojedyncze jasne piksele wynikające z niedoskonałości binarnej reprezentacji), obraz różnicowy jest poddany filtracji medianowej. Zastosowano filtr medianowy o jądrze  $5 \times 5$ , który skutecznie eliminuje drobne zakłócenia bez rozmywania rzeczywistych krawędzi.

W przeciwnieństwie do filtrów liniowych, filtr medianowy nie uśrednia wartości, lecz wybiera wartość środkową z otoczenia, co zachowuje kontury oraz struktury geometryczne obiektów. Etap ten poprawia czytelność danych wejściowych do segmentacji.

## 2.6 Segmentacja i zliczanie defektów

W ostatnim etapie przetworzony obraz poddawany jest segmentacji za pomocą funkcji `cv2.findContours()`, która wykrywa granice spójnych obszarów w obrazie binarnym. Każdy wykryty kontur jest analizowany pod względem powierzchni z użyciem funkcji `cv2.contourArea()`.

Aby wyeliminować szum lub nieistotne obiekty, zastosowano filtrację opartą na rozmiarze: akceptowane są jedynie te kontury, których pole powierzchni mieści się w przedziale od 1 do 300 pikseli kwadratowych. Kontury spełniające ten warunek są uznawane za defekty.

Na koniec liczba wykrytych defektów jest zliczana i wyświetlana użytkownikowi w interfejsie graficznym. Obiekty są również wizualizowane jako prostokąty obrysowane na obrazie testowym, co umożliwia ich szybką lokalizację.

Listing 3: Segmentacja i zliczanie defektów na obrazie

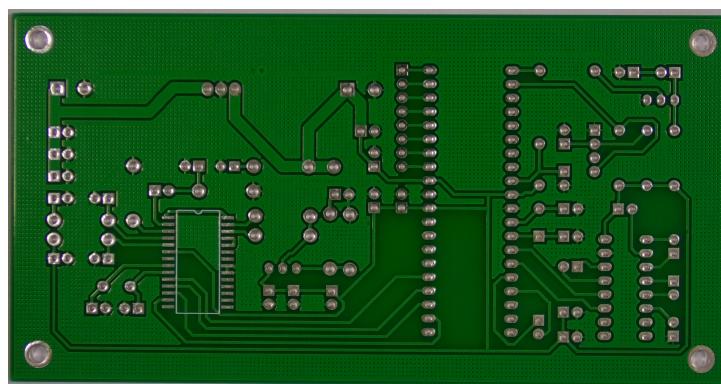
```

1 cnts = cv2.findContours(final_img, cv2.RETR_LIST, cv2.
    CHAIN_APPROX_SIMPLE)[-2]
2 blobs = [cnt for cnt in cnts if 0 < cv2.contourArea(cnt) < 300]
3 defect_count = len(blobs)

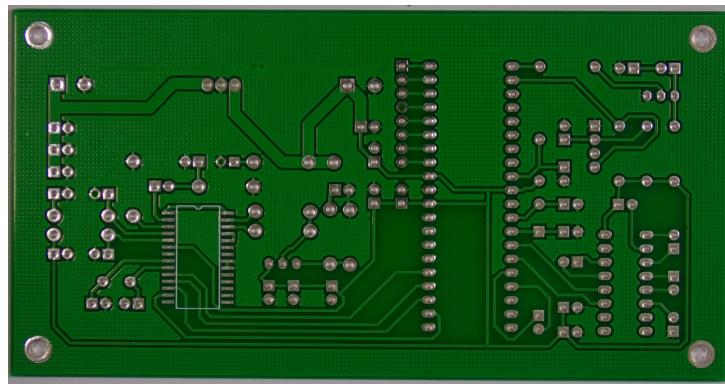
```

## 3. Wyniki i weryfikacja

W trakcie testów z użyciem obrazów zawierających celowe defekty, algorytm konsekwentnie lokalizował błędy w sposób zgodny z oczekiwaniami. Proces był wizualizowany krok po kroku w interfejsie graficznym. Dodatkowo wynik został przedstawiony ilościowo jako liczba wykrytych defektów.



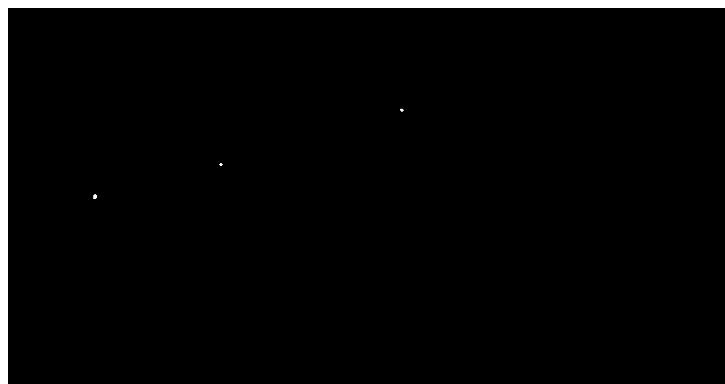
Rysunek 1: Obraz referencyjny RGB



Rysunek 2: Obraz testowy RGB z defektem



Rysunek 3: Obraz różnicowy po odejmowaniu



Rysunek 4: Wynik końcowy po medianie i detekcji konturów

## 4. Wnioski

Zrealizowana metoda, oparta wyłącznie na klasycznym przetwarzaniu obrazów, potwierdziła swoją skuteczność w detekcji usterek płyt PCBA. Otrzymane wyniki są zgodne z rezultatami zaprezentowanymi w pracy naukowej i potwierdzają, że metoda ta może być z powodzeniem wykorzystywana w zadaniach inspekcji wizualnej w przemyśle.

## 5. Bibliografia

1. P. Kumar, M. Panchal, R. Sahu, „Comparative Study of Image Processing and Transfer Learning Techniques for an Automated PCB Fault Detection System”, *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, Volume 9 Issue VI, June 2021.

## Załączniki

Do pracy dołączono plik źródłowy programu o nazwie `main.py`.