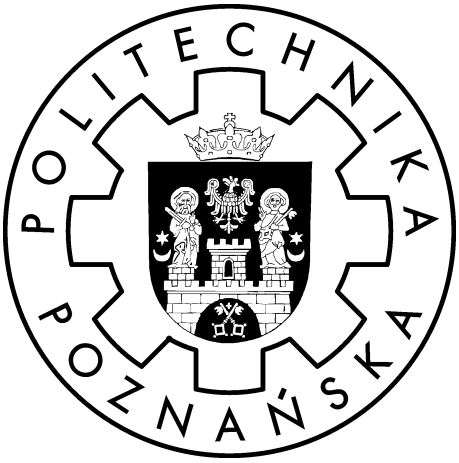
Politechnika Poznańska
Wydział Elektryczny
Instytut Automatyki, Robotyki i Inżynierii Informatycznej



Praca dyplomowa magisterska

BADAWCZY SYSTEM ROZPOZNAWANIA WYKORZYSTUJĄCY OBRAZ TĘCZÓWKI OKA

Jakub Lamprecht

Promotor
dr inż. Tomasz Piaścik

Poznań, 2019

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wprowadzenie

Ogolny opis pracy - wstep do tematyki + motywacja wyboru tematu.

1.1 Cel i zakres pracy

Celem tej pracy było zaprojektowanie i stworzenie aplikacji pozwalającej na przeprowadzenie procesu rozpoznawania tęczówki oka oraz przegląd literatury i implementacja algorytmów normalizacji, enkodowania oraz dopasowania tęczówki. Jednym z wymagań dla tworzonego programu, było aby działał on w dwóch trybach:

- Krokowym - tryb umożliwiający użytkownikowi stopniowe przejście przez proces rozpoznawania z możliwością wyboru metod oraz parametrów dla poszczególnych kroków, a także podglądu wyników dla każdego z nich.
- Wsadowym - tryb umożliwiający automatyczne przeprowadzenie procesu rozpoznawania opisanego przez plik konfiguracyjny wygenerowany w trybie krokowym dla większego zbioru obrazów.

Ważnym aspektem dla tworzonej aplikacji była również użyteczność i przyjazność interfejsu użytkownika, który miał zachęcać do korzystania z aplikacji.

1.2 Struktura pracy

Opis poszczególnych rozdziałów - zrób jak już wymyślisz rozdziały.

Rozdział 2

Podstawy teoretyczne

Rozdział 3

Opis technologii i aplikacji

Częśc wymagań wobec tej pracy dotyczyła nie tylko algorytmów i procesu rozpoznawania tęczówki, ale także sposobu w jaki aplikacja została stworzona i w jaki sposób jest ona użytkowana. Aplikacja miała by przyjazna dla użytkowników i pozwalać na działanie w trybie krokowym. Miała pozwalać również na proste rozszerzanie jej o kolejne metody. Wymagania te w duży sposób wpłynęły na wybór technologii użytych w tej pracy, które opisane zostaną w dalszej części tego rozdziału.

3.1 Użyte technologie

Ponieważ praca jest ściśle związana z przetwarzaniem obrazu, w tym wypadku obrazu oka w celu analizy tęczówki, zdecydowano się na użycie biblioteki OpenCV. Upraszcza ona pracę z obrazem oraz operacje na nim w znacznym stopniu, a także dostarcza wiele gotowych funkcji, które przyspieszają tempo tworzenia eplikacji.

Dużą wagę w pracy należało poświęcić części klienckiej, w celu stworzenia przejrzystego i przyjaznego interfejsu użytkownika. Wiele aplikacji zaniedbuje ten aspekt, w wyniku czego użytkownicy często gubią się w trakcie korzystania z aplikacji. Problem ten najczęściej próbuje się maskować przygotowaniem rozległej dokumentacji aplikacji i jej widoków, co nie poprawia w żaden sposób jakości korzystania z niej. Problemy te prowadzą do zniechęcenia i gorszej jakości doświadczenia użytkownika, co końcowo prowadzi do zaprzestania korzystania z aplikacji. Aby uniknąć tych problemów postanowiono wykorzystać technologie pozwalające na tworzenie zachęcających i interaktywnych interfejsów. Najlepiej w tym obszarze sprawdzają się technologie internetowe takie jak HTML (HyperText Markup Language), CSS (Cascading Style Sheets) oraz JS (JavaScript).

Technologie te nie dają bezpośredniej możliwości użycia w nich bilbioteki OpenCV, nastomiast możliwe jest stworzenie dodatkowej warstwy, która wykonywałaby przetwarzanie obrazu z użyciem OpenCV, a następnie komunikowałaby się z interfejsem i przekazywała do niego wymagane dane. W związku z tym w pracy postanowiono użyć języka Python, który dostarcza wielu rozwiązań umożliwiających taką komunikację.

Zaprojektowanie aplikacji w ten sposób pozwala na rozdzielenie jej na warstwę prezentacyjną oraz warstwę logiki aplikacyjnej, co jest zgodne z dobrymi praktykami tworzenia oprogramowania. Pozwala to również na uniezależnienie tych warstw od siebie, dzięki czemu możliwe byłoby stworzenie wielu interfejsów korzystających z tego samego procesu przetwarzania obrazu.

3.1.1 Warstwa logiki aplikacji

Tak jak wcześniej wspomniano jako główny język programowania odpowiedzialny za warstwę logiki aplikacji, przetwarzanie obrazu oraz komunikację z interfejsem użytkownika wybrany został język Python w parze z biblioteką OpenCV.

Tworzenie aplikacji internetowych z wykorzystaniem języka Python bez żadnej dodatkowej biblioteki byłoby zadaniem żmudnym oraz nie dostarczajacym żadnej dodatkowej wartości. W związaku z tym zdecydowano się na użycie biblioteki **Flask**, która dostarcza gotowych rozwiązań w świecie aplikacji internetowych, upraszcza projetkowanie systemu oraz dostarcza prosty w obsłudze sposób na komunikację z interfejsem użytkownika. Dodatkowo razem z biblioteką dostarczany jest deweloperski serwer WWW, dzięki czemu nie trzeba instalować i konfigurować żadnych innych zależności.

Najpopularniejszym sposobem komunikacji między serwerem a interfejsem użytkownika jest zaprojektowanie interfejsu programistycznego wykorzystującego protokół HTTP w architekturze REST (Representational State Transfer). Jego zadaniem jest zapewnienie aplikacji klienckiej zestawu usług wykonujących ściśle określone operacje zależne od użytej metody protokołu HTTP. REST zakłada, że każda z takich usług jest bezstanowa, a wynik jej działania jest w całości oparty o dane przychodzące wraz z zapytaniem. Każda z usług ma swój unikalny identyfikator w postaci adresu URL (Uniform Resource Locator) pod który aplikacja kliencka może wysyłać zapytania.

Korzystanie z architektury REST wymusza dobre praktyki projektowania interfejsów programistycznych i jest lubianym wzorcem wśród programistów aplikacji internetowych ze względu na przejrzystość tworzonego kodu oraz poniekąd samodokumentujący się kod. W konteście aplikacji tworzonej w ramach niniejszej pracy, zastosowanie architektury REST pozwala na rozbicie całego procesu rozpoznawania tęczówki na mniejsze kroki odpowiadające poszczególnym etapom procesu lub nawet poszczególnym algorytmom. Takie rozbicie można uzyskać przez zdefiniowanie osobnej usługi dla każdego z zaimplementowanych algorytmów. Takie rozwiązanie zapewnia prostą skalowalność, ponieważ dodanie kolejnego algorytmu wymaga jedynie dodania nowej usługi oraz zmiany adresu pod który wysyłane jest zapytanie.

Biorąc pod uwagę te zalety w pracy zdecydowano się na użycie **Flask-RESTful**, które jest rozszerzeniem dla biblioteki Flask. Dostarcza ono szereg metod i uproszczeń w tworzeniu interfejsów w architekturze REST.

Oprócz wyżej wymienionych bibliotek należy zainstalować również wszelkie ich zależności, takie jak przykładowo biblioteka **Numpy** w przypadku OpenCV. Wraz z rozwojem aplikacji ilość użytych bibliotek i zależności będzie tylko rosnąć, co w dłuższym czasie spowoduje problemy z utrzymaniem informacji o tym jakie biblioteki i które ich wersje są wymagane do uruchomienia aplikacji. Często informacje te przechowywane są w postaci pliku tekstowego w katalogu projektu, jednak rozwiązanie to jest mało odporne na błąd ludzki, ponieważ nie trudno zapomnieć o aktualizacji takiego pliku.

Dodatkowo pracując jednocześnie przy kilku projektach w języku Python może okazać się, że kilka z nich wymaga dokładnie tej samej biblioteki, ale w różnych wersjach. Problem ten można rozwiązać używając wirtualnych środowisk. Są to środowiska Pythona, które są odizolowane i niezależne od głównej instalacji Pythona. W rzeczywistości są to osobne foldery, w których instalowana jest odpowiednia wersja Pythona oraz wymagane biblioteki w odpowiednich wersjach.

W celu rozwiązania tych problemów w pracy zdecydowano się na użycie narzędzia **Pipenv**. Umożliwia ono automatyczną generację writualnego środowiska wewnątrz projektu oraz automatyczne śledzenie zależności projektu i ich wersji. Informacje te przechowywane są w dwóch plikach *Pipfile* oraz *Pipfile.lock*, które są generowane, czytane oraz utrzymywane automatycznie. Poniżej przestawiono

podstawowe komendy narzędzia Pipenv:

pipenv install - uruchomienie w katalogu projektu inicjalizuje środowisko wirtualne oraz pliki *Pipfile* i *Pipfile.lock*

pipenv install <packageName> - instaluje paczkę wewnątrz środowiska wirtualnego

pipenv shell - uruchamia powłokę w której dostępny jest python oraz biblioteki zainstalowane w środowisku wirutalnym

3.1.2 Warstwa prezentacyjna

Tak jak wcześniej wspomniano, aplikacja kliencka napisana została z wykorzystaniem technologii internetowych takich jak HTML, CSS oraz JS. Aplikacje korzystające z tych technologii historycznie zwykle używane były wewnątrz przeglądarek. W ostatnich czasach technologie internetowe zyskują coraz większą popularność, co przełożyło się również na rozwój dostępnych bibliotek oraz zwiększenie zasięgu problemów rozwiązywanych tymi technologiami.

Obecnie aplikacje internetowe mogą przyjmować nie tylko postać stron internetowych, ale także zwyczajnych, wieloplatformowych aplikacji desktopowych dzięki takim narzędziom jak użyty w niniejszej pracy **Electron**. Jest to biblioteka, która używa NodeJS oraz Chromium do stworzenia jednolitego środowiska uruchomieniowego. Dzięki takiemu połączeniu możliwe jest wygenerowanie okna aplikacji, które w rzeczywistości jest procesem Chromium przedstawiającym konkretną stronę internetową. Dodatkowo wykorzystanie NodeJS pozwala na dostęp oraz modyfikację systemu plików użytkownika aplikacji, dzięki czemu możliwe przetwarzanie, usuwanie i modyfikowanie plików bez potrzeby przeysłania ich na jakikolwiek serwer. Jest to nieosiągalne przy użyciu przeglądarkowej implementacji języka JavaScript.

Tworzenie aplikacji korzystając z podstawowych możliwości HTML, CSS oraz JS zapewne zajęłoby bardzo dużo czasu, dlatego w świecie technologii internetowych powstaje wiele bibliotek ułatwiających tworzenie skomplikowanych i interaktywnych aplikacji. Jedną z nich jest **React**, który został użyty w tej pracy. Jest to biblioteka JavaScript umożliwiająca tworzenie interfejsów użytkownika. Każdy interfejs można rozłożyć na poszczególne bloki odpowiadające za pewną funkcjonalność np. wyświetlenie obrazu, zebranie danych od użytkownika, czy przełączenie między dwoma widokami. Bloki te nazywa się komponentami i mogą być one wykorzystywane wielokrotnie w różnych miejscach aplikacji. Tworzenie reużywalnych komponentów jest podstawą biblioteki React, co sprawia, że jest ona idealnym kandydatem do zbudowania interfejsu aplikacji niniejszej pracy.

Komponenty powinny być w miarę możliwości od siebie niezależne, a ich implementacja powinna dokonywać jedynie operacji wymaganych właśnie przez ten komponent. Każdy z komponentów cechuje pojęcie stanu - informacji opisujących stan komponentu w danym momencie. W momencie, gdy komponent nie jest wyświetlany, jest on niszczony, a wraz z nim jego stan, co pozwala na zwolnienie zajętej pamięci. Istnieją natomiast informacje, które są ważne dla działania całej aplikacji, z których korzysta wiele różnych komponentów w celu wyświetlenia danych, w związku z czym muszą być przechowywane przez cały cykl życia aplikacji. Dokładnie taki przypadek można zaobserwować w niniejszej pracy, np. informacje o segmentacji obrazu tęczówki potrzebne będą nie tylko w celu wyświetlenia tych danych, ale także w celu wykorzystania ich w kolejnych etapach procesu przetwarzania. Do przechowywania danych poza komponentami użyta została biblioteka **Redux**.

Jednym z wymogów postawionych przed aplikacją jest działanie w dwóch trybach: krokowym oraz wsadowym. Każdy z tych trybów składa się z kilku kroków, np. tryb krokowy składa się z następujących kroków:

- wybór przetwarzanego obrazu,
- przetwarzanie wstępne,
- segmentacja,
- normalizacja,
- kodowanie,
- wybór obrazów do procesu dopasowania,
- dopasowanie,
- prezentacja wyników.

Dla każdego z tych kroków konieczne będzie stworzenie innego widoku, ponieważ każdy z nich będzie wyglądał nieco inaczej. W aplikacji złożonej z tak wielu widoków prosto o problemy związane z przejściami między nimi, które mogą skutkować nieprzewidywalnymi błędami, np. załadowanie widoku normalizacji przed widokiem segmentacji może skutkować błędem krytycznym aplikacji. W celu uniknięcia takich błędów interfejs użytkownika można opisać w postaci automatu skończonego, w którym stanami są poszczególne widoki, a zmiana widoku definiowana jest przez odpowiednie przejścia. Zapewnia to przewidywalność działania systemu i uodparnia go na błedy podobne do tych wspomnianych wcześniej. W celu opisu interfejsu aplikacji za pomocą automatu skończonego wykorzystana została biblioteka **xState**.

Część komponentów, która tworzy aplikację jest bardzo uniwersalna i powszechna w wielu innych zastosowaniach. Są to komponentu typu *przycisk, link, lista elementów, tabela, ikona*. W związku z ich powszechnością powstało wiele bibliotek gotowych komponentów, dzięki czemu tworząc aplikację można skupić się na logice biznesowej stojącej za aplikacją, zamiast na nowo implementować przykładowo działanie przycisku. Większość z tych bibliotek dodatkowo zapewnia podstawowe style, dzięki czemu aplikacje tworzone z ich wykorzystaniem wymagają mniejszego nakładu pracy, by aplikacja wyglądała zachęcająco. Aby zmniejszyć potrzebny nakład pracy do wytworzenia interfejsu użytkownika, w pracy użyta została biblioteka komponentów **Ant Design**.

Podobnie jak w przypadku Pythona, ilość bibliotek i narzędzi używanych w ramach aplikacji jest bardzo duża, a manualne utrzymywanie informacji o nich oraz ich wersjach byłoby niemożliwe. W związku z tym wykorzystany został menadżer pakietów **Yarn** umożliwiający prostą instalację bibliotek i narzędzi, który automatycznie zapisuje zainstalowane paczki oraz ich wersje jako zależności projektu w plikach *package.json* oraz *yarn.lock*. Dzięki korzystaniu z Yarna wszystkie zależności i biblioteki można zainstalować wywołując w folderze zawierającym plik *package.json* komendę `yarn install`.

W ramach pracy wykorzystane zostały również inne narzędzia, do których wyboru przyczyniła się wyłącznie osobista preferencja:

Babel - kompiler JavaScript, pozwalający na korzystanie z najnowszych oraz testowych funkcjonalności JavaScript przez transformację kodu źródłowego w nowszej wersji do kodu wynikowego w starszej wersji.

Styled Components - biblioteka pozwalająca na definiowanie styli CSS dla poszczególnych komponentów w plikach źródłowych o rozszerzeniu **.js*.

Webpack - narzędzie uruchamiające kompilatory kodu źródłowego przekształcające go do postaci rozumianej przez przeglądarki oraz łączące wiele plików w pojedyńczy plik wynikowy.

Axios - biblioteka ułatwiająca wysyłanie zapytań do serwera.

3.1.3 Inne technologie

Podczas pracy nad projektem używany był system kontroli wersji Git. W trakcie programowania często zachodzi potrzeba zmiany istniejącego i działającego kodu w celu dodania nowych funkcjonalności. Niezapisanie poprzedniej wersji kodu może skutkować niemożliwością powrotu do wersji przed wprowadzeniem zmian, tym samym generując dodatkową pracę wymaganą do naprawienia powstałych problemów. System kontroli wersji Git pozwala na zapisanie aktualnego stanu pracy w dowolnym momencie. Dzięki temu wprowadzanie zmian w działającym kodzie jest znacznie mniej ryzykowne, ze względu na możliwość powrotu do dowolnej poprzedniej wersji. Korzystając z tego systemu kontroli wersji uzyskujemy również dostęp do historii poszczególnych plików, a także możliwość dodawania komentarzy podczas zapisywania nowych wersji, dzięki czemu łatwiej zrozumieć co dane zmiany robią i dlaczego zostały wprowadzone.

Dodatkowym atutem jest możliwość przechowywania całości projektu w zewnętrznym serwisie. Dzięki temu w razie awarii sprzętu, na którym tworzony jest projekt, w najgorszym wypadku stracona zostanie jedynie część pracy, a nie jej całość.

3.2 Struktura kodu

Struktura projektu podzielona została na trzy główne foldery:

- **/app** - zawiera kod źródłowy oraz konfiguracje narzędzi użytych przez aplikację kliencką,
- **/sever** - zawiera kod źródłowy oraz konfigurację narzędzi użytych przez aplikację serwerową dokonującej przetwarzania tęczówki,
- **/thesis** - zawiera kod źródłowy pracy dyplomowej w formacie **.tex*

Używając takiego podziału, łatwo odnaleźć miejsce odpowiadające za szukaną funkcjonalność. W głównym folderze projektu znajduje się również plik *README.md*, w którym opisane zostały instrukcje dotyczące uruchomienia projektu.

3.2.1 Struktura projektu aplikacji klienckiej

- **/dist** - zawiera kod wynikowy wygenerowany przez Webpack,
- **/src/js** - zawiera kod źródłowy napisany w JavaScript,
  - **actions** - folder dla akcji używanych przez bibliotekę Redux,
  - **api** - folder zawierający funkcje pomocnicze wysyłające zapytania do serwera o odpowiednie algorytmy przetwarzania tęczówki,
  - **components** - folder zawierający komponenty uniwersalne dla całej aplikacji,
  - **constants** - folder zawierający pliki eksportujący stałe używane przez resztę kodu,

  - **helpers** - folder zawierający funkcje pomocnicze,
  - **reducers** - folder w którym definiowane są konsekwencje akcji (Redux)
  - **stateMachine** - folder w którym znajduje się opis interfejsu użytkownika za pomocą maszyny stanów (xState)
  - **store** - folder definiujący scentralizowany stan aplikacji (Redux)
  - **views** - folder zawierający implementacje poszczególnych widoków dla obu trybów działania aplikacji

- **/webpack** - zawiera pliki konfiguracyjne Webpacka opisujące sposób przetwarzania kodu źródłowego.

3.2.2 Struktura projektu aplikacji serwerowej

- **/.venv** - folder zawierający wirtualne środowisko projektu wraz z potrzebnymi bibliotekami,
- **/const** - folder zawierający stałe używane przez resztę kodu,
- **/processing** - folder zawierający implementacje algorytmów dla poszczególnych etapów procesu,
- **/resources** - folder zawierający kod definiujący strukturę stworzonego API, kod eksponujący na zewnątrz algorytmy zaimplementowane w folderze *processing*,
- **/temp** - folder tymczasowy zawierający wyniki etapów procesu w postaci zdjęć,
- **/utils** - folder zawierający funkcje pomocnicze.

3.3 Działanie aplikacji

W tej części rozdziału przedstawiony zostanie opis interfejsu użytkownika za pomocą maszyny stanów, a także opisane zostaną poszczególne widoki aplikacji oraz jej możliwości.

3.3.1 Opis widoków aplikacji za pomocą grafu

3.3.2 Opis interfejsu aplikacji

Wszystkie zrzuty ekranu aplikacji, do których odwołuje się poniższy opis można znaleźć w dodatku A na stronach 20 - **??**.

Po uruchomieniu aplikacji oczom użytkownika ukazuje się ekran startowy A.1 witający użytkownika i proszący go o wybranie jednego z dwóch trybów działania aplikacji.

Wybranie trybu krokowego przenosi użytkownika do kreatora, który przeprowadzi go przez proces rozpoznawania tęczówki. Wszystkie kroki kreatora widoczne są w górnej części widoku. Aktywny krok jest zawsze oznaczony kolorem niebieskim. Pierwszym krokiem kreatora jest wybranie obrazu, który chcemy poddać przetwarzaniu (rysunek A.2). W celu wyboru użytkownik może kliknąć w wyświetlone pole lub przeciągnąć na nie wybrany obraz. Po lewo od kreatora znajduje się rozwijane menu, które pozwala użytkonikowi na przełączenie się w inny tryb lub przejście do ekranu domowego w każdym momencie działania aplikacji. Przejście pomiędzy trybami jest równoznaczne z wykasowaniem dotychczasowo wybranych informacji w poprzednio aktywnym trybie.

Kliknięcie przycisku “Next” spowoduje przejście do kolejnego kroku kreatora Jeżeli użytkownik nie wybrał obrazu, kreator nie przejdzie do następnego kroku i poinformuje użytkownika o konieczności wyboru obrazu.

Następnym krokiem kreatora jest przetwarzanie wstępne (rysunek A.4). Widok ten przedstawia użytkownikowi obecny stan wybranego obrazu po lewej stronie oraz formularz pozwalający na kontrolowanie metody przetwarzania i jej paramterów po prawej stronie. Pod obrazem dostępny jest przycisk “Save", którego kliknięcie poprosi użytkownika o wybranie miejsca, w którym wyświetlony obraz ma zostać zapisany. Kliknięcie przycisku “Process” znajdującego się pod formularzem skutkuje zaaplikowaniem wybranego algorytmu z wybranymi parametrami. Jeżeli przycisk “Process” został wciśnięty chociaż jeden raz, to pojawi się przycisk “Revert last”, którego kliknięcie powoduje cofnięcie się do poprzedniego stanu obrazu. Zmiana wybranego algorytmu skutkuje również zmianą dostępnych do modyfikacji paramterów (rysunek A.3).

Na samym dole widoku dostępne są dwa przyciski. Przycisk “Next”, podobnie jak poprzednio, powoduje przejście do następnego kroku kreatora, a przycisk “Previous” do poprzedniego kroku kreatora. Przejście do poprzedniego kroku kasuje wszelkie zapisane informacje związane z aktualnym krokiem. W tym kroku kreatora możliwe jest przejście do kolejnego kroku nie uruchamiając żadnego z algorytmów przetwarzania wstępnego.

Kolejne kroki kreatora funkcjonują w bardzo podobny sposób. Różnicą może być ilość wyświetlanych obrazów. Jeżeli wymagane jest wyświetlenie informacji o kilku obrazach, to prezentowane są one w formie zakładek. Sytuację taką można zaobserwować w widoku procesu normalizacji (rysunek A.5), gdzie możliwy jest podgląd przed i po normalizacji oryginalnego obrazu, maski oraz maski nałożonej na zdjęcie.

Następnie przechodzimy do kodowania, którego widok ma taką samą strukturę jak widoki przetwarzania wstępnego czy normalizacji, ale pozwala na podgląd innych obrazów oraz na wybór innych metod.

Kolejnym krokiem po kodowaniu jest wybór obrazów, do których chcemy spróbować dopasować obraz wybrany w kroku pierwszym (rysunek A.6). W celu wyboru obrazu należy użyć przycisku znajdującego się w górnej częsci ekranu. Możliwe jest wybranie wielu obrazów na raz.

Po zatwierdzeniu obrazów użytkownik zostaje przeniesiony do widoku dopasowania, gdzie ponownie może wybrać paramtery i metodę dla danego etapu procesu. Kliknięcie przycisku “Next” w tym widoku uruchamia proces, na który składa się przetworzenie wszystkich wybranych obrazów w dokładnie ten sam sposób, co obraz wybrany w kroku pierwszym oraz przeprowadzenie operacji dopasowania. Ponieważ przetwarzanie kilku obrazów może potrwać pewien czas, użytkownik przenoszony jest do nowego ekranu (rysunek A.7), gdzie wyświetlana jest mu informacja o obecnym stanie aplikacji wraz z animowaną ikoną, która zapewnia, że aplikacja nie zacięła się.

Rozdział 4

Zaimplementowane algorytmy

Zgodnie z założeniami w niniejszej pracy zaimplementowane zostały rozwiązania dla następujących etapów procesu rozpoznawania tęczówki:

- przetwarzanie wstępne,
- normalizacja,
- kodowanie,
- dopasowanie.

Algorytmy segmentacji tęczówki oraz usuwania zakłóceń zostały pominięte, ponieważ były przedmiotem innej pracy dyplomowej. W dalszej części rozdziału opisane zostaną zaimplementowane algorytmy dla wyżej wymienionych etapów.

4.1 Przetwarzanie wstępne

Odpowiednie przygotowanie obrazu przed rozpoczęciem procesu rozpozanawania jest bardzo ważne. Pozwala ono na polepszenie jakości zdjęcia przez usunięcie zakłóceń czy poprawienie kontrastu, co znacznie wpływa na jakość działania całego systemu.

Różne metody segmentacji wymagają odmiennego wstępnego przygotowania obrazu. Często proces przygotowania obrazu różni się nawet dla poszczególnych etapów procesu segmentacji, tzn. innych operacji może wymagać obraz w trakcie szukania parametrów źrenicy, a jeszcze innego w trakcie szukania parametrów tęczówki. Z tego względu przetwarzanie wstępne może byc dość skomplikowane i składac się z wielu różnorodnych kroków.

Ponieważ algorytmy segmentacji często polegają na konkretnych operacjach przygotowawczych, których może być wiele, umożliwienie użytkownikowi ich parametryzacji bardzo skomplikowałoby interfejs aplikacji i znacznie pogorszyło by jego czytelność, jednocześnie dając użytkownikowi niewiele pola do manipulacji tą częścią procesu.

Ze względu na zależność procesu przygotowania obrazu od wybranego algorytmu segmentacji, implementacja przetwarzania wstępnego przeniesiona została do implementacji poszczególnych metod segmentacji, a użytkownikowi udostępnione zostały wyłącznie uniwersalne algorytmy takie jak:

- filtr Gaussa
- filtr medianowy

- normalizacja histogramu
- filter2D

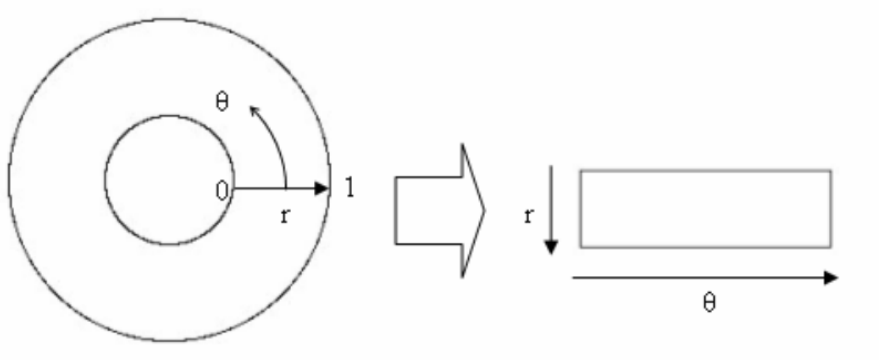
Implementacje tych alogrytmów wykorzystują podstawowe funkcje z biblioteki OpenCV takie jak: `GaussianBlur`, `medianBlur`, `equalizeHist` oraz `filter2D`.

4.2 Normalizacja

Porównywanie tęczówki może być problematyczne, ze względu na potencjalne niezgodności w wymiarach obrazu. Najczęstszą przyczyną tych niespójności jest zmiana rozmiaru tęczówki będąca wynikiem rozszerzania lub kurczenia się źrenicy oka w zależności od intensywności oświetlenia w otoczeniu. Niespójności te mogą wynikać również z innych źródeł np. różna odległość w trakcie pobierania zdjęcia, różnice w obrocie kamery, inna rotacja oka, czy nawet nachylenie głowy w trakcie pobierania zdjęcia [1].

W celu umożliwienia porównania dwóch obrazów, z których każde mogło być zrobione w odmiennych warunkach, obraz poddaje się procesowi normalizacji, który ma za zadanie uspójnić rozmiary tęczówki w systemie.
Ważną informacją dla procesu normalizacji jest także to, że środek źrenicy i tęczówki nie muszą znajdować się w tym samym punkcie, co musi być wzięte pod uwagę przy opracowywaniu rozwiązania.

W aplikacji zdecydowano się na użycie normalizacji metodą Daugmana [1], która polega na przekształceniu obrazu tęczówki z postaci pierścienia do postaci prostokąta przez zmianę układu współrzędnych, w którym prezentowane są wartości poszczególnych punktów tęczówki. Zmiana ta polega na przejściu z opisu wartości pikseli w kartezjańskim układzie współrzędnych (x, y), do opisu za pomocą pary współrzędnych biegunowych (r, θ), gdzie r jest wartością z przedziału $[0, 1]$, a θ jest kątem z przedziału $[0, 2\pi]$.



Rysunek 4.1: Model normalizacji Daugmana [3]

Wybrany rozmiar prostokąta decyduje o ilości informacji znajdujących się w znormalizowanym obrazie. Wiersze w tak uzyskanym prostokącie można interpretować jako kolejne okręgi na obrazie tęczówki. Jak widać na powyższym rysunku 4.1, wybrana wysokość prostokąta decyduje o ilości okręgów - rozdzielczość promieniowa , a szerokość prostokąta decyduje o rozdzielczości kątowej.

Przejście ze współrzędnych kartezjańskich (x,y) do współrzędnych biegunowych (r,θ) opisane zostało przez Daugmana [1] równaniami 4.1:

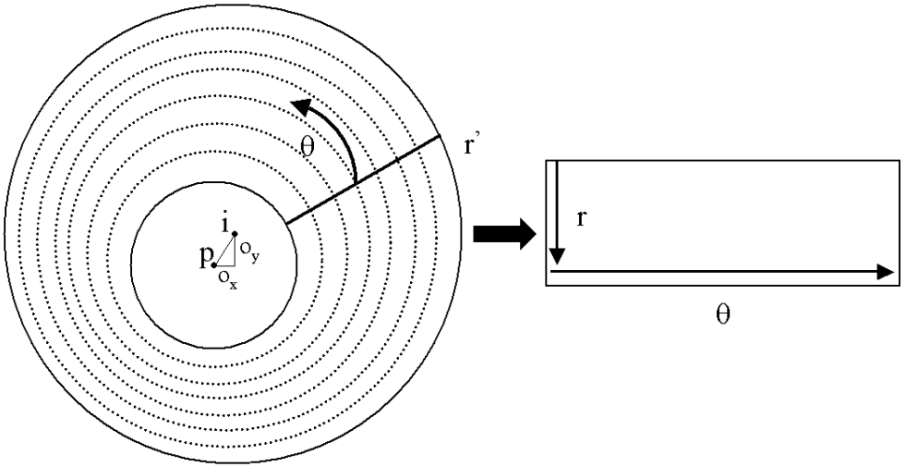
$$I(x(r,\theta),y(r,\theta)) \rightarrow I(r,\theta),$$
$$x(r,\theta) = (1-r)x_p(\theta) + r\,x_s(\theta), \tag{4.1}$$
$$y(r,\theta) = (1-r)y_p(\theta) + r\,y_s(\theta),$$

gdzie:

x_p, y_p - współrzędne na okręgu granicznym źrenicy wzdłuż kierunku θ,

x_s, y_s - współrzędne na okręgu granicznym tęczówki wzdłuż kierunku θ.

Tak jak wcześniej wspomniano, środki źrenicy i tęczówki nie zawsze znajdują się na jednej osi. Z tego względu wyznaczając kolejne punkty znormalizowanego obrazu należy uwzględnić różną odległość między granicą źrenicy a tęczówki w zależności od rozpatrywanego kąta θ, co dobrze zobrazowane zostało na rysunku 4.2. Niezależnie od tych różnic, wzdłuż każdego kierunku θ wybierana jest stała liczba punktów w taki sposób, aby otrzymany obraz był prostokątem o stałych wymiarach.



Rysunek 4.2: Model normalizacji Daugmana z uwzględnieniem niewspółosiowości tęczówki i źrenicy [5]

Uwzględnienie tej niewspółosiowości polega na wyznaczeniu wartości odległości między granicami źrenicy i tęczówki dla każdego rozpatrywanego kąta [3] zgodnie z równaniami 4.2.

$$r' = \sqrt{\alpha}\beta \pm \sqrt{\alpha\beta^2 - \alpha - r_I^2},$$
$$\alpha = o_x^2 + o_y^2, \tag{4.2}$$
$$\beta = \cos\left(\pi - \arctan\left(\frac{o_y}{o_x}\right) - \theta\right).$$

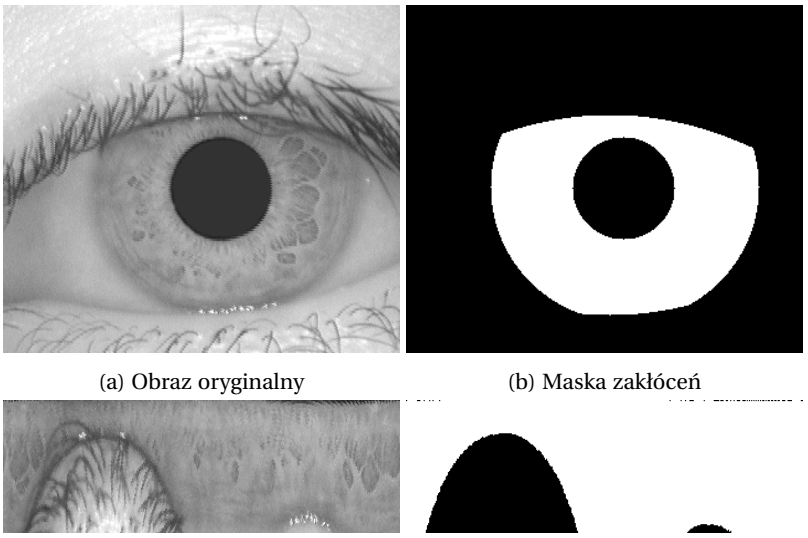
gdzie:

o_x, o_y - różnica położeń między środkiem źrenicy i tęczówki,

r' - odległość między granicą źrenicy, a granicą tęczówki pod kątem θ,

r_I - promień tęczówki

Jeżeli w procesie segmentacji wygenerowana została dodatkowo maska zawierająca informacje o zakłoceniach znajdujących się w obrębie tęczówki, takich jak powieki, czy rzęsy, wygenerowaną maskę również należy poddać procesowi normalizacji, aby umożliwić korzystanie z niej podczas porównywania tęczówek. Przykład procesu normalizacji przedstawiony został na rysunku 4.3.



(a) Obraz oryginalny (b) Maska zakłóceń

(c) Tęczówka po normalizacji (d) Maska zakłóceń po normalizacji

Rysunek 4.3: Przykład normalizacji metodą Daugmana

Takie rozwiązanie zapewnia odporność na rozszerzanie i kurczenie się źrenicy, niewspółosiowość okręgów tęczówki i źrenicy a także różnicę położenia ich między różnymi obrazami. Metoda ta nie kompensuje jednak różnic rotacji tęczówki. Kompensacja ta jest natomiast zapewniana w procesie dopasowywania obrazów, który opisany został w późniejszej części tego rozdziału.

4.3 Kodowanie

Kodowanie tęczówki można rozumieć jako opis jej cech. Dobry algorytm ekstrakcji cech sprawi, że wynik miary dopasowania dla obrazów tej samej tęczówki będzie znajdował się w innym przedziale niż podczas porównywania obrazów dwóch różnych tęczówek. Dzięki temu w końcowym etapie procesu rozpoznawania możliwe jest podjęcie decyzji o rozpoznaniu bądź nierozpoznaniu tęczówki.

Aby zapewnić dobrą jakość identyfikacji tęczówki, proces kodowania powinien wyciągać z obrazu tylko najważniejsze i najbardziej rozróżnialne jego cechy. Oppenheim i Lim [4] pokazali w swojej pracy, że najważniejsze cechy obrazu niesie ze sobą widmo fazowe. Widmo amplitudowe zawiera informacje o mniej indywidualnych cechach, a także jest zależne od czynników zewnętrznych takich jak kontrast czy oświetlenie. Z tego względu podczas kodowania tęczówki wykorzystane powinno być właśnie widmo fazowe obrazu. W celu jego uzyskania należy przenieść znormalizowany obraz z dziedziny przestrzennej do dziedziny częstotliwości.

Daugman [1] w swojej pracy zaproponował użycie w tym celu filtrów Gabora, dzięki którym można uzyskać połączoną reprezentację obrazu w przestrzeni oraz częstotliwości. Filtry te powstają w wyniku modulacji sinusoidy oraz cosinusoidy za pomocą funkcji Gaussowskiej. Częstotliwość środkowa filtru wyznaczana jest przez częstotliwość sinusoidy, natomiast przepustowość filtru określana jest przez szerokość wykorzystanej funkcji Gaussowskiej.

Jedną z wad filtrów Gabora jest występowanie niezerowej składowej stałej dla przepustowości większej niż jedna oktawa [2]. Zerową składową stałą dla każdej przepustowości można natomiast otrzy-

mać przez zastosowanie filtru, którego charakterystyka częstotliwościowa amplitudowa ma rozkład Gaussowski nie w skali liniowej, a w skali logarytmicznej. Charakterystyka częstotliwościowa amplitudowa takiego filtru jest opisana równaniem 4.3:

$$G(f) = \exp\left(\frac{-\left(\log\left(f/f_0\right)\right)^2}{2\left(\log\left(\sigma/f_0\right)\right)^2}\right), \tag{4.3}$$

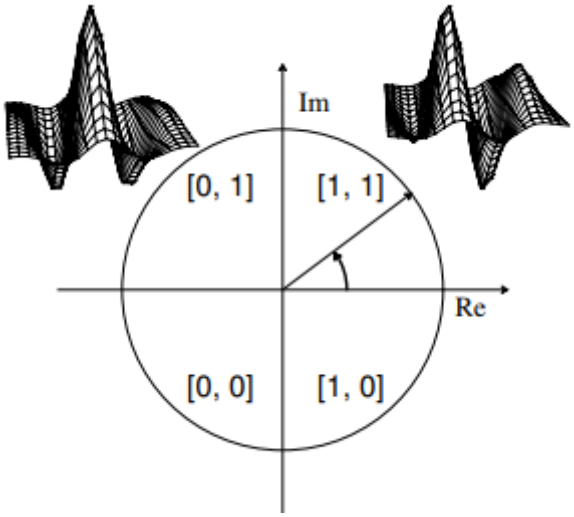
gdzie:

f_0 - częstotliwość środkowa filtra,

σ - przepustowość filtra.

W celu zakodowania cech charakterystycznych znormalizowanego obrazu tęczówki, dla każdego wiersza obrazu obliczany jest jego splot z falkami Log Gabora. Jeżeli moduł wyniku splotu w danym punkcie jest bardzo blisko zera, wówczas informacja fazowa jest nieznacząca, a punkt ten oznaczany jest w masce zakłóceń jako bit nieznaczący.

Wynik splotu obrazu z filtrem jest następnie poddawany kwantyzacji [1] do czterech wartości odpowiadających czterem ćwiartkom płaszczyzny zespolonej przez sprawdzenie znaku części rzeczywistej i urojonej uzyskanego wyniku. Proces kwantyzacji przedstawiony jest na rysunku 4.4. W wyniku kwantyzacji znormalizowany obraz tęczówki przekształcany jest do wzoru tęczówki w postaci ciągu bitów. Przykładowy proces kodowania przedstawiony został na rysunku 4.5

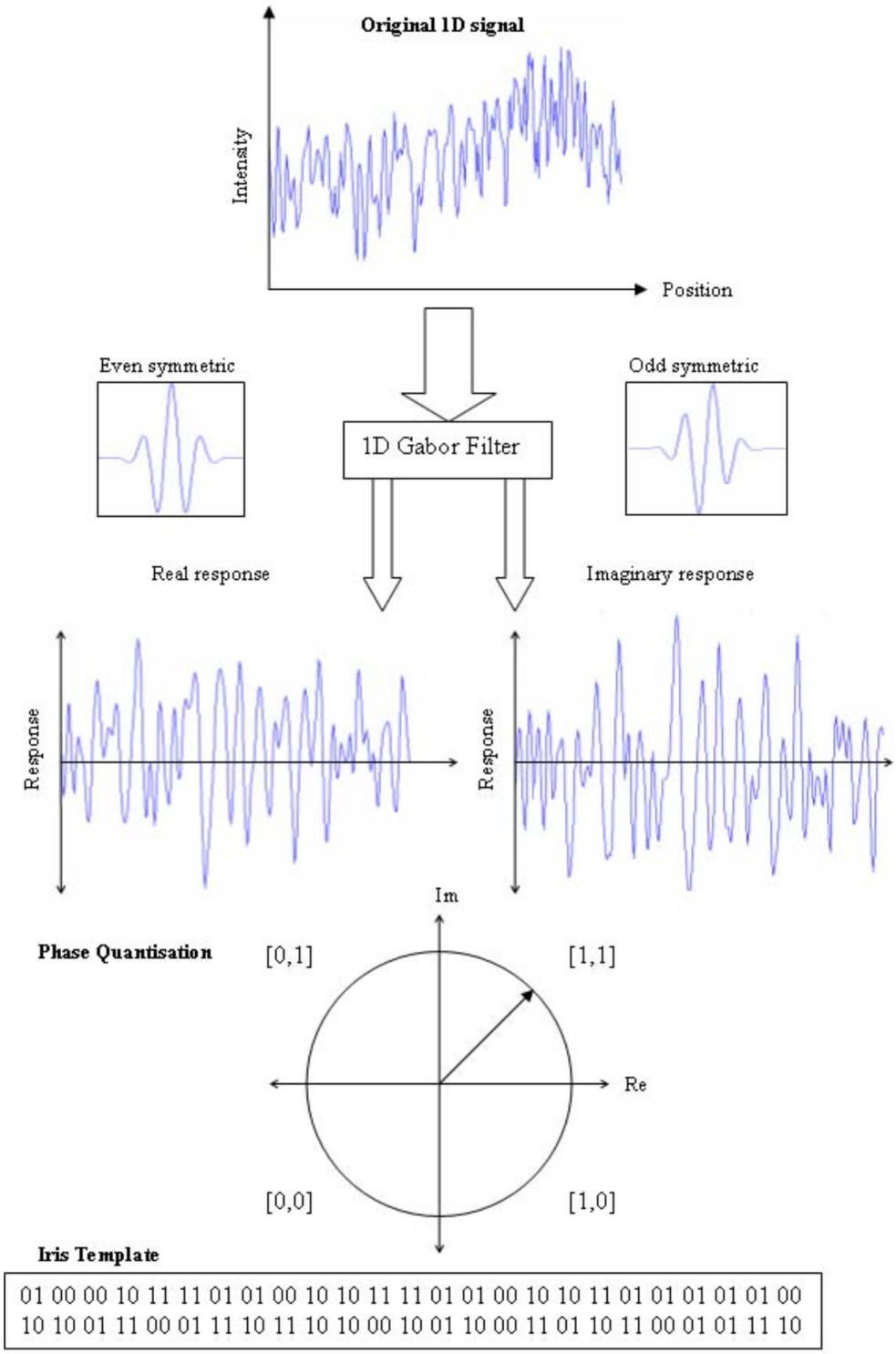


Rysunek 4.4: Ilustracja procesu kwantyzacji

W celu uzyskania większej ilości informacji o tęczówce możliwe jest zastosowanie kilku filtrów Gabora o różnych parametrach. Wówczas obraz poddawany jest splotowi z każdym z takich filtrów, w związku z czym powielana jest liczba bitów kodujących każdy punkt tęczówki.

Z każdym punktem na obrazie tęczówki związana jest pojedyńcza wartość w masce zakłóceń. W wyniku procesu kodowania każdy element obrazu jest reprezentowany przez przynajmniej dwie wartości. W związku z tym należy zaktualizować maskę zakłóceń, aby miała ona ten sam rozmiar co otrzymany wzór tęczówki. W tym celu każdy bit w masce zakłóceń jest powielany tyle razy, ile wartości reprezentuje pojedyńczy punkt we wzorze tęczówki. Przykładowo, jeżeli w procesie kodowania wykorzystany został jeden filtr, transformacja przykładowej maski wyglądałaby następująco:

$$0|1|1|0|0|1 \rightarrow 00|11|11|00|00|11$$



Rysunek 4.5: Ilustracja procesu kodowania

Na rysunku 4.6 przestawiony został przykładowy wynik procesu kodowania zaimplementowanego w tej pracy.

(a) Znormalizowany obraz tęczówki

(b) Znormalizowana maska zakłóceń

(c) Wzór tęczówki obrazu znormalizowanego

(d) Dostosowana maska zakłóceń do nowego rozmiaru

Rysunek 4.6: Wyniki przykładowego kodowania za pomocą filtrów Gabora.

4.4 Dopasowanie

W procesie kodowania obraz tęczówki przekształcony zostaje do wzoru tęczówki w postaci ciągu bitowego, dzięki czemu porównanie dwóch obrazów tęczówek sprowadza się do porównania dwóch ciągów bitowych. Stopień różnicy między dwoma wzorami tęczówki, można określić obliczając odległość Hamminga. Mówi ona o liczbie miejsc, w których dwa słowa bitowe przyjmują różne wartości. Ustalając pewien próg wartości odległości Hamminga można podjąć decyzję, czy dwa dane ciągi reprezentują tę samą tęczówkę.

Wyznaczenie progu dla odległości Hamminga zdefiniowanej w ten sposób zależałoby od długości porównywanych ciągów bitowych. Aby uniezależnić wartość progu od tej długości, miarę można zdefiniować jako sumę niezgodnych bitów podzieloną przez całkowitą liczbę bitów 4.4:

$$HD = \frac{1}{N} \sum_{j=1}^{N} X_j \oplus Y_j, \tag{4.4}$$

gdzie:

X_j, Y_j reprezentują wartości bitów na miejscu j w ciągach odpowiednio X oraz Y,

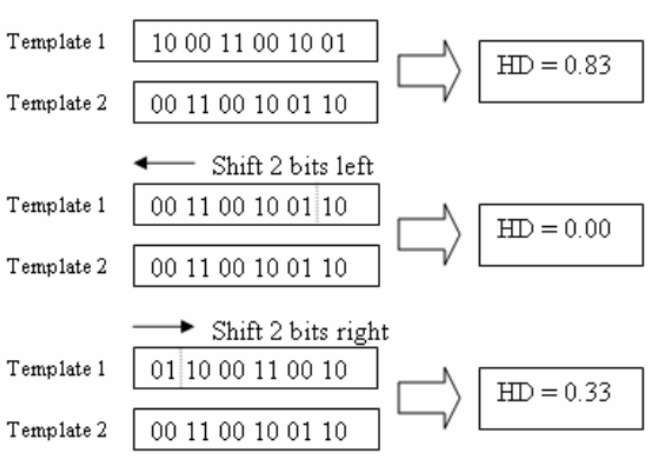
N - długość ciągów X i Y,

$\oplus$ - operator alternatywy rozłącznej.

W trakcie procesu kodowania oprócz wzoru tęczówki tworzony jest także wzór dla maski zakłóceń. Zawiera on informacje o tym, które bity we wzorze tęczówki reprezentują tęczówkę i powinny być uwzględnione, a które bity reprezentują zakłócenia takie jak powieki, czy rzęsy i nie powinny być porównywane w procesie dopasowywania.

Jako że porównywane są dwa obrazy, generowane są również dwie maski zakłóceń i należy uwzględnić każdą z nich. Ponieważ elementy znaczące w masce zakłóceń oznaczone zostały kolorem białym, co odpowiada wartości 1 w postaci bitowej. Ostateczną maskę dla procesu dopasowania można zdefiniować jako iloczyn logiczny tych dwóch masek - w ten sposób stworzony zostanie jeden ciąg bitowy reprezentujący bity znaczące za pomocą wartości 1 oraz bity nieznaczące za pomocą wartości 0. Równanie 4.5 przedstawia definicję odległości Hamminga z uwzględnieniem obu masek zakłóceń:

$$HD = \frac{1}{\sum_{j=1}^{N} \left(Xm_j \cap Ym_j\right)} \sum_{k=1}^{N} (X_k \oplus Y_k) \cap (Xm_k \cap Ym_k), \tag{4.5}$$



Rysunek 4.7: Schemat przedstawiający pojedyńcze przesunięcie wzoru tęczówki. Przykład ten przedstawia wzór do wygenerowania którego wykorzystany został jeden filtr Gabora [3].

gdzie:

X, Y - wzory tęczówki,

Xm, Ym - wzory masek zakłóceń dla wzorów tęczówek X, Y,

N - długość wzorów tęczówek i masek zakłóceń,

$\oplus$ - operator alternatywy rozłącznej,

$\cap$ - operator iloczynu logicznego.

Tak jak wcześniej wspomniano, poprzednie procesy w żaden sposób nie uwzględniały możliwości różnic w rotacji tęczówki na pobranych obrazach. W celu kompensacji tych niespójności podczas procesu dopasowania jeden z wzorów tęczówki poddawany jest przesunięciom bitowym, które odpowiadają rotacji tęczówki o kąt zależny od rozdzielczości kątowej wybranej podczas procesu normalizacji. Maska zakłóceń odpowiadająca temu wzorowi również poddawana jest tym przesunięciom. Jedno przesunięcie w procesie dopasowania odpowiada dwóm przesunięciom bitowym - jednemu w lewo i drugiemu w prawo, z których oba wykonywane są względem oryginalnego wzoru tęczówki. Proces ten zobrazowany został na rysunku 4.7

W zależności od tego ile bitów koduje pojedyńczy punkt siatkówki, tyle bitów zmienia swoje miejsca w trakcie pojedyńczego przesunięcia. Liczba przesuniętych bitów zależy od ilości filtrów Gabora użytych w procesie kodowania, ponieważ każdy z takich filtrów wygeneruje dwa bity reprezentujące pojedyńczy punkt siatkówki.

Odległość Hamminga obliczana jest dla każdego z tych przesunięć. Decyzja o tym, czy dwa wzory reprezentują tę samą tęczówkę podejmowana jest na podstawie najmniejszej uzyskanej wartości, która reprezentuje najlepsze dopasowanie dwóch wzorów. Liczba przesunięć potrzebna do kompensacji niespójności rotacji zależy od tego z jaką dokładnością kątową pobierane są zdjęcia tęczówki.

Rozdział 5

Eksperyment

5.1 Opis eksperymentu

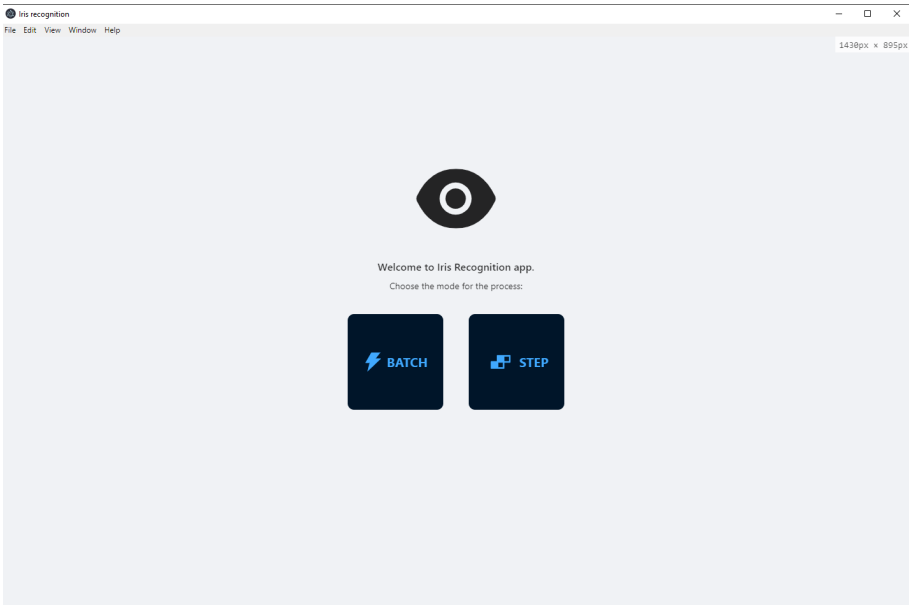
5.2 Wyniki eksperymentu

Rozdział 6

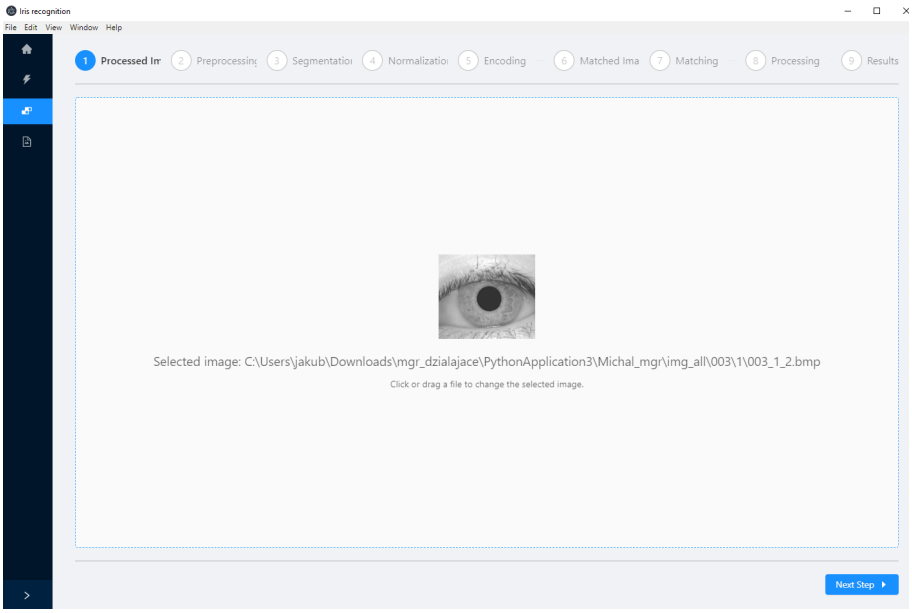
Podsumowanie i kierunki dalszego rozwoju

Dodatek A

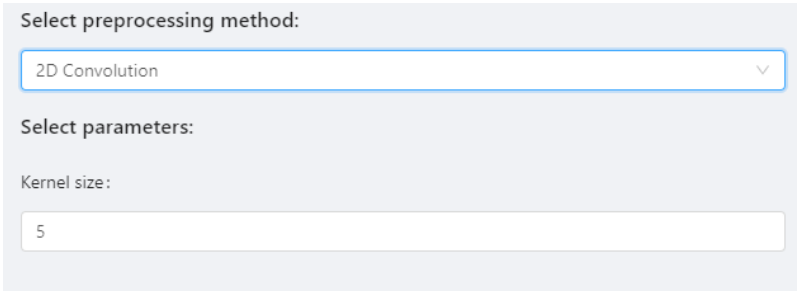
Zrzuty ekranu interfejsu użytkownika



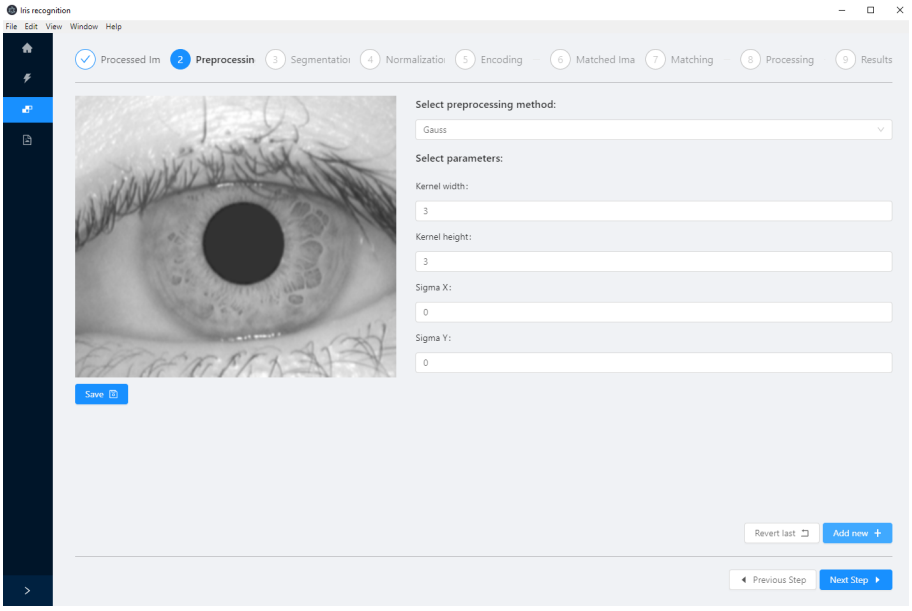
Rysunek A.1: Zrzut ekranu widoku startowego.



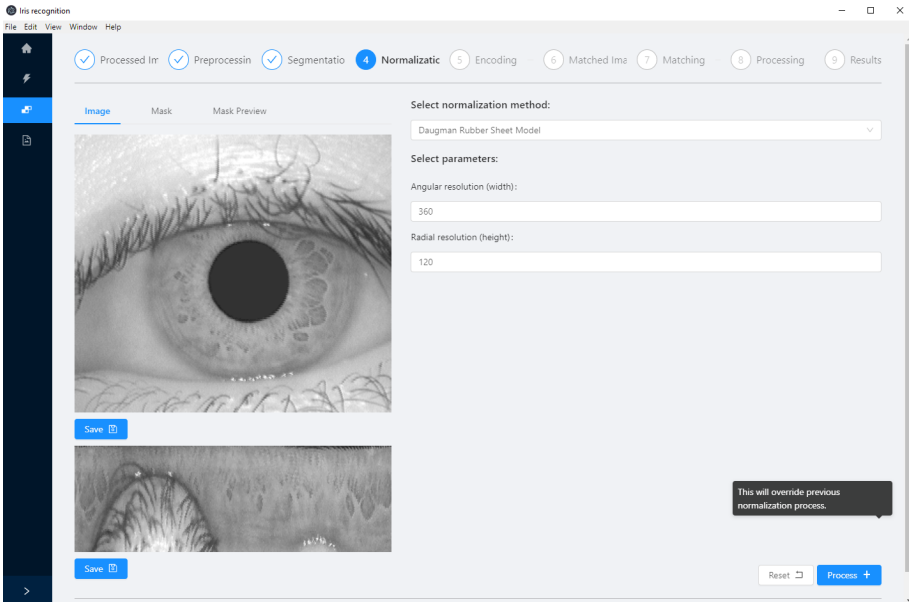
Rysunek A.2: Widok wyboru obrazu poddawanego procesowi.



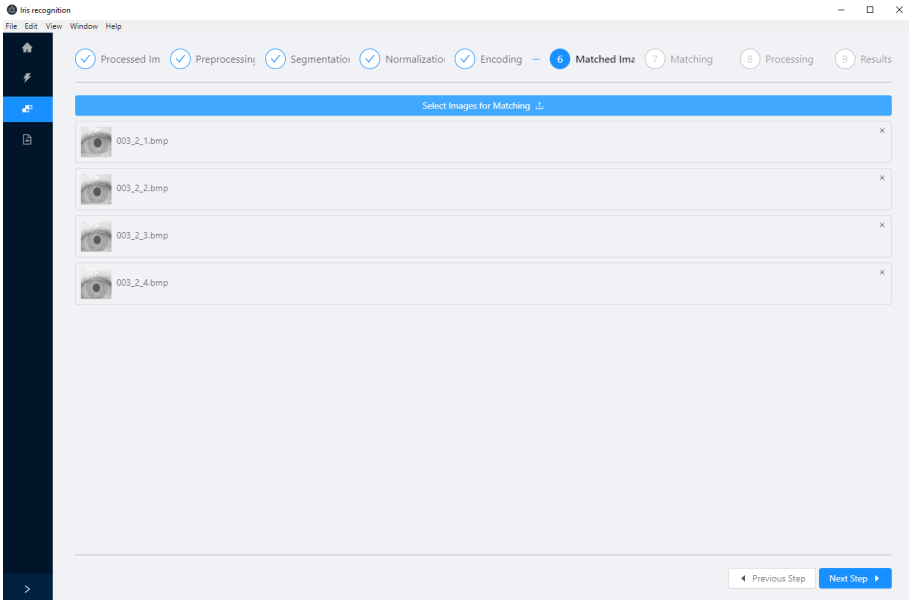
Rysunek A.3: Zmiana dostępnych paramterów po zmianie wybranej metody.



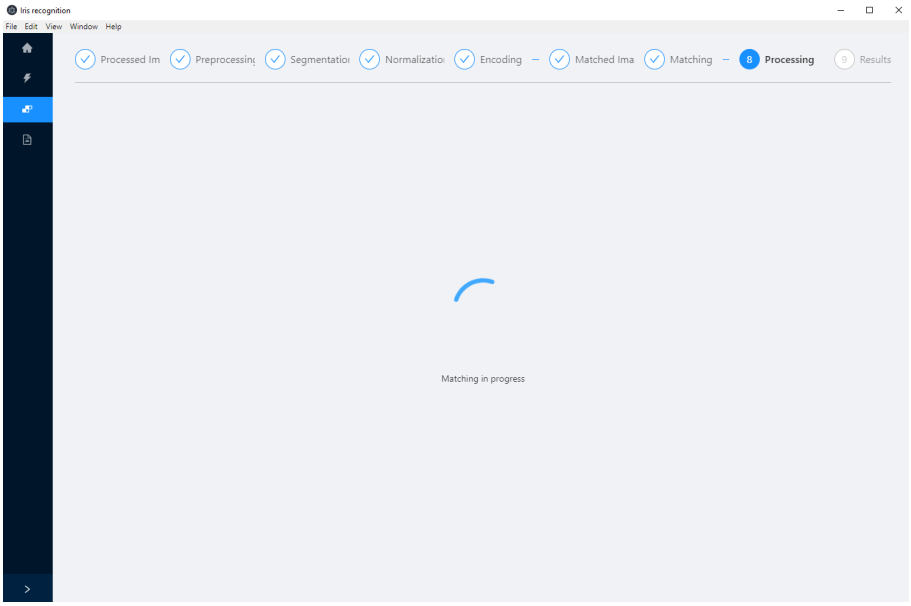
Rysunek A.4: Widok kroku przetwarzania wstępnego.



Rysunek A.5: Zrzut ekranu widoku procesu normalizacji.



Rysunek A.6: Wybór obrazów do których chcemy dopasować przetwarzany obraz.



Rysunek A.7: Widok podczas przetwarzania obrazów oraz dopasowania.

Literatura

[1] J. Daugman. How iris recognition works. *IEEE Trans. Cir. and Sys. for Video Technol.*, 14(1):21–30, January 2004.

[2] David J. Field. Relations between the statistics of natural images and the response properties of cortical cells. *J. Opt. Soc. Am. A*, 4(12):2379–2394, Dec 1987.

[3] Libor Masek. Recognition of human iris patterns for biometric indetification. The University of Western Australia, 2003.

[4] A. V. Oppenheim and J. S. Lim. The importance of phase in signals. *Proceedings of the IEEE*, 69(5):529–541, May 1981.

[5] H. Proenca and L. A. Alexandre. Iris recognition: An analysis of the aliasing problem in the iris normalization stage. In *2006 International Conference on Computational Intelligence and Security*, volume 2, pages 1771–1774, Nov 2006.

Zasoby internetowe

[A] Project Gutenberg.
http://www.gutenberg.net

[B] The AspectJ Project.
http://www.eclipse.org/aspectj/

[C] The Gauntlet (Universal Robustness Corpus).
http://www.michael-maniscalco.com/testset/gauntlet/

[D] Manzini's Large Corpus.
http://www.mfn.unipmn.it/~manzini/lightweight/corpus/

[E] Yuta Mori, Suffix Array Construction Benchmark
http://homepage3.nifty.com/wpage/benchmark/index.html

[F] Michael Maniscalco, The MSufSort Algorithm.
http://www.michael-maniscalco.com/msufsort.htm

[G] Peter Sanders, Skew algorithm.
http://www.mpi-inf.mpg.de/~sanders/programs/suffix/

[H] M. Douglas McIlroy, ssort.c
http://cm.bell-labs.com/cm/who/doug/source.html

[I] Dmitry A. Malyshev, Archon
http://kvgate.com/index.php?root/comp/arch/archon/

[J] Giovanni Manzini, A Lightweight Suffix Array and BWT Construction Algorithm
http://web.unipmn.it/~manzini/lightweight/ds.tgz

[K] Yuta Mori, libdivsufsort project homepage.
http://code.google.com/p/libdivsufsort/

[L] N.Jesper Larsson, qsufsort.c
http://www.larsson.dogma.net/qsufsort.c

[M] Klaus-Bernd Schürmann i Jens Stoye, bpr downloadpage.
http://bibiserv.techfak.uni-bielefeld.de/download/tools/bpr.html

Instytut Automatyki, Robotyki i Inżynierii Informatycznej, Wydział Elektryczny
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ jlamprecht-masterthesis,
    author = "Jakub Lamprecht",
    title = "{Badawczy system rozpoznawania wykorzystujący obraz tęczówki oka}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2019",
}
```