

POLITECHNIKA POZNAŃSKA
WYDZIAŁ ELEKTRYCZNY
Instytut Automatyki, Robotyki i Inżynierii Informatycznej
Zakład Automatyki i Robotyki



PRACA INŻYNIERSKA

***Mechanizmy uczenia maszynowego
we wsparciu autonomii lotu
grupy bezzałogowych statków powietrznych***

Alicja Kuźniewska

Milena Molska

Promotor: dr inż. Wojciech Giernacki

Poznań, 2019

Streszczenie

Celem niniejszej pracy inżynierskiej było wykorzystanie mechanizmów uczenia maszynowego do zorganizowanego i bezpiecznego sterowania grupą bezzałogowych statków powietrznych (UAV). Do celów badawczych wybrane zostały jednostki firmy Parrot - Bebop 2, a środowisko w jakim te symulacje przeprowadzono to Sphinx, wsparcie symulatorem Gazebo pod kontrolą systemu operacyjnego Ubuntu/ROS (Robot Operating System). Zadania szczegółowe obejmowały: autonomiczne lądowanie oparte na głębokim uczeniu ze wzmacnieniem, omijanie przeszkód wykorzystujące przepływ optyczny oraz rozpoznawanie obiektów przez głębokie sieci neuronowe. W tym celu zaimplementowano środowisko z wyuczonym agentem oraz głęboką sieć neuronową SSD Multibox.

Słowa kluczowe: *bezzałogowy statek powietrzny, uczenie maszynowe, autonomiczne lądowanie, omijanie przeszkód, rozpoznawanie obiektów, Bebop 2, ROS, Gazebo, Parrot Sphinx.*

Abstract

The aim of this engineering work was using machine learning mechanisms to organized and safely control a group of unmanned aerial vehicles. For research purposes, Parrot's Bebop 2 unmanned aerial vehicles (UAV) units were selected, and the environment in which these simulations were carried out is the Sphinx, operating under the Gazebo simulator and ROS/Ubuntu system. Issues that have been chosen were an autonomous landing based on deep reinforcement learning and avoiding obstacles using optical flow and the objects recognition by deep neural networks. For this purpose, the environment with the learned agent and deep neural network SSD Multibox were implemented.

Spis treści

| | |
|--|-----------|
| 1 Wprowadzenie | 3 |
| 1.1 Cel pracy i zadania szczegółowe | 3 |
| 1.2 Przygotowanie środowiska symulacyjnego | 4 |
| 1.2.1 Wybrane narzędzia | 4 |
| 1.2.2 Konfiguracja środowiska | 6 |
| 1.2.3 Uruchomienie wielu jednostek UAV | 11 |
| 2 Rozszerzona rzeczywistość we wsparciu prototypowania lotów UAV | 13 |
| 2.1 Wprowadzenie | 13 |
| 2.2 Implementacja oraz uruchomienie | 15 |
| 3 Algorytm omijania przeszkód przy wykorzystaniu głębokiego uczenia | 18 |
| 3.1 Opis wybranej metody | 18 |
| 3.1.1 Single Shot Multibox Detector | 18 |
| 3.1.2 Przepływ optyczny | 23 |
| 3.2 Stworzenie świata symulacyjnego | 24 |
| 3.3 Implementacja algorytmu | 25 |
| 3.4 Zastosowanie algorytmu dla grupy robotów latających | 30 |
| 4 Algorytm autolądowania na platformie | 33 |
| 4.1 Opis wybranej metody | 33 |
| 4.2 Dostosowanie świata symulacji | 35 |
| 4.3 Implementacja algorytmu autolądowania | 36 |
| 4.4 Zastosowanie algorytmu dla grupy robotów latających | 39 |
| 5 Weryfikacja proponowanych rozwiązań w eksperymentach symulacyjnych i rzeczywistym locie UAV | 42 |
| 5.1 Testy rozpoznania obiektów przez sieć neuronową | 42 |
| 5.2 Omijanie przeszkód | 44 |
| 5.2.1 Testy omijania przeszkód w pomieszczeniu z jednym obiektem | 45 |
| 5.2.2 Świat z dwoma przeszkodami | 47 |
| 5.2.3 Świat z trzema obiektami | 49 |

| | | |
|--|--|-----------|
| 5.3 | Śledzenie jednostki wiodącej | 50 |
| 5.4 | Testy skuteczności algorytmu autolądowania | 52 |
| 5.4.1 | Lądownie na platformie statycznej | 54 |
| 5.4.2 | Ruch platformy z prędkością 0,07 m/s | 55 |
| 5.4.3 | Ruch platformy z prędkością 0,1 m/s | 56 |
| 5.4.4 | Ruch platformy z prędkością 0,2 m/s | 57 |
| 5.4.5 | Weryfikacja rozwiązań na rzeczywistym UAV | 58 |
| 5.5 | Unikanie kolizji jednostek UAV | 59 |
| 6 | Zakończenie | 63 |
| 6.1 | Podsumowanie | 63 |
| 6.2 | Potencjalne kierunki rozwoju pracy | 63 |
| Bibliografia | | 65 |
| Spis zawartości załączonej płyty CD | | 70 |

Wprowadzenie

1.1 Cel pracy i zadania szczegółowe

Bezzałogowe statki powietrzne (ang. Unmanned Aerial Vehicles) zyskały ogromną popularność w rozwoju nowoczesnych technologii. Pomimo ich pierwotnego użycia w celach militarnych, znalazły szerokie zastosowanie w otaczającym nas świecie. Niewątpliwie wielką zaletą jest możliwość wykorzystania ich w celach mobilnych tam, gdzie nie jest to możliwe dla robotów naziemnych. Warto również wspomnieć, że zbudowanie w pełni działającego robota wymaga mniejszego nakładu finansowego. Jednak mniejsze koszty niosą za sobą większe wymagania programistyczne, co dla niektórych osób może być wadą, a dla innych wręcz przeciwnie - wyzwaniem. Pomimo wielu pozytywów, roboty latające posiadają również ograniczenia takie jak niewielkie możliwości obliczeniowe lub chociażby mała pojemność akumulatorów, umożliwiająca krótkie loty. Podczas kolizji są podatne na uszkodzenia, dlatego w swojej pracy zdecydowałyśmy się na wykorzystanie niskobudżetowych statków powietrznych Bebop 2 firmy Parrot. Ze względu na ograniczenia, które są związane z użytkowaniem robotów latających, skupiłyśmy się na zadaniach dążących do usprawnienia autonomicznych lotów i nie wymagających dodatkowych nakładów sprzętowych.

Poniżej znajduje się lista, w której przedstawiłyśmy cele szczegółowe do zrealizowania dla tej pracy:

- Przygotowanie symulatora UAV wspartego wirtualną rzeczywistością (zadanie zespołowe).
- Przygotowanie środowiska ROS do zaimplementowania kilku jednostek UAV oraz mechanizmów sterowania ich ruchem (zadanie zespołowe).
- Implementacja algorytmów omijania przeszkód przy wykorzystaniu deep learning (zadanie indywidualne – **Alicja Kuźniewska**).

- Implementacja algorytmów autolądowania przy wykorzystaniu reinforcement learning (zadanie indywidualne – **Milena Molska**).
- Testy opracowanych rozwiązań (zadanie indywidualne – każda z nas wykonała we własnym zakresie).

1.2 Przygotowanie środowiska symulacyjnego

Milena Molska

1.2.1 Wybrane narzędzia

Aby umożliwić wykonanie ustalonych zadań, konieczne było przystosowanie wszelakich narzędzi zarówno programistycznych, symulacyjnych oraz graficznych. W pracy wykorzystałyśmy:

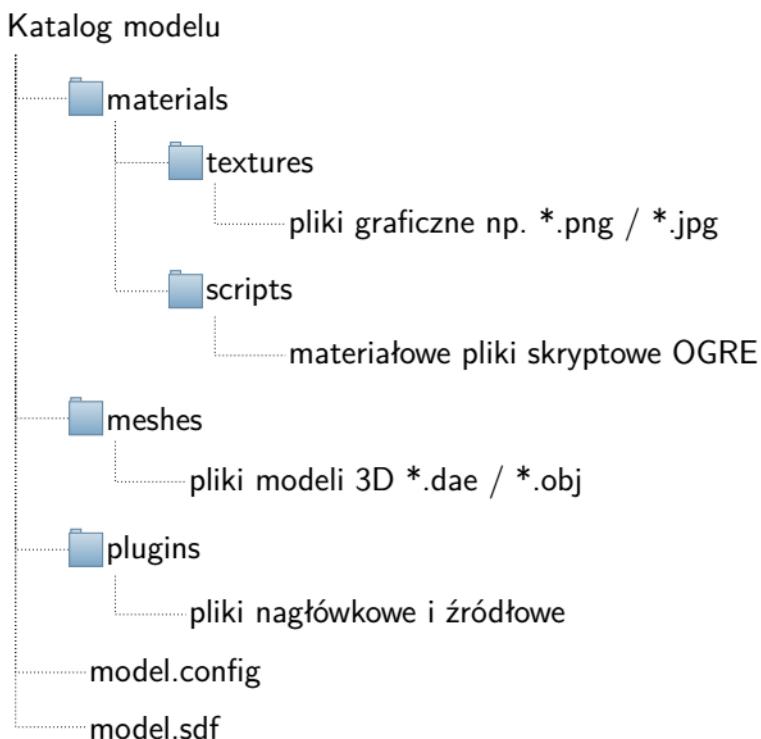
- **Linux Ubuntu 16.04.05 LTS (Xenial Xerus)** – system operacyjny, na którym zostały uruchomione wszystkie elementy projektu. W chwili rozpoczęcia pracy, była to najnowsza, stabilna dystrybucja OS. Oficjalnie dostępna jest wersja Ubuntu 18.04.1 LTS (Bionic Beaver), jednak dla bibliotek takich jak: bebop_autonomy i drl-landing zalecane jest uruchomienie ich pod systemem ROS Kinetic, który jest dostępny tylko dla wersji 16.
- **ROS (Robot Operating System) Kinetic Kame** – platforma programistyczna umożliwiająca tworzenie oprogramowania do sterowania robotów. Wybrana wersja została dostosowana do bibliotek użytych w pracy. Ogólną zasadę działania tego systemu omówiono w rozdziale 1.2.
- **Catkin Tools** – kompilator dedykowany dla platformy ROS, który pozwala na komplikację programów w wielu językach, m.in. C++ oraz Python. Wielką zaletą jest to, że pozwala na dołączenie do projektu bibliotek, bez względu na to w jakim języku zostały napisane.
- **Gazebo 7.0** – środowisko symulacyjne robotów 3D. Dostarcza użytkownikowi wysokiej jakości silnik fizyki, dzięki któremu pozwala na częściowe odzwierciedlenie rzeczywistego zachowania robotów i reakcji otoczenia.
- **Sphinx 0.29.1** – narzędzie pozwalające na symulację jednostek UAV firmy Parrot, w tym Bebop 2. Oparte zostało na środowisku Gazebo w wersji 7.0, w celu wizualizacji otoczenia i zachowania robota latającego. Więcej informacji o tym symulatorze zostało zawartych w rozdziale 1.2.

- **bebop_autonomy 0.7.1** – sterownik działający w ROSie dla jednostek UAV firmy Parrot: Bebop 1 oraz Bebop 2. Więcej informacji odnośnie tego pakietu znajduje się w rozdziale 1.2.
- **Blender v.2.79** – oprogramowanie do tworzenia modeli i animacji 3D.
- **Gimp** – edytor grafiki rastrowej, w którym wykonano część ilustracji zawartych w pracy.
- **SketchUp** – oprogramowanie do tworzenia modeli 3D. Jego istotną zaletą jest możliwość wymiarowania obiektów w jednostkach metrycznych, która została wykorzystana do przeskalowania użytych obiektów.
- **ArUco Library** – biblioteka oparta na OpenCv, służąca do oszacowania położenia kamery za pomocą czarno-białych, kwadratowych znaczników.
- **drl-landing** – pakiet zawierający implementację autonomicznego lądowania opartego na głębokim uczeniu ze wzmacnieniem. Dokładniej został opisany w rozdziale 4.3.
- **object_detection_tensorflow** – biblioteka rozpoznająca obiekty, oparta na głębokich sieciach neuronowych – dokładniej została opisana w rozdziale 3.
- **position_controller** – pakiet sterowania pozycyjnego dla Bebop 2, autorstwa Bartłomieja Kuleckiego. Zawiera dostrojony regulator PID, możliwość zadawania konkretnego punktu docelowego oraz trajektorii lotu.
- **Optical-Flow-based-Obstacle-Avoidance** – biblioteka zawierająca implementację obliczania przepływu optycznego. Dokładniej zostanie ona opisana w rozdziale 3.3.
- **Matplotlib** – biblioteka języka Python, służąca do przygotowywania i przedstawiania danych. Umożliwia szybkie generowanie wykresów 2D oraz 3D.
- **NumPy** – biblioteka języka Python, służąca do wykonywania naukowych obliczeń. Jej wielką zaletą jest możliwość odczytu surowych danych z plików *.csv, co ułatwiło w naszej pracy wizualizację trajektorii.
- **Online ArUco markers generator** – generator znaczników ArUco dostępny online. Umożliwia stworzenie markerów o różnych wymiarach i wyeksportowanie ich do formatu *.pdf, co posłużyło do stworzenia platformy do testów na rzeczywistym UAV.

- **Git** – system kontroli wersji pozwalający na przechowywanie kopii zapasowych utworzonych programów. Oprócz tego wykorzystywany również do pobierania repozytoriów, zawierających biblioteki dla platformy ROS.
- **SimpleScreenRecorder** – program do nagrywania ekranu pod systemem Linux. Został użyty do udokumentowania testów symulacyjnych.
- **Overleaf** – edytor L^AT_EX online, w którym została wykonana część pisemna pracy dyplomowej. Jego ogromną zaletą jest możliwość edycji dokumentu przez wiele osób jednocześnie, co znacznie usprawniło pracę.

1.2.2 Konfiguracja środowiska

Praca dyplomowa została zrealizowana na platformie sprzętowej Bebop 2 firmy Parrot, dlatego kluczowe było znalezienie symulatora, który pozwoli na wykonanie zaplanowanych zadań. Wybór padł na Sphinx, który jest narzędziem stworzonym przez programistów tej samej firmy co jednostka UAV. Uruchomiono je w środowisku symulacyjnym Gazebo w wersji 7.0. Umożliwiło to wizualizację zachowania robota latającego oraz jego otoczenia. Jest to niewątpliwie ogromną zaletą, gdyż użytkownik ma możliwość generowania własnego świata symulacyjnego, opartego na gotowych lub stworzonych przez siebie modelach 3D.



Rysunek 1.1: Struktura katalogów modeli w Gazebo

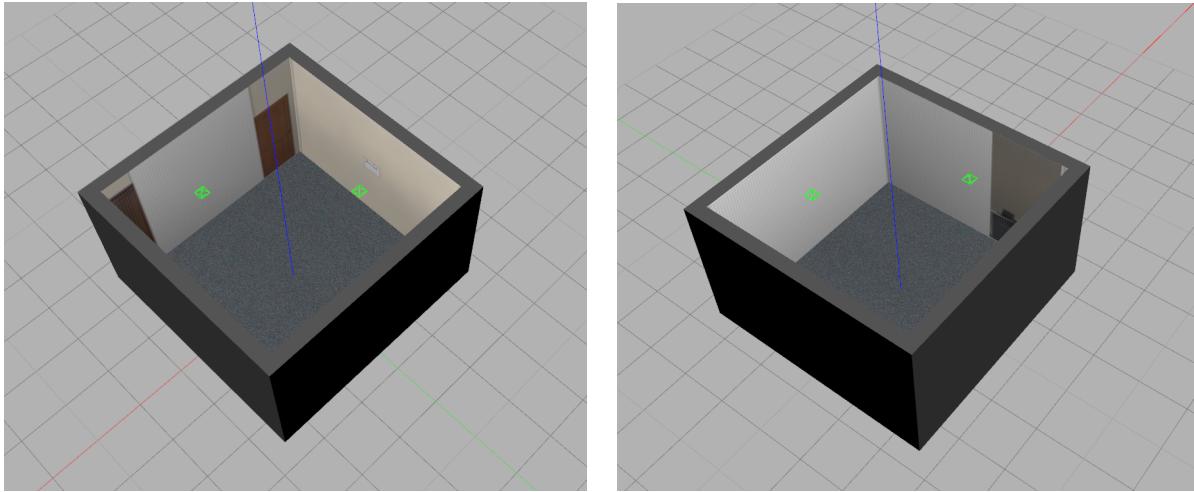
W celu korzystania z własnych obiektów należy dostosować ich format pliku do wymaganego przez Gazebo tj. STL, Collada lub OBJ, gdzie Collada i OBJ są rekomendowane. Następnym krokiem jest stworzenie pliku modelu o rozszerzeniu SDF. Zawiera on podstawowe informacje o przedmiocie takie jak: nazwa, pozycja w wygenerowanym świecie, obszar kolizji i ścieżka do pliku z modelem 3D. W przypadku, gdy chcemy odwzorować jego fizyczne zachowanie, możliwe jest również dodanie masy i inercji przedmiotu. Bardziej szczegółowe informacje odnośnie plików SDF znajdują się na stronie internetowej [1]. Do katalogu modelu, opcjonalne jest dodanie pliku *.conf, zawierającego meta-dane dotyczące obiektu. Standardową strukturę katalogów modeli w Gazebo przedstawiono na rysunku 1.1.

W niniejszej pracy inżynierskiej stworzyłyśmy model pomieszczenia, który miał być odwzorowaniem powstającego w tym samym czasie AeroLab w sali nr 109 budynku Wydziału Elektrycznego Politechniki Poznańskiej. W celu zachowania odpowiednich proporcji, za pomocą narzędzia on-line [2] wygenerowałyśmy uproszczony plan pokoju o wymiarach 4,55x4,62 m. Następnie w programie Blender wykonałyśmy model 3D pomieszczenia o odpowiednich proporcjach, na który naniosłyśmy tekstury wykonane w edytorze graficznym Gimp. Kolejnym krokiem było przeskalowanie modelu w programie SketchUp i wyeksportowanie go do formatu *.dae. Następnie napisałyśmy plik *.config oraz *.sdf, stworzyłyśmy strukturę katalogu według wcześniej omówionego standardu i przeniosłyśmy folder obiektu do domyślnego katalogu modeli Gazebo: *./gazebo/models*. Dzięki temu uzyskałyśmy możliwość odwoływanego się do modelu poprzez jego nazwę i domyślny folder simulatatora, co w dalszym programowaniu zapewniło uniwersalność ścieżki pliku.

Po skonfigurowaniu modelu pomieszczenia, kolejnym krokiem było napisanie pliku opisującego generowany świat. Oprogramowanie Gazebo wykorzystuje do tego format *.world. Silnik fizyki świata został przez nas dodany na podstawie wytycznych z dokumentacji Sphinx [3]. Zawierał on:

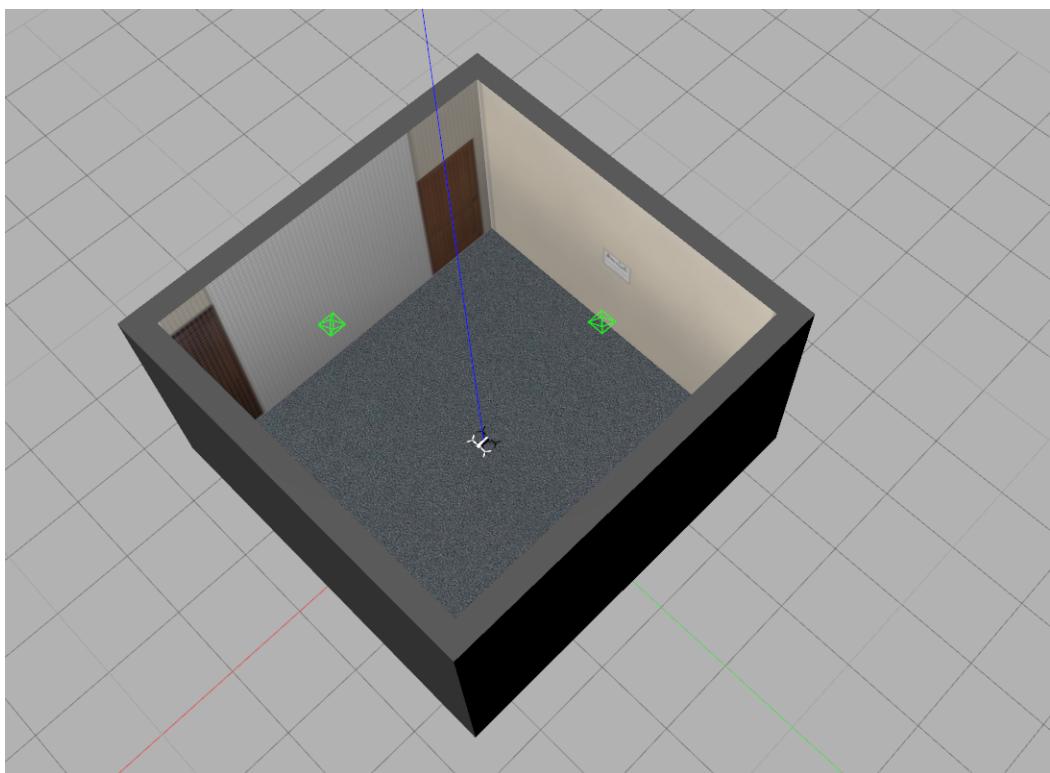
- wartości siły grawitacji w osiach X, Y, Z,
- wartości pola magnetycznego w osiach X, Y, Z,
- wartość maksymalnego odstępu czasu w jakim robot i środowisko mogą wzajemnie na siebie oddziaływać,
- stosunek czasu symulacji do czasu rzeczywistego,
- maksymalną liczbę połączeń między dwoma elementami,
- czas, po jakim zostanie zaktualizowany silnik fizyki.

Zaimplementowałyśmy również model pomieszczenia oraz oświetlenie. Gotowy świat przedstawiony został na rysunku 1.2.



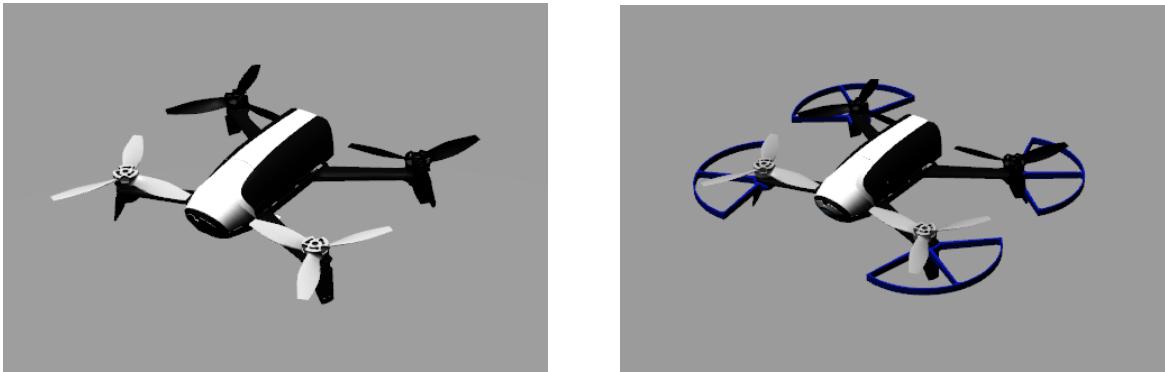
Rysunek 1.2: Świat symulacyjny w Gazebo

Aby możliwe było uruchomienie oprogramowania Sphinx w wygenerowanym przez nas otoczeniu, konieczne było przeniesienie pliku *.world do katalogu symulatora, w którym znajdowały się przykładowe światy: `/opt/parrot-sphinx/usr/share/sphinx/worlds`. Przy pominięciu ostatniego kroku, w dalszych simulacjach jednostka UAV przejawiała dziwne zachowania, takie jak np. wznoszenie się w nieskończoność i rozbijanie się o ściany. Obraz po uruchomieniu Sphinx'a według instrukcji zawartych w [3] przedstawiono na rysunku 1.3.



Rysunek 1.3: Symulator z uruchomionym modelem jednostki Bebop 2

Rzeczywista jednostka Bebop 2, na której miały odbyć się planowane testy, została wyposażona w owiewki chroniące śmigła na wypadek kolizji. Tym samym masa UAV zwiększyła się o 0.05 kg co stanowi 10% wagi samego robota latającego. Bazując na tych informacjach zdecydowałyśmy się na wprowadzenie zmian w domyślnym modelu Bebopa w Sphinxie, mając w planach zmianę wyglądu oraz masę i inercję, a także zwiększenie obszaru kolizji. Zmiana samego wyglądu nie była problemem, gdyż po znalezieniu w plikach symulatora modelu podstawy, w edytorze 3D dodałyśmy do niego owiewki.

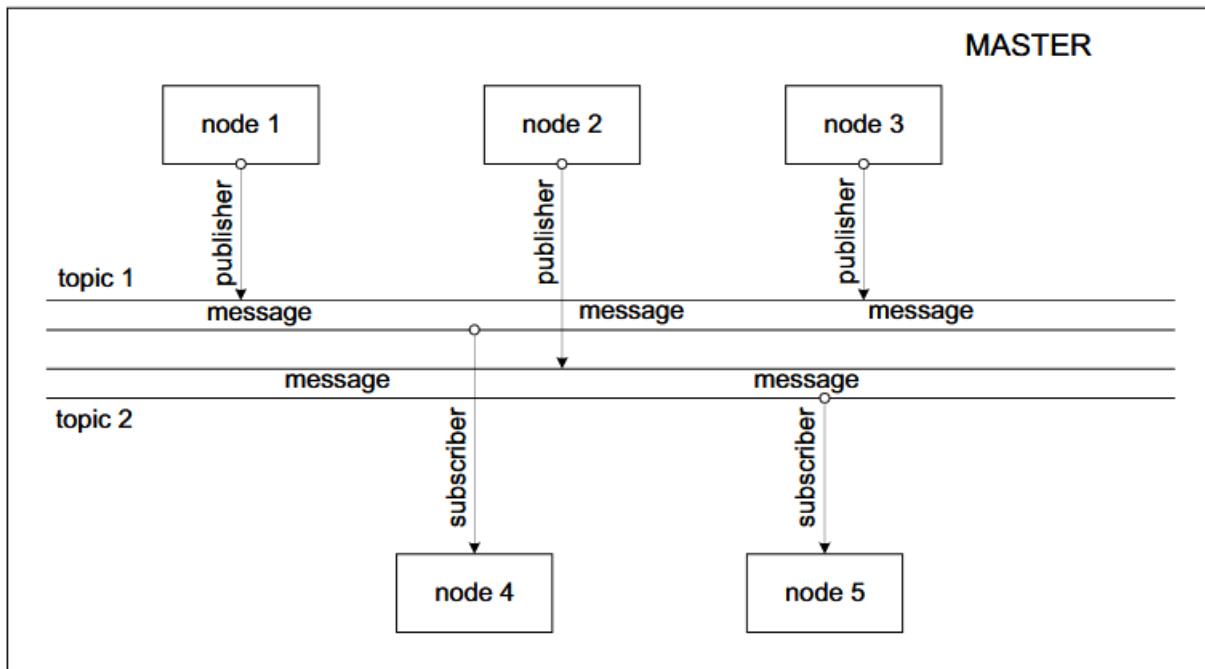


Rysunek 1.4: Model 3D Bebopa 2 przed i po modyfikacji

Większym problemem okazało się znalezieniem pliku SDF modelu UAV. Po skontaktowaniu się z osobami z firmy Parrot poprzez ich forum, otrzymałyśmy odpowiedź zwrotną, której oryginał znajduje się w [4]. Niemożliwe okazały się modyfikacje w fizyce jednostki UAV ze względu na szyfrowanie pliku z rozszerzeniem *.sdf, dlatego musiałyśmy zadowolić się tylko i wyłącznie zmianą wizualną, która została przedstawiona na rysunku 1.4.

Mając skonfigurowane środowisko symulacyjne, dalszym działaniem było dodanie komunikacji z jednostką UAV. Nasz wybór padł na bebop_autonomy [5] - sterownik dla Bebop 1 i Bebop 2, działający pod Robot Operating System. W celu dalszego zrozumienia funkcjonalności użytego pakietu, konieczne jest opisanie podstawowych zasad architektury systemu ROS.

Robot Operating System jest elastycznym systemem do tworzenia oprogramowania i sterowania robotów. Jego działanie opiera się na małych, działających wspólnie programach, które nazywane są węzłami (ang. Node). W celu kontroli poprawności wymiany danych pomiędzy nimi, istnieje nadzędny węzeł: ROS Master. Programy komunikują się między sobą wysyłając wiadomości (ang. Messages). Są one zorganizowane w tematach (ang. Topic). Aby temat mógł przekazywać dane do innego, musi je publikować (ang. Publisher). Natomiast temat, który odbiera te dane jest wtedy subskrybentem (ang. Subscriber) [6]. Wiadomości mają określony typ, który mówi jakich danych możemy się spodziewać np. typ *sensor_msgs/Image* mówi, że jako informację zwrotną otrzymamy obraz z kamery. Ogólną ideę działania systemu przedstawiono na rysunku 1.5.



Rysunek 1.5: Działanie projektu na platformie ROS [6]

Oprogramowanie systemu ROS składa się z pakietów (ang. Packages), które możemy dołączyć do swojego projektu. Jednym z takich pakietów jest właśnie `bebop_autonomy`, który oferuje podstawowe funkcje sterowania jednostką UAV takie jak:

- wzniesienie się - **`takeoff`**,
- lądownie - **`land`**,
- awaryjne odłączenie napędów - **`reset`**,
- sterowanie prędkością liniową i kątową w zakresie osi - **`cmd_vel`**,
- sterowanie obrotem kamery w kierunkach góra/dół i prawo/lewo - **`camera_control`**.

Użytkownik ma również możliwość odczytu:

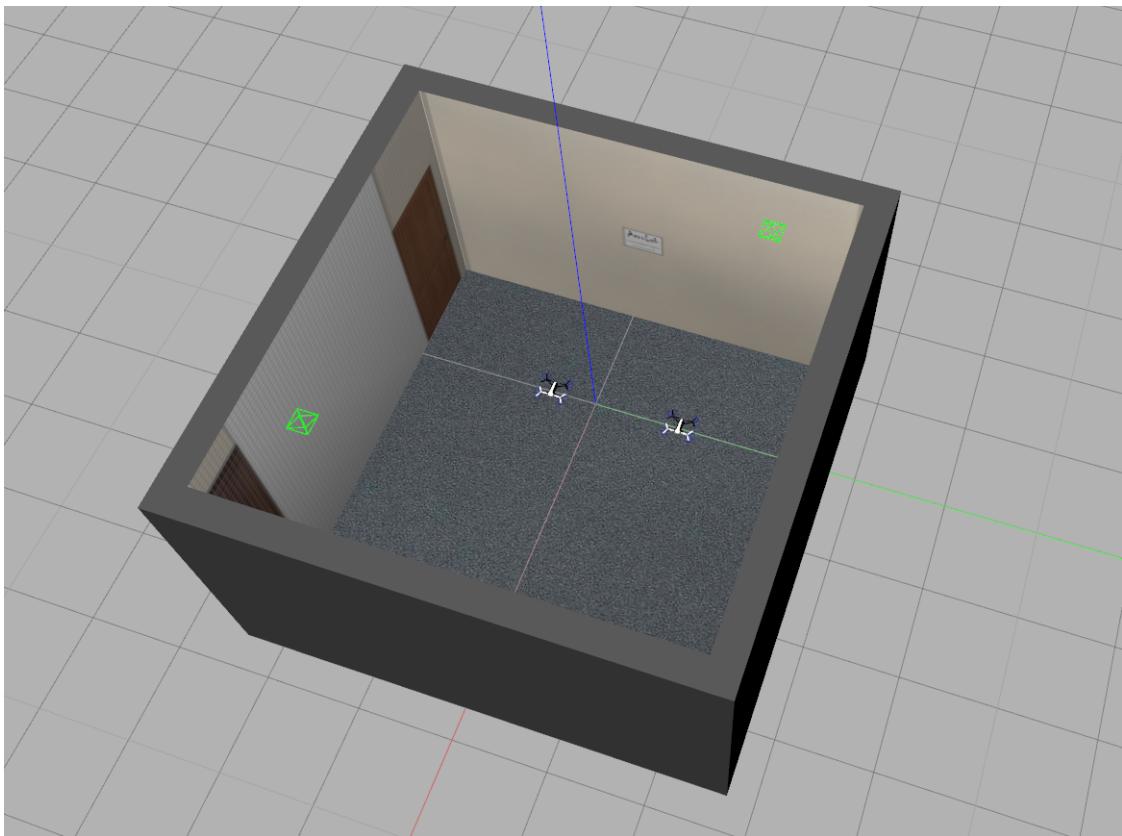
- obrazu z przedniej kamery - **`image_raw`**,
- informacji o kamerze i danych kalibracyjnych - **`camera_info`**,
- danych odometrycznych jednostki UAV - **`odom`**,
- informacji o pochyleniu i obróceniu kamery - **`joint_states`**.

Sterownik został zainstalowany i uruchomiony według wytycznych ze strony internetowej [5].

1.2.3 Uruchomienie wielu jednostek UAV

Po wykonaniu wszystkich działań opisanych w rozdziale 1.2, udało nam się uruchomić symulator Sphinx w stworzonym świecie z jedną jednostką UAV oraz podstawowym sterowaniem. Na potrzeby późniejszego rozwinięcia zadań, kolejnym krokiem było uruchomienie dwóch robotów latających w jednej symulacji wraz ze sterownikami. Koncepcyjnie miały zostać stworzone dwie jednostki: lider oraz śledzący. Aby było to możliwe, musiały zostać spełnione pewne wymagania sprzętowe. Każdy symulowany UAV tworzy punkt dostępu Wi-Fi, dlatego dla dwóch jednostek potrzebne są dwie karty sieciowe. W naszym przypadku, drugą była zewnętrzna karta USB: TP-LINK TL-WN725N.

Następnie napisałyśmy dwa pliki z rozszerzeniem *.drone, które według założeń Sphinx'a są plikami opisującymi jednostki UAV [2]. Ważnym aspektem jest tutaj podanie poprawnej nazwy interfejsu Wi-Fi, która pojawia się w terminalu po wprowadzeniu polecenia *ifconfig*. Nadałyśmy również nowe nazwy: *bebop_leader* i *bebop_follower*.

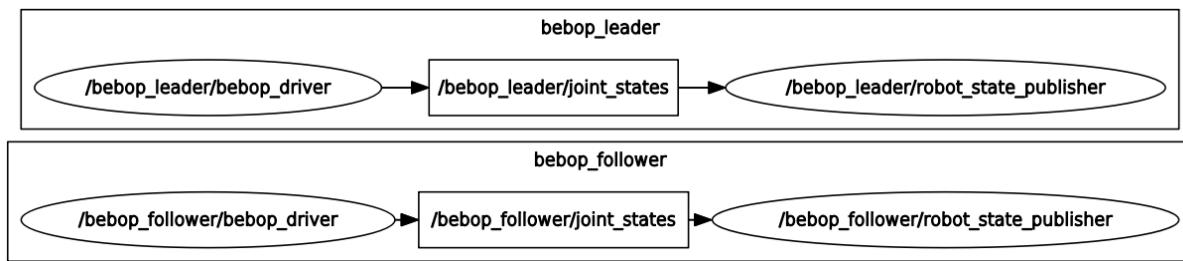


Rysunek 1.6: Dwie jednostki UAV uruchomione w symulacji

Przy późniejszym uruchomieniu sterownika dla więcej niż jednej jednostki, problemem okazała się najnowsza wersja oprogramowania sprzętowego użyta w Sphinxie. Bazując na znalezionym temacie na forum firmy Parrot [7], pobrałyśmy jego starszą wersję: Bebop 2 - 4.4.2 ze strony [8] a w plikach *.drone zmieniłyśmy ścieżkę dostępu. Następnie przenosłyśmy je do katalogu zawierającego inne jednostki: */opt/parrot-sphinx/usr/share/sphinx/drones*.

Tak przygotowane pliki pozwoliły na uruchomienie symulacji przedstawionej na rysunku 1.6.

Chcąc stworzyć możliwość sterowania obiema jednostkami, konieczne było dodanie drugiego węzła do pakietu bebop_autonomy z odpowiednią nazwą i adresem IP symulowanego UAV. W rezultacie otrzymałyśmy strukturę projektu w systemie ROS przedstawioną na rysunku 1.7.



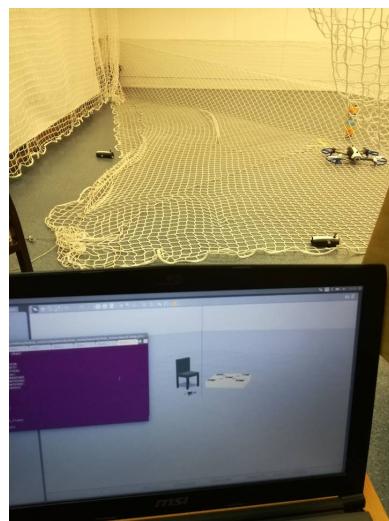
Rysunek 1.7: Graf przedstawiający strukturę węzłów i tematów w systemie ROS

Rozszerzona rzeczywistość we wsparciu prototypowania lotów UAV

2.1 Wprowadzenie

Alicja Kuźniewska

Rozszerzona rzeczywistość (ang. Augmented Reality) jest tematem popularnym i szybko rozwijającym się, a jej wykorzystanie można zauważać w każdej sferze życia. Jest to interaktywne połączenie elementów cyfrowych w środowisku realnym, którego doświadczamy codziennie. Polega np. na nałożeniu obrazu fikcyjnego na widok świata w czasie rzeczywistym, dzięki czemu zwiększone zostają doznania odbiorcy. Na znaleziony na obrazie znacznik, nakładany jest wykrawany model 3D i w ten sposób oblicze przedstawione na kamerze wygląda całkowicie inaczej niż poza nią. Przykład tej różnicy przedstawiony jest poniżej. Na rysunku 2.1. można dostrzec jak bardzo różni się widok z Gazebo, a środowisko rzeczywiste. W symulatorze widoczne jest krzesło, platforma oraz robot latający, tymczasem w realnym świecie są dwie baterie ze znacznikami oraz jednostka UAV.



Rysunek 2.1: Przykład różnicy pomiędzy widokiem w rzeczywistości oraz w symulatorze

Jak wcześniej wspominałyśmy, AR ma wiele zastosowań - w dzisiejszych czasach między innymi używana jest w medycynie. Pozwala ona przykładowo na nałożenie obrazu na skórę pacjenta, w celu zobrazowania żył biegących w danej części ciała, czy też obrazu narządów wewnętrznych podczas operacji. Prowadzi to do zmniejszenia ilości pomyłek popełnianych przez lekarzy. Wykorzystywana jest również w sferze wojskowej do doskonalenia umiejętności żołnierzy, poprzez tworzenie misji szkoleniowych wykorzystujących nałożony widok na środowisko realne. Rozszerzona rzeczywistość wykorzystywana jest również, aby pomóc w poruszaniu się po lotnisku za pomocą aplikacji wykorzystującej tę technologię. W tej pracy została ona użyta do stworzenia symulatora, w którym możliwe będzie zadawanie obiektów, których w realnym świecie nie ma. Pozwala to na zmniejszenie kosztów nakładu finansowego, jak również zwiększa bezpieczeństwo przyszłych testów. Biorąc pod uwagę wyposażenie sali laboratoryjnej, do zrealizowania tego zadania zdecydowały się na wykorzystanie systemu OptiTrack, który posiada własne znaczniki. Posiada on wiele funkcji, jednak w tym przypadku stosowany jest do śledzenia pozycji oraz orientacji przedmiotów. W środowisku tym wykryte markery kojarzone są w jeden obiekt, a na podstawie ich rozmieszczenia, obliczane jest położenie środka masy w rzeczywistości. Poniższy rysunek przedstawia ogólną koncepcję działania symulatora.



Rysunek 2.2: Wykorzystane interfejsy

Jak zostało wcześniej wspomniane, OptiTrack oblicza położenie środka masy. Informacje te zostają wykorzystane do umieszczenia modeli na danych pozycjach w interfejsie symulacyjnym Gazebo. Przekazanie wartości współrzędnych odbywa się poprzez wysyłanie wiadomości pomiędzy odpowiednimi tematami systemu ROS.

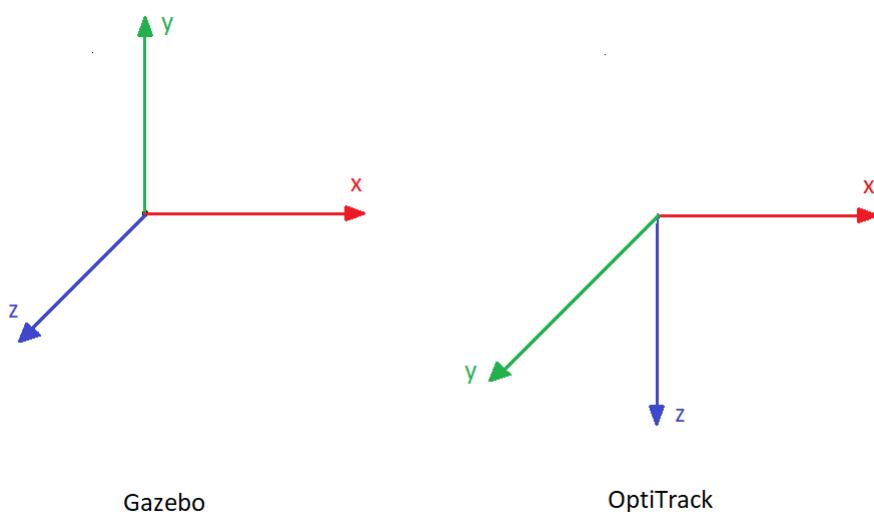


Rysunek 2.3: Bateria UAV z naniesionymi znacznikami wykorzystywanymi w OptiTracku

2.2 Implementacja oraz uruchomienie

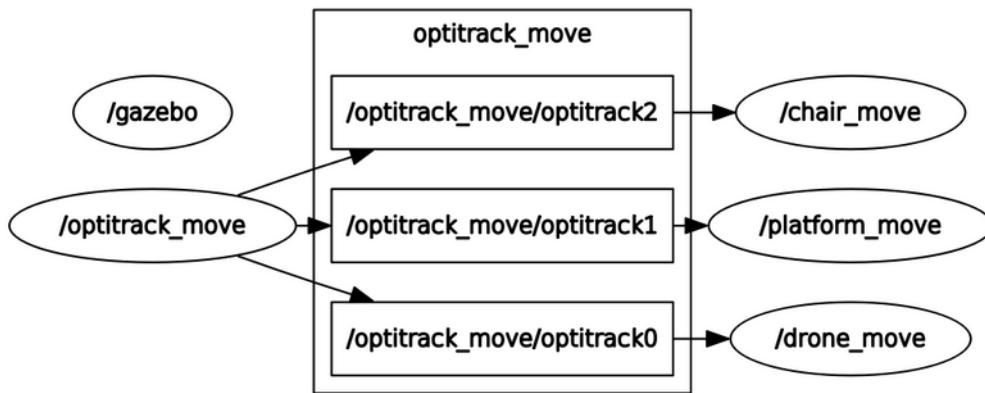
Alicja Kuźniewska

W celu implementacji użyłyśmy gotowego pakietu, który został stworzony przez Łukasza Halbiniaka oraz doktorantów dr Kaczmarka z IARII o nazwie *Optitrack* oraz zmodyfikowałyśmy go na własne potrzeby. Bazowo składał się z dwóch paczek o nazwie *optitack_move* oraz *drone_move*. Pierwsza z wymienionych zawierała w sobie dwa skrypty oraz plik uruchomieniowy *optitrack_move.launch*, który pozwalał na zmianę takich parametrów jak: liczba przetwarzanych modeli, lokalny adres oraz adres serwera. Po uruchomieniu wyżej wymienionych programów, okazało się, że nie działają one prawidłowo, mimo komplikacji bez błędów, więc musiały ulec modyfikacjom. W skrypcie o nazwie *optitrack_move*, wchodzący w skład pakietu, zrealizowana jest inicjalizacja systemu śledzącego ruch, pobranie parametrów sieci, liczby obiektów oraz dodanie ich do wektora zawierającego opublikowaną pozycję. Ze względu na typ wiadomości tematu o nazwie */optitrack_move/optitrack*, do którego wysyłane są dane obiektów, współrzędne aktualnej pozycji musiały zostać przypisane do odpowiednich typów zmiennych. Jest to główny skrypt przetwarzający położenie z systemu. Chciałyśmy, aby w symulatorze Gazebo była możliwość umieszczenia oraz poruszania modelami: Bebopa, krzesła i platformy ze znacznikami Aruco, dlatego poza zmodyfikowaniem paczki *drone_move*, konieczne było dodanie pakietów związanych z pozostałyimi obiekty. W skrypcie odpowiadającym za robota latającego zrealizowana jest subskrypcja tematu */optitrack_move/optitrack0*, w którym zawarte są koordynaty obiektu z systemu. Musiały one zostać przekształcone, ze względu na różniące się układy współrzędnych pomiędzy OptiTrackiem a Gazebo.



Rysunek 2.4: Zależności pomiędzy układem współrzędnych w symulatorze Gazebo i w systemie OptiTrack

Następnie w programie utworzono klienta Gazebo, do którego zostaje wysłana wiadomość z systemu ROS z koordynatami. Dane te trafiają do tematu o nazwie `/gazebo/set_mode_state`. Dalej, przypisywana zostaje odpowiednia nazwa modelu oraz następuje publikowanie współrzędnych. Jak już zostało wspomniane powyżej, niezbędne było stworzenie jeszcze dwóch paczek z obiekttami, w celu dodania ich do symulatora. Jedyne czym różniły się one od wcześniej opisywanego pakietu, była nazwa modelu i subskrybowany temat z danymi odnośnie położenia.

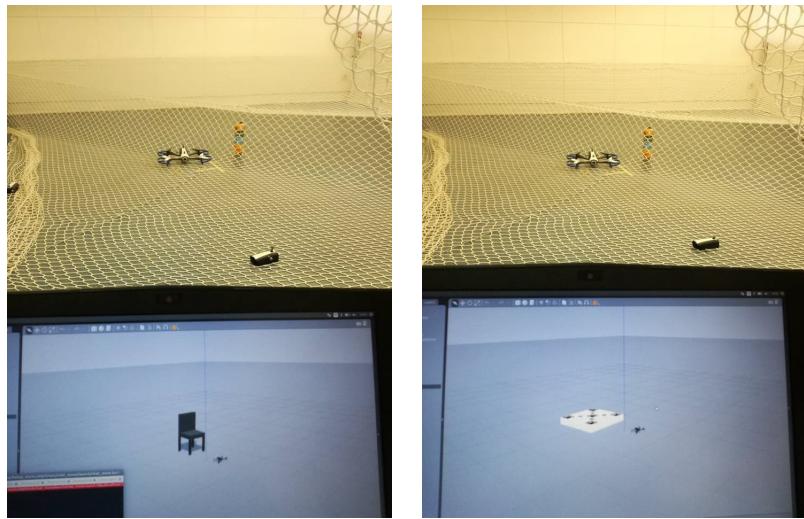


Rysunek 2.5: Graf węzłów i tematów pakietu *Optitrack*

Poniżej zamieściliśmy listę kroków, w celu uruchomienia symulatora wspartego rozszerzoną rzeczywistością. Zostaje ona umieszczona ze względu na trudności jakie mogą wystąpić.

1. roscore
2. rosrun gazebo_ros gazebo
3. `rosrun gazebo_ros spawn_model -database parrot_bebop_2 -gazebo -model parrot_bebop_2 -y 1`

Po wpisaniu powyższych komend w terminalu, uruchomiony zostanie symulator Gazebo oraz umieszczony w nim model jednostki UAV Bebop 2. Działanie powyżej opisanego rozwiązania przedstawione jest na rysunku 2.6 oraz udokumentowane nagraniem na płycie CD: *Multimedia/Testy rzeczywiste/ar_simulator_test.png*.



(a)

(b)

Rysunek 2.6: Działanie zmodyfikowanego przez nas pakietu *Optitrack*

- a) Widok z Gazebo, na którym jest krzesło oraz UAV, podczas gdy w rzeczywistości jest bateria ze znacznikami oraz Bebop.
- b) Widok z Gazebo, na którym jest platforma oraz robot latający, podczas gdy w rzeczywistości jest bateria ze znacznikami oraz Bebop.

Algorytm omijania przeszkód przy wykorzystaniu głębokiego uczenia

Alicja Kuźniewska

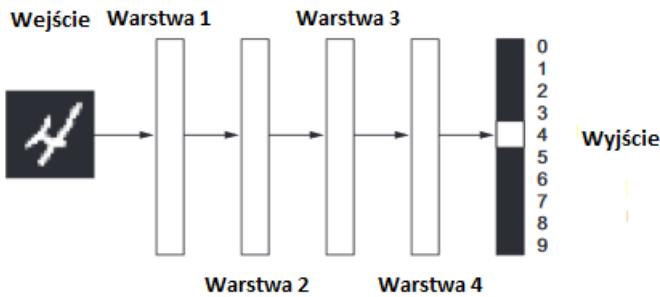
3.1 Opis wybranej metody

Rozpoznawanie przeszkód jest tematem popularnym ze względu na szybko rozwijającą się sferę robotyki. Do tej pory zostało opracowanych wiele algorytmów zdobywających informacje o przeszkodzie z wykorzystaniem różnego zaplecza sprzętowego. Jednak nadal pozostaje to trudnym przedmiotem badań. Większość opracowanych metod opiera się na pozyskiwaniu danych z obrazu głębi w celu określenia, w jakiej odległości znajduje się obiekt. Mając do dyspozycji tylko pojedynczą kamerę RGB, trudno jest określić czy obiekt znajdujący się na obrazie stanowi zagrożenie dla jednostki UAV. Wybór metody omijania przeszkód został podyktowany ograniczeniami sprzętowymi, w związku z tym zdecydowałam się na wykorzystanie SSD Multibox (ang. Single Shot Multibox Detector) do detekcji obiektów oraz metody opartej na przepływie optycznym, zwracającej informacje o wystąpieniu potencjalnej przeszkody. Następnie na podstawie tej informacji zostało wykonane ominięcie wykrytego obiektu.

3.1.1 Single Shot Multibox Detector

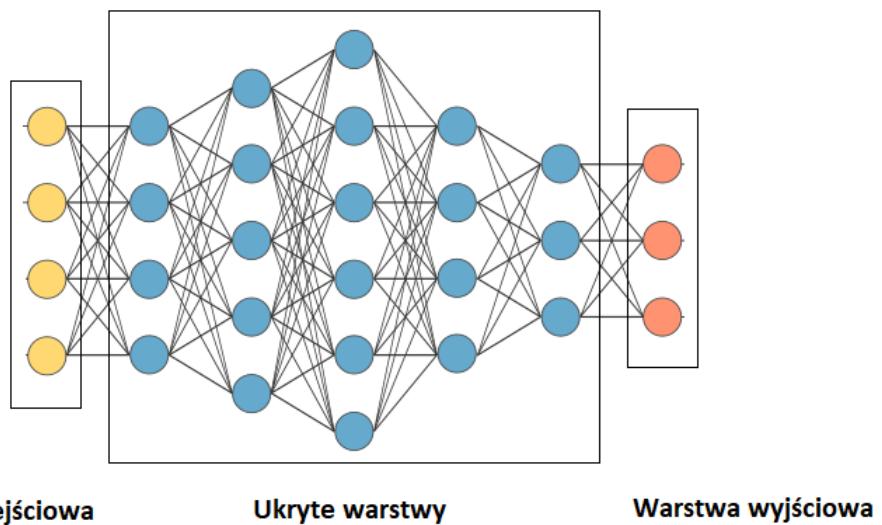
Głębokie uczenie (ang. Deep Learning) jest podkategorią uczenia maszynowego. To algorytm, który kładzie nacisk na uczenie się kolejnych warstw coraz bardziej znaczących reprezentacji danych. Niektóre wykorzystania obejmują setki a nawet tysiące kolejnych warstw reprezentacji, a inne bazują na jednej lub kilku (ang. Shallow Learning). Ich postęp zależy od danych treningowych. Sieci neuronowe to matematyczna struktura, przez którą uczą się reprezentacje danych. Składają się z powłok ułożonych jedna na drugiej. Przedstawienie jak wygląda schemat sieci kilkuwarstwowej przedstawiony jest poniżej.

Klasyfikacja obrazów opiera się na sieci CNN (ang. Covolutional Neural Network). Pierwszym krokiem w tym działaniu jest przejście wejściowego obrazu przez serie konwo-



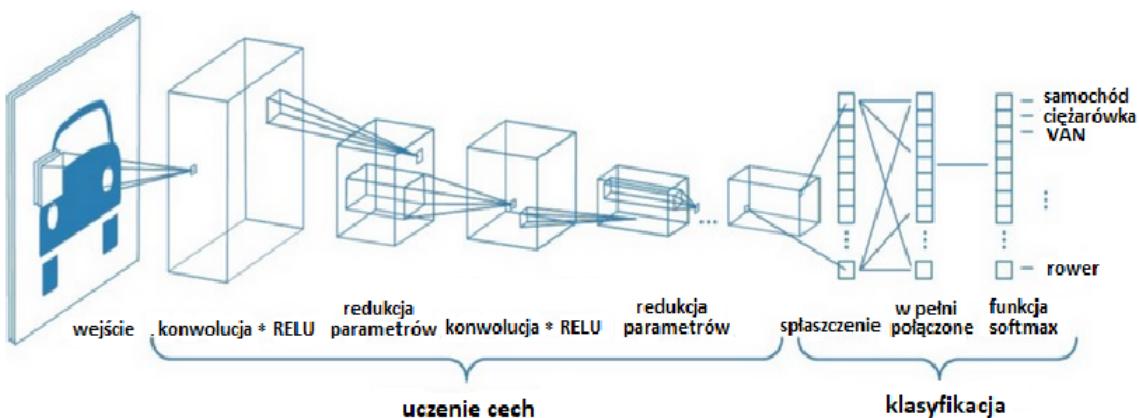
Rysunek 3.1: Schemat sieci neuronowej kilkuwarstwowej [9]

lucyjnych warstw. Taka powłoka pozwala na wyodrębnienie cech z obrazu. Konwolucja jest to operacja matematyczna, która jako wejścia przyjmuje macierz obrazu oraz filtr. Zachowuje ona związek między pikselami obrazu poprzez uczenie się cech widoku przy użyciu małych kwadratów z danych wejściowych. W celu otrzymania mapy cech mnożona jest macierz obrazu z macierzą filtra. Podczas tego mnożenia filtr w każdej jednostce czasu jest przesuwany po obrazie o jeden piksel i mnożony z częścią macierzy obrazu, którą pokrywa. W ten sposób obliczane są kolejne wartości macierzy cech. Wynikiem tego mnożenia jest mapa cech. Więcej informacji odnośnie przedstawionych powyżej informacji można znaleźć w [10]. Następnie po każdej warstwie konwolucyjnej stosuje się nieliniową powłokę (ReLU). Celem tego zastosowania jest włączenie nieliniowości do systemu. Poprawia to szybkość trenowania oraz pozwala na zniwelowanie problemu z gradientem znikającym. Jest to niedogodność, w której dolne warstwy trenują wolniej. Przedostatnim krokiem jest zmniejszenie liczby parametrów, gdy obraz jest za duży, ale z zachowaniem najważniejszych informacji (ang. Pooling).



Rysunek 3.2: Warstwa w pełni połączona [10]

W ostatnim kroku wprowadza się spłaszczoną do wektora macierz obrazu do w pełni połączonej warstwy (ang. Fully Connected Layer). Dzięki niej dane wejściowe są łączone w model. Ukrytymi warstwami są powłoki konwolucyjne, które zostały opisane wyżej. Wyjście jednej z nich jest wejściem dla kolejnej. Wyjście z powłoki wyjściowej jest sklasyfikowane za pomocą funkcji aktywacji takiej jak softmax. Wartości tej funkcji są znormalizowane tak, aby suma aktywacji dla powłoki wynosiła 1. Dzięki czemu danymi wyjściowymi są prawdopodobieństwa przynależności do odpowiednich klas. Na rysunku 3.2 został przedstawiony opisany powyżej sposób działania.Więcej informacji na ten temat znajduje się w [10].

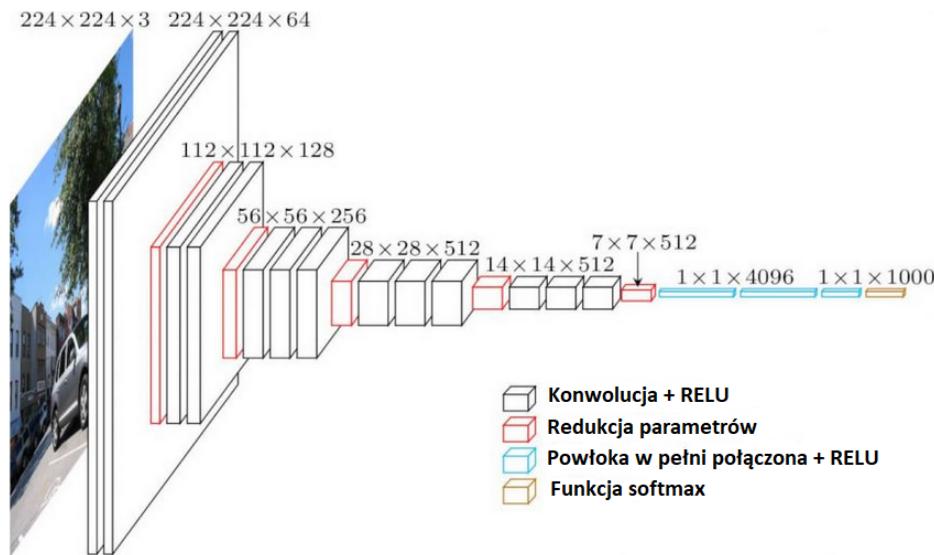


Rysunek 3.3: Sposób działania CNN [10]

Różnica między detekcją obrazów a przynależnością do odpowiednich klas jest taka, że podczas klasyfikacji zwracana jest informacja do jakiej grupy należy obiekt znajdujący się na dostarczonym obrazie wejściowym. Natomiast podczas detekcji, celem jest narysowanie obramowania wokół rozpoznanego obiektu, aby zlokalizować jego położenie na obrazie. Wybór algorytmu do detekcji padł na SSD Multibox, ponieważ jest on w stanie dokonać rozpoznania na obrazach o 59 klatkach na sekundę. Został on opracowany przez Wei Liu z Uniwersytetu w północnej Karolinie.

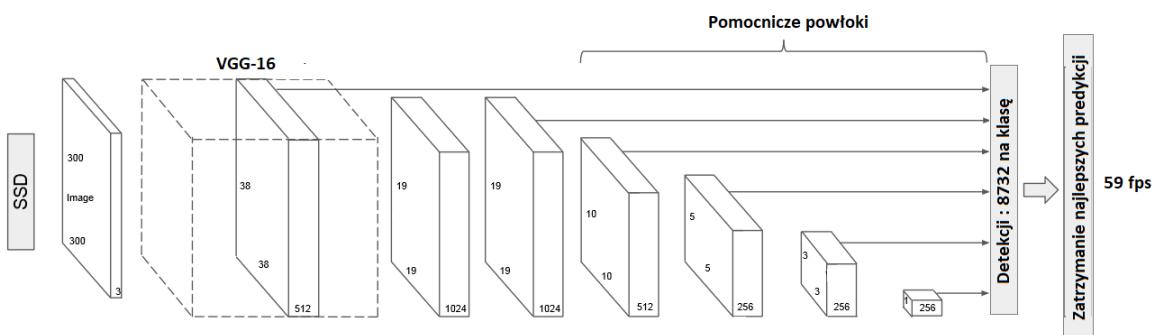
Nazwa Single Shot wzięła się z tego, że zadanie lokalizacji obiektu oraz klasyfikacji odbywa się przy pojedynczym przekazaniu do sieci, Multibox określa technikę obramowania obiektu, a Detector oznacza sieć, która dokonuje detekcji obiektu oraz jego klasyfikacji.

Bazową siecią jest VGG-16. Jest to wstępnie przetrenowana sieć konwolucyjna CNN posiadająca szesnaście warstw. Zasada działania takiej sieci została opisana powyżej. Różnica pomiędzy przedstawionym sposobem działania, a VGG-16 wiąże się tylko z ilością powłok. Rozmiary widoczne na rysunku 3.4 są to wymiary wejściowych warstw do kolejnych w sieci.Więcej informacji na temat tej sieci można znaleźć w [11]. Jej architektura została przedstawiona poniżej.



Rysunek 3.4: Architektura sieci VGG-16 [11]

VGG-16 została użyta jako bazowa, ponieważ zapewnia dużą wydajność w zadaniach rozpoznania obiektów. W budowie SSD zostały pominięte w pełni połączone warstwy z VGG-16, a zamiast nich użyte pomocnicze powłoki konwolucyjne. Z każdej była generowana mapa cech wyodrębnionych z obrazu, który został podany na wejściu. Dzięki takiemu zastosowaniu możliwe było wyodrębnienie atrybutów w wielu skalach oraz stopniowe zmniejszanie rozmiaru obrazu wejściowego do każdej kolejnej warstwy, co widoczne jest na rysunku 3.5.



Rysunek 3.5: Architektura SSD [12]

W regresyjnej metodzie propozycji współrzędnych obramowania obiektów wykorzystywana jest konwolucyjna sieć neuronowa. Konwolucja używana jest do redukcji wymiarów. Funkcja utraty Multibox łączyła dwie funkcje krytycznego zgubienia, które zostały wykorzystane w SSD. Pierwsza z nich, funkcja strat pewności, mierzy jak pewna jest sieć względem wyliczonego obramowania wokół obiektu, druga natomiast jak daleko od ram z bazy treningowej znajdują się te przewidziane przez sieć. Druga funkcja nazywa się funkcją utraty lokalizacji. Poglądowe przedstawienie obliczeń dla całkowitej funkcji utraty Multibox przedstawione jest poniżej, jednak trzeba mieć na uwadze, że jest to poglądowy wzór:

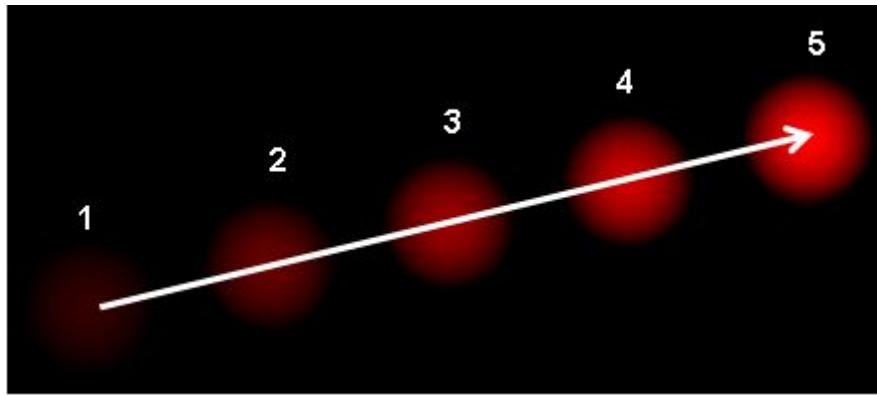
$$\text{multibox_loss} = \text{confidence_loss} + \alpha * \text{location_loss}, \quad (3.1)$$

gdzie multibox_loss to utrata Multibox, confidence_loss to utrata pewności ramy wokół obiektu, location_loss to utrata lokalizacji obramowania, a alpha służy do zrównoważenia zgubienia lokalizacji. Redukując w ten sposób funkcje strat i poprawiając przewidywanie sieci. W Multibox wybierane są stałe rozmiary obramowań obiektów, które określają gdzie na obrazie ze zbioru treningowego znajduje się obiekt (ang. Prior). Te rozmiary są zaakceptowane na podstawie wyniku stosunku obszaru pokrywania się między ramami przewidzianymi przez sieć, a tymi ze zbioru treningowego do obszaru całości (ang. IoU), gdy wynik ten jest równy lub większy niż 0.5. Więcej informacji odnośnie SSD można znaleźć w [12].

W celu wytrenowania sieci niezbędne są bazy do wyuczenia wraz ze zdefiniowanymi oprawami oraz etykietami obiektów. Dzięki czemu po wytrenowaniu, im bardziej domyślne będą obramowania tym lepsze będą przewidywania ram. Jest to pierwszy krok w wyuczeniu sieci. Następnie tworzona jest mapa wyodrębnionych cech z obrazów z bazy treningowej, czyli zbiór dominujących atrybutów na obrazie. Podczas następnego kroku zachowane zostają zarówno pozytywne jak i negatywne przewidywania obramowań. Musi być to zachowane w stosunku 3:1, aby sieć była w stanie odpowiednio przewidzieć ramy, ale również była w stanie powiedzieć co stanowi nieprawidłową predykcję. Kwalifikacja do prawidłowych lub nieprawidłowych zależy od wartości IoU. W kolejnym kroku generowane są dodatkowe przykłady treningowe z łatami oryginalnego obrazu przy różnych wartościach IoU oraz kilkoma losowymi by sieć była bardziej odporna na różne wielkości danych wejściowych. Dodatkowo, aby polepszyć wyniki uczenia niektóre obrazy są odwracane, dzięki czemu obiekty pojawiają się po lewej i po prawej stronie. Ostatnim krokiem jest zatrzymywanie najlepszych predykcji przez sieć, a odrzucanie tych gorszych. Jeśli wartość funkcji utraty pewności obramowania oraz wartość IoU jest mniejsza od pewnego dobranego progu to taka predykcja jest uznana za gorszą (ang. Non-maximum Suppression). Więcej informacji odnośnie trenowania sieci można znaleźć w [12].

3.1.2 Przepływ optyczny

Przepływ optyczny opisuje się jako pole wektorowe ruchu pomiędzy dwiema sąsiadującymi klatkami obrazu [13]. Przemieszczenie danego punktu definiuje jego początek i koniec. Każdy wektor zawiera informacje o prędkości zmiany położenia danego punktu oraz, w którą stronę ten ruch się odbywa pod postacią takich parametrów jak kierunek, zwrot i długość tego wektora. [13]



Rysunek 3.6: Przedstawienie położenia czerwonego punktu w 5 następujących po sobie klatkach, strzałka obrazuje wektor ruchu [14]

Metoda przepływu optycznego ma dwa założenia : jasność punktu nie zmienia się w czasie oraz ruch sąsiadujących pikseli jest podobny. Te informacje można znaleźć w [14]

Z pierwszego założenia wynika:

$$I(x, y, t) = (x + dx, y + dy, t + dt) \quad (3.2)$$

gdzie x, y, t to współrzędne piksela, I(x,y,t) to intensywność, dx i dy opisują przesunięcia między klatkami w pewnym przedziale czasowym dt.

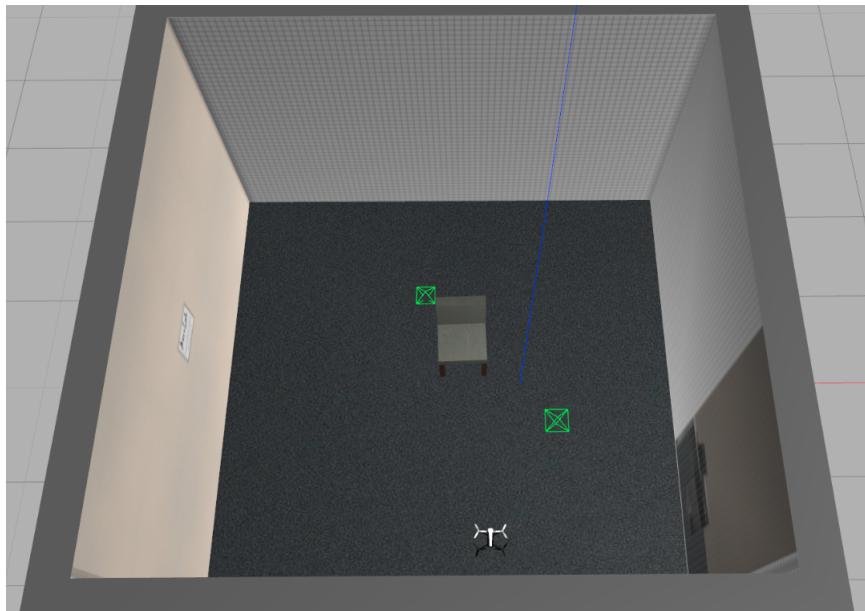
$$\frac{\partial I}{\partial x} \frac{dx}{dt} + \frac{\partial I}{\partial y} \frac{dy}{dt} + \frac{\partial I}{\partial t} = 0 \quad (3.3)$$

Powyzszy wzór jest nazywany równaniem przepływu optycznego, w którym $\frac{\partial I}{\partial x}$ i $\frac{\partial I}{\partial y}$ to gradienty obrazu, a $\frac{dx}{dt}$ i $\frac{dy}{dt}$ są niewiadomymi, biorąc to pod uwagę jest to równanie niemożliwe do rozwiązania. Przyjmując jak wynika z drugiego założenia przedstawionego wcześniej, że ruch sąsiadujących pikseli jest podobny otrzymujemy n^2 równań z taką samą liczbą niewiadomych. Przedstawione założenia wykorzystuje algorytm Lucas-Kanade [14]. Jest to podejście z grupy gradientowej. Polega ono na znalezieniu położenia punktów charakterystycznych na obrazie oraz wyznaczeniu śladu ich ruchu określając ich nową lokalizację na każdej nadchodzącej klatce. Aby można było poradzić sobie z dużymi ruchami w algorytmie tym wykorzystywane są piramidy rozdzielczości [15]. Polega to na badaniu obrazów z różną rozdzielczością zaczynając od najwyższego poziomu. Na tym poziomie obraz ma

najniższą rozdzielcość. W ten sposób duże ruchy są przechwytywane i staja się małymi. Więcej informacji na temat tego podejścia można znaleźć w [13], [14] oraz [15]. Metoda ta została wykorzystana w pracy dyplomowej do obliczania przepływu optycznego na kolejnych klatkach obrazu, dane te zostały wykorzystane w algorytmie omijania przeskód, który jest przedstawiony w sekcji 3.3.

3.2 Stworzenie świata symulacyjnego

Docelowo sieć SSD miała być wykorzystywana do rozpoznana obiektów znajdujących się na obrazie z tematu `/bebop_leader/image_raw`. Jest to informacja dla obserwatora lub użytkownika, aby wiedział co znajduje się w pobliżu jednostki UAV i z czym się ona mierzy. Konieczne było stworzenie modeli przeskód, które ta sieć była w stanie rozpoznać. Dodatkowo kolejnym aspektem, który należało wziąć pod uwagę było jedno z ograniczeń metody przepływu optycznego, którym jest brak możliwości wykrycia zmian pomiędzy dwiema klatkami, jeśli na przetwarzanym fragmencie obrazu był widok jednobarwny. W takim wypadku nie doszłoby do wykrycia przeskody, ponieważ taka cecha powoduje, że generowany przepływ jest zerowy. Ze względu na powyżej wymienione ograniczenia zdecydowałem się na stworzenie modelu 3D krzesła ze względu na wymiary oraz prostą konstrukcję. W pierwszym wygenerowanym przeze mnie świecie i wykorzystywanym w większości przypadków do sprawdzenia funkcjonalności, znajdowało się tylko jedno krzesło w stworzonym wcześniej i opisany w rozdziale pierwszym pomieszczeniu. Należało również dodać dodatkowe źródła światła, dzięki czemu sieć miała większą pewność rozpoznania. Wpływ oświetlenia na ten parametr będzie przedstawiony w sekcji 5.1. Model tej przeskody musiał być stworzony oraz musiała być do niego dodana odpowiednia kolidyjność. Na poniższym rysunku widać świat w Gazebo z dodaną przeskodą, który był wykorzystywany w większości testów. Ze względu na późniejsze próby, przeskody zostaną umieszczone w zarówno pomieszczeniu jak i w pustym świecie.



Rysunek 3.7: Pomieszczenie z przeskodą

3.3 Implementacja algorytmu

W celu implementacji sieci SSD Multibox (omówionej w sekcji 3.1.1.) posłużyłem się gotowym pakietem o nazwie *object_detection_tensorflow* [16]. Zawiera on pliki uruchomieniowe *.launch oraz skrypty napisane w języku Python, w których skład wchodzi skrypt o nazwie *object_detecton_tensorflow.py* z zaimplementowaną siecią oraz skrypt o nazwie *download_all_models.py* do instalacji wszystkich modeli zawartych w bazie danych COCO. Jest to wytrenowana baza danych. Pozwala na segmentację obrazu w celu rozpoznania obiektów. Zawiera 91 klas obiektów. Pozwala na jednoczesną detekcję 5 różnych przedmiotów. Została ona użyta, ponieważ zawiera wytrenowany model sieci SSD Multibox, który został wykorzystany w tej pracy dyplomowej. Pakiet ten został zaimplementowany do wykorzystania w systemie ROS, który został przedstawiony wcześniej. W plikach uruchomieniowych zawarte są wszystkie nazwy ścieżek, tematów oraz nazw wykorzystywanych w skrypcie, a brak ich dostosowania pod własny system spowodowałby nieprawidłowe działanie lub błędy podczas uruchomienia skryptu *object_detecton_tensorflow.py*. W skrypcie tym tworzony jest węzeł o nazwie */object_detection_tensorflow_node* oraz użyta jest biblioteka tensorflow, wykorzystywana przy głębokich sieciach neuronowych. W pliku o nazwie *ssd_mobilenet_v2.launch*, który uruchamia sieć SSD konieczne było dokonanie zmian takich jak nazwa tematu, który zawiera obraz z kamery, ścieżka do miejsca, gdzie znajduje się model, jego nazwa, liczba klas oraz ścieżka do miejsca, gdzie znajduje się plik w formacie *.pbtxt, który zawiera etykiety. W tym pliku uruchomieniowym możliwa jest również zmiana wartości progu, obrotu oraz zmiennej typu bool odpowiadającej za renderowanie. Po edycji tematu zawierającego obraz z kamery, który miał być prze-

twarzany przez sieć, przystąpiłam do wstępniego uruchomienia skryptu. Niestety, mimo że uruchamiał się bez błędów nie był publikowany temat, w którym zawierał się obraz z detekcją obiektów, dlatego przystąpiłam do modyfikacji skryptu. Okazało się, że wartość zmiennej render w pliku uruchomieniowym została ustawiona na False, natomiast w kodzie następowało ograniczenie, że jeśli wartość ta jest prawdziwa to tylko wtedy następuje publikacja odpowiedniego tematu. Fragment kodu, który za to odpowiada przedstawiony jest na rysunku 3.8.

```
self.render = rospy.get_param('~render', True)
if self.render:
    self.image_pub = rospy.Publisher("detections/image_raw/compressed", CompressedImage, queue_size=5)
```

Rysunek 3.8: Fragment kodu gotowego skryptu [14]

Po zmianie wartości zmiennej render na True oraz uruchomieniu skryptu, udało się dokonać detekcji obiektów znajdujących się na obrazie z kamery. Poniżej znajduje się lista tematów, do których wysyłane są dane oraz tematów, które są nasłuchiwanie.

Subskrybowane :

- /bebop_leader/image_raw

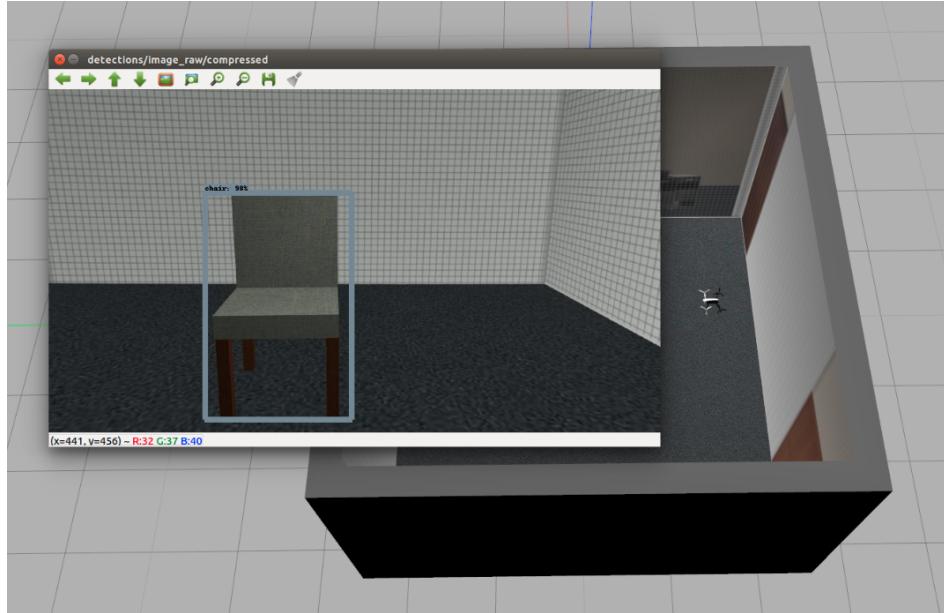
Publikowane :

- /bebop_leader/image_raw/detection
- /detections/image_raw/compressed

Chciałam również, aby w terminalu pojawiała się informacja o tym jaki obiekt został rozpoznany oraz ile wynosiła pewność tego rozpoznania. Funkcjonalność ta została dodana przeze mnie w skrypcie. Dla wygody napisałam dodatkowy pakiet o nazwie *detection_image*, w którym zawarta jest subskrypcja tematu */detections/image_raw*. Paczka ta została stworzona tylko do wyświetlania obrazu z wcześniej wspomnianego tematu. Przy uruchomieniu należy pamiętać, że detekcja jest umieszczona na obrazie skompresowanym, który nie może być odbierany, dlatego dodanie tej części musi nastąpić w terminalu poprzez poniżej przedstawioną komendę

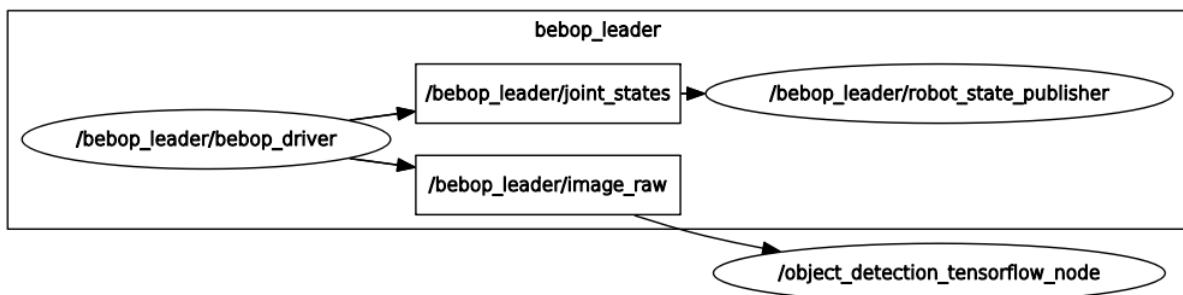
```
rosrun detection_image detection_image _image_transport:=compressed
```

Działanie tego skryptu na stworzonym przeze mnie świecie w Gazebo zostało pokazane na rysunku 3.9.



Rysunek 3.9: Działanie skryptu w pomieszczeniu z przeszkodą

Graf węzłów i tematów systemu ROS po uruchomieniu biblioteki *bebop_autonomy* oraz pakietu *object_detection_tensorflow* przedstawiony jest na rysunku 3.10.



Rysunek 3.10: Graf węzłów i tematów

Po wykonaniu tej części przeszłam do napisania skryptu obliczającego przepływ optyczny metodą Lucas-Kanade, którego opis został umieszczony w sekcji 3.1.2. W celu implementacji, wykorzystałam pakiet *Optical-Flow-based-Obstacle-Avoidance* [17], którego nazwa została zmieniona na *obstacle_avoidance*. W skład tego pakietu wchodzi skrypt napisany w języku C++, a symulacje wykonane zostały przez autora w środowisku Unity3d. Został on zaimplementowany do wykorzystania w systemie ROS. Biorąc pod uwagę, że ja symulacje przeprowadzałam w Gazebo konieczną była zmiana algorytmu omijania przeszkody, ale to rozwiązanie zostało przedstawione później. Skrypt ten został mocno przeze mnie zmodyfikowany, dodałam wiele funkcjonalności, dlatego zdecydowałam się na jego szczegółowy opis.

W podprogramie głównym została zawarta inicjalizacja oraz uruchomienie węzła o nazwie `Obstacle_Avoidance`. Węzeł z pakietu `image_transport` korzysta z `Obstacle_Avoidance`. Ta paczka ROS'a pozwala na przesyłanie oraz odbieranie wiadomości, w których zawarte są obrazy. W programie stworzony został odbiorca o nazwie `image_sub`, który subskrybuje temat `/bebop_leader/image_raw` o typie wiadomości `sensor_msgs/Image`. Jego funkcją jest `imageCb`, która jako argument przyjmuje wiadomość, w której zawarty jest obraz. Utworzeni zostali również odbiorcy `leader_takeoff_sub` oraz `follower_takeoff_sub`, którzy nasłuchują wiadomości na następujących tematach `/bebop_leader/takeoff` i `/bebop_follower/takeoff`. Gdy jakaś zostanie opublikowana w powyżej wspomnianych tematach to w programie zostaje ustawiona odpowiednia zmienna określająca, że dany robot latający jest w powietrzu. W programie istnieje również `threshold_info_sub`, który czeka na zmiany w temacie `/bebop_leader/threshold_reached_info`. Pakietem, w którym zostało zrealizowane sprawdzenie czy lokalizacja robota jest krytyczna oraz publikacja odpowiedniej wiadomości jest `position_controller`, funkcjonalność ta została dodana przeze mnie. W powyższym temacie zawarte są dane o tym czy progowe położenie jednostki w wykorzystywanym pomieszczeniu zostało osiągnięte, jeśli ma wartość `true` to z programu zostaje wysłana wiadomość przez publikującego `vel_pub` do `/bebop_leader/cmd_vel`, który nakazuje stanąć. W temacie tym zawarte są informacje o sterowaniu siłą ciągu, kątami roll (wokół osi x), pitch (wokół osi y) i yaw (wokół osi z). Publikowane są w nim odpowiednie wartości prędkości. Z poziomu tego programu jest to wykonywane w pętli głównej ROS'a działającej z częstotliwością 30 Hz. W niej również znajduje się wykonanie algorytmu omijania przeskody na podstawie odpowiednich danych z przepływu optycznego. W przypadku każdego odbiorcy wykonywana jest funkcja subskrybująca, która wykorzystuje kilka argumentów, a mianowicie:

- nazwa tematu, na którym wiadomości są nasłuchiwanie,
- rozmiar kolejki, w której zapisane są przychodzące wiadomości
- funkcja, która będzie wykonywana za każdym razem, gdy pojawi się nowa wiadomość.

W funkcji `imageCb`, o której wspomniałam wyżej realizowane jest obliczenie przepływu optycznego oraz rysowanie wektorów przepływu. Dalej występuje zmiana formatu obrazu z typu dla wiadomości ROS na typ OpenCV z kodowaniem BGR. Następnie następuje przypisanie obrazu do macierzy o nazwę `frame`. Kolejnym krokiem jest skopiowanie wartości tej macierzy do macierzy o nazwie `image`, która będzie podlegała dalszym modyfikacjom. Kolor obrazu zawartego w tej macierzy musi zostać zmieniony na odcień szarości i po tym kroku macierz o nazwie `gray` jest macierzą przetwarzaną. Dzięki tej czynności w obliczeniach wykorzystywany jest tylko jeden kanał, a tylko te odcienie mogą być przetwarzane przez funkcje obliczającą przepływy. Następnie tworzona jest macierz,

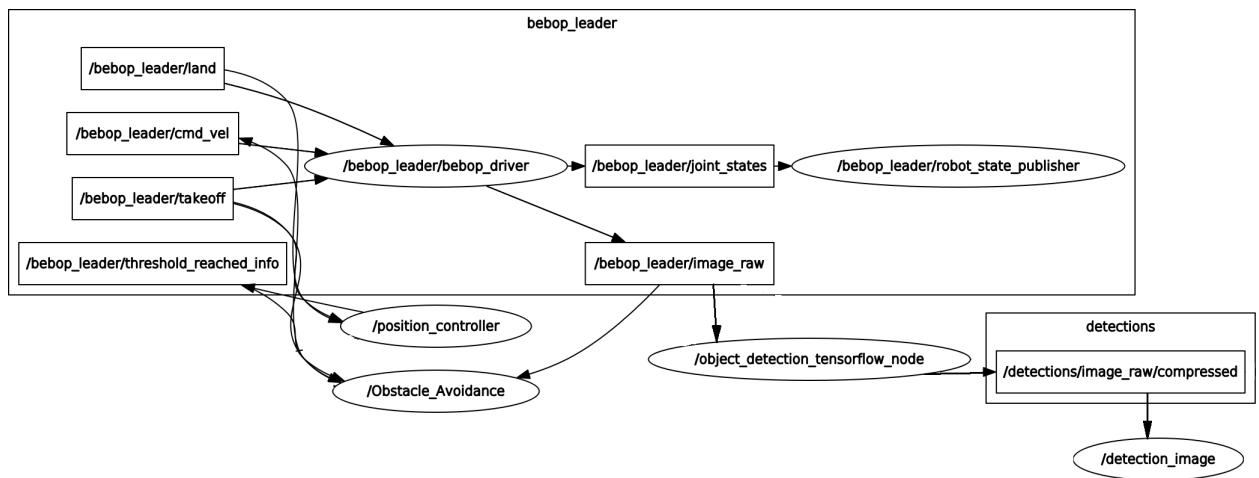
która jest wykorzystywana w funkcji *goodFeatureToTrack* z biblioteki OpenCV. Funkcja ta odpowiada za wyodrębnienie z obrazu charakterystycznych narożników - jest ona wykonywana co około 3 sekundy. Następnie wykonana zostaje funkcja, która poprawia ich położenie (*cornerSubPix*). Powyżej wymienione kroki są niezbędne do inicjalizacji punktów charakterystycznych. Kolejno tworzona jest macierz, która przechowuje poprzednią klatkę obrazu o nazwie *prevGray*. Jest to konieczne do obliczania przepływu optycznego, gdzie wykorzystywana jest klatka poprzednia, bieżąca jak również wektor punktów charakterystycznych na poprzedniej klatce oraz na bieżącej. Po tej funkcji wywoływana jest funkcja rysująca wektory przepływu o nazwie *Draw_flowVectors*. Jako argumenty przyjmuje ona punkty z poprzedniej klatki oraz bieżącej. W funkcji tej obliczana jest również ilość punktów po prawej i lewej stronie klatki, które będą wykorzystane później do algorytmu omijania przeszkody. Po wykonaniu funkcji opisanej powyżej w funkcji *imageCb* rysowane są punkty na obrazie zawartym w macierzy *image* na podstawie punktów charakterystycznych z poprzedniej klatki oraz bieżącej. Następnie zerowana jest zmienna odpowiadająca za inicjalizacje, następuje wyświetlenie obrazu z punktami charakterystycznymi oraz zamieniane są wektory punktów, tak aby wektor z punktami bieżącej klatki stał się wektorem poprzedniej klatki, zamieniane są również obrazy zawarte w macierzach *gray* oraz *prevGray*. Ostatnią czynnością jest wywołanie funkcji odpowiadającej za ustawienie zmiennych *obstacle*, zwraca ona informacje po której stronie znajduje się przeszkoda. Po przekroczeniu ustalonych progów odnośnie ilości punktów po każdej stronie klatki obrazu ustawiana jest odpowiednia zmienna definiująca stronę. Gdy procedura omijania przeszkody jest rozpoczęta zerowane są wektory punktów charakterystycznych, po jej zakończeniu ponownie są one inicjalizowane. Na poniższym rysunku przedstawione jest działanie zaraz po inicjalizacji punktów charakterystycznych.



Rysunek 3.11: Działanie pakietu *obstacle_avoidance* w pomieszczeniu z przeszkodą

Omijanie zrealizowane jest w funkcji o nazwie *avoiding*. Gdy wartość zmiennej odpowiedzialnej za rozpoczęcie procedury jest równa *true* oraz zmiennej odpowiadającej za stronę, po której znajduje się przeszkoła, przez 2 sekundy następuje obrót jednostki UAV dookoła osi z (kąt yaw) w stronę przeciwną do tej, po której znajduje się obiekt. Następnie jest zatrzymywany, w tym miejscu po pewnym opóźnieniu zostają resetowane zmienne odpowiadające za rozpoczęcie omijania oraz ustawiane jest zezwolenie na inicjalizację punktów charakterystycznych na obrazie. Następnie jeśli UAV nie wykrywa żadnej przeszkoły to leci przed siebie.

Na poniższym rysunku przedstawiony jest graf węzłów oraz tematów po uruchomieniu pakietu *object_detection_tensorflow*, *bebop_autonomy*, *position_controller*, oraz *obstacle_avoidance*.



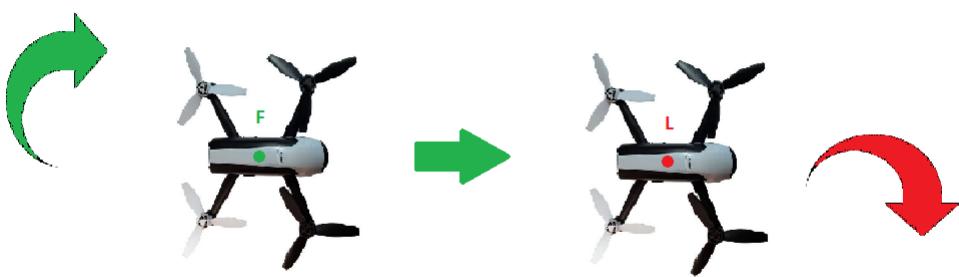
Rysunek 3.12: Graf węzłów i tematów

3.4 Zastosowanie algorytmu dla grupy robotów latających

Mając na uwadze bezpieczeństwo w wykorzystaniu omawianego wyżej rozwiązania dla więcej niż jednego UAV zdecydowałem się na scenariusz, w którym omijanie przeszkód jest wykonywane przez wiodącego robota, którym jest lider. Natomiast drugi śledzi jego trajektorię z ustalonym wcześniej dystansem. Nie podejmuje on żadnych dodatkowych działań poza podążaniem za nim. Funkcjonalność ta została opracowana przeze mnie i umieszczona w pakiecie *position_controller*, który zawiera sterowanie pozycyjne.

Współrzędne położenia podążającego robota zostają przeliczone za pomocą funkcji trygonometrycznych tak jak przedstawiono poniżej.

$$x_F = x_L - 0.5 * \cos(\Psi_L) \quad (3.4)$$



Rysunek 3.13: Metoda śledzenia

$$y_F = y_L - 0.5 * \sin(\Psi_L) \quad (3.5)$$

Zmienne są tylko x i y oraz kąt Ψ tj. kąt obrotu wokół osi z, dlatego to właśnie ich dotyczą obliczenia. Aktualny kąt Ψ , lokalizacja x oraz y jednostki wiodącej są przekazywane do śledzącego i na tej podstawie wykonywane są kalkulacje. Natomiast wysokość na jaką są wzniesione jest jednakowa dla obu robotów latających i stała. Aby dystans pomiędzy UAV się nie zmieniał został on ustalony na konkretną wartość tj. 0,5 m.

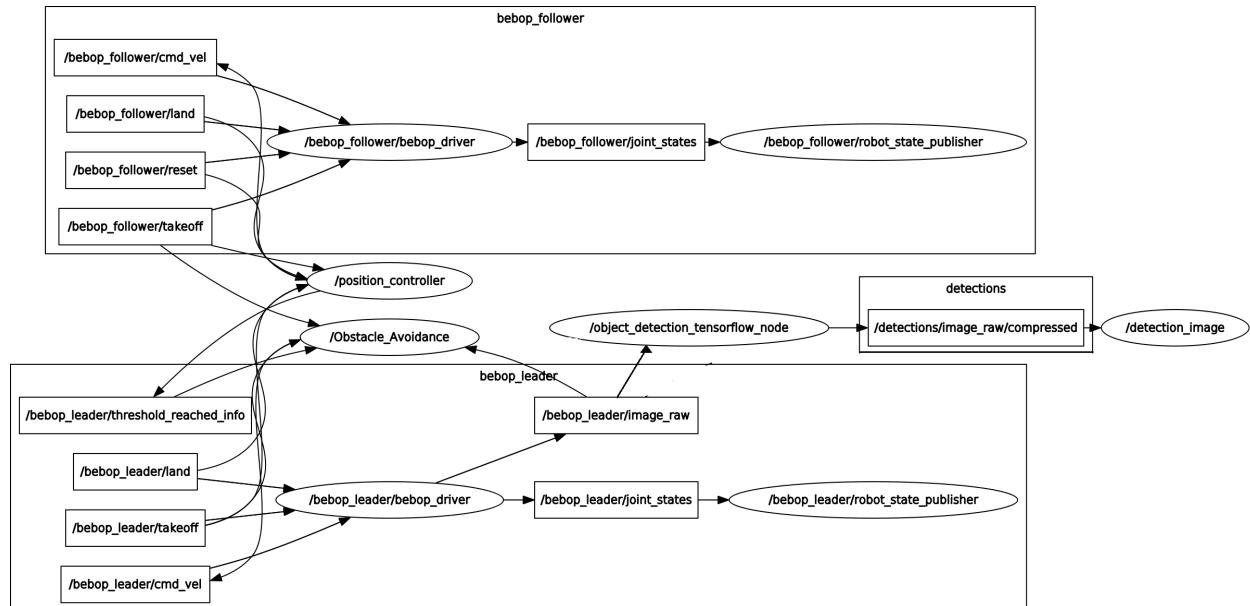
Jak już wspomniałam powyżej do zaimplementowania algorytmu podążania został zmodyfikowany przez mnie pakiet *position_controller*. Dodana została do niego subskrypcja następujących tematów :

- */bebop_leader/takeoff* - informacja odnośnie wznoszenia Bebopa
- */gazebo/default/pose/info* - aktualna pozycja lidera w Gazebo
- */bebop_leader/land* - informacja odnośnie wylądowania Bebopa

Funkcja obsługująca pierwszy z powyższych zmienia wartość zmiennej wykorzystanej do rozpoczęcia funkcjonalności śledzenia. Inicjowane jest to zadanie jeśli dwie jednostki UAV są podniesione. Pod takie zastosowanie w pakiecie *Optical-Flow-based-Obstacle-Avoidance* zostało dodane ograniczenie w pętli głównej o rozpoczęciu działania jeśli oba roboty latające są w powietrzu. Odpowiada za to konkretna zmienna w pakiecie do omijania przeszkód określająca zastosowanie dla dwóch UAV. Gdy ma ona wartość *true* oraz w tematach */bebop_leader/takeoff* i */bebop_follower/takeoff* pojawi się wiadomość to dopiero rozpoczyna się algorytm omijania przeszkód wraz ze śledzeniem robota wiodącego. Rola obsługi drugiego z podanych powyżej sprowadza się do przeliczenia wartości kąta z kwaterionów na kąt Eulera Ψ . Natomiast obsługa ostatniego zmienia wartość wykorzystaną do zakończenia działania zarówno omijania przeszkód jak i śledzenia. Dodana funkcjonalność podążania określona jest w metodzie o nazwie *follow*, która przelicza wartości współrzędnych i przekazuje je jako cel dla robota śledzącego. Kolejną napisaną przez mnie metodą jest *save_data*, która umożliwia wygenerowanie plików csv, w których zawarte są aktualne położenia leadera oraz followera w czasie wykonania zadania.

Posłużyły one do stworzenia wykresów. Kolejną modyfikacją z mojej strony strony było dodanie metody o nazwie *position_leader_threshold_room*, której zasada działania została przedstawiona w sekcji 3.3. W celu przetestowania algorytmu posłużyłem się napisanym pakietem do sterowania za pomocą klawiatury, który został wspomniany w sekcji 4.4.

Graf węzłów oraz tematów po uruchomieniu wszystkich potrzebnych pakietów do zestawienia omijania przeszkód przez grupę UAV został przedstawiony poniżej.



Rysunek 3.14: Graf węzłów i tematów w zastosowaniu algorytmu dla grupy UAV

Algorytm autoladowania na platformie

Milena Molska

4.1 Opis wybranej metody

Wybierając algorytm autoladowania na platformie, kluczowym kryterium było korzystanie tylko i wyłącznie z elementów, które są częścią jednostki UAV. Chciałam uniknąć podawania dodatkowych danych z zewnętrznych urządzeń, gdyż stanowiło by to ograniczenia dla możliwości zastosowania tego manewru. Dlatego wybrałam metodę opartą na głębokim uczeniu ze wzmacnieniem (DRL - ang. Deep Reinforcement Learning)[18], autorstwa Alejandro Rodriguez-Ramos z Politechniki w Madrycie. Technika ta nie potrzebuje dokładnej estymacji stanu na podstawie np. systemu Motion-Capture, lecz określenia pozycji robota latającego względem platformy na podstawie znaczników ArUco umieszczonych na niej. W celu lepszego wyjaśnienia zastosowanej metody, zdecydowałam się na przedstawienie teorii jej działania.

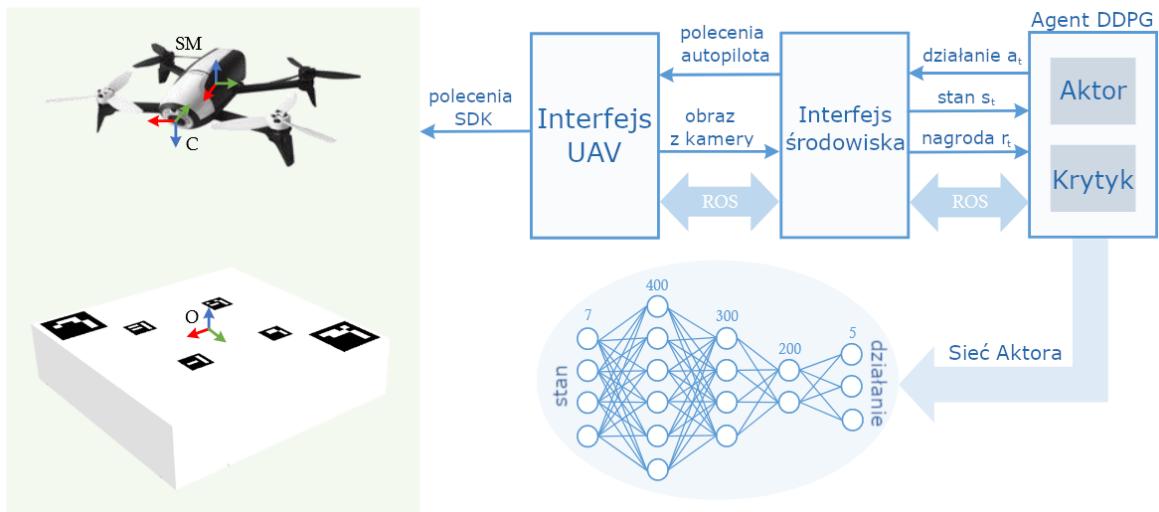
Idea RL (ang. Reinforcement Learning) opiera się na agencie, którego nauka odbywa się we wzajemnie oddziałyującym otoczeniu. Pod wpływem jego ruchów środowisko może zmieniać stany oraz dostarczać informację odnośnie nagrody (ang. reward), za pomocą której oszacowana zostaje jego skuteczność. Celem ucznia w ogólnym przypadku jest znalezienie najlepszego możliwego ruchu dla danego stanu w każdym taktcie, które maksymalizuje otrzymaną nagrodę. Nagrodą jest wartość liczbową, która rośnie podczas wykonywania działań przybliżających agenta do osiągnięcia celu. Powszechnie stosowaną techniką uczenia ze wzmacnieniem jest Q-learning, który ma na celu oszacowanie optymalnej wartości funkcji celu, prowadzącej do najkorzystniejszych działań agenta. W kontekście DRL osiągnięto możliwość stałego uczenia funkcji używając głębokich sieci neuronowych. Ważnym elementem w tym przypadku okazało się dołączenie bufora powtórki doświadczenia, w celu przewyciężenia korelacji próbek. Zostało to połączone ze schematem aktor-krytyk (ang. Actor-Critic Paradigm), co w rezultacie dało nową technikę DDPG (ang. Deep Deterministic Policy Gradients). Jej ogromną zaletą jest zdolność do uczenia się od ciągłych stanów i przestrzeni działania. To właśnie ta metoda została wykorzystana do wyuczenia agenta autoladowania na platformie.

Koncepcja zakłada wykonanie manewru lądowania, bazując tylko i wyłącznie na pozycji jednostki UAV względem platformy, obliczonej na podstawie znaczników ArUco. Stan w danym momencie $s_t \in S$ (gdzie $S \in [-1, 1]$) definiowany jest w lokalnym układzie jednostki UAV (SM na rysunku 4.1) poprzez pozycje x, y, z robota latającego względem platformy, różnicę w pozycji x i y względem poprzedniego taktu (\dot{x}, \dot{y}), znormalizowany kąt obrotu yaw względem platformy (ψ) oraz binarny stan czujnika kontaktowego zamieszczonego na niej ($C \in [0, 1]$). Sformułowanie przestrzeni stanu przedstawiono w (4.1).

$$S = \{x, y, z, \dot{x}, \dot{y}, \psi, C\} \quad (4.1)$$

Natomiast przestrzeń działań $A \in [-1, 1]$ opisana równaniem (4.2), zawiera w sobie prędkość kątową względem osi x, y, z o kąt roll, pitch i yaw ($\phi, \theta, \dot{\psi}$) oraz siłę nośną jednostki UAV (\dot{z}), a także zdolność agenta do odcięcia napędów (τ).

$$A = \{\dot{\psi}, \theta, \phi, \dot{z}, \tau\} \quad (4.2)$$



Rysunek 4.1: Architektura wybranego systemu autolądowania opartego na głębokim uczeniu ze wzmacnieniem [18]. SM to układ lokalny jednostki UAV, którego początek to jej środek ciężkości, C - układ odniesienia przedniej kamery Bebopa, natomiast O to układ lokalny platformy, znajdujący się na środku jej powierzchni.

Ważnym aspektem uczenia ze wzmacnieniem jest konstrukcja funkcji nagrody. Autorzy omawianego systemu obrali za cel stopniowe nagradzanie płynnego i bezpiecznego lotu. Stąd funkcja nagrody r zawiera karę za przekroczenie maksymalnych wartości pozycji robota latającego względem platformy w osiach x, y, z (r_1), wartości zachęcające agenta do odcięcia napędów na bezpiecznej wysokości UAV (r_2 i r_3) oraz komponent, informujący o postępach w danym takcie względem poprzedniego (r_4). Matematyczne sformułowanie

funkcji nagrody przedstawiono w (4.3), (4.4) i (4.5).

$$r = \begin{cases} r_1 = -100 & \text{jeśli } x > x_{max}, y > y_{max} \text{ lub } z > z_{max} \\ r_2 = -50 & \text{jeśli } \tau > 0.8 \text{ i } (z^\tau > z^{\tau_{max}} \text{ lub } z^\tau < 0) \\ r_3 = 100e^{k(z^{\tau^*} - z^\tau)} & \text{jeśli } \tau > 0.8 \text{ i } (z^\tau < z^{\tau_{max}} \text{ lub } z^\tau > 0) \\ r_4 & \text{w przeciwnym razie} \end{cases} \quad (4.3)$$

$$r_4 = shaping[t] - shaping[t - 1] \quad (4.4)$$

$$shaping[t] = -\alpha_1 \sqrt{x^2 + y^2 + z^2 + \psi^2} - \alpha_2 \sqrt{\dot{x}^2 + \dot{y}^2} - \alpha_3 \sqrt{\dot{\psi}^2 + \theta^2 + \phi^2 + \dot{z}^2} \quad (4.5)$$

$\alpha_1, \alpha_2, \alpha_3$ i k to stałe, które dobrano eksperymentalnie o wartościach odpowiednio: 100, 10, 1, 1.55. Natomiast z^τ , $z^{\tau_{max}}$ oraz z^{τ^*} reprezentują kolejno aktualną, maksymalną oraz pożądaną wysokość odcięcia napędów przez jednostkę UAV.

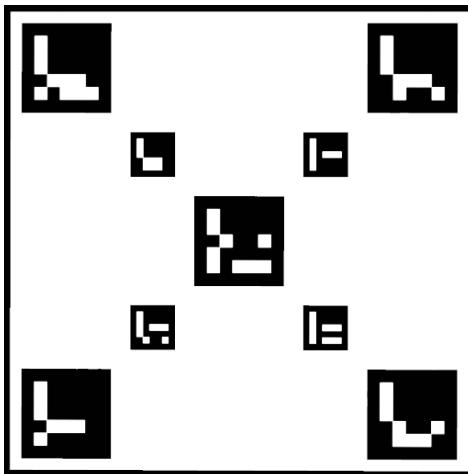
Podczas implementacji omawianej metody, wykorzystałam gotowy pakiet o nazwie drl-landing [17], napisany pod oprogramowanie Aerostack. Architektura zastosowanego systemu została przedstawiona na rysunku 1.4. W jej skład wchodzi wspomniany agent DDPG, wraz z podłączoną siecią neuronową. Zbudowana jest ona z trzech ukrytych warstw sieci skierowanych - po 400, 300 i 200 neuronów każda - oraz warstwy wejścia i wyjścia (odpowiednio po 7 i 5 neuronów). Autorzy pakietu zamieścili również wyuczonego dla 4800 przypadków agenta oraz implementację pozwalającą na korzystanie z niego. Kolejnym elementem jest interfejs środowiska odpowiedzialny za:

- wykrywanie znaczników ArUco oraz obliczanie położenia UAV względem platformy na ich podstawie,
- wymianę informacji między agentem a robotem latającym (udostępnianie stanu jednostki UAV oraz interpretacja działań podjętych przez agenta),
- resetowanie środowiska przed rozpoczęciem kolejnego epizodu.

Komunikacja między poszczególnymi elementami odbywa się przez system ROS omówiony w rozdziale 1.2.

4.2 Dostosowanie świata symulacji

Według założeń, jednostka UAV miała lądować na ruchomej platformie pokrytej znacznikami ArUco. Dlatego konieczne było dodanie modelu o odpowiednich wymiarach i rozmieszczeniu markerów. Po skontaktowaniu się z autorami pakietu drl-landing [19], otrzymałam od nich gotowy model platformy zawierający tekstury, pliki skryptowe i plik *.sdf, zgodny z opisem zawartym w ich kodzie (tj. o rozmiarach 1,2 x 1,2 x 0,3 m). Rozmieszczenie markerów zostało przedstawione na rysunku 4.2. Na potrzeby późniejszych testów symulacyjnych, zdecydowałam się na dołączenie platformy zarówno do pustego świata, jak i do tego z pomieszczeniem.



Rysunek 4.2: Rozmieszczenie znaczników ArUco na platformie

Kolejnym krokiem było napisanie programu, który poruszałby platformą po zadanej trajektorii i z określona prędkością. W tym celu posłużyłam się pluginem [1] - biblioteką C++, uruchamianą podczas startu symulatora Gazebo. Ma ona dostęp do jego API, co pozwala na realizację zadań takich jak: przesuwanie obiektów, dodawanie i usuwanie modeli, odczyt danych z czujników itd. Wykorzystałam do tego klasę PoseAnimation z biblioteki common [20]. Jej argumentami są: nazwa, czas trwania animacji (wyrażony w sekundach) oraz wartość logiczna określająca zapętlenie ruchu. Aby uzyskać trajektorię, należy określić punkty (podając współrzędne w układzie globalnym XYZ i rotacje modelu za pomocą kwaternionów) oraz momenty czasowe im odpowiadające.

4.3 Implementacja algorytmu autolądowania

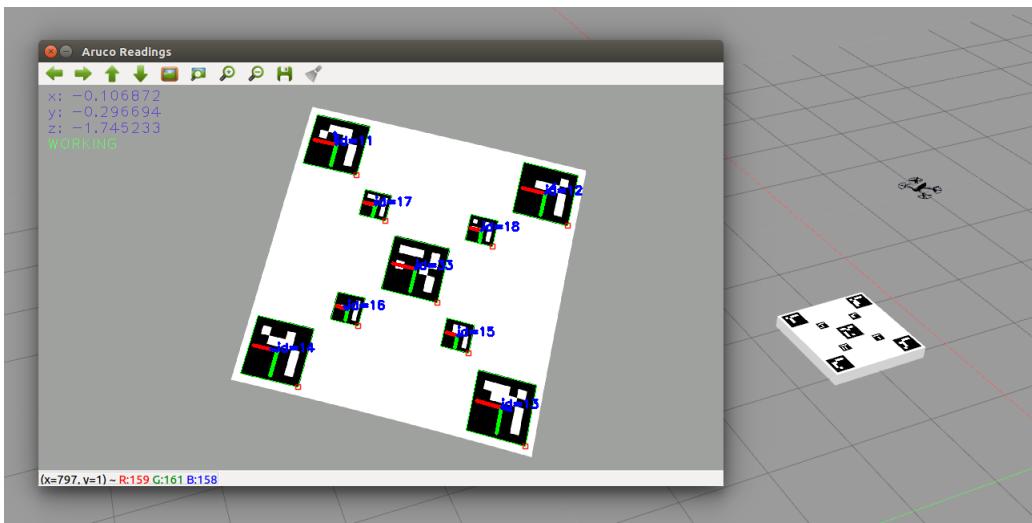
W celu wykorzystania agenta DDPG w symulatorze Sphinx, posłużyłam się opisany w rozdziale 4.1 pakietem drl-landing. Jednak jak już wcześniej wspomniałam, został on napisany pod system Aerostack, dlatego musiałam wprowadzić kilka znaczących zmian w jego kodzie.

Pierwszą rzeczą było dodanie subskrybcji odpowiednich tematów, które umożliwiałyby odczyt informacji niezbędnych do wykonania manewru i dokumentacji wyników:

- */bebop_leader/image_raw* - obraz z przedniej kamery Bebopa, publikowany przez sterownik bebop_autonomy.
- */bebop_leader/odom_conv* - dane dotyczące aktualnej pozycji UAV, publikowane przez autorski skrypt bebop_dronemsgsros opisany w dalszej części rozdziału. Temat ten został dodany w celu dokumentacji trajektorii lotu jednostki UAV.

Następnie, przetworzone dane musiały zostać publikowane do tematów:

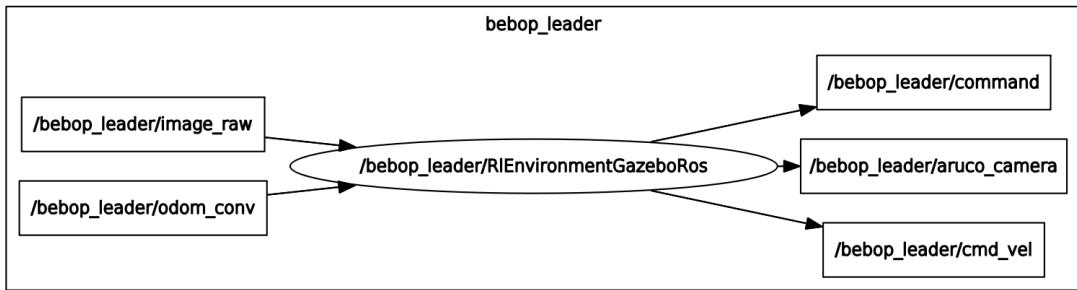
- `/bebop_leader/cmd_vel` - sterowanie kątami roll, pitch i yaw oraz siłą ciągu. Subskrybowany przez `bebop_autonomy`.
- `/bebop_leader/aruco_camera` - publikowanie obrazu z kamery przetworzonego przez funkcję wraz z naniesionymi informacjami odnośnie wykrycia znaczników, obliczonej odległości jednostki UAV od platformy oraz położeniu i orientacji markerów za pomocą narysowanych układów współrzędnych.
- `/bebop_leader/command` - publikowanie informacji związanych z komendami TAKE OFF i LAND. Ze względu na komplikację związaną z zapętlaniem, omówiony w dalszej części tekstu.



Rysunek 4.3: Wykrycie platformy przez jednostkę UAV

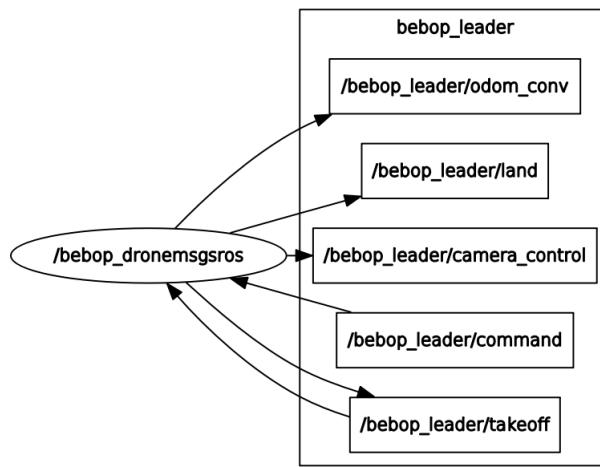
Istotne zmiany związane z publikowaniem wiadomości nastąpiły przy tematach dotyczących sterowania jednostką UAV. Z poziomu agenta odbywa się ono poprzez podawanie wartości kąta roll, pitch i yaw oraz siły nośnej. W oryginalnej wersji programu, informacje te publikowano do 3 różnych tematów, korzystających z typów wiadomości pakietu DroneMsgsROS [21] tj. `dronePitchRollCmd`, `droneDYawCmd` oraz `droneDAltitudeCmd`. W wykorzystanej przeze mnie bibliotece `bebop_autonomy`, wszystkie wymienione dane można wysyłać w jednym temacie o typie wiadomości `geometry_msgs/Twist`.

Kolejnym problemem było publikowanie komend takich jak TAKE OFF i LAND. Dla symulatora Aerostack jest to realizowane za pomocą jednego tematu o typie wiadomości: `droneCommand` [21], który zawiera wartości liczb całkowitych od 0 do 9, odpowiadające konkretnym polecaniom. Dla zastosowanego sterownika odbywa się to poprzez oddzielne tematy: `/bebop_leader/takeoff` oraz `/bebop_leader/land`, publikujące puste wiadomości (ang. `empty_msgs`). Biorąc pod uwagę zapętlenie misji w oryginalnym kodzie (po wyładowaniu Bebop startował i wykonywał w kółko wyuczone zadanie), zdecydowałem się na dodanie tych funkcjonalności w zewnętrznym skrypcie, aby możliwe było jednorazowe wykonanie manewru. Graf programu środowiska został przedstawiony na rysunku 4.4.



Rysunek 4.4: Graf tematów użyty przez środowisko RIEnvironmentGazeboROS

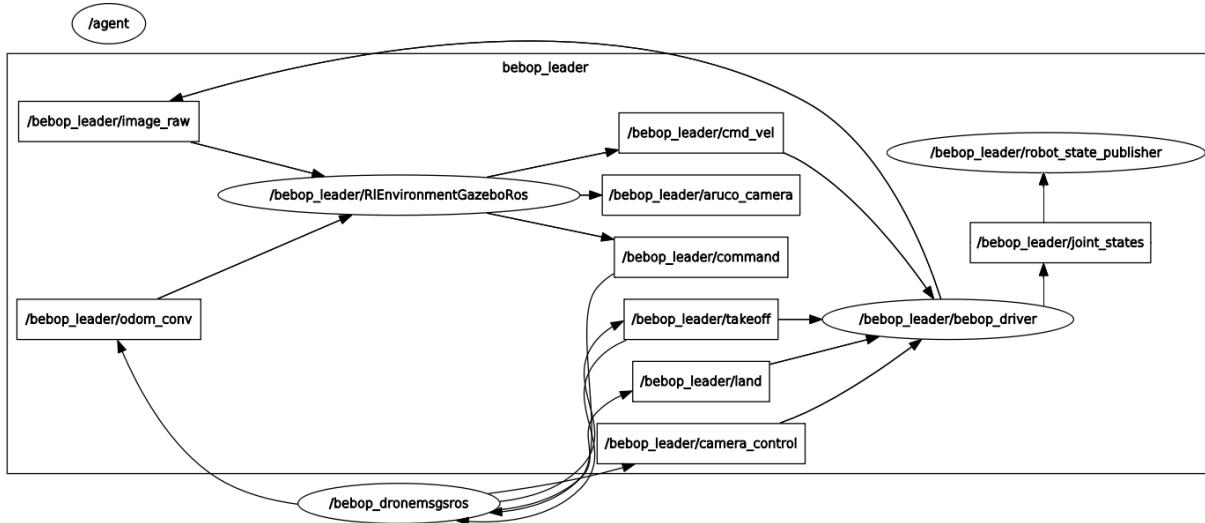
Początkowo powstanie oddzielnego skryptu uruchamianego przed startem agenta miało na celu zmianę pozycji kamery Bebopa. W wytycznych zawartych w artykule [8], miała ona być skierowana w dół. Posłużyłem się tutaj tematem `/bebop_leader/camera_control` i jako kąt obrotu w osi y, podałem -90 stopni. Jednak przy zaistniałych problemach z rozbieżnością typów wiadomości, zdecydowałem się również na implementację polecenia *take off* oraz *land*. Idea działania owego skryptu polega na tym, że po jego uruchomieniu jednostka UAV startuje i jednocześnie zmienia pozycję kamery na wcześniej wspomnianą. Następnie subskrybuje temat, który z głównego programu przekazuje informacje odrębnie zadania i w chwili pojawiienia się polecenia RESET (sygnalizującego, że UAV znalazł platformę i chce wylądować) wysyła wiadomość do tematu `/bebop_leader/land`. Kolejną funkcjonalnością owego skryptu jest odczytywanie pozycji robota latającego z symulatora Gazebo i wysyłanie jej do tematu `/bebop_leader/odom_conv`. Teoretycznie pozycja jednostki UAV publikowana jest przez sterownik Bebopa, jednak częstotliwość z jaką to następuje to tylko 5 Hz. Stąd zdecydowałem się na dodanie osobnego tematu, który aktualizuje informacje odrębnie pozycji, odczytując ją z Gazebo, który publikuje dane z częstotliwością 10 Hz. Graf tematów autorskiego skryptu przedstawiłem na rysunku 4.5.



Rysunek 4.5: Graf tematów użytych przez autorski skrypt `bebop_dronemsgsros`

Podczas wstępnych testów działania programu, problemem było dziwne zachowanie

agentu w końcowej części zadania. Dla znacznej ich części, jednostka po próbie lądowania co chwilę wznała się i opadała. Wyglądało to tak, jakby na zmianę podawane były dwa polecenia: odcięcie napędów oraz ich załączenie. Po dłużej analizie kodu, okazało się, że w funkcji programu komunikującej się z agentem, brakowało wysłania informacji odnośnie potwierdzenia zakończenia manewru.



Rysunek 4.6: Graf węzłów i tematów po uruchomieniu całego oprogramowania

Graf węzłów systemu ROS po uruchomieniu biblioteki bebop_autonomy, zewnętrznego skryptu, zmodyfikowanego programu środowiska oraz agenta przedstawiony został na rysunku 4.6.

4.4 Zastosowanie algorytmu dla grupy robotów latających

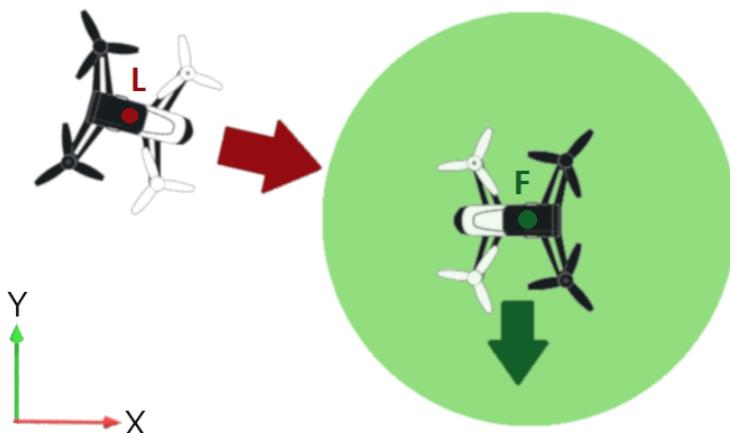
W celu bezpiecznego zastosowania omówionej metody dla więcej niż jednej jednostki UAV, niezbędne było zaimplementowanie unikania kolizji. Wybrałam scenariusz, w którym jeden robot latający ma większy priorytet (lider) i nie podejmuje żadnych dodatkowych działań, natomiast drugi (śledzący) usuwa mu się z toru lotu. Postawiłam zastosować algorytm, opisany w książce dr inż. Wojciecha Giernackiego [22] i wykorzystać do tego sterowanie pozycyjne z pakietu position_controller [23].

Dla jednostek UAV zdefiniowałam minimalny dystans, w którym mogą się wzgółdem siebie poruszać. Jest on wyznaczany między punktami L i F, ze wzoru na odległość w trójwymiarowym układzie współrzędnych:

$$|LF| = \sqrt{(x_L - x_F)^2 + (y_L - y_F)^2 + (z_L - z_F)^2} \quad (4.6)$$

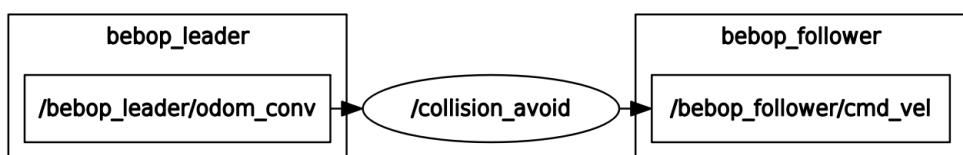
Gdy bezpieczna przestrzeń między jednostkami UAV zostaje przekroczena, następuje ruch ze strony Followera, który odsuwa się z toru ruchu. Sposób uniknięcia kolizji, opiera się

na porównywaniu prędkości lidera w globalnych osiach X i Y. Gdy jest ona większa w osi X, to śledzący wykonuje manewr w osi Y i na odwrót (rysunek 4.6). Dystans z jakim następuje unik, zależy od tego, jak bardzo przekroczona zostanie strefa. Wartość ta jest dodawana lub odejmowana od odpowiedniej współrzędnej i przekazywana jako target dla śledzącego.



Rysunek 4.7: Metoda unikania kolizji dla dwóch jednostek UAV

Jak już wcześniej wspomniałem, do uniknięcia kolizji postanowiłem wykorzystać gotowe sterowanie pozycyjne. W tym celu utworzyłem pakiet *collision_avoid*, do którego dodałem niezbędne funkcje z *position_controller*. Zaimplementowałem również subskrypcję tematu */bebop_leader/odom_conv*, zawierającego informację o współrzędnych położenia lidera w symulacji oraz funkcję odczytującą te dane. Następnie napisałem metodę o nazwie *speed_read()*, obliczającą prędkość lidera na podstawie różnicy pozycji w czasie. Opisany algorytm antykolizyjny zaimplementowałem jako metodę *collision_check()*, której warunek bezpiecznej strefy sprawdzany jest cyklicznie. Dodatkowo w celu dokumentacji trajektorii obu jednostek UAV, dodałem funkcję *save_data()*, zawierającą zapisywanie danych dotyczących pozycji XYZ w danej próbce czasu. Na rysunku 4.8 przedstawiłem graf tematów autorskiego pakietu.



Rysunek 4.8: Graf tematów autorskiego pakietu *collision_avoid*

Aby możliwe było przeprowadzenie wstępnych testów algorytmu, niezbędna okazała się możliwość swobodnego sterowania liderem. Dlatego zdecydowałem się również na zaimplementowanie poruszania jednostką UAV i jej kamerą za pomocą klawiatury. W tym celu wykorzystałem pakiet *teleop_twist_keyboard*. Pozwala on na manipulację robotami, których sterowanie odbywa się za pomocą tematu o typie wiadomości geome-

try_msgs::Twist. Postanowiłem go zmodyfikować, tworząc bazujący na oryginale pakiet o nazwie bebop_keyboard.

```
milena@milena-GV62-BRC:~/bebop_ws$ rosrun bebop_keyboard bebop_leader_keyboard.py
Reading from the keyboard and Publishing to Twist!
-----
Moving around:
  q   w   e
  a   s   d
Move vertically:
  -   up
  +   down

Basic commands:
  1 Take off
  2 Land

Camera control:
  i Up
  k Down
  j Left
  l Right
  0 Default

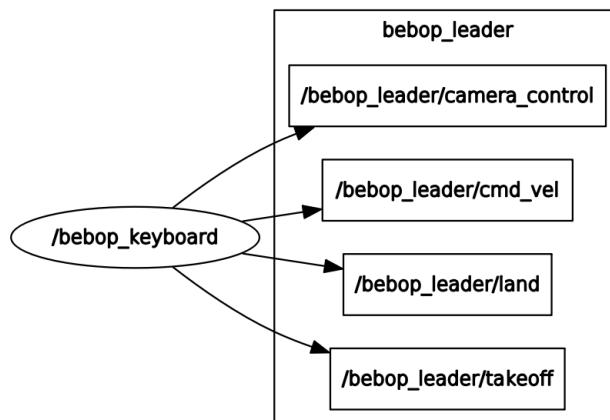
Speed linear:
n/m -> +/- 
Speed angular:
N/M -> +/- 

CTRL-C to quit

currently:      speed 0.5      turn 1.0
```

Rysunek 4.9: Sterowanie UAV za pomocą klawiatury - interfejs.

Jak przedstawiłem na rysunku 1.9, *bebop_keyboard* pozwala na swobodne sterowaniem ruchem jednostki UAV oraz zawiera podstawowe komendy takie jak *take off* i *land*. Istnieje również możliwość regulacji prędkości liniowej i kątowej. Z mojej strony dodałem sterowanie obrotem kamery góra/dół i prawo/lewo oraz przycisk powrotu do jej domyślnej pozycji. Dostosowałem również klawisze dla własnej wygody użytkowania. Graf tematów, z których korzysta pakiet sterowania klawiaturą, przedstawiłem na rysunku 4.10.



Rysunek 4.10: Graf tematów, z których korzysta pakiet bebop_keyboard

Weryfikacja proponowanych rozwiązań w eksperymentach symulacyjnych i rzeczywistym locie UAV

Z uwagi na to, że praca ta ma głównie charakter symulacyjny, testy przeprowadziłyśmy w programie Sphinx, działającym pod Gazebo i systemem ROS. Dzięki strukturze wybranych narzędzi, istnieje teoretyczna możliwość przeniesienia opracowanego rozwiązania na jednostki fizyczne, dokonując tylko drobnych poprawek w kodzie. Jednak z uwagi na przewagę testów symulacyjnych nad rzeczywistymi pod względem bezpieczeństwa, to właśnie na nie się zdecydowałyśmy.

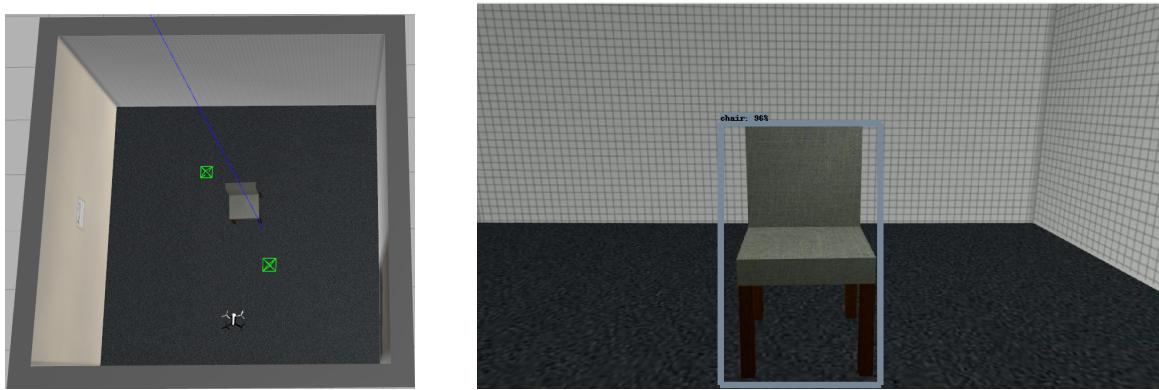
5.1 Testy rozpoznania obiektów przez sieć neuronową

Alicja Kuźniewska

Celem testów na sieciach neuronowych było sprawdzenie pewności procentowej SSD Multibox w rozpoznaniu obiektów znajdujących się na obrazie pochodząącym z kamery robota latającego, który wzleciał na wysokość o różnym stopniu oświetlenia. Przedstawione poniżej wyniki będą zestawione z wyglądem pomieszczenia, w którym odbywała się weryfikacja. Jednostka, która była wykorzystywana wzleciała na wysokość jednego metra.

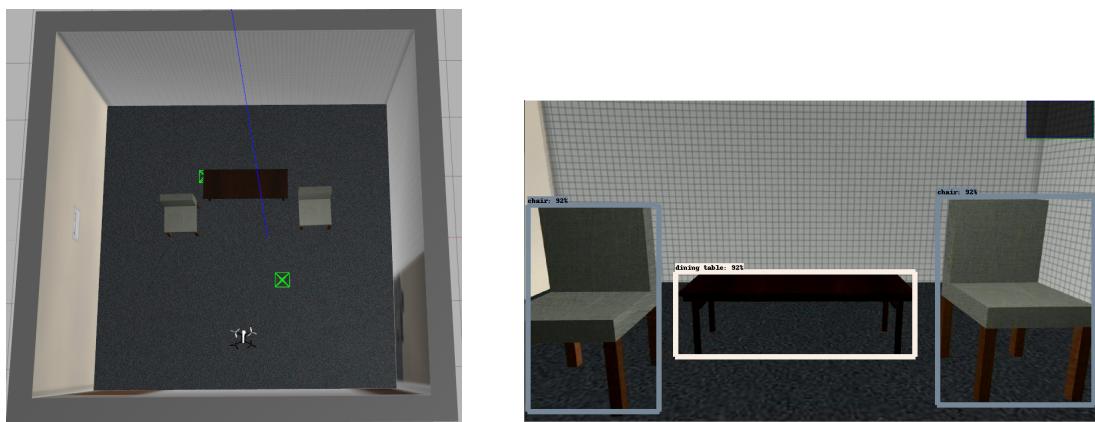
Jak widać na powyższym rysunku wynik rozpoznania to 96% pewności detekcji. Obiekt znajdujący się na obrazie został rozpoznany jako krzesło i biorąc pod uwagę co faktycznie znajduje się na widoku jest to prawidłowe rozpoznanie. W pomieszczeniu tym zostały zastosowane dwa źródła światła, jedno z nich było wzniesione na wysokość ponad jednego metra, dzięki czemu oświetlało one charakterystyczne krawędzie przedmiotu. Pozwoliło to na bezproblemową detekcję z bardzo wysokim wynikiem procentowym.

W kolejnym teście zostały użyte trzy obiekty. Biorąc pod uwagę ułożenie źródeł światła lewy oraz prawy obiekt był inaczej oświetlony. Na obrazie nie były widoczne wszystkie krawędzie przedmiotu po lewej stronie, środkowy obiekt jest celowo taki ciemny,



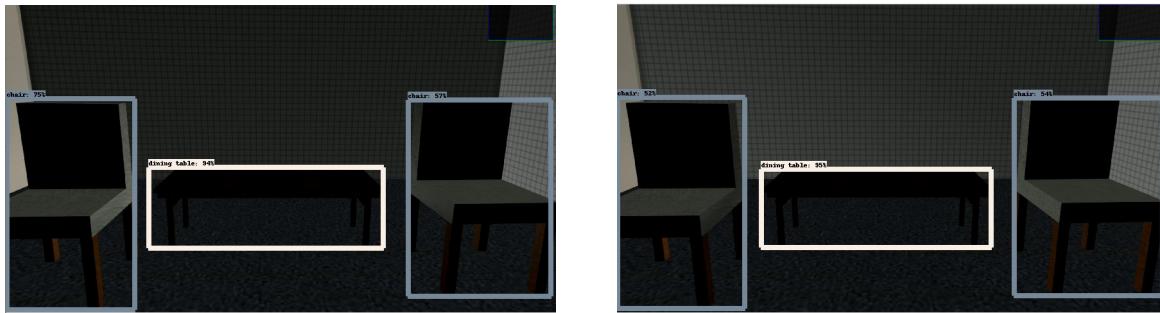
Rysunek 5.1: Wynik testu w pomieszczeniu o 2 źródłach światła

by sprawdzić czy to w znaczny sposób wpłynie na detekcję. Jednak jak widać na poniższym rysunku sieć dobrze sobie poradziła z wynikiem procentowej pewności stojącej na poziomie 92% dla wszystkich przeskódek znajdujących się na obrazie. Rozpoznanie również było słuszarne, ponieważ dwa obiekty zostały rozpoznane jako krzesła, a pozostały obiekt jako stół jadalny co jest zgodne z widokiem z kamery.



Rysunek 5.2: Wynik testu w pomieszczeniu o 2 źródłach światła

W następnych weryfikacjach zostało usunięte jedno źródło, a pozostałe zostało opuszczane, aby nie oświetlało dużej powierzchni. Nie została dokonana zmiana ułożenia obiektów. Poniżej znajdują się wyniki testów, na prawym rysunku oświetlenie zostało ułożone przed obiektami w prawym rogu, natomiast na lewym za obiektami po lewo. Wyniki prezentują się następująco :



Rysunek 5.3: Wynik testu w pomieszczeniu z jednym źródłem oświetlenia

Przy słabszym źródle sieć juz niestety nie radzi sobie tak dobrze jak w przypadku dwóch ostatnich testów, ale nadal wyniki są obiecujące. Detekcja wszystkich przedmiotów została wykonana prawidłō, obramowania wokół obiektów znajdują się tam gdzie powinny, co zdawało się, że będzie trudne przy takim stopniu oświetlenia. Pewności procentowe są wyższe na prawym rysunku. Po lewej jest to 75%, dla środkowego 94%, niestety prawy przedmiot ma niskie wskazanie, 57%. Inaczej wygląda to dla drugiego wyniku, pewność na środku pozostaje na takim samym poziomie. Jednak pozostałe obiekty mają pewność, która wynosi nieco ponad 50%.

Dla porównania poniżej zamieszczam wynik testu, dla pustego świata z jedną przeszkodą, obraz jest jasny, a mimo to nie udało się dokonać detekcji a co dopiero rozpoznania obiektu. Pokazuje to tylko, że dobre oświetlenie nie zawsze zapewnia dobre rozpoznanie.



Rysunek 5.4: Pusty świat z jednym przedmiotem

5.2 Omijanie przeszkód

Alicja Kuźniewska

Testy te zostały wykonane w celu weryfikacji skuteczności wybranego rozwiązania na podstawie zachowania robota latającego. Skuteczność oznacza tutaj ominięcie przeszkody przez UAV. Nie jest znane jej położenie, a wskazania opierają się na informacjach dostarczanych z przepływu optycznego. Gdy Bebop ominął zadaną przeszkodę zostało to

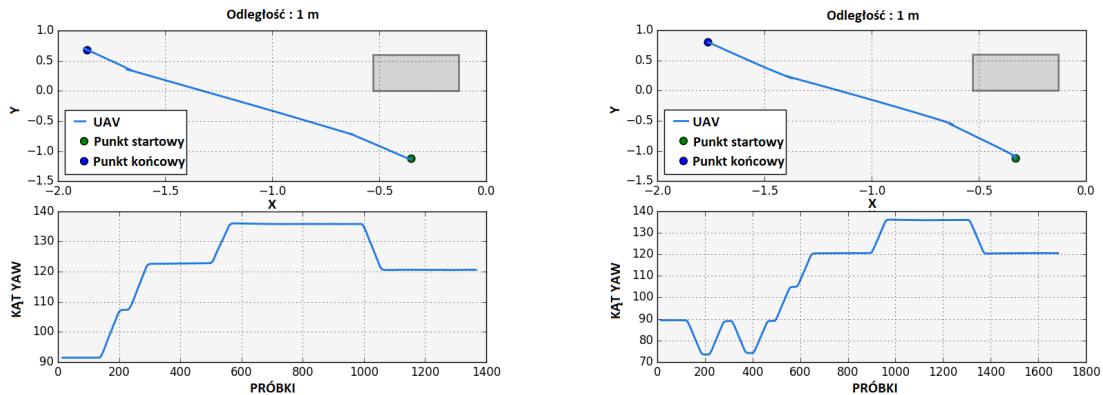
uznane za wykonane zadanie oraz test ten miał wynik pozytywny. Za ominięcie rozumiałam osiągnięcie takiego położenia przez przedmiot badań, aby dalszy ruch nie powodował zderzenia z umieszczonym wcześniej w świecie obiektem. Wynikiem testów były dane dotyczące trajektorii lotu zapisane w plikach *.csv. Na ich podstawie zostały wygenerowane wykresy. Zostały wykonane również nagrania symulacji, z których wybrane zostały umieszczone na płycie w folderze Multimedia/Testy omijania przeskód. Symulacje zostały dobrane przeze mnie ze względu na różną liczbę przedmiotów w wykorzystywanym świecie. Zostały wykorzystane różne warianty symulacyjne. Sekcje 5.2.1-5.2.3 zawierają opisy prób dla :

- pomieszczenia z jednym obiektem w różnych odległościach od przedmiotu,
- pustego świata z dwoma przeskodami,
- pustego świata z trzema przedmiotami.

Założeniem dla wszystkich testów było wzniesienie UAV na wysokość 1 m. Przeszkody znajdowały się w takiej odległości, aby przepływ optyczny miał szanse na inne wskazania niż 0, tzn. UAV musiał być umieszczony z odpowiednim dystansem, aby w dostarczanym do obliczeń źródle widoku nie znajdował się jednobarwny obraz. Spowodowałoby to zerowe wskazania, a co za tym idzie brak detekcji przedmiotu. Są to ograniczenia, które musiały być wzięte pod uwagę. Momentem zakończenia danej weryfikacji było wylądowanie wymuszone ręcznie gdy jednostka znalazła się już w takiej lokalizacji, aby dalszy ruch nie spowodował kolizji z obiektem. Brana była pod uwagę współrzędna y, nie było ograniczenia współrzędnej x na jakiej znalazł się robot latający. Jeśli wartość y na jakiej znajdował się przedmiot badań była większa od wartości tej samej współrzędnej przeskody to zadanie to było przeze mnie uznane za wykonane. Wynikami prób będą wykresy zależności y(x) oraz przedstawienie zmiany kąta yaw w czasie trwania ruchu.

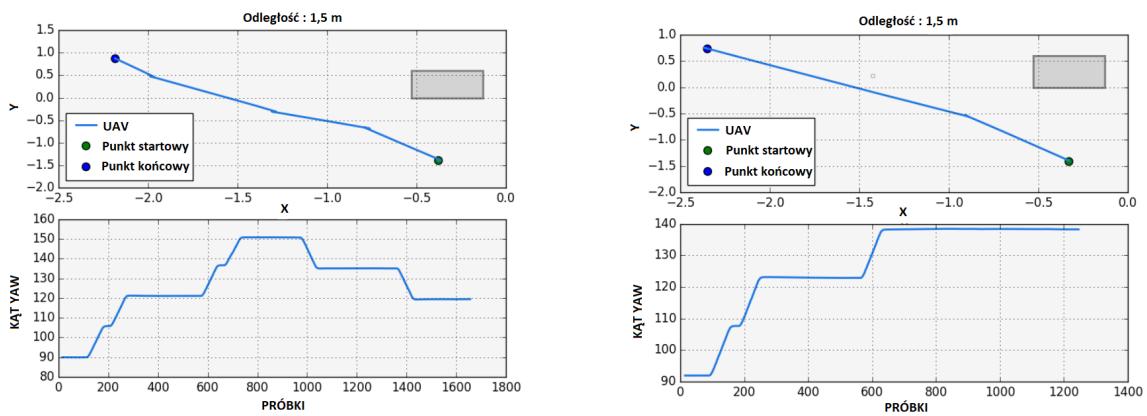
5.2.1 Testy omijania przeskód w pomieszczeniu z jednym obiektem

Pierwszym wariantem był test w stworzonym wcześniej pomieszczeniu o znanych wymiarach i opisanym w sekcji 1.2, a w nim umieszczona została jedna przeskoda. Była ona umieszczona w odległości 1.5m oraz 1m od jednostki UAV. Przyjęłam to za najprostszy scenariusz ze względu na ilość obiektów, wszystkie próby zostały zakończone sukcesem. Robot latający ominął zadany przedmiot oraz został zatrzymany w momencie gdy nieagrażała mu już żadna kolizja ze wspomnianym wcześniej obiektem. Zamieszczony poniżej rysunek przedstawiający trajektorię ruchu zostało wygenerowane od momentu wzniesienia się na odpowiednią odległość opisaną wcześniej do zakończenia przemieszczania się. Na wykresach został umieszczony prostokąt, który symbolizuje omawiany przedmiot. Poniżej zostaną przedstawione wyniki prób, wygląd scenerii został umieszczony w sekcji 3.2 na rysunku 3.7.



Rysunek 5.5: Wybrane trajektorie ruchu oraz zmiany kąta dla próby z oddaloną przeszkodą o 1 m

Kolejnym wariantem było odsunięcie przeszkody od jednostki na odległość 1,5 m, by sprawdzić czy zmieni to skuteczność wykonania zadania. Teoretycznie powinno to polepszyć wyniki, ponieważ więcej krawędzi charakterystycznych znajdowało się na obrazie, więc przepływ optyczny powinien mieć lepsze wskazania. Jak widać przebyta droga na osi X w następnej scenerii jest dłuższa niż w poprzednim przypadku. Na rysunku 5.5 po prawej stronie widać, że obroty wokół osi z w początkowej fazie ruchu często się zmieniają, jednak później dochodzi do zmniejszenia częstotliwości zmian, po lewej stronie na początku zmiany te są jeszcze częstsze. Natomiast na rysunku 5.6 po lewej stronie widać znacznie mniejszą częstotliwość zmian. Niestety nie jest to parametr, który ma swoją powtarzalność, zmiany te są nieregularne. Bierze się to z faktu, że wskazania przepływu są różne mimo bardzo podobnych obrazów. Bez zakłóceń rotacja ta powinna ustalać się w końcowych fazach ruchu gdy nie ma już obiektu na obrazie, problemy te zostaną opisane poniżej.



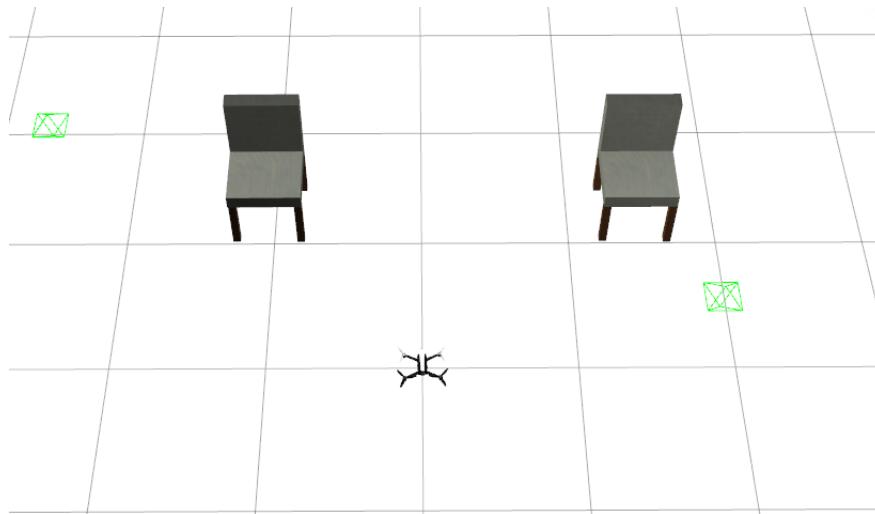
Rysunek 5.6: Wybrane trajektorie ruchu oraz zmiany kąta dla próby z oddaloną przeszkodą o 1,5 m

Jak widać na powyższych rysunkach obiekt został ominięty z sukcesem, współrzędna

y robota latającego jest większa niż ta sama koordynata przeszkody, co oznacza, że jednostka UAV znajduje się za przedmiotem, a właśnie to było celem rozwiązania. Dla zastosowanego wariantu metoda ma skuteczność bardzo wysoką, jednak jest to oparte na wartościach przepływu, więc jeśli w którymś momencie wskazania będą zerowe, a UAV będzie leciał dalej to niestety w następnych klatkach nie zostanie ona wykryta, a co za tym idzie ominięcie nie zakończy się sukcesem. W scenerii, w której były prowadzone wyżej przedstawione testy była siatka na ścianie na przeciwko jednostki UAV wprowadzało to bardzo dużo zakłóceń, ze względu na to, że podczas ustalania krawędzi charakterystycznych pod uwagę były również brane jej krawędzie. Ze względu na to w fazie ruchu gdzie przeszkoda nie znajduje się już na obrazie nadal następują tak częste zmiany kąta yaw oraz wydłużenie toru ruchu w osi X. Gdy Bebop na obrazie miał tylko jednolitą ścianę takie zakłócenia nie miały miejsca, jednak sceneria była odwzorowaniem sali w rzeczywistości, więc to nie mogło być wyeliminowane.

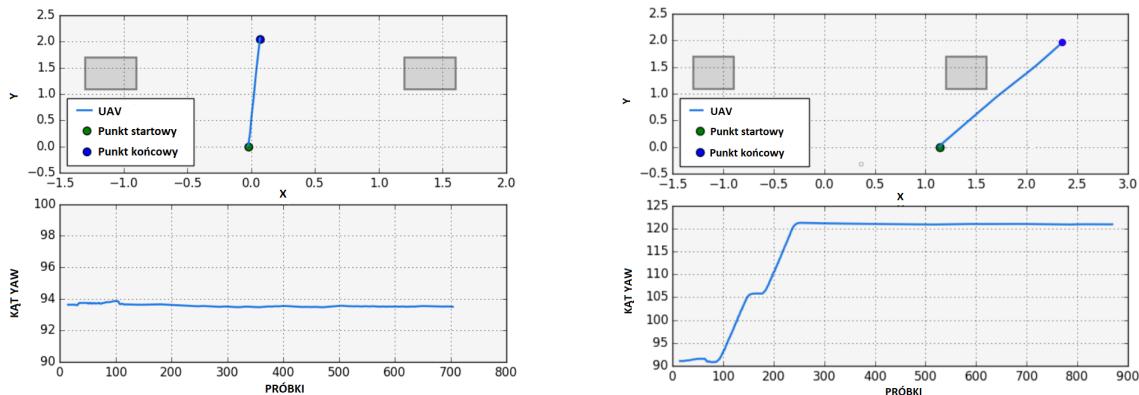
5.2.2 Świat z dwoma przeszkodami

Biorąc pod uwagę przedstawione wcześniej problemy zdecydowałam się na testy w scenerii, która nie generowałaby takich ograniczeń, dlatego został przeze mnie wykorzystany pusty świat z symulatora Sphinx, w którym umieściłam dwa obiekty oddalone od siebie o zadaną odległość, która się nie zmieniała. Wariantami prób w takim przypadku był różny punkt startu jednostki latającej. Poniżej przedstawiam również jak wyglądała sceneria testów przeprowadzonych w tej sekcji.



Rysunek 5.7: Wykorzystany świat dla dwóch obiektów

Na przedstawionych poniżej wynikach widać, że gdy któryś z przedmiotów został wykryty następowało jego ominięcie.

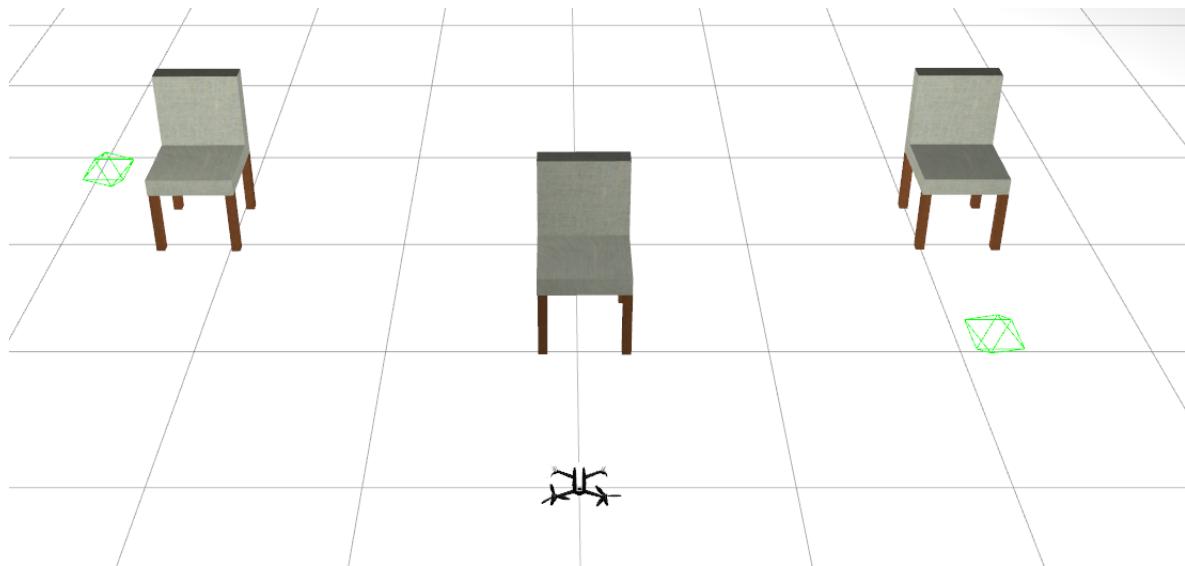


Rysunek 5.8: Wybrane trajektorie ruchu oraz zmiany kąta dla omijania dwóch przeszkód

Jak widać na powyższym rysunku w takim scenariuszu dużo lepiej przedstawiają się wyniki. UAV wykrywa przeszkodę oraz omija ją, ale nie ma dodatkowych źródeł generujących przepływ optyczny, dzięki czemu obroty wokół osi z po ominięciu wyrównują się i już nie następuje ich zmiana. Tylko w początkowej fazie ruchu rotacja ta występuje, co jest przewidzianym zachowaniem oraz w pełni uzasadnionym. Biorąc pod uwagę przedstawione wykresy widać, że w tym wypadku omijanie udaje się oraz przebyta droga w osi x jest znacznie krótsza. Na pierwszym widoczne jest, że nie została wykryta żadna przeszkoda, co jest prawidłowym postępowaniem, ponieważ nie została ona umieszczona w miejscu, w którym mogłyby nastąpić rozpoznanie. W takim wypadku UAV polecał przed siebie z tym samym kątem, w którym był na początku. Na drugim widoczne jest, że w pierwszej fazie ruchu następują obroty. Została wtedy dokonana detekcja obiektu, następnie gdy orientacja się zmieniała następowało ponowne wykrycie. Działo się tak aż do momentu, w którym jednostka była w takim położeniu, że przedmiot nie znajdował się już na obrazie lub obliczany przepływ miał zerowe wskazania. W obu przypadkach współrzędna y robota latającego jest większa niż ta sama koordynata omawianego obiektu, co oznacza, że został on ominięty.

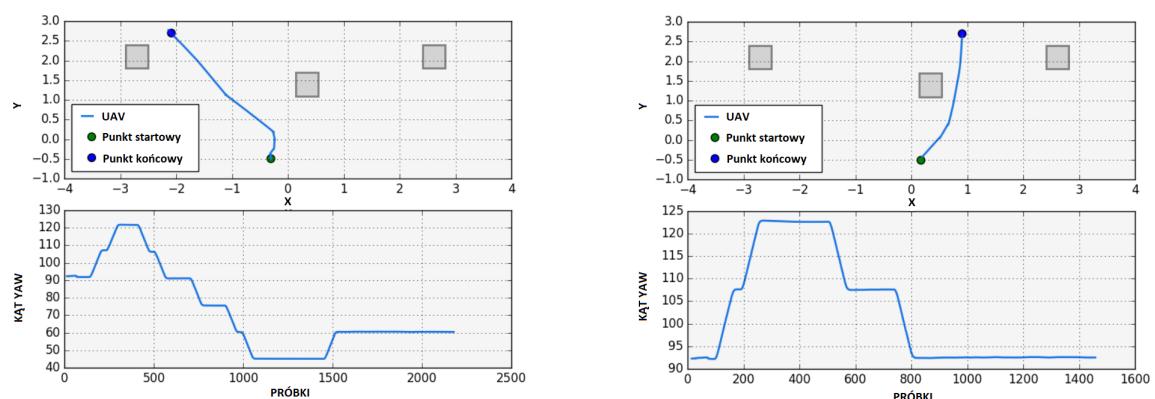
5.2.3 Świat z trzema obiektyami

Następnym wariantem, w którym zostały przeprowadzone testy był tak jak w poprzednim przypadku pusty świat, jednak zostały w nim umieszczone trzy przeszkody. Dwie bardziej oddalone i po przeciwnych stronach, a pozostała w pewnej odległości od jednostki latającej. Tutaj także zostały wykorzystane różne pozycje startowe w celu pokazania różnych możliwości. Poniżej przedstawiam scenerię, w której zostały przeprowadzone próby.



Rysunek 5.9: Wykorzystany świat dla trzech obiektów

Poniżej przedstawiam wyniki testów dla tego wariantu, widać na nich, że najbliższa przeszkoda została wykryta oraz następuje jej ominięcie. Podczas jej omijania widać, że wykrywane są kolejne w taki sposób aby nie nastąpiła kolizja.



Rysunek 5.10: Wybrane trajektorie ruchu oraz zmiany kąta dla omijania trzech przeszkód

Na powyższym rysunku widać, że w początkowej fazie ruchu doszło do detekcji przeszkody najbliższej, kąt yaw się zmienia aż do momentu gdy jednostka nie znajdzie się

w takiej orientacji, że dalszy ruch nie będzie powodował kolizji. Zauważalne jest, że w tym scenariuszu jest więcej źródeł generujących przepływ optyczny, co jest widoczne w wykorzystywanym świecie. Obroty jednostki ustają, gdy na obrazie nie będzie znajdowała się przeszkoda lub wskazanie przepływu są zerowe. W obu trajektoriach widać, że współrzędna y UAV jest większa od tej samej koordynaty wszystkich znajdujących się w świecie przeskódeł. Zadanie zostało wykonane prawidłowo. Na prawym wykresie zauważalne jest, że nastąpiło dużo mniej zmian orientacji niż na prawym. Jednak w obu przypadkach w końcowej fazie ruchu wartość kąta się ustala i jednostka porusza się bez zmian orientacji. Wyniki prezentują się dużo korzystniej niż w przypadku pierwszego wariantu z pomieszczeniem, niestety dalej nie zawsze przeszkoda jest wykrywana. Jednostka musi znaleźć się w odpowiednim położeniu oraz orientacji, aby doszło do detekcji charakterystycznych krawędzi, a tylko takie wykrycie może sprawić, że generowane będą niezerowe wartości przepływu.

Podsumowując powyższe próby metoda przepływu optycznego posiada wiele ograniczeń. Efektywniej wykrywa przedmioty w większych odległościach, a co za tym idzie jest lepsza do zastosowania w większej przestrzeni. Ze względu na ograniczenia czasowe nie zdecydowałam się na zastąpienie tego rozwiązania inną, jednak należałyby to zrobić w przyszłości.

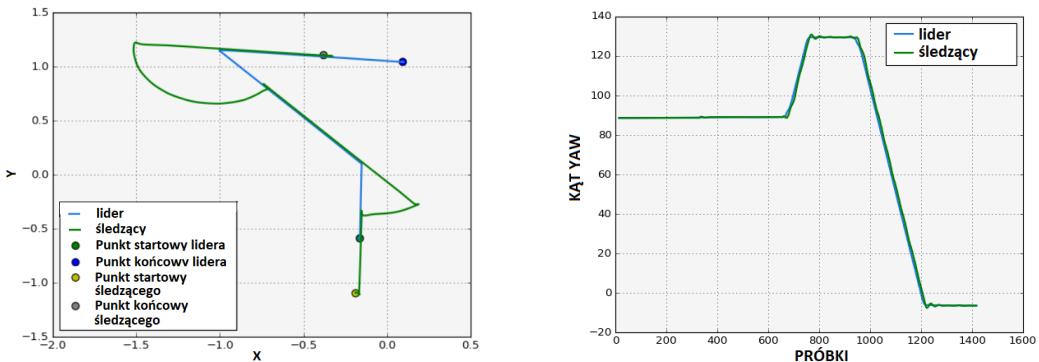
5.3 Śledzenie jednostki wiodącej

Alicja Kuźniewska

W celu sprawdzenia poprawnego działania algorytmu śledzenia lidera, który opisałam w sekcji 3.4 zdecydowałam się na następujące warianty.

1. Wymuszenie śledzenia w pustym pomieszczeniu za pomocą ręcznego sterowania liderem przy użyciu pakietu *bebop_keyboard*, który został opisany w sekcji 4.4.
2. Umieszczenie dwóch UAV w pomieszczeniu z przeszkodą. W sytuacji tej lider wykonuje manewr omijania przeszkody, a śledzący go śledzi z zadanym dystansem.
3. Umieszczenie dwóch UAV w pustym świecie z trzema przeszkodami. W sytuacji tej lider wykonuje manewr omijania przeszkody, a śledzący go śledzi z zadanym dystansem.

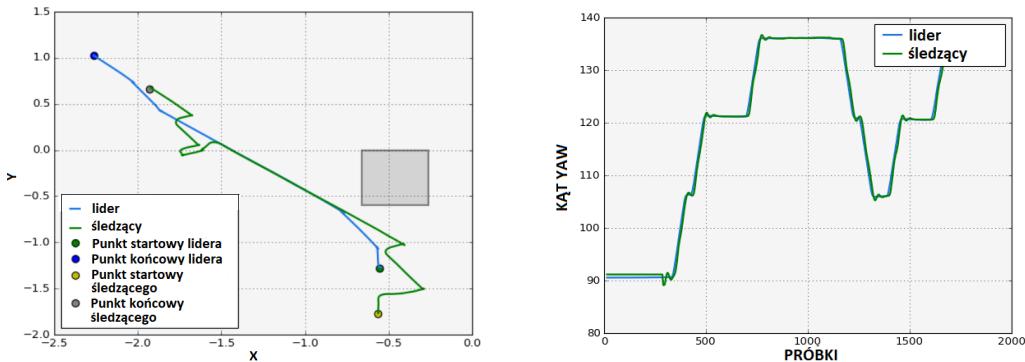
Pierwszy test miał na celu sprawdzenie działania algorytmu, bez wykorzystywania wcześniej stworzonego rozwiązania omijania. Miał to na celu sprawdzenie czy jednostka śledząca porusza się tak jak powinna, dobiera odpowiednie kąty i przelicza odpowiednio swoje położenie względem lidera. Wybrane nagrania symulacji zostały umieszczone na płycie CD w katalogu *Multimedia/Testy omijania*.



Rysunek 5.11: Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia

Bazując na powyższych wykresach można stwierdzić, że dla wymuszenia z poziomu klawiatury śledzenie działa tak jak powinno. Na początku jednostka śledząca jest odsunięta od wiodącej o zadaną odległość. Gdy kąt yaw lidera zmienia się to na taką samą wartość zmienia się yaw śledzącego. Podczas tej zmiany widoczne jest, że dla jednostki śledzącej obliczane jest jej położenie, aby przemieszczenie miało taki charakter jak ruch po okręgu, w tym wypadku środkiem tego okręgu jest jednostka wiodąca.

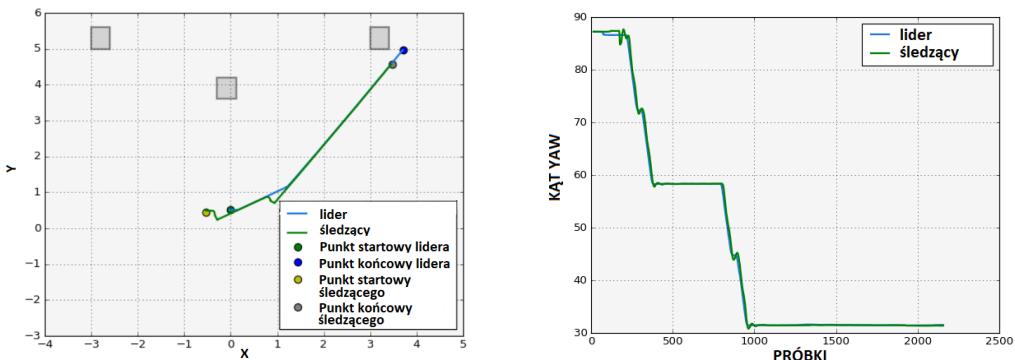
Kolejnym testem było sprawdzenie czy śledzenie działa prawidłowo gdy lider wykonuje manewr omijania przeszkody. Można zauważyć, że gdy przez wiodącą jednostkę zostaje wykryta przeszkoda i obraca się on w odpowiednią stronę to śledząca odpowiednio oblicza swoje położenie w jakim ma się znaleźć i podąża za liderem. Zmiany kąta obrotu wokół osi z są takie same, co spełnia oczekiwania.



Rysunek 5.12: Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia

Ostatnią próbą była weryfikacja działania opisanego algorytmu w pustym świecie z trzema obiektami. W przypadku tym tak jak w poprzednim jednostka wiodąca wykonuje omijanie, a śledzący ma ją śledzić. Jak można zobaczyć na poniższym wykresie trajektorii algorytm działa prawidłowo, tak jak w poprzednim teście. Jest na nim przedstawiony tor ruchu i zmiany kąta yaw.

Na wszystkich opisanych powyżej przypadkach algorytm działa prawidłowo, jednostka



Rysunek 5.13: Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia

śledząca leci za leaderem przyjmując jego położenie jako współrzędne środka okręgu, po którym ma się poruszać w przypadku zmiany orientacji. Gdy jednostka wiodąca leci przed siebie to follower leci za nią, co było założeniem tej metody.

5.4 Testy skuteczności algorytmu autolądowania

Milena Molska

Celem prób, była obserwacja zachowania jednostki UAV oraz weryfikacja wybranego rozwiązania pod względem skuteczności w określonych sytuacjach. Jako skuteczność rozumiałam tutaj zdolność wykrycia platformy, podążanie za nią i wylądowanie na niej. Wynikiem testów były nagrania symulacji oraz dane dotyczące trajektorii ruchu zapisane w plikach *.csv, na podstawie których wygenerowałam wykresy. Wybrane filmy umieściłam na płycie CD w folderze *Multimedia/Testy autolądowania*. Symulacje testowe, zostały przede mnie dobrane ze względu na prędkość poruszania się platformy.

Stąd sekcje 5.4.1 - 5.4.4, zawierają opisy prób dla:

- platformy statycznej,
- platformy poruszającej się z prędkością 0,07 m/s po linii prostej,
- platformy poruszającej się z prędkością 0,1 m/s po linii prostej,
- platformy poruszającej się z prędkością 0,2 m/s po linii prostej.

Warunkiem dla każdej symulacji było umieszczenie jednostki UAV na współrzędnych świata $XYZ=(0,0,1)$ z przednią kamerą opuszczoną w dół o 90 stopni. Położenie platformy i jej tor ruchu miały być tak zaplanowane, aby znaczniki ArUco nie znajdowały się w zasięgu widzenia bebopa. Zadaniem agenta, było znalezienie platformy, zbliżenie się do niej oraz wylądowanie. Przy czym warunkiem odcięcia napędów UAV, było to, aby jego wysokość nad platformą, nie przekraczała pewnej ustalonej wartości bezpiecznej tj. 0,7 m.

Dla każdego przypadku środowiska przeprowadziłam 20 testów. Ich wyniki zamieściłam w tabeli 5.1.

Tabela 5.1: Wyniki testów symulacyjnych

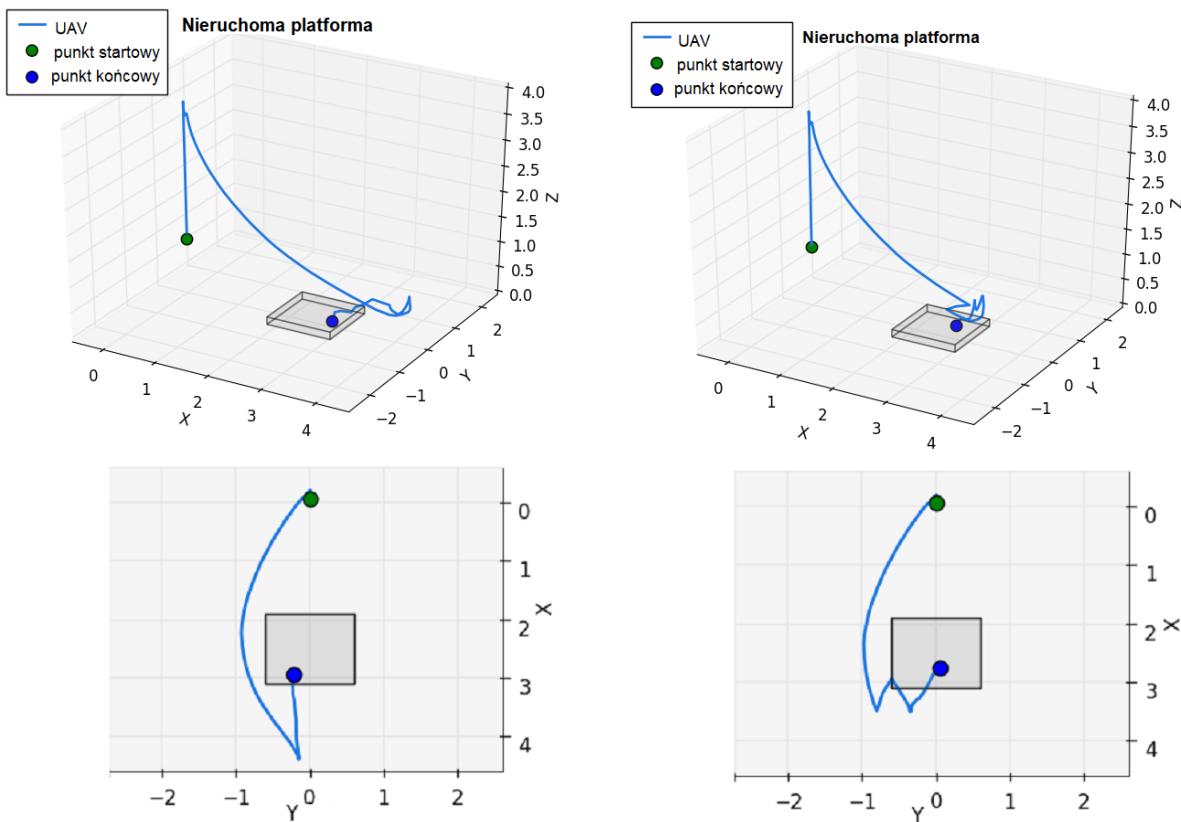
| Czasy przeprowadzenia manewru autolądowania dla platformy: | | | | | |
|--|-------------------------|-------------------------------|-----------------------------|----------------------------|----------------------------|
| Lp. | statycznej 2,5 m [s] | statycznej 1,5 x 1,5 m [s] | o prędkości 0,07 m/s [s] | o prędkości 0,1 m/s [s] | o prędkości 0,2 m/s [s] |
| 1. | 00:12.18 | 00:21.97 | 00:28.45 | 00:16.72 | 00:09.05 |
| 2. | 00:10.68 | 00:19.77 | 00:24.55 | - | - |
| 3. | 00:09.90 | 00:19.32 | 00:05.78 | 00:16.40 | - |
| 4. | 00:16.69 | 00:27.89 | 00:18.07 | 00:13.60 | 00:23.29 |
| 5. | 00:11.53 | 00:19.34 | 00:10.37 | - | - |
| 6. | 00:12.15 | 00:19.71 | 00:25.92 | - | 00:12.98 |
| 7. | 00:11.70 | 00:16.99 | 00:54:21 | 00:24.07 | - |
| 8. | 00:11.12 | 00:18.27 | 00:26.12 | 00:13.89 | 00:10.21 |
| 9. | 00:09.72 | 00:24.43 | 00:15.52 | - | - |
| 10. | 00:08.63 | 00:21.17 | 00:09.81 | 00:12.23 | 00:09.59 |
| 11. | 00:12.28 | 00:19.42 | 00:16.73 | 00:13.82 | - |
| 12. | 00:10.13 | 00:22.59 | 00:09.68 | 00:31.07 | - |
| 13. | 00:13.42 | 00:20.82 | 00:19.54 | 00:11.80 | 00:19.41 |
| 14. | 00:12.65 | 00:18.41 | 00:34.21 | 00:17.65 | - |
| 15. | 00:11.04 | 00:17.94 | 00:25.87 | - | - |
| 16. | 00:11.37 | 00:25.39 | 00:15.92 | 00:20.31 | 00:32.12 |
| 17. | 00:10.82 | 00:22.51 | 00:23.61 | 00:16.76 | - |
| 18. | 00:09.71 | 00:21.91 | 00:15.33 | 00:12.95 | - |
| 19. | 00:10.89 | 00:21.15 | 00:08.64 | 00:19.18 | 00:15.71 |
| 20. | 00:12.42 | 00:19.79 | 00:31.76 | 00:18.32 | 00:28.56 |
| średni czas: | 00:11.48 | 00:20.94 | 00:22.54 | 00:17.25 | 00:17.88 |
| skuteczność: | 100.00% | 100.00% | 100.00% | 75.00% | 45.00% |

Dla wszystkich wykonanych prób, zmierzyłam czas od momentu wykrycia platformy do wylądowania na niej. W przypadku testu zakończonego niepowodzeniem, nie wpisałam go do tabeli. Dla każdego rodzaju obliczyłam średni czas wykonania manewru oraz skuteczność zastosowanej metody. W dalszych podrozdziałach przybliżyłam każdy rodzaj przeprowadzonego testu.

Po analizie rezultatów, zdecydowałam się również na test autolądowania (na nieruchomojej platformie) w warunkach rzeczywistego lotu, który opisałam w rozdziale 5.4.5.

5.4.1 Lądownie na platformie statycznej

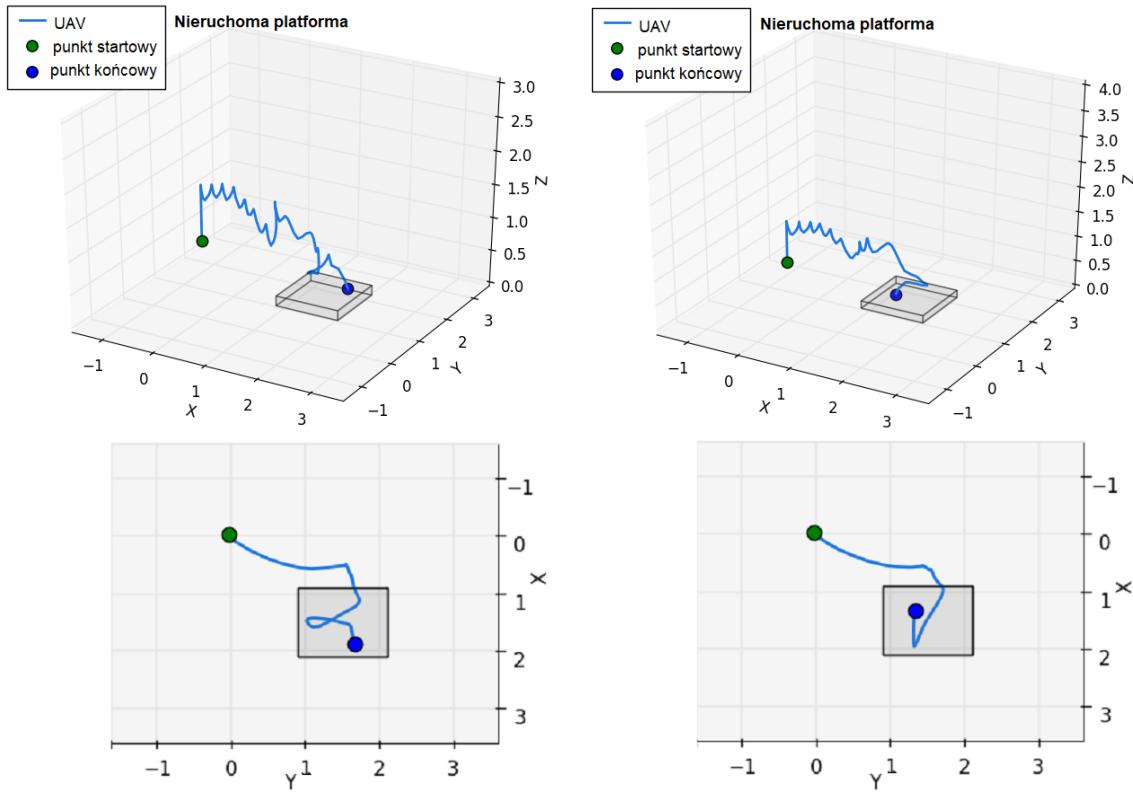
Pierwszym wariantem testu była platforma statyczna, której środek ciężkości znajdował się na wprost jednostki UAV w odległości 2,5 m. Dla tego układu, na początku symulacji podczas wznoszenia się robota latającego, na kamerze widoczne są dwa znaczniki, na podstawie których może oszacować względną pozycję. Przyjęłam to za najprostszą wersję próby i taką również się okazała. Na 20 wykonanych symulacji, wszystkie zakończyły się powodzeniem tj. UAV bez problemu znalazł platformę i na niej wylądował. Średni czas od wykrycia znaczników do zakończenia manewru wyniósł 11,48 s. Na rysunku 5.14 przedstawiłam wybrane trajektorie lotu jednostki UAV.



Rysunek 5.14: Wybrane trajektorie lotu jednostki UAV dla testów ze statyczną platformą oddaloną o 2,5 m w osi X od punktu startu

W celu sprawdzenia zachowania agenta dla innego położenie platformy, umieściłam ją na współrzędnych świata XY = (1,5, 1,5). W tym przypadku podczas początkowego wznoszenia się, widoczny jest tylko jeden marker, co jest utrudnieniem dla agenta. Jednak dla przeprowadzonych testów, zastosowana metoda osiągnęła 100% skuteczności. Wykonanie manewru lądownia zajęło mu średnio 20,94 s, czyli więcej czasu niż dla poprzedniego przypadku. Na podstawie przykładowych trajektorii z rysunku 5.15, można zauważyć charakterystyczne zachowanie podczas zgubienia z zasięgu widzenia platformy, tj. w momencie, gdy nie jest w stanie wykryć znacznika ArUco, unosi się w górę, aby zwiększyć

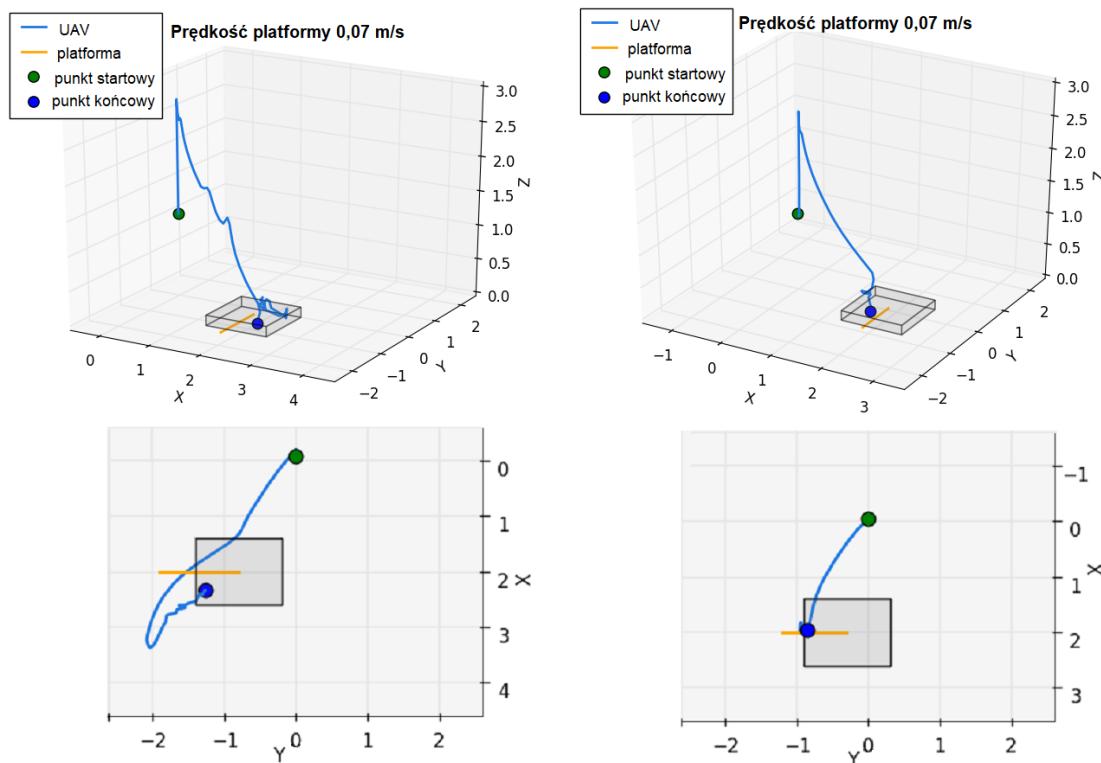
kąt widzenia. W tym przypadku UAV o wiele częściej gubił platformę, co tłumaczy wydłużenie czasu.



Rysunek 5.15: Wybrane trajektorie lotu jednostki UAV dla testów ze statyczną platformą oddaloną o 1,5 m w osi X i 1,5 m w osi Y od punktu startu

5.4.2 Ruch platformy z prędkością 0,07 m/s

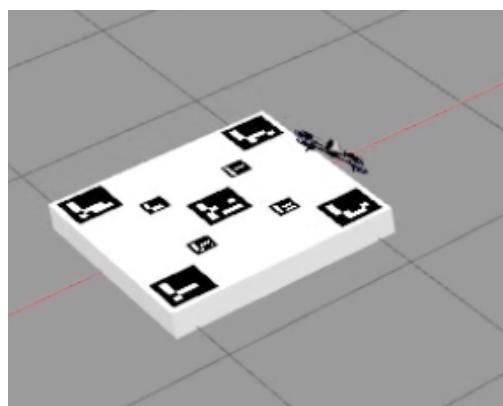
Kolejną próbą dla użytego algorytmu był ruch platformy w linii prostej z prędkością 0,07 m/s. Jej trajektoria oddalona była o 2 m w osi X od miejsca startu jednostki UAV. Na 20 przeprowadzonych symulacji każda zakończyła się sukcesem. Jednak wystąpiły tutaj duże rozbieżności czasu jeśli chodzi o wykonanie manewru. Najszybciej udało się agentowi wykonać zadanie w czasie 5,78 s. Natomiast najgorszy wynik wynosił aż 34,21 s. Rozbieżności te wynikają z różnych pozycji startowych platformy w osi Y oraz gubienia jej z zasięgu widzenia kamery. Jednak warto wspomnieć, że za każdym razem jednostka UAV była w stanie ponownie ją odnaleźć i przeprowadzić manewr w pełni prawidłowo. Średni czas z wszystkich prób wyniósł 22,54 s. Na rysunku 5.16 przedstawiłem wybrane trajektorie jednostki UAV oraz ruchomej platformy.



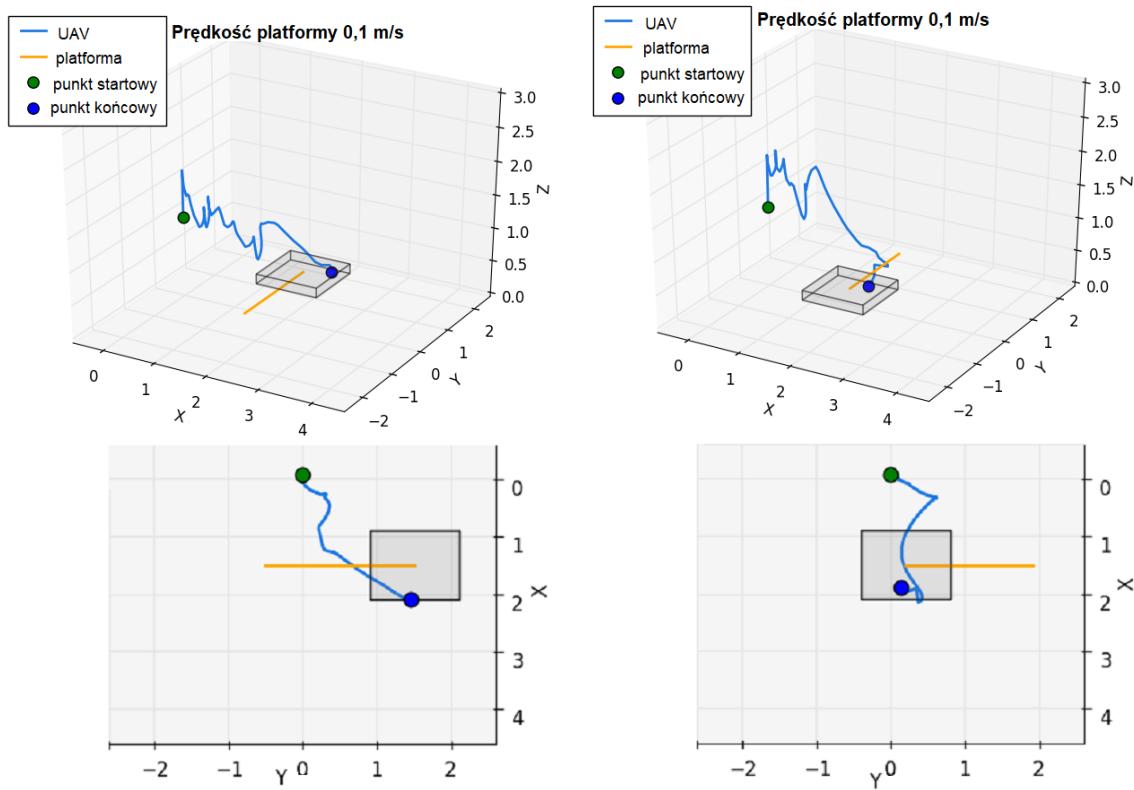
Rysunek 5.16: Wybrane trajektorie lotu jednostki UAV dla trzech testów z ruchomą platformą o prędkości 0,1 m/s, oddaloną o 2 m w osi X od punktu startu

5.4.3 Ruch platformy z prędkością 0,1 m/s

Chcąc utrudnić zadanie, postanowiłem zwiększyć prędkość platformy do 0,1 m/s, a jej pozycję w osi X zmieniłem na 1,5 m. Dla 20 wykonanych doświadczeń osiągnęłem skuteczność 75%. Jednak warto wspomnieć, dlaczego 5 z nich uznałem za nieudane. Patrząc czysto teoretycznie, jednostka UAV wylądowała na platformie, jednak ze skutkiem przedstawionym na rysunku 5.17.



Rysunek 5.17: Przykład nieprawidłowego wylądowania jednostki UAV podczas testów

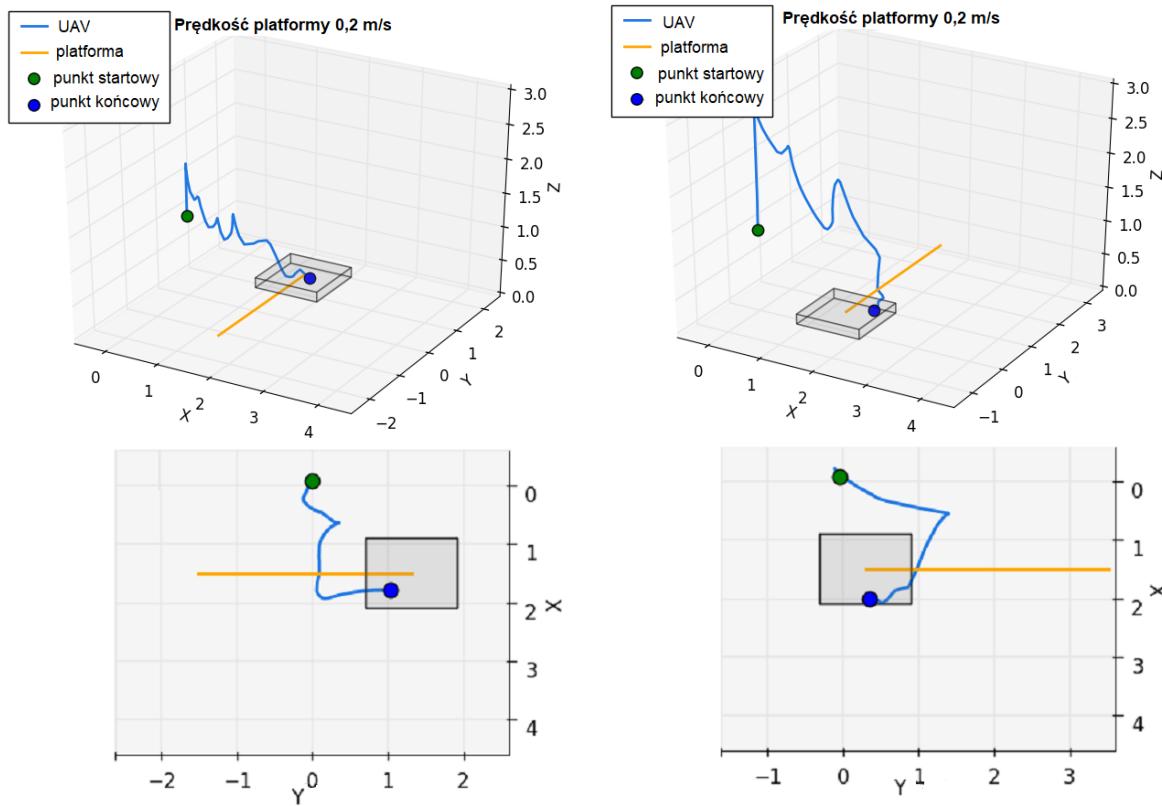


Rysunek 5.18: Wybrane trajektorie lotu jednostki UAV dla testów z ruchomą platformą o prędkości 0,1 m/s, oddaloną o 1,5 m w osi X od punktu startu

Przed przeprowadzeniem oficjalnych prób, zmienione zostały wartości skalujące podawane przez agenta wartości poruszania UAV w osi z, aby mógł on bez problemu nadążyć za platformą. Dla udanych prób najlepszy czas przeprowadzenia manewru wyniósł 11,8, najgorszy 31,07 s. Natomiast średnia z wszystkich symulacji - 17,25 s.

5.4.4 Ruch platformy z prędkością 0,2 m/s

Ostatnimi testami symulacyjnymi autolądowania było zwiększenie prędkości platformy do wartości 0,2 m/s. Dla tego wariantu skuteczność wyniosła tylko 45%. Jednak nieudane próby nie były takie same jak te opisane w sekcji 5.4.3. Problemem tutaj okazała się trudność zlokalizowania platformy, podczas wykonania zbyt gwałtownych ruchów, po których UAV nie był w stanie nadążyć za platformą. Jednak dla przypadków, w których ponownie udało się ją zlokalizować rezultaty były bardzo obiecujące. Najszybciej udało się przeprowadzić manewr w czasie 9,05 s, a najdłużej potrwał 32,12 s. Średnia z pozytywnych prób wyniosła 17,88 s, co jest bardzo zbliżonym wynikiem do prędkości dwa razy mniejszej.



Rysunek 5.19: Wybrane trajektorie lotu jednostki UAV dla testów z ruchomą platformą o prędkości 0,2 m/s, oddaloną o 1,5 m w osi X od punktu startu

5.4.5 Weryfikacja rozwiązań na rzeczywistym UAV

Mimo wstępnego założenia wykonania tylko i wyłącznie testów symulacyjnych, przeprowadziłem również próbę na rzeczywistym sprzęcie. Ze względu na ograniczoną ilość czasu, dotyczyła ona przypadku platformy statycznej.

W tym celu, wykonałem białą, papierową matę o wymiarach 1,2x1,2 m i za pomocą generatora [22], utworzyłem znaczniki ArUco o odpowiednich wymiarach:

- 0,235x0,235 m - markery o ID: 11, 12, 13, 14 i 33,
- 0,117x0,117 m - markery o ID: 15, 16, 17 i 18.

Następnie nakleiłem je według rozmieszczenia pokazanego na rysunku 4.2. Test przeprowadziłem w sali 109, budynku Wydziału Elektrycznego Politechniki Poznańskiej na jednostce Bebop 2 tj. tej samej, na której odbywały się symulacje. Mata została przytwierdzona do podłogi, a jej wysokość w programie zmieniona na 0,01 m, w celu poprawnego oszacowania względnej wysokości UAV. Wykonałem dwie próby i obie przebiegły pomyślnie. Zostały one zarejestrowane za pomocą kamery nagrywającej przestrzeń testową oraz kamery z Bebopa z nałożonym wykrywaniem znaczników. W katalogu *Multimedia/Testy rzeczywiste* znajdują się wykonane nagrania.



Rysunek 5.20: Pomieszczone testowe w sali 109 na Wydziale Elektrycznym Politechniki Poznańskiej - Aerolab

Dla rozmieszczenia obiektów przedstawionych na rysunku 5.20, przeprowadziłam również test symulacyjny w pomieszczeniu, którego nagranie znajduje się w katalogu *Multi-media/Testy autolądowania* pod nazwą *real_test_simulation.mp4*.

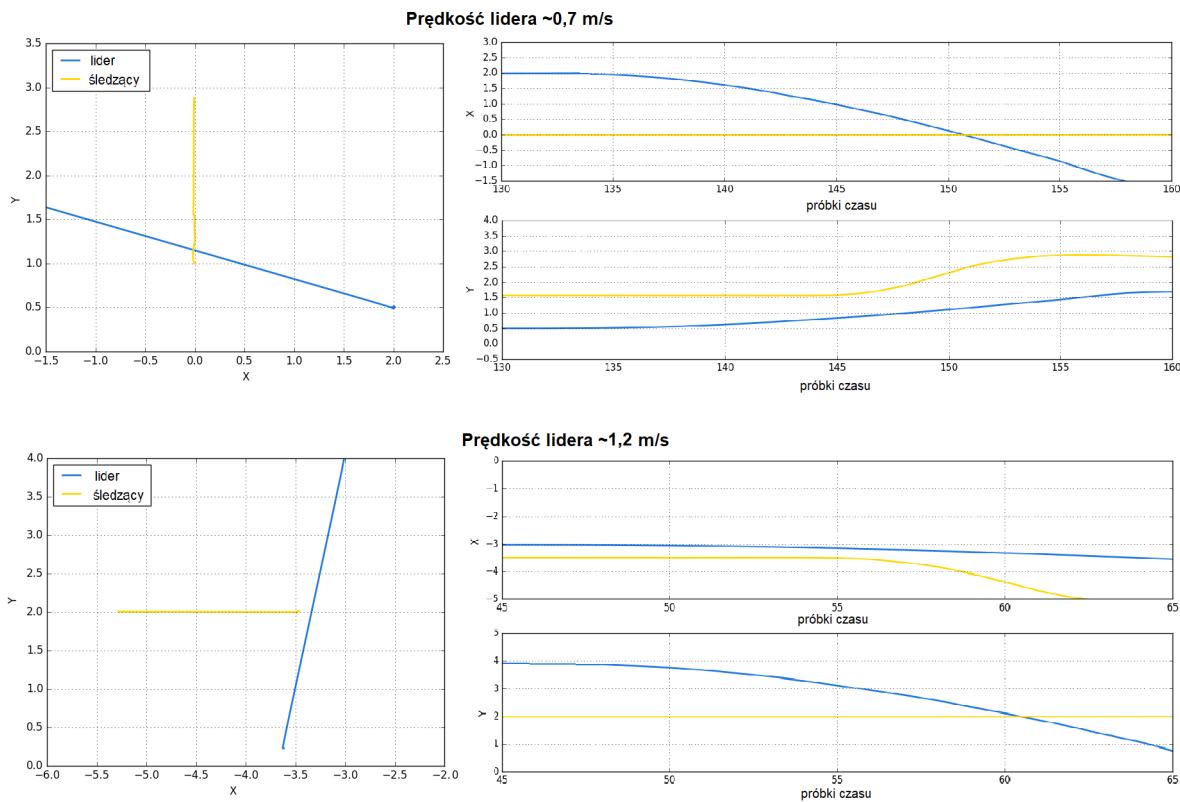
5.5 Unikanie kolizji jednostek UAV

Milena Molska

W celu sprawdzenia algorytmu unikania kolizji, który opisałam w rozdziale 4.4 posłużyłam się wariantami testowymi:

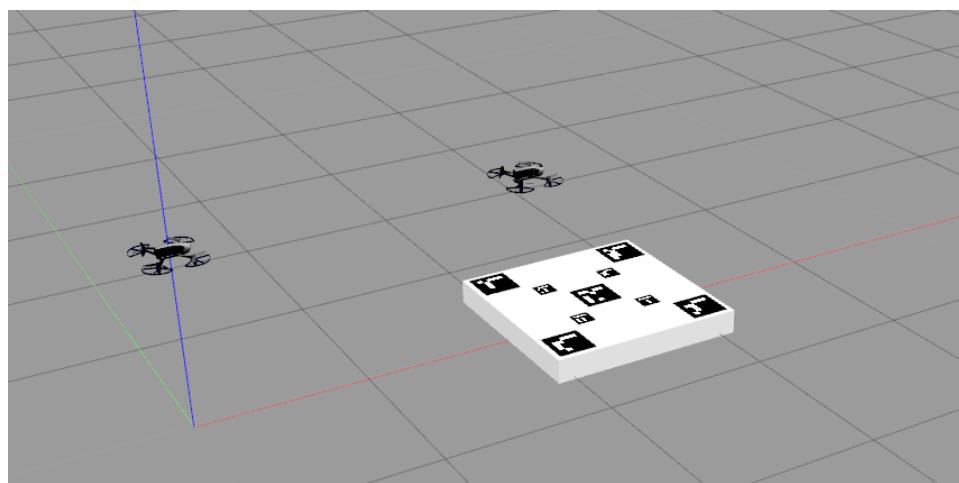
1. Wymuszanie kolizji za pomocą ręcznego sterowania liderem z poziomu klawiatury dla losowych prędkości i pozycji względem siebie.
2. Umieszczenie śledzącego UAV w kolizyjnym miejscu, podczas wykonywania przez lidera manewru autolądowania.

Pierwsza próba miała na celu sprawdzenie poprawności działania algorytmu dla różnych prędkości i pozycji lidera oraz wybranie optymalnych wymiarów strefy bezpieczeństwa. Warto również dodać, że dla tych prób lider poruszał się w linii prostej ruchem jednostajnie przyspieszonym, na wysokość tej samej co śledzący. Na rysunku 5.21 przedstawiłam otrzymane trajektorie lotu obu jednostek UAV. Bardzo dobrze jest na ich widoczne założenie algorytmu, tj. gdy prędkość lidera jest większa w osi X, to unik następuje w osi Y i odwrotnie. Dla prędkości do 1,5 m/s, bezpieczna strefa antykolizyjna ma promień 1 m, natomiast dla większych prędkości musiałam zmodyfikować ją do 1,5 m.



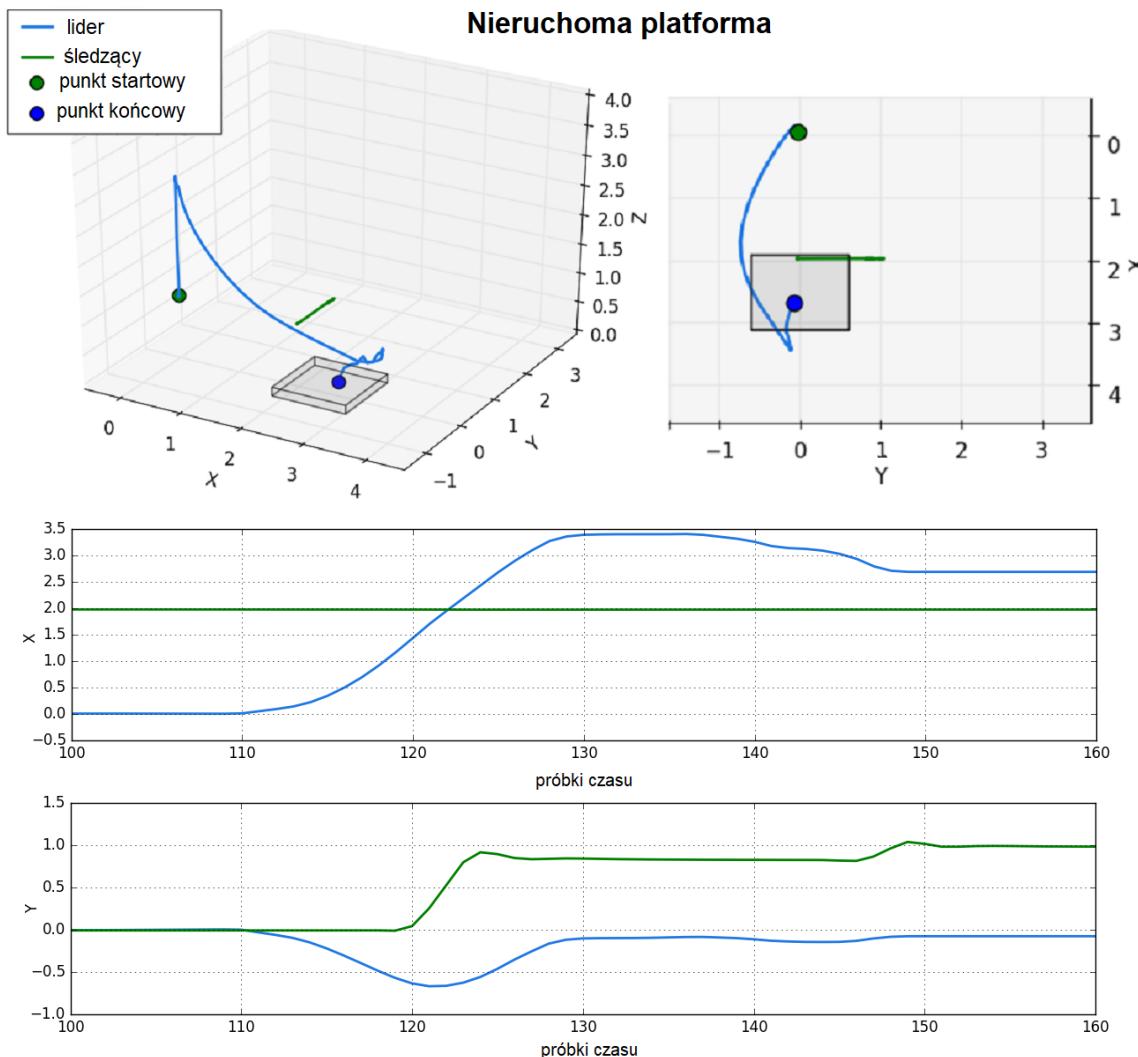
Rysunek 5.21: Wybrane trajektorie lotu jednostek UAV dla testu antykolizyjnego

Kolejną próbą jaką zrobiłem było sprawdzenie systemu antykolizyjnego dla dwóch jednostek UAV w momencie, gdy lider wykonuje manewr autolądowania, a śledzący znajduje się na torze jego ruchu. Pierwsze testy sporządziłem dla statycznej platformy z rozmieszczeniem jednostek UAV, jak przedstawiłem na rysunku 5.22.



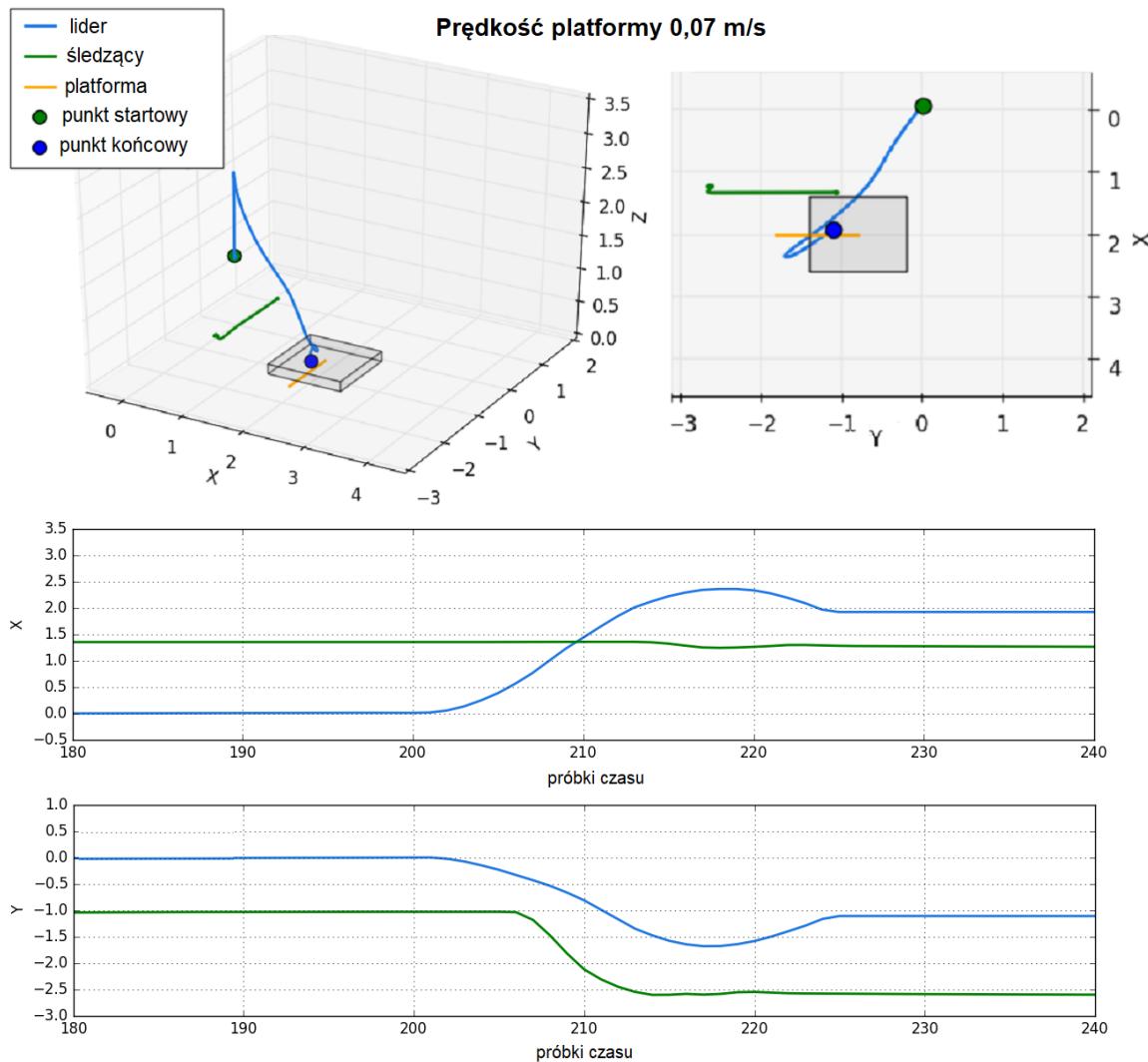
Rysunek 5.22: Ustawienie jednostek UAV podczas testu dla grupy UAV

Dla tak przeprowadzonego testu, mimo znacznej prędkości lidera, śledzący bez problemu uniknął kolizji i pozwolił na bezpieczne lądowanie. Na rysunku 5.23 przedstawiłem trajektorie lotu obu jednostek, a nagraną próbę umieściłem na płycie w katalogu *Multimedia/Testy autolądowania* pod nazwą *collision_avoid_autolanding_test1.mp4*.



Rysunek 5.23: Trajektorie lotu jednostek UAV dla testu ze statyczną platformą i algorytmem antykolizyjnym

Ostatnim testem było sprawdzenie autolądowania i unikania kolizji dla ruchomej platformy. Do tej próby wybrałem prędkość, dla której jednostka UAV we wcześniejszych testach osiągnęła 100% skuteczność tj. 0,07 m/s. Na rysunku 5.24 przedstawiłem trajektorie dla obu jednostek oraz tor ruchu platformy. Test zakończył się sukcesem zarówno pod względem uniknięcia zderzenia, jak i pod względem przeprowadzenia manewru lądowania na platformie. Udokumentowałem próbę nagraniem zamieszczonym na płycie CD o ścieżce *Multimedia/Testy autolądowania/collision_avoid_autolanding_test2.mp4*.



Rysunek 5.24: Trajektorie lotu jednostek UAV dla testu z ruchomą platformą i algorytmem antykolizyjnym

Zakończenie

6.1 Podsumowanie

Cele postawione dla pracy zostały zrealizowane. Zaimplementowano oraz udoskonalono symulator rozszerzonej rzeczywistości dla jednostek UAV oparty na systemie OptiTrack. Dodano wyuczone sieci neuronowe w celu detekcji przedmiotów. Zastosowano przepływ optyczny do omijania przeszkód dla grupy UAV. Wykorzystano wyuczonego agenta do przeprowadzenia manewru autolądowania dla dwóch jednostek. Dla wymienionych składowych przeprowadzono testy symulacyjne i rzeczywiste.

Oprogramowanie dla jednostek UAV zostało napisane głównie w języku C++ oraz Python. W pracy zdecydowano się na podzielenie projektu na pakiety, które można zastępować w zależności od wymagań użytkownika.

Zastosowane rozwiązania sprawdzono symulacyjnie w programie Sphinx działającym pod ROS/Gazebo oraz podczas testów rzeczywistych. Symulator AR prawidłowo rozmieszcza modele obiektów w symulowanym świecie i wykonuje przemieszczenia zgodne z rzeczywistymi. Jednostka dla większości przypadków omija przeszkody, a dla dwóch jednostek poprawnie działa algorytm śledzenia. Metoda jest efektywniejsza, gdy zastosowano światy symulacyjne bez zakłóceń. Manewr autolądowania działa poprawnie dla większości przetestowanych możliwości, a dla dwóch jednostek UAV nie następuje kolizja.

6.2 Potencjalne kierunki rozwoju pracy

Niniejsza praca ma potencjał na udoskonalenie i rozwój. Dla metody omijania przeszkód w przyszłości warto byłoby zastosować inne rozwiązania wykrywania obiektów. Podczas testów zauważono, że aktualna metoda jest lepsza do zastosowania w większych przestrzeniach, ze względu na efektywniejsze wykrywanie przedmiotów w większych odległościach. Można byłoby się zdecydować na zastosowanie detekcji za pomocą markera szachownicy, dzięki czemu zyskałoby się możliwość obliczenia odległości od przedmiotu. Kolejnym sposobem modyfikacji byłoby zastosowanie reakcyjnego omijania. W takim wy-

padku należałyby zmieniać orientację robota w taki sposób, aby wykryty obiekt nie znajdował się w centrum obrazu. Następnym byłoby zamontowanie na jednostce kamery z obrazem głębi, dzięki czemu możliwe byłoby dostarczenie tego widoku jako dane do wyuczenia sieci neuronowej. Dobrym sposobem modyfikacji byłoby również zamontowanie czujników odległości. Dla manewru autolądowania warto byłoby opracować system dla grupy jednostek UAV, pozwalający na mapowanie terenu w celu odnalezienia platformy. Aby uniknąć zderzenia z innymi robotami latającymi na otwartej przestrzeni, można by było wykorzystać dodatkowo system GPS i czujniki odległości.

Bibliografia

- [1] OFICJALNA STRONA GAZEBO <http://gazebosim.org> [Dostęp: 30.01.2019].
- [2] <https://www.leroymerlin.pl/planer-3d/aplikacja-planer-3d.html>
[Dostęp: 30.01.2019].
- [3] DOKUMENTACJA PARROT-SPHINX
<https://developer.parrot.com/docs/sphinx/index.html> [Dostęp: 30.01.2019].
- [4] <https://forum.developer.parrot.com/t/change-mass-and-inertia-of-bebop-2/8463>
[Dostęp: 30.01.2019].
- [5] DOKUMENTACJA BEBOP AUTONOMY
Dostępna w internecie: <https://bebop-autonomy.readthedocs.io/>
[Dostęp: 30.01.2019].
- [6] ZADAROWSKA K., *Wprowadzenie do systemu operacyjnego ROS: symulator turtlesim*. Wydział Elektroniki, Politechnika Wrocławskiego.
Dostępny w internecie: <https://bit.ly/2FPm8tn> [Dostęp: 30.01.2019]
- [7] <https://forum.developer.parrot.com/t/running-multiples-drones-in-a-simulation/7646/7> [Dostęp: 30.01.2019].
- [8] FIRMWARE IMAGES FOR SPHINX:
<http://plf.parrot.com/sphinx/firmwares/index.html> [Dostęp: 30.01.2019].
- [9] FRANÇOIS CHOLLET : *Deep Learning with Python*.
Dostępny w internecie: <https://bit.ly/2T5UNXY> [Dostęp: 30.01.2019]
- [10] PRABHU: *Understanding of Convolutional Neural Network (CNN) - Deep Learning*
Dostępny w internecie: <https://bit.ly/2L7mBKK> [Dostęp: 30.01.2019]
- [11] MUNEEB UL HASSAN: *VGG16 – Convolutional Network for Classification and Detection* Dostępny w internecie: <https://neurohive.io/en/popular-networks/vgg16/> [Dostęp: 30.01.2019]
- [12] EDDIE FORSON: *Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning*. Dostępny w internecie: <https://bit.ly/2wqox7L>
[Dostęp: 30.01.2019]

- [13] KURNYTA Sz.
Optymalizacja algorytmu wyznaczającego przepływ optyczny dla obrazów kolorowych. Akademia Górnictwo-Hutnicza w Krakowie, 2008
[Dostęp: 30.01.2019]
- [14] DOKUMENTACJA OPENCV – PRZEPŁYW OPTYCZNY:
https://docs.opencv.org/3.4/d7/d8b/tutorial_py_lucas_kanade.html
[Dostęp: 30.01.2019]
- [15] AMBROZIAK L.
Metoda rozpoznawania przeszkodek przez bezzałogowy statek powietrzny z wykorzystaniem jednej kamery. Politechnika Białostocka, 2011
[Dostęp: 30.01.2019]
- [16] OBJECT_DETECTION_TENSORFLOW:
https://github.com/karolmajek/object_detection_tensorflow
[Dostęp: 30.01.2019]
- [17] OPTICAL-FLOW-BASED-OBSTACLE-AVOIDANCE:
<https://github.com/zainmehdi/Optical-Flow-based-Obstacle-Avoidance>
[Dostęp: 30.01.2019]
- [18] ALEJANDRO RODRIGUEZ-RAMOS: *A Deep Reinforcement Learning Technique for Vision-Based Autonomous Multirotor Landing on a Moving Platform*.
Politechnika w Madrycie. Dostępny w internecie: <https://link.do/ej2t0>
[Dostęp: 30.01.2019]
- [19] DRL_LANDING: <https://github.com/alejodosr/drl-landing> [Dostęp: 30.01.2019]
- [20] GAZEBO API REFERENCE:
<http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/index.html>
[Dostęp: 30.01.2019]
- [21] DRONEMSGSROS:
<https://github.com/Ahrovan/dronemsgsros> [Dostęp: 30.01.2019]
- [22] WOJCIECH GIERNACKI: *Drony i bezzałogowe statki powietrzne*.
Wydawnictwo Politechniki Poznańskiej, 2018.
- [23] BARTŁOMIEJ KULECKI: *Autonomiczna rejestracja sekwencji obrazów przy wykorzystaniu grupy bezzałogowych statków powietrznych*. Manuskrypt, w opracowaniu, Politechnika Poznańska, 2019 r.
- [24] ONLINE ARUCO MARKERS GENERATOR <http://chev.me/arucogen/>
[Dostęp: 30.01.2019]
- [25] PYTHON 2.7.15 DOCUMENTATION <https://docs.python.org/2.7/>
[Dostęp: 30.01.2019]

Spis rysunków

| | | |
|------|---|----|
| 1.1 | Struktura katalogów modeli w Gazebo | 6 |
| 1.2 | Świat symulacyjny w Gazebo | 8 |
| 1.3 | Symulator z uruchomionym modelem jednostki Bebop 2 | 8 |
| 1.4 | Model 3D Bebopa 2 przed i po modyfikacji | 9 |
| 1.5 | Działanie projektu na platformie ROS [6] | 10 |
| 1.6 | Dwie jednostki UAV uruchomione w symulacji | 11 |
| 1.7 | Graf przedstawiający strukturę węzłów i tematów w systemie ROS | 12 |
| 2.1 | Przykład różnicy pomiędzy widokiem w rzeczywistości oraz w symulatorze | 13 |
| 2.2 | Wykorzystane interfejsy | 14 |
| 2.3 | Bateria UAV z naniesionymi znacznikami wykorzystywanyimi w OptiTracku | 14 |
| 2.4 | Zależności pomiędzy układem współrzędnych w symulatorze Gazebo i w systemie OptiTrack | 15 |
| 2.5 | Graf węzłów i tematów pakietu <i>Optitrack</i> | 16 |
| 2.6 | Działanie zmodyfikowanego przez nas pakietu <i>Optitrack</i> | 17 |
| 3.1 | Schemat sieci neuronowej kilkuwarstwowej [9] | 19 |
| 3.2 | Warstwa w pełni połączona [10] | 19 |
| 3.3 | Sposób działania CNN [10] | 20 |
| 3.4 | Architektura sieci VGG-16 [11] | 21 |
| 3.5 | Architektura SSD [12] | 21 |
| 3.6 | Przedstawienie położenia czerwonego punktu w 5 następujących po sobie klatkach, strzałka obrazuje wektor ruchu [14] | 23 |
| 3.7 | Pomieszczenie z przeszkodą | 25 |
| 3.8 | Fragment kodu gotowego skryptu [14] | 26 |
| 3.9 | Działanie skryptu w pomieszczeniu z przeszkodą | 27 |
| 3.10 | Graf węzłów i tematów | 27 |
| 3.11 | Działanie pakietu <i>obstacle_avoidance</i> w pomieszczeniu z przeszkodą | 29 |
| 3.12 | Graf węzłów i tematów | 30 |
| 3.13 | Metoda śledzenia | 31 |
| 3.14 | Graf węzłów i tematów w zastosowaniu algorytmu dla grupy UAV | 32 |

| | | |
|------|---|----|
| 4.1 | Architektura wybranego systemu autoladowania opartego na głebokim uczeniu ze wzmacnieniem [18]. SM to układ lokalny jednostki UAV, którego początek to jej środek ciężkości, C - układ odniesienia przedniej kamery Bebopa, natomiast O to układ lokalny platformy, znajdujący się na środku jej powierzchni. | 34 |
| 4.2 | Rozmieszczenie znaczników ArUco na platformie | 36 |
| 4.3 | Wykrycie platformy przez jednostkę UAV | 37 |
| 4.4 | Graf tematów użyty przez środowisko RlEnvironmentGazeboROS | 38 |
| 4.5 | Graf tematów użytych przez autorski skrypt bebop_dronemsgsros | 38 |
| 4.6 | Graf węzłów i tematów po uruchomieniu całego oprogramowania | 39 |
| 4.7 | Metoda unikania kolizji dla dwóch jednostek UAV | 40 |
| 4.8 | Graf tematów autorskiego pakietu collision_avoid | 40 |
| 4.9 | Sterowanie UAV za pomocą klawiatury - interfejs. | 41 |
| 4.10 | Graf tematów, z których korzysta pakiet bebop_keyboard | 41 |
| 5.1 | Wynik testu w pomieszczeniu o 2 źródłach światła | 43 |
| 5.2 | Wynik testu w pomieszczeniu o 2 źródłach światła | 43 |
| 5.3 | Wynik testu w pomieszczeniu z jednym źródłem oświetlenia | 44 |
| 5.4 | Pusty świat z jednym przedmiotem | 44 |
| 5.5 | Wybrane trajektorie ruchu oraz zmiany kąta dla próby z oddaloną przeszkodą o 1 m | 46 |
| 5.6 | Wybrane trajektorie ruchu oraz zmiany kąta dla próby z oddaloną przeszkodą o 1,5 m | 46 |
| 5.7 | Wykorzystany świat dla dwóch obiektów | 47 |
| 5.8 | Wybrane trajektorie ruchu oraz zmiany kąta dla omijania dwóch przeszkód | 48 |
| 5.9 | Wykorzystany świat dla trzech obiektów | 49 |
| 5.10 | Wybrane trajektorie ruchu oraz zmiany kąta dla omijania trzech przeszkód | 49 |
| 5.11 | Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia | 51 |
| 5.12 | Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia | 51 |
| 5.13 | Trajektoria ruchu oraz zmiany kąta lidera oraz śledzącego podczas śledzenia | 52 |
| 5.14 | Wybrane trajektorie lotu jednostki UAV dla testów ze statyczną platformą oddaloną o 2,5 m w osi X od punktu startu | 54 |
| 5.15 | Wybrane trajektorie lotu jednostki UAV dla testów ze statyczną platformą oddaloną o 1,5 m w osi X i 1,5 m w osi Y od punktu startu | 55 |
| 5.16 | Wybrane trajektorie lotu jednostki UAV dla trzech testów z ruchomą platformą o prędkości 0,1 m/s, oddaloną o 2 m w osi X od punktu startu | 56 |
| 5.17 | Przykład nieprawidłowego wylądowania jednostki UAV podczas testów | 56 |
| 5.18 | Wybrane trajektorie lotu jednostki UAV dla testów z ruchomą platformą o prędkości 0,1 m/s, oddaloną o 1,5 m w osi X od punktu startu | 57 |

| | |
|---|----|
| 5.19 Wybrane trajektorie lotu jednostki UAV dla testów z ruchomą platformą o prędkości 0,2 m/s, oddaloną o 1,5 m w osi X od punktu startu | 58 |
| 5.20 Pomieszczone testowe w sali 109 na Wydziale Elektrycznym Politechniki Poznańskiej - Aerolab | 59 |
| 5.21 Wybrane trajektorie lotu jednostek UAV dla testu antykolizyjnego | 60 |
| 5.22 Ustawienie jednostek UAV podczas testu dla grupy UAV | 60 |
| 5.23 Trajektorie lotu jednostek UAV dla testu ze statyczną platformą i algorytmem antykolizyjnym | 61 |
| 5.24 Trajektorie lotu jednostek UAV dla testu z ruchomą platformą i algorytmem antykolizyjnym | 62 |

Spis zawartości załączonej płyty CD

Płyta CD

| |
|---------------------------|
| spis.txt |
| └── Oprogramowanie |
| └── Multimedia |
| └── Grafiki |
| └── Modele 3D |
| └── praca inżynierska.pdf |
| └── praca inżynierska.tex |