

Politechnika  
Śląska

## PROJEKT INŻYNIERSKI

Budowa mapy otoczenia z wykorzystaniem robota mobilnego

Krzysztof GRĄDEK

Nr albumu: 300362

**Kierunek:** Automatyka i Robotyka

**Specjalność:** Technologie Informacyjne

**PROWADZĄCY PRACĘ**

dr inż. Krzysztof Jaskot

Katedra Automatyki i Robotyki

Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2025



## Tytuł pracy

Budowa mapy otoczenia z wykorzystaniem robota mobilnego

## Streszczenie

Projekt koncentruje się na implementacji systemu autonomicznej nawigacji robota mobilnego, z naciskiem na dwa kluczowe aspekty: tworzenie mapy otoczenia oraz realizację precyzyjnej nawigacji z punktu do punktu w zmapowanej przestrzeni wykorzystując rozwiązanie typu SLAM. Rozwiązanie opiera się na dwóch współpracujących ze sobą mikrokontrolerach - Raspberry Pi 4, który odpowiada za obsługę czujnika RPLidar A1 oraz wykonywanie algorytmów mapowania i nawigacji, oraz Arduino Nano zarządzającym silnikami z enkoderami, zapewniającymi precyzyjne sterowanie ruchem robota. Wykonana konfiguracja zrealizowana została w językach C++ oraz Python, z wykorzystaniem narzędzi z ekosystemu ROS 2 (ang. "Robot Operating System 2"), takich jak Nav2 (ang. "Navigation 2") i SLAM Toolbox (ang. "Simultaneous Localization and Mapping Toolbox").

## Słowa kluczowe

mapowanie, robot mobilny, lokalizacja, SLAM, **ROS 2**

## Thesis title

Construction of an Environment Map Using a Mobile Robot

## Abstract

The project focuses on implementing an autonomous mobile robot navigation system, emphasizing two key aspects: environment mapping and precise point-to-point navigation in the mapped space using SLAM type solution. The solution is based on two cooperating microcontrollers - Raspberry Pi 4, which handles the RPLidar A1 sensor and executes mapping and navigation algorithms, and Arduino Nano managing motors with encoders, providing precise robot motion control. The implemented configuration was realized using C++ and Python languages, utilizing tools from the ROS 2 (Robot Operating System 2) ecosystem, such as Nav2 (Navigation 2) and SLAM Toolbox (Simultaneous Localization and Mapping Toolbox).

## Key words

mapping, mobile robot, localization, SLAM, **ROS 2**



# Spis treści

<b>1 Wstęp</b>	<b>1</b>
1.1 Wprowadzenie w problem . . . . .	1
1.2 Osadzenie problemu w dziedzinie . . . . .	2
1.3 Cel pracy . . . . .	2
1.4 Zakres pracy . . . . .	2
1.5 Struktura pracy . . . . .	3
1.6 Wkład własny autora . . . . .	3
<b>2 Analiza tematu</b>	<b>5</b>
2.1 Osadzenie tematu w kontekście aktualnego stanu wiedzy i badań . . . . .	5
2.2 Szczegółowe sformułowanie problemu . . . . .	6
2.3 Przegląd narzędzi i literatury . . . . .	7
2.3.1 ROS 2 . . . . .	7
2.3.2 SLAM Toolbox . . . . .	8
2.3.3 Navigation2 (Nav2) i lokalizacja . . . . .	9
2.3.4 ROS 2 Control i sterowanie napędami . . . . .	10
2.4 Wybór rozwiązań . . . . .	10
<b>3 Wymagania i narzędzia</b>	<b>11</b>
3.1 Wymagania funkcjonalne . . . . .	11
3.2 Przypadki użycia . . . . .	12
3.3 Specyfikacja komponentów . . . . .	13
3.3.1 Jednostki sterujące . . . . .	13
3.3.2 Napęd . . . . .	13
3.3.3 Zasilanie . . . . .	13
3.4 Budowa robota i sposób połączenia silników z kontrolerem . . . . .	14
3.5 Metodyka i etapy realizacji . . . . .	19
3.5.1 Etap 1: Przygotowanie platformy sprzętowej . . . . .	19
3.5.2 Etap 2: Implementacja sterowania napędem . . . . .	19
3.5.3 Etap 3: Integracja sensorów . . . . .	19
3.5.4 Etap 4: Implementacja oprogramowania . . . . .	19

<b>4 Specyfikacja użytkowa</b>	<b>21</b>
4.0.1 Wymagania sprzętowe i programowe . . . . .	21
4.0.2 Sposób aktywacji i korzystania z robota z przykładem działania . . . . .	22
4.1 Administracja systemem . . . . .	32
4.2 Kwestie bezpieczeństwa . . . . .	34
4.3 Scenariusze korzystania z systemu . . . . .	34
<b>5 Specyfikacja techniczna</b>	<b>35</b>
5.1 Wykorzystane technologie . . . . .	35
5.2 Architektura systemu . . . . .	36
5.3 Struktura systemu i objaśnienie działania algorytmów . . . . .	37
5.3.1 Arduino . . . . .	37
5.3.2 LiDAR . . . . .	41
5.3.3 Sterowanie silnikami i model robota . . . . .	41
5.3.4 Mapowanie . . . . .	45
5.3.5 Nawigacja i lokalizacja . . . . .	47
5.4 Diagramy UML prezentujące działanie konkretnych programów . . . . .	49
<b>6 Weryfikacja i walidacja</b>	<b>53</b>
6.1 Model V . . . . .	53
6.2 Organizacja eksperymentów . . . . .	55
6.3 Przypadki testowe . . . . .	55
6.4 Wykryte i usunięte błędy . . . . .	55
<b>7 Podsumowanie i wnioski</b>	<b>57</b>
7.1 Uzyskane wyniki i wnioski . . . . .	57
7.2 Kierunki dalszych prac . . . . .	59
<b>Bibliografia</b>	<b>61</b>
<b>Spis skrótów i symboli</b>	<b>65</b>
<b>Lista dodatkowych plików, uzupełniających tekst pracy</b>	<b>67</b>
<b>Spis rysункów</b>	<b>70</b>

# Rozdział 1

## Wstęp

Poniższy projekt obejmuje implementację systemu autonomicznej nawigacji robota mobilnego, z naciskiem na dwa kluczowe aspekty: tworzenie mapy otoczenia oraz realizację precyzyjnej nawigacji z punktu do punktu w zmapowanej przestrzeni. Tego typu zadania są kluczowe w dziedzinie robotyki mobilnej, umożliwiając robotom samodzielne poruszanie się w nowych nieznanych przestrzeniach. W tym rozdziale przedstawiono cel pracy, jej zakres oraz strukturę.

### 1.1 Wprowadzenie w problem

Jednoczesna lokalizacja i mapowanie - SLAM (ang. "Simultaneous localization and mapping") to proces, w którym robot konstruuje mapę nieznanego środowiska podczas jednoczesnej lokalizacji w tym środowisku i śledzenia swojej trajektorii poruszania się [2]. Jest to jedno z kluczowych zagadnień w robotyce mobilnej, umożliwiając robotom samodzielne poruszanie się w przestrzeni. W praktyce SLAM jest realizowany za pomocą zestawu sensorów, takich jak skanery laserowe, kamery RGB-D, czy IMU (ang. "Inertial Measurement Unit"), oraz algorytmów, które przetwarzają dane z tych sensorów w celu budowy mapy i lokalizacji robota.

## 1.2 Osadzenie problemu w dziedzinie

Ten projekt zalicza się do dziedziny robotyki mobilnej i systemów autonomicznych. Roboty mobilne są szeroko wykorzystywane w przemyśle, logistyce, czy badaniach naukowych. Realizacja systemu SLAM i autonomicznej nawigacji wymaga integracji wielu zaawansowanych technologii. Główne wyzwania techniczne obejmują wykorzystanie i synchronizację:

- Sensorów w tym LiDAR (ang. "Light Detection and Ranging")
- Algorytmów SLAM
- Platform programistycznych dla robotów jak **ROS 2** (ang. "Robot Operating System 2")
- Systemów nawigacji jak **Nav2**
- Systemów sterowania jak **ROS 2 Control**
- Systemów wizualizacji i analizy danych
- Systemów komunikacji i zarządzania danymi

## 1.3 Cel pracy

Głavnym celem pracy jest zaprojektowanie i implementacja systemu mapowania otoczenia z wykorzystaniem robota mobilnego. W ramach realizacji tego zadania przewidziano budowę platformy mobilnej, implementację systemu sterowania robotem oraz integrację niezbędnych czujników i urządzeń. Kluczowym elementem jest realizacja algorytmów SLAM, które umożliwiają jednoczesną lokalizację robota i tworzenie mapy otoczenia. Dodatkowo, system ma zapewniać możliwość nawigacji z punktu do punktu oraz sterowania manualnego.

## 1.4 Zakres pracy

Realizacja projektu obejmuje szereg wzajemnie powiązanych zadań. Na pierwszy etap składa się dogłębna analiza istniejących rozwiązań w dziedzinie mapowania i nawigacji robotów mobilnych, która stanowi podstawę do dalszych prac. Na tej bazie opracowany jest projekt systemu sterowania robotem, a następnie przeprowadzona jego implementacja. Kolejnym krokiem jest integracja komponentów sprzętowych i programowych w spójny system. Szczególną uwagę poświęcono implementacji algorytmów SLAM i nawigacji, które stanowią rdzeń funkcjonalności robota. Całość prac kończy seria testów i walidacja stworzonego rozwiązania.

## 1.5 Struktura pracy

Praca składa się z siedmiu następujących rozdziałów:

- Rozdział pierwszy zawiera wstęp, w którym przedstawiono cel pracy, jej zakres oraz strukturę.
- Rozdział drugi zawiera analizę tematu, szczegółowe sformułowanie problemu, osadzenie go w kontekście aktualnego stanu wiedzy i badań, przegląd narzędzi i literatury oraz uzasadnienie wyboru rozwiązania.
- Rozdział trzeci zawiera wymagania i narzędzia, w którym opisano wymagania funkcjonalne, przypadki użycia, specyfikację komponentów, budowę robota, metodykę i etapy realizacji projektu.
- Rozdział czwarty zawiera specyfikację użytkową, w którym przedstawiono wymagania sprzętowe i programowe, sposób aktywacji i korzystania z robota z przykładem działania, administrację systemem, kwestie bezpieczeństwa, oraz scenariusze korzystania z systemu.
- Rozdział piąty zawiera specyfikację techniczną, w którym przedstawiono wykorzystane technologie, architekturę systemu, strukturę systemu z objaśnieniem działania algorytmów, oraz diagramy UML prezentujące działanie konkretnych programów.
- Rozdział szósty zawiera weryfikację i validację, w którym opisano model v jaki został zastosowany do testowania systemu. Opisano tu również organizację eksperymentów, przypadki testowe oraz wykryte i usunięte błędy.
- Rozdział siódmy zawiera podsumowanie, w którym przedstawiono uzyskane wyniki, oraz kierunki dalszych prac.

## 1.6 Wkład własny autora

W ramach pracy autor samodzielnie:

- Zaprojektował i zbudował platformę mobilną
- Zaimplementował sterowniki urządzeń
- Zintegrował komponenty sprzętowe i programowe
- Zaimplementował i dostosował algorytmy SLAM
- Przeprowadził testy i optymalizację systemu



# Rozdział 2

## Analiza tematu

W niniejszym rozdziale przedstawiono analizę problemu jednoczesnej lokalizacji i mapowania (SLAM) oraz autonomicznej nawigacji robotów mobilnych. Omówiono aktualny stan wiedzy w tej dziedzinie, sformułowano szczegółowo problem badawczy oraz dokonano przeglądu dostępnych rozwiązań i algorytmów. Na podstawie tej analizy wybrano optymalne narzędzia i metody do realizacji projektu.

### 2.1 Osadzenie tematu w kontekście aktualnego stanu wiedzy i badań

Historię rozwoju algorytmów SLAM, można podzielić na trzy epoki, jak zrobiono to w artykule "Przegląd SLAM: od pojedynczego czujnika do heterogenicznej fuzji" (ang. "SLAM Overview: From Single Sensor to Heterogeneous Fusion") [3]: 1986-2005 jako epokę klasyczną, podczas której powstały pierwsze algorytmy SLAM, 2006-2015 jako epokę uczenia maszynowego i optymalizacji, która poświęcona była rozszerzaniu i ulepszaniu algorytmów SLAM, oraz 2016-obecnie jako epokę łączenia algorytmów SLAM z głębokim uczeniem. Pierwsze wizje probabilistycznego systemu SLAM zostały przedstawione na konferencji ICRA w 1986. Od tego momentu zaczęto uznawać problem mapowania jako jeden z głównych problemów w robotyce. Początkowymi algorytmami były EKF (ang. "Extended Kalman Filter") oraz FastSLAM, które były stosunkowo proste i miały wiele ograniczeń. Z rozpoczęciem drugiej epoki czujniki LiDAR stały się coraz bardziej popularne, co pozwoliło na uzyskanie dokładniejszych map. W 2016 roku nastąpił przełom w dziedzinie SLAM, kiedy to pojawiły się pierwsze algorytmy łączące SLAM z głębokim uczeniem. Zastosowanie takich algorytmów w dużym stopniu poprawiło jakość map w wysoce dynamicznych środowiskach.

Stosowanie algorytmów SLAM opisano w pracy "Analiza porównawcza nawigacji wewnętrznej opartej na technologii LiDAR SLAM dla pojazdów autonomicznych"(ang. "A Comparative Analysis of LiDAR SLAM-Based Indoor Navigation for Autonomous Vehicles")[14], gdzie przedstawiono zastosowanie algorytmów SLAM jako fundamentalną technologię do nawigacji robotów mobilnych wewnętrz budynków. Opisano w niej rozwój tej technologii i jej wady przy zastosowaniu w środowiskach z brakiem znacznych cech charakterystycznych z wykorzystaniem technik wizualnych. Z tego powodu coraz częściej stosuje się sensory LiDAR, które pozwalają na uzyskanie dokładnych map bez wcześniej wspomnianych ograniczeń. Dlatego też SLAM na bazie LiDAR jest często wykorzystywane w rozwiązańach przemysłowych, jak np. AGV (ang. "Automated Guided Vehicle") czy roboty magazynowe.

Artykuł "LiDAR i jego zastosowanie w niesamowitych odkryciach w Gwatemali"(ang. "LiDAR and Its Applications to Incredible Discoveries in Guatemala") [8] przedstawia zastosowanie mapowania z użyciem LiDAR-a do tworzenia map 3D grobli znalezionych w Gwatemali. Ukazuje to szerokie zastosowanie technologii LiDAR w różnych dziedzinach, od robotyki po archeologię.

Podsumowując SLAM to jedno z kluczowych zagadnień w dziedzinie robotyki mobilnej. W literaturze przedmiotu można znaleźć wiele prac naukowych, które zajmują się analizą i porównaniem różnych algorytmów SLAM, jak i ich zastosowań w praktyce. W ostatnich latach obserwuje się znaczny rozwój technologii związanych z mapowaniem i nawigacją robotów mobilnych, co pozwala na tworzenie coraz bardziej zaawansowanych systemów autonomicznych.

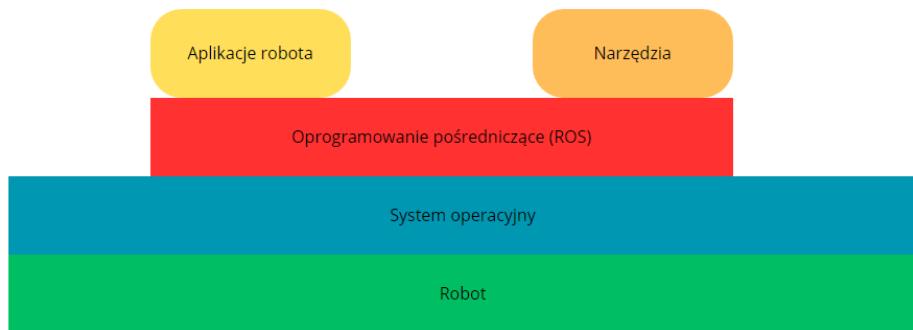
## 2.2 Szczegółowe sformułowanie problemu

Problem postawiony w niniejszej pracy obejmuje dwa główne aspekty. Pierwszy z nich to mapowanie otoczenia, które wymaga efektywnej akwizycji danych z czujników, przetwarzania chmur punktów pobranych z czujnika, estymacji pozycji robota oraz łączenia kolejnych skanów w spójną mapę. Drugi aspekt to autonomiczna nawigacja, gdzie system musi zapewniać precyzyjną lokalizację w znanej mapie, planowanie ścieżki z uwzględnieniem przeszkód oraz dokładną kontrolę ruchu robota.

## 2.3 Przegląd narzędzi i literatury

### 2.3.1 ROS 2

**ROS 2** (ang. "Robotic operation system 2") to platforma programistyczna dla robotów będąca oprogramowaniem pośrednim (ang. "middleware"), czyli warstwą programową pomiędzy systemem operacyjnym, a aplikacjami użytkownika do wykonywania oprogramowania aplikacji w domenie robota [13]. Na poniżej ilustracji przedstawiono położenie takiego pośrednika.

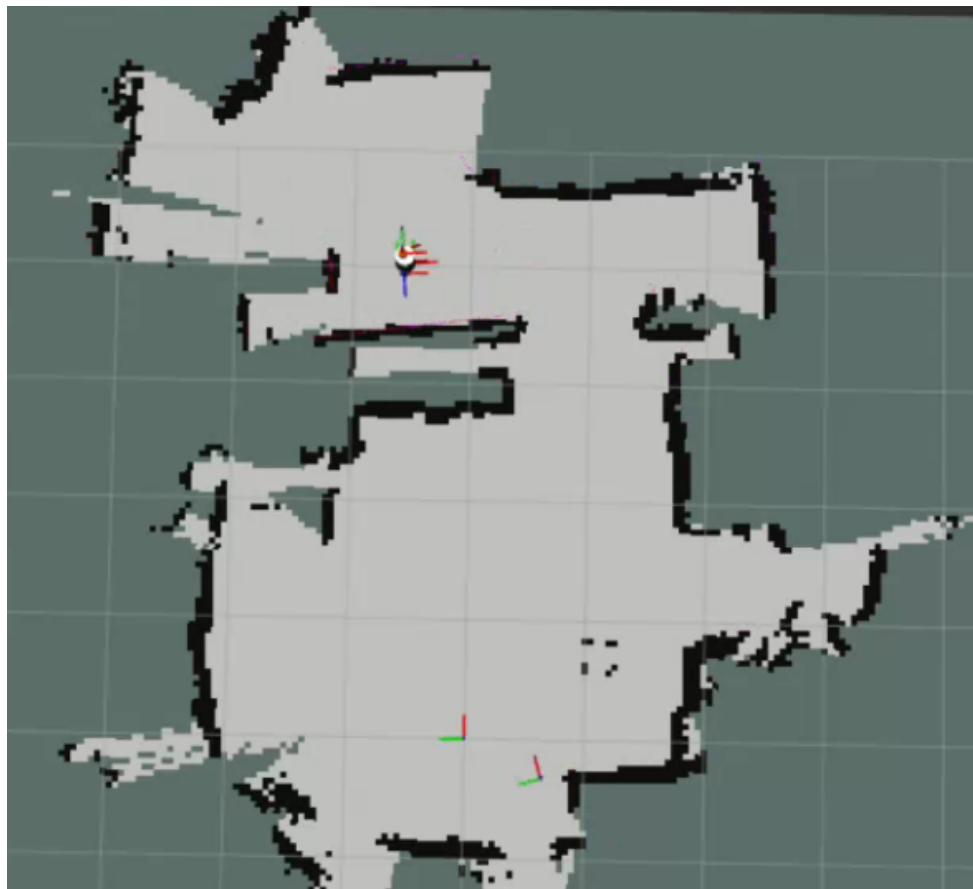


Rysunek 2.1: Reprezentacja pośrednika w systemie robota [13]

Zasadniczo **ROS 2** to otwartoźródłowe oprogramowanie bazujące na usłudze dystrybucji danych - DDS (ang. "Data Distribution Service"), które dostarcza ustandaryzowane narzędzia do organizacji kodu aplikacji w modularne pakiety, zapewniania współbieżnego wykonania kodu na wiele dostępnych plików wykonywalnych, oraz komunikację między tymi modułami podczas równoległego uruchomienia w systemie robota[9].

### 2.3.2 SLAM Toolbox

**SLAM Toolbox** to zestaw narzędzi i rozwiązań do tworzenia map 2D w czasie rzeczywistym, stworzony przez Steve'a Mecenski. W historii rozwoju algorytmów SLAM i **ROS 2** stosowano różne rozwiązania, takie jak GMapping, Karto, Cartographer oraz Hector. Spośród nich tylko Cartographer, opracowany przez Google, oferował możliwość tworzenia map w czasie rzeczywistym [10]. Zastosowanie **SLAM Toolbox** pozwala na tworzenie map w czasie rzeczywistym obszarów, do nawet  $24\ 000\ m^2$  przez niewykwalifikowanych użytkowników. Przykład działania **SLAM Toolbox** przedstawiono na rysunku poniżej.



Rysunek 2.2: Mapa dwóch pomieszczeń utworzona za pomocą SLAM Toolbox [10]

**SLAM Toolbox** oferuje 3 główne tryby pracy:

- **Mapowanie synchroniczne** - Ten tryb pozwala na mapowanie i lokalizację w przestrzeni zachowując dane poprzednich pomiarów. Pozwala to na większą dokładność mapy kosztem szybkości i odporności na przerwania.
- **Mapowanie asynchroniczne** - W tym trybie mapowanie i lokalizacja odbywają się wyłącznie na podstawie aktualnych pomiarów gdy ostatnie pomiary zakończą się i zostanie spełnione kryterium dokładności. Pozwala to na szybsze i mniej podatne na przerwania mapowanie kosztem jakości.
- **Lokalizacja** - W tym trybie robot dopasowuje aktualne pomiary do istniejącej mapy w celu określenia swojej pozycji. System tworzy tymczasowe punkty odniesienia z nowych pomiarów, które są używane do precyzyjnej lokalizacji. Po określonym czasie te tymczasowe punkty są usuwane, przywracając oryginalną mapę. Tryb ten może również działać bez wcześniejszej mapy, wykorzystując tylko lokalne pomiary do nawigacji.

### 2.3.3 Navigation2 (Nav2) i lokalizacja

**Nav2** jest to pakiet narzędzi do nawigacji robotów mobilnych w **ROS 2**. Zawiera on zestaw algorytmów do tworzenia modeli środowiska z sensorów, dynamicznego planowania ścieżki, obliczania prędkości silników i omijania przeszkód. Pakiet wykorzystuje drzewa zachowań (ang. "Behavior Trees") do definiowania zachowań robota, przez implementację wielu niezależnych zadań. Niektóre z nich odpowiadają za np. obliczanie trasy do celu, inne za naprawę błędów, a jeszcze inne za omijanie przeszkód. Dzięki komunikacji między tymi zadaniami, robot jest w stanie wykonywać skomplikowane zadania nawigacyjne [11].

Do lokalizacji robota na mapie można wykorzystać wcześniej opisany **SLAM Toolbox**, jednak w pakuiecie **Nav2** dostępny jest również algorytm **AMCL** (ang. "Adaptive Monte Carlo Localization") [5], który pozwala na lokalizację robota na mapie z wykorzystaniem filtra cząsteczkowego. Algorytm ten polega na generowaniu losowych próbek (cząsteczek) reprezentujących możliwe położenia robota, a następnie porównywaniu ich z pomiarami z czujników. Cząsteczki, które najlepiej pasują do pomiarów są wybierane, a reszta jest odrzucana. W ten sposób algorytm estymuje pozycję robota na mapie [5].

### 2.3.4 ROS 2 Control i sterowanie napędami

ROS 2 Control to platforma do sterowania, zarządzania i komunikacji pomiędzy urządzeniami w robotach z oprogramowaniem **ROS 2** [4]. Rozwiązań to umożliwia w łatwy sposób zarządzanie silnikami, enkoderami, czy innymi urządzeniami w robocie. Pakiet **Diffdrive Arduino** stanowi rozszerzenie tej platformy o interfejs pomiędzy **ROS 2 Control** a Arduino. Pakiet ten pozwala również na sterowanie prędkością silników, odczyt enkoderów, obliczanie odometrii oraz transformację między układem odometrii a układem bazowym [12].

## 2.4 Wybór rozwiązań

Na podstawie analizy dostępnych narzędzi, w projekcie zdecydowano się na wykorzystanie **SLAM Toolbox** do mapowania, **AMCL** do lokalizacji podczas nawigacji, **Nav2** do planowania ścieżki i kontroli ruchu oraz **ROS 2 Control z DiffDrive Arduino** do sterowania napędami. Wybór ten podkutowany jest stabilnością rozwiązań, oraz dobrą integracją komponentów w ekosystemie **ROS 2** oraz aktywnym wsparciem społeczności i dostępnością dokumentacji.

# Rozdział 3

## Wymagania i narzędzia

W niniejszym rozdziale przedstawiono wymagania funkcjonalne systemu, przypadki użycia w formie diagramu UML, szczegółową specyfikację wykorzystanych komponentów sprzętowych oraz metodykę i etapy realizacji projektu.

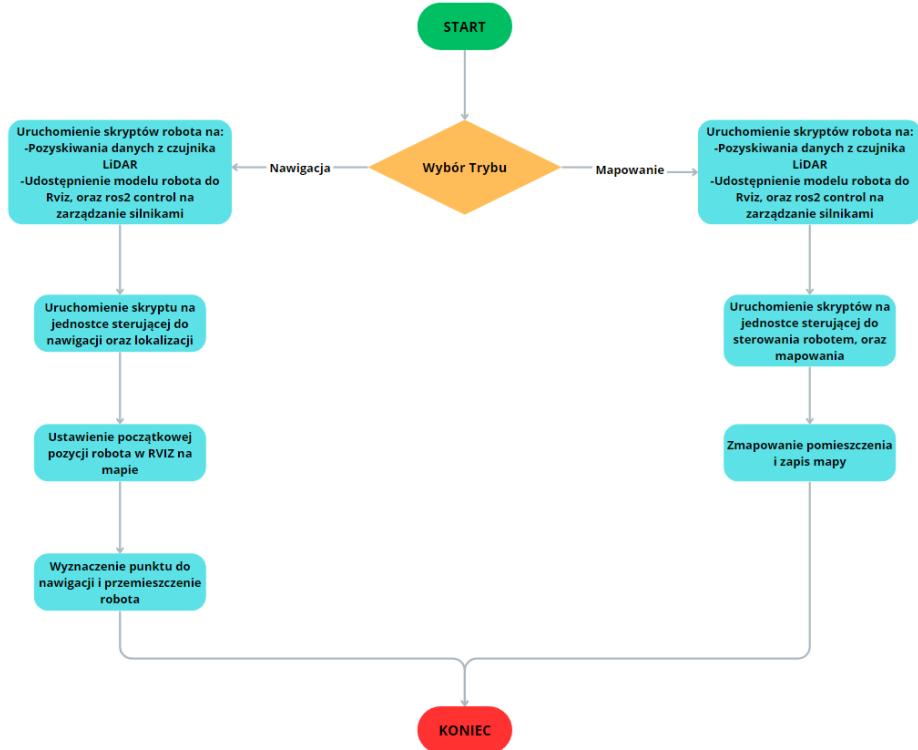
### 3.1 Wymagania funkcjonalne

System powinien realizować następujące funkcje:

- Zdalne sterowanie robotem poprzez klawiaturę (**ros2 run teleop\_twist\_keyboard teleop\_twist\_keyboard -r /cmd\_vel:=/diff\_cont/cmd\_vel\_unstamped**) w celu eksploracji i mapowania otoczenia [7].
- Tworzenie i zapisywanie mapy otoczenia w czasie rzeczywistym .
- Lokalizacja robota na zapisanej mapie z wykorzystaniem algorytmu **AMCL**.
- Autonomiczna nawigacja do wyznaczonych punktów na mapie z omijaniem przeszkód.

## 3.2 Przypadki użycia

Na rysunku 3.1 przedstawiono diagram przypadków użycia systemu. System umożliwia użytkownikowi zdalne sterowanie robotem, tworzenie mapy otoczenia, lokalizację robota na mapie oraz autonomiczną nawigację do wyznaczonych punktów.



Rysunek 3.1: Diagram przypadków użycia systemu

## 3.3 Specyfikacja komponentów

### 3.3.1 Jednostki sterujące

- Raspberry Pi 4 - główny komputer zarządzający robotem:
  - System operacyjny **Ubuntu 22.04**
  - **ROS 2 Humble**
  - Komunikacja przez SSH z jednostką sterującą zachowaniem robota
- Arduino Nano - sterownik silników:
  - Obsługa enkoderów
  - Komunikacja szeregową z Raspberry Pi

### 3.3.2 Napęd

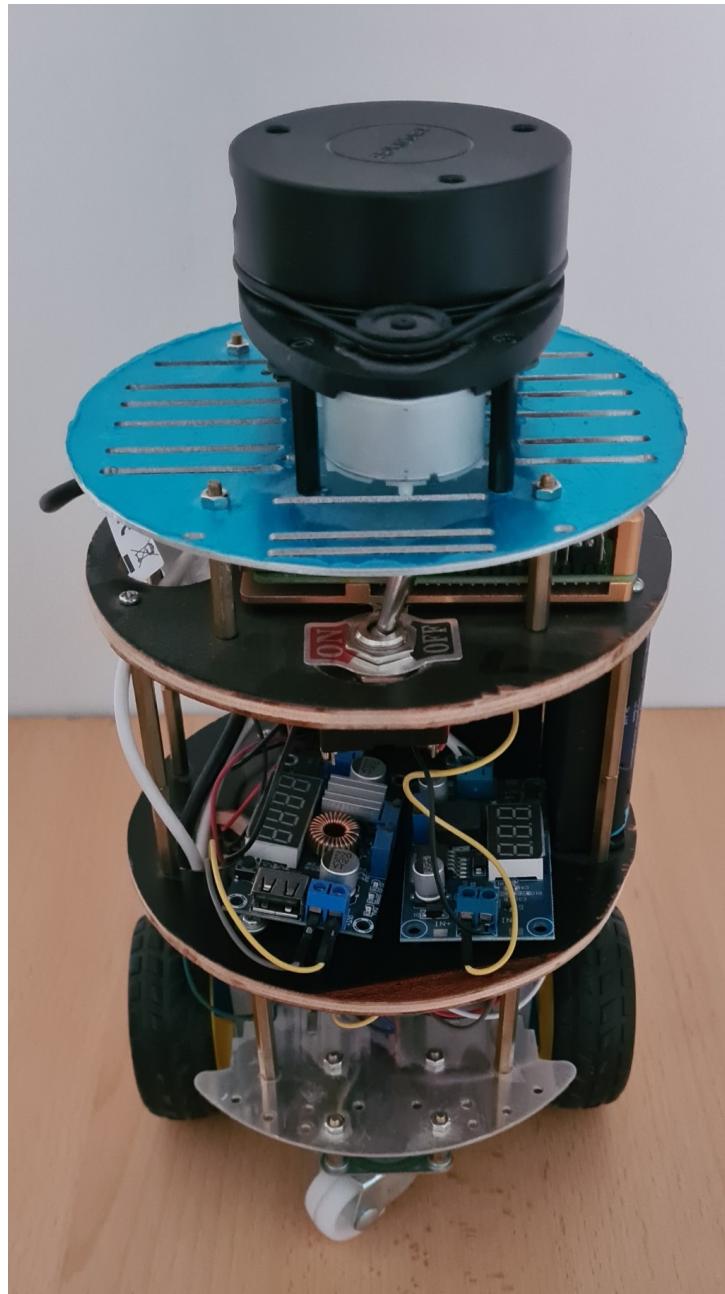
- 2x silnik DC 12V 240RPM z metalową przekładnią
- Wbudowane enkodery magnetyczne Halla
- Sterownik L298N - dwukanałowy mostek H

### 3.3.3 Zasilanie

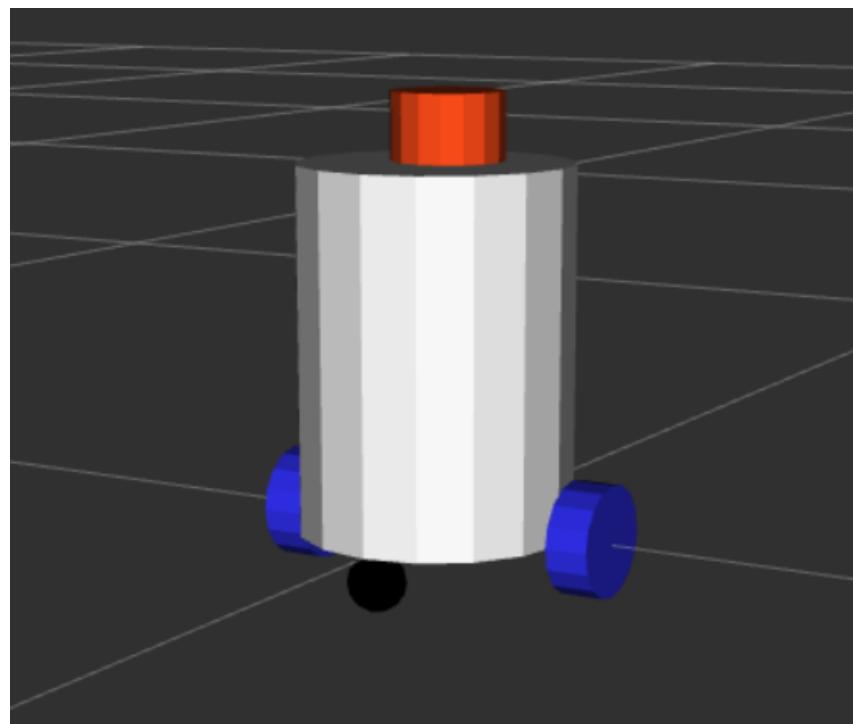
- 6x akumulatory Li-ion 18650:
  - 4 ogniska (2S2P) dla silników
  - 2 ogniska (2S) dla elektroniki
- 2x przetwornica step-down:
  - 12V dla silników
  - 5V dla Raspberry Pi

### 3.4 Budowa robota i sposób połączenia silników z kontrolerem

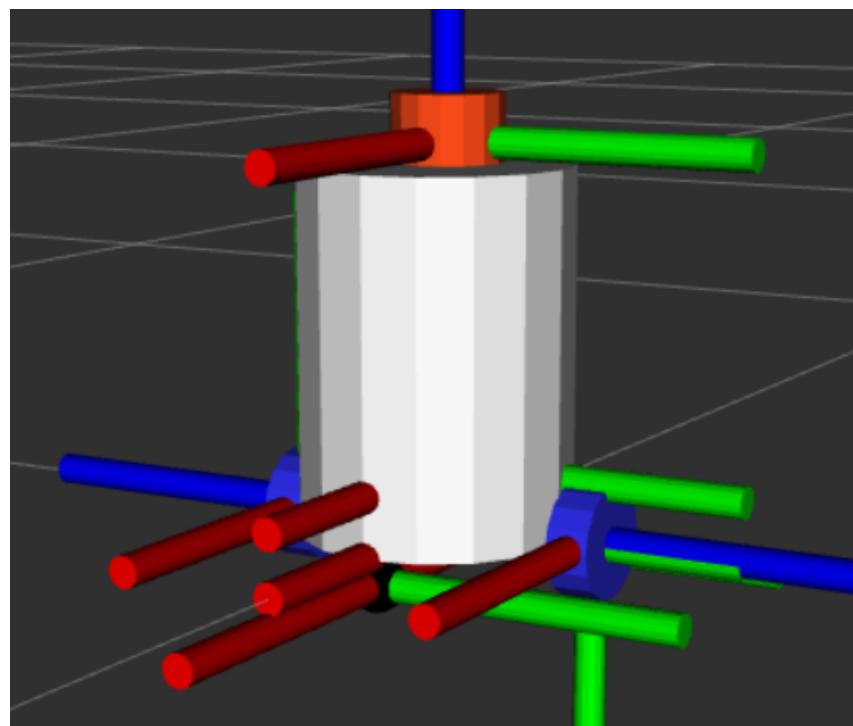
Sekcja ta zawiera zdjęcie zbudowanego robota, wizualizację w **RVIZ**, uproszczony schemat wyjaśniający budowę, oraz schematy połączeń elektrycznych.



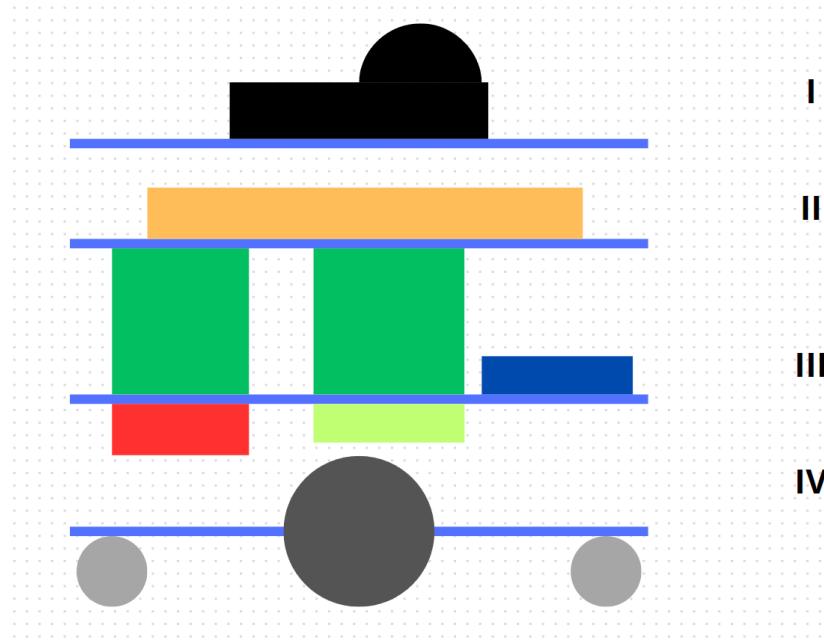
Rysunek 3.2: Zdjęcie przedstawiające zbudowanego robota



Rysunek 3.3: Wizualizacja robota w RVIZ



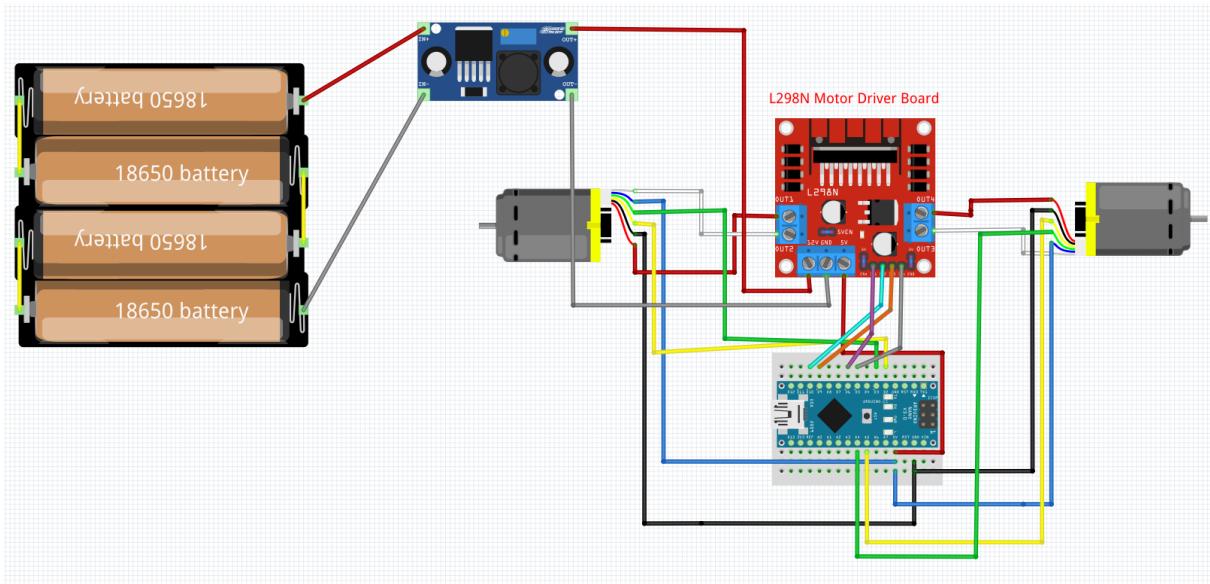
Rysunek 3.4: Wizualizacja robota w RVIZ z osiami



Rysunek 3.5: Uproszczony schemat przedstawiający budowę robota

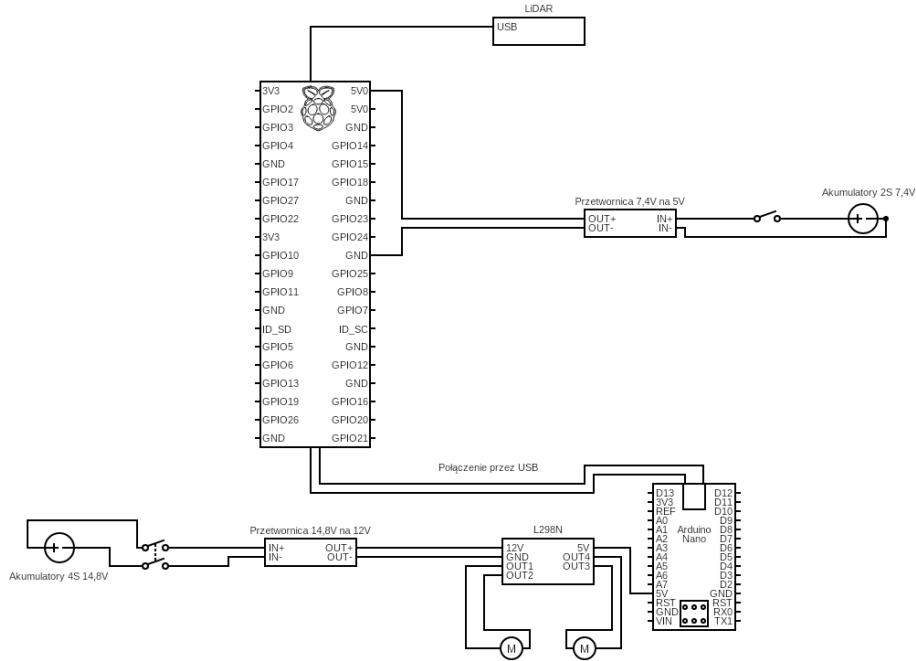
Na schemacie z rysunku 3.5 przedstawiono konkretne poziomy robota, takie jak:

- Poziom I - LiDAR RPLidar A1 (czarny element)
- Poziom II - Raspberry Pi 4 (pomarańczowy element)
- Poziom III - Akumulatory (zielone elementy) i przetwornice (niebieski element)
- Poziom IV - Arduino Nano (jasnozielony element), sterownik L298N (czerwony element) i koła (jasnoszare elementy są kołami podporowymi, a ciemnoszare to koła napędowe)



Rysunek 3.6: Schemat połączenia silników z Arduino i L298N

Na zaprezentowanym schemacie rys 3.6 przedstawiono sposób połączenia silników z Arduino i sterownikiem L298N. Silniki DC z enkoderami są zasilane z akumulatorów Li-ion, a sterowane przez Arduino Nano za pomocą sterownika L298N. Enkodery są podłączone do Arduino, które odczytuje impulsy i oblicza prędkość i położenie robota. Komunikacja między Arduino a Raspberry Pi odbywa się przez port szeregowy, co umożliwia przesyłanie danych o prędkości i położeniu robota.



Rysunek 3.7: Schemat elektryczny

Na schemacie 3.7 przedstawiono sposób połączenia wszystkich komponentów elektrycznych w robocie. Akumulatory zasilają silniki i elektronikę, a przetwornice step-down dostarczają odpowiednie napięcia do poszczególnych komponentów. Sterownik L298N steruje silnikami, a Arduino Nano odczytuje enkodery i przesyła dane do Raspberry Pi. Raspberry Pi zarządza robotem, odbiera dane z czujników i steruje silnikami.

## 3.5 Metodyka i etapy realizacji

Realizacja projektu podzielona jest na 4 etapy, które obejmują przygotowanie platformy sprzętowej, implementację sterowania napędem, integrację sensorów oraz implementację oprogramowania. Każdy etap składa się z kilku zadań, które muszą zostać wykonane w celu osiągnięcia zamierzonego celu. Poniżej przedstawiono szczegółowy opis poszczególnych etapów.

### 3.5.1 Etap 1: Przygotowanie platformy sprzętowej

- Instalacja systemu **Ubuntu 22.04** na Raspberry Pi i komputerze sterującym.
- Konfiguracja połączenia SSH
- Instalacja **ROS 2 Humble** na Raspberry Pi i komputerze sterującym

### 3.5.2 Etap 2: Implementacja sterowania napędem

- Podłączenie silników do sterownika L298N
- Programowanie Arduino - obsługa silników i enkoderów
- Implementacja komunikacji szeregowej z Raspberry Pi

### 3.5.3 Etap 3: Integracja sensorów

- Montaż i konfiguracja LiDAR-a
- Kalibracja czujników
- Opracowanie układu mechanicznego i obudowy

### 3.5.4 Etap 4: Implementacja oprogramowania

- Konfiguracja pakietów **ROS 2**:
  - **SLAM Toolbox** do mapowania
  - **Nav2** do nawigacji z **AMCL** do lokalizacji
  - **Diffdrive Arduino** do sterowania napędem
- Integracja i testy systemu



# Rozdział 4

## Specyfikacja użytkowa

W tym rozdziale przedstawiono wymagania użytkownika oraz specyfikację funkcjonalną systemu. Opisano wymagania sprzętowe i programowe, sposób aktywacji i korzystania z robota z przykładem działania, administrację systemem, kwestie bezpieczeństwa, oraz scenariusze korzystania z systemu.

### 4.0.1 Wymagania sprzętowe i programowe

Projekt ten stworzony był z myślą o następujących wymaganiach dla robota:

Sprzętowe:

- Poruszać się w przestrzeni za pomocą silników, czyli np. przemieszczanie się do przodu, do tyłu, skręcanie w lewo i w prawo po korytarzach, czy w pomieszczeniach z równym podłożem.
- Skanować pomieszczenia za pomocą LiDAR-a, czyli zbieranie danych o otoczeniu wokół robota.

Programowe:

- Tworzyć mapę otoczenia, czyli zapisywanie danych z LiDAR-a w formie mapy 2D w czasie rzeczywistym i wizualizację tych danych w programie **RVIZ**.
- Lokalizować się na mapie, czyli określanie pozycji robota na zapisanej mapie.
- Nawigować do wyznaczonych punktów, czyli planowanie trasy do punktów na mapie i omijanie przeszkód.

#### 4.0.2 Sposób aktywacji i korzystania z robota z przykładem działania

Sekcja ta zawiera szczegółowy opis korzystania z robota, od uruchomienia, sterowanie, mapowanie, zapis mapy, po lokalizację i nawigację.

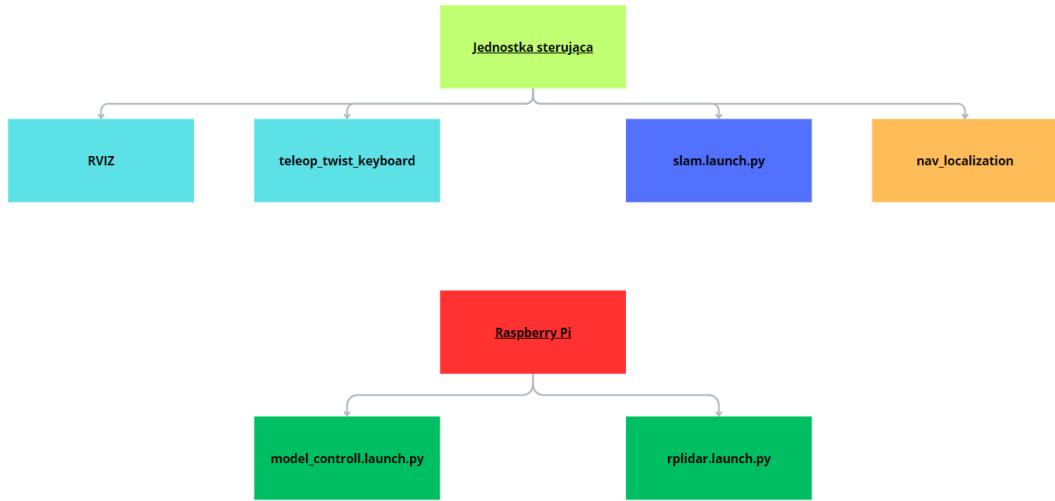


Rysunek 4.1: Zdjęcie środowiska testowego.

Procesy te przeprowadzane były na korytarzu widocznym na zdjęciu 4.1. Takie środowisko dostarczyło wystarczająco dużo miejsca do nawigacji i mapowania dla robota. Zapewniono również równe podłożę, co pozwoliło na stabilne i bezpieczne poruszanie się robota.

Pierwszym krokiem w procesie eksploatacji robota jest uruchomienie go, w tym celu należy włączyć zasilanie dla Raspberry Pi i silników załączając przełączniki przy Raspberry Pi i u podstawy robota.

Następnie należy przygotować terminale na jednostce sterującej (dwa terminale mają być połączone przez protokół SSH z Raspberry Pi, a cztery terminale mają być przygotowane na samej jednostce sterującej do uruchomienia późniejszych skryptów **ROS 2**).



Rysunek 4.2: Schemat przedstawiający na jakim urządzeniu należy uruchomić poszczególne skrypty.

Rysunek 4.2 przedstawia na jakim urządzeniu należy uruchomić poszczególne skrypty. Dla jednostki sterującej są to **RVIZ** do wizualizacji mapowania, skrypt **teleop\_twist\_keyboard** do manualnego sterowania, **slam.launch.py** do tworzenia mapy, oraz **nav\_localization.launch.py** do lokalizacji i nawigacji. Dla Raspberry Pi są to skrypty: **model\_controll.launch.py** - odpowiedzialne za sterowanie silnikami i udostępnienie modelu robota do RVIZ, oraz **rplidar.launch.py** odpowiedzialny za odczyt danych z LiDAR-a i udostępnianie tych informacji innym skryptom.

Przygotowanie terminali polega na odpowiednim: Dla jednostki sterującej: wejściu do katalogu roboczego, uruchomienie komend `source /opt/ros/humble/setup.bash`, oraz `source /install/setup.bash`.

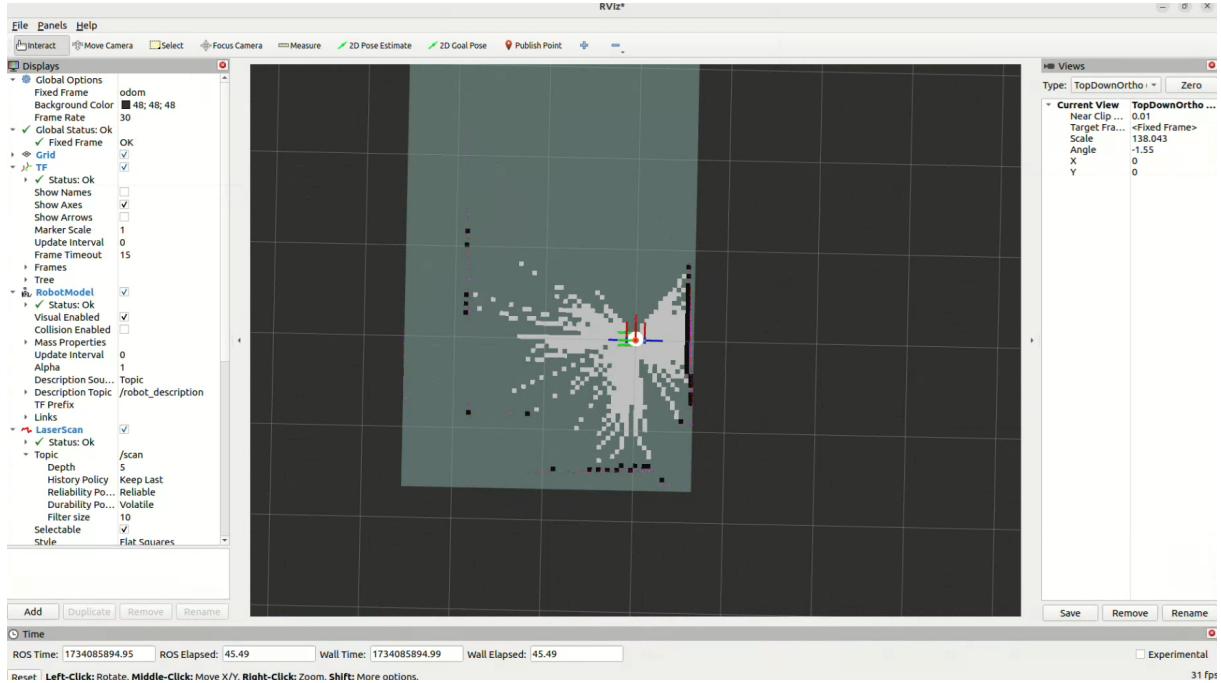
Dla Raspberry Pi: wejściu do katalogu roboczego, uruchomienie komend `source /opt/ros/humble/setup.bash`, oraz `source /install/setup.bash`.

Następnie, należy uruchomić 2 skrypty na Raspberry Pi, które odpowiadają za uruchomienie odpowiednich węzłów **ROS 2**. Pierwszy skrypt odpowiada za uruchomienie węzła odpowiedzialnego za sterowanie silnikami i udostępnienie modelu robota do RVIZ, a drugi za uruchomienie węzła odpowiedzialnego za odczyt danych z LiDAR-a. Odpowiednie komendy to: `ros2 launch robot_slam model_controll.launch.py` oraz `ros2 launch robot_slam rplidar.launch.py`.

Po uruchomieniu tych skryptów, należy uruchomić program **RVIZ**, który pozwala na wizualizację mapy i położenia robota. Komenda to: `rviz2`.

Następnym krokiem jest uruchomienie skryptu odpowiedzialnego za sterowanie robotem za pomocą klawiatury. Komenda to: `ros2 run teleop_twist_keyboard teleop_twist_keyboard -r /cmd_vel:=/diff_cont/cmd_vel_unstamped`.

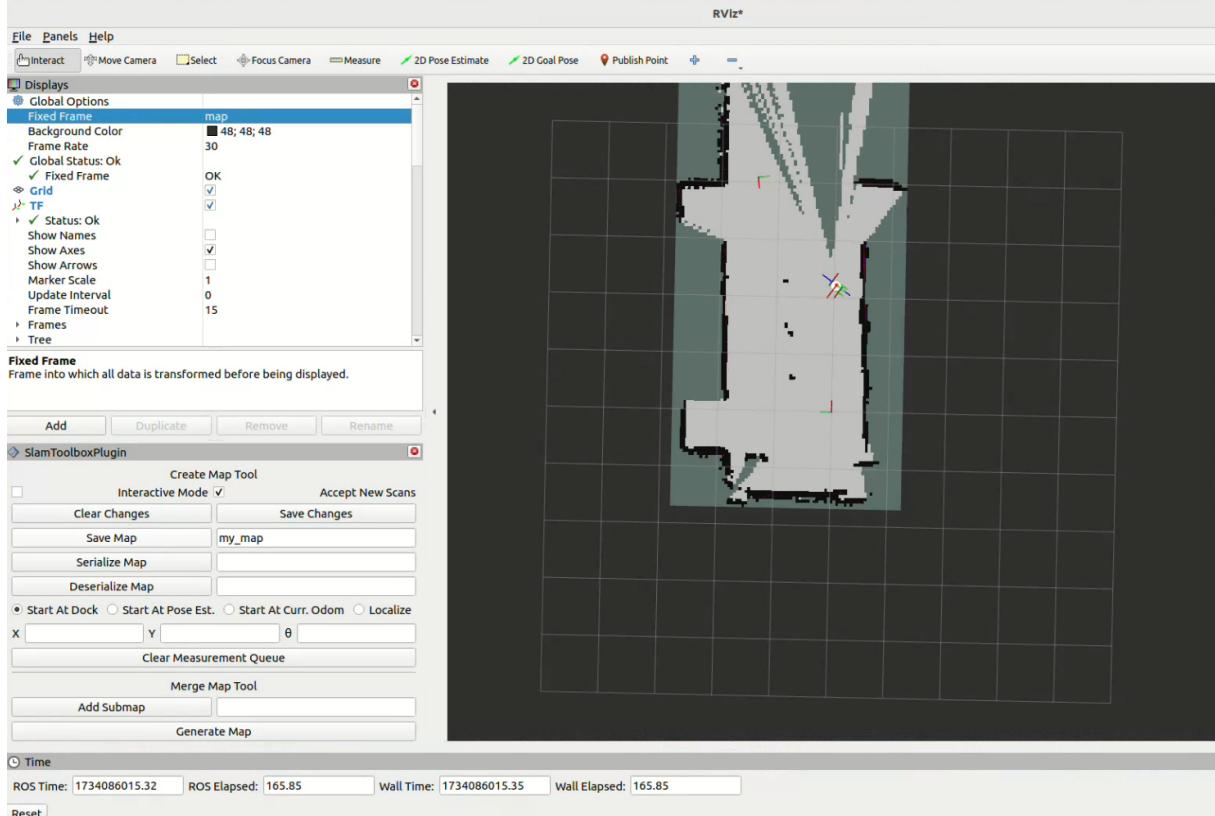
Kolejnym krokiem jest uruchomienie skryptu odpowiedzialnego za mapowanie otoczenia. Komenda to: `ros2 launch robot_slam slam.launch.py`. W tym momencie gdy robot przemieszcza się przez ręczne sterowanie za pomocą `teleop_twist_keyboard`, pomieszczenie jest mapowane.



Rysunek 4.3: Okno RVIZ po uruchomieniu skryptów.

Na rysunku 4.3 przedstawiono okno **RVIZ** po uruchomieniu skryptów. Na tym etapie można zauważać, że robot jest widoczny na środku ekranu, a także widoczne są dane z LiDAR-a (w formie fioletowych i czerwonych punktów), które są wykorzystywane do mapowania pomieszczenia. Jaśniejsze pola to pusta przestrzeń, natomiast czarne to przeszkody i ściany.

Gdy utworzona zostanie zadowalająca mapa, należy ją zapisać. Można to wykonać przez komendę `ros2 run nav2_map_server map_saver -f map`, lub przez dodanie panelu do **RVIZ** z zestawu narzędzi **SLAM Toolbox** i zapisanie mapy z poziomu tego panelu.



Rysunek 4.4: Zmapowane pomieszczenie i zapis mapy jako plik my\_map.

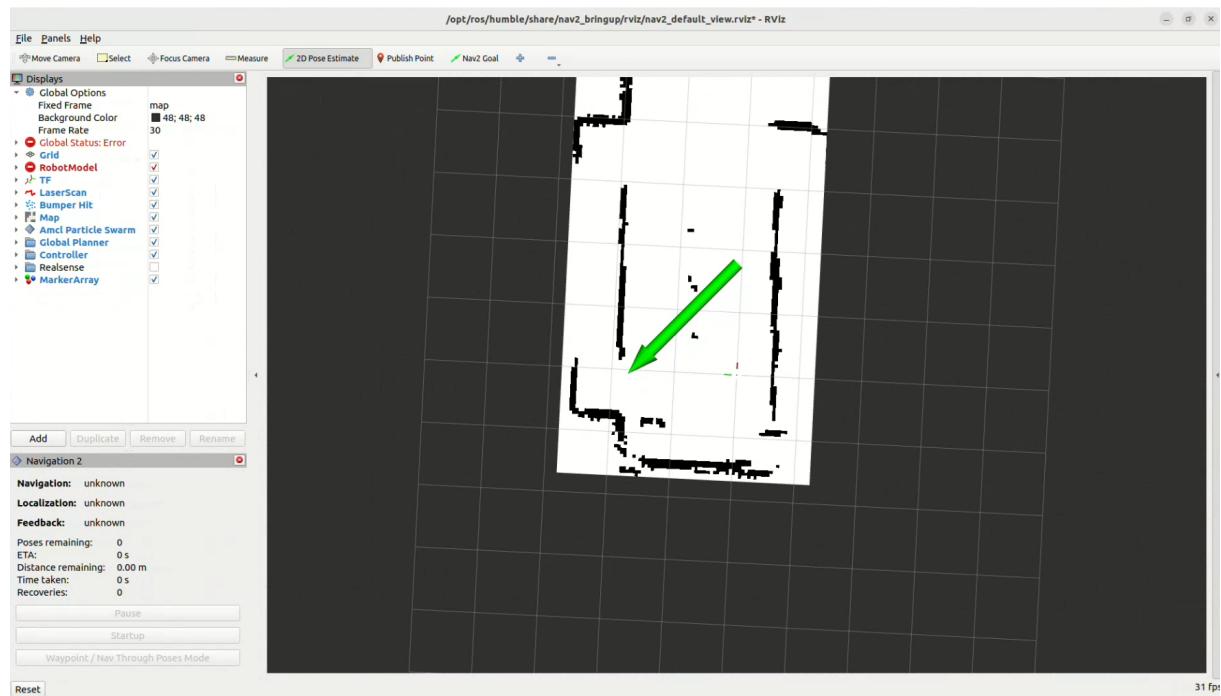
Rysunek 4.4 przedstawia mapę korytarza. Na mapie widoczne są ściany, oraz pojedyncze czarne punkty, jakimi są nogi od stołu. Po lewej stronie rysunku widoczny jest panel **Slam Toolbox**, gdzie wypełniona jest nazwa pliku mapy. Należy kliknąć przycisk **Save Map**, aby zapisać mapę jako pliki pgm i yaml.

Z tak zapisaną mapą można zamknąć skrypty odpowiedzialne za **sterowanie, mapowanie i RVIZ**.

Następnie, należy uruchomić skrypt odpowiedzialny za lokalizację robota na mapie, oraz nawigację robota.

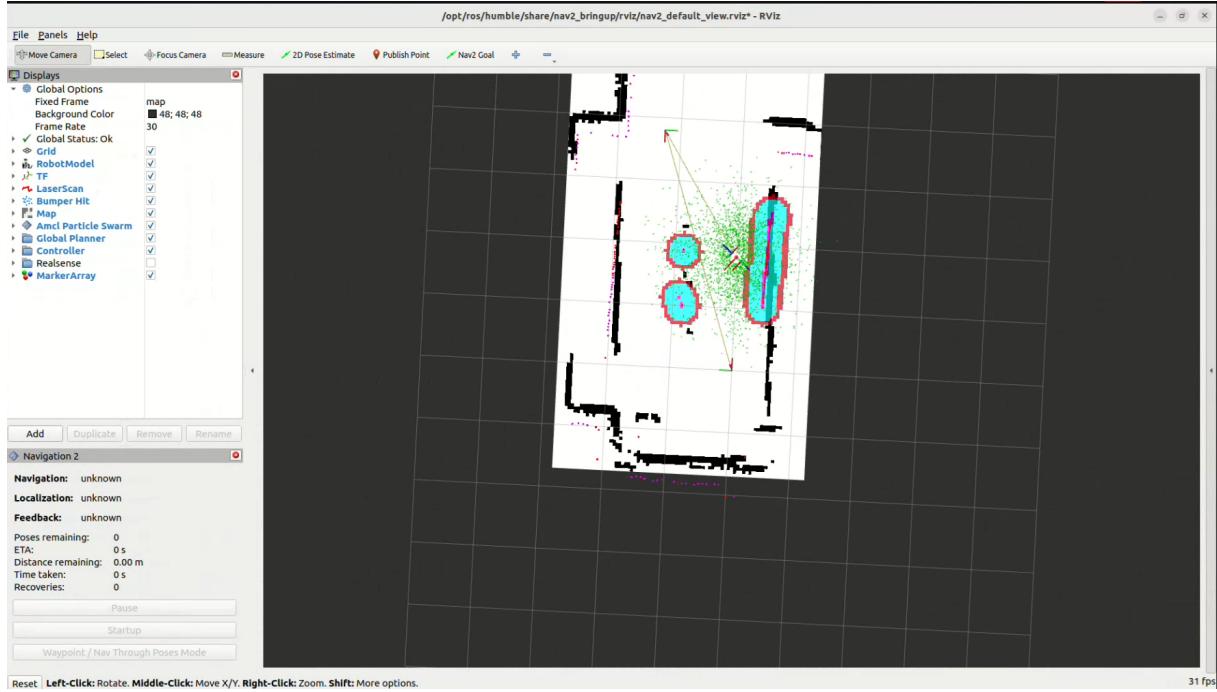
Komenda to: **ros2 launch robot\_slam nav\_localization.launch.py**. W wyniku uruchomienia tego programu uruchomiony zostaje nowy ekran **RVIZ** z widoczną wcześniej zapisaną mapą.

Należy na niej umieścić robota przez wybranie z górnego zestawu narzędzi **2D Pose Estimate**, a następnie kliknięcie na mapie w miejscu gdzie znajduje się robot, należy w tym momencie przez obrócenie w odpowiednim kierunku zielonej strzałki ustalić kierunek w jakim znajduje się robot, w wyniku czego algorytm **AMCL** tworzy chmurę przewidywanych punktów w których może znajdować się robot.



Rysunek 4.5: RVIZ po uruchomieniu skryptu do lokalizacji i nawigacji i wybraniu pozycji robota.

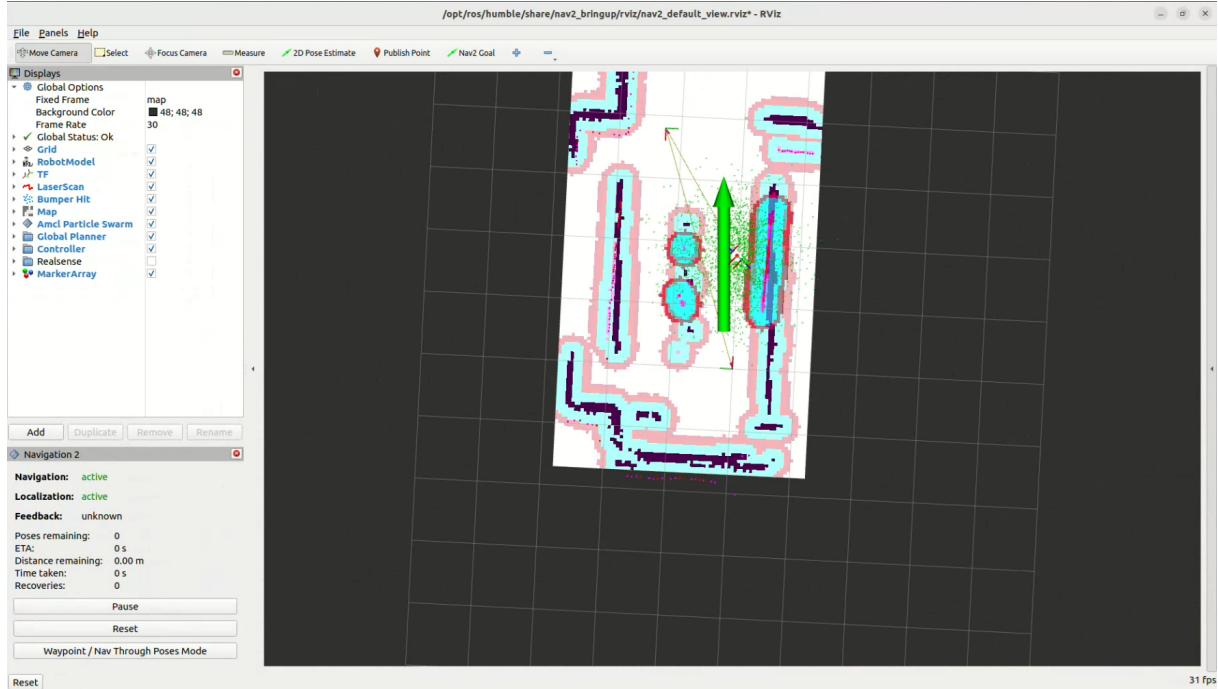
Na rysunku 4.5 przedstawiono okno **RVIZ** po uruchomieniu skryptu do lokalizacji i nawigacji. Widoczna jest wcześniej zapisana mapa, gdzie puste przestrzenie teraz widoczne są jako białe pola, a przeszkody jako czarne. Na mapie zaznaczono zieloną strzałką pozycję robota.



Rysunek 4.6: Wizualizacja rozproszonych punktów lokalizacji robota.

Rysunek 4.6 ukazuje chmurę przewidywanych punktów lokalizacji robota wyznaczanych przez algorytm **AMCL**, model robota, który pojawił się w miejscu gdzie wcześniej zaznaczono pozycję robota. Widoczna jest również lokalna mapa kosztów w postaci jasnoniebieskich i jasnofioletowych pól. Niebieskie pola oznaczają strefę niebezpieczną wokół przeszkodej, natomiast fioletowe pola oznaczają wykrywane w czasie rzeczywistym przeszkodej. Na podstawie tej mapy robot koryguje trasę o nowo napotkane przeszkodej, które nie były wcześniej na mapie.

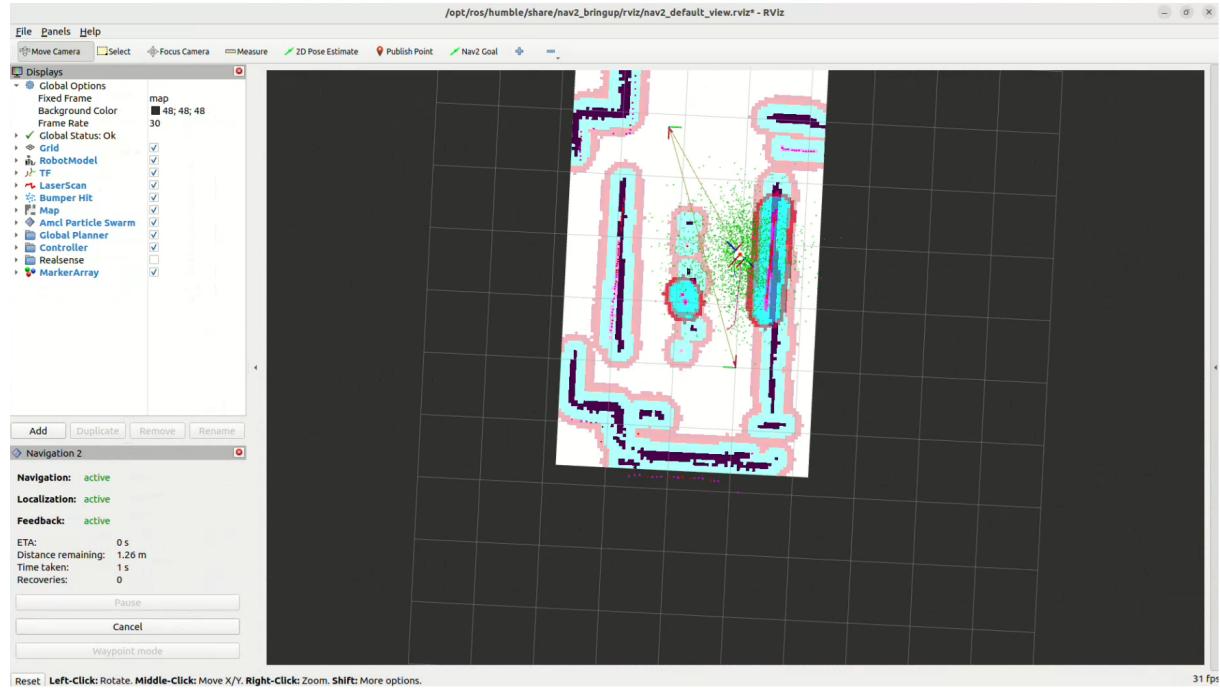
Następnie należy wybrać cel do którego robot ma się przemieścić, przez wybranie z górnego zestawu narzędzi **Nav2 Goal**, a następnie kliknięcie na mapie w miejscu gdzie ma się znaleźć cel z ustawieniem zielonej strzałki w kierunku w jaki ma się ustawić. Na



Rysunek 4.7: Wybór celu robota.

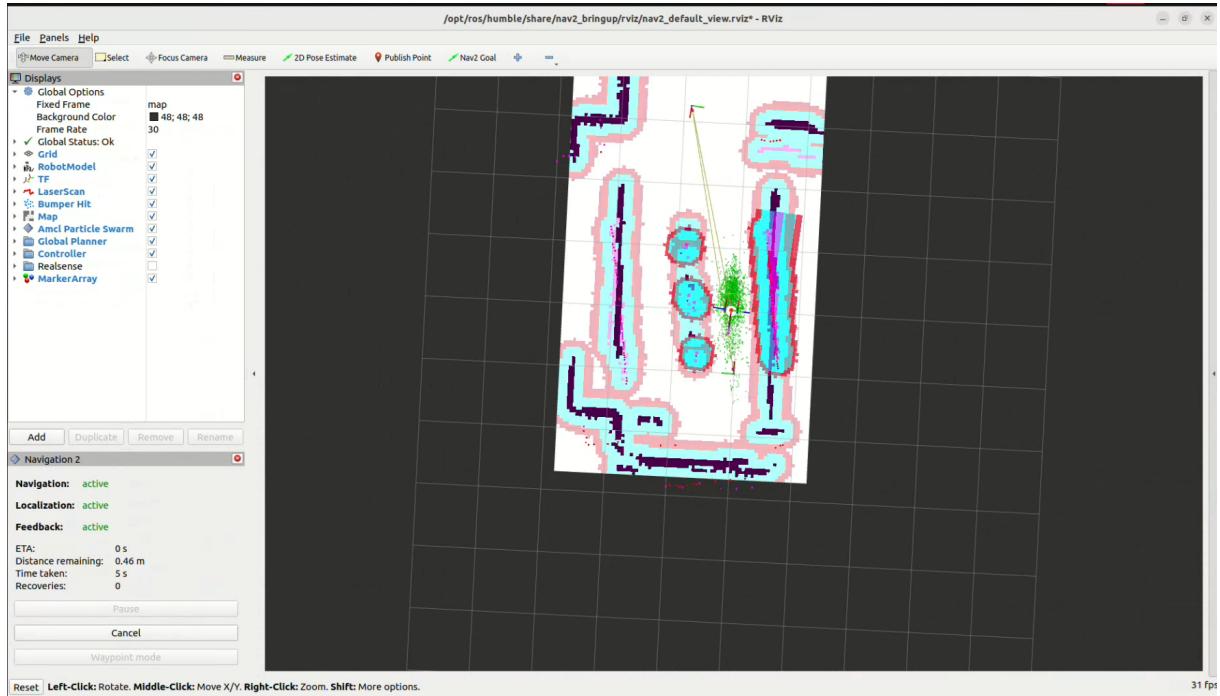
rysunku 4.7 widoczne jest okno **RVIZ**, po wybraniu celu do którego robot ma się przemieścić. Na mapie widoczna jest zielona strzałka wskazująca cel i kierunek w jakim ma się ustawić robot. Pojawia się również globalna mapa kosztów w postaci niebieskich obramowań. Widoczne wcześniej na mapie ściany i przeszklody zostały zmienione z czarnych na ciemnofioletowe i pojawiło się wokół nich niebieskie obramowanie jako globalna mapa kosztów. Globalna mapa kosztów wykorzystywana jest do wyznaczenia trasy w oparciu o dane z lokalizacji i mapy.

Robot po wybraniu celu zaczyna poruszać się w kierunku celu, omijając przeszkody na swojej drodze.



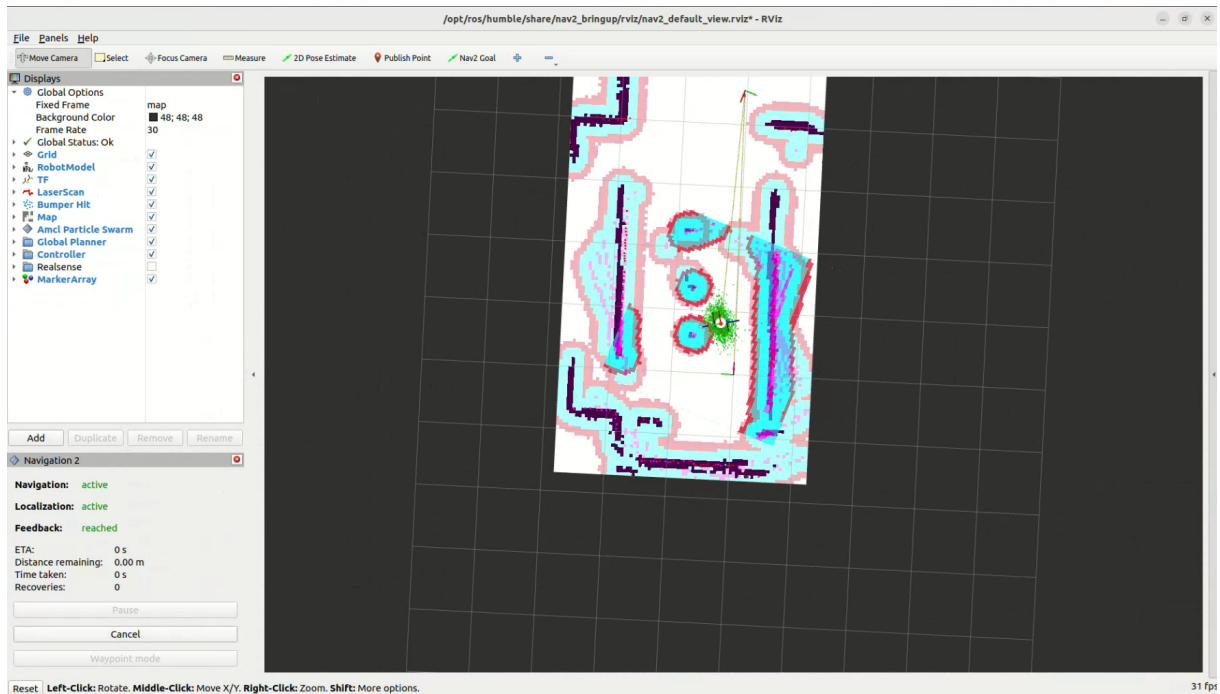
Rysunek 4.8: Wyznaczenie trasy.

Rysunek 4.8 przedstawia jak wyznaczona zostaje trasa robota jako różowa linia na podstawie globalnej mapy kosztów. Linia ta zmienia się w przypadku wykrycia przeszkody na lokalnej mapie kosztów.



Rysunek 4.9: Nawigacja robota i zmniejszanie się chmury przewidywanej lokalizacji robota.

Na rysunku 4.9 ukazane jest jak robot porusza się w kierunku celu, omijając przeszkody na swojej drodze. Widać jak chmura przewidywanej lokalizacji robota zmniejsza się w miarę poruszania się robota.



Rysunek 4.10: Dotarcie robota do celu.

Rysunek 4.10 pokazuje stan gdy robot dotarł do celu. Chmura przewidywanej lokalizacji robota znaczco zmniejszyła się, co wynika z utwierdzenia pozycji robota. Po lewej stronie widoczny jest panel **Nav2** gdzie pole informacji zwrotnej (ang. "Feedback") zwraca wiadomość o dotarciu robota do celu.

## 4.1 Administracja systemem

System nie wymaga specjalnej administracji, jednakże w przypadku problemów z działaniem, można skorzystać z narzędzi diagnostycznych dostępnych w **ROS 2**, jak i z dokumentacji dostępnej na stronie internetowej **ROS 2**. W celu modyfikacji np. prędkością poruszania robota, przy ręcznym sterowaniu, można zmienić prędkość za pomocą klawiszy q i z, które kolejno zwiększają i zmniejszają maksymalną prędkość, jak jest to widoczne na rysunku poniżej 4.11.

```
This node takes keypresses from the keyboard and publishes them
as Twist/TwistStamped messages. It works best with a US keyboard layout.
-----
Moving around:
  u    i    o
  j    k    l
  m    ,    .

For Holonomic mode (strafing), hold down the shift key:
-----
  U    I    O
  J    K    L
  M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5          turn 1.0
```

Rysunek 4.11: Terminal z uruchomionym skryptem teleop\_twist\_keyboard.

Do modyfikacji prędkości podczas nawigacji, można zmienić prędkość w pliku konfiguracyjnym **Nav2 (nav2\_params.yaml)**. Odpowiadają za to następujące parametry:

---

```
1  velocity_smoother:
2    ros__parameters:
3      use_sim_time: True
4      smoothing_frequency: 20.0
5      scale_velocities: False
6      feedback: "OPEN_LOOP"
7      max_velocity: [0.26, 0.0, 1.0]
8      min_velocity: [-0.26, 0.0, -1.0]
9      max_accel: [2.5, 0.0, 3.2]
10     max_decel: [-2.5, 0.0, -3.2]
11     odom_topic: "odom"
12     odom_duration: 0.1
13     deadband_velocity: [0.0, 0.0, 0.0]
14     velocity_timeout: 1.0
```

---

Rysunek 4.12: Fragment kodu z pliku nav2\_params.yaml

Można tutaj dostosować parametry, takie jak maksymalna prędkość aplikowana dla każdej z osi [x, y, theta], maksymalne przyspieszenie dla każdej z osi [x, y, theta], oraz maksymalne hamowanie dla każdej z osi [x, y, theta].

## 4.2 Kwestie bezpieczeństwa

Robot opracowany został z myślą o omijaniu przeszkód, nawet tych których nie było na wcześniej stworzonej mapie pozwalając na bezpieczne poruszanie się w przestrzeni. Należy jednak pamiętać że robot ten nie został przygotowany do pracy w środowiskach bez równego podłoża, dlatego należy unikać przemieszczania się po nierównym terenie co może prowadzić do przewrócenia się robota.

## 4.3 Scenariusze korzystania z systemu

Robot ten pozwala na mapowanie różnego rodzaju pomieszczeń, oraz na nawigację do wyznaczonych punktów, co pozwala na zastosowanie go w różnego rodzaju zastosowaniach, jak np. inspekcja pomieszczeń. Testy przeprowadzane były w małych pomieszczeniach nie przekraczających  $40\text{ m}^2$ , jednakże robot ten jest w stanie mapować pomieszczenia nawet do  $24\,000\text{ m}^2$  co wynika z dokumentacji **Nav2** [11].

# Rozdział 5

## Specyfikacja techniczna

Rozdział ten zawiera specyfikację techniczną systemu, w tym opis wykorzystanych technologii, architekturę systemu, strukturę systemu, opis działania algorytmów oraz diagramy UML wyjaśniające działanie poszczególnych programów.

### 5.1 Wykorzystane technologie

Projekt wykorzystuje następujące technologie:

- **ROS 2 Humble** - platforma do tworzenia oprogramowania dla robotów zainstalowana na Raspberry Pi i komputerze sterującym.
- **SLAM Toolbox** - pakiet do mapowania otoczenia, zainstalowany za pomocą komendy `sudo apt install ros-humble-slam-toolbox`
- **Nav2** - pakiet do nawigacji robota wraz z algorytmem **AMCL** do lokalizacji, zainstalowany za pomocą komendy `sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup`
- **DiffDrive Arduino** - pakiet będący rozszerzeniem zestawu narzędzi **ROS 2 Control**, zapewniający jego funkcjonalność, jaką jest zarządzanie różnego rodzaju kontrolerami, wraz z połączeniem między **ROS 2** i **ROS 2 Arduino Bridge** [12].
- **ROS Arduino Bridge** - pakiet zapewniający interfejs między **ROS 2** i Arduino, umożliwiający dostęp do sensorów analogowych i cyfrowych, serwomechanizmów PWM oraz danych odometrycznych z enkoderów. Jest to kompletnie rozwiązanie do sterowania robotem kontrolowanym przez Arduino w środowisku **ROS 2** [6].

## 5.2 Architektura systemu

Sposób działania systemu oparty jest na współpracujących ze sobą programach, których sposób połączenia zaprezentowano na rysunku 5.1.



Rysunek 5.1: Wizualizacja połączeń skryptów w systemie

## 5.3 Struktura systemu i objaśnienie działania algorytmów

W tej sekcji zawarto szczegółowe opisy funkcjonalności każdego ze skryptów 5.1, które są odpowiedzialne za działanie robota.

Wszystkie skrypty znajdują się w załączniku **Oprogramowanie**.

### 5.3.1 Arduino

Oprogramowanie to odpowiedzialne jest za sterowanie silnikami i zbieranie danych z enkoderów. Opiera się na pakiecie **ROS Arduino Bridge**, który zapewnia interfejs między **ROS 2** i Arduino.

Po przetestowaniu skryptów stwierdzono brak potrzeby modyfikacji kodu źródłowego, więc nie wprowadzono żadnych zmian. Główne komponenty tego oprogramowania to:

- **ROSArduinoBridge.ino** - skrypt obsługuje komunikację szeregową z prędkością 57600 bodów (jednostka szybkości transmisji danych) i może kontrolować silniki poprzez różne sterowniki (np. Pololu VNH5019, MC33926 czy **L298N**). Główna pętla programu ciągle nasłuchuje komend przez port szeregowy, wykonuje obliczenia PID w regularnych odstępach czasu oraz aktualizuje stan serwomechanizmów jeśli są używane.
- **motor\_driver.ino** - plik ten zawiera różne definicje sterowników silników, gdzie w tym projekcie wykorzystano **L298N**, który jest dwukanałowym mostkiem H. Program obsługuje również inne sterowniki jak Pololu VNH5019 czy MC33926.

W kodzie zaimplementowano trzy podstawowe funkcje:

- **initMotorController()** - inicjalizacja sterownika silników
- **setMotorSpeed(int i, int spd)** - ustawienie prędkości pojedynczego silnika
- **setMotorSpeeds(int leftSpeed, int rightSpeed)** - funkcja do ustawiania prędkości obu silników

Dla sterownika **L298N** prędkość silników jest kontrolowana przez sygnały PWM wysyłane na odpowiednie piny. Ujemne wartości prędkości powodują obrót silnika w przeciwnym kierunku. Program zapewnia też ograniczenie maksymalnej prędkości do 255 (**8-bit PWM**).

- **encoder\_driver.ino** - Ten plik zawiera definicje obsługi enkoderów do odczytu pozycji kół robota. W implementacji wykorzystano przerwania do dokładnego zliczania impulsów z enkoderów.

Plik zawiera między innymi:

- Definicje wejść/wyjść dla enkoderów lewego i prawego koła
- Wykorzystanie przerwań PCINT (ang. "Pin Change Interrupt") do zliczania impulsów
- Funkcje do odczytu i resetowania liczników enkoderów

---

```
1  /* Interrupt routine for LEFT encoder */
2  ISR (PCINT2_vect){
3      static uint8_t enc_last=0;
4      enc_last <<=2; //shift previous state
5      enc_last |= (PIND & (3 << 2)) >> 2; //read current state
6      left_enc_pos += ENC_STATES[(enc_last & 0x0f)];
7  }
8  /* Interrupt routine for RIGHT encoder */
9  ISR (PCINT1_vect){
10     static uint8_t enc_last=0;
11     enc_last <<=2; //shift previous state
12     enc_last |= (PINC & (3 << 4)) >> 4; //read current state
13     right_enc_pos += ENC_STATES[(enc_last & 0x0f)];
14 }
```

---

Rysunek 5.2: Fragment kodu źródłowego z pliku encoder\_driver.ino z obsługą przerwań do precyzyjnego zliczania impulsów z enkoderów

- **encoder\_driver.h**, oraz **motor\_driver.h** - pliki zawierające deklaracje funkcji i zmiennych, definicje wejść/wyjść dla enkoderów i sterowników silników.

---

```

1  ****
2  Motor driver function definitions – by James Nugen
3  ****
4
5 #ifdef L298_MOTOR_DRIVER
6     #define RIGHT_MOTOR_BACKWARD 5
7     #define LEFT_MOTOR_BACKWARD 6
8     #define RIGHT_MOTOR_FORWARD 9
9     #define LEFT_MOTOR_FORWARD 10
10    #define RIGHT_MOTOR_ENABLE 12
11    #define LEFT_MOTOR_ENABLE 13
12#endif
13
14 void initMotorController();
15 void setMotorSpeed(int i, int spd);
16 void setMotorSpeeds(int leftSpeed, int rightSpeed);

```

---

Rysunek 5.3: Kod źródłowy z pliku motor\_driver.h

---

```

1  /*
2  ****
3  Encoder driver function definitions – by James Nugen
4  ****
5
6 #ifdef ARDUINO_ENC_COUNTER
7 //below can be changed, but should be PORTD pins;
8 //otherwise additional changes in the code are required
9 #define LEFT_ENC_PIN_A PD2 //pin 2
10 #define LEFT_ENC_PIN_B PD3 //pin 3
11
12 //below can be changed, but should be PORTC pins
13 #define RIGHT_ENC_PIN_A PC4 //pin A4
14 #define RIGHT_ENC_PIN_B PC5 //pin A5
15#endif
16
17 long readEncoder(int i);
18 void resetEncoder(int i);
19 void resetEncoders();

```

---

Rysunek 5.4: Kod źródłowy z pliku encoder\_driver.h

- **commands.h** - zawiera definicje komend wysyłanych z Raspberry Pi do Arduino. W pliku zdefiniowano między innymi komendy do sterowania silnikami, odczytu enkoderów i resetowania enkoderów.
- **diff\_controller.h** - plik zawierający implementację regulatora PID dla sterowania prędkością kół robota. Regulator wykorzystuje zaawansowane techniki sterowania, w tym:
  - Unikanie skoku pochodnej (ang. "derivative kick") poprzez wykorzystanie poprzedniego wejścia zamiast poprzedniego błędu
  - Plynne dostrajanie parametrów dzięki użyciu członu całkującego ITerm zamiast scałkowanego błędu
  - Inicjalizację zapobiegającą skokom przy uruchomieniu
  - Anti-windup poprzez ograniczenie wyjścia i zatrzymanie całkowania gdy wyjście jest nasycone

Regulator PID (ang. "Proportional-Integral-Derivative"), w którym zastosowane zostały domyślne wartości po testach i zadowalających wynikach. Działa w następujący sposób:

- Człon proporcjonalny (P) - generuje sygnał sterujący proporcjonalny do uchybu
- Człon całkujący (I) - eliminuje błąd w stanie ustalonym poprzez całkowanie uchybu
- Człon różniczkujący (D) - poprawia odpowiedź przejściową poprzez przewidywanie zmian uchybu

Plik zawiera struktury danych i funkcje do:

- Przechowywania stanu regulatora dla każdego koła
- Resetowania stanu regulatorów
- Obliczania nowych wartości sterujących
- Aktualizacji regulatorów na podstawie odczytów z enkoderów

### 5.3.2 LiDAR

Skrypt ten odpowiada za odczyt danych z LiDAR-a i przesyłanie ich do Raspberry Pi. Program ten wykorzystuje pakiet **rplidar\_ros** udostępniony przez producenta SLAMTEC. Do obsługi LiDAR-a stworzono plik wykonawczy **rplidar.launch.py**. Ten plik odpowiedzialny jest za konfigurację i uruchomienie węzła obsługującego czujnik RPLidar A1.

Plik definiuje parametry pracy czujnika, takie jak:

- Port szeregowy do komunikacji z czujnikiem
- Prędkość transmisji (115200 bodów)
- Tryb skanowania (standardowy)
- Częstotliwość publikowania danych (2,5 Hz)
- Zakres pomiarowy (od 0,15 m do 12 m wynikający z parametrów czujnika)
- Zakres kątowy skanowania (pełny obrót 360 stopni)

Program sprawdza również dostępność portu szeregowego i zgłasza odpowiedni komunikat w przypadku braku połączenia z czujnikiem. Dane z czujnika są publikowane do systemu **ROS 2** w systemie tematów (ang. "topic"), wiadomości z węzła LiDAR-a przesyłane są do tematu "**scan**" i mogą być wykorzystywane przez inne węzły na zasadzie subskrypcji, na przykład do tworzenia mapy otoczenia czy nawigacji.

### 5.3.3 Sterowanie silnikami i model robota

W celu sterowania robotem wykorzystano pakiet **DiffDrive Arduino**, który jest rozszerzeniem zestawu narzędzi **ROS 2 Control**.

Utworzono następujące pliki: **model\_controll.launch.py**, który odpowiada za uruchomienie i konfigurację podstawowych komponentów sterowania robota. Pliki konfiguracyjne **model\_main.xacro**, **model.urdf.xacro**, **internal\_macros.xacro**, **ros2\_control.xacro** definiują model kinematyczny i wizualny robota w formacie URDF/XACRO, a także konfigurację kontrolerów **ROS 2 Control**.

- **model\_controll.launch.py** - główny plik wykonawczy sterujący robotem. Odpowiada za:
  - Uruchomienie i konfigurację podstawowych komponentów sterowania:
    - \* robot\_state\_publisher - publikuje transformacje robota na podstawie modelu URDF (ang. "Unified Robot Description Format")
    - \* twist\_mux - zarządza priorytetyzacją komend prędkości, czyli przypisuje odpowiednie komendy do poprawnego korzystania z **ROS2 Control** i np. sterowania za pomocą klawiatury.
    - \* controller\_manager - zarządza kontrolerami **ROS 2 Control**
    - \* diff\_cont - kontroler napędu różnicowego
    - \* joint\_broad - nadawca stanu połączeń
  - Obsługę parametrów:
    - \* use\_sim\_time (domyślnie: fałszywe) - przełącznik trybu symulacji
    - \* use\_ros2\_control (domyślnie: prawdziwe) - włączanie/wyłączanie **ROS 2 Control**
  - Sekwencyjne uruchamianie komponentów z opóźnieniami
  - Ładowanie plików konfiguracyjnych (URDF, YAML)
- **model\_main.xacro** - plik definiujący model kinematyczny i wizualny robota w formacie URDF/XACRO. Zawiera:
  - Definicje materiałów do wizualizacji (biały - walec stanowiący korpus robota, pomarańczowy - LiDAR, niebieski - koła, czarny - koła podporowe)
  - Strukturę robota:
    - \* base\_link - podstawowy punkt odniesienia
    - \* container - główny korpus w kształcie walca
    - \* LiDAR - czujnik laserowy na górze konstrukcji
    - \* Koła napędowe (lewe i prawe) - połączenia ciągłe (ang. "continuous joints"), zależne od prędkości (ang. "velocity dependent")
    - \* Koła podporowe (przód i tył) - połączenia stałe (ang. "fixed joints")
  - Właściwości fizyczne elementów:
    - \* Masy
    - \* Momenty bezwładności
    - \* Parametry kolizji

Model ten umożliwia wizualizację robota w **RVIZ** oraz symulację jego dynamiki w środowisku **ROS 2**.

- **model.urdf.xacro** - jest to plik spajający definicję modelu z pliku model\_main.xacro z konfiguracją kontrolerów z pliku **ros2\_control.xacro**.

---

```

1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name=
3      robot_slam">
4      <xacro:arg name="use_ros2_control" default="true"/>
5      <xacro:arg name="sim_mode" default="false"/>
6      <xacro:include filename="model_main.xacro"/>
7      <xacro:include filename="ros2_control.xacro"/>
8      <xacro:ros2_control_robot use_ros2_control="$(arg_
          use_ros2_control)" sim_mode="$(arg_sim_mode)"/>
</robot>

```

---

Rysunek 5.5: Kod źródłowy pliku model.urdf.xacro

- **internal\_macros.xacro** - plik zawiera momenty bezwładności dla poszczególnych elementów robota.
- **ros2\_control.xacro** - w tym pliku zawarto konfigurację silników dla Ros2 Control, zdefiniowano w nim nazwy dla poszczególnych kół, częstotliwość odświeżania zgodną z skryptem **ROS Arduino Bridge**, czyli 57600 bodów, zliczenia enkoderów na obrót koła - 2194 wyznaczone przez testy, oraz ustawienie maksymalnej i minimalnej prędkości dla każdego silnika co w przypadku tego projektu wynosi -5 dla minimalnej i 5 dla maksymalnej.
- **twist\_mux.yaml** - plik konfiguracyjny, wykorzystywany do udostępnienia cmd\_vel do sterowania robotem. Jest to potrzebne przy sterowaniu z klawiatury.

- **my\_controllers.yaml** - jest to plik konfiguracyjny potrzebny do integracji systemu z pakietem **Diffdrive Arduino**. Zdefiniowano w nim nazwy kół zgodne z poprzednimi plikami konfiguracyjnymi, częstotliwość aktualizacji, szerokość kół i ich rozstaw.
- 

```
1     ros__parameters:  
2  
3         publish_rate: 50.0 # Increased for better velocity  
4             updates  
5  
6         base_frame_id: base_footprint  
7  
8         left_wheel_names: [ 'left_wheel_joint' ]  
9         right_wheel_names: [ 'right_wheel_joint' ]  
10        wheel_separation: 0.14 # set to 14 cm  
11        wheel_radius: 0.035    # set to 3.5 cm  
12        use_stamped_vel: false
```

---

Rysunek 5.6: Fragment kodu z pliku my\_controllers.yaml

### 5.3.4 Mapowanie

W celu mapowania otoczenia robota wykorzystano pakiet SLAM Toolbox, który pozwala na tworzenie mapy otoczenia na podstawie danych z LiDAR-a. Utworzono plik **slam.launch.py**, który odpowiada za konfigurację i uruchomienie węzła odpowiedzialnego za mapowanie otoczenia. Wykorzystano również i przystosowano plik konfiguracyjny **mapper\_params\_online\_async.yaml** z pakietu **SLAM Toolbox**, w którym zdefiniowano parametry pracy algorytmu.

- **slam.launch.py** - plik ten odpowiada za konfigurację i uruchomienie węzła odpowiedzialnego za mapowanie otoczenia. Program ten publikuje mapę otoczenia w systemie **ROS 2**, która może być wykorzystywana do nawigacji robota.
- **mapper\_params\_online\_async.yaml** - plik konfiguracyjny zawarty w pakiecie **SLAM Toolbox**, wybrana została wersja online async, zamiast online sync, ze względu na lepszą wydajność. W pliku zdefiniowano parametry pracy algorytmu, takie jak:

---

```

1 solver_plugin: solver_plugins::CeresSolver
2 ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
3 ceres_preconditioner: SCHUR_JACOBI
4 ceres_trust_strategy: LEVENBERG_MARQUARDT
5 ceres_dogleg_type: TRADITIONAL_DOGLEG
6 ceres_loss_function: HUBERLOSS # Changed from None
7 ceres_huber_loss_delta: 2.0

```

---

Rysunek 5.7: Fragment kodu z pliku mapper\_params\_online\_async.yaml

Wprowadzono tutaj zmianę w funkcji straty z `None` na **HuberLoss**, co pozwala na lepsze radzenie sobie z błędami wynikającymi z ślizgania się kół robota. Dodano również parametr **ceres\_huber\_loss\_delta**, który określa jak duże błędy są ignorowane przez algorytm.

```
1 # \textbf{ROS 2} Parameters
2 odom_frame: "odom"
3 map_frame: "map"
4 base_frame: "base_footprint"
5 scan_topic: "/scan"
6 use_map_saver: true
7 use_sim_time: false
8 mode: "mapping" #localization
```

---

Rysunek 5.8: Fragment kodu z pliku mapper\_params\_online\_async.yaml

Ustawiono parametry **ROS 2**, tak by odpowiadały tym jakie zastosowano w robocie, w tym nazwa tematu **/scan** z LiDAR-a, nazwy ramek oraz tryb mapowania.

```
1 debug_logging: false
2 throttle_scans: 2
3 transform_publish_period: 0.05 #if 0 never publishes
4 odometry
5 map_update_interval: 2.0
6 resolution: 0.05
7 max_laser_range: 12.0 #RP Lidar A1
8 minimum_time_interval: 0.2
9 transform_timeout: 0.5
10 tf_buffer_duration: 30.
11 stack_size_to_use: 60000000 // program needs a larger stack
12 size
13 enable_interactive_mode: true
14 minimum_travel_distance: 0.05 # Reduced from 0.1
```

---

Rysunek 5.9: Fragment kodu z pliku mapper\_params\_online\_async.yaml

Zmodyfikowano również parametry takie jak okres publikacji transformacji dla lepszej synchronizacji, maksymalny zakres pomiarowy zainstalowanego LiDAR-a, rozmiar stosu (ze względu na dużą ilość danych przetwarzanych przez algorytm). Zmodyfikowano również ogólne parametry takie jak minimalny dystans, tak by częściej aktualizować mapę.

Ważnym aspektem SLAM Toolbox jest to że wykorzystuje bibliotekę **Ceres Solver** [1]. Jest to narzędzie, które pomaga znaleźć najlepsze dopasowanie między różnymi pomiarami, minimalizując błąd średniokwadratowy. Dzięki temu SLAM Toolbox może tworzyć dokładne mapy, nawet gdy niektóre pomiary są niedokładne lub błędne.

Działa to na zasadzie iteracyjnego poprawiania oszacowań pozycji robota:

$$\min_x \frac{1}{2} \sum_i \rho_i(\|f_i(x_i)\|^2) \quad (5.1)$$

- $x$  to pozycje robota, które chcemy znaleźć
- $f_i(x_i)$  mierzy, jak bardzo nasze oszacowanie pozycji różni się od rzeczywistych pomiarów
- $\rho_i$  to funkcja, która pomaga ignorować błędne pomiary (działa jako filtr odrzucający "podejrzane" dane)
- Cały proces dąży do znalezienia takich pozycji robota, dla których suma wszystkich błędów jest najmniejsza

### 5.3.5 Nawigacja i lokalizacja

Do nawigacji i lokalizacji robota wykorzystano pakiet **Nav2**, który pozwala na wyznaczanie trasy i lokalizację robota na mapie.

W tym celu utworzono plik **nav\_localization.launch.py**, oraz dwa pliki konfiguracyjne **amcl.yaml** - plik zawierający wyłącznie konfigurację **AMCL** i **nav2\_params.yaml** - plik bazujący na pliku konfiguracyjnym z **Nav2**.

- **nav\_localization.launch.py** - program ten tworzy środowisko nawigacyjne poprzez uruchomienie zestawu węzłów **ROS 2**, które realizują poszczególne funkcje nawigacyjne.

Kluczowe komponenty systemu to serwer mapy (ang. "**map\_server**"), który zarządza zapisaną wcześniej mapą otoczenia, oraz węzeł **AMCL** realizujący lokalizację robota. Za ruch robota odpowiada zestaw węzłów nawigacyjnych: kontroler ruchu (ang. "**controller\_server**") wykonujący zaplanowane trajektorie, projektant ścieżki (ang. "**planner\_server**") wyznaczający optymalną trasę, moduł zachowań awaryjnych (ang. "**behavior\_server**") reagujący w sytuacjach nietypowych, np. napotkanie nowej przeszkody, oraz nawigator oparty na drzewach zachowań (ang. "**bt\_navigator**") koordynujący całość.

Dodatkowo uruchamiany jest węzeł RVIZ zapewniający wizualizację mapy, robota i trasy.

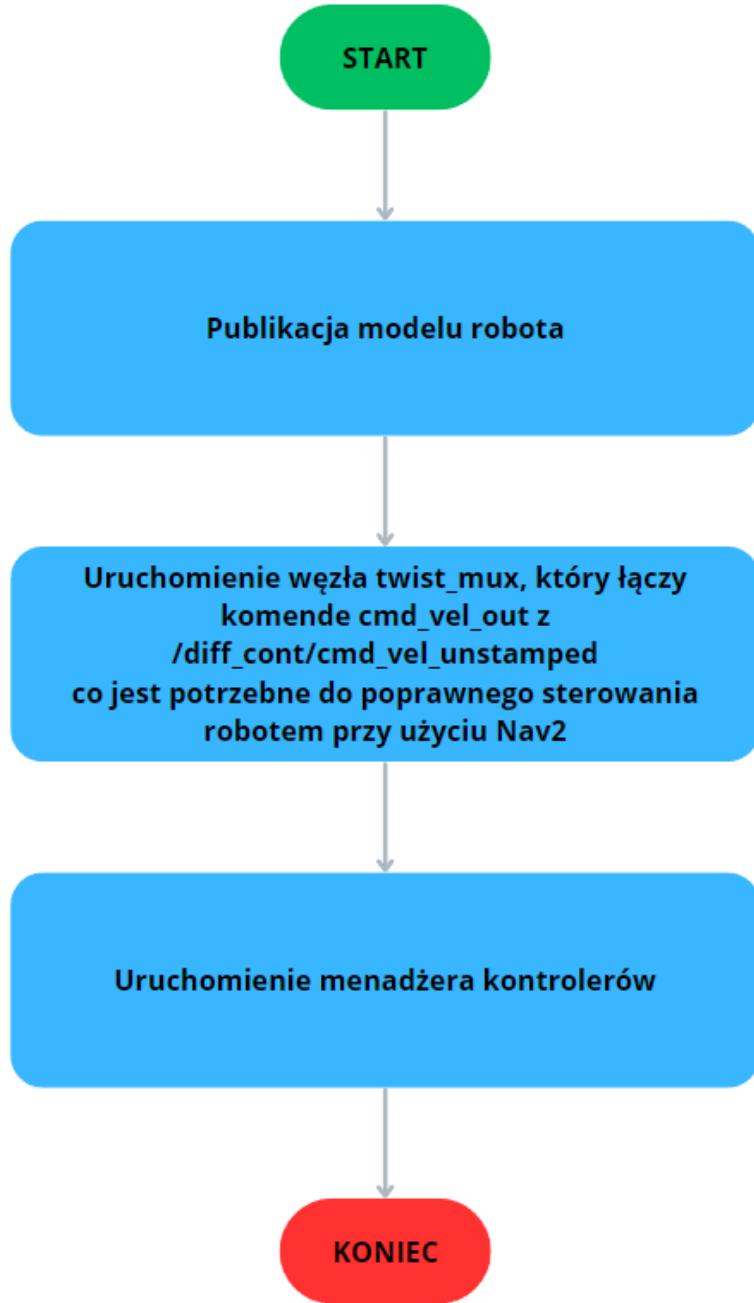
Wszystkie węzły są skonfigurowane do pracy z przekształceniами układów współrzędnych (ang. "**transforms**") poprzez odpowiednie mapowanie tematów **/tf** i **/tf\_static**. Parametry węzłów są dynamicznie konfigurowane z użyciem plików YAML, które mogą być modyfikowane w zależności od potrzeb konkretnej aplikacji.

Do nawigacji tworzone są dodatkowe mapy kosztów (ang. "**costmap**") lokalna i globalna. Globalna mapa odpowiada za wyznaczanie trasy w oparciu o wcześniejsze zapisane na mapie przeszkody, natomiast lokalna mapa kosztów jest używana do unikania kolizji z przeszkodami wykrywanymi w czasie rzeczywistym które nie były wcześniejsze zapisane na mapie.

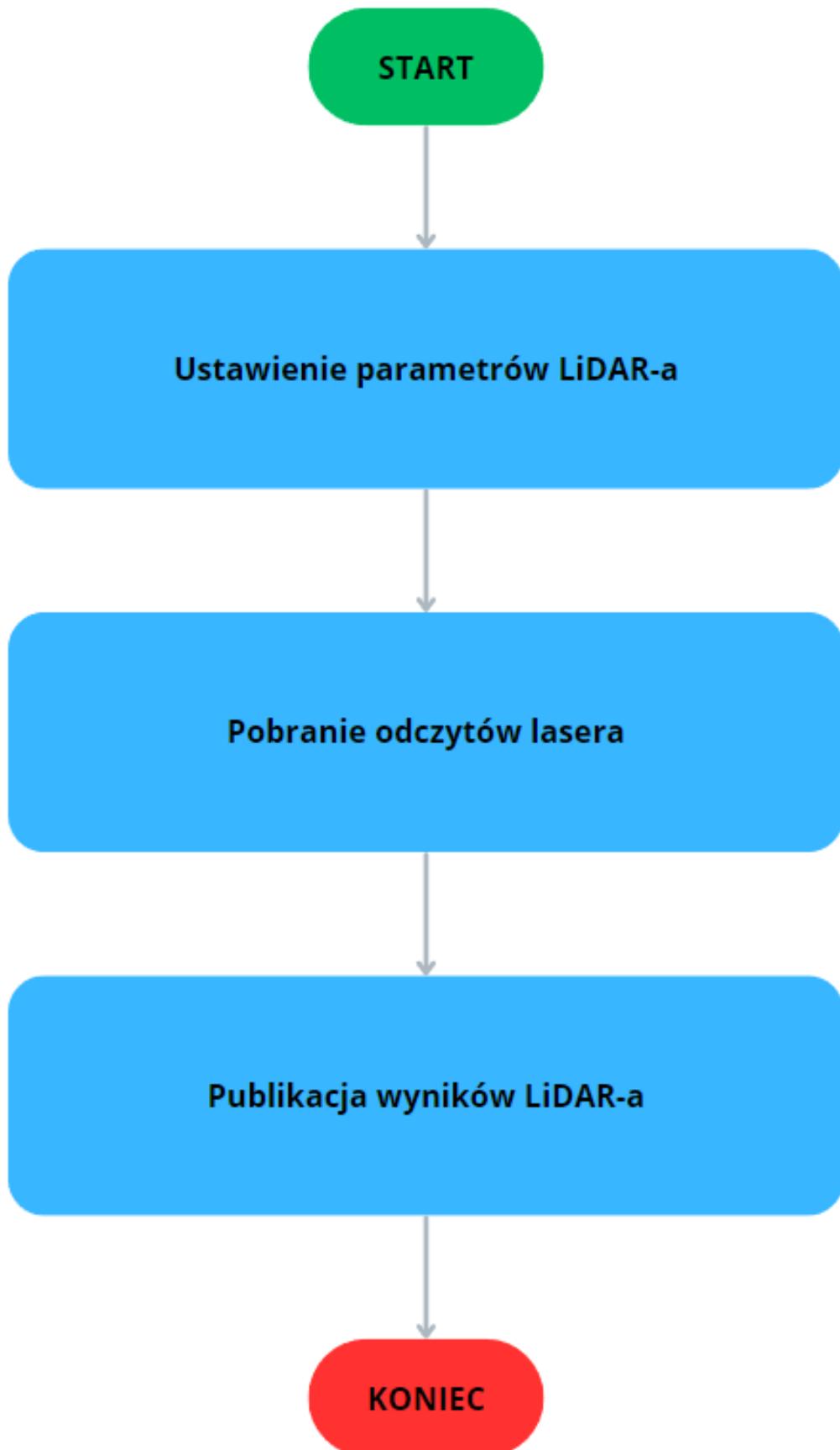
- **amcl.yaml** - plik konfiguracyjny algorytmu **AMCL** odpowiedzialnego za precyzyjną lokalizację robota na mapie. Wykorzystano tutaj podstawowe parametry **AMCL** które wcześniej znajdowały się w pliku nav2\_params.yaml i zmieniono je na potrzeby systemu. Zmieniono parametry takie jak:  
**do\_beamskip** na prawdziwe co poprawiło prędkość działania programu,  
**laser\_max\_range** i **laser\_min\_range** zostały dostosowane do zainstalowanego LiDAR-a,  
**max\_particles** i **min\_particles** zostały zmienione kolejno na 10000 i 3000 z 2000 i 500 dla lepszej jakości lokalizacji przez zwiększenie ilości przewidywanych punktów,  
**recovery\_alpha\_fast** i **recovery\_alpha\_slow** zostały kolejno zmienione z 0 na 0.1 i 0 na 0.001, co pozwala na lepszą naprawę lokalizacji przez dodawanie losowych punktów,  
**update\_min\_a** i **update\_min\_d** zostały zmienione z 0.2 i 0.25 na 0.1 i 0.1 co pozwala na częstsze aktualizacje filtra przy mniejszym przejechanym dystansie,  
**scan\_topic** ustawiono na **/scan** zgodny z tym jaki wysyłany jest z rplidar.launch.py. Dodano również **use\_sim\_time: false** ze względu na to że program jest na rzeczywistym modelu a nie w symulacji.
- **nav2\_params.yaml** - jest to główny plik konfiguracyjny **Nav2**, w którym zdefiniowano parametry pracy systemu nawigacyjnego. Główną zmianą było wyciągnięcie parametrów **AMCL** z tego pliku. Dostosowano również parametry takie jak obszary w mapach kosztów - globalnej **cost\_scaling\_factor** i **inflation\_radius** z 3 na 2 i 0.7 na 0.3 oraz lokalnej **cost\_scaling\_factor** i **inflation\_radius** z 3 na 2 i 0.7 na 0.3. Zmiana ta została zastosowana aby łatwiej nawigować w małych pomieszczeniach przez zmniejszenie obszarów niebezpieczeństwa w tych mapach.

## 5.4 Diagramy UML prezentujące działanie konkretnych programów

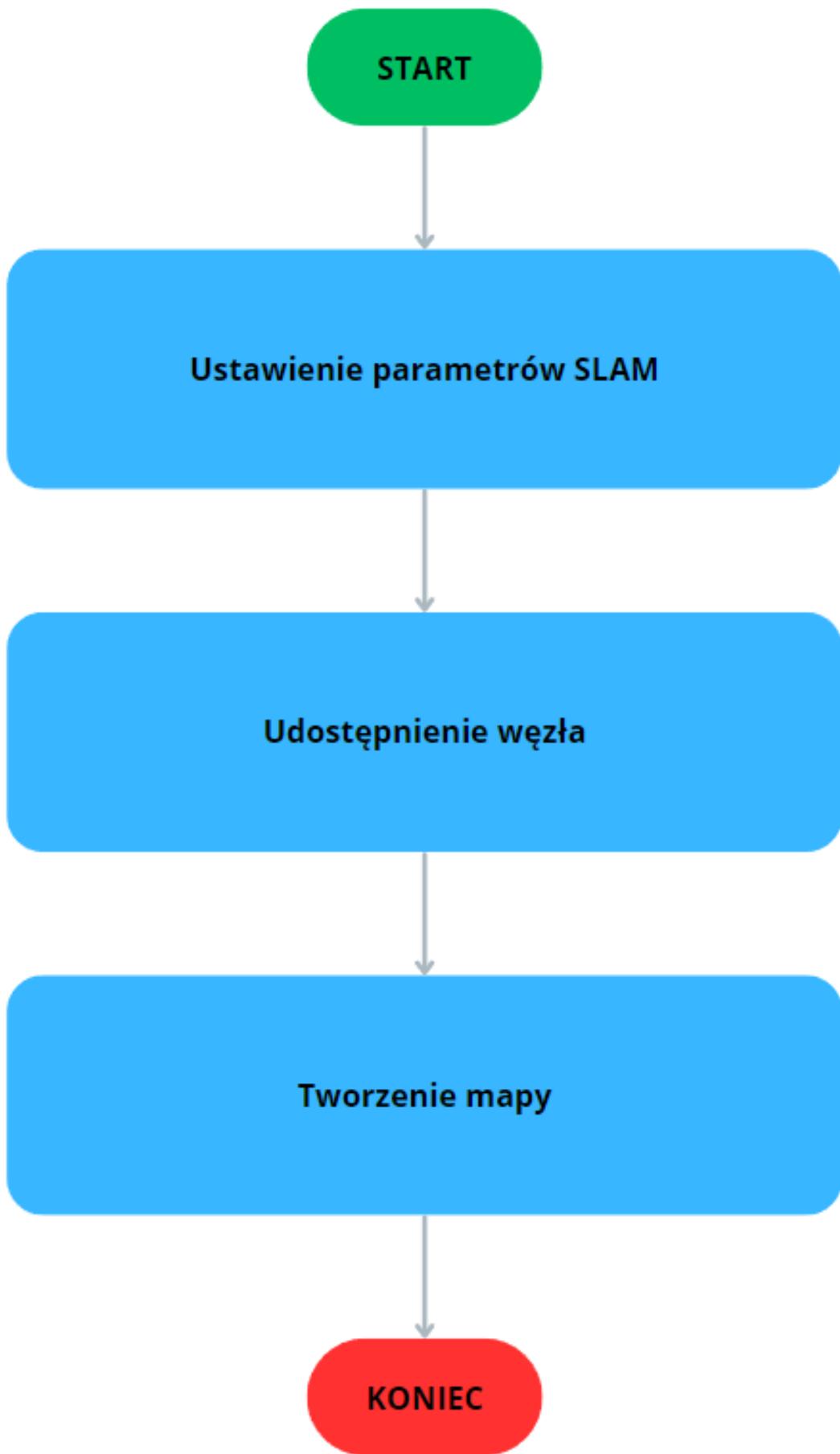
Sekcja ta zawiera diagramy UML prezentujące działanie poszczególnych programów, które składają się na system sterowania robotem.



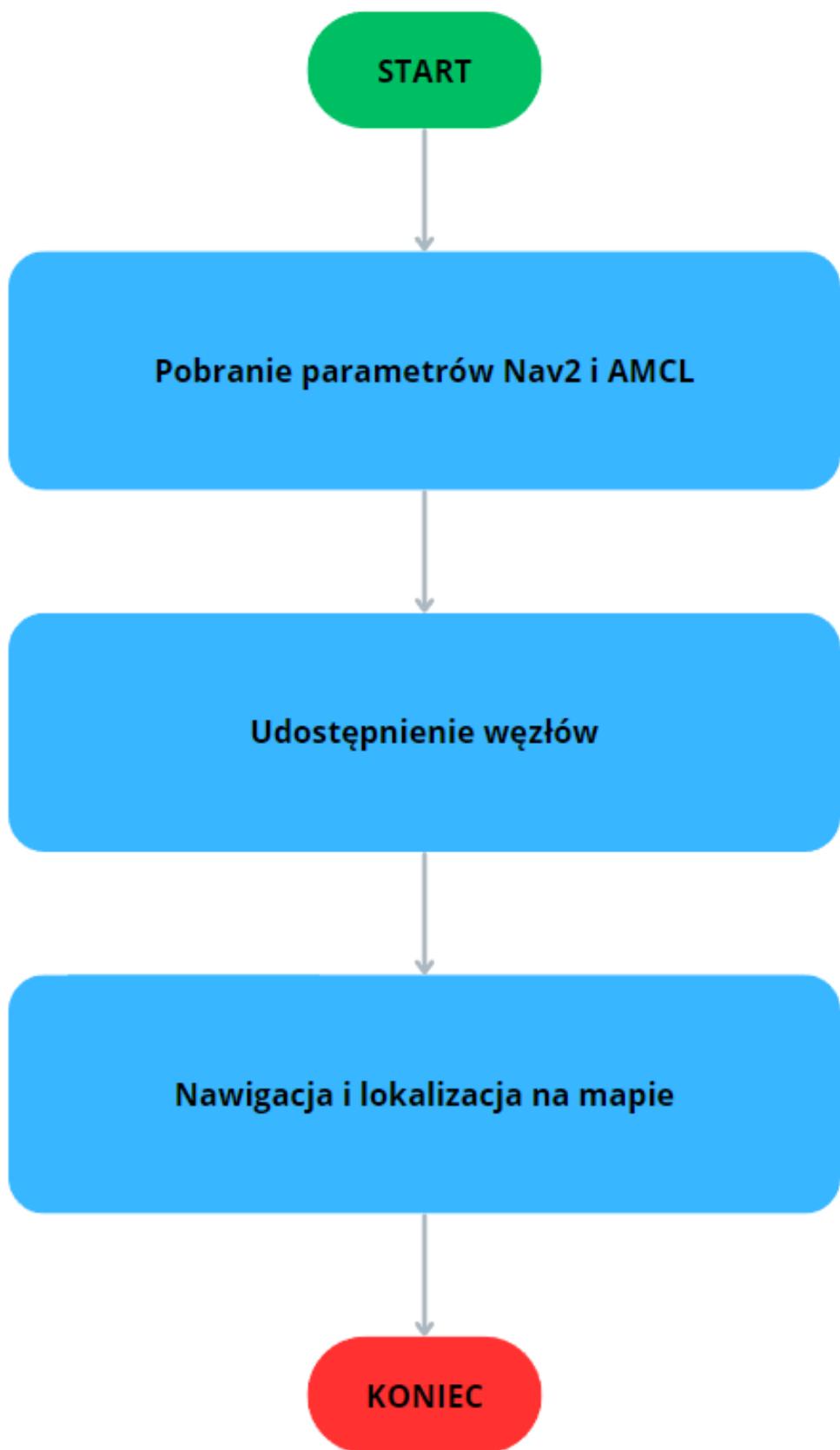
Rysunek 5.10: Diagram UML prezentujący działanie programu model\_controll.launch.py



Rysunek 5.11: Diagram UML prezentujący działanie programu `rplidar.launch.py`



Rysunek 5.12: Diagram UML prezentujący działanie programu `slam.launch.py`



Rysunek 5.13: Diagram UML prezentujący działanie programu `nav_localization.launch.py`

# Rozdział 6

## Weryfikacja i walidacja

W tym rozdziale przedstawiono proces weryfikacji i walidacji systemu. Wykonano to z wykorzystaniem modelu V, który pozwolił na przetestowanie poszczególnych komponentów systemu, integrację modułów oraz weryfikację zgodności z wymaganiami projektu. Opisano tu również organizację eksperymentów, przypadki testowe oraz wykryte i usunięte błędy.

### 6.1 Model V

W ramach pracy zastosowano model V do testowania systemu. Jest on popularnym modelem w inżynierii oprogramowania, który przedstawia proces tworzenia oprogramowania w formie litery V. Jej lewa strona reprezentuje fazy definiowania wymagań i projektowania systemu, natomiast prawa strona reprezentuje fazy testowania i walidacji systemu.

Na początku zdefiniowano wymagania projektu, które obejmowały automatyczne tworzenie mapy otoczenia, autonomiczną nawigację robota, precyzyjną lokalizację oraz wykrywanie i omijanie przeszkód. Na tej podstawie przeprowadzono szczegółową analizę wymagań funkcjonalnych, określając potrzebę zdalnego sterowania robotem, możliwość zapisywania i odczytu map, autonomicznej nawigacji do zadanych punktów oraz bezpiecznego poruszania się.

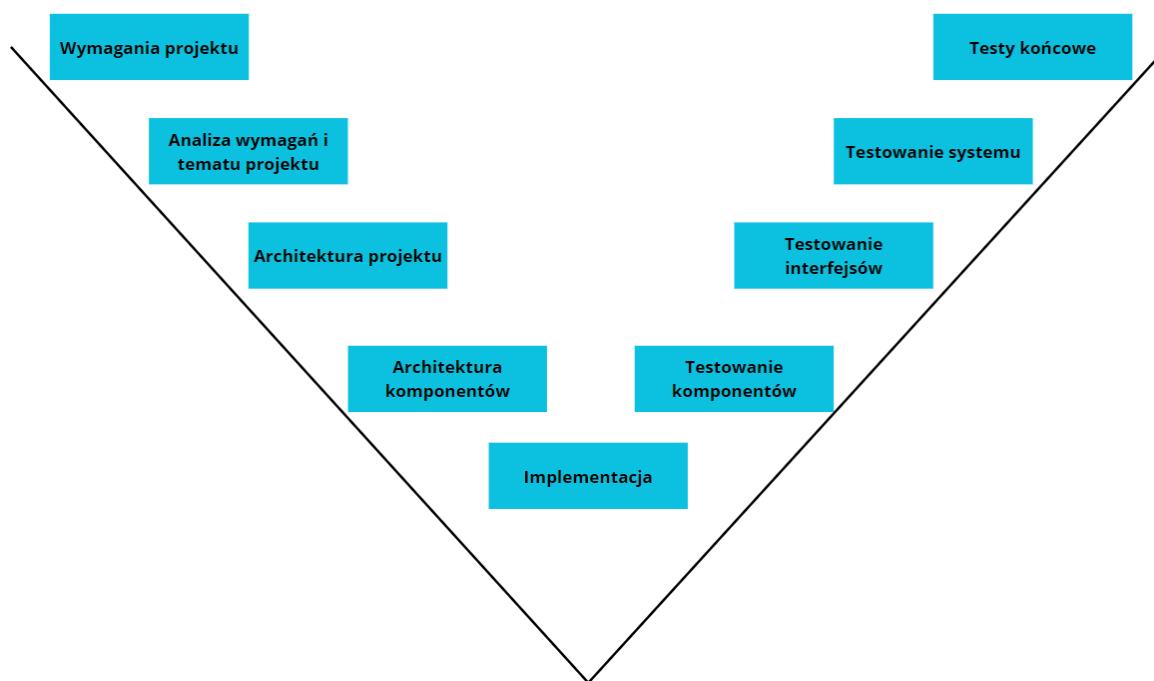
Następnie zaprojektowano architekturę systemu, dzieląc go na podsystemy odpowiedzialne za napęd, sensory i sterowanie. Określono przepływ danych między modułami oraz interfejsy komunikacyjne. Na tym etapie wybrano również kluczowe technologie, takie jak **ROS 2**, **Nav2** i **SLAM Toolbox**. W ramach projektowania szczegółowego dobrano komponenty sprzętowe, opracowano projekt mechaniczny i elektryczny oraz zaplanowano strukturę oprogramowania wraz z interfejsami programowymi.

Implementacja systemu objęła fizyczną budowę robota, programowanie sterowników, integrację sensorów oraz implementację algorytmów.

Następnie rozpoczęto procesy testowania, zaczynając od testów jednostkowych poszczególnych komponentów. Weryfikowano działanie napędów, sprawdzano sensory, testowano sterowniki oraz algorytmy.

Kolejnym etapem było testowanie interfejsów, gdzie skupiono się na komunikacji między modułami, synchronizacji danych i ogólnym przepływie informacji w systemie. Następnie przeprowadzono testy całego systemu, weryfikując funkcjonalności mapowania otoczenia, lokalizacji robota, planowania trasy i omijania przeszkód.

Końcowym etapem były testy walidacyjne, podczas których sprawdzono zgodność systemu z pierwotnymi wymaganiami. Przeprowadzono testy wydajności i niezawodności, potwierdzając że zrealizowany system spełnia wszystkie założone cele.



Rysunek 6.1: Model V

Na rysunku 6.1 przedstawiono model V, który pokazuje proces tworzenia systemu sterowania robotem mobilnym. Na lewej stronie znajdują się fazy definiowania wymagań i projektowania systemu, natomiast na prawej stronie znajdują się fazy testowania i walidacji systemu.

## 6.2 Organizacja eksperymentów

W ramach organizacji eksperymentów przeprowadzono testowanie jednostkowe, gdzie każdy moduł systemu, taki jak sterowanie silnikami, odczyt danych z LiDAR-a, tworzenie mapy, lokalizacja i nawigacja, został przetestowany indywidualnie. Następnie przeprowadzono testy integracyjne, gdzie sprawdzano współdziałanie poszczególnych modułów. Obejmowało to testy sterowania manualnego podczas mapowania, lokalizacji robota na zapisanej mapie oraz automatycznej nawigacji do wyznaczonych punktów. Na końcu przeprowadzono weryfikację, gdzie system został zweryfikowany pod kątem spełnienia wymagań projektu.

## 6.3 Przypadki testowe

Przypadki testowe obejmowały różne scenariusze. Pierwszym było testowanie zdalnego sterowania, gdzie sprawdzono czy robot reaguje poprawnie na polecenia z klawiatury. Drugim było testowanie tworzenia mapy, gdzie sprawdzono czy system poprawnie tworzy mapę otoczenia na podstawie danych z LiDAR-a. Kolejnym było testowanie lokalizacji, gdzie sprawdzono czy system poprawnie lokalizuje robota na zapisanej mapie. Ostatnim było testowanie nawigacji, gdzie sprawdzono czy system poprawnie planuje trasę i nawigację robota do wyznaczonych punktów, omijając przeszkody.

## 6.4 Wykryte i usunięte błędy

Podczas testowania systemu wykryto i usunięto kilka istotnych błędów. Pierwszym był **problem z zasilaniem Raspberry Pi**, co zostało rozwiązane przez wyłączenie zbędnej funkcji szukania monitora. Rozwiązano ten problem przez zastosowanie komendy sudo systemctl set-default multi-user.target.

Drugim **problemem był związany z kołami** - przez rozmiar kół i materiał z jakich zostały wykonane, robot miał problem z ślizganiem się kół co powodowało różnice między rzeczywistą pozycją robota, a tą która była wyznaczana podczas mapowania. Rozwiązano ten problem poprzez zastosowanie funkcji **HuberLoss** w algorytmie mapowania.

Na wczesnym etapie testów pojawił się również **problem z błędnią lokalizacją robota na mapie**, który również uniemożliwiał nawigację przy użyciu **Nav2**. Powodował sytuacje gdy podczas mapowania robot oddalał się od ściany w **RVIZ** zbliżała się do niej i po chwili model przenosił się na poprawne miejsce, powodowało to zmapowanie dwóch ścian - poprawnej i przed przeniesieniem. Rozwiążaniem problemu było poprawne podłączenie silników do sterownika i Arduino.



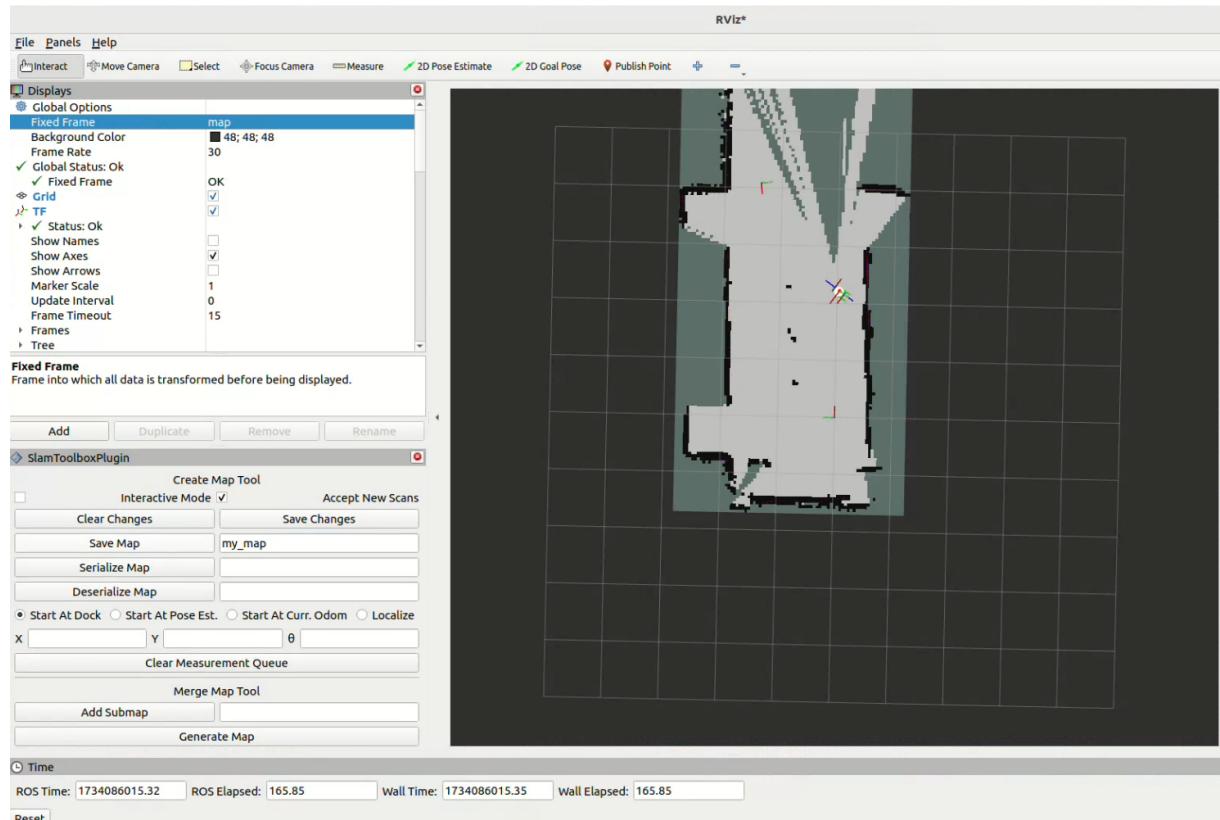
# Rozdział 7

## Podsumowanie i wnioski

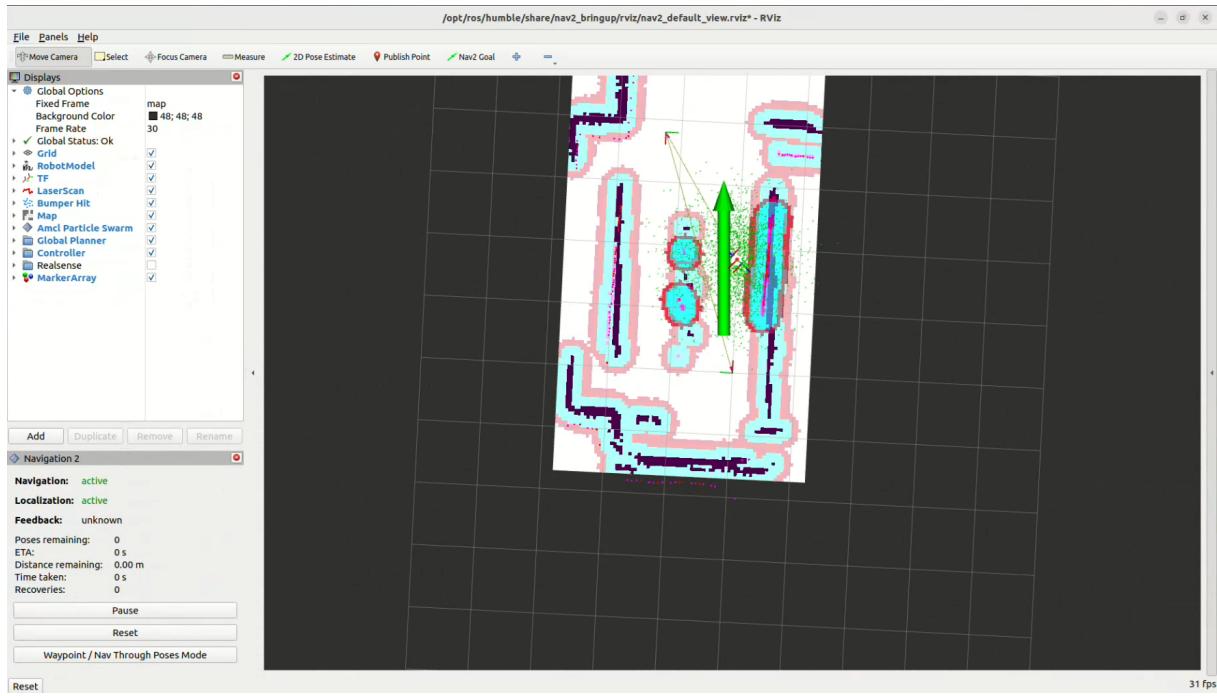
Ten rozdział zawiera uzyskane wyniki z wnioskami oraz kierunki dalszych prac.

### 7.1 Uzyskane wyniki i wnioski

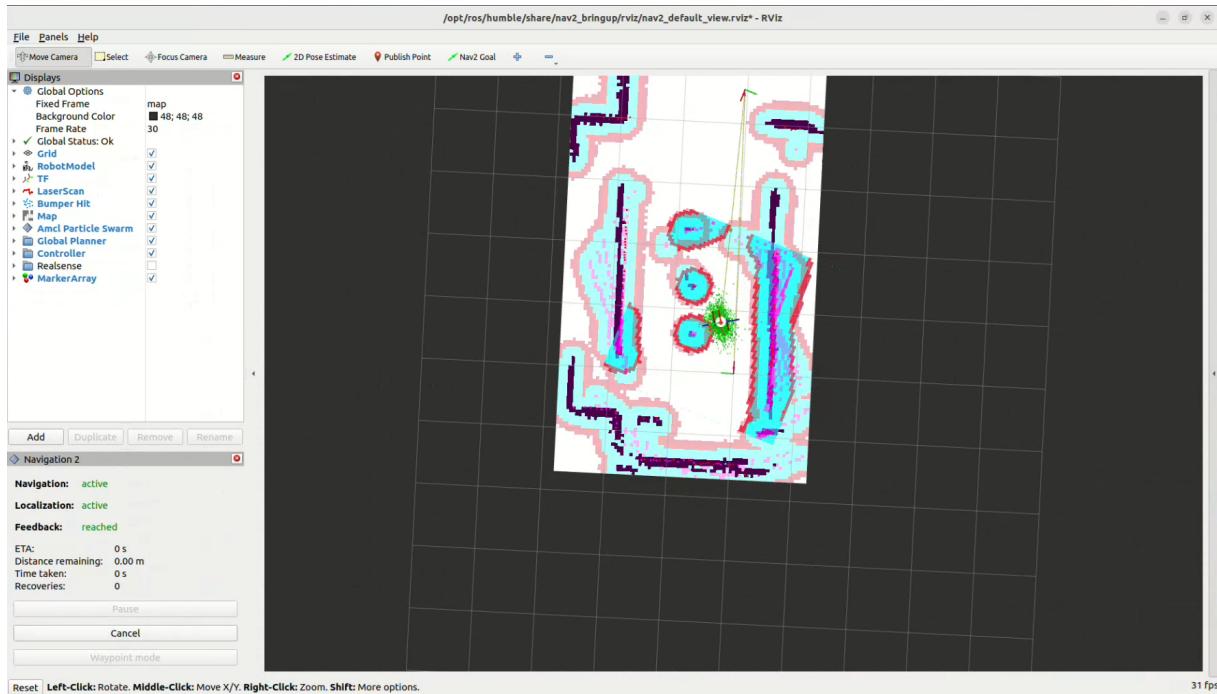
W wyniku przeprowadzonych prac udało się zrealizować wszystkie założone cele projektu. System autonomicznej nawigacji robota mobilnego został zaprojektowany, zaimplementowany i przetestowany. Robot poprawnie tworzy mapę otoczenia, lokalizuje się na zapisanej mapie oraz nawigował do wyznaczonych punktów, omijając przeszkody.



Rysunek 7.1: Wizualizacja mapy otoczenia.



Rysunek 7.2: Rozpoczęcie nawigacji.



Rysunek 7.3: Dotarcie do celu

## 7.2 Kierunki dalszych prac

W przyszłości możliwe jest rozszerzenie funkcjonalności systemu o dodatkowe elementy, takie jak:

- Integracja z dodatkowymi sensorami, takimi jak kamery RGB-D, aby poprawić dokładność mapowania i lokalizacji.
- Rozbudowa systemu o możliwość ładowania akumulatorów bez potrzeby wyciągania ich z robota i zamontowanie modułu BMS.
- Stworzenie obudowy dla robota, aby zabezpieczyć go przed uszkodzeniami mechanicznymi.



# Bibliografia

- [1] Sameer Agarwal, Keir Mierle i The Ceres Solver Team. *Ceres Solver*. Wer. 2.2. Paź. 2023. URL: <https://github.com/ceres-solver/ceres-solver>.
- [2] Luis Bermudez. *Medium - Overview of SLAM*. 2024. URL: <https://medium.com/machinevision/overview-of-slam-50b7f49903b7> (term. wiz. 17.04.2024).
- [3] Weifeng Chen, Chengjun Zhou, Guangtao Shang, Xiyang Wang, Zhenxiong Li, Chonghui Xu i Kai Hu. „SLAM Overview: From Single Sensor to Heterogeneous Fusion”. W: *Remote Sensing* 14.23 (2022). ISSN: 2072-4292. DOI: 10.3390/rs14236033. URL: <https://www.mdpi.com/2072-4292/14/23/6033>.
- [4] Bence Magyar Christoph Fröhlich Denis Stogl i Sai Kishor Kothakota. *ros2 control*. 2024. URL: <https://control.ros.org/humble/index.html> (term. wiz. 01.12.2024).
- [5] Dieter Fox. „KLD-Sampling: Adaptive Particle Filters.” W: sty. 2001, s. 713–720.
- [6] Patrick Goebel. *ROS.org - ros arduino bridge*. 2012. URL: [https://wiki.ros.org/ros\\_arduino\\_bridge](https://wiki.ros.org/ros_arduino_bridge) (term. wiz. 25.12.2012).
- [7] Graylin Trevor Jay. *Teleop Twist Keyboard*. 2015. URL: [https://wiki.ros.org/teleop\\_twist\\_keyboard](https://wiki.ros.org/teleop_twist_keyboard) (term. wiz. 22.01.2015).
- [8] Luna Kang. „Real Archaeology”. W: () .
- [9] Matti Kortelainen. „A short guide to ROS 2 Humble Hawksbill”. W: *School of Computing, University of Eastern Finland, Kuopio, Finland* (2023), s. 5.
- [10] Steve Macenski i Ivona Jambrecic. „SLAM Toolbox: SLAM for the dynamic world”. W: *Journal of Open Source Software* 6.61 (2021), s. 2783. DOI: 10.21105/joss.02783. URL: <https://doi.org/10.21105/joss.02783>.
- [11] Steve Macenski, Francisco Martín, Ruffin White i Jonatan Ginés Clavero. „The Marathon 2: A Navigation System”. W: *CoRR* abs/2003.00368 (2020). arXiv: 2003.00368. URL: <https://arxiv.org/abs/2003.00368>.
- [12] Josh Newans. *ROS.org - diffdrive arduino*. 2020. URL: [https://wiki.ros.org/diffdrive\\_arduino](https://wiki.ros.org/diffdrive_arduino) (term. wiz. 08.12.2020).

- [13] Francisco Martin Rico. *A Concise Introduction to Robot Programming with ROS2*. Broken Sound Parkway NW: Taylor i Francis, 2022. ISBN: 1032264659.
- [14] Qin Zou, Qin Sun, Long Chen, Bu Nie i Qingquan Li. „A Comparative Analysis of LiDAR SLAM-Based Indoor Navigation for Autonomous Vehicles”. W: *IEEE Transactions on Intelligent Transportation Systems* 23.7 (2022), s. 6907–6921. DOI: 10.1109/TITS.2021.3063477.

## **Dodatki**



# Spis skrótów i symboli

SLAM jednoczesna lokalizacja i mapowanie (ang. *Simultaneous Localization and Mapping*)

LiDAR urządzenie wykonujące wykrywanie światła i określanie odległości (ang. *Light Detection and Ranging*)

IMU inercyjna jednostka pomiarowa (ang. *Inertial Measurement Unit*)

RGB-D kamera rejestrująca obraz RGB oraz informację o głębi (ang. *RGB-Depth*)

Nav2 system nawigacji dla **ROS 2** (ang. *Navigation 2*)

ROS system operacyjny dla robotów (ang. *Robot Operating System*)

ROS 2 Control system kontroli robotów dla **ROS 2** (ang. *Robot Operating System 2 Control*)

SLAM Toolbox zestaw narzędzi do jednoczesnej lokalizacji i mapowania (ang. *Simultaneous Localization and Mapping Toolbox*)



# **Lista dodatkowych plików, uzupełniających tekst pracy**

Do pracy załączono następujące dodatkowe pliki:

- Pliki źródłowe programu w folderze **Oprogramowanie**
- Film pokazujący działanie robota w pliku **Robot.mp4**



# Spis rysunków

2.1	Reprezentacja pośrednika w systemie robota [13] . . . . .	7
2.2	Mapa dwóch pomieszczeń utworzona za pomocą SLAM Toolbox [10] . . . . .	8
3.1	Diagram przypadków użycia systemu . . . . .	12
3.2	Zdjęcie przedstawiające zbudowanego robota . . . . .	14
3.3	Wizualizacja robota w RVIZ . . . . .	15
3.4	Wizualizacja robota w RVIZ z osiami . . . . .	15
3.5	Uproszczony schemat przedstawiający budowę robota . . . . .	16
3.6	Schemat połączenia silników z Arduino i L298N . . . . .	17
3.7	Schemat elektryczny . . . . .	18
4.1	Zdjęcie środowiska testowego. . . . .	22
4.2	Schemat przedstawiający na jakim urządzeniu należy uruchomić poszczególne skrypty. . . . .	23
4.3	Okno RVIZ po uruchomieniu skryptów. . . . .	25
4.4	Zmapowane pomieszczenie i zapis mapy jako plik my_map. . . . .	26
4.5	RVIZ po uruchomieniu skryptu do lokalizacji i nawigacji i wybraniu pozycji robota. . . . .	27
4.6	Wizualizacja rozproszonych punktów lokalizacji robota. . . . .	28
4.7	Wybór celu robota. . . . .	29
4.8	Wyznaczenie trasy. . . . .	30
4.9	Nawigacja robota i zmniejszanie się chmury przewidywanej lokalizacji robota. . . . .	31
4.10	Dotarcie robota do celu. . . . .	31
4.11	Terminal z uruchomionym skryptem teleop_twist_keyboard. . . . .	32
4.12	Fragment kodu z pliku nav2_params.yaml . . . . .	33
5.1	Wizualizacja połączenia skryptów w systemie . . . . .	36
5.2	Fragment kodu źródłowego z pliku encoder_driver.ino z obsługą przerwań do precyzyjnego zliczania impulsów z enkoderów . . . . .	38
5.3	Kod źródłowy z pliku motor_driver.h . . . . .	39
5.4	Kod źródłowy z pliku encoder_driver.h . . . . .	39
5.5	Kod źródłowy pliku model.urdf.xacro . . . . .	43

5.6	Fragment kodu z pliku my_controllers.yaml . . . . .	44
5.7	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	45
5.8	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	46
5.9	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	46
5.10	Diagram UML prezentujący działanie programu model_controll.launch.py	49
5.11	Diagram UML prezentujący działanie programu rplidar.launch.py . . . . .	50
5.12	Diagram UML prezentujący działanie programu slam.launch.py . . . . .	51
5.13	Diagram UML prezentujący działanie programu nav_localization.launch.py	52
6.1	Model V . . . . .	54
7.1	Wizualizacja mapy otoczenia. . . . .	57
7.2	Rozpoczęcie nawigacji. . . . .	58
7.3	Dotarcie do celu . . . . .	58