

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 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"), SLAM Toolbox (ang. "Simultaneous Localization and Mapping Toolbox") oraz ROS2 Control (ang. "Robot Operating System 2 Control").

## **Słowa kluczowe**

mapowanie, robot mobilny, lokalizacja, SLAM, ROS

## **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), SLAM Toolbox (Simultaneous Localization and Mapping Toolbox) and ROS2 Control (Robot Operating System 2 Control).

## **Key words**

mapping, mobile robot, localization, SLAM, ROS



# 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 . . . . .	5
2.2 Szczegółowe sformułowanie problemu . . . . .	5
2.3 Przegląd narzędzi i literatury . . . . .	6
2.3.1 ROS 2 . . . . .	6
2.3.2 SLAM Toolbox . . . . .	7
2.3.3 Navigation2 (Nav2) i lokalizacja . . . . .	9
2.3.4 ROS2 Control i sterowanie napędami . . . . .	9
2.4 Wybór rozwiązań . . . . .	9
<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 . . . . .	18
3.5.1 Etap 1: Przygotowanie platformy sprzętowej . . . . .	18
3.5.2 Etap 2: Implementacja sterowania napędem . . . . .	18
3.5.3 Etap 3: Integracja sensorów . . . . .	18
3.5.4 Etap 4: Implementacja oprogramowania . . . . .	18

<b>4 Specyfikacja użytkowa</b>	<b>19</b>
4.0.1 Wymagania sprzętowe i programowe . . . . .	19
4.0.2 Sposób aktywacji i korzystania z robota z przykładem działania . . . . .	20
4.1 Administracja systemem . . . . .	27
4.2 Kwestie bezpieczeństwa . . . . .	27
4.3 Scenariusze korzystania z systemu . . . . .	28
<b>5 Specyfikacja techniczna</b>	<b>29</b>
5.1 Wykorzystane technologie . . . . .	29
5.2 Architektura systemu . . . . .	30
5.3 Struktura systemu i objaśnienie działania algorytmów . . . . .	31
5.3.1 Arduino . . . . .	31
5.3.2 LiDAR . . . . .	35
5.3.3 Sterowanie silnikami i model robota . . . . .	35
5.3.4 Mapowanie . . . . .	39
5.3.5 Nawigacja i lokalizacja . . . . .	42
5.4 Diagramy UML prezentujące działanie konkretnych programów . . . . .	44
<b>6 Weryfikacja i walidacja</b>	<b>49</b>
6.1 Model V . . . . .	50
6.2 Organizacja eksperymentów . . . . .	52
6.3 Przypadki testowe . . . . .	52
6.4 Wykryte i usunięte błędy . . . . .	53
<b>7 Podsumowanie i wnioski</b>	<b>55</b>
7.1 Uzyskane wyniki . . . . .	55
7.2 Kierunki dalszych prac . . . . .	57
<b>Bibliografia</b>	<b>59</b>
<b>Spis skrótów i symboli</b>	<b>63</b>
<b>Lista dodatkowych plików, uzupełniających tekst pracy</b>	<b>65</b>
<b>Spis rysункów</b>	<b>68</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 ROS2 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, osadzenie go w kontekście aktualnego stanu wiedzy, analizę literatury, stan aktualny dziedziny oraz uzasadnienie wyboru rozwiązania.
- Rozdział trzeci zawiera wymagania i narzędzia, w którym opisano wymagania funkcjonalne, przypadki użycia, specyfikację komponentów, sposób połączenia sterowania i metodykę wraz z etapami realizacji projektu.
- Rozdział czwarty zawiera specyfikację użytkową, w którym przedstawiono wymagania użytkownika oraz specyfikację funkcjonalną.
- Rozdział piąty zawiera specyfikację techniczną, w którym przedstawiono podstawowe pojęcia i definicje z dziedziny robotyki mobilnej, jak i implementację zastosowanego rozwiązania.
- Rozdział szósty zawiera weryfikacje i walidacje, w którym przedstawiono testy i wyniki działania systemu.
- Rozdział siódmy zawiera podsumowanie, w którym przedstawiono wnioski z pracy oraz możliwości dalszego rozwoju projektu.

## 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

Problem jednoczesnej lokalizacji, mapowania oraz autonomicznej nawigacji robotów mobilnych stanowi jeden z kluczowych obszarów badań w dziedzinie robotyki. W ostatnich latach obserwuje się znaczący postęp w tej dziedzinie, głównie dzięki rozwojowi wydajnych algorytmów optymalizacji, poprawie jakości i dostępności czujników jak LiDAR, wzrostowi mocy obliczeniowej komputerów oraz powstaniu zaawansowanych platform programistycznych jak ROS 2.

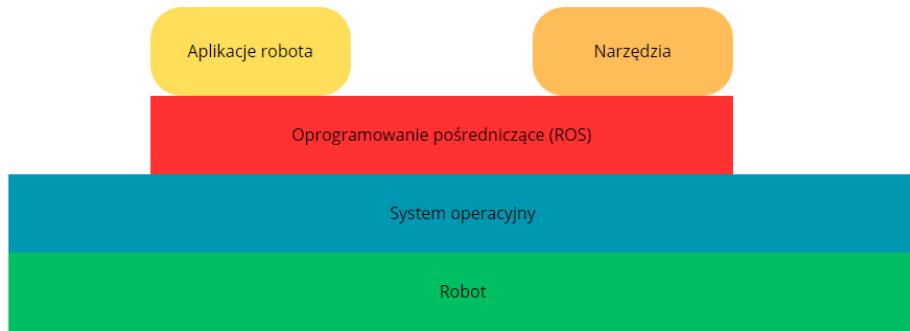
### 2.2 Szczegółowe sformułowanie problemu

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

ROS2 (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 [11]. Na poniżej ilustracji przedstawiono położenie takiego pośrednika.



Rysunek 2.1: Reprezentacja pośrednika w systemie robota [11]

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 wspólnego wykonania kodu na wiele dostępnych plików wykonywalnych, oraz komunikację między tymi modułami podczas równoległego uruchomienia w systemie robota.[7]

### 2.3.2 SLAM Toolbox

SLAM Toolbox to zestaw narzędzi i rozwiązań do tworzenia map 2D w czasie rzeczywistym, stworzone przez Steve'a Mecenski. Stosowane algorytmy w przeszłości to np. GMapping, Karto, Cartographer oraz Hector, jednakże prawie żaden z nich nie potrafił tworzyć map w czasie rzeczywistym, jedynie Cartographer stworzony przez Google miał takie możliwości, lecz przestał być on wspierany. [8] 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 [8]

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. [9]

Do lokalizacji robota na mapie można wykorzystać wcześniej opisany SLAM Toolbox, jednak w pakuie Nav2 dostępny jest również pakiet AMCL (ang. "Adaptive Monte Carlo Localization") [4], 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. [4]

### 2.3.4 ROS2 Control i sterowanie napędami

ROS2 Control to platforma do sterowania, zarządzania i komunikacji pomiędzy urządzeniami w robotach z oprogramowaniem. [3]. Rozwiązywanie to umożliwia w łatwy sposób zarządzanie silnikami, enkoderami, czy innymi urządzeniami w robocie. Dzięki zastosowaniu ROS2 Control można wykorzystać np. pakiet diffdrive\_arduino, do sterowania robotem z napędem różnicowym za pomocą Arduino. Pakiet ten pozwala również na sterowanie prędkością silników, odczyt enkoderów, obliczanie odometrii oraz transformację między układem odometrii a układem bazowym. [10]

## 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 ROS2 Control z DiffDrive Arduino do sterowania napędami. Wybór ten podyktowany 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.

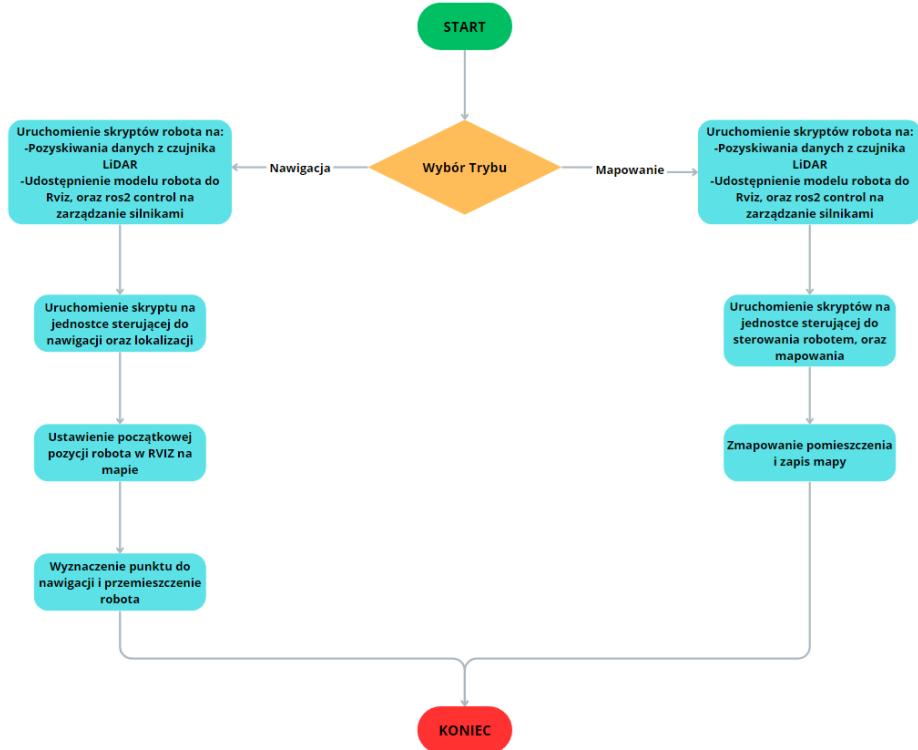
### 3.1 Wymagania funkcjonalne

System powinien realizować następujące funkcje:

- Zdalne sterowanie robotem poprzez klawiaturę (teleop\_twist\_keyboard [6]) w celu eksploracji i mapowania otoczenia
- 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

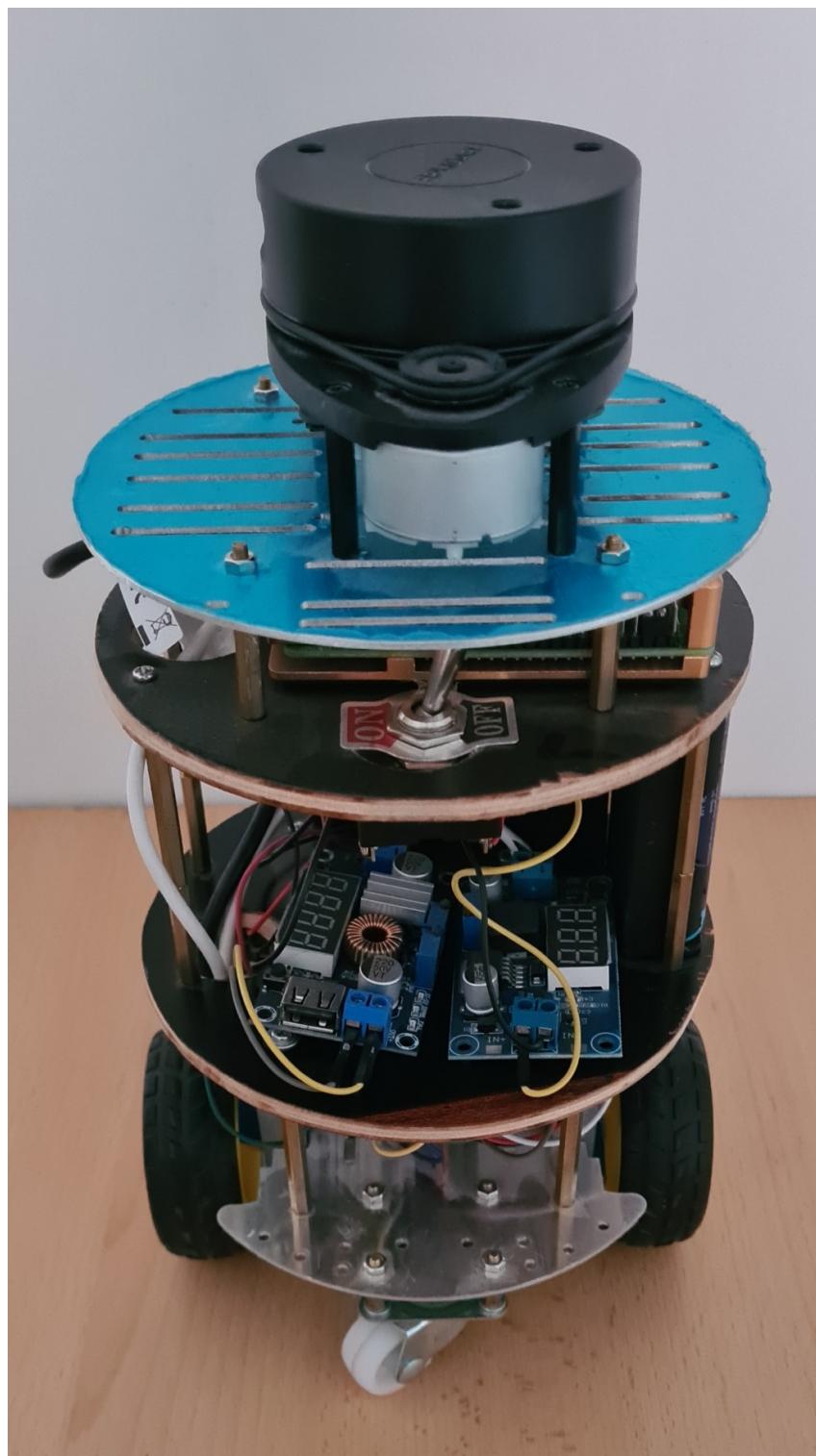
### 3.3.2 Napęd

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

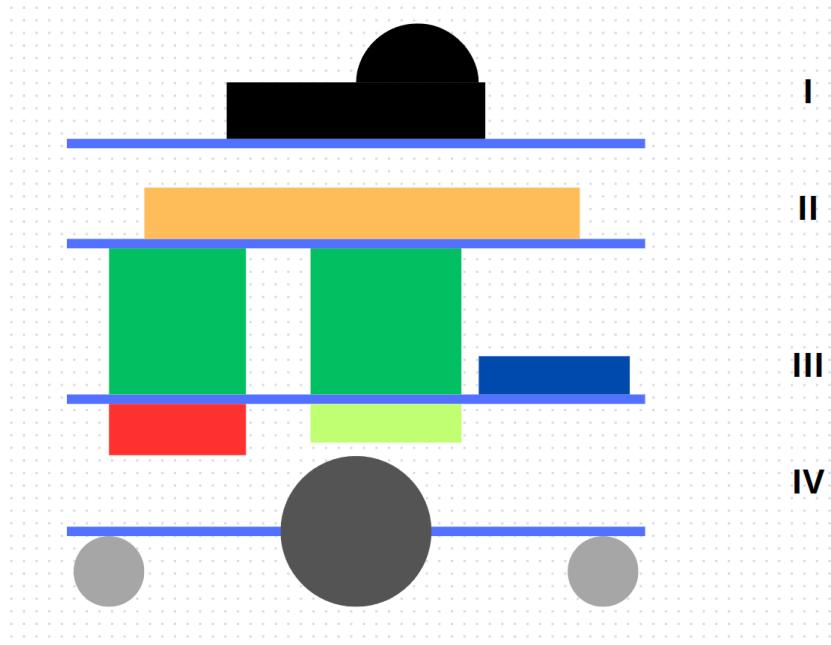
### 3.3.3 Zasilanie

- 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



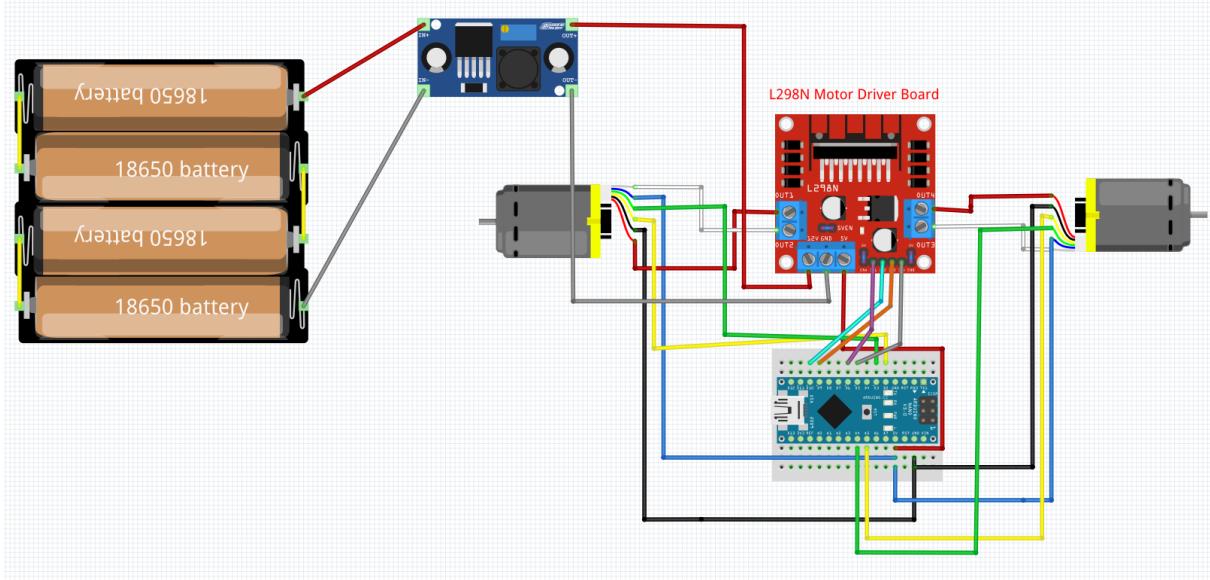
Rysunek 3.2: Zdjęcie przedstawiające zbudowanego robota



Rysunek 3.3: Uproszczony schemat przedstawiający budowę robota

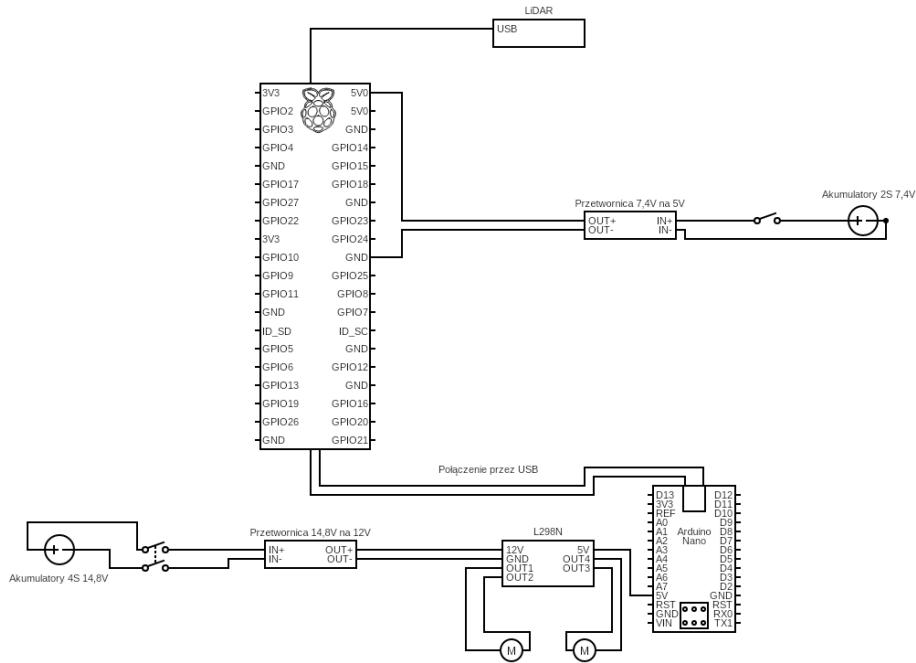
Na schemacie z rysunku 3.3 przedstawiono konkretnie poziomy robota, takie jak:

- 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 (Żółty element) i sterownik L298N (Czerwony element) z silnikami (Szare elementy)



Rysunek 3.4: Schemat połączenia silników z Arduino i L298N

Na zaprezentowanym schemacie rys. 3.4 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 3.5: Schemat elektryczny

Na schemacie 3.5 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

### 3.5.1 Etap 1: Przygotowanie platformy sprzętowej

- Instalacja systemu Ubuntu 22.04 na Raspberry Pi
- Konfiguracja połączenia SSH
- Instalacja ROS 2 Humble

### 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
  - ROS2 Control do sterowania napędem
- Integracja i testy systemu

# Rozdział 4

## Specyfikacja użytkowa

W tym rozdziale przedstawiono wymagania użytkownika oraz specyfikację funkcjonalną systemu. Opisano kategorie użytkowników, sposób obsługi, administrację systemem, kwestie bezpieczeństwa oraz przykłady działania 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 sinlików, 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, lokalizację i nawigację.

Pierwszym krokiem jest uruchomienie robota, w tym celu należy włączyć zasilanie dla Raspberry Pi i silników.

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). Przygotowanie tych terminali polega na odpowiednim:

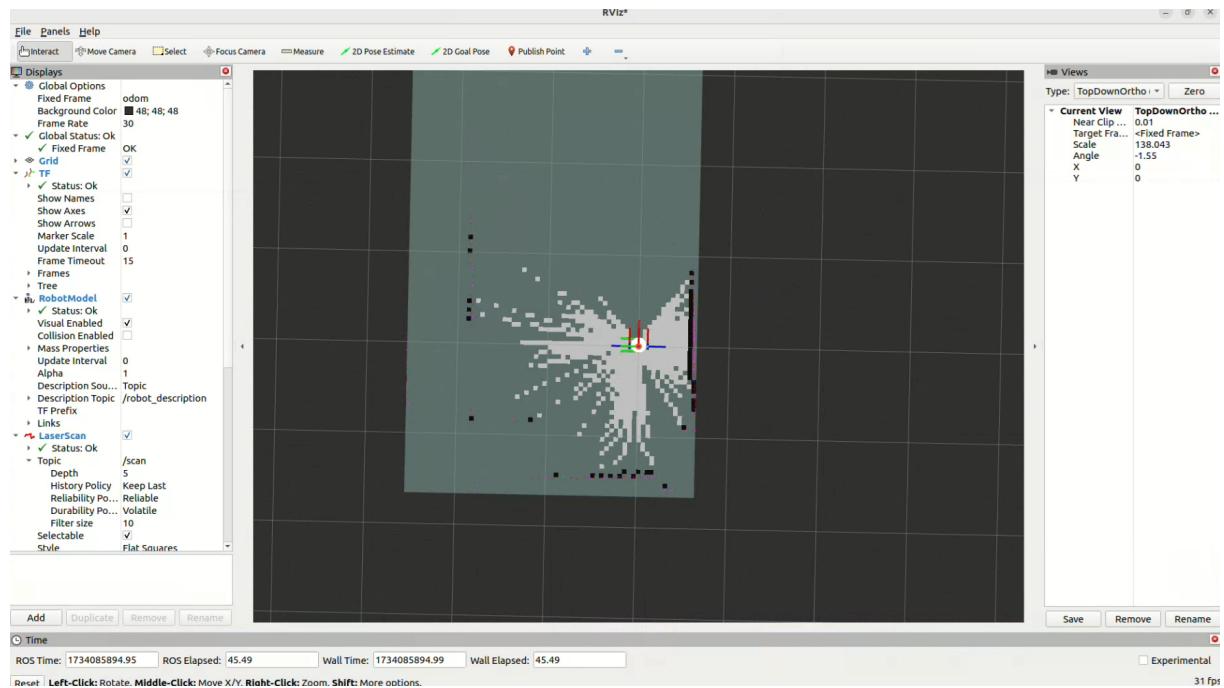
Dla jednostki sterującej: wejściu do katalogu roboczego, uruchomienie komend source /opt/ros/humble/setup.bash, oraz source /install/setup.bash.

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. 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.

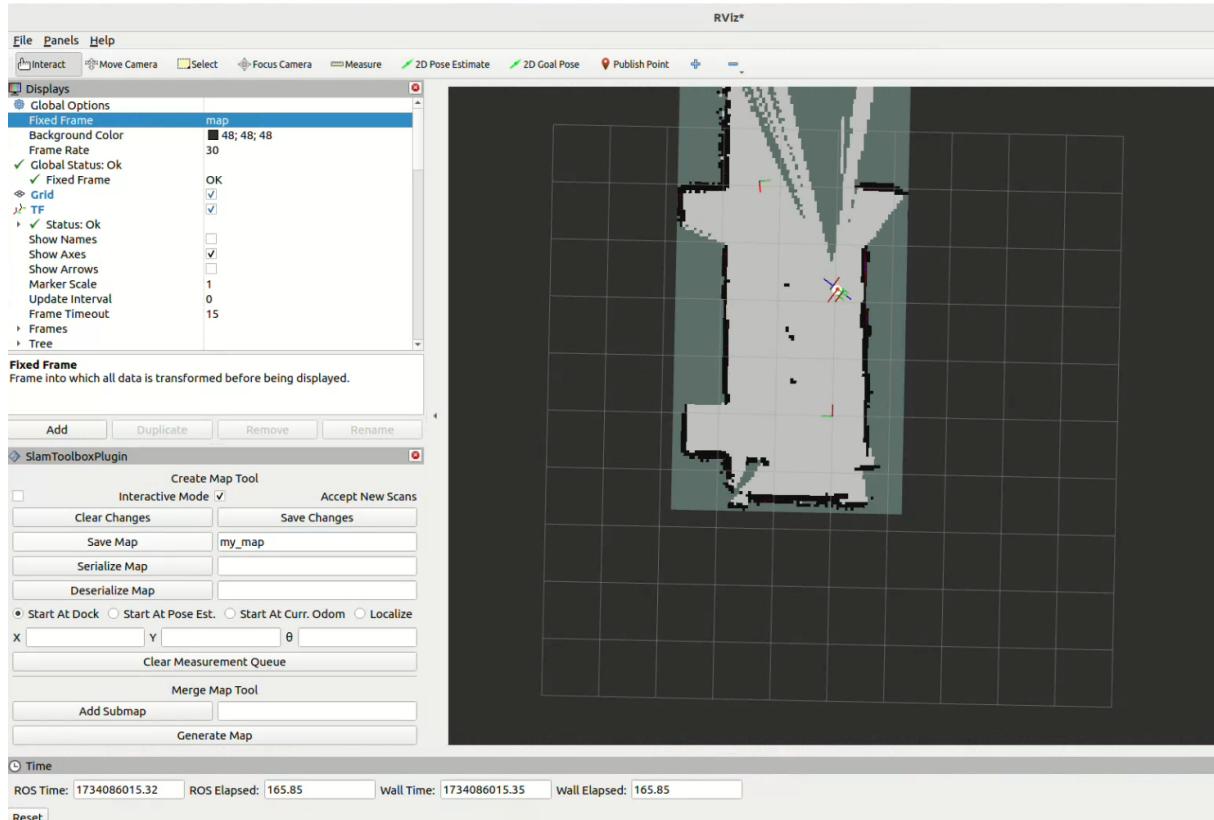
Po uruchomieniu tych skryptów, należy uruchomić program Rviz, który pozwala na wizualizację mapy i położenia robota. Komenda to: rviz2. Następnym krokiem jest uruchomienie skryptu odpowiedzialny za sterowanie robotem za pomocą klawiatury. Komenda to: ros2 launch robot\_teleop.launch.py. W tym momencie należy przejść do terminala na jednostce sterującej, i za pomocą klawiatury sterować robotem.

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 komendy z klawiatury, pomieszczenie jest mapowane.



Rysunek 4.1: Okno RVIZ po uruchomieniu skryptów.

Gdy wystarczająco dużo pomieszczenia zostanie zmapowane, należy zapisać mapę. 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.



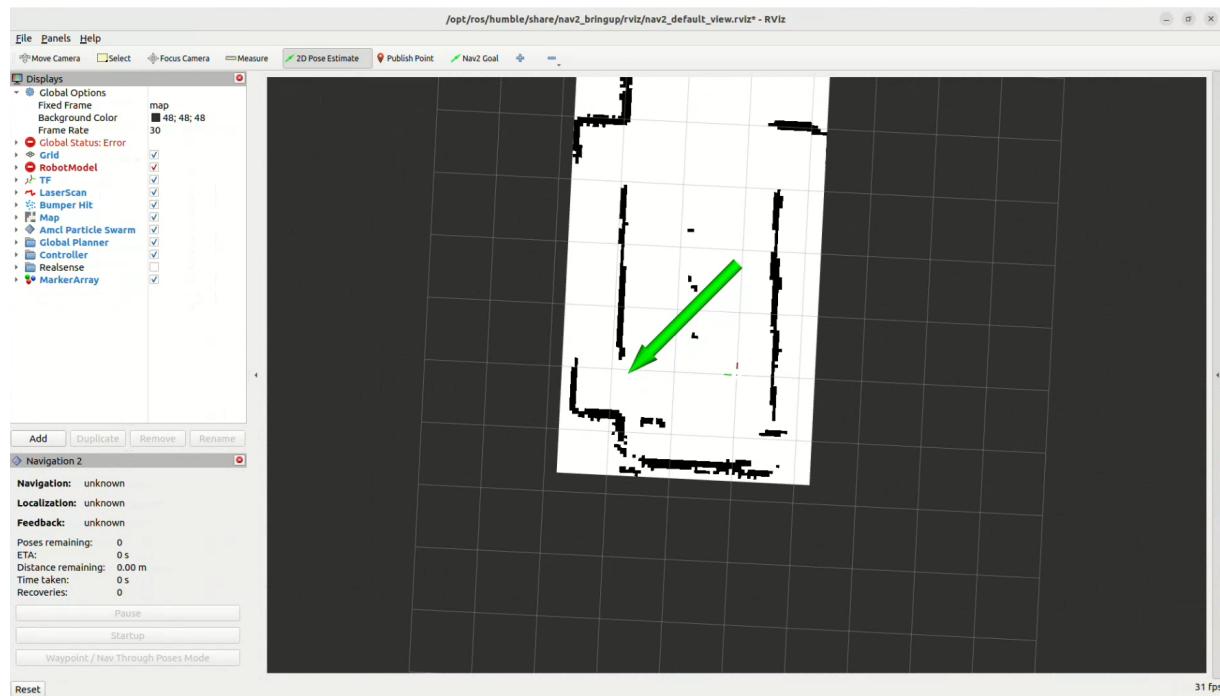
Rysunek 4.2: Zmapowane pomieszczenie i zapis mapy jako plik my\_map.

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

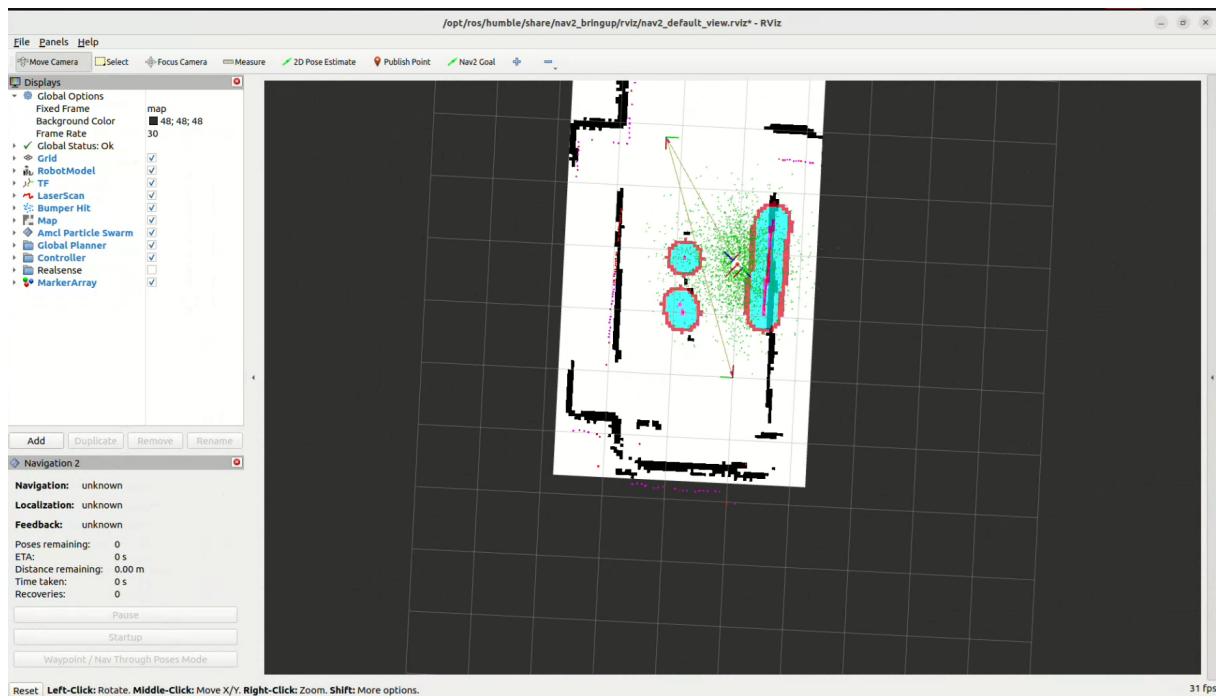
Następnie, należy uruchomić skrypt odpowiedzialny za lokalizację robota na mapie, oraz nawigację robota.

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.

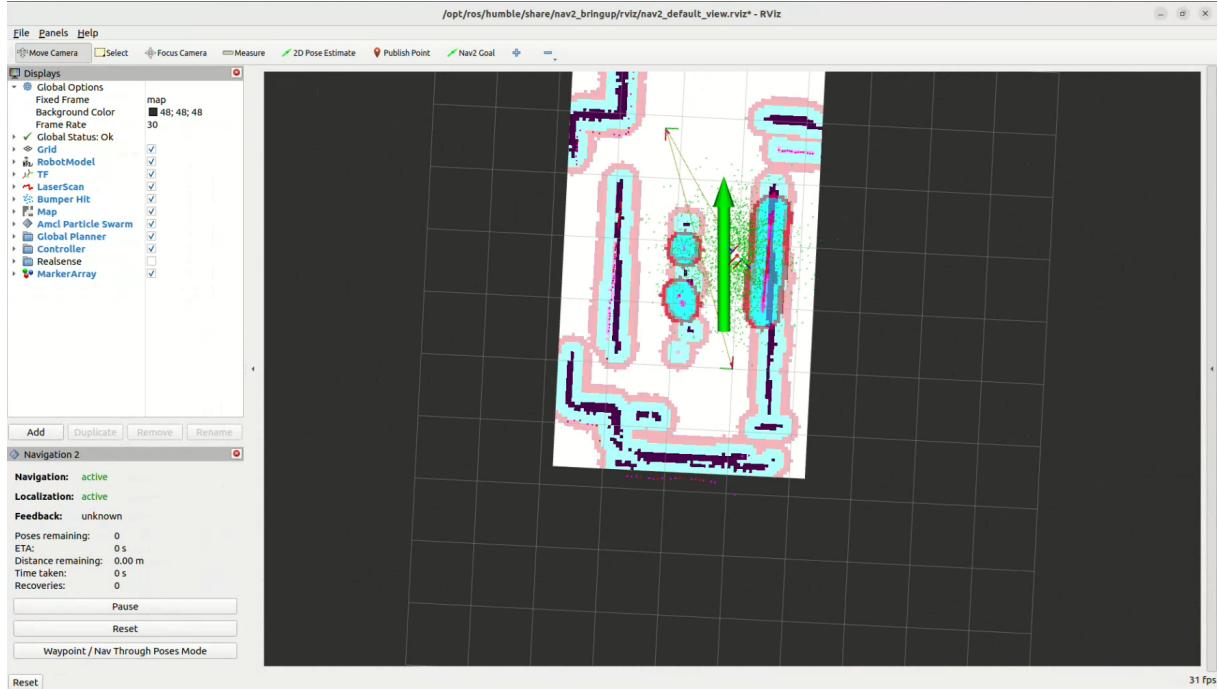


Rysunek 4.3: RVIZ po uruchomieniu skryptu do lokalizacji i nawigacji i wybraniu pozycji robota.



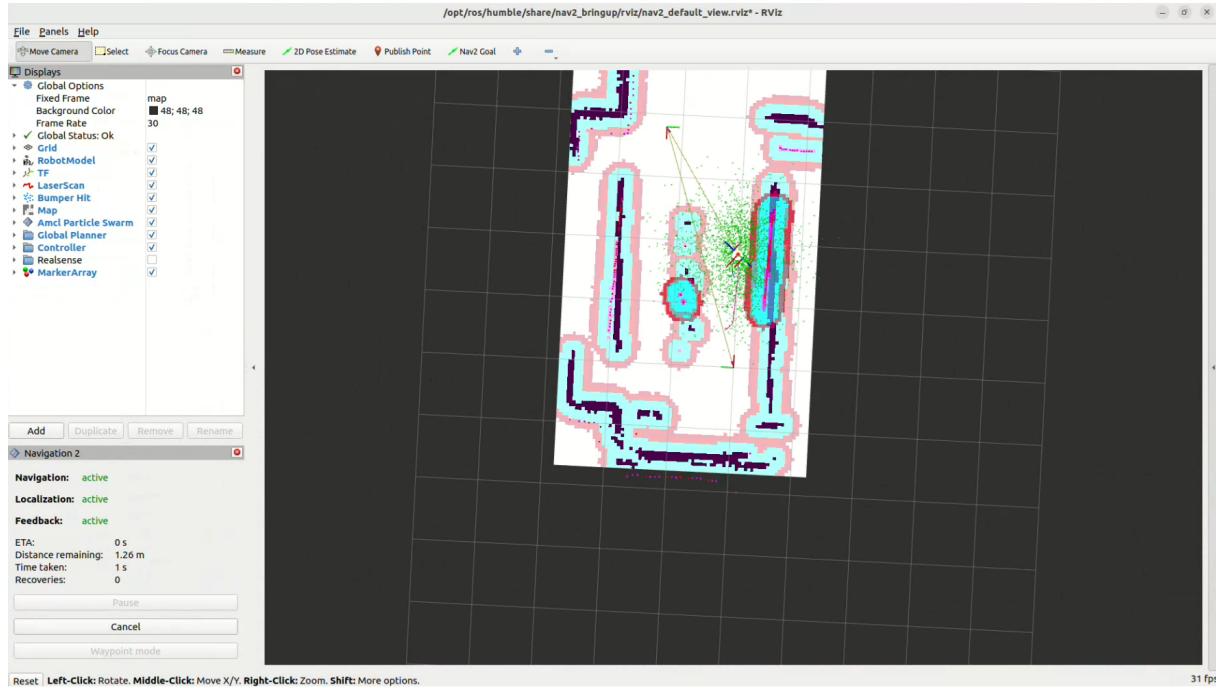
Rysunek 4.4: Wizualizacja rozproszonych punktów lokalizacji robota.

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.

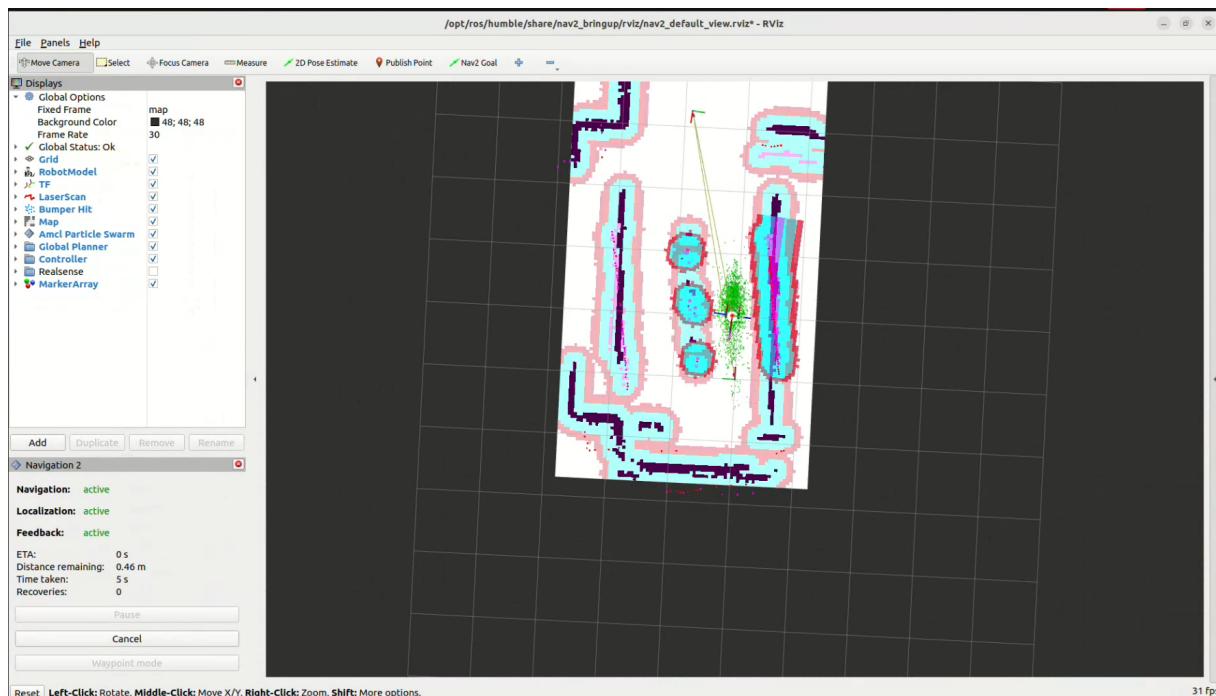


Rysunek 4.5: Wybór celu robota.

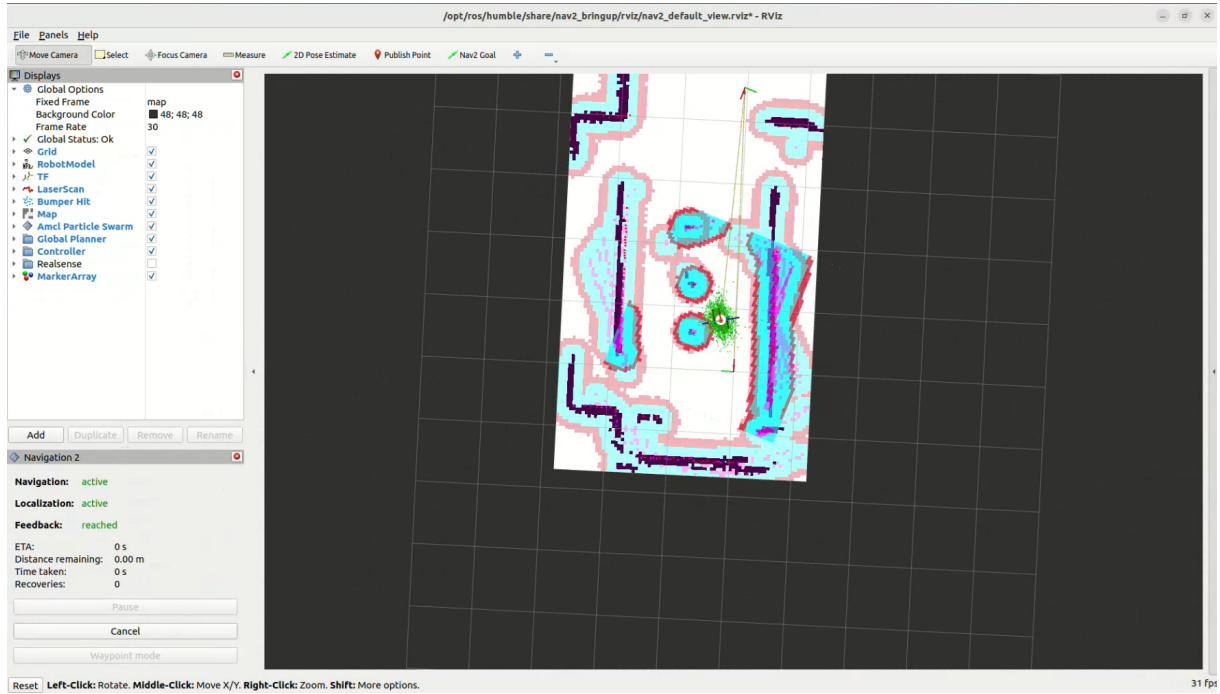
Robot po wybraniu celu zaczyna poruszać się w kierunku celu, omijając przeszkody na swojej drodze.



Rysunek 4.6: Wyznaczenie trasy.



Rysunek 4.7: Nawigacja robota i zmniejszanie się chmury przewidywanej lokalizacji robota.



Rysunek 4.8: Dotarcie 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 korzystaniu z klawiatury, można zmienić prędkość przez naciśnięcie q i w. Do modyfikacji prędkości podczas nawigacji, można zmienić prędkość w pliku konfiguracyjnym Nav2 (nav2\_params.yaml).

## 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 [9].

# 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
- SLAM Toolbox - pakiet do mapowania otoczenia
- Nav2 - pakiet do nawigacji robota wraz z algorytmem AMCL do lokalizacji
- ROS2 Control - platforma do sterowania robotem
- DiffDrive Arduino - pakiet zapewniający interfejs między ROS2 Control a Arduino
- ROS arduino bridge - pakiet zapewniający oprogramowanie do sterowania silnikami i zbierania danych z enkoderów

## 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[5], który zapewnia interfejs między ROS 2 a Arduino.

Program ten składa się z następujących plików:

- ROSArduinoBridge.ino - skrypt obsługuje komunikację szeregową z prędkością 57600 b/d (jednostka szybkości transmisji danych) i może kontrolować silniki poprzez różne sterowniki (np. Pololu VNH5019, MC33926 czy L298N). Główna pętla programu ciągle nasłuchuje komend przychodzących przez port szeregowy, wykonuje obliczenia PID w regularnych odstępach czasu oraz aktualizuje stan serwomechanizmów jeśli są używane. Cały kod znajduje się w pliku ROSArduinoBridge.ino w załączniku Oprogramowanie.

- 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 pinów 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, definiując porty i piny 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
  - Płynne 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 i mogą być wykorzystywane przez inne węzły, na przykład do tworzenia mapy otoczenia czy nawigacji.

### 5.3.3 Sterowanie silnikami i model robota

Za sterowanie silnikami odpowiadają następujące pliki:

- 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 ROS2 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 ROS2 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 (fixed joints)
  - Właściwości fizyczne elementów:
    - \* Masy
    - \* Momenty bezwładności
    - \* Parametry kolizji

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

- 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 ROSArduinoBridge, czyli 57600 bodów, zliczenia enkoderów na obrót koła - 2194 wyznaczone przez testy, oraz ustawienie maksymalnej i minimalnej prędkości dla każdego silnika.
- 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  # Reverted frame  
7  
8      left_wheel_names: [ 'left_wheel_joint' ]  
9      right_wheel_names: [ 'right_wheel_joint' ]  
10     wheel_separation: 0.14  
11     wheel_radius: 0.035  
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. Utworzony program składa się z następujących plików:

- `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.

Kluczowym elementem jaki należy tutaj opisać jest wykorzystanie biblioteki 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

Algorytm działa iteracyjnie, wykorzystuje do tego metodę Levenberga-Marquardta, która łączy w sobie:

- Szybkość metody Gaussa-Newtona (dobre do precyzyjnych korekt)
- Stabilność metody największego spadku (pomocne przy większych korektach)

Dzięki takiemu podejściu SLAM Toolbox może tworzyć dokładne mapy, nawet gdy niektóre pomiary są niedokładne.

---

```
1 # ROS 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 tutaj tryb mapowania, oraz nazwy ramek, z których korzysta program.

```
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 size
enable_interactive_mode: true
```

---

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

Dostosowano parametry po przeprowadzeniu testów i analizie wyników. Zmniejszono odstęp czasowy między aktualizacjami mapy, zwiększyto rozdzielczość mapy oraz zwiększyto czas oczekiwania na transformację. Dostosowano również rozmiar stosu, aby zapobiec przepełnieniu pamięci. Zmieniony został też maksymalny dystans na jaki pozwala zamontowany model LiDAR-a.

### 5.3.5 Nawigacja i lokalizacja

Do nawigacji i lokalizacji robota wykorzystano pakiet Nav2, który pozwala na wyznaczanie trasy i lokalizację robota na mapie. Program ten składa się z następujących plików:

- nav\_localization.launch.py - plik ten odpowiada za konfigurację i uruchomienie węzłów odpowiedzialnych za nawigację i lokalizację robota. Podstawowym elementem jest serwer mapy, który przechowuje i udostępnia mapę otoczenia. Za lokalizację odpowiada algorytm AMCL (adaptacyjna lokalizacja Monte Carlo), który wykorzystuje chmurę częstek do określenia pozycji robota na mapie. System kontroli ruchu składa się z kontrolera ruchu, który steruje silnikami, planera ścieżki wyznaczającego trasę oraz modułu zachowań awaryjnych, który pozwala na reagowanie w sytuacjach nieoczekiwanych. Całością zarządza nawigator oparty na drzewach zachowań.

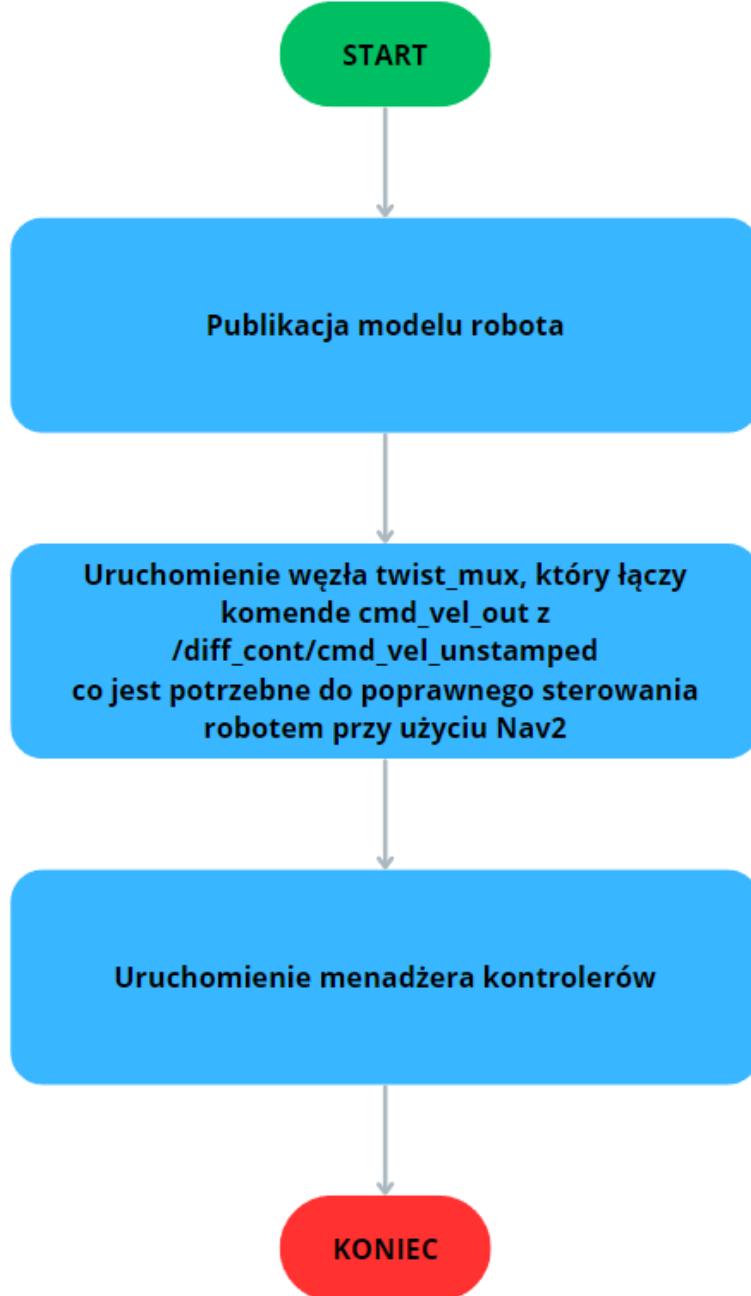
Program konfiguruje się poprzez zestaw parametrów, z których najważniejsze to praca w czasie rzeczywistym (zamiast symulacji), automatyczne uruchomienie systemu oraz ścieżki do plików z mapą i konfiguracją.

Szczególną uwagę poświęcono konfiguracji modułu AMCL, gdzie określono liczbę wykorzystywanych częstek (od 3000 do 10000), parametry modelu ruchu (wszystkie ustawione na 0.2) oraz warunki aktualizacji lokalizacji (minimalny przesuw 0.1m i obrót 0.1 radiana). Dodatkowo włączono funkcję globalnej lokalizacji przy starcie systemu, co pozwala na określenie pozycji bez znajomości stanu początkowego.

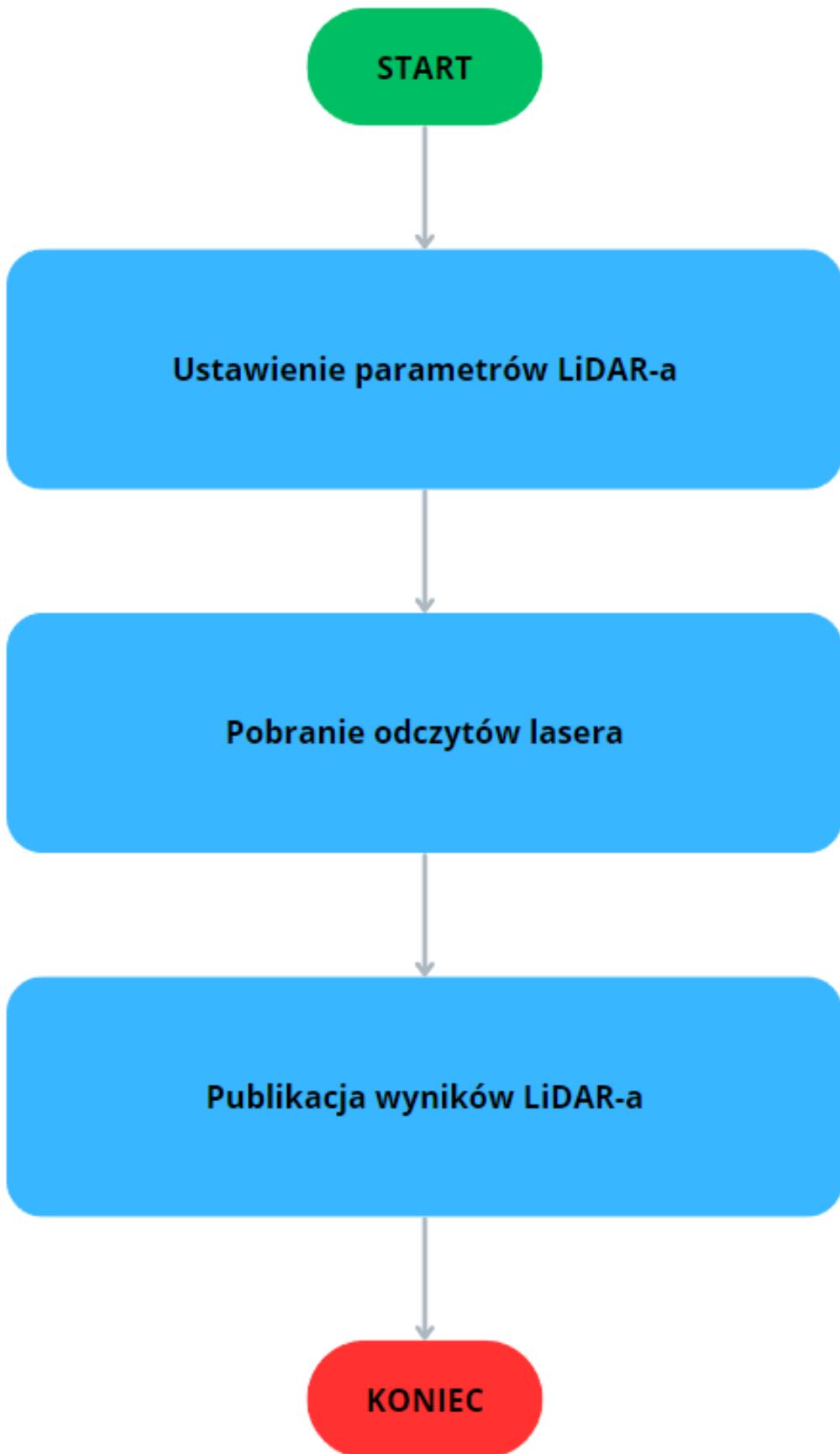
Do nawigacji tworzone są dodatkowe mapy kosztów (ang. "costmap") lokalna i globalna. Lokalna mapa kosztów odpowiada za omijanie obiektów które nie były na mapie podczas jej tworzenia, natomiast globalna mapa kosztów tworzy strefy wokół obiektów już zmapowanych, czyli np. ściany, co pozwala na tworzenie tras wokół przeszkód.

- amcl.yaml - plik konfiguracyjny algorytmu AMCL odpowiedzialnego za precyzyjną lokalizację robota na mapie. Zawiera parametry takie jak:
  - Współczynniki szumu odometrii (alpha1-5) ustawione na 0.2 dla zoptymalizowanej dokładności
  - Parametry filtracji wiązki lasera dla lepszej jakości skanów
  - Liczba cząstek (500-3000) dobrana dla optymalnej równowagi między dokładnością a wydajnością
  - Częstotliwość aktualizacji pozycji i orientacji (co 0.1m/rad) zapewniająca płynne śledzenie
  - Parametry adaptacji (szybkiej i wolnej) dla stabilnej lokalizacji
  - Konfiguracja ramek odniesienia i transformacji dla poprawnej integracji z systemem ROS

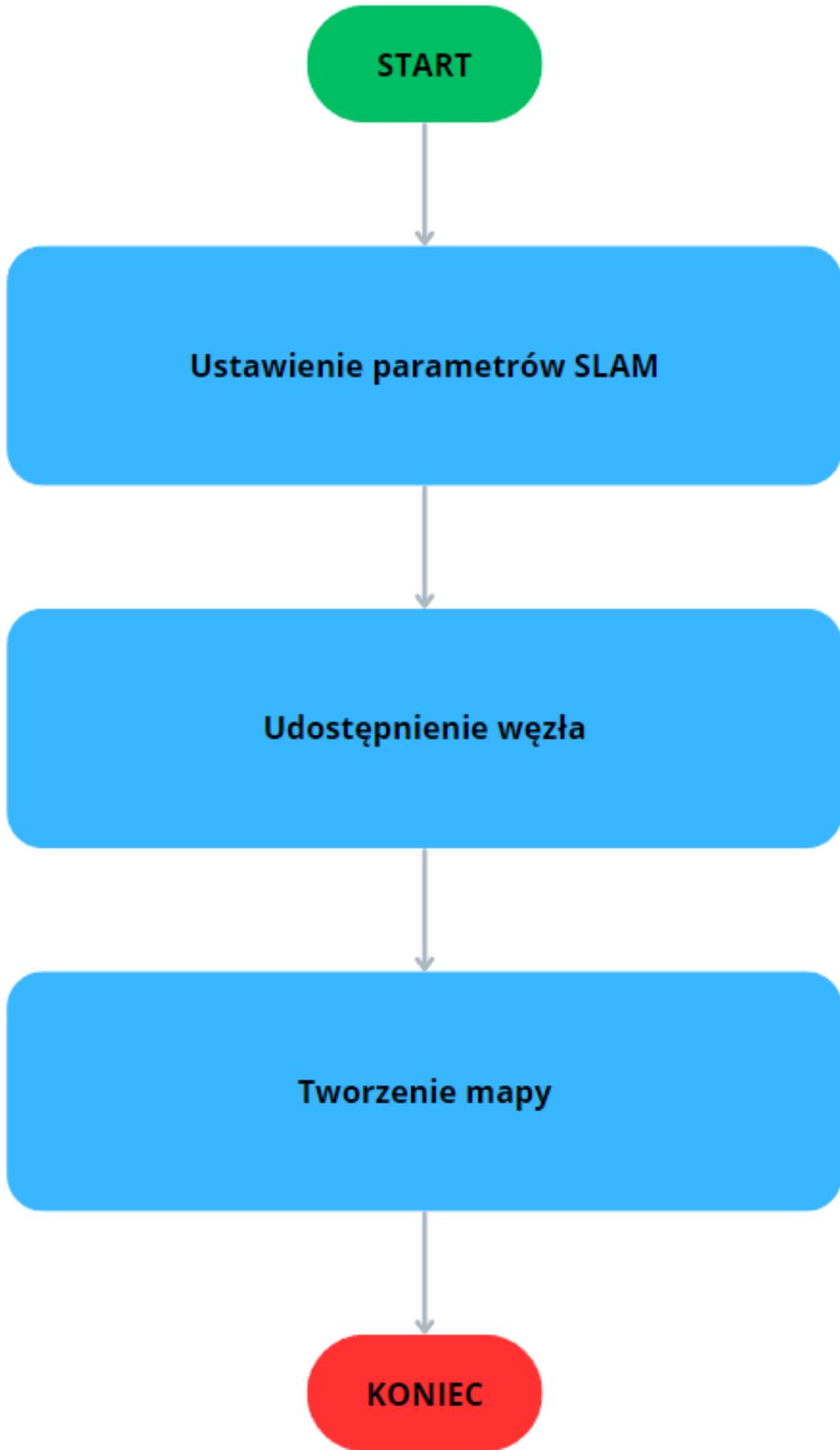
## 5.4 Diagramy UML prezentujące działanie konkretnych programów



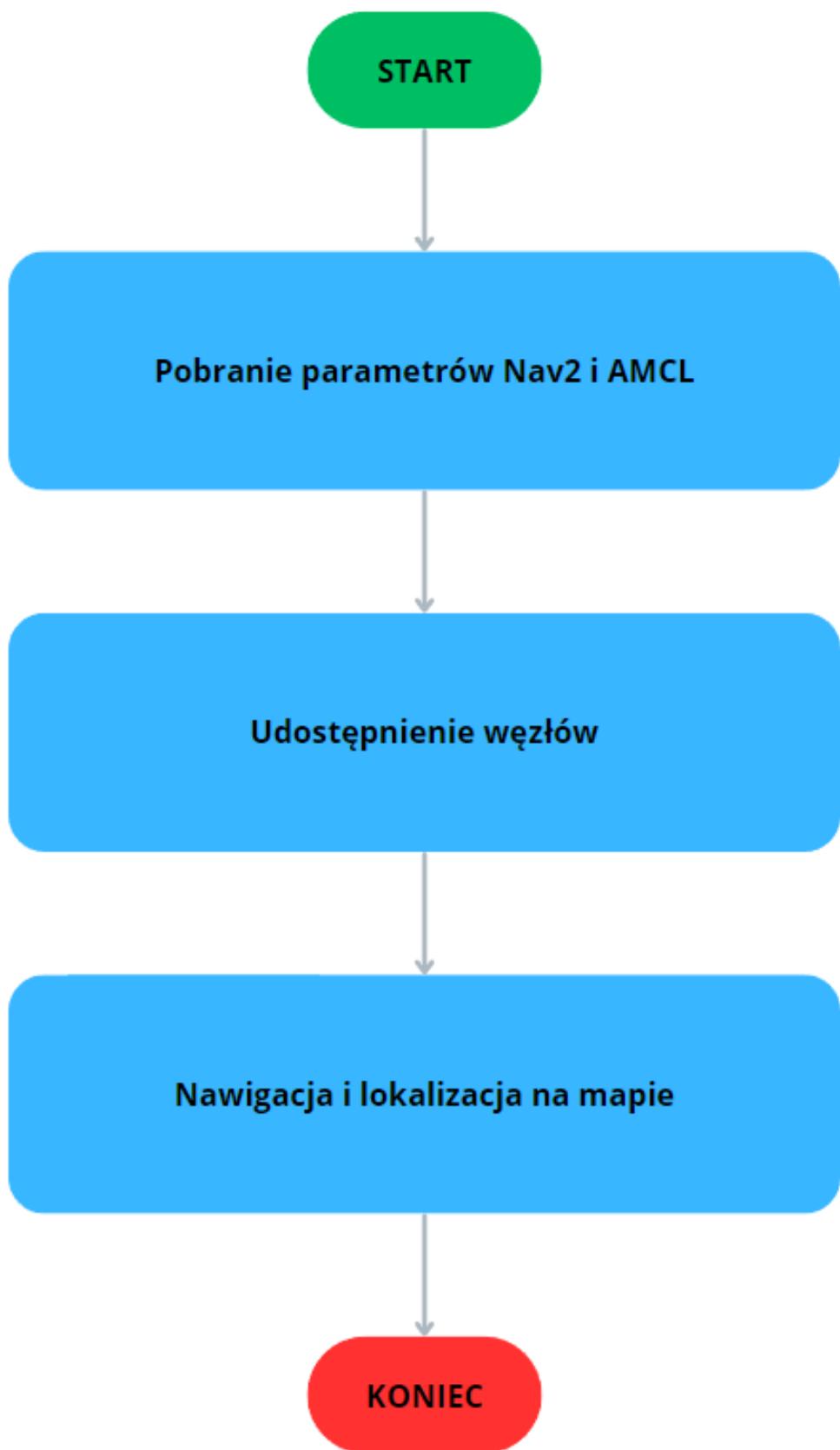
Rysunek 5.10: Diagram UML prezentujący działanie programu `model_controll.launch.py`



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



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



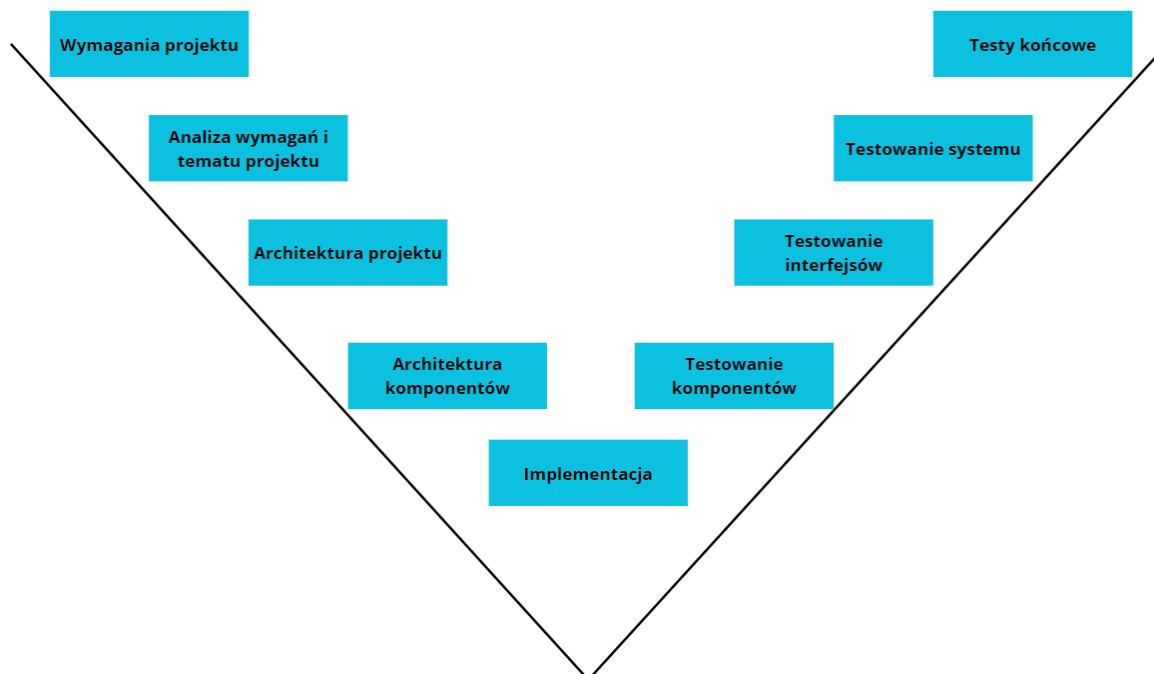
Rysunek 5.13: Diagram UML prezentujący działanie programu `nav_localization.launch.py`



# Rozdział 6

## Weryfikacja i walidacja

W 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.



Rysunek 6.1: Model V

## 6.1 Model V

- **Wymagania projektu:** Główne cele i założenia:
  - Automatyczne tworzenie mapy otoczenia
  - Autonomiczna nawigacja robota
  - Precyzyjna lokalizacja
  - Wykrywanie i omijanie przeszkód
- **Analiza wymagań i tematu:** Szczegółowe wymagania funkcjonalne:
  - Zdalne sterowanie robotem
  - Zapisywanie i odczyt map
  - Autonomiczna nawigacja do zadanych punktów
  - Bezpieczne poruszanie się
- **Architektura projektu:** Ogólna struktura systemu:
  - Podział na podsystemy (napęd, sensory, sterowanie)
  - Przepływ danych między modułami
  - Interfejsy komunikacyjne
  - Wybór technologii (ROS 2, Nav2, SLAM Toolbox)
- **Architektura komponentów:** Projektowanie szczegółowe:
  - Dobór komponentów sprzętowych
  - Projekt mechaniczny i elektryczny
  - Struktura oprogramowania
  - Interfejsy programowe
- **Implementacja:** Realizacja systemu:
  - Budowa robota
  - Programowanie sterowników
  - Integracja sensorów
  - Implementacja algorytmów

- **Testowanie komponentów:** Testy jednostkowe:
  - Weryfikacja napędów
  - Testy sensorów
  - Sprawdzenie sterowników
  - Testy algorytmów
- **Testowanie interfejsów:** Testy integracyjne:
  - Komunikacja między modułami
  - Synchronizacja danych
  - Przepływ informacji
  - Współpraca komponentów
- **Testowanie systemu:** Testy całościowe:
  - Mapowanie otoczenia
  - Lokalizacja robota
  - Planowanie trasy
  - Omijanie przeszkód
- **Testy końcowe:** Walidacja systemu:
  - Weryfikacja funkcjonalności
  - Testy wydajności
  - Testy niezawodności
  - Zgodność z wymaganiami

## 6.2 Organizacja eksperymentów

- **Testowanie jednostkowe:** Każdy moduł systemu, taki jak sterowanie silnikami, odczyt danych z LiDAR-a, tworzenie mapy, lokalizacja i nawigacja, został przetestowany indywidualnie. Testy jednostkowe obejmowały sprawdzenie poprawności działania poszczególnych funkcji i metod.
- **Testowanie integracyjne:** Po zakończeniu testów jednostkowych, moduły zostały zintegrowane i przetestowane pod kątem poprawności współdziałania. Testy integracyjne obejmowały sprawdzenie komunikacji między modułami oraz poprawności przesyłania danych, jak i ewentualnych błędów.
- **Testowanie systemowe:** Cały system został przetestowany w warunkach rzeczywistych. Testy systemowe obejmowały sprawdzenie poprawności działania systemu w różnych scenariuszach, takich jak zdalne sterowanie robotem, tworzenie mapy otoczenia, lokalizacja robota na mapie oraz autonomiczna nawigacja do wyznaczonych punktów.
- **Walidacja:** System został zweryfikowany pod kątem spełnienia wymagań użytkownika. Walidacja obejmowała sprawdzenie, czy system działa zgodnie z założeniami i spełnia oczekiwania użytkownika.

## 6.3 Przypadki testowe

Przypadki testowe obejmowały następujące scenariusze:

- **Testowanie zdalnego sterowania:** Sprawdzenie, czy robot reaguje poprawnie na polecenia z klawiatury.
- **Testowanie tworzenia mapy:** Sprawdzenie, czy system poprawnie tworzy mapę otoczenia na podstawie danych z LiDAR-a.
- **Testowanie lokalizacji:** Sprawdzenie, czy system poprawnie lokalizuje robota na zapisanej mapie.
- **Testowanie nawigacji:** Sprawdzenie, 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 następujące błędy:

- **Problem z zasilaniem:** Występowały problemy 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.
- **Problem 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.
- **Problem z błędnią lokalizacją robota na mapie:** Na wczesnym etapie testów pojawił się problem z sterowaniem robotem, 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 2 ścian - poprawnej i przed przeniesieniem. Rozwiązaniem problemu było poprawne podłączenie silników do sterownika i Arduino.

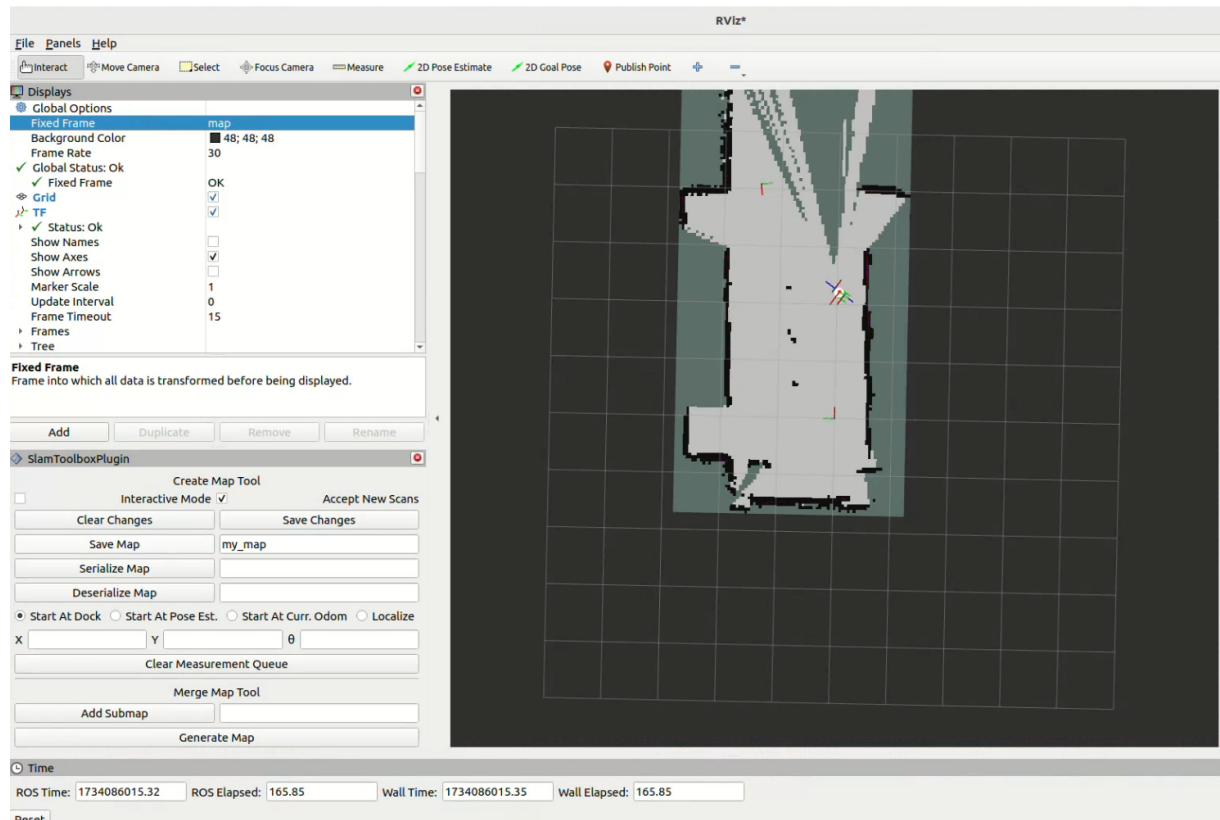


# Rozdział 7

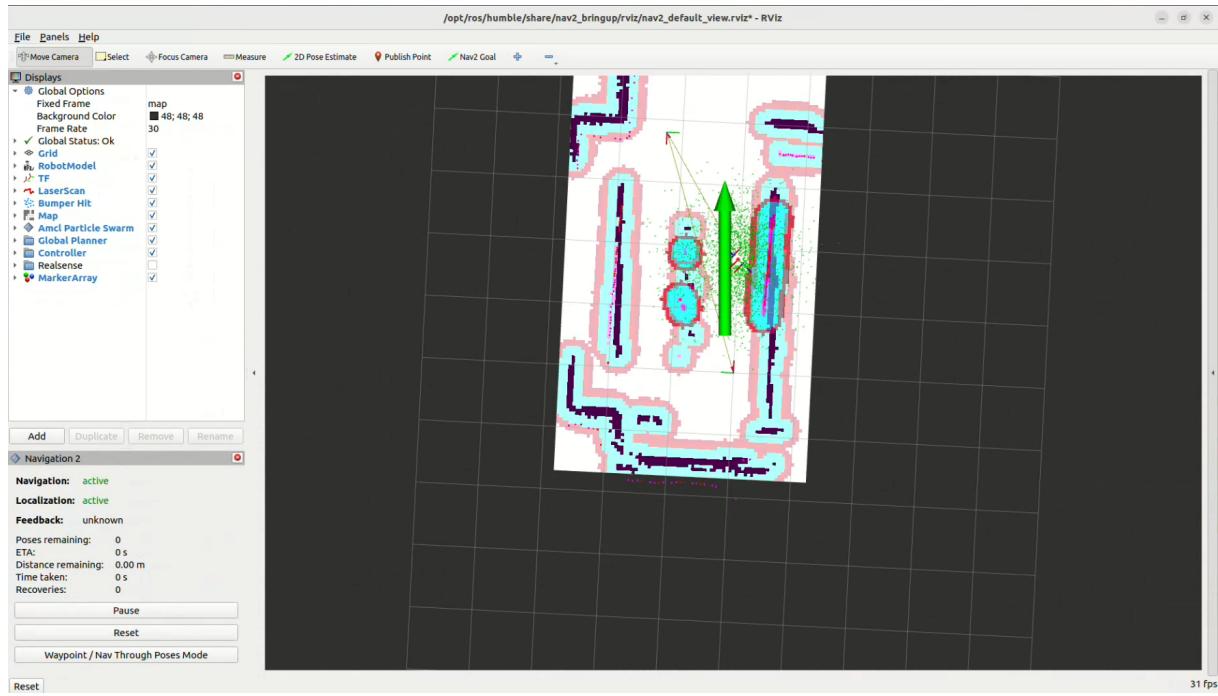
## Podsumowanie i wnioski

### 7.1 Uzyskane wyniki

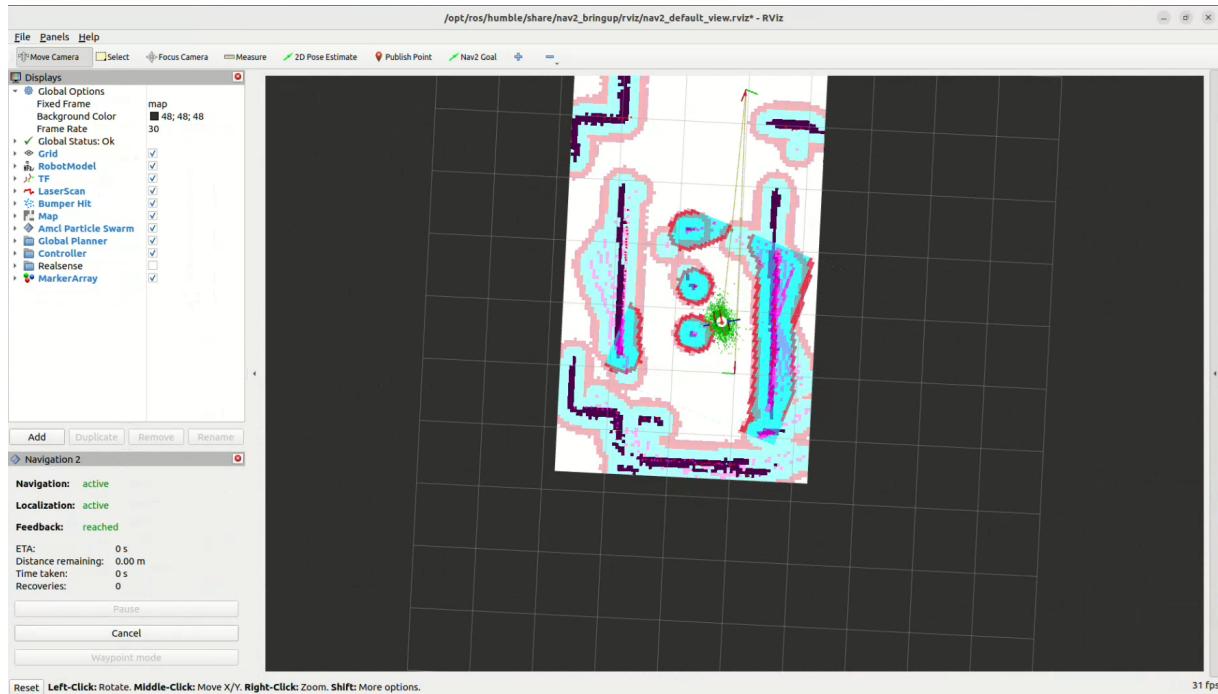
W wyniku przeprowadzonych prac udało się zrealizować wszystkie założone cele projektu. System autonomicznej nawigacji robota mobilnego został zaprojektowany, zaimplementowany i przetestowany. Robot poprawnie tworzy mapę otoczenia, lokalizuje się na zapisanej mapie oraz nawigował do wyznaczonych punktów, 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] 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).
- [4] Dieter Fox. „KLD-Sampling: Adaptive Particle Filters.” W: sty. 2001, s. 713–720.
- [5] 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).
- [6] 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).
- [7] Matti Kortelainen. „A short guide to ROS 2 Humble Hawksbill”. W: *School of Computing, University of Eastern Finland, Kuopio, Finland* (2023), s. 5.
- [8] 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](https://doi.org/10.21105/joss.02783). URL: <https://doi.org/10.21105/joss.02783>.
- [9] 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>.
- [10] 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).
- [11] Francisco Martin Rico. *A Concise Introduction to Robot Programming with ROS2*. Broken Sound Parkway NW: Taylor i Francis, 2022. ISBN: 1032264659.



## **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*)

ROS2 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 [11] . . . . .	6
2.2	Mapa dwóch pomieszczeń utworzona za pomocą SLAM Toolbox [8] . . . . .	7
3.1	Diagram przypadków użycia systemu . . . . .	12
3.2	Zdjęcie przedstawiające zbudowanego robota . . . . .	14
3.3	Uproszczony schemat przedstawiający budowę robota . . . . .	15
3.4	Schemat połączenia silników z Arduino i L298N . . . . .	16
3.5	Schemat elektryczny . . . . .	17
4.1	Okno RVIZ po uruchomieniu skryptów. . . . .	21
4.2	Zmapowane pomieszczenie i zapis mapy jako plik my_map. . . . .	22
4.3	RVIZ po uruchomieniu skryptu do lokalizacji i nawigacji i wybraniu pozycji robota. . . . .	23
4.4	Wizualizacja rozproszonych punktów lokalizacji robota. . . . .	24
4.5	Wybór celu robota. . . . .	25
4.6	Wyznaczenie trasy. . . . .	26
4.7	Nawigacja robota i zmniejszanie się chmury przewidywanej lokalizacji robota.	26
4.8	Dotarcie robota do celu. . . . .	27
5.1	Wizualizacja połączenia skryptów w systemie . . . . .	30
5.2	Fragment kodu źródłowego z pliku encoder_driver.ino z obsługą przerwań do precyzyjnego zliczania impulsów z enkoderów . . . . .	32
5.3	Kod źródłowy z pliku motor_driver.h . . . . .	33
5.4	Kod źródłowy z pliku encoder_driver.h . . . . .	33
5.5	Kod źródłowy pliku model.urdf.xacro . . . . .	37
5.6	Fragment kodu z pliku my_controllers.yaml . . . . .	38
5.7	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	39
5.8	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	40
5.9	Fragment kodu z pliku mapper_params_online_async.yaml . . . . .	41
5.10	Diagram UML prezentujący działanie programu model_controll.launch.py	44
5.11	Diagram UML prezentujący działanie programu rplidar.launch.py . . . . .	45
5.12	Diagram UML prezentujący działanie programu slam.launch.py . . . . .	46

5.13	Diagram UML prezenujący działanie programu nav_localization.launch.py	47
6.1	Model V . . . . .	49
7.1	Wizualizacja mapy otoczenia. . . . .	55
7.2	Rozpoczęcie nawigacji. . . . .	56
7.3	Dotarcie do celu . . . . .	56