

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 Studia literaturowe . . . . .	6
2.4 ROS 2 . . . . .	6
2.5 SLAM Toolbox . . . . .	7
2.5.1 Navigation2 (Nav2) i lokalizacja . . . . .	9
2.5.2 ROS2 Control i sterowanie napędami . . . . .	9
2.6 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 . . . . .	14
3.3.3 Zasilanie . . . . .	15
3.4 Budowa robota i sposób połączenia silników z kontrolerem . . . . .	17
3.5 Metodyka i etapy realizacji . . . . .	21
3.5.1 Etap 1: Przygotowanie platformy sprzętowej . . . . .	21
3.5.2 Etap 2: Implementacja sterowania napędem . . . . .	21
3.5.3 Etap 3: Integracja sensorów . . . . .	21
3.5.4 Etap 4: Implementacja oprogramowania . . . . .	21

<b>4 Specyfikacja użytkowa</b>	<b>23</b>
4.0.1 Wymagania sprzętowe i programowe . . . . .	23
4.0.2 Sposób aktywacji i korzystania z robota z przykadem działania . . . . .	24
4.1 Administracja systemem . . . . .	31
4.2 Kwestie bezpieczeństwa . . . . .	31
4.3 Scenariusze korzystania z systemu . . . . .	32
<b>5 Specyfikacja techniczna</b>	<b>33</b>
5.1 Idea robota autonomicznego mapującego w czasie rzeczywistym . . . . .	33
5.2 Architektura systemu . . . . .	34
5.3 Struktura systemu i objaśnienie działania algorytmów . . . . .	35
5.3.1 Skrypt Arduino . . . . .	35
5.3.2 Skrypt do obsługi LiDAR-a . . . . .	42
5.3.3 Skrypt do obsługi silników i tworzenia modelu robota . . . . .	43
5.3.4 Skrypt do tworzenia mapy . . . . .	49
5.3.5 Skrypty do nawigacji i lokalizacji . . . . .	51
5.3.6 Diagramy UML prezentujące działanie konkretnych programów . .	54
<b>6 Weryfikacja i walidacja</b>	<b>57</b>
6.1 Model V . . . . .	57
6.2 Organizacja eksperymentów . . . . .	58
6.3 Przypadki testowe . . . . .	58
6.4 Wykryte i usunięte błędy . . . . .	58
6.5 Wyniki badań eksperimentalnych . . . . .	59
<b>7 Podsumowanie i wnioski</b>	<b>61</b>
7.1 Uzyskane wyniki . . . . .	61
7.2 Kierunki dalszych prac . . . . .	61
<b>Bibliografia</b>	<b>63</b>
<b>Spis skrótów i symboli</b>	<b>67</b>
<b>Źródła</b>	<b>69</b>
<b>Lista dodatkowych plików, uzupełniających tekst pracy</b>	<b>71</b>
<b>Spis rysunków</b>	<b>74</b>
<b>Spis tabel</b>	<b>75</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 [1]. 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

Głównym 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 sześciu 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.

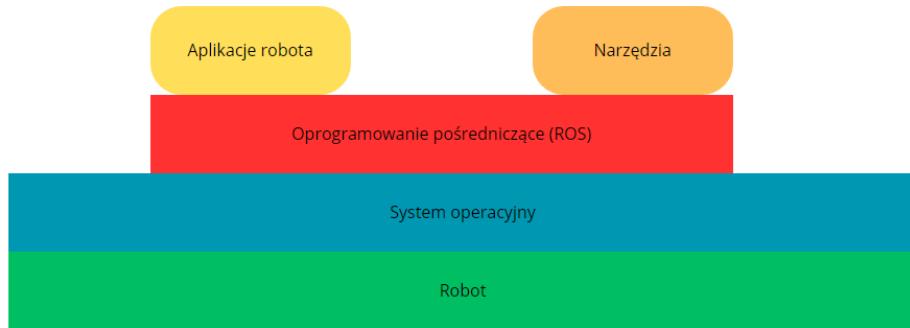
### 2.2 Szczegółowe sformułowanie problemu

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 Studia literaturowe

## 2.4 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 [10]. Na poniższej ilustracji przedstawiono położenie takiego pośrednika.

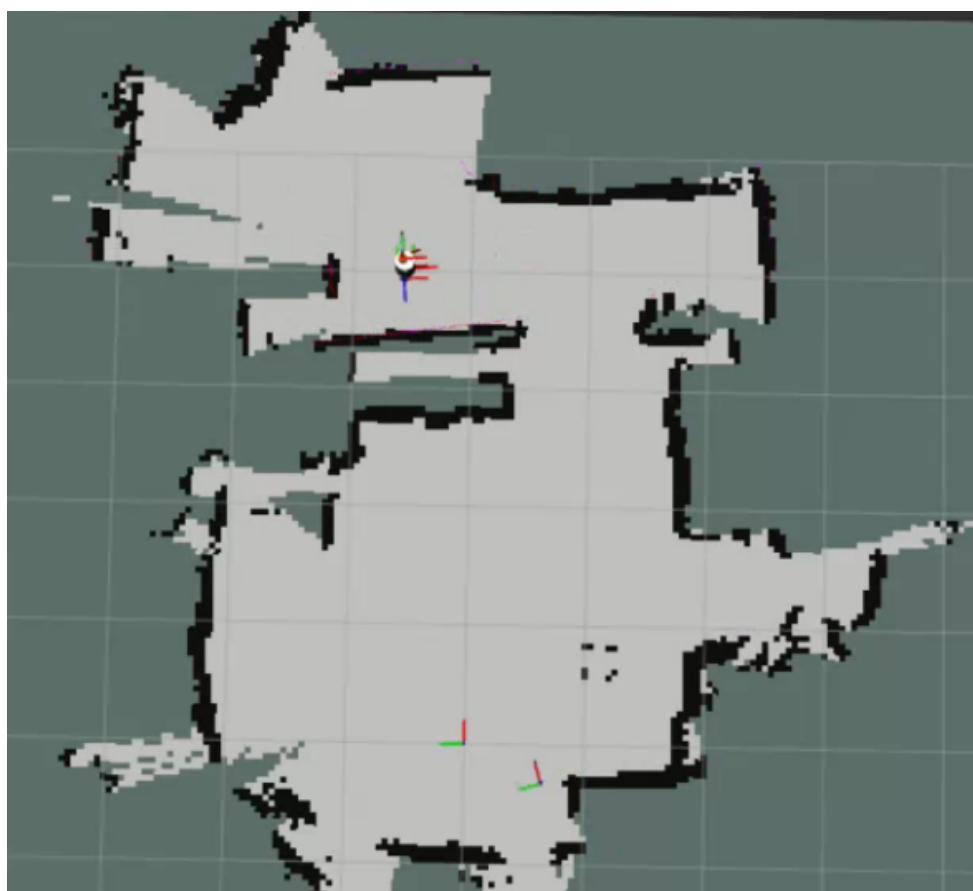


Rysunek 2.1: Reprezentacja pośrednika w systemie robota [10]

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.[6]

## 2.5 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. [7] 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 [7]

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.

### 2.5.1 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. [8]

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"), 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. [3]

### 2.5.2 ROS2 Control i sterowanie napędami

ROS2 Control to platforma do sterowania, zarządzania i komunikacji pomiędzy urządzeniami w robotach z oprogramowaniem. [2]. Dzięki takiemu rozwiązaniu można w łatwy sposób zarządzać silnikami, enkoderami, czy innymi urządzeniami w robocie. Dzięki temu 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. [9]

## 2.6 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.

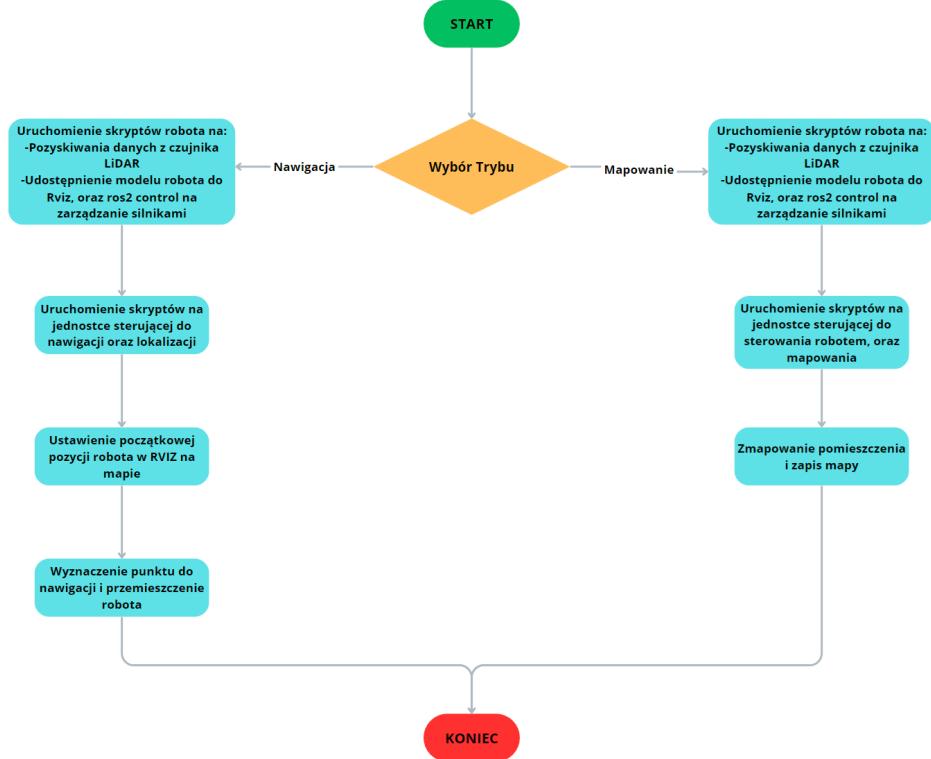
### 3.1 Wymagania funkcjonalne

System powinien realizować następujące funkcje:

- Zdalne sterowanie robotem poprzez klawiaturę (teleop\_twist\_keyboard [5]) 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



Rysunek 3.2: Raspberry Pi 4 model B

- Arduino Nano - sterownik silników:
  - Obsługa enkoderów
  - Komunikacja szeregową z Raspberry Pi



Rysunek 3.3: Arduino Nano

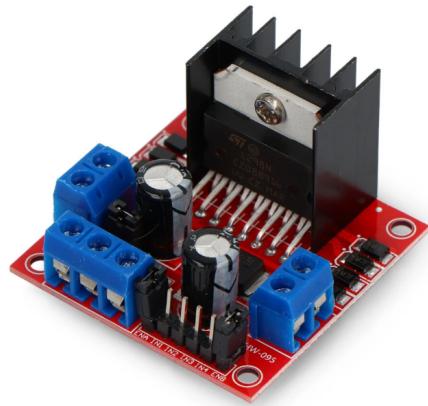
### 3.3.2 Napęd

- 2x silnik DC 12V 240RPM z metalową przekładnią
- Wbudowane enkodery magnetyczne Halla



Rysunek 3.4: Silnik DC 12V 240RPM typu L z przekładnią metalową - magnetyczny enkoder Halla - Waveshare 22346

- Sterownik L298N - dwukanałowy mostek H



Rysunek 3.5: L298N - dwukanałowy sterownik silników - moduł 12V/2A

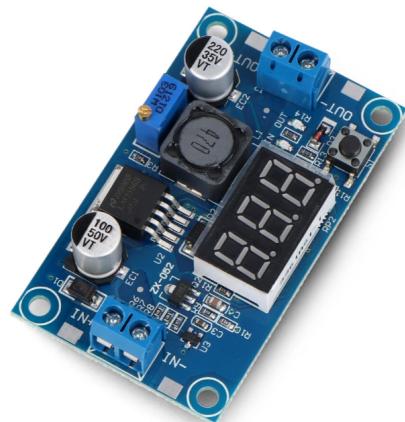
### 3.3.3 Zasilanie

- 6x akumulatory Li-ion 18650:
  - 4 ogniska (2S2P) dla silników
  - 2 ogniska (2S) dla elektroniki



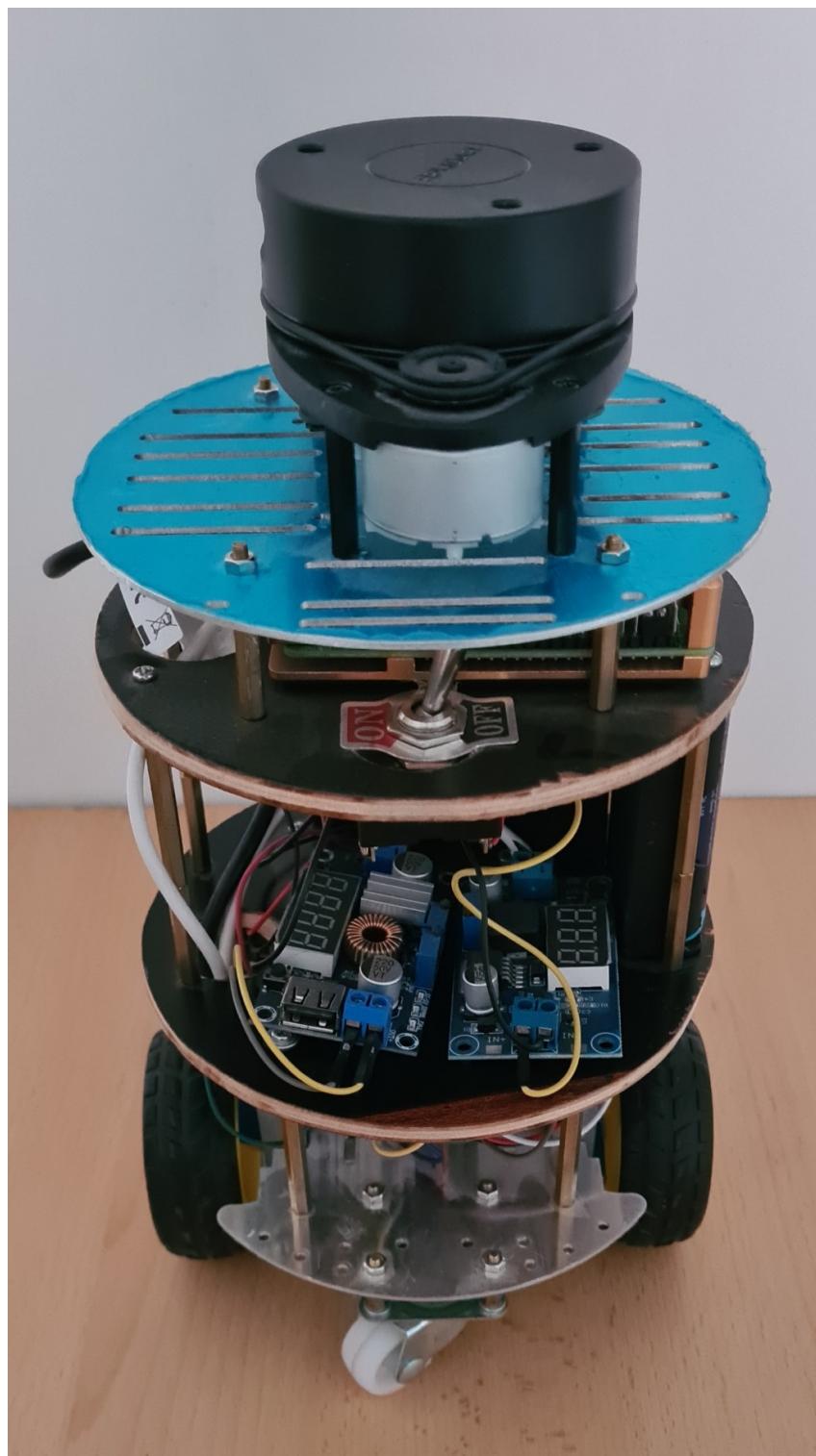
Rysunek 3.6: Ogniska 18650 Li-Ion XTAR - 2600mAh

- 2x przetwornica step-down:
  - 12V dla silników
  - 5V dla Raspberry Pi

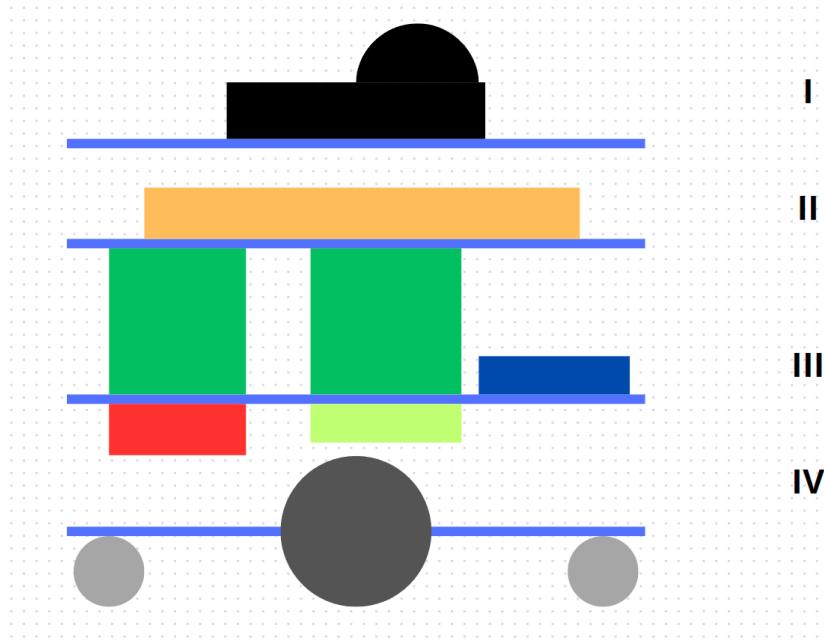


Rysunek 3.7: Przetwornica step-down LM2596 3,2V-35V 3A z wyświetlaczem

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



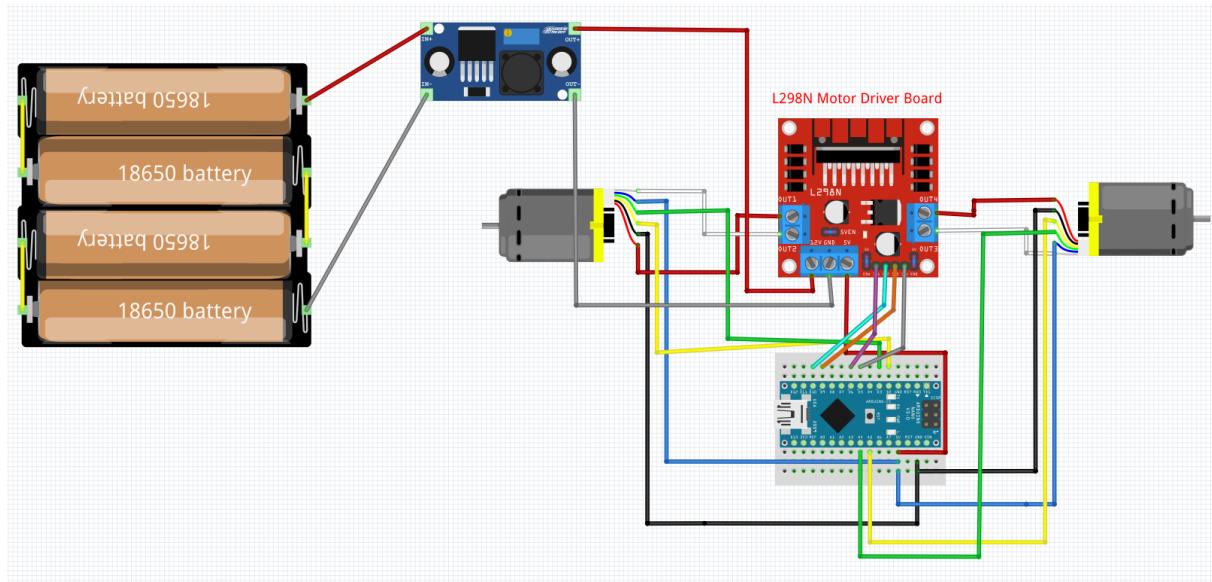
Rysunek 3.8: Zdjęcie przedstawiające zbudowanego robota



Rysunek 3.9: Uproszczony schemat przedstawiający budowę robota

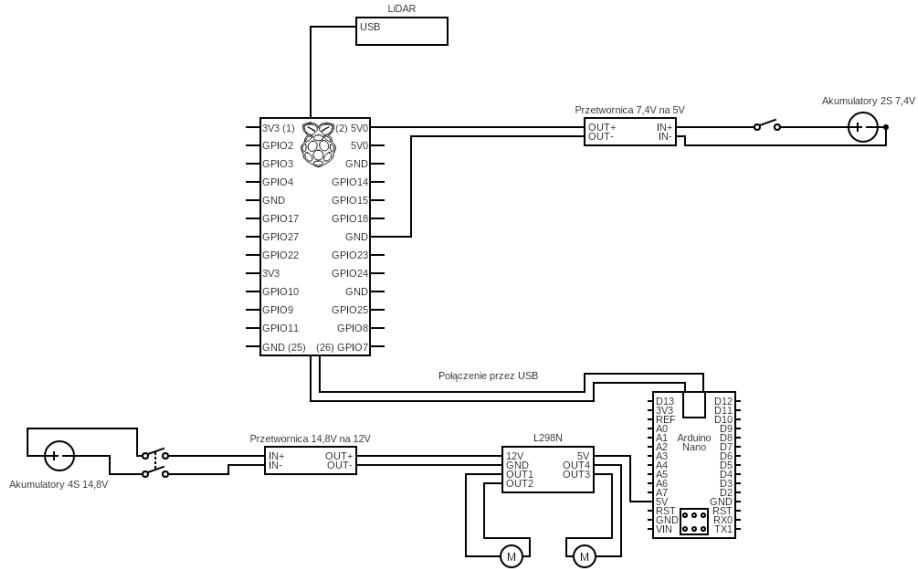
Na schemacie z rysunku 3.9 przedstawiono konkretnie poziomy robota, takie jak:

- 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 3.10: Schemat połączenia silników z Arduino i L298N

Na zaprezentowanym schemacie rys. 3.10 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 3.11: Schemat elektryczny

Na schemacie 3.11 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 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

W tej sekcji wyjaśnione zostaną poszczególne etapy jakie należy podjąć do poprawnego uruchomienia i korzystania z robota.

Na samym początku należy uruchomić Raspberry Pi, przez włączenie zasilania, oraz załączenie silników.

Następnie należy przygotować terminale na jednostce sterującej (dwa terminale mają być połączone przez protokół ssh z Raspberry Pi, a 5 terminali ma być przygotowanych na samej jednostce sterującej do uruchomienia późniejszych skryptów ROS). Przygotowanie tych terminali polega na odpowiednim:

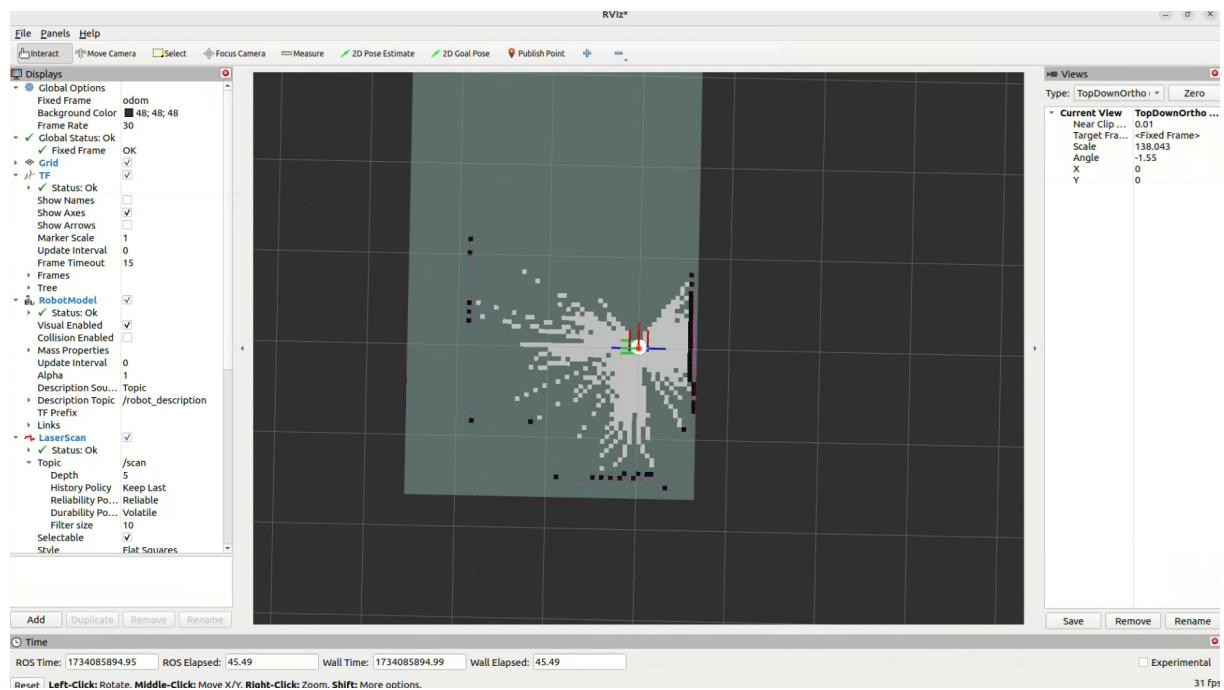
Dla jednostki sterującej: wejściu do katalogu roboczego, uruchomienie komend source /opt/ros/humble/setup.bash, oraz source /install/setup.bash.

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 przez ssh na Raspberry Pi, które odpowiadają za uruchomienie odpowiednich węzłów ROS. Pierwszy skrypt odpowiada za uruchomienie węzła odpowiedzialnego za sterowanie silnikami, 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.

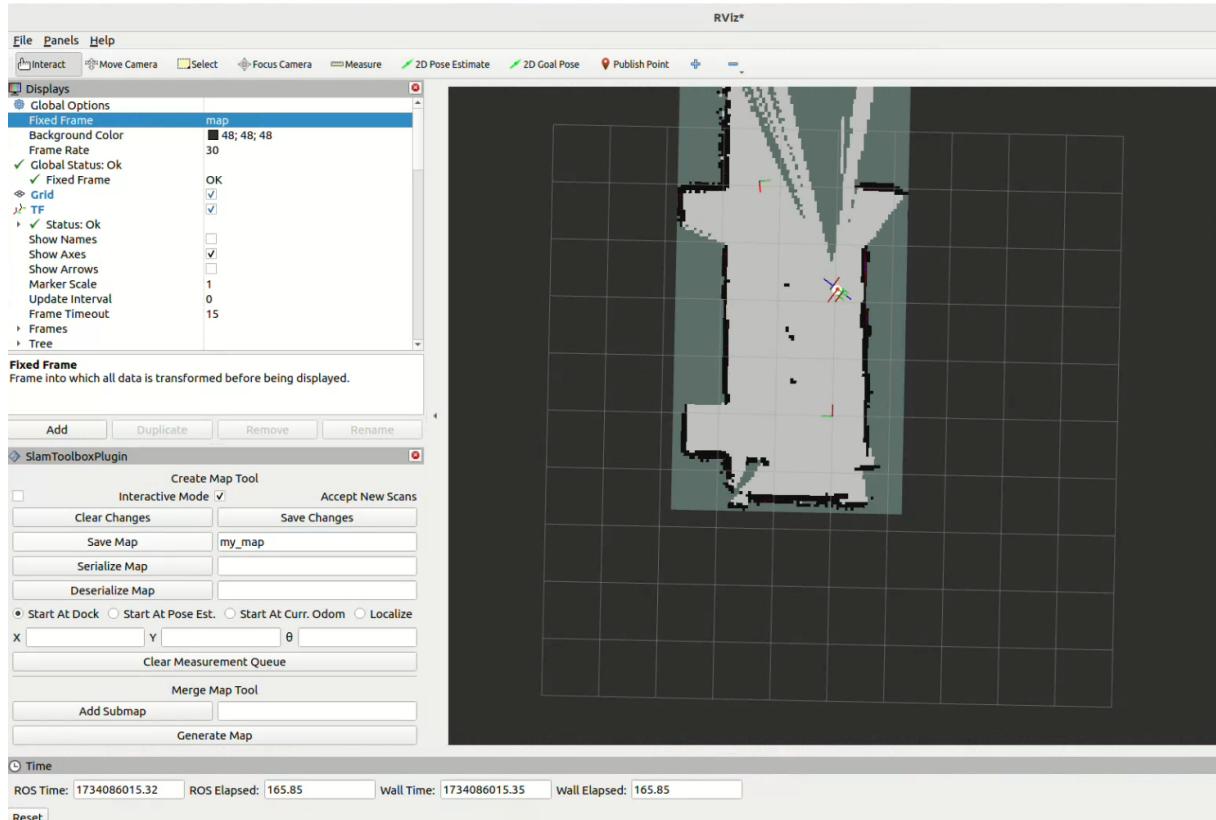
Po uruchomieniu tych skryptów, należy uruchomić skrypt odpowiedzialny za sterowanie robotem za pomocą klawiatury. Komenda to: ros2 launch robot\_teleop.launch.py. W tym momencie należy przejść do terminala na jednostce sterującej, i za pomocą klawiatury sterować robotem. Należy również uruchomić program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

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 zostaje zmapowane.



Rysunek 4.1: Wizualizacja w 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.



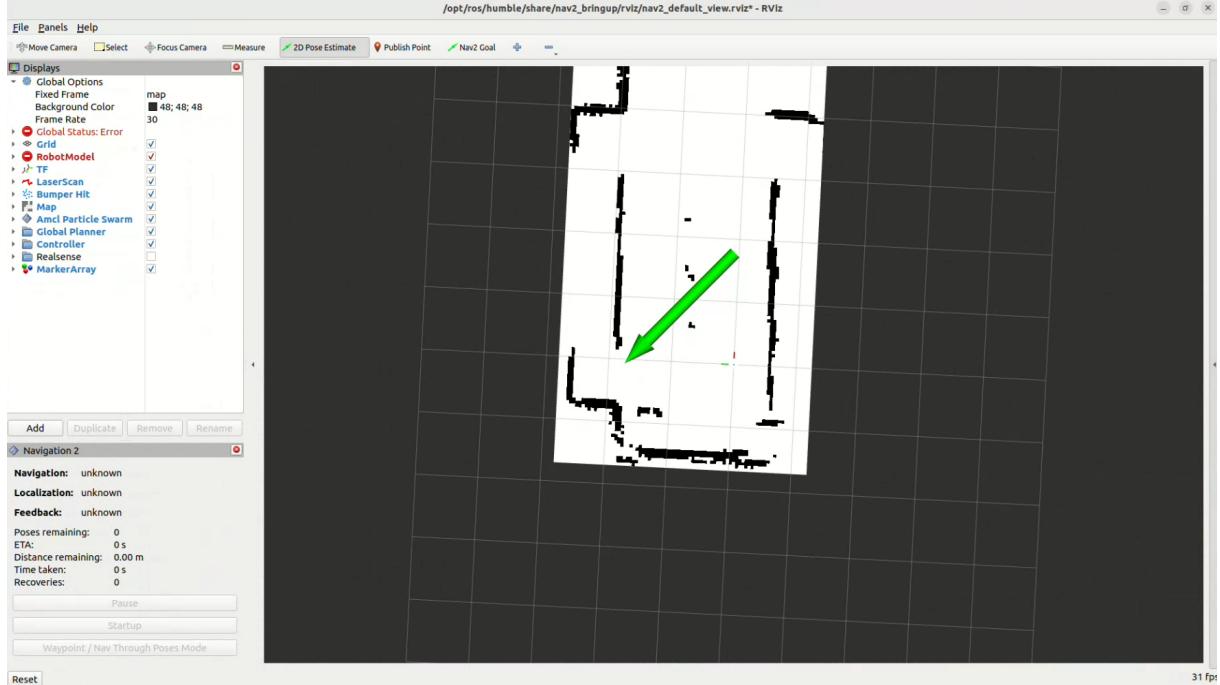
Rysunek 4.2: Wizualizacja zmapowanego pomieszczenia i zapisu mapy.

Z taką zapisaną mapą można zamknąć skrypty odpowiedzialne za sterowanie i mapowanie, jak i Rviz.

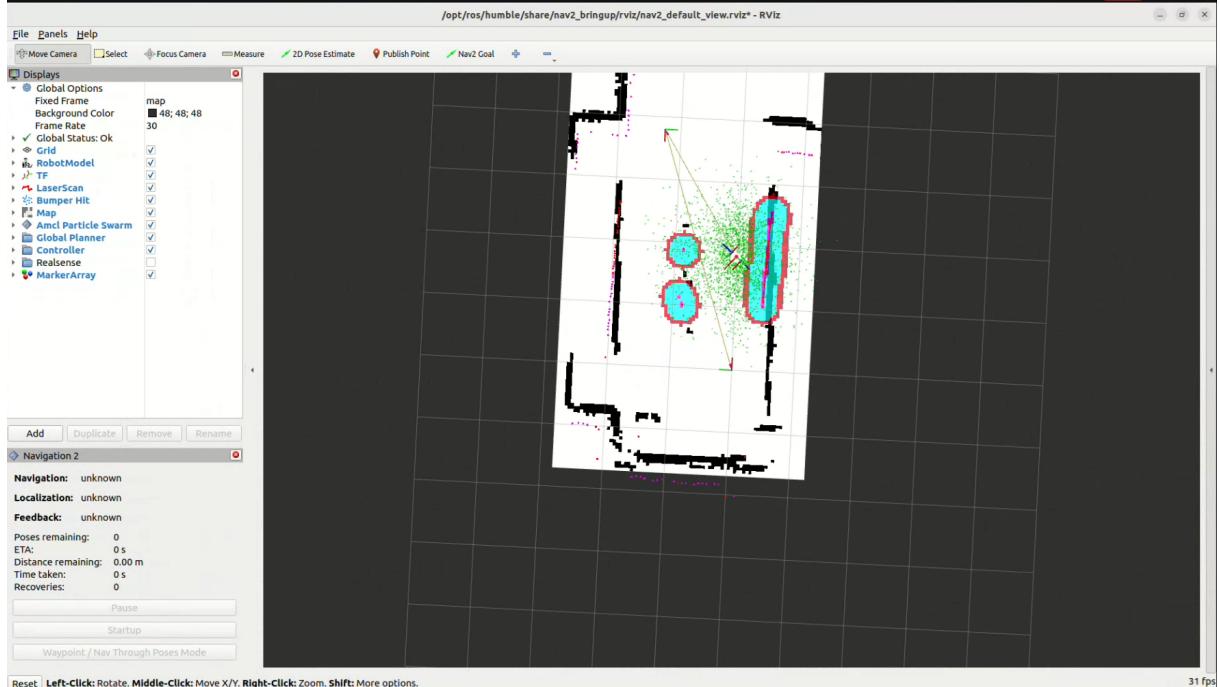
Następnie, należy uruchomić skrypt odpowiedzialny za lokalizację robota na mapie, oraz skrypt odpowiedzialny za nawigację robota.

Komendy to: `ros2 launch robot_slam localization_launch.py`,  
`ros2 launch robot_slam navigation_launch.py`. W wyniku uruchomienia tych skryptów powinien pojawić się nowy ekran Rviz, na którym będzie widoczna wcześniej zapisana mapa.

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. W tym momencie algorytm AMCL tworzy chmurę przewidywanych punktów w których może znajdować się robot.

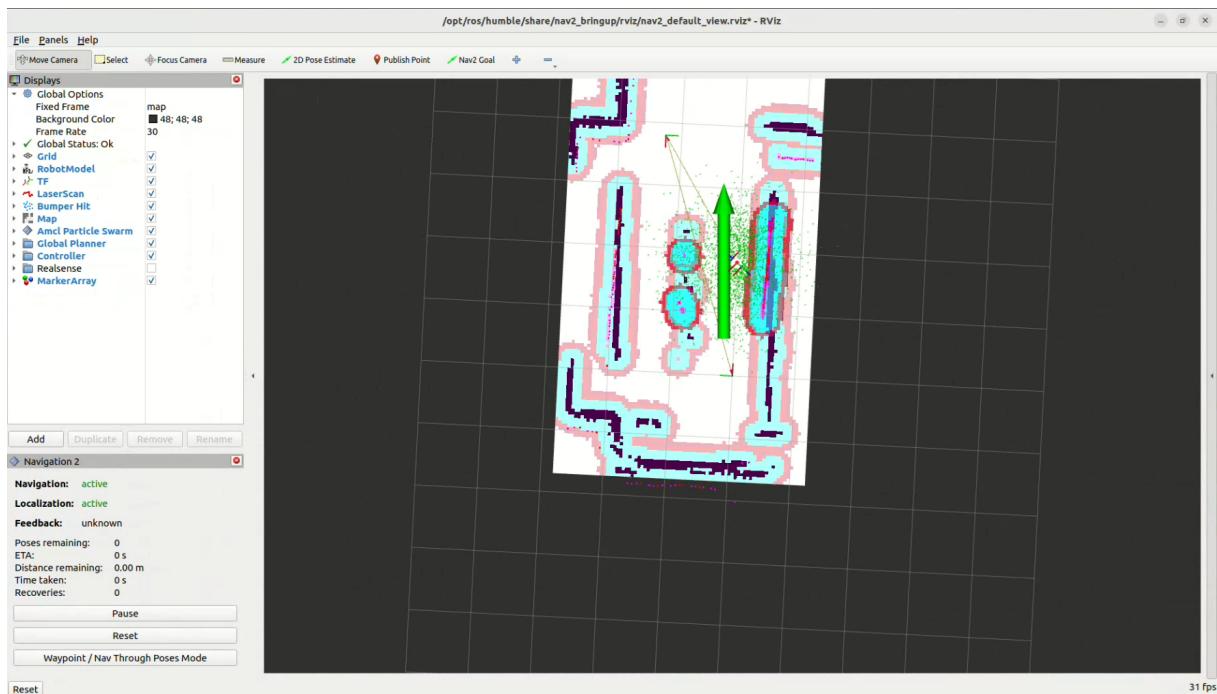


Rysunek 4.3: Wizualizacja po uruchomieniu skryptów do lokalizacji i nawigacji.



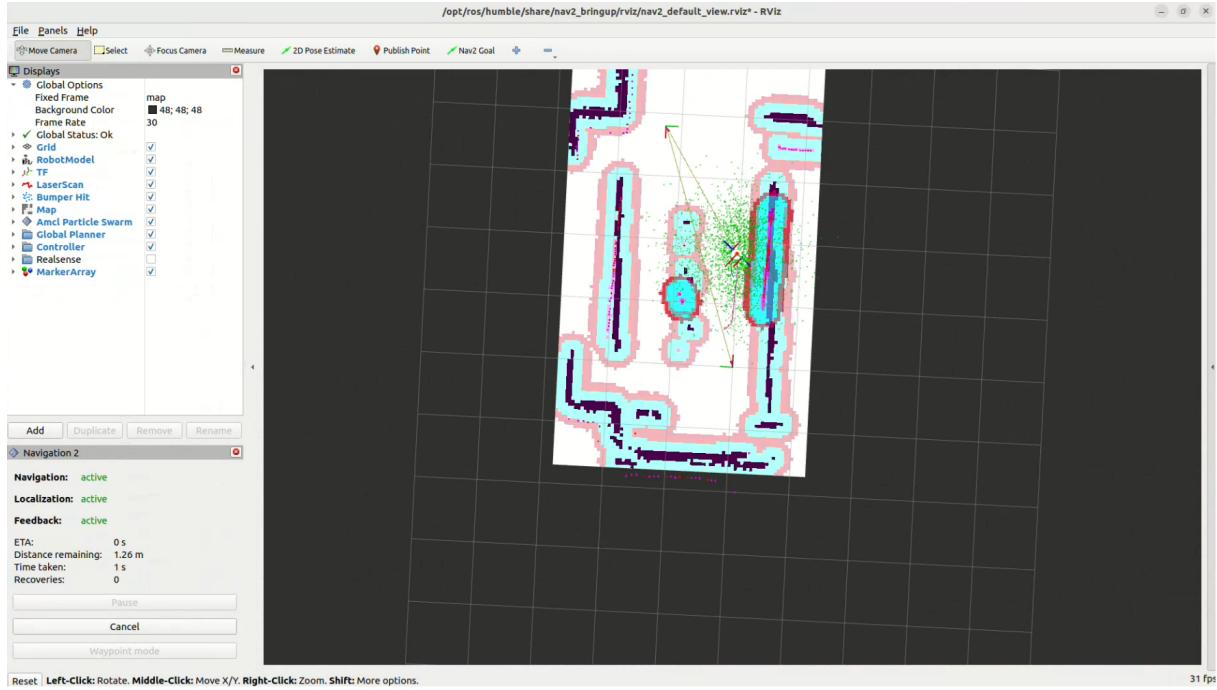
Rysunek 4.4: Wizualizacja rosproszonych punktów lokalizacji robota.

Następnie należy wybrać cel do którego robot ma się przemieścić, przez wybranie z górnego zestawu narzędzi 2D Goal Pose, a następnie kliknięcie na mapie w miejscu gdzie ma się znaleźć cel.

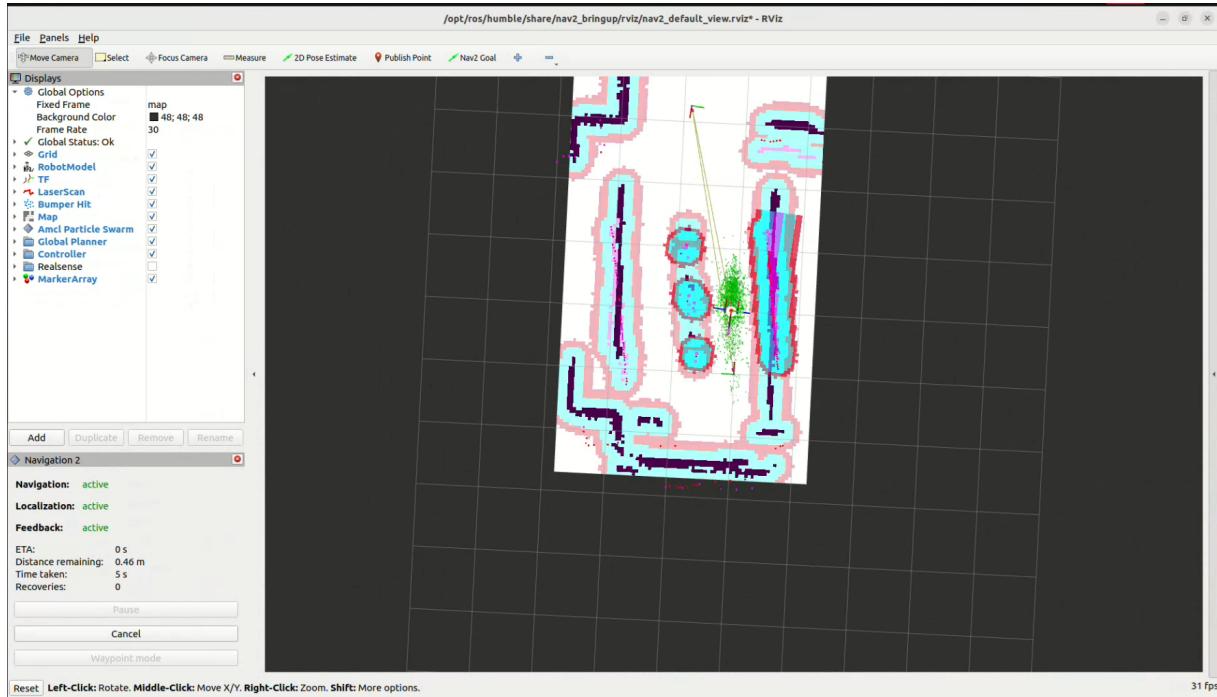


Rysunek 4.5: Wizualizacja po wybraniu celu do którego robot ma się przemieścić.

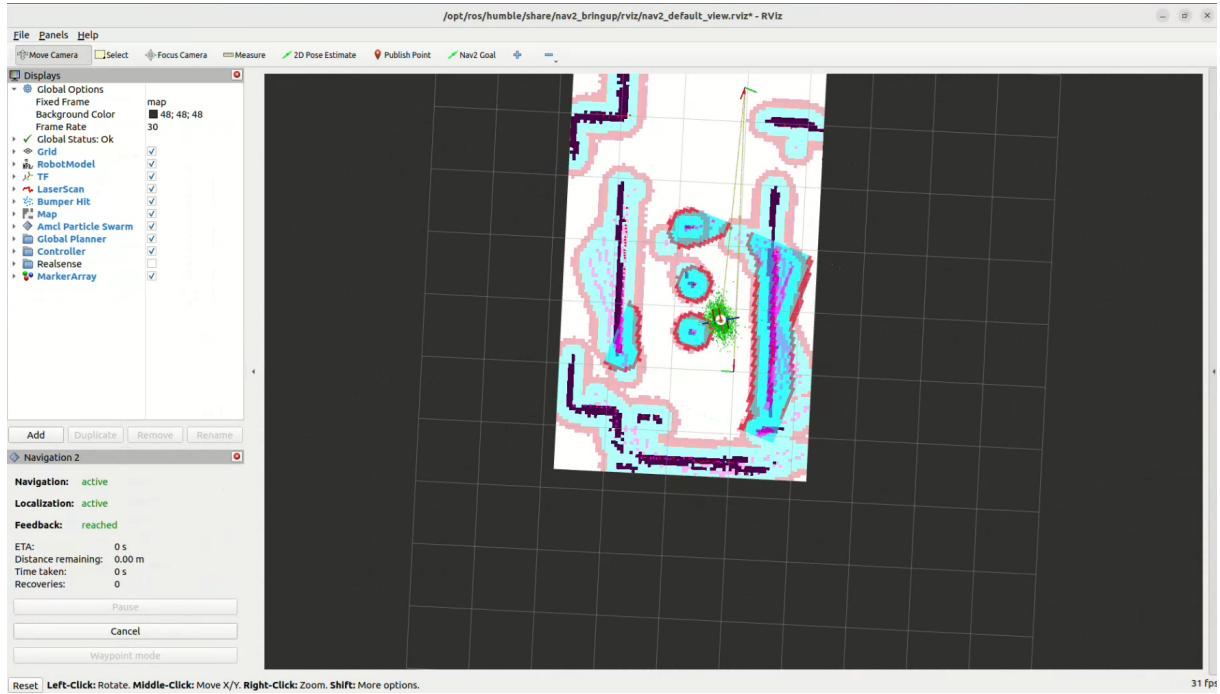
Robot powinien samodzielnie przemieścić się do tego celu, omijając przeszkody.



Rysunek 4.6: Wizualizacja wyznaczenia trasy.



Rysunek 4.7: Wizualizacja nawigacji robota i zmniejszania się chmury przewidywanej lokalizacji robota.



Rysunek 4.8: Wizualizacja po 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 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 zmapować pomieszczenia nawet do  $24\,000 \text{ m}^2$  co wynika z dokumentacji Nav2 [8].

# Rozdział 5

## Specyfikacja techniczna

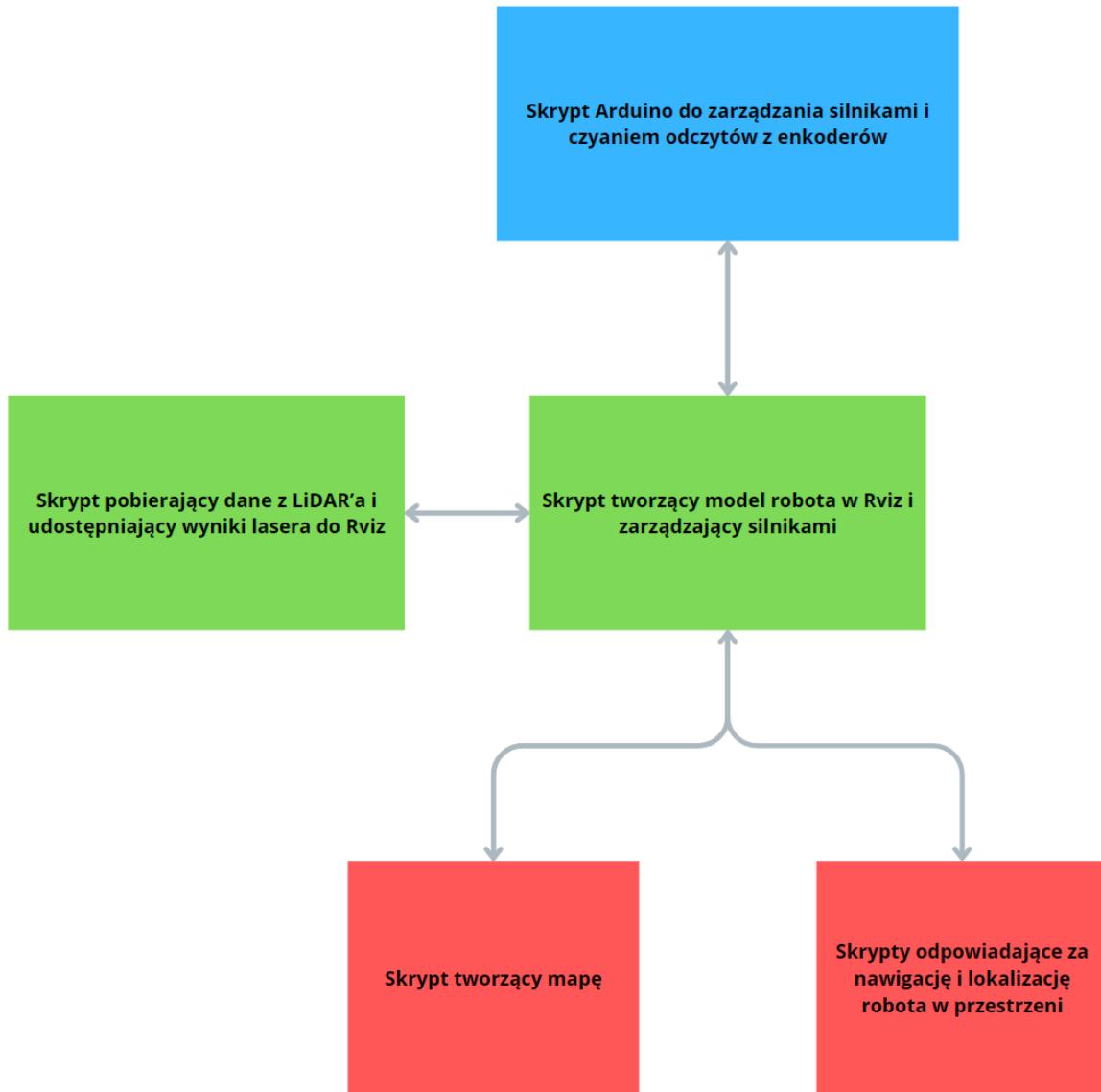
Rozdział ten zawiera specyfikację techniczną systemu, w tym opis wykorzystanych technologii, architekturę systemu, strukturę systemu, opis działania algorytmów oraz sposób połączenia skryptów w systemie.

### 5.1 Idea robota autonomicznego mapującego w czasie rzeczywistym

Robot autonomiczny mapujący w czasie rzeczywistym to robot mobilny, który samodzielnie przemieszcza się w przestrzeni, zbierając dane o otoczeniu i tworząc mapę otoczenia w czasie rzeczywistym. Robot ten wykorzystuje różnego rodzaju sensory, jak np. LiDAR, kamery, czy enkodery, do zbierania danych o otoczeniu. Dane te są przetwarzane przez algorytmy SLAM (Simultaneous Localization and Mapping), które pozwalają na określenie pozycji robota na mapie oraz na tworzenie mapy otoczenia. Robot ten może być wykorzystywany w różnego rodzaju zastosowaniach, jak np. inspekcja pomieszczeń, czy przemieszczanie się w trudno dostępnych miejscach.

## 5.2 Architektura systemu

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



Rysunek 5.1: Wizualizacja połączenia skryptów w systemie

## 5.3 Struktura systemu i objaśnienie działania algorytmów

W tej sekcji opisano poszczególne skrypty robota z rysunku 5.1

### 5.3.1 Skrypt Arduino

Skrypt Arduino jest odpowiedzialny za sterowanie silnikami i enkoderami. Odczytuje dane z enkoderów, oblicza prędkość i położenie robota, oraz steruje silnikami na podstawie danych o prędkościach otrzymanych od Raspberry Pi. Komunikuje się on z Raspberry Pi przez port szeregowy, co umożliwia przesyłanie danych o prędkości i położeniu robota. Skrypt ten jest zmodyfikowaną wersją skryptu `ros_arduino_bridge`[4], który został dostosowany do potrzeb tego projektu.

Najważniejsze funkcje tego programu znajdują się w plikach:

- ROSArduinoBridge.ino

Jest to główny program, gdzie znajdują się:

- Funkcja `setup()` - inicjalizująca kontroler silników, enkodery i piny wejścia/wyjścia
  - \* Odczytuje i przetwarza komendy z portu szeregowego
  - \* Wykonuje obliczenia PID dla kontroli silników
  - \* Aktualizuje stan enkoderów
  - \* Sprawdza warunki automatycznego zatrzymania
- Funkcja `loop()` - główna pętla programu, która:
  - \* Odczytuje enkoderów
  - \* Sterowanie silnikami
  - \* Reset enkoderów
  - \* Aktualizacja parametrów PID
  - \* Operacje na pinach cyfrowych i analogowych
- Zmienne kontrolujące:
  - \* Parametry komunikacji szeregowej
  - \* Timery kontrolne PID
  - \* Bufory komend i argumentów

- commands.h Zawiera definicje komend, które są wysyłane z Raspberry Pi do Arduino.
- 

```
1      #ifndef COMMANDS_H
2      #define COMMANDS_H
3
4      #define ANALOG_READ      'a'
5      #define GET_BAUDRATE    'b'
6      #define PIN_MODE        'c'
7      #define DIGITAL_READ    'd'
8      #define READ_ENCODERS   'e'
9      #define MOTOR_SPEEDS    'm'
10     #define MOTOR_RAW_PWM   'o'
11     #define PING            'p'
12     #define RESET_ENCODERS 'r'
13     #define SERVO_WRITE     's'
14     #define SERVO_READ      't'
15     #define UPDATE_PID      'u'
16     #define DIGITAL_WRITE   'w'
17     #define ANALOG_WRITE    'x'
18
19     #define LEFT           0
20     #define RIGHT          1
21
22     #endif
```

---

Rysunek 5.2: Komendy jakie są wysyłane z Raspberry Pi do Arduino

Plik zawiera definicje komend używanych do komunikacji między Raspberry Pi a Arduino. W tworzeniu projektu korzystano między innymi z komend e do dostosowania enkoderów silników i średnicy koła do rzeczywistych wartości.

- diff\_controller.h W tym pliku znajduje się implementacja regulatora PID (Proporcjonalno-Całkującąco-Różniczkującego) do sterowania silnikami. Jest to element krytyczny dla precyzyjnego sterowania robotem mobilnym.

Regulator wykorzystuje strukturę SetPointInfo do przechowywania danych:

- TargetTicksPerFrame - zadana prędkość w impulsach na ramkę
- Encoder - aktualna wartość enkodera
- PrevEnc - poprzednia wartość enkodera
- PrevInput - poprzednie wejście (zamiast poprzedniego błędu)
- ITerm - człon całkujący
- output - wyjście regulatora (sterowanie silnikiem)

Zdefiniowano następujące współczynniki regulatora:

- $K_p = 20$  - wzmacnienie członu proporcjonalnego
- $K_d = 12$  - wzmacnienie członu różniczkującego
- $K_i = 0$  - wzmacnienie członu całkującego
- $K_o = 50$  - współczynnik skalujący wyjście

Regulator zawiera trzy główne funkcje:

- resetPID() - inicjalizacja zmiennych regulatora
- doPID() - obliczanie nowego sygnału sterującego
- updatePID() - aktualizacja regulatorów obu silników

Algorytm działa poprzez obliczanie błędu między zadaną a aktualną prędkością, następnie wykorzystuje człony proporcjonalny, całkujący i różniczkujący do generowania sygnału sterującego silnikami.

```
1  /* PID routine to compute the next motor commands */
2  void doPID(SetPointInfo * p) {
3      long Perror;
4      long output;
5      int input;
6
7      input = p->Encoder - p->PrevEnc;
8      Perror = p->TargetTicksPerFrame - input;
9
10     output = (Kp * Perror - Kd * (input - p->PrevInput) + p->
11               ITerm) / Ko;
12     p->PrevEnc = p->Encoder;
13
14     output += p->output;
15     if (output >= MAX_PWM)
16         output = MAX_PWM;
17     else if (output <= -MAX_PWM)
18         output = -MAX_PWM;
19     else
20         p->ITerm += Ki * Perror;
21
22     p->output = output;
23     p->PrevInput = input;
24 }
```

---

Rysunek 5.3: Implementacja głównej funkcji regulatora PID

- encoder\_driver.h W tym pliku znajdują się definicje portów enkoderów i funkcje do ich obsługi.

```
1
2
3     #ifdef ARDUINO_ENC_COUNTER
4     #define LEFT_ENC_PIN_A PD2    //pin 2
5     #define LEFT_ENC_PIN_B PD3    //pin 3
6
7     #define RIGHT_ENC_PIN_A PC4   //pin A4
8     #define RIGHT_ENC_PIN_B PC5   //pin A5
9     #endif
10
11    long readEncoder(int i);
12    void resetEncoder(int i);
13    void resetEncoders();
```

Rysunek 5.4: Konfiguracja portów enkoderów z Arduino i sterownikiem L298N

- motor\_driver.h W tym pliku znajdują się definicje portów silników i funkcje do ich obsługi.
- 

```
1      #ifdef L298_MOTOR_DRIVER
2      #define RIGHT_MOTOR_BACKWARD 5
3      #define LEFT_MOTOR_BACKWARD 6
4      #define RIGHT_MOTOR_FORWARD 9
5      #define LEFT_MOTOR_FORWARD 10
6      #define RIGHT_MOTOR_ENABLE 12
7      #define LEFT_MOTOR_ENABLE 13
8      #endif
9
10
11     void initMotorController();
12     void setMotorSpeed(int i, int spd);
13     void setMotorSpeeds(int leftSpeed, int rightSpeed);
```

---

Rysunek 5.5: Konfiguracja portów silników

- motor\_driver.ino Przedstawiony kod zawiera definicje i implementacje dla różnych sterowników silnika.
  - **Struktura kodu:** Wykorzystuje dyrektywy preprocesora (`#ifdef`, `#elif`) do wyboru odpowiedniego sterownika silnika.
  - **Główne funkcje:**
    - \* `initMotorController()` – inicjalizuje sterownik silnika
    - \* `setMotorSpeed(int i, int spd)` – ustawia prędkość pojedynczego silnika
    - \* `setMotorSpeeds(int leftSpeed, int rightSpeed)` – ustawia prędkości obu silników
  - **Szczegóły implementacji L298:**
    - \* Obsługuje kierunek obrotów poprzez zmianę polaryzacji
    - \* Prędkość kontrolowana jest za pomocą PWM (0-255)
    - \* Wykorzystuje piny cyfrowe do sterowania kierunkiem i prędkością

Kod zapewnia jednolity interfejs dla różnych sterowników silnika, co ułatwia wymienność komponentów sprzętowych bez konieczności modyfikacji głównego kodu aplikacji.

- encoder\_driver.ino Ten plik zawiera logikę obsługi enkoderów. Główne komponenty to:
  - **Definicje enkoderów:** Plik wspiera różne typy enkoderów, wybierane przez dyrektywy preprocesora.
  - **Obsługa przerwań:** Zaimplementowano dwie procedury przerwań:
    - \* ISR (PCINT2\_vect) - dla lewego enkodera
    - \* ISR (PCINT1\_vect) - dla prawego enkodera
  - **Liczniki pozycji:**
    - \* left\_enc\_pos - pozycja lewego enkodera
    - \* right\_enc\_pos - pozycja prawego enkodera
  - **Główne funkcje:**
    - \* readEncoder() - odczyt pozycji enkodera
    - \* resetEncoder() - reset pojedynczego enkodera
    - \* resetEncoders() - reset obu enkoderów jednocześnie
  - **Tablica stanów:** ENC\_STATES przechowuje mapowanie stanów enkodera na kierunek obrotu.

Algorytm wykorzystuje technikę tablicy przeglądowej (lookup table) do dekodowania sygnałów z enkoderów, co pozwala na efektywne określanie kierunku i wielkości obrotu kół robota. System jest zoptymalizowany pod kątem szybkiej odpowiedzi na przerwania sprzętowe.

### 5.3.2 Skrypt do obsługi LiDAR-a

Do projektu wykorzystano zestaw narzędzi udostępnionych przez firmę Slamtec, które pozwalają na odczyt danych z LiDAR-a, jak i na tworzenie mapy otoczenia. Skrypt ten jest odpowiedzialny również za udostępnienie danych z LiDAR-a z pliku rplidar.launch.py, plik ten tworzy węzeł z pliku konfiguracyjnego rplidar.yaml, gdzie znajduje się konfiguracja LiDAR-a.

Ten kod reprezentuje plik konfiguracyjny systemu ROS2 (Robot Operating System 2) służący do uruchomienia sterownika dla skanera laserowego RPLidar. Poniżej przedstawiono szczegółową analizę jego działania.

- **Struktura pliku:**

- Import niezbędnych modułów z ROS2 Launch System
- Definicja funkcji generate\_launch\_description()
- Konfiguracja parametrów dla węzła RPLidar

- **Kluczowe elementy:**

- Sprawdzenie dostępności portu szeregowego
- Deklaracja argumentu serial\_port
- Konfiguracja węzła rplidar\_node

- **Parametry LiDAR-a:**

- Prędkość transmisji: 115200 baud
- Częstotliwość publikowania: 2.5 Hz
- Rozmiar kolejki: 500
- Minimalny zasięg: 0.15m
- Maksymalny zasięg: 12.0m
- Zakres kątowy: -pi do pi (-3.14159 do 3.14159)

- **Optymalizacje:**

- Zredukowana częstotliwość publikowania z 5.0 do 2.5 Hz
- Zmniejszony rozmiar kolejki z 1000 do 500
- Włączona kompensacja kątowa
- Włączona filtracja próbek

Skrypt ten jest odpowiedzialny za prawidłową inicjalizację i konfigurację skanera laserowego RPLidar, który jest kluczowym elementem w systemie mapowania i nawigacji robota.

### 5.3.3 Skrypt do obsługi silników i tworzenia modelu robota

Skrypt do obsługi silników i tworzenia modelu robota jest odpowiedzialny za sterowanie silnikami i tworzenie modelu robota. Skrypt ten odczytuje dane o prędkościach robota z klawiatury, przetwarza je i przesyła do Arduino. Skrypt ten jest odpowiedzialny również za udostępnienie modelu robota z pliku `rsp.launch.py`, plik ten tworzy węzeł z pliku konfiguracyjnego `model_main.xacro`, gdzie znajduje się model robota w formacie URDF.

Najważniejsze funkcje tego programu znajdują się w plikach:

- `model_controll.launch.py` Jest to główny plik wykonawczy, który spaja wszystkie funkcje w całość do użytku robota z ROS. Ten plik jest odpowiedzialny za uruchomienie wszystkich potrzebnych węzłów ROS 2 do obsługi robota:
  - RSP (Robot State Publisher) - publikuje stan robota w przestrzeni TF2
  - Twist Mux - multiplekser wiadomości twist do sterowania robotem
  - Controller Manager - zarządza kontrolerami robota:
    - \* Differential Drive Controller - kontroler napędu różnicowego
    - \* Joint State Broadcaster - publikuje stan przegubów robota

Skrypt wykorzystuje:

- Parametry w plikach konfiguracyjnych:
  - \* `twist_mux.yaml` - konfiguracja multipleksera
  - \* `my_controllers.yaml` - konfiguracja kontrolerów
  - \* `robot_description` - opis robota w formacie URDF
- Zdarzenia i timery do:
  - \* Opóźnionego uruchamiania kontrolerów
  - \* Uruchamiania węzłów w odpowiedniej kolejności

Po uruchomieniu skryptu robot jest gotowy do:

- Odbierania poleceń ruchu (`cmd_vel`)
- Publikowania swojego stanu w systemie TF2
- Kontrolowania silników przez interfejs ROS2 Control

- `rsp.launch.py` Jest to plik wykonawczy, który tworzy węzeł z pliku konfiguracyjnego `model_main.xacro`, gdzie znajduje się model robota w formacie URDF. Ten plik uruchamia proces tworzenia modelu robota w ROS 2. Oto główne komponenty:

- Parametry konfiguracyjne:**

- \* `use_sim_time` - kontroluje użycie czasu symulacji
  - \* `use_ross2_control` - włącza/wyłącza system ROS2 Control

- Przetwarzanie pliku URDF:**

- \* Lokalizacja pliku XACRO w pakiecie `robot_slam`
  - \* Konwersja XACRO do URDF z parametrami symulacji

- Węzeł `robot_state_publisher`:**

- \* Publikuje stan robota w systemie TF2
  - \* Wykorzystuje wygenerowany opis URDF
  - \* Konfigurowany przez parametry symulacji

Skrypt tworzy strukturę LaunchDescription, która uruchamia system w odpowiedniej konfiguracji, zapewniając publikowanie stanu robota w systemie ROS 2.

- model\_main.xacro Plik ten łączy w jeden plik xacro model robota i plik xacro do ros2\_control(odpowiedzialne za parametry silników i ustawienia dla paczki diff\_drive\_arduino). Ten plik URDF (Unified Robot Description Format) opisuje strukturę fizyczną robota mobilnego. Zawiera on następujące główne elementy:

– **Definicje materiałów:**

- \* Biały - dla głównego korpusu
- \* Pomarańczowy - dla LiDAR-a
- \* Niebieski - dla kół
- \* Czarny - dla kół podporowych

– **Struktura robota:**

- \* base\_link - podstawowy punkt odniesienia robota
- \* base\_footprint - punkt odniesienia na poziomie podłoża
- \* container - główny korpus robota (cylinder o promieniu 8.5cm i wysokości 22.5cm)
- \* lidar - czujnik LiDAR (cylinder o promieniu 3.5cm i wysokości 5.8cm)
- \* left\_wheel i right\_wheel - koła napędowe (cylindry o promieniu 3.5cm i szerokości 2.7cm)
- \* support\_wheels - przednie i tylne koła podporowe (sfery o promieniu 1.8cm)

– **Połączenia (joints):**

- \* Stałe (fixed) dla korpusu, LiDAR-a i kół podporowych
- \* Ciągłe (continuous) dla kół napędowych, umożliwiające nieskończony obrót

– **Właściwości fizyczne:**

- \* Zdefiniowane masy dla wszystkich elementów
- \* Momenty bezwładności dla kół i elementów obrotowych
- \* Punkty kolizji dla wykrywania zderzeń

Model uwzględnia wszystkie niezbędne elementy do symulacji fizycznej i wizualizacji robota w środowisku ROS 2.

- model.urdf.xacro Znajduje się tutaj model robota w formacie URDF. Ten plik XACRO (XML Macros) definiuje konfigurację robota dla systemu ROS2. Zawiera następujące główne elementy:

- **Deklaracja robota:**

- \* Nazwa: robot\_slam
    - \* Używa przestrzeni nazw xacro dla makr XML

- **Argumenty konfiguracyjne:**

- \* use\_ros2\_control - kontroluje użycie systemu ROS2 Control (domyślnie true)
    - \* sim\_mode - określa tryb symulacji (domyślnie false)

- **Dołączzone pliki:**

- \* model\_main.xacro - zawiera główny opis modelu robota
    - \* ros2\_control.xacro - zawiera konfigurację systemu ROS2 Control

- **Konfiguracja ROS2 Control:**

- \* Wykorzystuje zdefiniowane wcześniej argumenty
    - \* Pozwala na elastyczne przełączanie między trybem rzeczywistym a symulacją

Plik ten służy jako główny punkt wejścia dla opisu robota, łącząc model fizyczny z konfiguracją kontrolerów.

- my\_controllers.yaml

Ten plik zawiera konfigurację kontrolera ROS2 Control dla robota różnicowego.

System zarządzania kontrolerami działa z częstotliwością 30 Hz i obsługuje dwa główne komponenty: kontroler napędu różnicowego (diff\_cont) oraz nadajnik stanu przegubów (joint\_broad).

Kontroler napędu różnicowego pracuje z częstotliwością 50 Hz i wykorzystuje ramkę bazową base\_footprint. Parametry fizyczne robota obejmują rozstaw kół 0.14 m oraz promień koła 0.035 m, z oddzielną konfiguracją dla każdego koła.

Odometria wykorzystuje ramkę "odom" i zawiera macierze kowariancji dla pozycji oraz prędkości, z włączonym publikowaniem transformacji TF.

Limity ruchu zdefiniowane są następująco:

- Ruch liniowy (oś x): prędkość  $\pm 0.3$  m/s, przyspieszenie  $\pm 0.5$  m/s<sup>2</sup>, zryw 1.0 m/s<sup>3</sup>
- Ruch kątowy (oś z): prędkość  $\pm 1.0$  rad/s, przyspieszenie  $\pm 0.8$  rad/s<sup>2</sup>, zryw 1.5 rad/s<sup>3</sup>

System zawiera również sprzężenie zwrotne pozycji z krokiem czasowym 0.1s oraz wygładzanie prędkości ze stałą czasową 0.1s.

Ta konfiguracja zapewnia precyzyjną kontrolę ruchu robota z uwzględnieniem ograniczeń fizycznych i bezpieczeństwa.

- twist\_mux.yaml Opisane zostały tutaj parametry do obsługi paczki twist\_mux. Ten plik konfiguracyjny zawiera:

- **Parametry multipleksera:**

- \* Temat wejściowy: cmd\_vel (temat do sterowania robotem)
    - \* Timeout: 0.5s (maksymalny czas oczekiwania na nowe polecenia)
    - \* Priorytet: 10 (priorytet poleceń nawigacyjnych)

Te ustawienia pozwalają na zarządzanie wieloma źródłami poleceń ruchu dla robota, gdzie polecenia nawigacyjne mają wysoki priorytet i są automatycznie przerywane po przekroczeniu limitu czasowego.

### 5.3.4 Skrypt do tworzenia mapy

Skrypt do tworzenia mapy jest odpowiedzialny za tworzenie mapy otoczenia. Skrypt ten odczytuje dane z LiDAR-a, przetwarza je i tworzy mapę otoczenia. Skrypt ten jest odpowiedzialny również za udostępnienie mapy z pliku map\_server.launch.py, plik ten tworzy węzeł z pliku konfiguracyjnego map.yaml, gdzie znajduje się zapisana mapa otoczenia.

Pliki odpowiedzialne za działanie tego skryptu to:

- slam.launch.py Jest to główny plik wykonawczy, który spaja wszystkie funkcje w całość do użytku robota z ROS. Ten plik uruchamia proces mapowania SLAM w ROS 2. Oto główne komponenty:

- **Konfiguracja parametrów:**

- \* use\_sim\_time - kontroluje użycie czasu symulacji
    - \* mapper\_params\_online\_async.yaml - plik z parametrami SLAM Toolbox

- **Węzeł SLAM Toolbox:**

- \* Typ: async\_slam\_toolbox\_node
    - \* Nazwa: slam\_toolbox
    - \* Wyjście: wyświetlane na ekranie
    - \* Korzysta z asynchronicznego trybu mapowania

- **Proces uruchomienia:**

- \* Deklaracja argumentu use\_sim\_time
    - \* Utworzenie węzła SLAM Toolbox z odpowiednimi parametrami
    - \* Zwrócenie struktury LaunchDescription z zadeklarowanymi komponentami

Skrypt konfiguruje i uruchamia system SLAM w trybie asynchronicznym, co pozwala na wydajne tworzenie mapy w czasie rzeczywistym.

- `mapper_params_online_async.yaml` Opisane zostały tutaj parametry do obsługi paczki SLAM Toolbox. Ten plik konfiguracyjny zawiera parametry dla SLAM Toolbox. Główne sekcje to:

- **Parametry wtyczki Ceres Solver:**

- \* Typ solvera: SPARSE\_NORMAL\_CHOLESKY
  - \* Strategia optymalizacji: LEVENBERG\_MARQUARDT
  - \* Funkcja straty: HUBERLOSS z delta 2.0

- **Parametry ROS:**

- \* Układ współrzędnych: odom, map, base\_footprint
  - \* Temat skanowania: /scan
  - \* Tryb: mapowanie
  - \* Częstotliwość publikacji transformacji: 0.05s
  - \* Interwał aktualizacji mapy: 2.0s
  - \* Rozdzielcość: 0.05m

- **Parametry ogólne:**

- \* Minimalna odległość przejazdu: 0.05m
  - \* Minimalny kąt obrotu: 0.1rad
  - \* Rozmiar bufora skanów: 10
  - \* Maksymalny zasięg skanowania: 12.0m

- **Parametry dopasowania pętli:**

- \* Maksymalna odległość wyszukiwania: 3.0m
  - \* Minimalna wielkość łańcucha: 10
  - \* Maksymalna wariancja: 3.0
  - \* Minimalna odpowiedź: 0.45

- **Parametry korelacji:**

- \* Wymiar przestrzeni wyszukiwania: 0.5m
  - \* Rozdzielcość: 0.01m
  - \* Odchylenie rozmycia: 0.1m

- **Filtры предкоści:**

- \* Rozmiar bufora: 10
  - \* Próg predkości: 0.1m/s

Parametry te zostały zoptymalizowane pod kątem wydajności i dokładności mapowania w czasie rzeczywistym dla robota mobilnego.

### 5.3.5 Skrypty do nawigacji i lokalizacji

Skrypty do nawigacji i lokalizacji są odpowiedzialne za lokalizację robota na mapie oraz za nawigację robota. Skrypty te odczytują dane z mapy i z LiDAR-a, przetwarzają je i określają pozycję robota na mapie oraz planują trasę do wyznaczonego punktu. Skrypty te są odpowiedzialne również za sterowanie robotem, tak aby omijał przeszkody na swojej drodze.

Funkcjonalność nawigacji i lokalizacji została zaimplementowana przy użyciu pakietów Nav2 i AMCL, które pozwalają na lokalizację robota na mapie oraz na nawigację robota do wyznaczonego punktu. Podzielone zostały na dwa pliki wykonawcze:

- localization\_launch.py Jest to główny plik wykonawczy, który spaja wszystkie funkcje w całość do użytku robota z ROS. Ten plik uruchamia system lokalizacji w ROS 2. Główne komponenty to:
  - **Parametry konfiguracyjne:**
    - \* namespace - przestrzeń nazw dla węzłów
    - \* map - ścieżka do pliku z mapą
    - \* use\_sim\_time - kontrola użycia czasu symulacji
    - \* autostart - automatyczne uruchamianie systemu
  - **Węzły:**
    - \* map\_server - odpowiada za wczytanie i udostępnienie mapy
    - \* amcl - algorytm lokalizacji Monte Carlo
    - \* lifecycle\_manager - zarządza cyklem życia węzłów
    - \* rviz2 - wizualizacja w czasie rzeczywistym
  - **Parametry AMCL:**
    - \* Filtr cząsteczkowy: 3000-10000 cząstek
    - \* Model ruchu: współczynniki alpha1-5 = 0.2
    - \* Aktualizacja: min\_d = 0.1m, min\_a = 0.1rad
    - \* Odzyskiwanie: alpha\_slow = 0.001, alpha\_fast = 0.1
  - **Proces uruchomienia:**
    - \* Wczytanie parametrów z plików konfiguracyjnych
    - \* Uruchomienie węzłów w odpowiedniej kolejności
    - \* Konfiguracja transformacji (tf) i remapowania

System zapewnia lokalizację robota na wcześniej stworzonej mapie poprzez porównywanie aktualnych odczytów z czujników z danymi z mapy.

- `navigation_launch.py` Jest to główny plik wykonawczy, który spaja wszystkie funkcje w całość do użytku robota z ROS. Ten plik uruchamia system nawigacji w ROS 2. Główne komponenty to:

– **Parametry konfiguracyjne:**

- \* namespace - przestrzeń nazw dla węzłów
- \* use\_sim\_time - kontrola użycia czasu symulacji
- \* params\_file - ścieżka do pliku parametrów
- \* autostart - automatyczne uruchamianie systemu
- \* use\_composition - użycie kompozycji węzłów
- \* use\_respawn - ponowne uruchamianie węzłów po awarii

– **Główne węzły:**

- \* controller\_server - kontroler ruchu robota
- \* planner\_server - planowanie ścieżki
- \* behavior\_server - zarządzanie zachowaniami
- \* bt\_navigator - nawigacja oparta o drzewa zachowań
- \* waypoint\_follower - podążanie za punktami
- \* velocity\_smoothen - wygładzanie prędkości
- \* lifecycle\_manager - zarządzanie cyklem życia węzłów

– **Proces uruchomienia:**

- \* Deklaracja parametrów konfiguracyjnych
- \* Konfiguracja przekierowania tematów ROS
- \* Uruchomienie węzłów nawigacyjnych
- \* Zarządzanie cyklem życia węzłów

System zapewnia kompleksową nawigację autonomiczną, od planowania ścieżki po kontrolę ruchu robota, z możliwością dostosowania parametrów i zachowań. Plik ten posiada również plik konfiguracyjny `nav2_params.yaml`, gdzie znajdują się parametry do nawigacji robota:

– **AMCL (Adaptive Monte Carlo Localization):**

- \* Parametry modelu ruchu (`alpha1-5: 0.2`)
- \* Parametry przetwarzania skanów laserowych
- \* Parametry filtra cząsteczkowego (`500-2000 cząstek`)

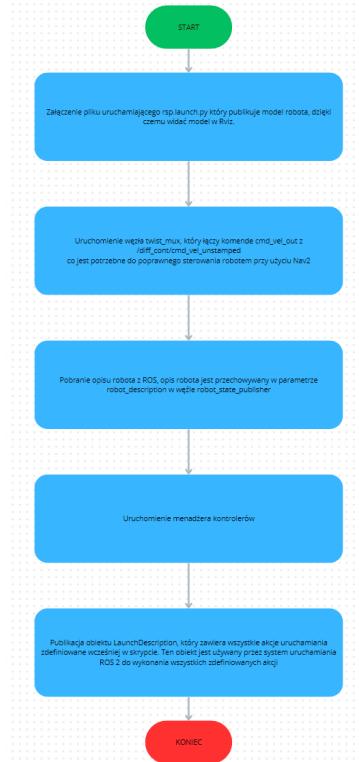
– **Nawigator BT (Behavior Tree):**

- \* Konfiguracja ramek odniesienia

- \* Lista dostępnych wtyczek do drzew zachowań
- \* Parametry kontroli wykonania
- **Kontroler ruchu:**
  - \* Częstotliwość kontroli: 20.0 Hz
  - \* Progi prędkości minimalnej
  - \* Parametry kontrolera DWB (Dynamic Window Approach)
- **Mapy kosztów:**
  - \* Lokalna: aktualizacja 5.0 Hz, okno 3x3m
  - \* Globalna: aktualizacja 1.0 Hz, rozdzielcość 0.05m
  - \* Konfiguracja warstw: przeszkody, inflacja, statyczna
- **Planowanie ścieżki:**
  - \* Algorytm NavfnPlanner
  - \* Tolerancja celu: 0.5m
  - \* Parametry optymalizacji
- **Zachowania robota:**
  - \* Obrót, cofanie, jazda po kierunku
  - \* Parametry asystenta teleoperacji
  - \* Limity prędkości i przyspieszenia
- **Wygładzanie prędkości:**
  - \* Częstotliwość: 20.0 Hz
  - \* Maksymalne prędkości: [0.26, 0.0, 1.0] m/s
  - \* Maksymalne przyspieszenia: [2.5, 0.0, 3.2] m/s<sup>2</sup>

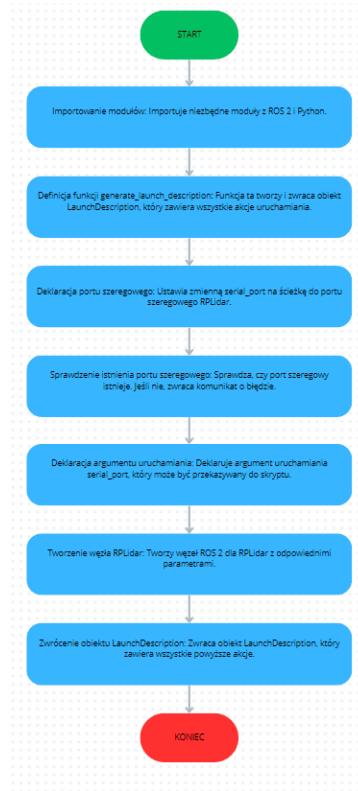
Te parametry zostały zoptymalizowane dla zapewnienia stabilnej i bezpiecznej nawigacji robota w różnych warunkach.

### 5.3.6 Diagramy UML prezentujące działanie konkretnych programów

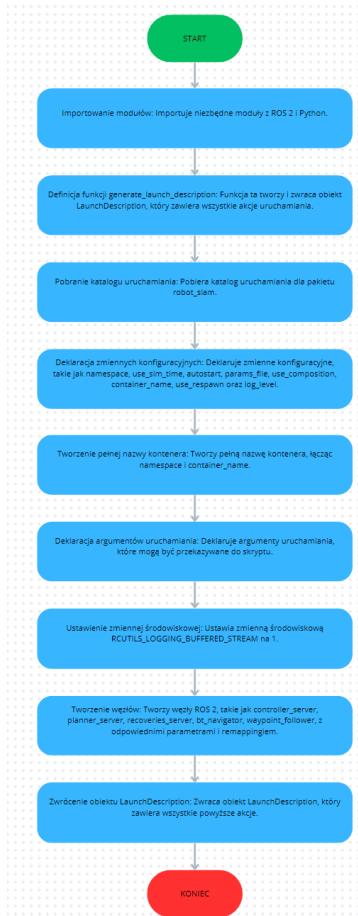


Rysunek 5.6: Diagram UML prezentujący działanie programu model\_controll.launch.py

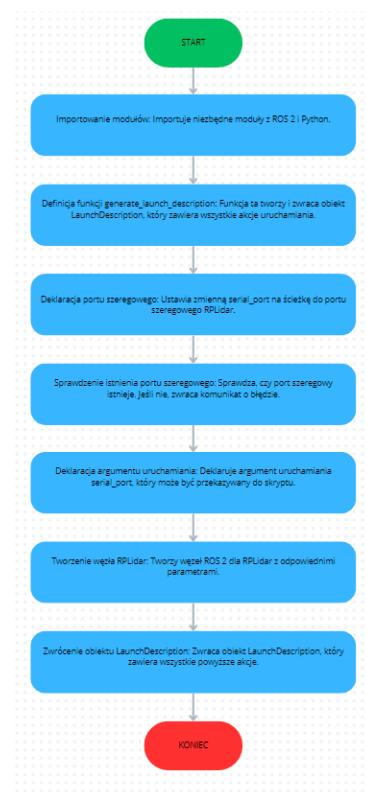
Krótka wstawka kodu w linii tekstu jest możliwa, np. **int a;** (biblioteka `listings`). Dłuższe fragmenty lepiej jest umieszczać jako rysunek, np. kod na rys ??, a naprawdę długie fragmenty – w załączniku.



Rysunek 5.7: Diagram UML prezentujący działanie programu rplidar.launch.py



Rysunek 5.8: Diagram UML prezentujący działanie programu navigation\_launch.py



Rysunek 5.9: Diagram UML prezentujący działanie programu `localization_launch.py`

# Rozdział 6

## Weryfikacja i walidacja

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

### 6.1 Model V

Model V składa się z następujących etapów:

- **Definiowanie wymagań:** Określenie wymagań funkcjonalnych i niefunkcjonalnych systemu.
- **Projektowanie systemu:** Opracowanie architektury systemu oraz szczegółowego projektu poszczególnych komponentów.
- **Implementacja:** Kodowanie poszczególnych modułów systemu.
- **Testowanie jednostkowe:** Testowanie poszczególnych modułów w celu weryfikacji ich poprawności.
- **Integracja i testowanie integracyjne:** Łączenie modułów i testowanie ich współprzedziałań.
- **Testowanie systemowe:** Testowanie całego systemu w celu weryfikacji spełnienia wymagań.
- **Walidacja:** Sprawdzenie, czy system spełnia oczekiwania użytkownika i działa zgodnie z założeniami.

## 6.2 Organizacja eksperymentów

Eksperymenty przeprowadzono zgodnie z model V, realizując następujące kroki:

- **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.
- **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 silników, które zostały rozwiązane przez zastosowanie odpowiednich przetwornic step-down.
- **Problem z kółkami:** Kółka robota miały tendencję do ślizgania się, co zostało rozwiązane przez zastosowanie gumowych opon.
- **Problem z błędna nawigacją:** Robot czasami poruszał się w nieoczekiwany sposób, co zostało rozwiązane przez poprawne podłączenie silników i kalibrację enkoderów.

## 6.5 Wyniki badań eksperymentalnych

Wyniki testów potwierdziły poprawność działania systemu. Robot poprawnie reagował na polecenia z klawiatury, tworzył mapę otoczenia, lokalizował się na zapisanej mapie oraz nawigował do wyznaczonych punktów, omijając przeszkody. System spełniał wszystkie wymagania funkcjonalne i niefunkcjonalne określone na początku projektu.



# 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. System spełnia wszystkie wymagania funkcjonalne i niefunkcjonalne określone na początku projektu.

### 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.
- Implementacja zaawansowanych algorytmów planowania ścieżki, które uwzględniają dynamiczne przeszkody.
- Rozbudowa systemu o funkcje autonomicznego ładowania baterii.
- Zastosowanie algorytmów uczenia maszynowego do optymalizacji nawigacji i mapowania.
- Integracja z systemami IoT (Internet of Things) w celu zdalnego monitorowania i sterowania robotem.



# Bibliografia

- [1] Luis Bermudez. *Medium - Overview of SLAM*. 2024. URL: <https://medium.com/machinevision/overview-of-slam-50b7f49903b7> (term. wiz. 17.04.2024).
- [2] 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).
- [3] Dieter Fox. „KLD-Sampling: Adaptive Particle Filters.” W: sty. 2001, s. 713–720.
- [4] 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).
- [5] 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).
- [6] Matti Kortelainen. „A short guide to ROS 2 Humble Hawksbill”. W: *School of Computing, University of Eastern Finland, Kuopio, Finland* (2023), s. 5.
- [7] 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>.
- [8] 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>.
- [9] 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).
- [10] 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*)



# Źródła

Jeżeli w pracy konieczne jest umieszczenie długich fragmentów kodu źródłowego, należy je przenieść w to miejsce.

---

```
1 if (_nClusters < 1)
2     throw std::string ("unknown number of clusters");
3 if (_nIterations < 1 and _epsilon < 0)
4     throw std::string ("You should set a maximal number of
5         iteration or minimal difference --- epsilon .");
6 if (_nIterations > 0 and _epsilon > 0)
7     throw std::string ("Both number of iterations and minimal
8         epsilon set --- you should set either number of iterations
9         or minimal epsilon .");
```

---



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

W systemie do pracy dołączono dodatkowe pliki zawierające:

- źródła programu,
- dane testowe,
- film pokazujący działanie opracowanego oprogramowania lub zaprojektowanego i wykonanego urządzenia,
- itp.



# Spis rysunków

2.1	Reprezentacja pośrednika w systemie robota [10] . . . . .	6
2.2	Mapa dwóch pomieszczeń utworzona za pomocą SLAM Toolbox [7] . . . . .	7
3.1	Diagram przypadków użycia systemu . . . . .	12
3.2	Raspberry Pi 4 model B . . . . .	13
3.3	Arduino Nano . . . . .	14
3.4	Silnik DC 12V 240RPM typu L z przekładnią metalową - magnetyczny enkoder Halla - Waveshare 22346 . . . . .	14
3.5	L298N - dwukanałowy sterownik silników - moduł 12V/2A . . . . .	15
3.6	Ogniwa 18650 Li-Ion XTAR - 2600mAh . . . . .	15
3.7	Przetwornica step-down LM2596 3,2V-35V 3A z wyświetlaczem . . . . .	16
3.8	Zdjęcie przedstawiające zbudowanego robota . . . . .	17
3.9	Uproszczony schemat przedstawiający budowę robota . . . . .	18
3.10	Schemat połączenia silników z Arduino i L298N . . . . .	19
3.11	Schemat elektryczny . . . . .	20
4.1	Wizualizacja w Rviz po uruchomieniu skryptów. . . . .	25
4.2	Wizualizacja zmapowanego pomieszczenia i zapisu mapy. . . . .	26
4.3	Wizualizacja po uruchomieniu skryptów do lokalizacji i nawigacji. . . . .	28
4.4	Wizualizacja rossproszonych punktów lokalizacji robota. . . . .	28
4.5	Wizualizacja po wybraniu celu do którego robot ma się przemieścić. . . . .	29
4.6	Wizualizacja wyznaczenia trasy. . . . .	30
4.7	Wizualizacja nawigacji robota i zmniejszania się chmury przewidywanej lokalizacji robota. . . . .	30
4.8	Wizualizacja po dotarciu robota do celu. . . . .	31
5.1	Wizualizacja połączenia skryptów w systemie . . . . .	34
5.2	Komendy jakie są wysyłane z Raspberry Pi do Arduino . . . . .	36
5.3	Implementacja głównej funkcji regulatora PID . . . . .	38
5.4	Konfiguracja portów enkoderów z Arduino i sterownikiem L298N . . . . .	39
5.5	Konfiguracja portów silników . . . . .	40
5.6	Diagram UML prezentujący działanie programu model_controll.launch.py .	54

5.7	Diagram UML prezenujący działanie programu rplidar.launch.py . . . . .	55
5.8	Diagram UML prezenujący działanie programu navigation_launch.py . . .	55
5.9	Diagram UML prezenujący działanie programu localization_launch.py . .	56

# Spis tabel