Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Sterowanie i symulacja robotów

Sprawozdanie z projektów i ćwiczeń laboratoryjnych (cześć mobilna)

Kamil Szczepanik, Piotr Hondra

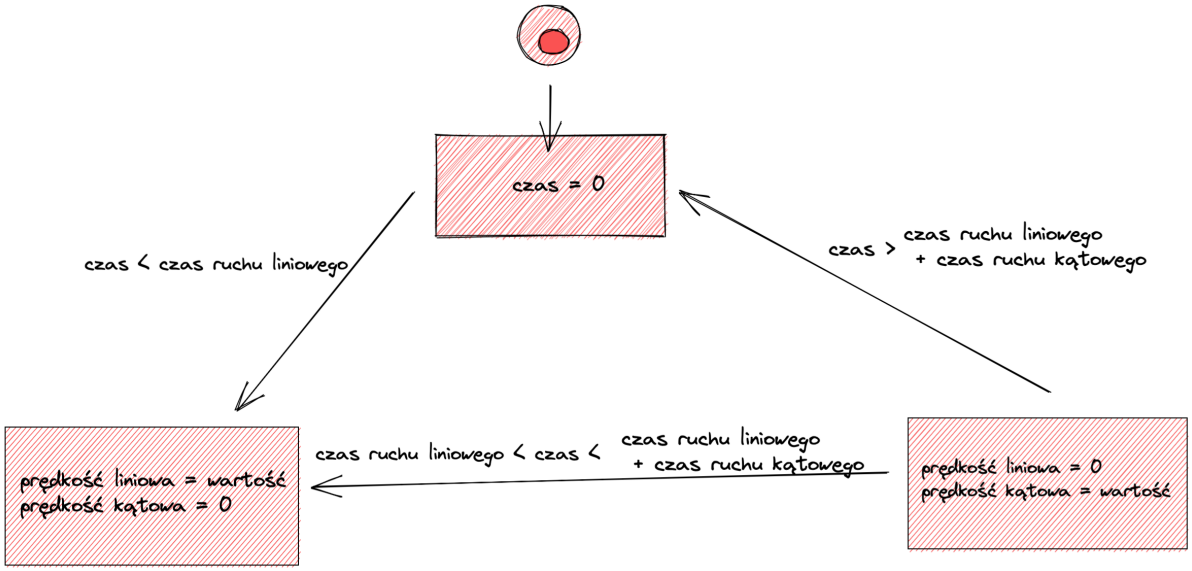
Warszawa, 2021

Spis treści

1. Laboratorium 1

1.1. Sterowanie prędkościowe

Sterowanie prędkościowe polega na obliczaniu czasu wykonywania ruchu obrotowego oraz liniowego, znając długość boku kwadratu po którym chcemy się poruszać oraz zadaną prędkość.



Rys. 1.1: Automat stanów dla sterowania prędkościowego

Czasy wykonania prędkości obliczone zostały przy użyciu podstawowych zależności kinematyki.

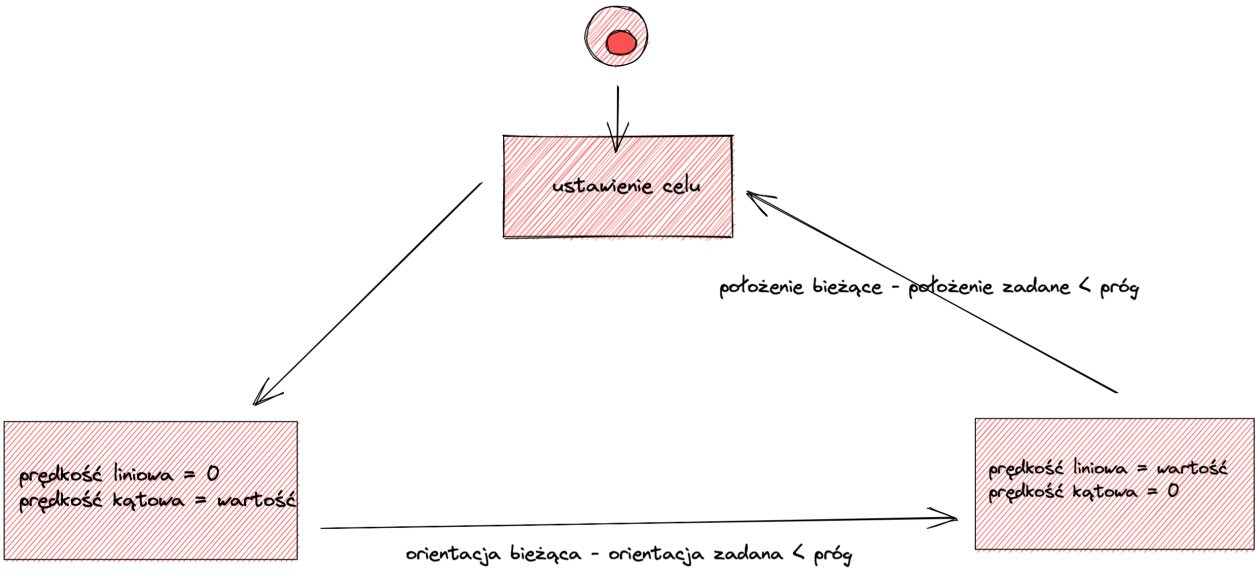
```
def calc_time_lin(square_size, vel_lin):
    return square_size/vel_lin

def calc_time_ang(vel_ang):
    return (math.pi/2)/vel_ang
```

Warunkiem dotarcia do celu jest przekroczenie limitu czasu.

1.2. Sterowanie odometryczne

Sterowanie odometryczne polega na okresowym pobieraniu danych o pozycji robota, opierających się na odometrii.



Rys. 1.2: Automat stanów dla sterowania odometrycznego

Prędkość kątowa jest niezerowa dopóki różnica między orientacja zadaną, a obecną jest nie mniejsza od arbitralnie ustawionego progu. Analogicznie w przypadku prędkości liniowej oraz pozycji.

```
def get_distance(goal_pose):
    global pose
    pose
    return math.sqrt(pow(goal_pose[0] - pose.position.x, 2) +
                     pow(goal_pose[1] - pose.position.y, 2))


def get_angle_diff(goal_pose):
    global pose
    theta = tf.transformations.euler_from_quaternion(
        [pose.orientation.x, pose.orientation.y,
         pose.orientation.z, pose.orientation.w])[2]
    goal_angle = math.atan2(
        goal_pose[1]-pose.position.y, goal_pose[0]-pose.position.x)
    return goal_angle - theta


def is_distance_achived(goal_pose, tolerance):
    print("distance_diff: ", get_distance(goal_pose))
    return get_distance(goal_pose) < tolerance


def is_angle_achived(goal_pose, tolerance):
    print("angle_diff: ", get_angle_diff(goal_pose))
    return abs(get_angle_diff(goal_pose)) < tolerance
```

Warunkiem dotarcia do celu jest osiągnięcie zadanej położenia z dokładnością do progu.

2. Projekt 1

2.1. Struktura oprogramowania stworzonego do zbierania danych

Diagram strukturalny został przedstawiony na rysunku 2.1

2.2. Opis działania węzła zbierającego dane

1. Pobranie transformacji początkowej
2. Pobranie /mobile_base_controller/odom oraz /robot_pose
3. Przekształcenie układu odom do map na podstawie transformacji początkowej
4. Zapisywanie danych do pliku csv

2.3. Opis działania węzła sterującego robotem

System działa zgodnie z schematem z rysunku 2.2

2.4. Sposób analizy danych

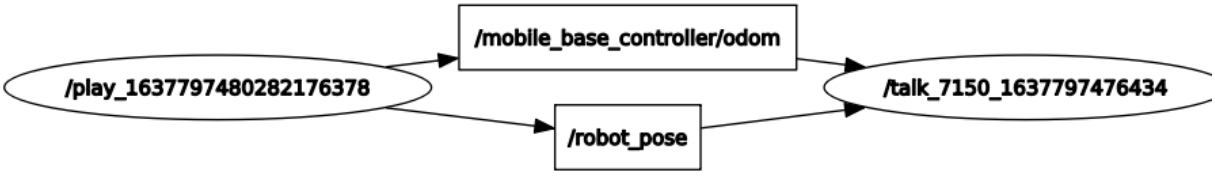
2.4.1. Analiza w symulatorze

Pierwszą częścią analizy było porównanie sterowania na symulatorze `Gazebo`. Obejmowało ono porównanie sterowania prędkościowego i odometrycznego na podstawie prędkości zadanych i wykonanych za pomocą rqt_graph.

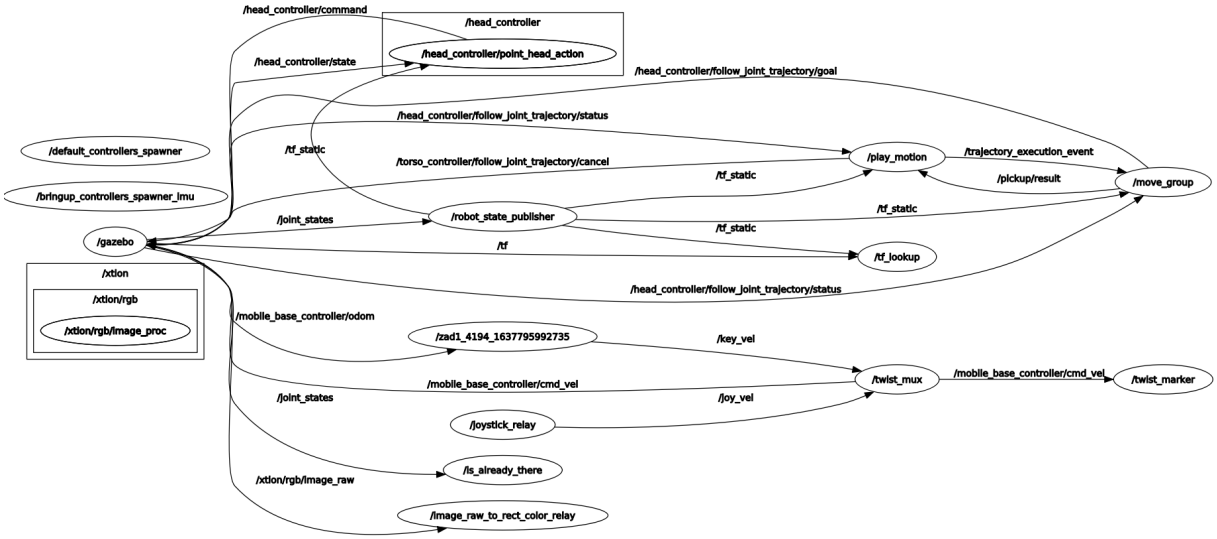
2.4.2. Analiza na rzeczywistym robocie

Drugą częścią analizy były testy na rzeczywistym robocie Tiago. Użyto tylko sterowania prędkościowego. Analiza polegała na sprawdzeniu:

1. trasy jaką wykonał robot.
   W tym celu wykreślono położenie zadane i wykonane robota na płaszczyźnie.
2. błędu całkowitego
3. błędu średniego
4. błedu chwilowego
   Narysowano przebieg zależności błędu chwilowego w kolejnych chwilach czasu.



Rys. 2.1: Struktura węzła zbierającego dane

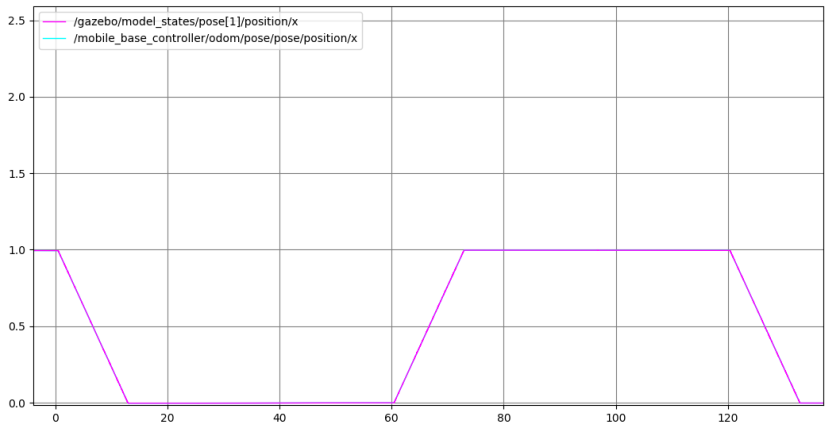


Rys. 2.2: Struktura węła sterującego robotem

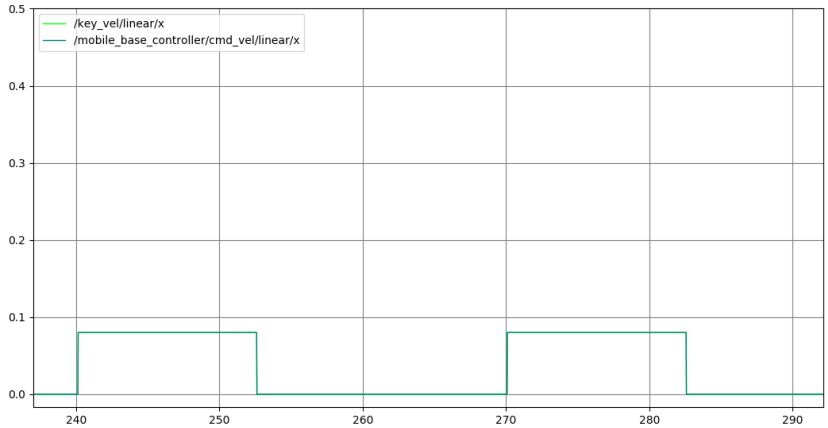
2.5. Wykresy i wnioski

2.5.1. Analiza w symulatorze

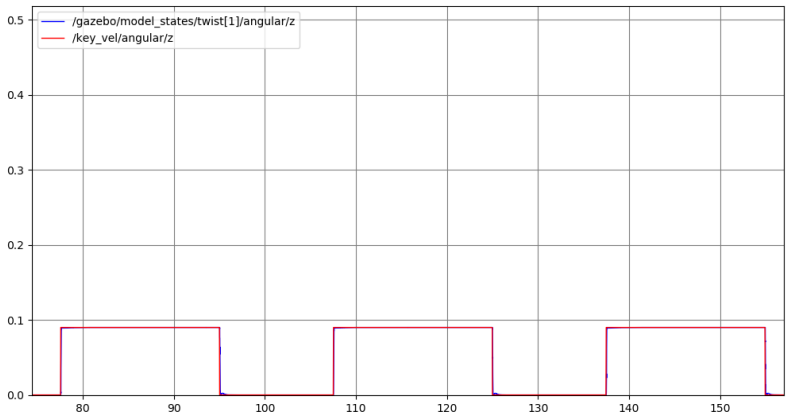
W symulatorze parametry zadane robotowi (pozycja i prędkości) są praktycznie takie same jak dane z symulowanego świata. Wynika to z tego, że symulator jest idealnym światem,



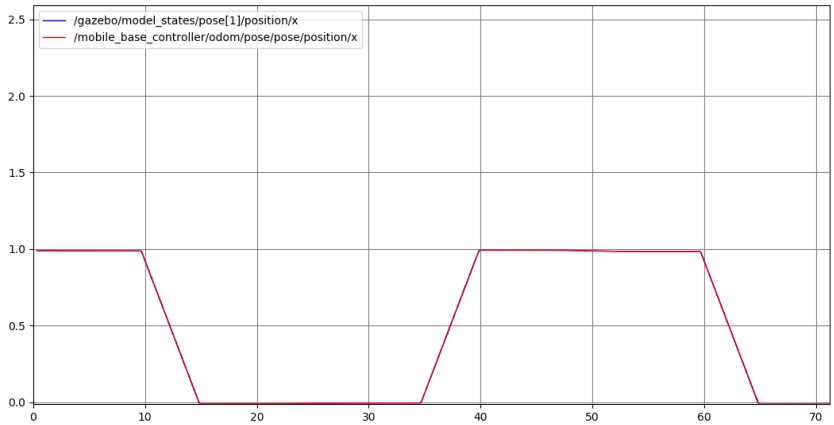
Rys. 2.3: Pozycje zadane i wykonane dla małej prędkości - sterowanie prędkościowe



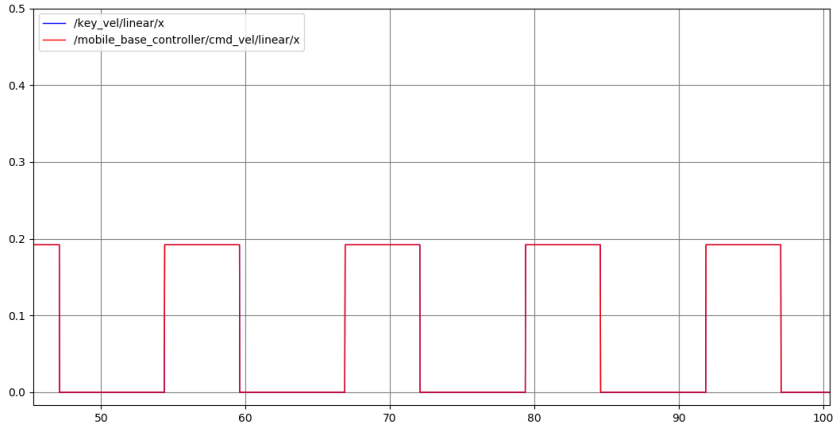
Rys. 2.4: Prędkości liniowe zadane i wykonane dla małej prędkości - sterowanie prędkościowe



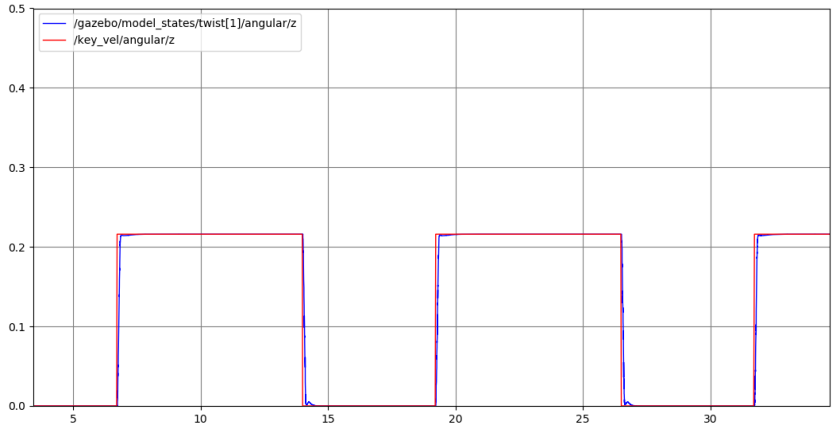
Rys. 2.5: Prędkości kątowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



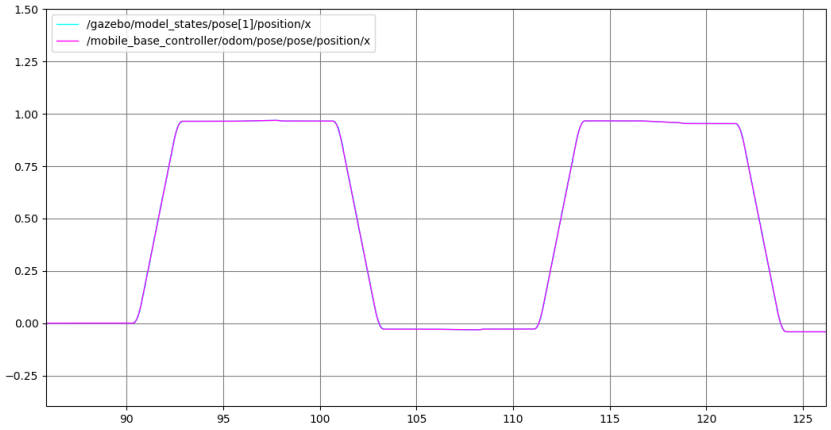
Rys. 2.6: Pozycje zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



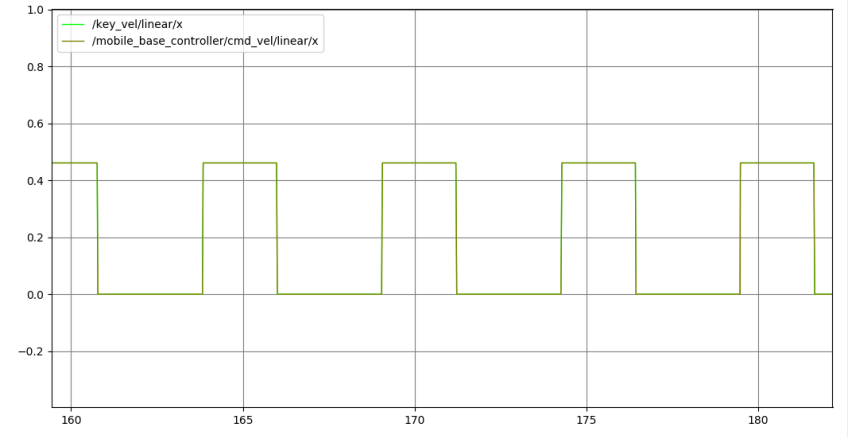
Rys. 2.7: Prędkości liniowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



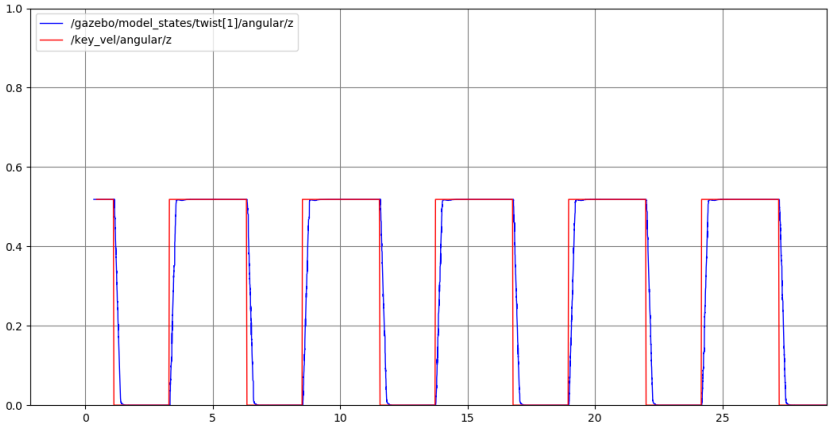
Rys. 2.8: Prędkości kątowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



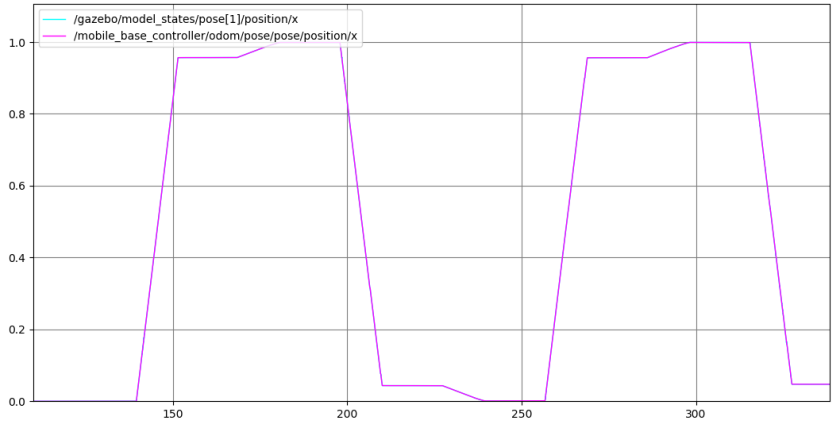
Rys. 2.9: Pozycje zadane i wykonane dla dużej prędkości - sterowanie prędkościowe



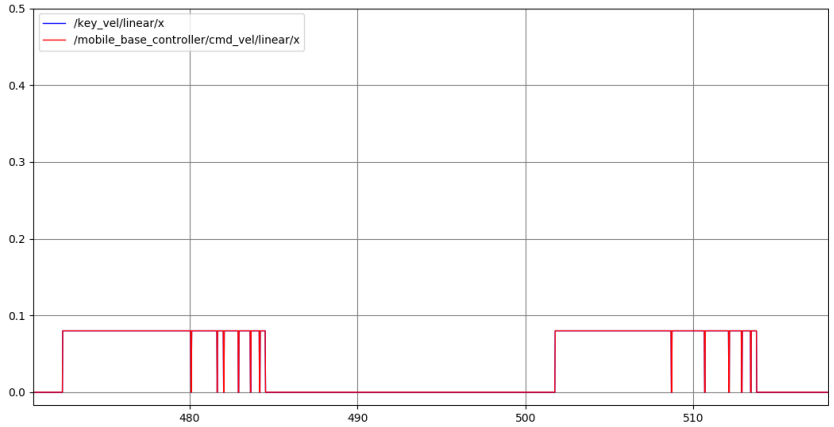
Rys. 2.10: Prędkości liniowe zadane i wykonane dla dużej prędkości - sterowanie prędkościowe



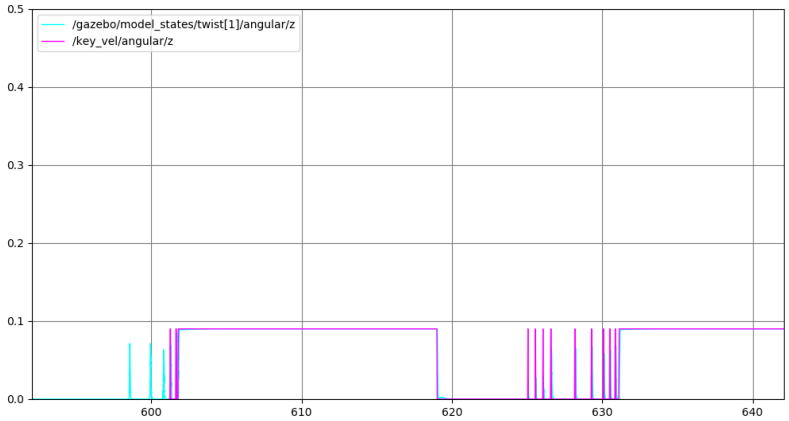
Rys. 2.11: Prędkości kątowe zadane i wykonane dla dużej prędkości - sterowanie prędkościowe



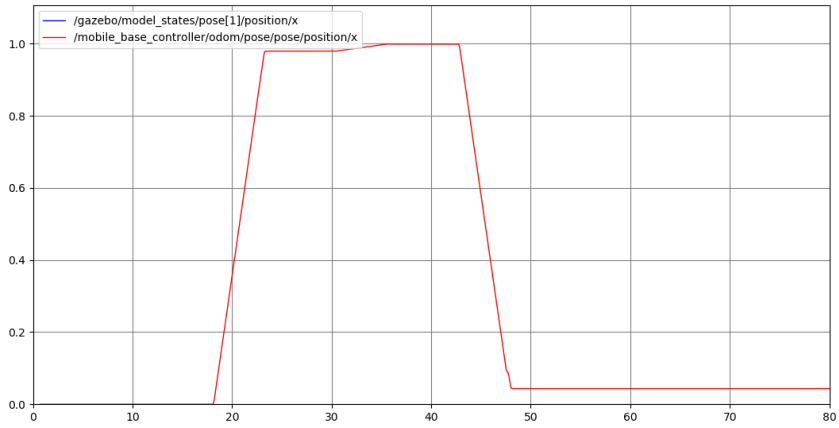
Rys. 2.12: Pozycje zadane i wykonane dla małej prędkości - sterowanie prędkościowe



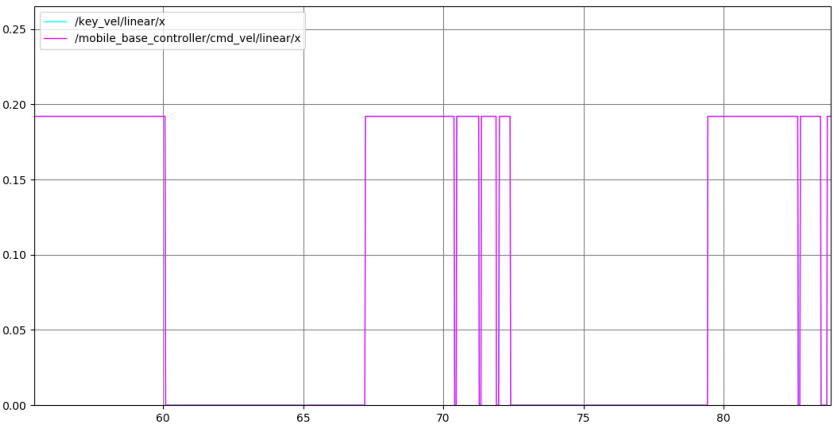
Rys. 2.13: Prędkości liniowe zadane i wykonane dla małej prędkości - sterowanie prędkościowe



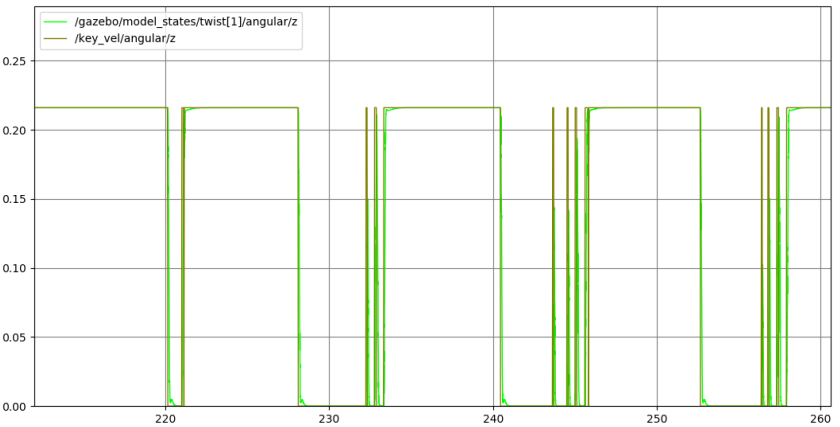
Rys. 2.14: Prędkości kątowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



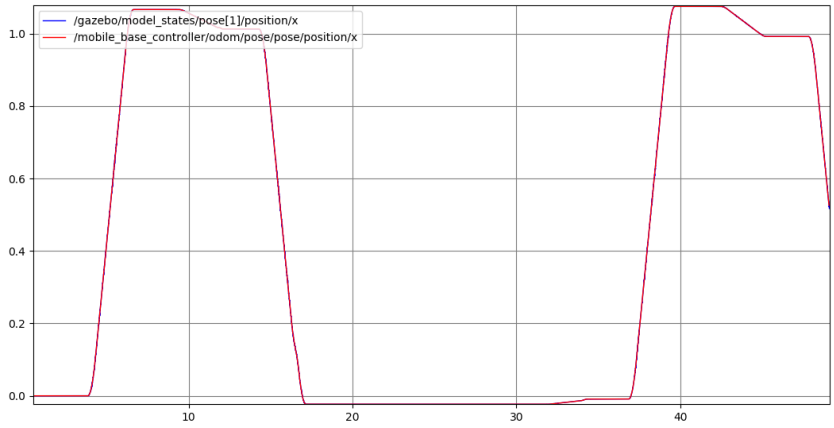
Rys. 2.15: Pozycje zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



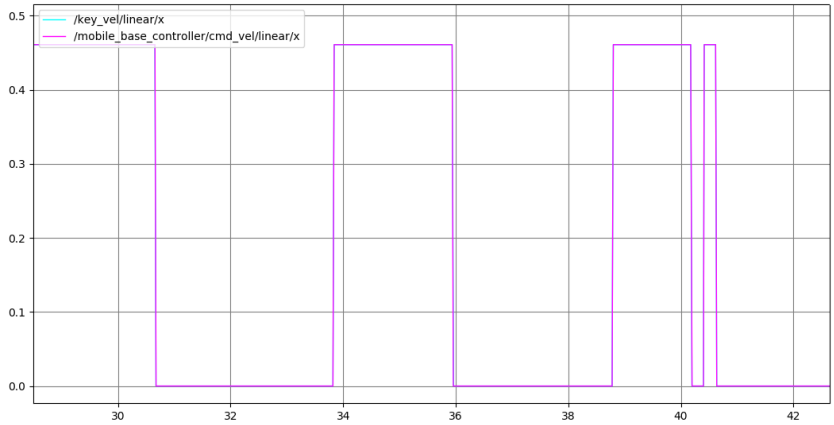
Rys. 2.16: Prędkości liniowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



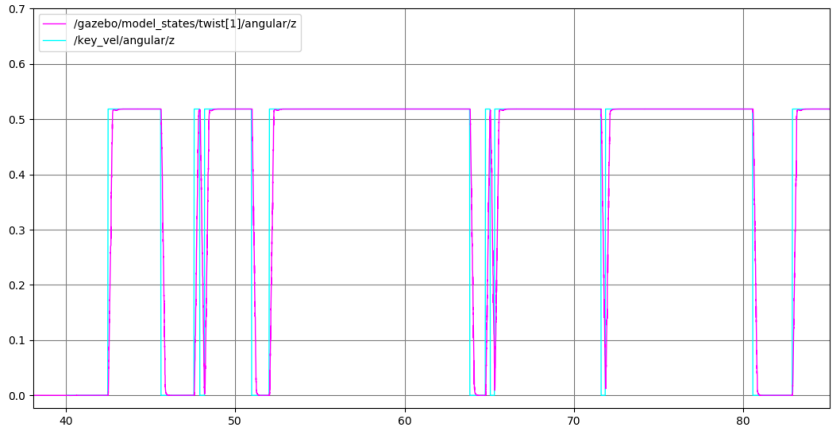
Rys. 2.17: Prędkości kątowe zadane i wykonane dla średniej prędkości - sterowanie prędkościowe



Rys. 2.18: Pozycje zadane i wykonane dla dużej prędkości



Rys. 2.19: Prędkości liniowe zadane i wykonane dla dużej prędkości



Rys. 2.20: Prędkości kątowe zadane i wykonane dla dużej prędkości

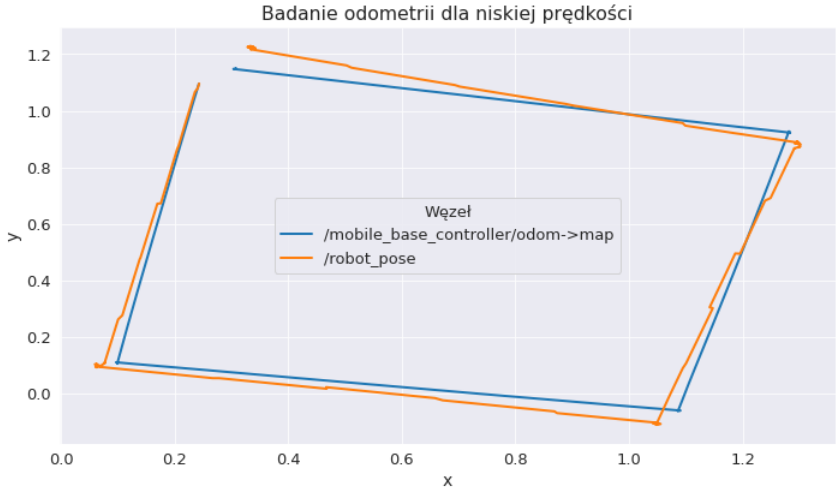
gdzie nie występują szumy pomiarowe, opóźnienia czy inne nieprzewidziane zakłócenia mogące spowodować rozbieżności w tych wykresach.

2.5.2. Analiza na rzeczywistym robocie

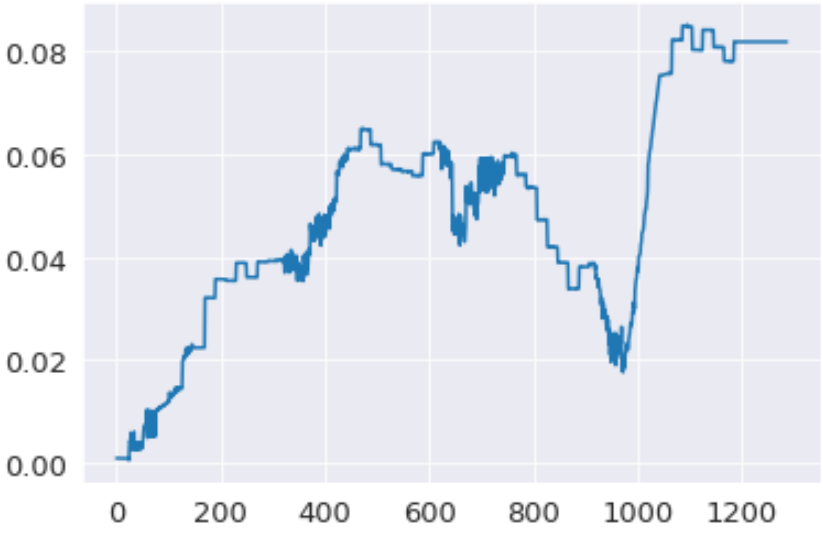
Algorytm sterowania prędkościowego został także przetestowane na prawdziwym robocie w środowisku laboratoryjnym. Eksperyment pozwolił na zebranie danych w postaci rosbaga czyli zapisu przechwyconych tematów podczas trwania działania programu. Tak jak w symulacji doświadczenie przeprowadzono w trzech wielkościach prędkości.

Z rosbagów udało się wyłuskać informacje o pozycji zadanej i wykonanej robota. Z tych danych zostały wygenerowane wykresy,które przedstawiają te tory jazdy.

Dodatkowo wyznaczono błędy pozycji. Błędy w chwili pomiaru zostały wyznaczone na wykresie.

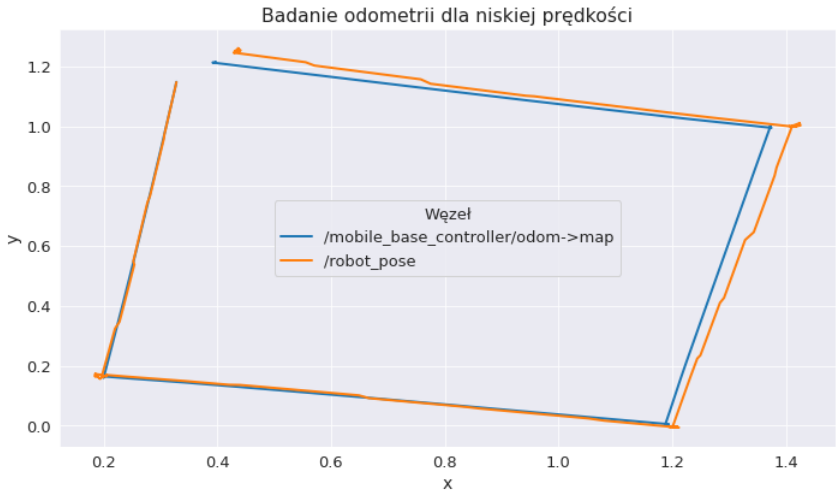


Rys. 2.21: Badanie odometrii dla niskiej prędkości

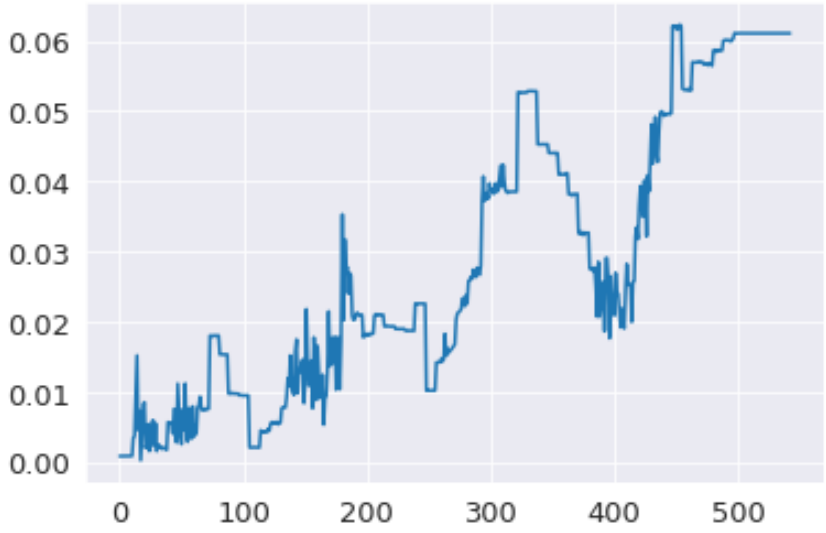


Rys. 2.22: Błąd chwilowy dla niskiej prędkości

Błąd zakumulowany: $62,62313$ **Błąd średni:** $0,048658$

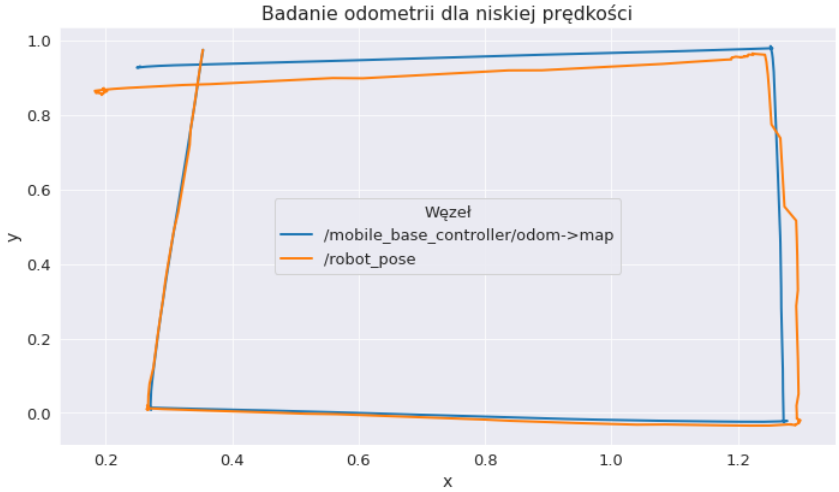


Rys. 2.23: Badanie odometrii dla średniej prędkości

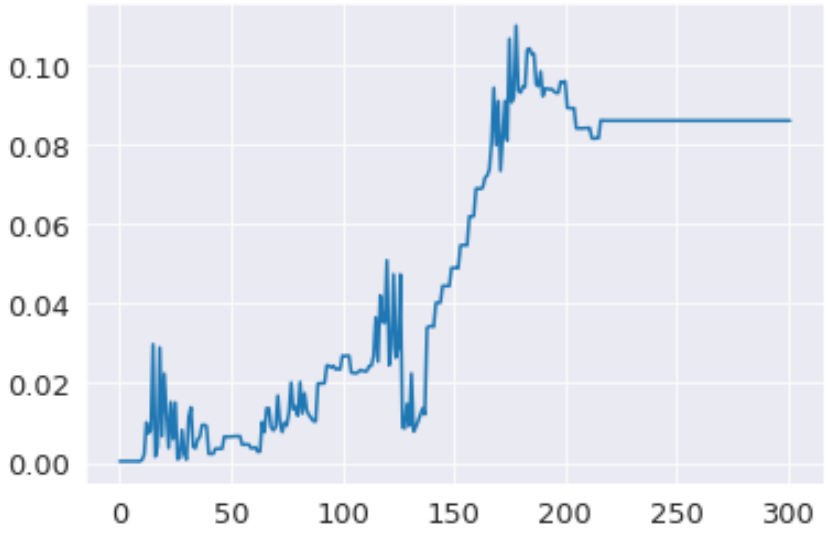


Rys. 2.24: Błąd chwilowy dla średniej prędkości

Błąd zakumulowany: $62,62313$ **Błąd średni:** $0,048658$



Rys. 2.25: Badanie odometrii dla dużej prędkości



Rys. 2.26: Błąd chwilowy dla dużej prędkości

Błąd zakumulowany: 62, 62313 **Błąd średni:** 0, 048658

Wnioski

Z ilustracji przedstawiających tor jazdy rzeczywistego robota i lokalizację zmierzoną odometrycznie można wywnioskować, że dane te bardzo się różnią. Błąd lokalizacji zależy również od prędkości. Najmniejszy średni błąd miał test przy prędkości średniej.

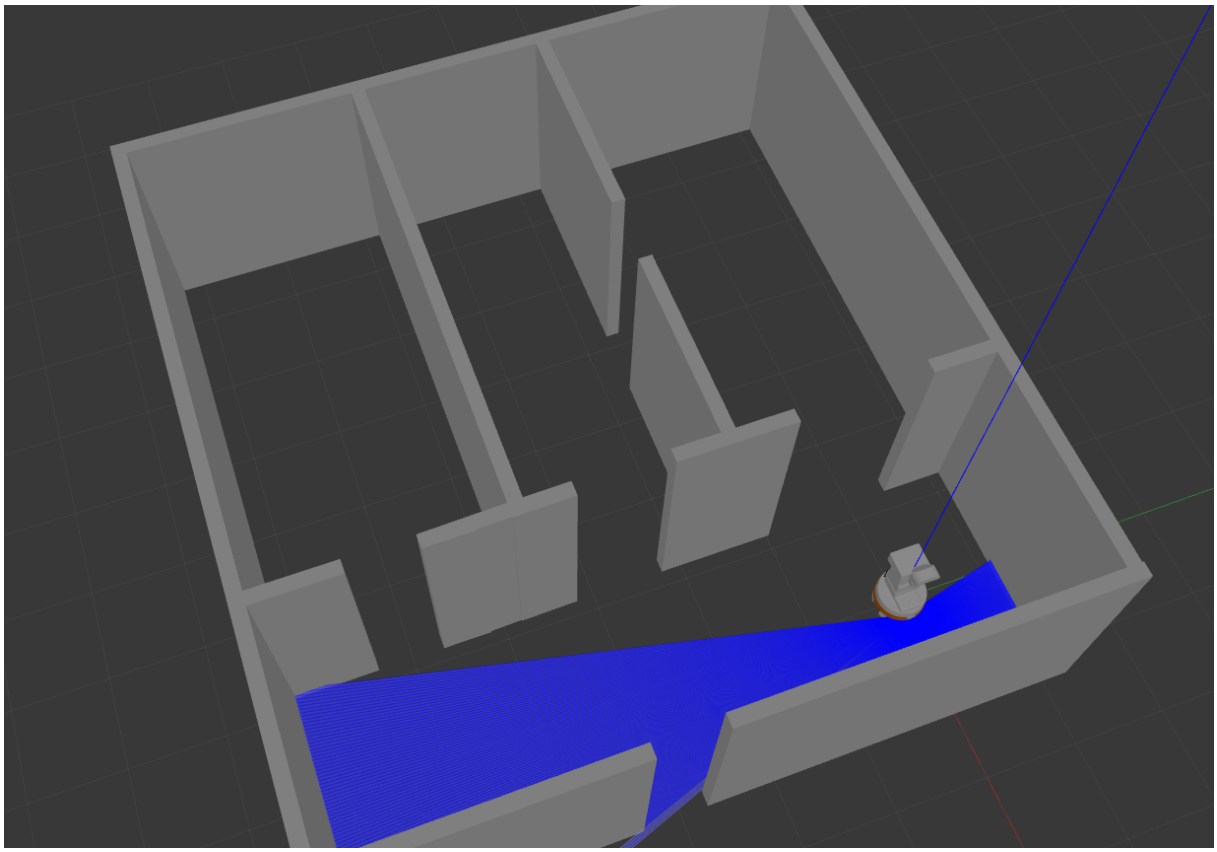
3. Laboratorium 2

3.1. Stworzenie środowisk do symulacji

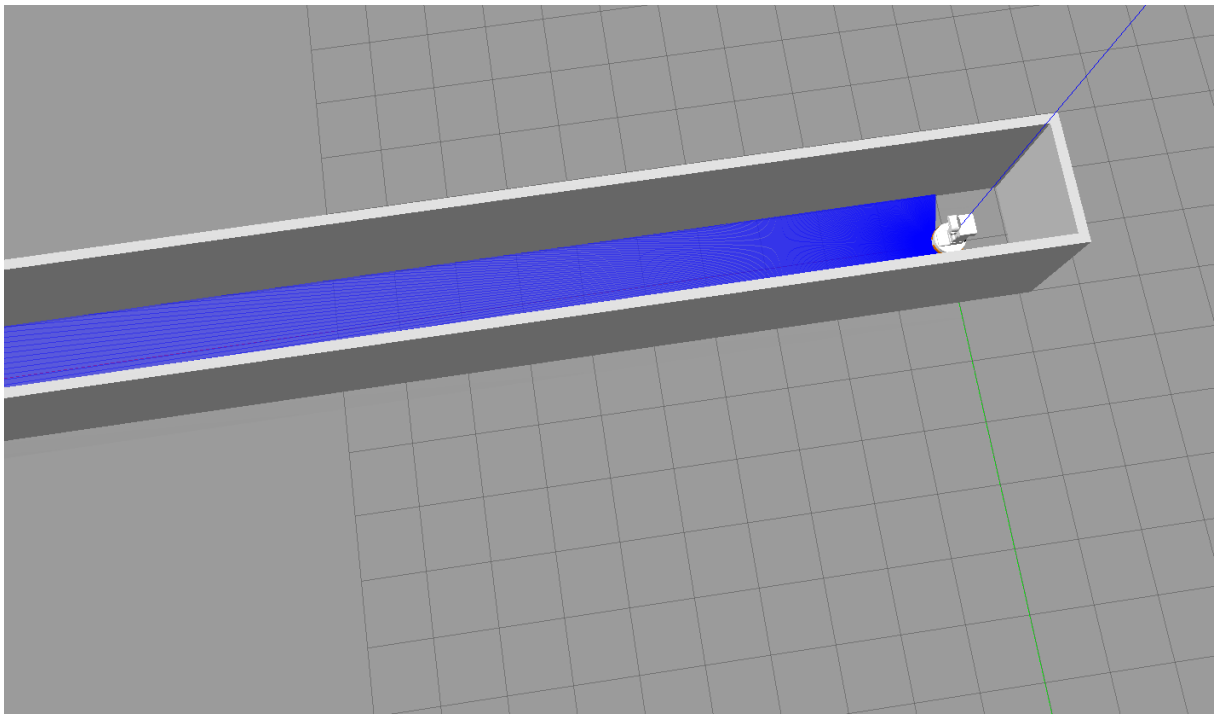
Pierwszym zadaniem laboratoryjnym było stworzenie światów potrzebnych do symulacji zachowania robota w różnych środowiskach. W symulatorze Gazebo należało zbudować dwa światy. Pierwszym z nich był "parterowy budynek", czyli pokoje połączone przejściami o różnych szerokościach. Przejścia powinny mieć szerokość 100%, 150%, 200%, 300% średnicy robota. Gotowy świat przedstawiono na rys. 3.1. Drugim środowiskiem miał być długi na przynajmniej $25m$ korytarz bez drzwi. Przedstawiono go na rys. 3.2

3.2. Utworzenie pliku startowego z konfiguracją modułu SLAM

W tej części należało utworzyć plik startowy *roslaunch*, który uruchomi symulację robota w przygotowanym wcześniej świecie. Dodatkowo powinien on też być tak skonfigurowany aby przechwytywana była mapa publikowana przez węzeł SLAM (Simultaneous localization and mapping). Węzeł ten jest częścią pakietu *gmapping*, którego opis jest dostępny tutaj: `http://wiki.ros.org/gmapping`. Jest to węzeł tworzący mapę na podstawie danych z czujnika laserowego oraz danych odometrycznych. Na bieżąco buduje on mapę i lokalizuje na niej robota.



Rys. 3.1: Świat "budynek parterowy" z przejściami 100%, 150%, 200%, 300% średnicy robota



Rys. 3.2: Świat "korytarz bez drzwi"

3.3. Budowa mapy przez zdalne sterowanie, zapisanie jej oraz automatyczne wczytanie przy starcie systemu

Mając gotowy plik *roslaunch* należało włączyć go żeby rozpocząć symulację, a następnie należało włączyć węzeł SLAM

```
rosrun gmapping slam_gmapping scan:=scan_raw
```

Ważne aby węzeł korzystał z tematu "scan_raw", ponieważ na nim publikowane są dane z czujnika laserowego. Następnie włączono węzeł zdalnego sterowania

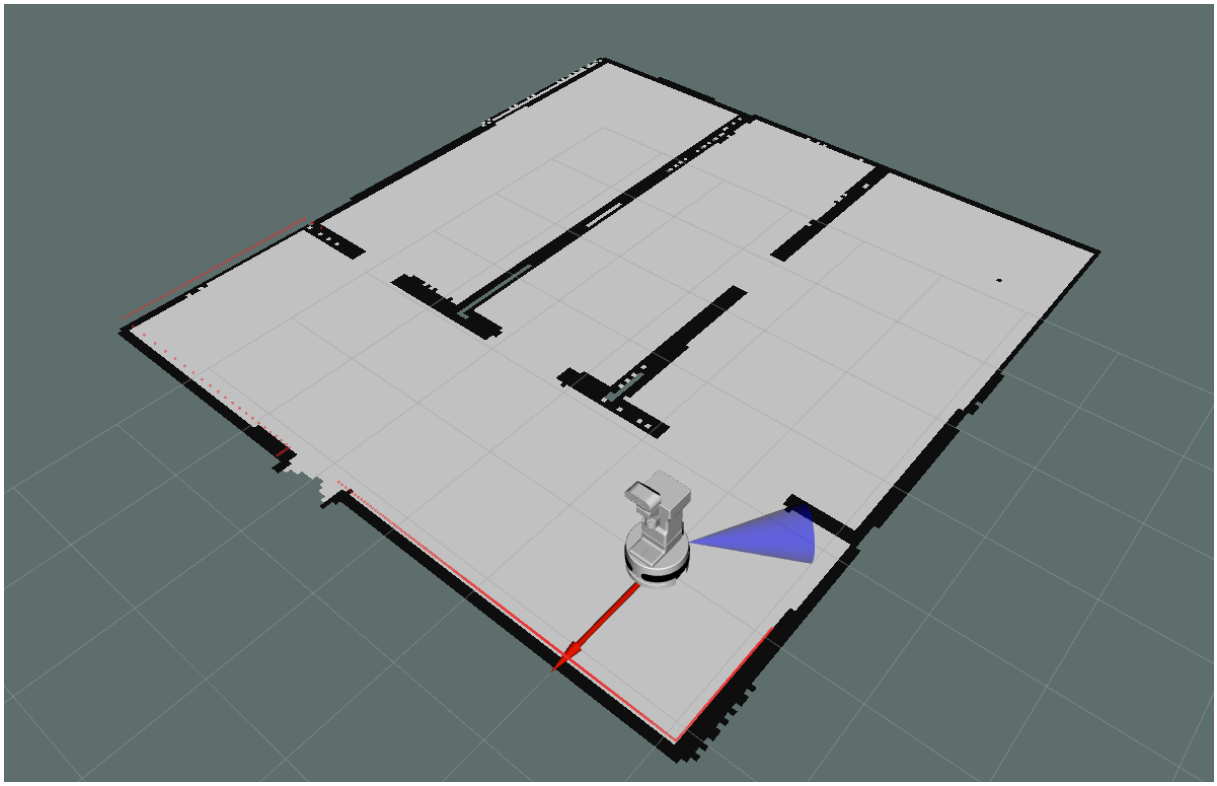
```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=key_vel
```

Stworzył on możliwość poruszania się robotem za pomocą przycisków z klawiatury. Dzięki wizualizacji w *RViz* można było sterować robotem tak aby jeździł po pokoju i tworzył mapę. Tworzyła się ona na bieżąco wraz z przemieszczaniem się robota. Ważne aby podczas wykonywania SLAM robot nie zahaczył albo nie wjechał w ścianę. Może to spowodować błędy w tworzeniu mapy przez wywołane błędy w odometrii. Po stworzeniu mapy całego budynku, zapisano ją poprzez uruchomienia węzła *map_server*. Następnie zmodyfikowano plik *roslaunch* tak, aby mapa była wczytywana przy starcie systemu. Uzupełnienie pliku tiago_navi.launch zostało przedstawione na listingu 3.1. Należało pamiętać o dodaniu przekształcenia układu świata i mapy, żeby środek stworzona mapa wizualizowała się w środku świata.

Wynikiem jest mapa przedstawiona na rys. 3.3

```
...
<node name="map_server" pkg="map_server" type="map_server"
    args="$(find stero_mobile_init)/maps/$(arg world).yaml"/>
  <node name="map_odom_tranform" pkg="tf"
    type="static_transform_publisher"
    args="0 0 0 0 0 0 map odom 10"/>
  <node name="navi" pkg="stero_mobile_init" type="navi"
    output="screen">
    <remap from="cmd_vel" to="key_vel"/>
    <rosparam
        file="$(find stero_mobile_init)/param/local_costmap_params.yaml"
        command="load"/>
    <rosparam
        file="$(find stero_mobile_init)/param/global_costmap_params.yaml"
        command="load"/>
    <rosparam
        file="$(find stero_mobile_init)/param/local_planner_params.yaml"
        command="load"/>
    <rosparam
        file="$(find stero_mobile_init)/param/global_planner.yaml"
        command="load"/>
  </node>
  ...
```

Listing 3.1: Fragment tiago_navi.launch



Rys. 3.3: Mapa stworzona modułem SLAM gmapping

4. Projekt 2

Celem projektu drugiego była implementacja modułu planowania lokalnego zawierającego:

— generowanie trajektorii
— zachowanie uwolnienia z zakleszczenia
— wykonywanie ścieżki

4.1. Struktura sterownika robota

Należało stworzyć węzeł, który będzie nawigował robota, aby dojechał on do wyznaczonego celu. W tym celu należało wykorzystać stworzoną na laboratorium 2 mapę przedstawiającą układ pomieszczeń symulowanego budynku. Na podstawie tej mapy można było utworzyć globalną mapę kosztów, czyli mapa z wyznaczonymi obszarami dozwolonymi, dozwolonymi ale niepreferowanymi oraz niedozwolonymi. Dzięki takiej mapie kosztów możliwe jest stworzenie ścieżki z punktu startowego do celu. Ścieżka ta nie zmienia się do końca trwania ruchu.

Do wyznaczania trajektorii potrzebne jest stworzenie lokalnej mapy kosztów, która jest tworzona na bieżąco na podstawie odczytu z czujnika laserowego i danych odometrycznych. Ruch robota generowany jest na podstawie planu lokalnego stworzonego przez planera lokalnego. Jest to konieczne jeśli robot znajduje się w zmiennym otoczeniu, ponieważ planer lokalny oblicza trajektorię na kilka chwil w przód oraz bierze pod uwagę aktualne otoczenie, przez co prędkość zadawana sterownikowi wyznaczana jest dynamicznie. To pozwala na omijanie wcześniej niezarejestrowanych przeszkód. Planer lokalny do wyznaczania trajektorii uwzględnia też ścieżkę globalną aby znaleźć jak najbardziej optymalną trasę.

Na wypadek gdyby robot się "zakleszczył", to znaczy planer lokalny nie bedzie mógł wygenerować nowej ścieżki, uruchamiane jest zachowanie odzyskiwania *recoverybehaviour*, którego celem jest wydostanie się ze stanu zablokowania.

Aby te wszystkie funkcjonalności działały należy skonfigurować warstwy mapy globalnej i lokalnej.

Warstwy **mapy globalnej**:

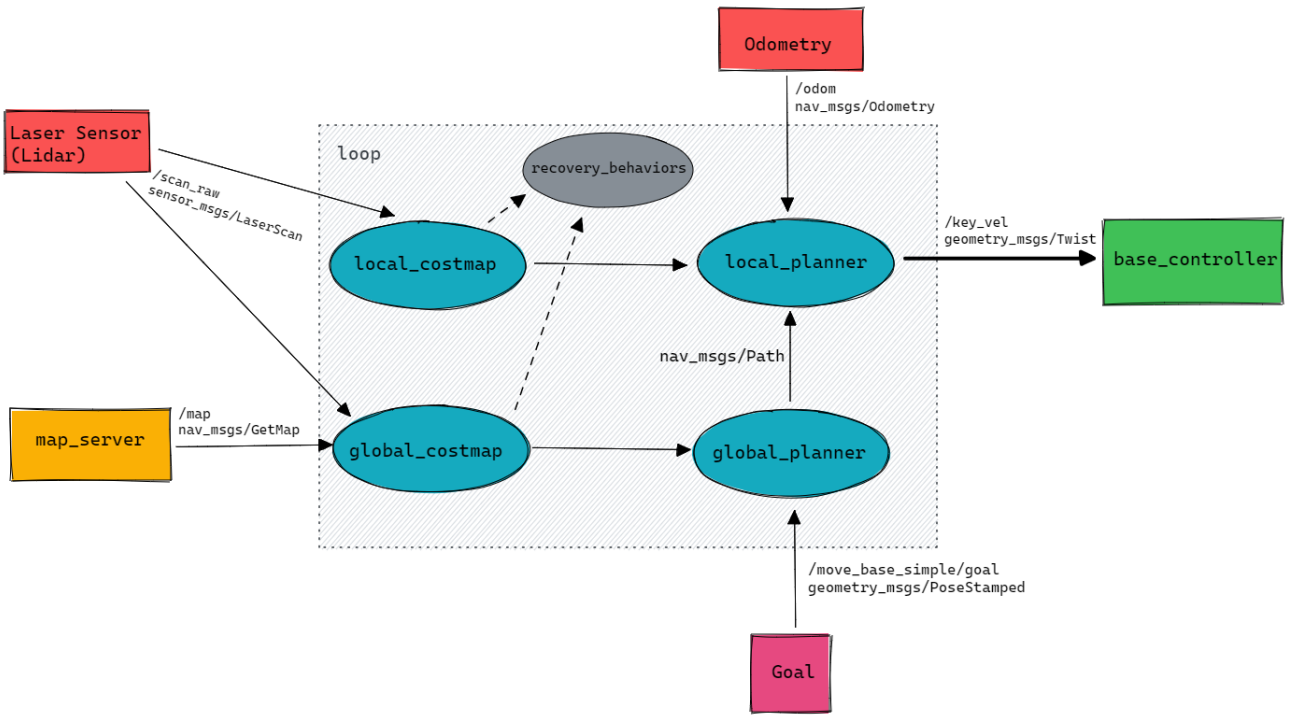
— mapa statyczna
— warstwa nadmuchania ścian

Warstwy **mapy lokalnej**:

— mapa przeszkód
— warstwa nadmuchania ścian

4.2. Opis działania węzła planującego

Strukturę systemu nawigacji robota pokazano na rys. 4.1.



Rys. 4.1: Struktura sterownika robota

4.3. Pliki konfiguracyjne map kosztów oraz planera lokalnego

Plik konfiguracyjny globalnej mapy kosztów:

```
global_costmap:
    # warstwy globalnej mapy kosztow
    plugins:
        - {name: static_map, type: "costmap_2d::StaticLayer"}
        - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

    static_map: true # mapa statyczna, czyli ta utworzona na
                     laboratorium za pomoca gmapping (SLAM)

    # warstwa nadmuchania
    inflation_layer:
        radius: 0.5 # promień nadmuchania - odleglosc od sciany
        cost_scaling_factor: 10 # do policzenia funkcji kosztu

    publish_frequency: 1.0
    robot_radius: 0.26 # promien robota
    update_frequency: 10
```

Plik konfiguracyjny lokalnej mapy kosztów:

```
local_costmap:
    # warstwy lokalnej mapy kosztow
    plugins:
        - {name: obstacles, type: "costmap_2d::VoxelLayer"}
        - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
```

```
    # warstwa przeszkod powstaje na biezaco z topica /scan_raw,
    czyli odczytu z czujnika laserowego

    rolling_window: true
    obstacles:
        observation_sources: base_scan
        base_scan: {data_type: LaserScan,
                    sensor_frame: base_laser_link,
                    clearing: true,
                    marking: true,
                    topic: /scan_raw}

    # warstwa nadmuchania
    inflation_layer:
        inflation_radius: 0.05 # promien nadmuchania
        cost_scaling_factor: 5 # do policzenia funkcji kosztu

    publish_frequency: 1.0
    robot_radius: 0.26 # promien robota
```

Plik konfiguracyjny planera lokalnego:

```
local_planner:
    # Robot Configuration Parameters
    # predkosci i przyspieszenia
    max_vel_x: 0.3
    min_vel_x: 0.05
    max_vel_y: 0
    max_rot_vel: 0.5
    min_rot_vel: -0.5
    acc_lim_theta: 2.0
    acc_lim_x: 2.0
    holonomic_robot: false # robot jest nieholonomiczny

    # Goal Tolerance Parameters
    yaw_goal_tolerance: 0.15   # 0.05
    xy_goal_tolerance: 0.2    # 0.10

    # Forward Simulation Parameters
    sim_time: 3.0  # czas symulacji ruchu (3 sekundy do przodu)
    vx_samples: 20
    vtheta_samples: 40

    # Trajectory Scoring Parameters
    meter_scoring: true # odległość w metrach
    penalize_negative_x: false # nie ma kary za oddalenie sie od celu

    gdist_scale: 2 # wspolczynnik mowiacy jak bardzo kontroler
                   # powinien probować dążyć do celu
    pdist_scale: 1 # współczynnik mowiacy jak blisko kontroler
                   #powinien trzymac sie sciezki
```

```
occdist_scale: 0.1 # współczynnik mowiacy jak bardzo kontroler
                   # powinien unikac przeszkod
```

4.4. Wyjaśnienie zastosowanych parametrów

Mapa globalna i lokalna

Bardzo ważnymi parametrami mapy lokalnej i mapy globalnej był promień warstwy nadmuchania **inflation_radius**. W mapie globalnej ustawiono go na $0.5m$. Celowo jest to duża wartość, ponieważ dzięki temu ścieżka globalna wyznaczana jest w taki sposób, że w przejściach odległości od ścian jest wystarczająco bezpieczna. W parametrach mapy lokalnej wartość ta jest o wiele mniejsza bo równa $0.05m$. Takie ustawienie dawało najlepsze wyniki, prawdopodobnie dlatego, że planer lokalny wyznaczał trasę zbliżoną do ścieżki globalnej oraz w momencie wąskiego przejścia robot nie zatrzymywał się z powodu zbyt dużego kosztu wejścia w tą strefę. Tak się działo w przypadku gdy parametr ten ustawiono na większą wartość. Algorytm szukał alternatywnej drogi ale jednocześnie nie chciał oddalać się od ścieżki globalnej.

Ważnym parametrem jest **robot_radius** czyli promień robota. Ta informacja jest potrzebna do wyznaczania bezkolizyjnej trajektorii ruchu.

Planer lokalny

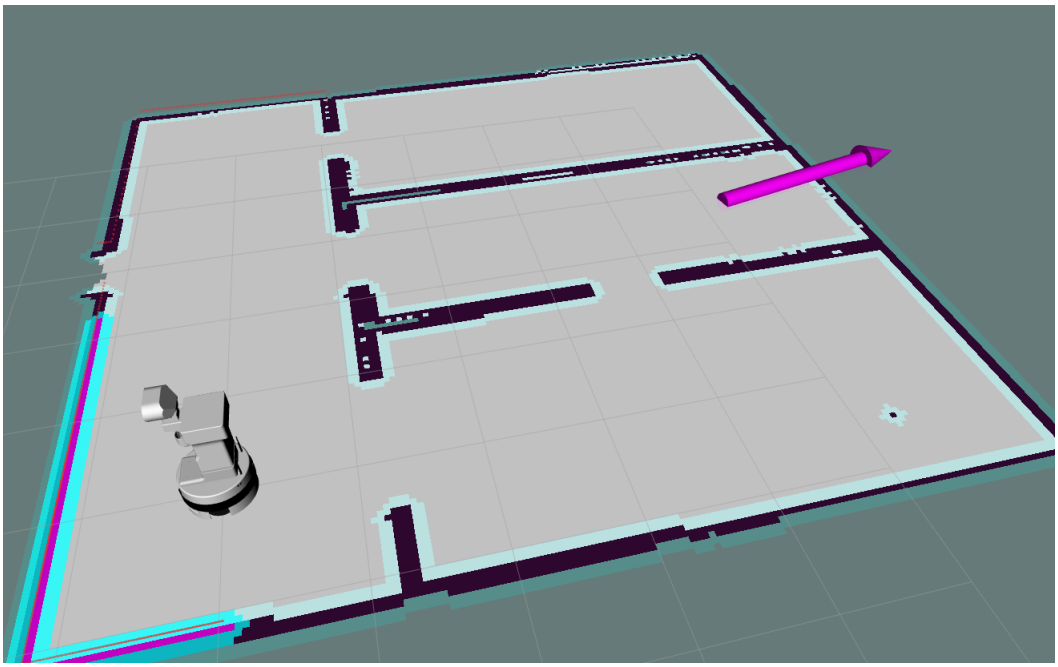
Parametrem, który ma duży wpływ na poruszanie się robota jest **pdist_scale**, czyli parametr mówiący jak bardzo robot powinien trzymać się ścieżki globalnej. Należało nim odpowiednio manipulować. Powinien on być wystarczająco mały aby robot mógł oddalić się od ścieżki, np. w ominięciu przeszkody. Równocześnie współczynnik ten powinien być na tyle duży aby robot trzymał się optymalnej ścieżki globalnej co może być ważne np. w przejeżdżaniu przez wąskie przejście.

Kolejnym parametrem jaki trzeba było ustawić to **occdist_scale**, który odpowiada za to jak bardzo planer ma unikać przeszkód. Im ten współczynnik jest większym tym generowany plan będzie wyznaczany dalej od np. ścian. Nie może być on zamały aby robot mógł przejeżdżać przez wąskie przejścia. Kiedy parametr ustawiono na zbyt dużą wartość, robot zaczynał się kręcić przed przejściem szukając alternatywnej ścieżki, która nie mogła być wyznaczona.

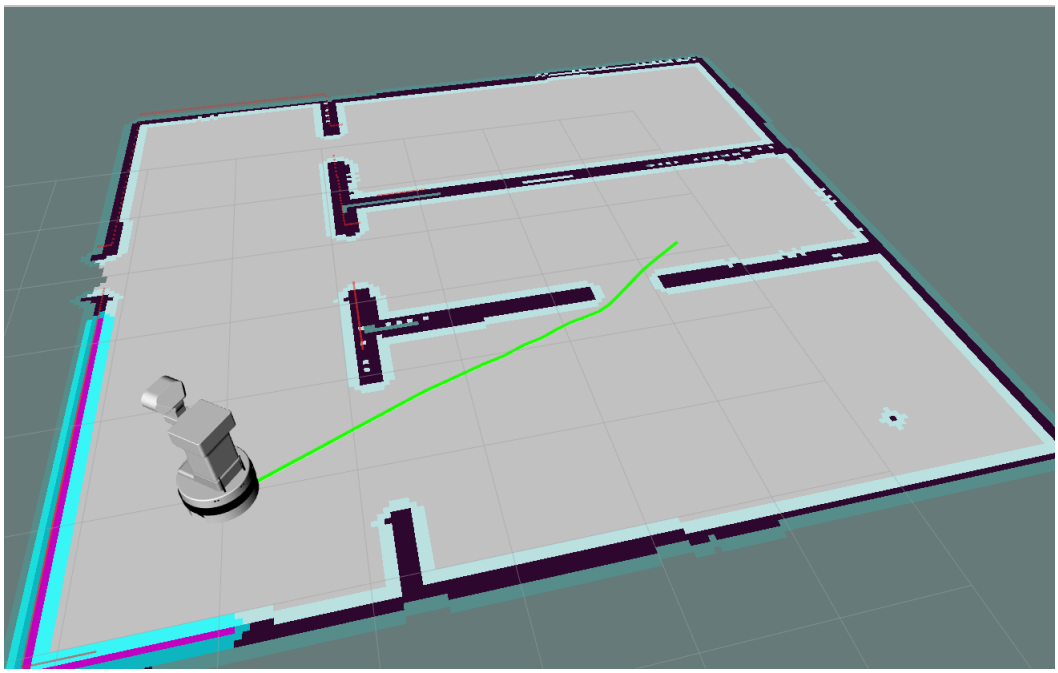
4.5. Wizualizacja przykładowych ścieżek zaplanowanych we własnym środowisku

Na rys. 4.2 przedstawiono sposób zadawania punktu docelowego. Jest to publikowanie wiadomości *PoseStamped* na temacie */move_base_simple/goal* za pomocą funkcjonalności Rviz.

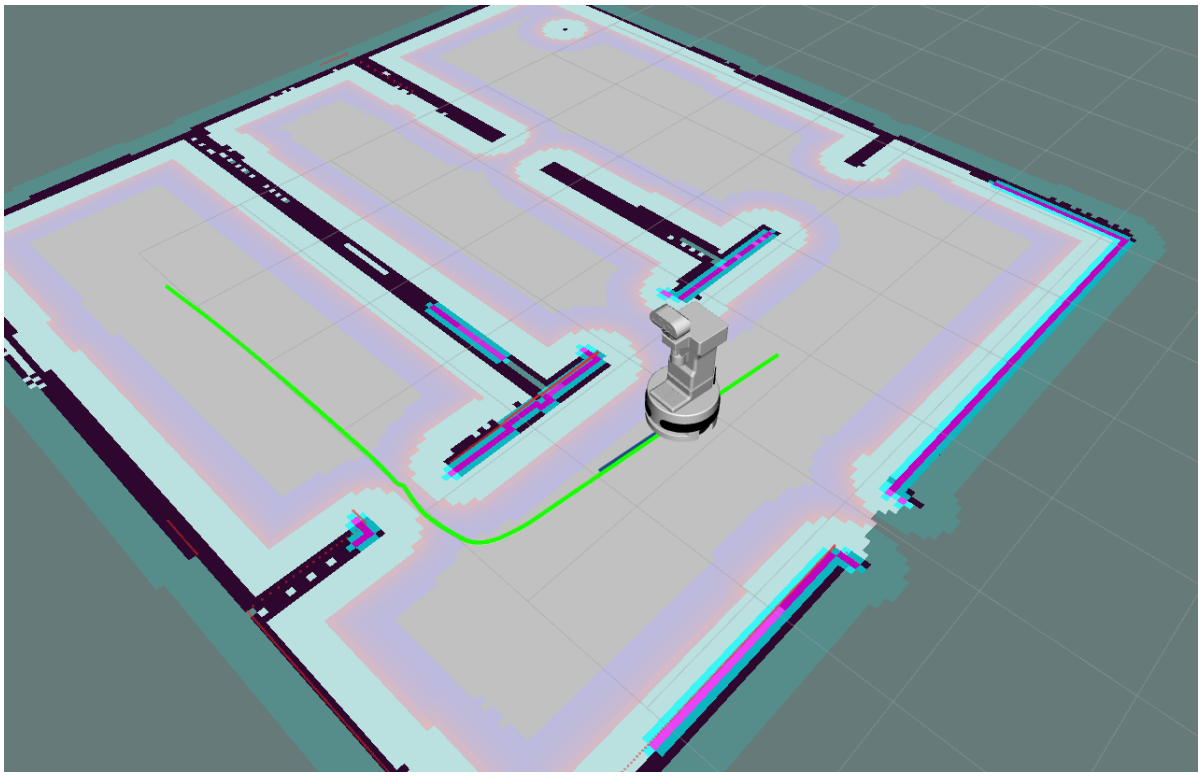
Na rys. 4.3 oraz 4.4 jest pokazana przykładowa ścieżka globalna wyznaczona przez planera globalnego.



Rys. 4.2: Zadawanie robotowi punktu docelowego na temacie */move_base_simple/goal*



Rys. 4.3: Przykładowa ścieżka globalna



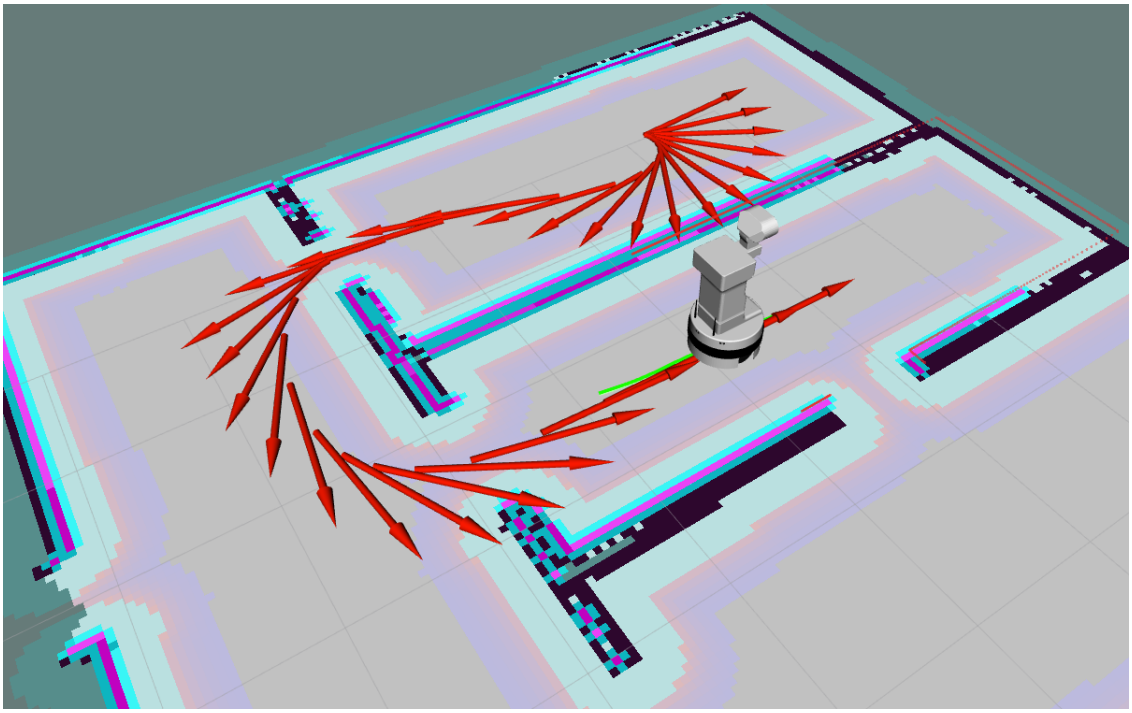
Rys. 4.4: Przykładowa ścieżka globalna

4.6. Weryfikacja działania

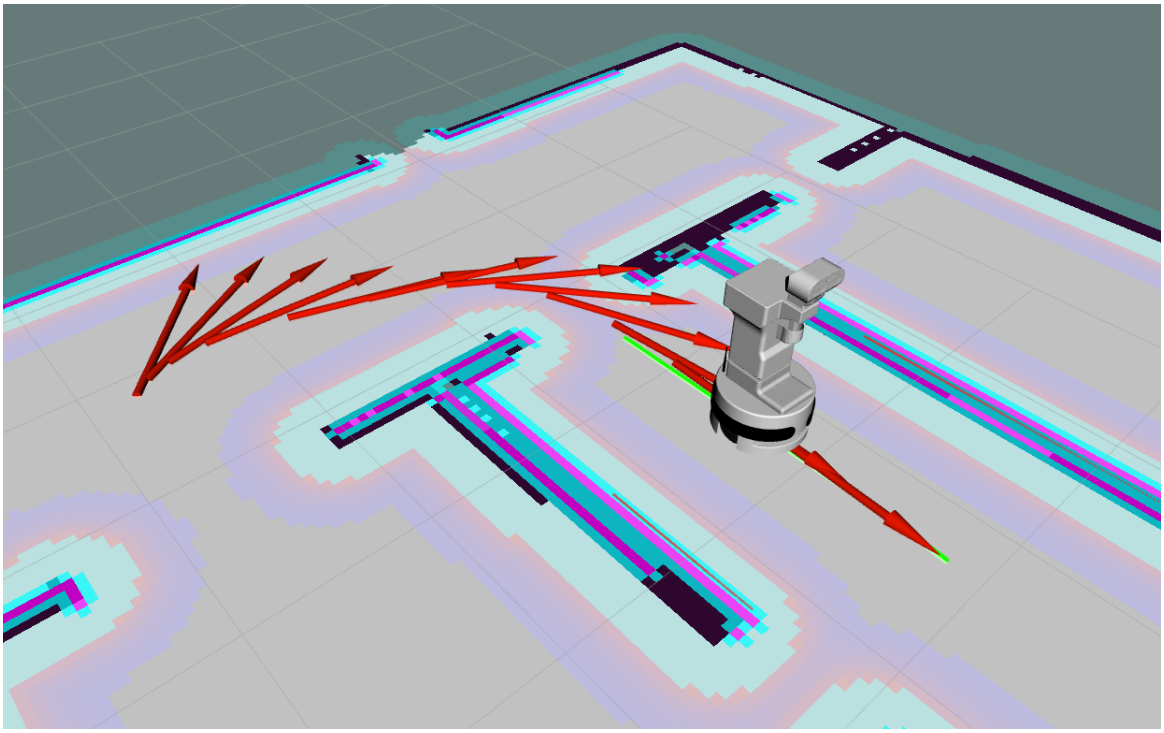
Żeby zweryfikować działanie systemu, przeprowadzono eksperymenty. Podano robotowi cel oraz obserwowano jego trajektorię. Wyniki przedstawiono na rys. 4.5 i 4.6.

Rezultat działania jest na prawdę zadowalający. Robot z sukcesem pokonuje przejścia wszystkich rozmiarów. Planer lokalny zazwyczaj wyznacza trasę nakładającą się ze ścieżką globalną. Odchodzi od tego tylko w przypadku kiedy robot musi się odwrócić.

Na laboratorium przetestowano również omijanie przeszkód. Test zakończył się sukcesem dla przeszkody w postaci cylindra jak i dla sześcianu.



Rys. 4.5: Ruch robota do wyznaczonego celu



Rys. 4.6: Ruch robota do wyznaczonego celu