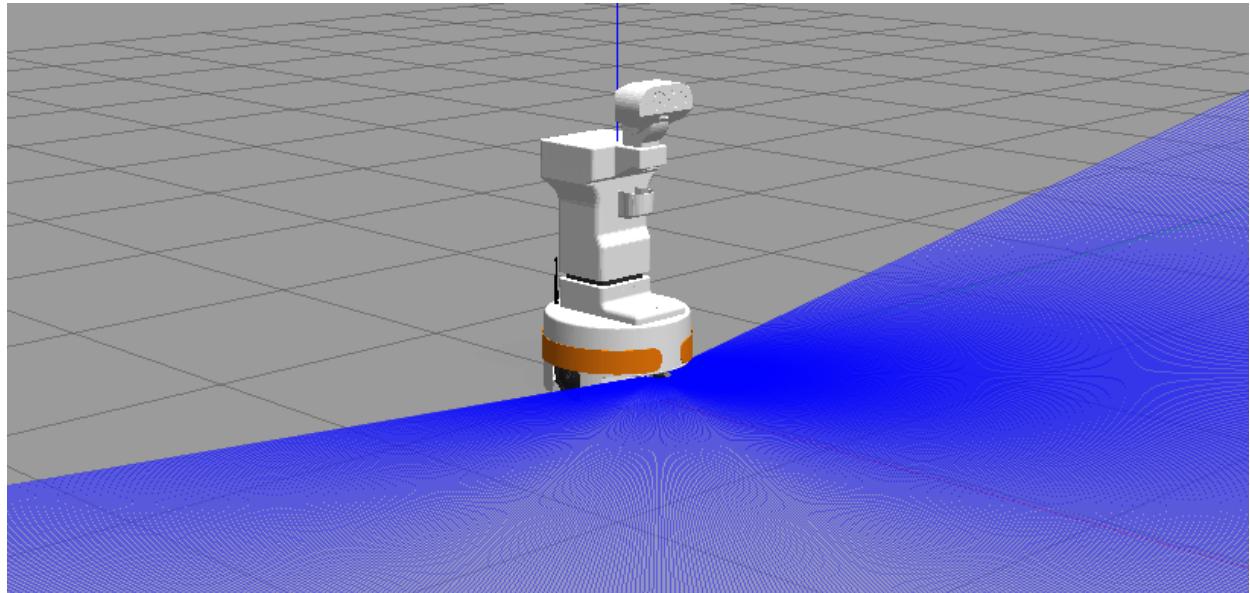


PROJEKT 1



Rysunek 1.1 Model robota Tiago w symulatorze Gazebo

1. Wstęp

Tematem pierwszego projektu wykonywanego w ramach przedmiotu STERO była implementacja względnego sterowania pozycyjnego w dwóch wariantach – w pierwszym interpolacja liniowa punktów miała zachodzić na podstawie zadawanych prędkości, a w drugim na podstawie danych odometrii. Krótko mówiąc zadanie polegało na implementacji dwóch różnych sposobów samo-lokalizacji robota. Pierwszym i bardzo prymitywnym sposobem było estymowanie pozycji jedynie na podstawie prędkości jakie zadawane były robotowi w przeszłości. W drugim wypadku robot określa swoje położenie na podstawie odometrii, a więc znajomości historii obrotów kół – w naszym zadaniu sprowadzało się to do nasłuchiwanego odpowiedniego tematu publikującego te dane. Oprócz zaimplementowania obydwu trybów jazdy należało jeszcze porównać te dane mierząc ich chwilowy i sumaryczny błąd i wyciągnąć wnioski.

Efektem naszej pracy na laboratorium pierwszym był node „*lab1*” nadający na temacie /key_vel prędkości zadawane robotowi. W zależności od jednego z czterech trybów pracy robot powinien poruszać się po kwadracie o szerokości dwóch metrów zgodnie lub przeciwnie do ruchu wskazówek zegara oraz estymować swoją pozycję zgodnie z jednym z opisanych wyżej trybów. W ramach projektu należało porównać jakość lokalizacji podczas

jazdy w jednym kierunku dla prędkości niskich, średnich i dużych zadanych przez prowadzącego.

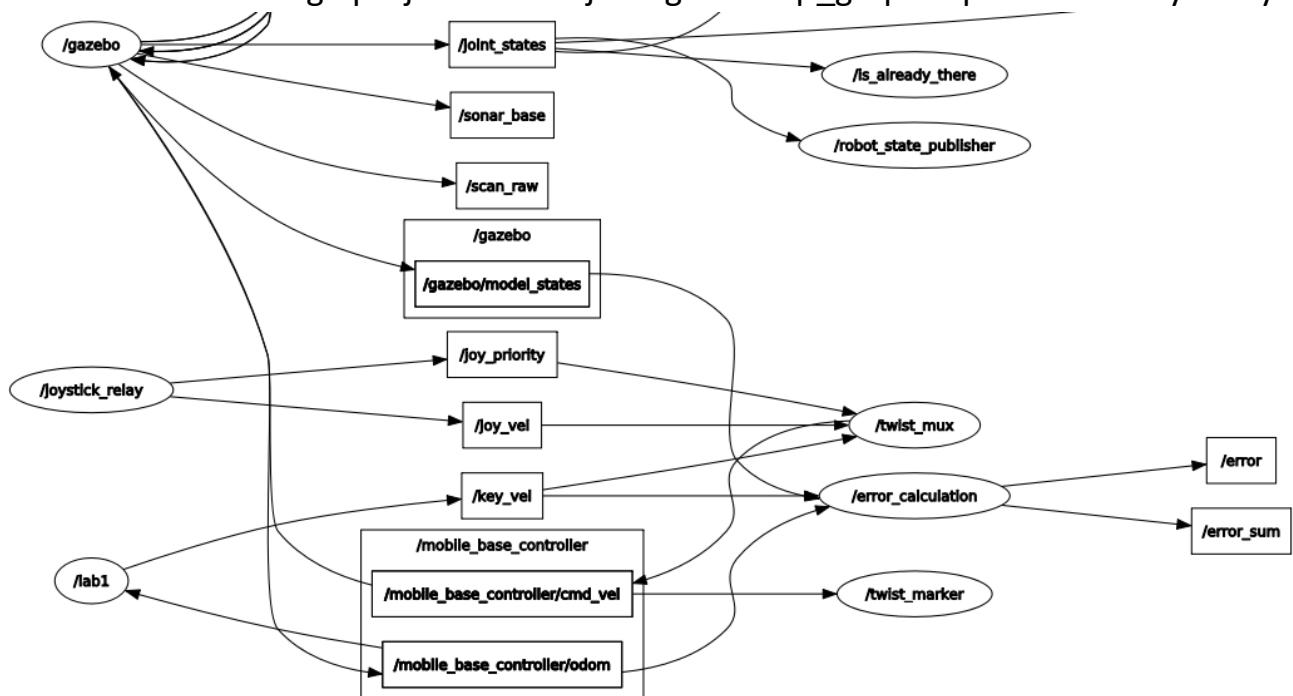
Prędkości:	Prędkość liniowa	Prędkość kątowa
Niskie	0,23421053	0,11578947
Średnie	0,562105272	0,277894728
Duże	1,3490526528	0,6669473472

Tabela 1 Prędkości liniowe i kątowe zadawane podczas eksperymentów

Stworzyliśmy więc własny node nazwany „*error_calculation*” wykonujący kilka zadań:

- Odczytywanie prędkości zadawanych na temacie */key_vel*
- Estymacja swojej pozycji jedynie na podstawie danych przekazywanych z tematu */key_vel*
- Odczytywanie danych odometrii z tematu */mobile_base_controller/odom*
- Porównywanie szacowanej lokalizacji własnej (otrzymywanej inaczej w zależności od trybu pracy) z rzeczywistą lokalizacją odczytywaną z tematu */gazebo/model_states/pose[1]/position*
- Liczenie chwilowego błędu samolokalizacji i publikowanie go na temacie */error*
- Liczenie sumarycznego błędu samolokalizacji i publikowanie go na temacie */error_sum*

Całość działania naszego projektu ilustruje fragment rqt_graph-u przedstawiony na Rys 1.2.



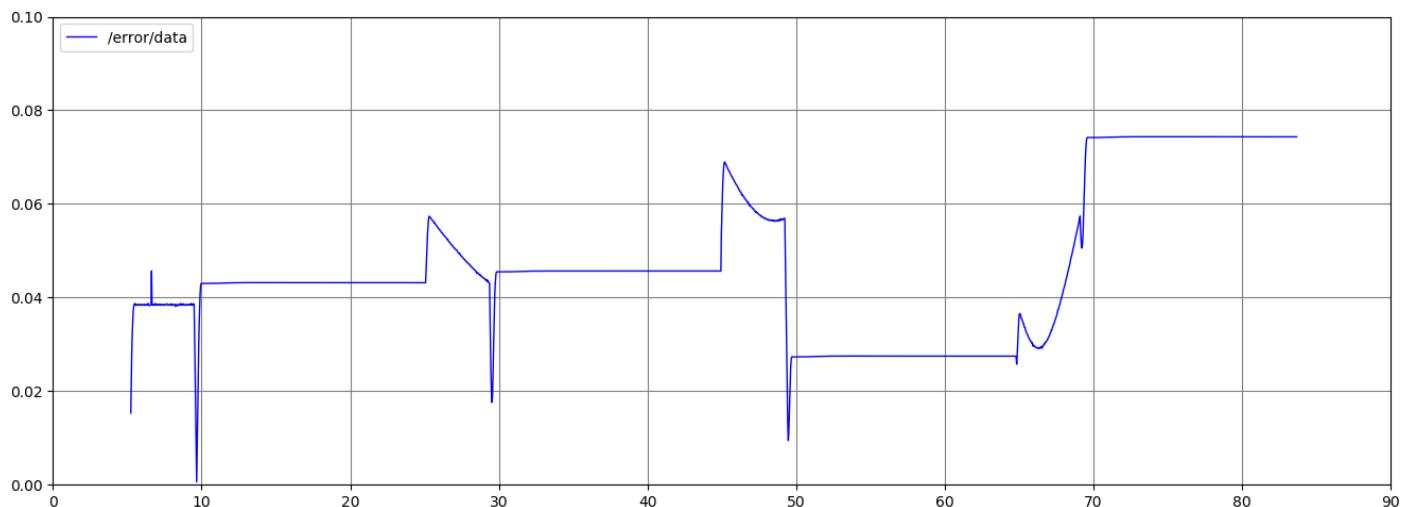
Rysunek 2 Wycinek rqt_graph'a przedstawiający współpracę stworzonych przez nas node'ów

Stworzony podczas laboratorium node „*lab1*” publikuje na temacie */key_vel* odpowiednie wartości prędkości. Node „*error_calculation*” słucha tematu */key_vel* oraz publikuje

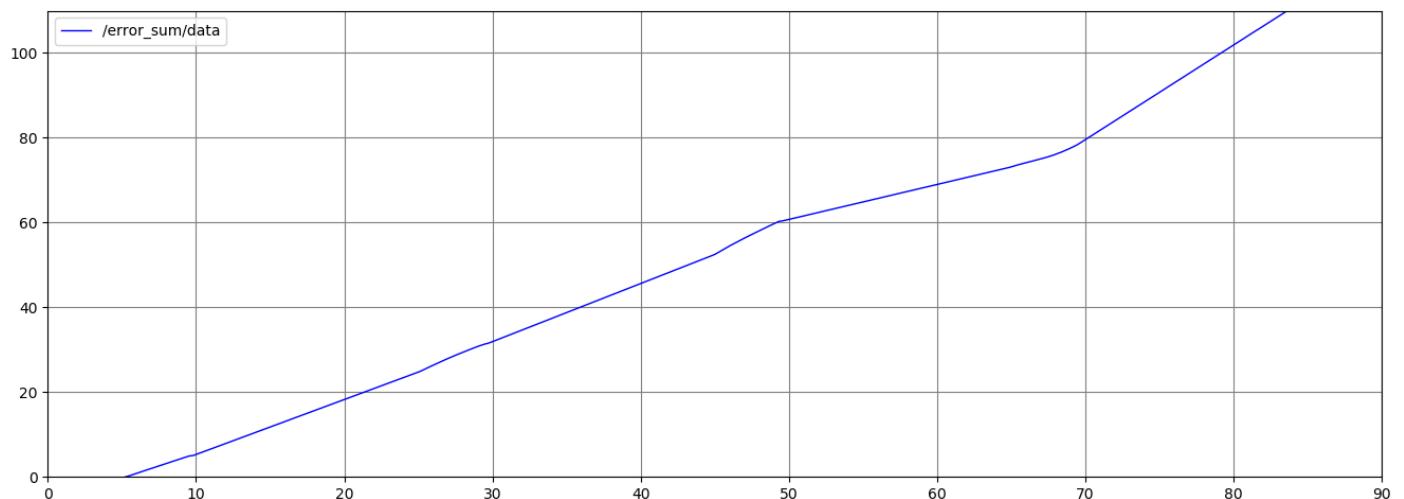
obliczone błędy na tematach które później „nasłuchujemy” za pomocą narzędzia *rqt_plot*. Wiedząc już jak działają odpowiednie node'y możemy przejść do analizy naszych prac.

2. Analiza błędów

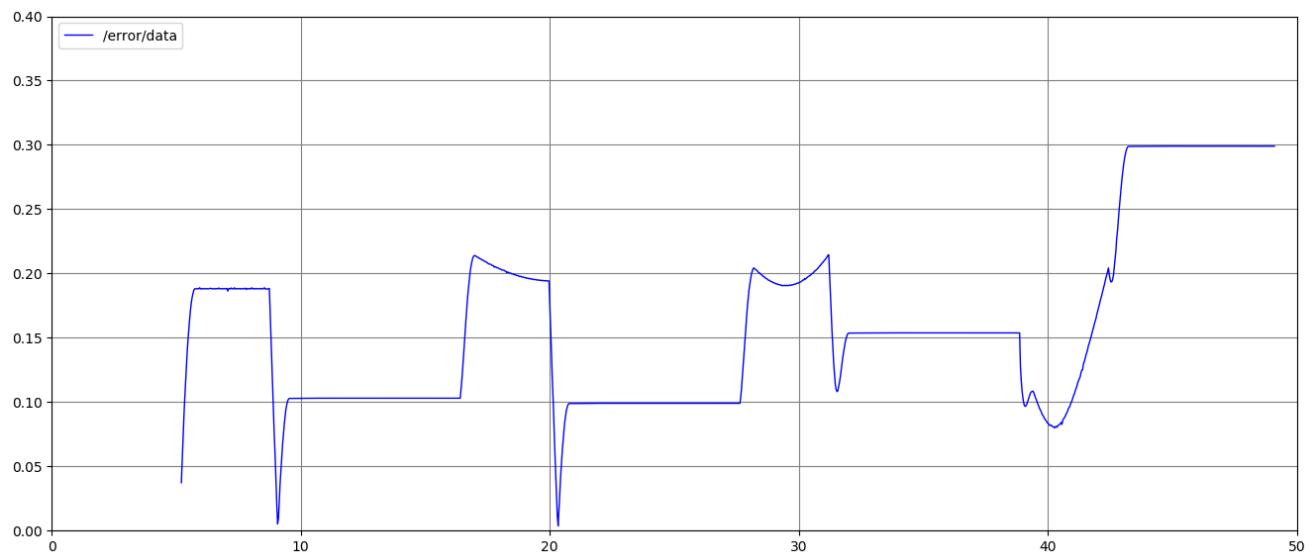
2.1 Interpolacja liniowa punktów na podstawie zadawanych prędkości



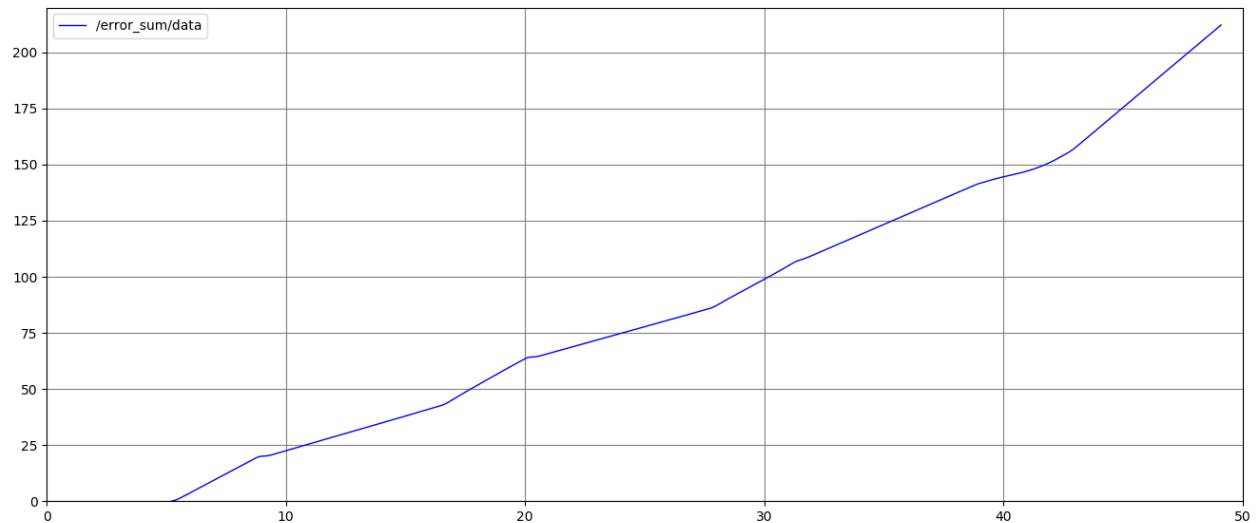
Rysunek 1.3 - Wykres chwilowego błędu estymacji na podstawie zadanych prędkości przy prędkości niskiej



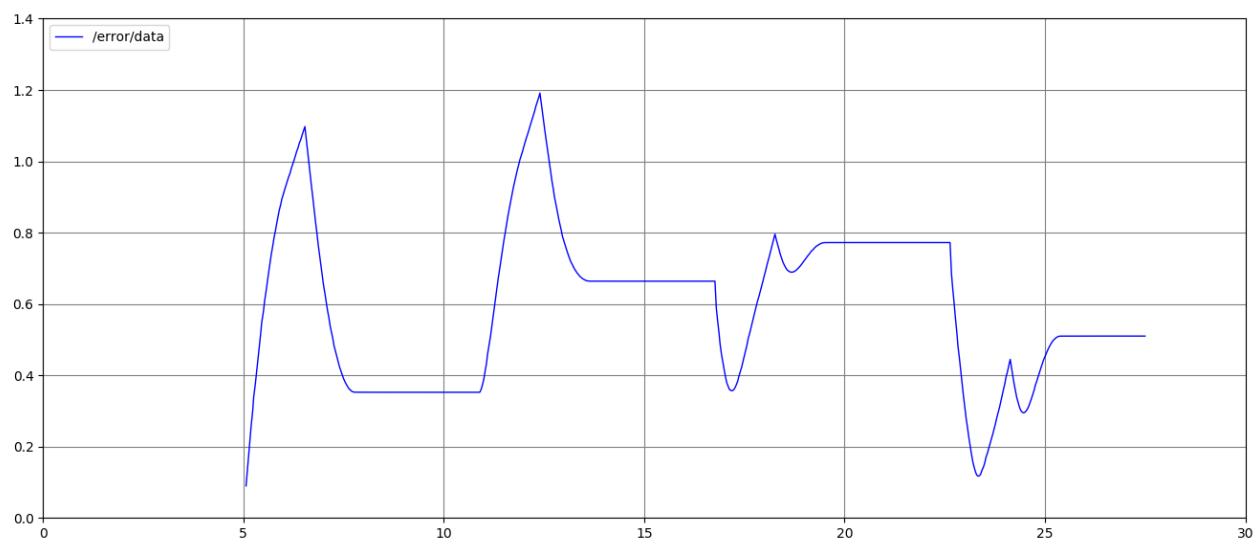
Rysunek 1.4 - Wykres sumarycznego błędu estymacji na podstawie zadanych prędkości przy prędkości niskiej



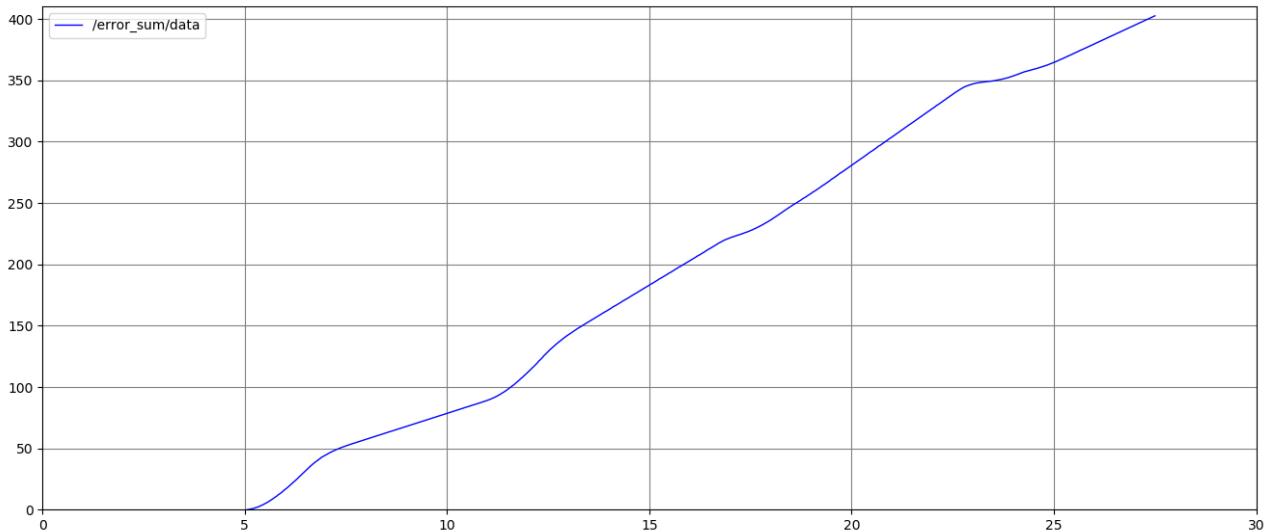
Rysunek 1.5 - Wykres chwilowego błędu estymacji na podstawie zadanych prędkości dla prędkości średniej



Rysunek 1.6 - Wykres sumarycznego błędu estymacji na podstawie zadanych prędkości dla prędkości średniej



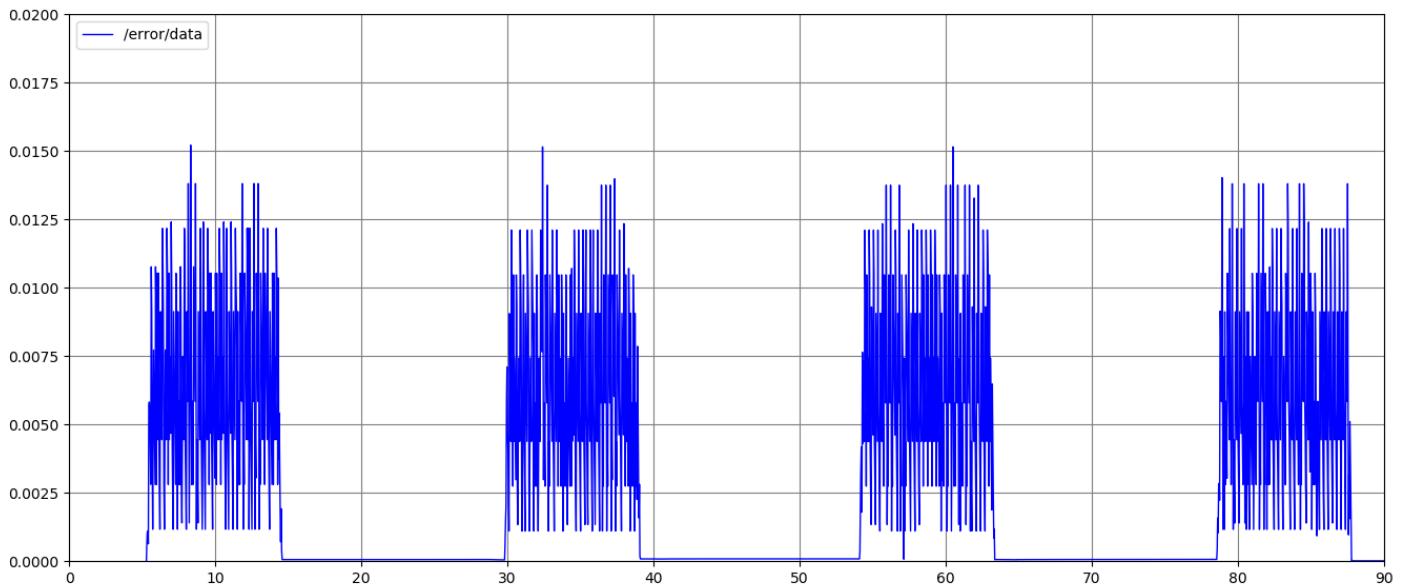
Rysunek 1.7 - Wykres chwilowego błędu estymacji na podstawie zadanych prędkości dla prędkości wysokiej



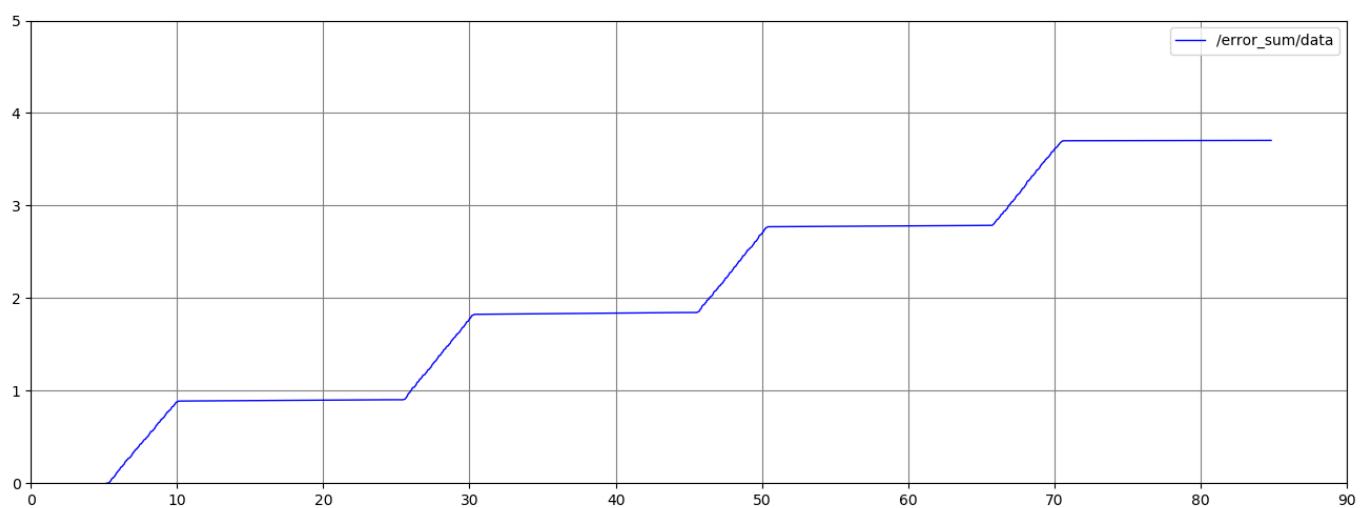
Rysunek 1.8 Wykres sumarycznego błędu estymacji na podstawie zadanych prędkości dla prędkości wysokiej

Z powyższych wykresów widzimy, że błąd estymacji, w tym trybie poruszania się, gwałtownie rośnie wraz ze wzrostem prędkości poruszania się robota. Widzimy również wzrostową tendencję błędu chwilowego, co oznacza że im dłużej robot poruszał się względem punktu początkowego tym estymacja jego pozycji była gorsza. Na każdym z wykresów prędkości chwilowych widzimy okresy wypłaszczeń i zmian wartości błędu. Ruch robota (jako ruch po kwadracie) składa się z obrotów i ruchu wzdłuż jednej współrzędnej. Podczas ruchu liniowego pozycje estymowana i rzeczywista mogą przemieszczać się względem siebie, a więc wartość błędu chwilowego może się zmieniać, natomiast podczas obrotu pozycje te „zastygają” w związku z czym pozycja między nimi się nie zmienia i błąd liczony jako wartość bezwzględna odległości między nimi pozostaje stały.

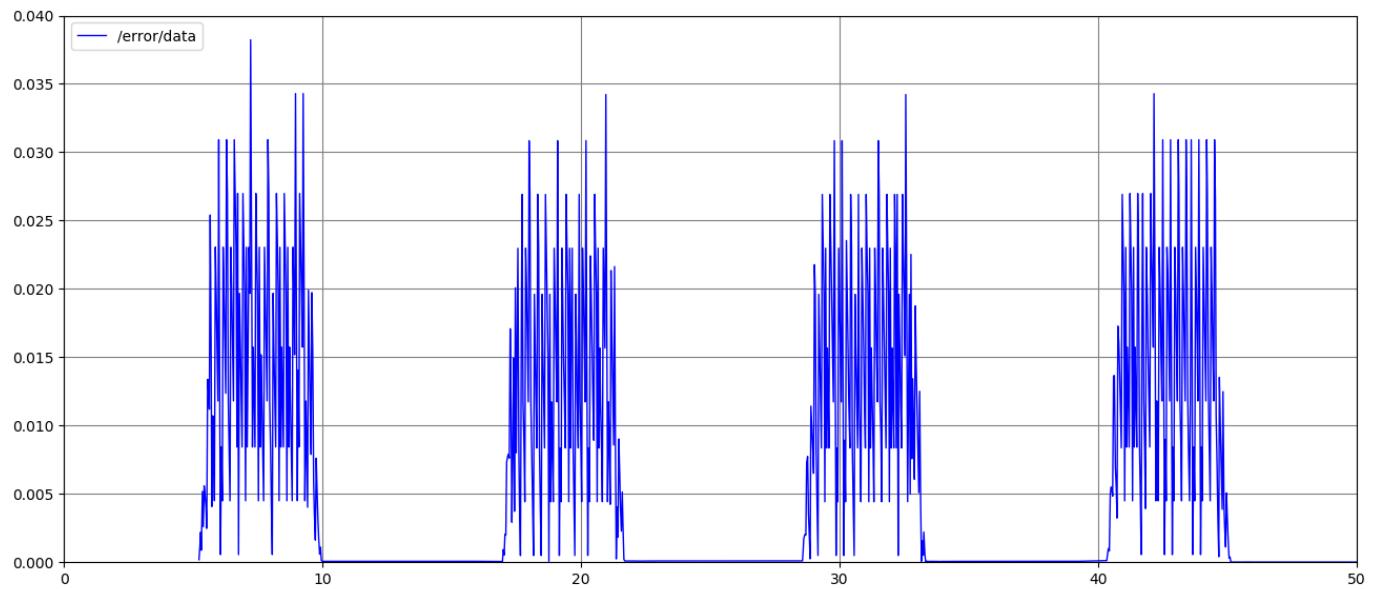
2.1 Interpolacja liniowa punktów na podstawie odometrii



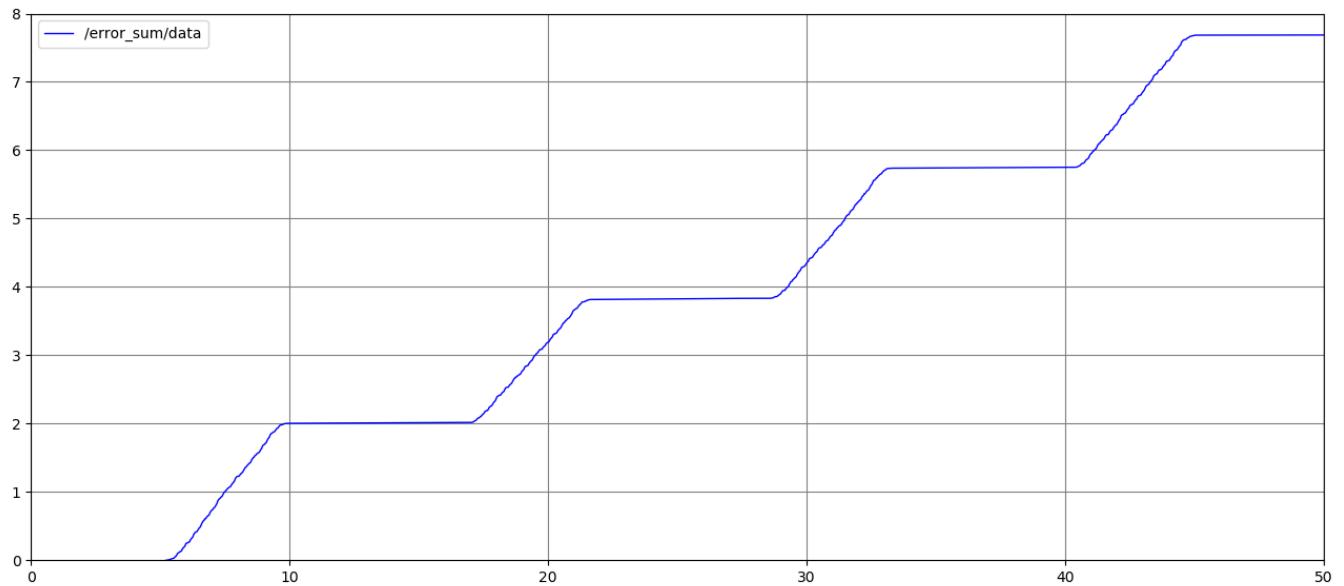
Rysunek 1.9 - Wykres chwilowego błędu estymacji na podstawie odometrii dla prędkości niskiej



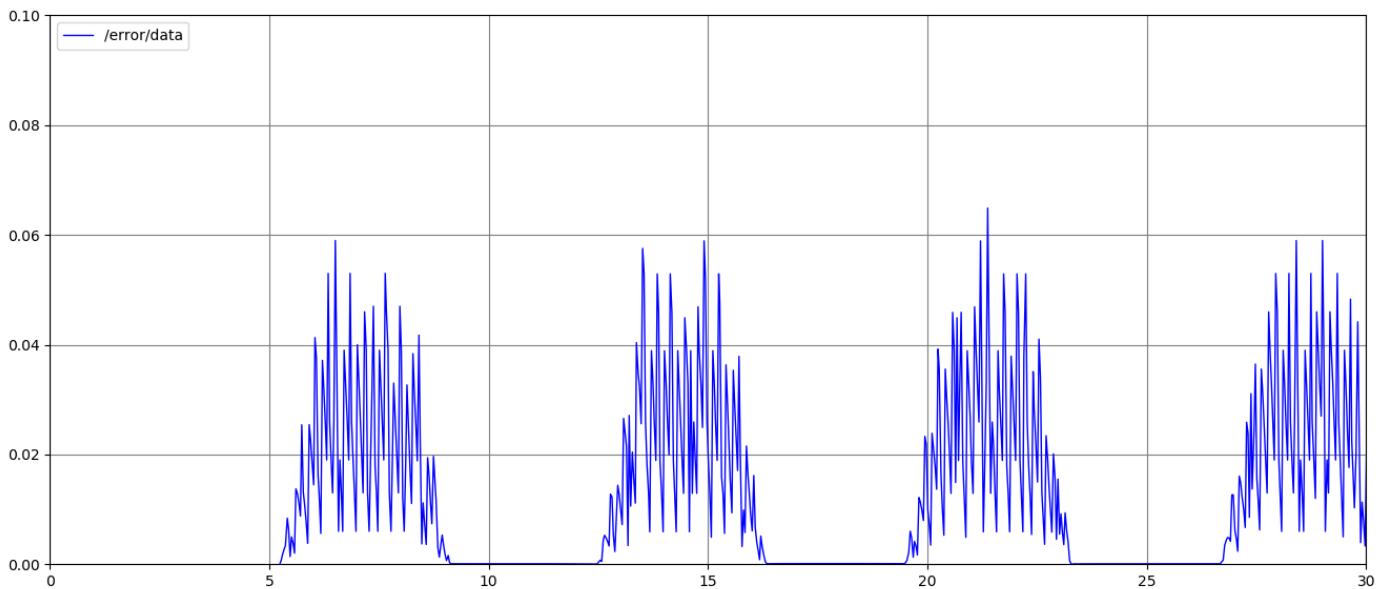
Rysunek 1.10 - Wykres sumarycznego błędu estymacji na podstawie odometrii dla prędkości niskiej



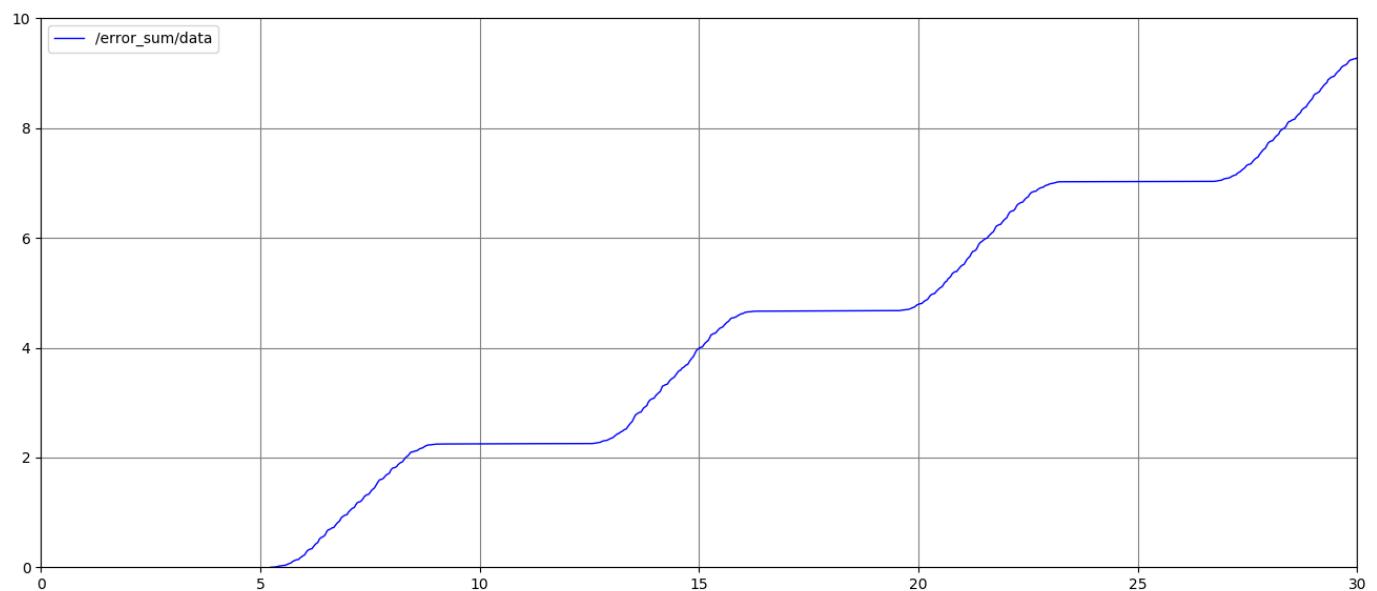
Rysunek 1.11 - Wykres chwilowego błędu estymacji na podstawie odometrii dla prędkości średniej



Rysunek 1.12 - Wykres sumarycznego błędu estymacji na podstawie odometrii dla prędkości średniej



Rysunek 1.13 - Wykres chwilowego błędu estymacji na podstawie odometrii dla prędkości wysokiej

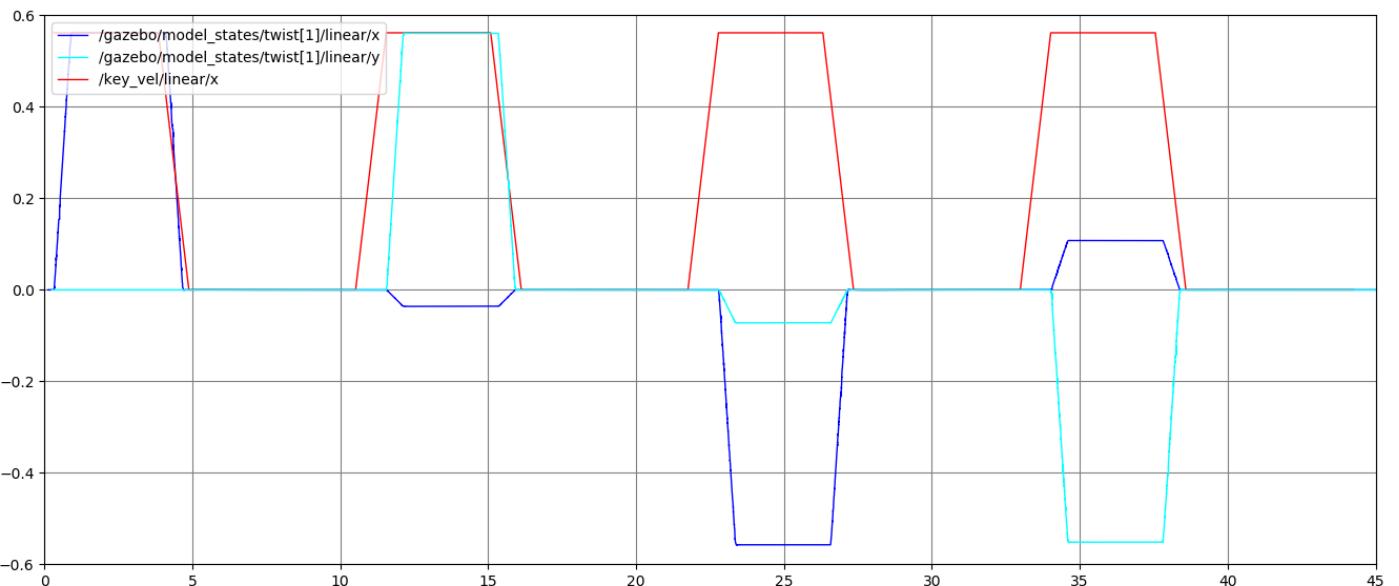


Rysunek 1.14 - Wykres sumarycznego błędu estymacji na podstawie odometrii dla prędkości wysokiej

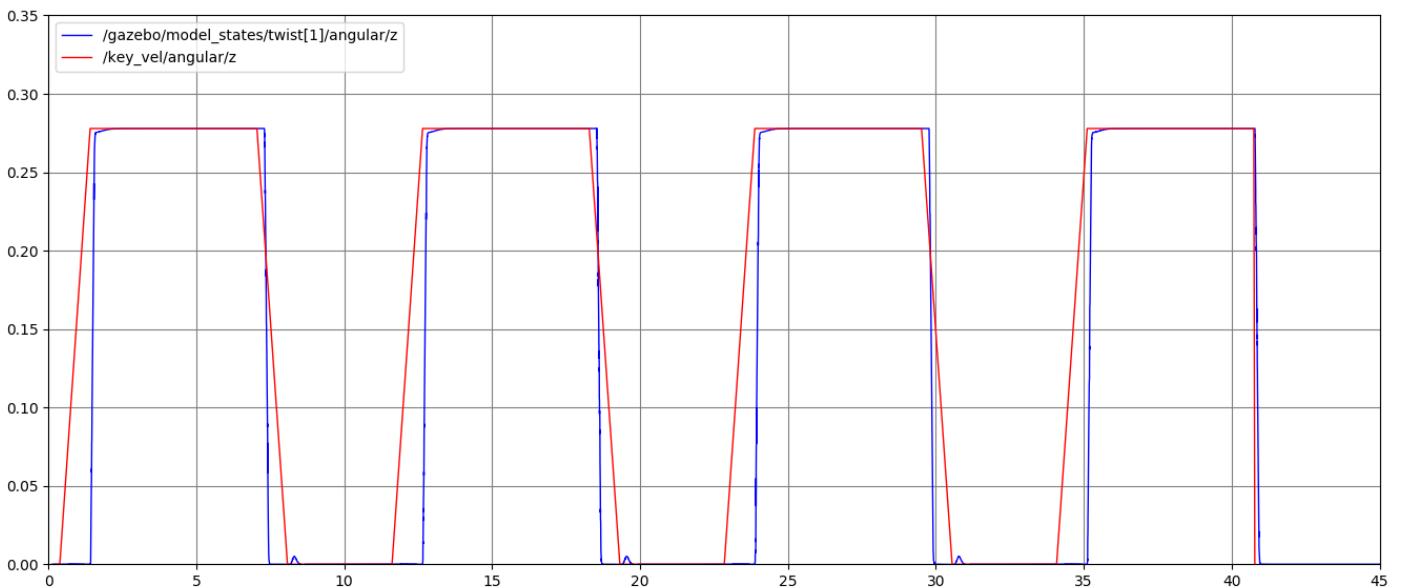
Tak jak w przypadku poprzedniego trybu pracy wraz ze wzrostem prędkości obserwujemy wzrost sumarycznego błędu. Tym razem jednak błąd ten nie rośnie tak gwałtownie, a przy przeskoku z prędkości średniej do wysokiej prawie się nie zmienia. Tak jak dla poprzedniego trybu widzimy wyraźnie momenty wypłaszczeń, różnica jest tylko taka, że przy lokalizacji na podstawie odometrii błąd dla tych chwil jest praktycznie zerowy. Oznacza to, że po zatrzymaniu pozycja rzeczywista i estymowana robota jest praktycznie identyczna.

3. Analiza prędkości zadanych/wykonanych

Oprócz analizy błędów warto jeszcze przyjrzeć się wykresom przedstawiającym przebiegi prędkości zadanej odczytywanej z tematu `/key_vel` i rzeczywistej odczytywanej z tematu `/gazebo/model_states`. Przykładowo dla Interpolacji liniowej punktów na podstawie zadawanych prędkości i prędkości drugiej wykresy te wyglądają następująco:



Rysunek 1.15 - Wykres przedstawiający rzeczywiste i zadane prędkości liniowe dla pierwszego trybu pracy i prędkości drugiej

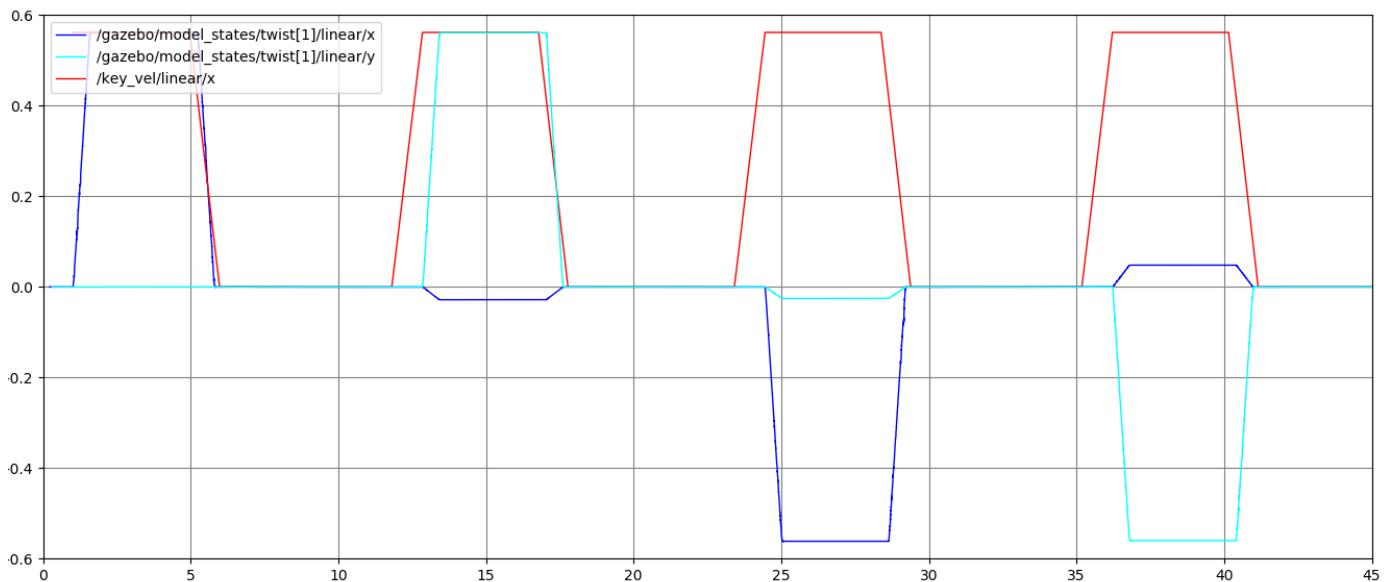


Rysunek 1.16 - Wykres przedstawiający rzeczywiste i zadane prędkości kątowe dla pierwszego trybu pracy i prędkości drugiej

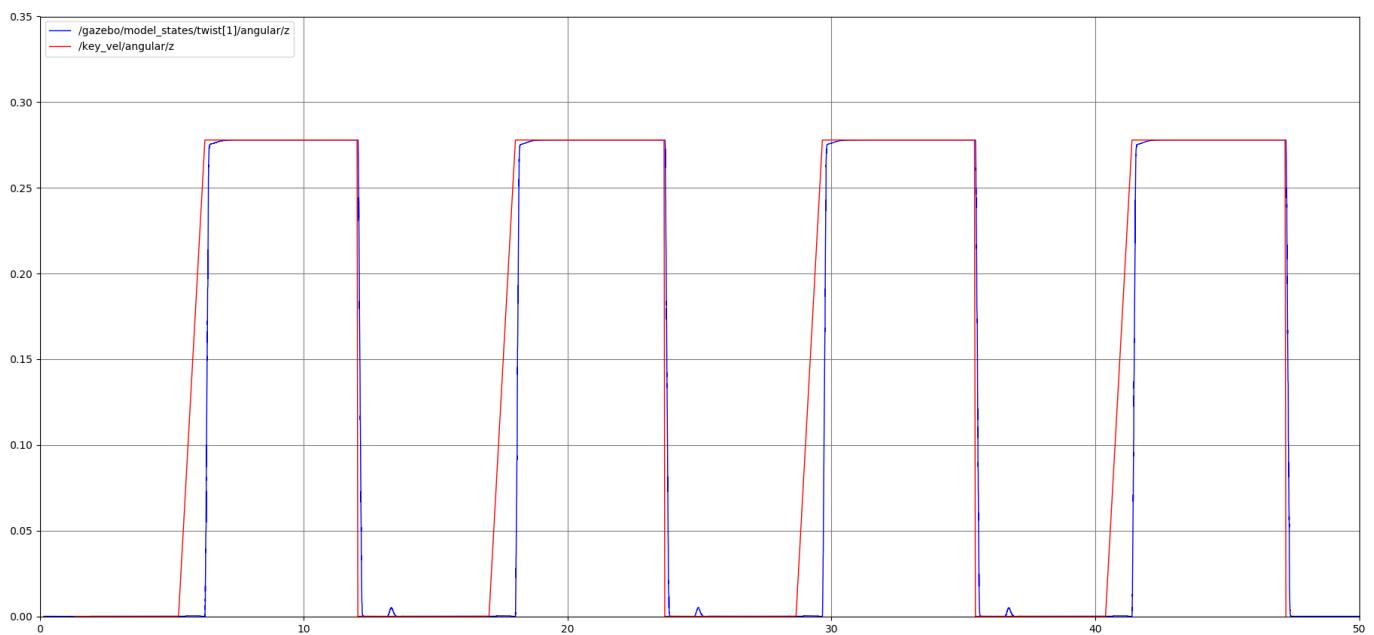
Wykres prędkości liniowych składa się z następujących po sobie momentów ruchu „na wprost” oraz momentów wykonywania obrotu. W miejscach gdzie prędkości są zerowe tiago obraca się. Analizując ten wykres możemy wywnioskować że w miejscu pierwszego „piku” błędna może być jedynie pozycja x w której robot zaczyna obrót ponieważ prędkość liniowa y jest zerowa. Następnie możemy zauważać że robot wykonał nieprecyzyjny obrót –

wartość kąta obrotu była większa niż $\pi/2$ co objawia się ujemną prędkością liniową wzdłuż osi x. Dwa ostatnie „piki” świadczą o ruchu kolejno wzdłuż osi x w kierunku przeciwnym do początkowego i o ruchu wzdłuż osi y przeciwnie do zwrotu osi. Ponadto obserwujemy kumulowanie się błędów – z każdym kolejnym obrotem moduł wartości prędkości względem tej osi wzdłuż której robot w idealnym przypadku nie powinien się poruszać jest coraz większy.

Natomiast dla Interpolacji liniowej punktów na podstawie zadawanych prędkości dla prędkości 2 wykres prędkości wyglądają następująco:



Rysunek 1.17 - Wykres przedstawiający rzeczywiste i zadane prędkości liniowe dla drugiego (odometrycznego) trybu pracy i prędkości drugiej



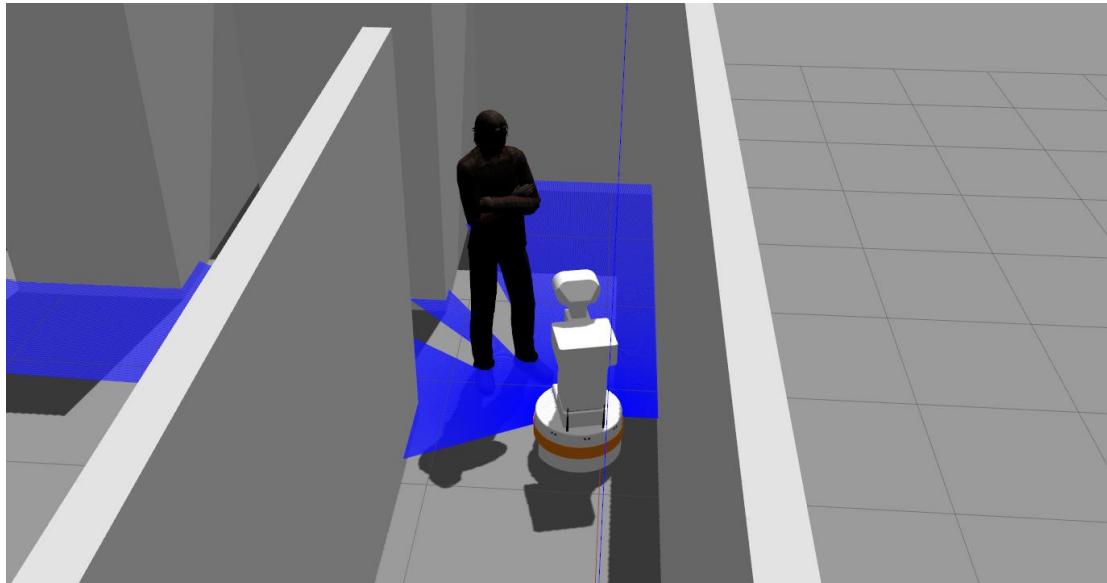
Rysunek 1.18 - Wykres przedstawiający rzeczywiste i zadane prędkości liniowe dla drugiego (odometrycznego) trybu pracy i prędkości drugiej

Porównując wykresy prędkości liniowych dla prędkości drugiej pracy można stwierdzić że błędy w trybie odometrii są mniejsze - prędkości w kierunkach prostopadłych do tego w których robot aktualnie się porusza są znacznie mniejsze, przez co mniejszy jest również błąd sumaryczny. Analizując wykresy prędkości kątowych nie zauważamy znaczących różnic między dwoma trybami ruchu. Widzimy jednak, że zawsze wzrost prędkości nadawanej na temacie /key_vel poprzedza wzrost prędkości rzeczywistej co jest zgodne z oczekiwaniami. Zadziwia natomiast tak gwałtowny wzrost rzeczywistej prędkości obrotowej. Najwidoczniej, prędkość ta jest na tyle mała, że przyspieszanie trwające ułamek sekundy wygląda jak skok jednostkowy. Niemniej spodziewaliśmy się zaobserwować skokowy wzrost prędkości na kanale /key_vel i nieco stopniowy wzrost rzeczywistej prędkości, więc wykresy nieco nas zaskoczyły.

4. Wnioski

Analizując wyniki interpolacji liniowej punktów na podstawie zadawanych prędkości oraz na podstawie danych odometrycznych możemy stwierdzić że odometria daje bardziej szczegółową estymację pozycji robota. Używając odometrii jako sposobu samo-lokalizacji robota otrzymujemy dużo dokładniejszą estymację pozycji w jakiej robot się znajduje - błędy lokalizacji są rzędu wielkości mniejsze od błędów lokalizacji podczas estymacji pozycji na podstawie prędkości.

PROJEKT 2



Rysunek 2.1 Model robota Tiago napotykającego przeszkodę w symulatorze Gazebo

1. Wstęp

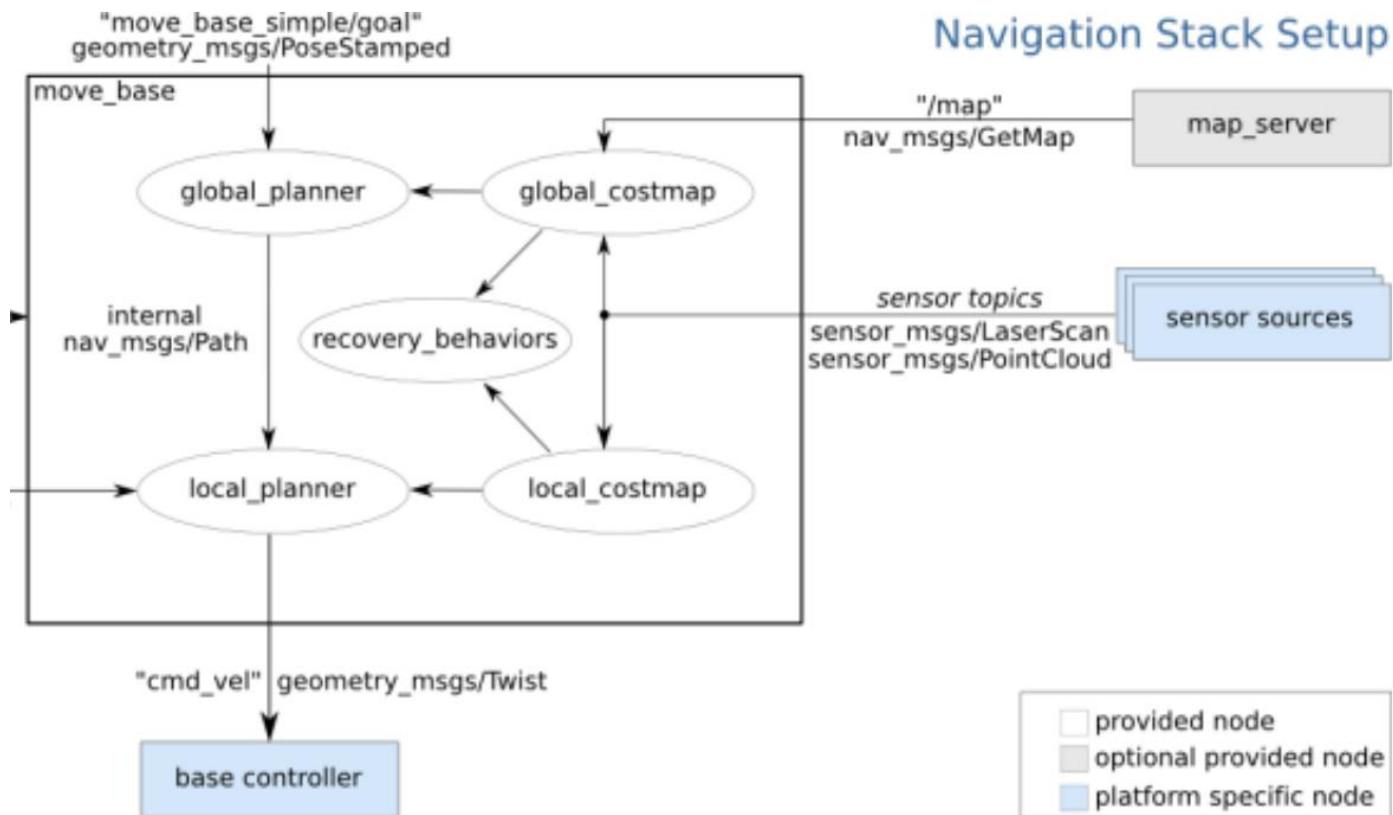
Drugim projektem wykonywanym w ramach przedmiotu STERO była implementacja systemu nawigacji robota złożona z kilku etapów. Stworzony przez nas node powinien, wykorzystując zewnętrzne biblioteki, wygenerować trajektorię prowadzącą do zadanego punktu, a następnie przy użyciu lokalnego i globalnego planera przeprowadzić robota do celu zadając mu odpowiednie prędkości na temacie `/key_vel`. Dodatkowo robot powinien wykryć i odpowiednio zareagować na pojawienie się niespodziewanych przeszkód na trasie i ciągle informować o chwilowym statusie pokonywania trasy.

2. Implementacja

Struktura współpracy poszczególnych elementów stworzonego przez nas node'a „`my_move_base`” przedstawiona jest na rysunku 2.2. Podstawą jest planowanie trajektorii prowadzącej do punktu końcowego opartej o globalną i statyczną mapę świata. Mapa ta, stworzona podczas laboratorium 2, publikowana jest przez `map_server` i na jej podstawie tworzony jest obiekt klasy `costmap_2d::Costmap2DROS`. Obiekt ten staje się więc globalną mapą kosztów, którą planer globalny „`my_global_planner`” wykorzystuje do stworzenia trajektorii. Tak przynajmniej powinno to wyglądać, bo zarówno globalna mapa kosztów jak i trajektoria tworzone są w zależności od dużej liczby dostrajanych parametrów, od których doboru zależy jakość ostatecznej trasy robota. W przypadku globalnej mapy kosztów najważniejszymi parametrami są:

- *Static_map: true* – globalna mapa kosztów powinna być statyczna
- *Inflation_layer/inflation_radius: 0,75* - promień inflacyjnego „powiększania” przeszkód
- *Inflation_layer/cost_scaling_factor: 4* – współczynnik kosztu w obrębie „pola inflacji” – im większy tym bardziej planer globalny będzie unikał wyznaczania trajektorii przez „pole inflacji”.

Wartości *inflation_radius* i *cost_scaling_factor* wyznaczyliśmy analizując trajektorie generowane przez planer globalny. W przypadku global planera ważnym parametrem jest *use_dijkstra: false* – ustawienie tego parametru na *false* zmusza planer do użycia algorytmu A* przy wyliczaniu trajektorii – zgodnie z treścią zadania.



Rysunek 2.2 Struktura pracy poszczególnych modułów "my_move_base"

W idealnym i statycznym świecie dwa powyższe moduły powinny być w stanie prowadzić robota. Ponieważ jednak świat, a nawet jego ogromnie uproszczony model stworzony podczas laboratorium 2, nie jest idealnym miejscem, a na domiar złego pojawiają się w nim nieprzewidziane przeszkody, konieczne jest również użycie planera radzącego sobie z lokalnymi przeszkodami występującymi na ścieżce – planera lokalnego.

Użyty przez nas lokalny planer to *base_local_planer*. Działa on w oparciu o stworzoną ścieżkę globalną oraz lokalną mapę kosztów wygenerowaną przy pomocy czujnika laserowego, który na bieżąco, podczas działania *Tiago* skanuje otoczenie i wykrywa przeszkody, które następnie są nanoszone na lokalną mapę kosztów. Oprócz tego planer ten odpowiada za generowanie chwilowej trajektorii wraz z chwilowymi sygnałami prędkości z których korzysta *Tiago*, zatem w jego parametrach należy określić m.in limity prędkości i przyśpieszeń.

Lokalna mapa kosztów to bardzo ważny element działania lokalnego planera – to na nią nanoszone są wszystkie przeszkody nieuwzględnione w globalnej mapie. Dzięki niej jest również możliwe uniknięcie statycznych przeszkód np. ścian, w przypadku gdy mapa globalna obarczona jest błędem. Najważniejszymi parametrami mapy lokalnej, które najbardziej wpływając na jakość sterowania i wyznaczenia ścieżki:

- *Static_map*: false oraz *rolling_window*: true – powoduje że mapa lokalna przemieszcza się wraz z robotem, nie jest statyczna.
- *Width*: 5 oraz *height*: 5 – rozmiar mapy lokalnej, im większy tym *Tiago* posiada więcej informacji o lokalnym otoczeniu – mapa lokalna obejmuje większy obszar, zawiera więcej informacji.
- *Inflation_layer/Infaltion_radius*: 0.25 - określa “powiększenie” przeszkody
- *Inflation_layer/cost_scaling factor*: 12 - współczynnik zwiększający wartość kosztu.

Parametry *cost_scaling_factor* oraz *inflation_radius* wyznaczyliśmy na podstawie kilkunastu prób z różnymi wartościami. Intuicyjnie spodziewaliśmy się, że *inflation_radius* mapy lokalnej powinien być mniejszy niż mapy globalnej. Trajektoria powinna być generowana w miarę możliwości daleko od wszelkiego rodzaju ścian i przeszkód, stąd większe „rozepchanie” tych przeszkód przez większy *inflation_radius* globalnej mapy kosztów. Podobna analiza doprowadziła nas do wniosku, że *cost_scaling factor* powinien być znacznie większy w przypadku mapy lokalnej. To ona zaznacza na sobie przeszkody które rzeczywiście otaczają robota, więc to przeszkód na niej zaznaczonych robot powinien się bardziej „bać”.

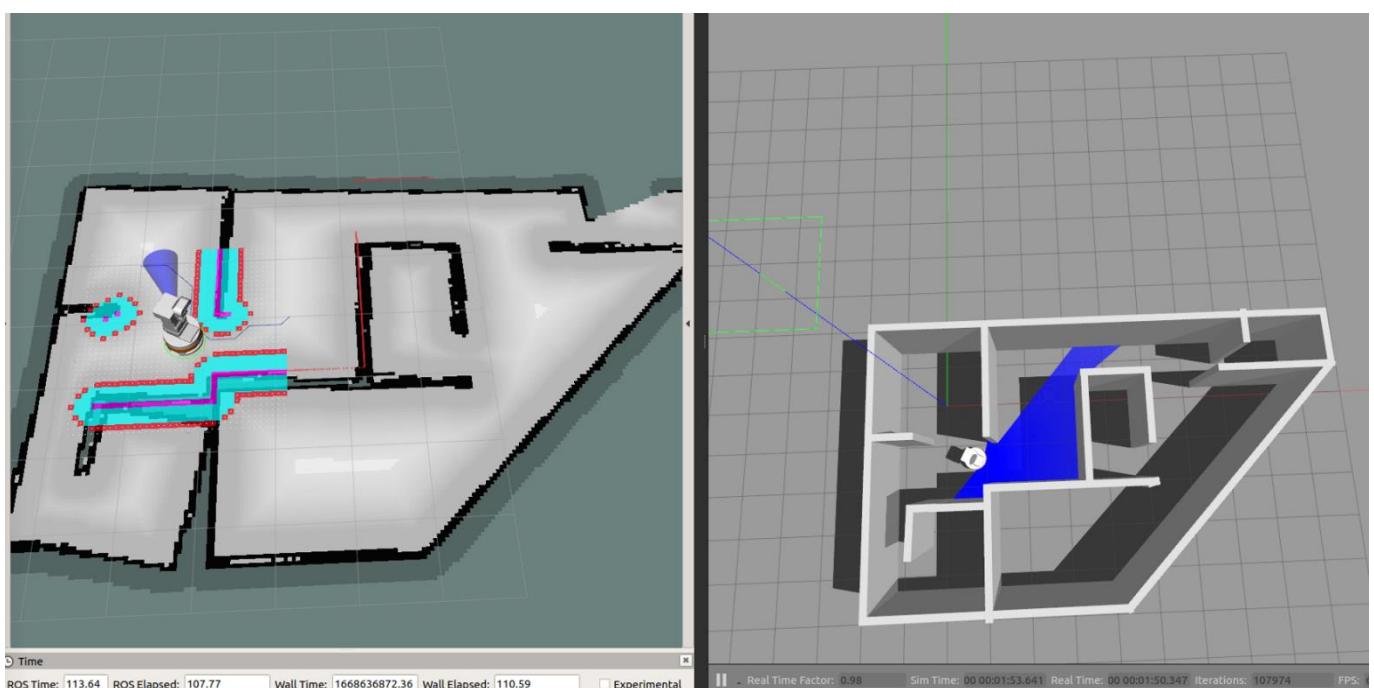
Kolejnym elementem działania node-a *my_move_base* jest recovery behavior. Jest to kluczowy element w przypadku zakleszczenia *Tiago* przez przeszkody ze świata zewnętrznego. Powoduje obrót robota wokół własnej osi i „czyszczenie mapy lokalnej” do momentu zniknięcia przeszkody.

Całość działania stworzonego przez nas programu opiera się na nieskomplikowanym automacie stanowym wykonujące kolejno operacje stworzenia globalnej mapy i planu, lokalnej mapy i planu, wyznaczenia prędkości pozwalających na ruch wzdłuż wyliczonej

trajektorii itp.. W każdym stanie na topick-u `/nav_info` publikowana jest informacja o stanie tiago.

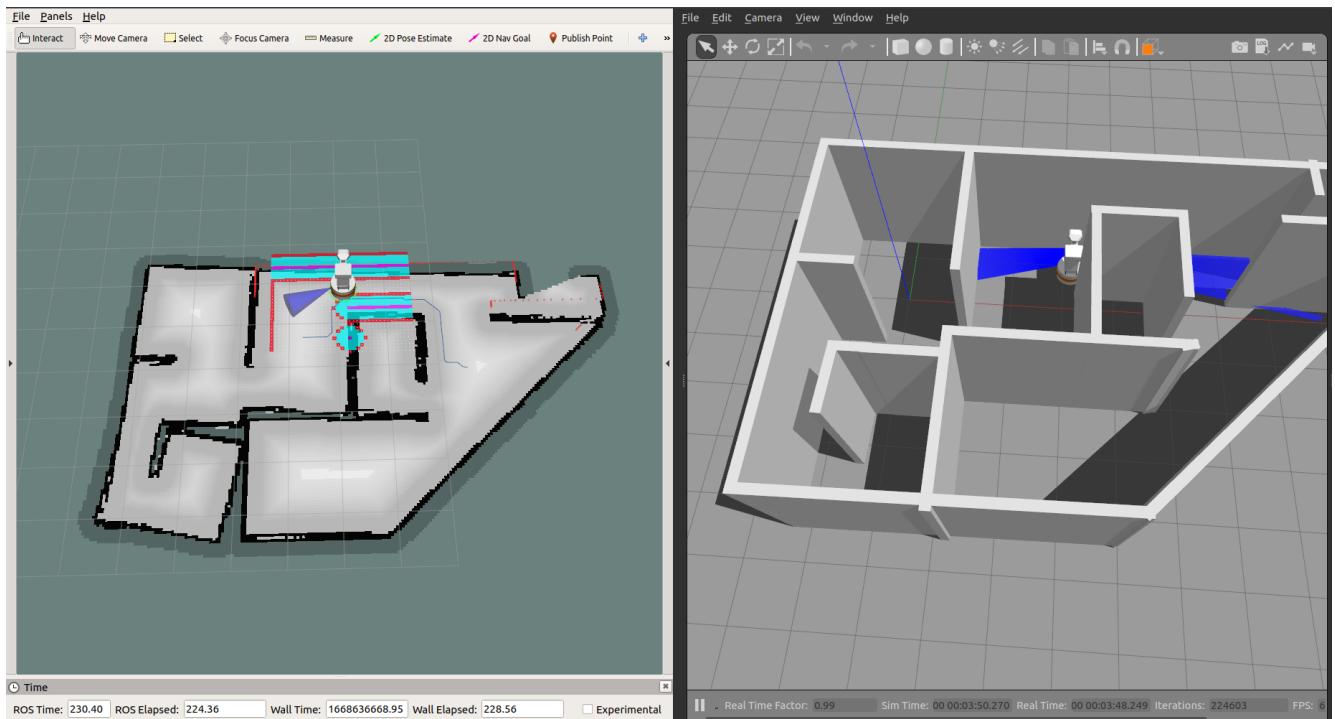
3. Pierwsze próby

Gdy zaimplementowany przez nas node był na tyle sprawny by jakkolwiek przeprowadzić robota od miejsca startu do celu rozpoczęliśmy próby działania systemu w świecie stworzonym przez nas na laboratorium 2. Od początku obserwowałyśmy wiele problemów z jazdą robota, wydawało nam się jednak, że winnym tego zachowania jest sam algorytm lub parametry poszczególnych jego modułów. Niestety wraz z rozwojem algorytmu zorientowaliśmy się, że największe problemy stwarza samo środowisko w jakim pracuje Tiago.



Rysunek 2.3 Robot Tiago próbujący wyjechać z miejsca startowego

Stworzony przez nas model świata jest bardzo ciasnym miejscem w którym tuż po wyjeździe z miejsca startowego model zmuszony jest pokonywania wąskich przejść i korytarzy. Pierwsze takie miejsce przedstawione jest na Rysunku 2.3. Korytarz prowadzący do pokoju “południowego” jest zbyt wąski w związku z czym jedyną drogą robota do reszty “świata” jest wąski, północny korytarz. Przy odpowiednim doborze parametrów Tiago radzi sobie z przejściem przedstawionym na Rysunku 2.3, jednak kolejny wąski korytarz okazuje się być dla biednego Tiago korytarzem śmierci. Ponadto zauważaliśmy sporą rozbieżność między modelem świata a stworzoną mapą, szczególnie w wyżej wspomnianym korytarzu, przez co robot często wjeżdżała w ścianę, mimo iż na mapie w Rviz miał sporo miejsca na przejazd.



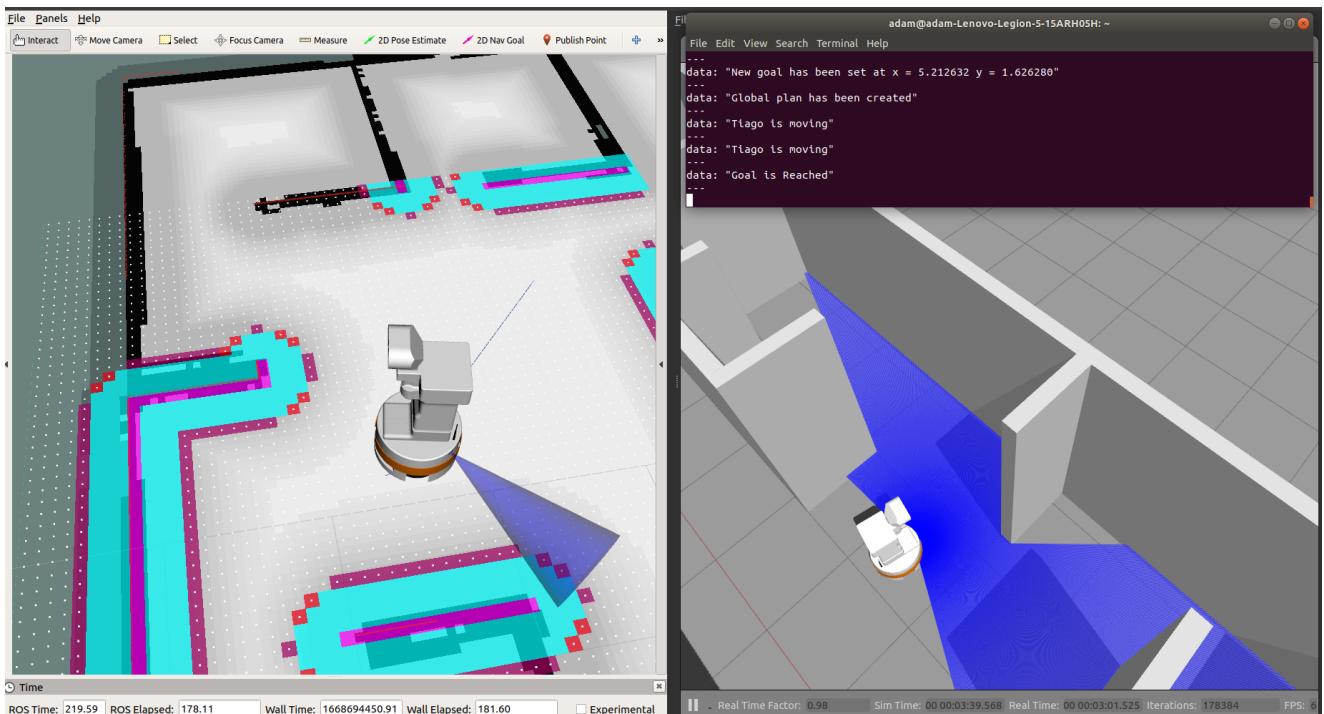
Rysunek 2.4 Robot Tiago przy wjeździe do Korytarza Śmierci

Ze względu na powyższe problemy stworzony przez nas świat nie nadawał się do dostrajania parametrów i dalszych prób algorytmu. W szczególności z powodu na ograniczone miejsce ciężko w ogóle myśleć o wstawianiu dodatkowych, nieuwzględnionych w mapie globalnej, przeszkód. W związku z tym zdecydowaliśmy się na zmianę środowiska w którym porusza się Tiago. Do dalszych testów wykorzystywaliśmy, oczywiście po otrzymaniu zgody, świat naszych kolegów z grupy Michała Kwarcińskiego i Kacpra Marchlewicza.

4. Demonstracja pracy robota

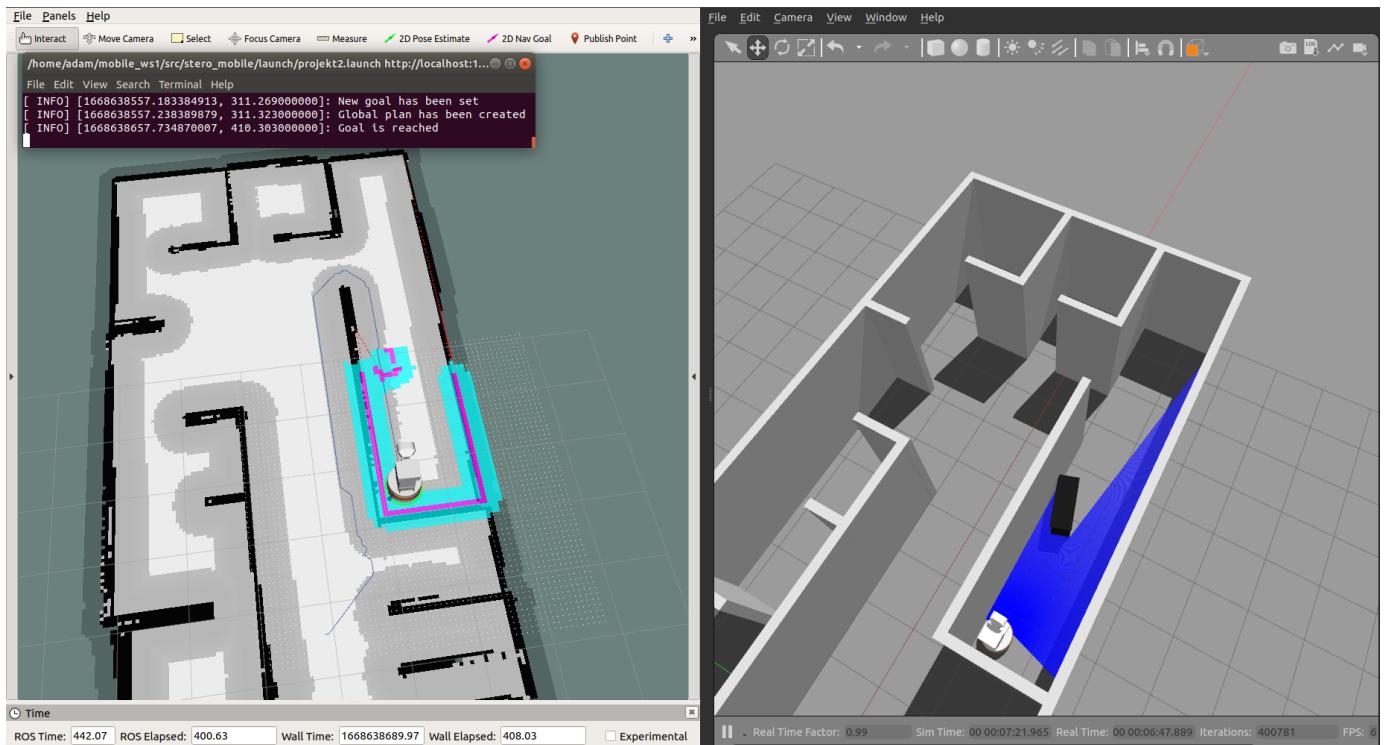
Poniżej przedstawiona jest demonstracja pracy robota w (prawie) ostatecznej wersji skryptu. Prawie, ponieważ po zakończeniu testów doszlifowaliśmy jeszcze kilka parametrów, co miało jednak jedynie pozytywny wpływ na pracę algorytmu.

Rysunek 2.5 przedstawia Tiago po dotarciu do celu. W konsoli udokumentowana jest historia komunikatów opisujących chwilowy stan robota, wraz z ostatnim komunikatem – goal reached.



Rysunek 2.5 Tiago pokonuje prostą trajektorię

Przejazd robota od punktu startowego do celu w statycznym środowisku nie jest zbyt imponujący. Utrudnijmy więc zadanie wstawiając pojedynczą przeszkodę blisko trajektorii robota.

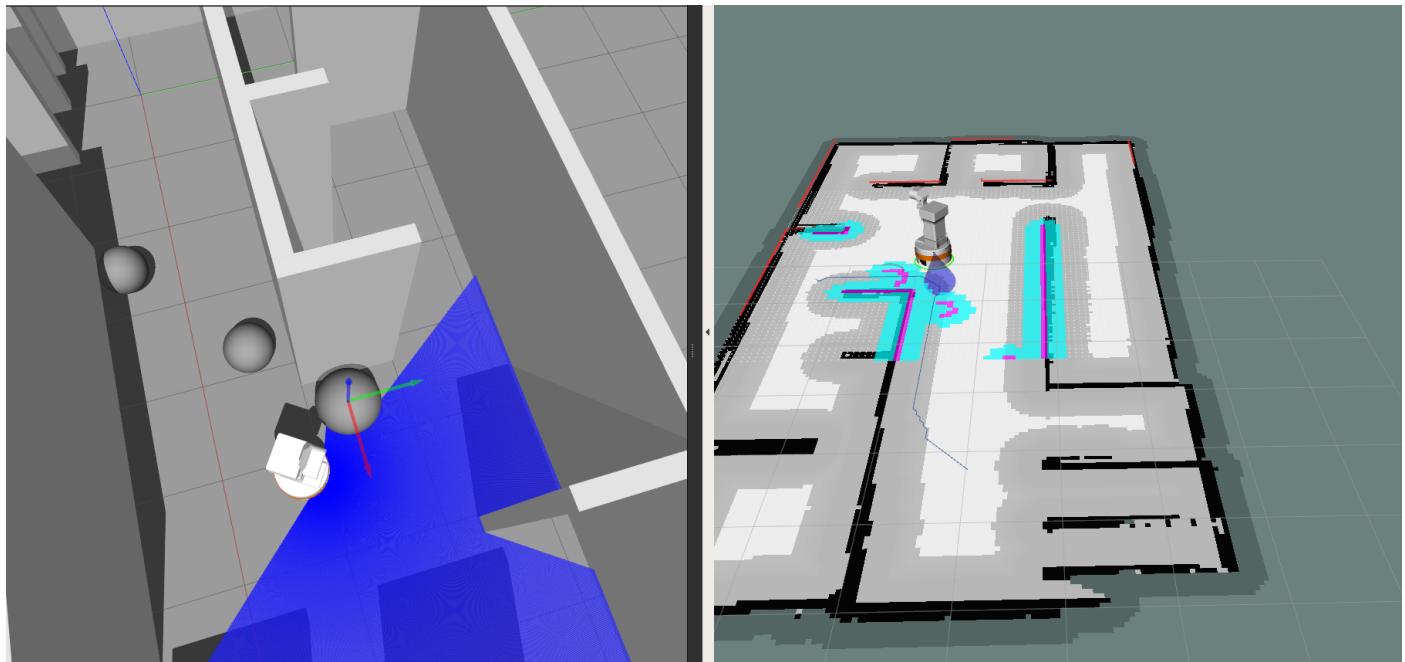


Rysunek 2.6 Robot Tiago po ominięciu szafki stojącej mu na drodze

Rysunek 2.6 przedstawia model Tiago w punkcie końcowym - po dotarciu do celu. Porównując trajektorię z Rviz z modelem świata z symulatora Gazebo widzimy, że robot

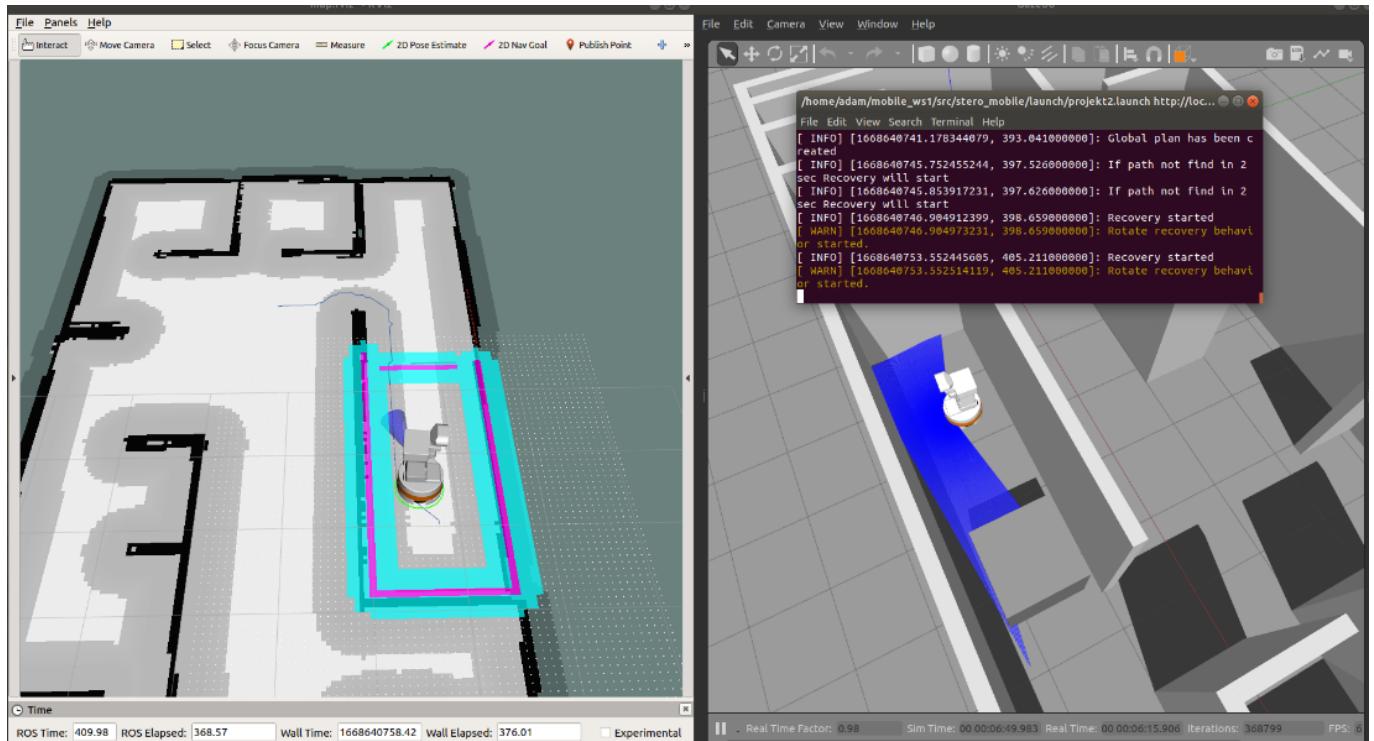
podczas jazdy musiał ominąć szafkę która niespodziewanie pojawiła się w korytarzu. W tym wypadku planer lokalny zadziałał prawidłowo i robot bez większych problemów zboczył z trajektorii i objechał szafkę.

Kolejnym podniesieniem trudności zadania będzie dodanie wielu przeszkód. *Rysunek 2.7* został zrobiony podczas omijania przez Tiago ostatniej z trzech kul stojących mu na drodze do zadanego celu. Tym razem ponownie lokalny planer zadziałał prawidłowo i cel trasy został bezpiecznie osiągnięty.



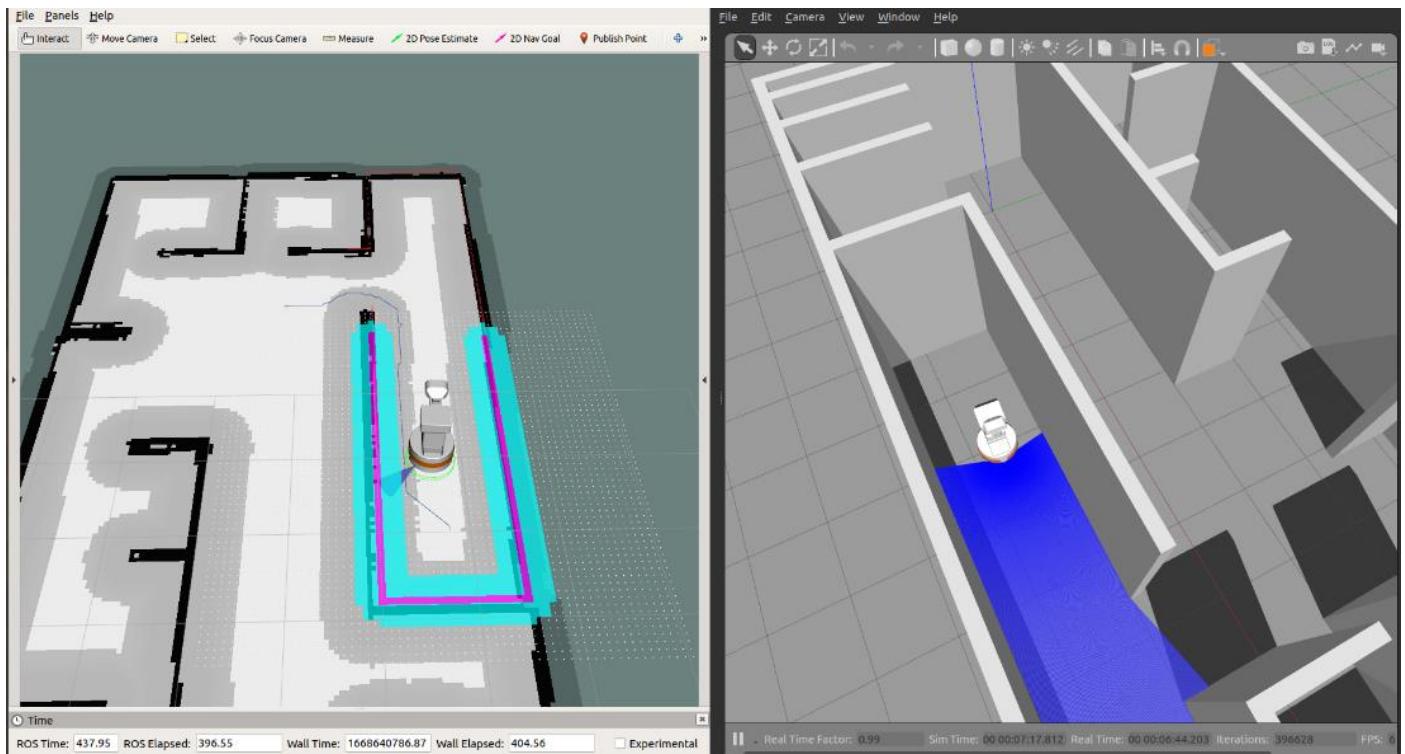
Rysunek 2.7 Robot Tiago manewruje między kulami

Sprawdźmy jak Tiago poradzi sobie w sytuacji zakleszczenia. W tym celu nakazujemy mu przejechanie do korytarza bez wyjścia i blokujemy mu jedyną drogę wyjazdu. Następnie zadajemy mu trajektorię, której wykonanie wymagałoby przejechanie przez przeszkodę blokującą wyjście. Po otrzymaniu takiej instrukcji, robot dojeżdża do przeszkody, obraca się próbując ją obejść i ostatecznie włącza tryb *Recovery*, przeznaczony do takich właśnie sytuacji. Sytuację w jakiej znajduje się robot w tej chwili przedstawia *rysunek 2.8*. Na konsoli dostajemy informację o stanie w jakim znajduje się robot.



Rysunek 2.8 Robot Tiago przechodzi w tryb Recovery

Robot w tym trybie obraca się do momentu do pójki nie wykryje przejścia umożliwiającego mu dojazd do celu. Usuwamy więc przeszkadzający mu sześciian, po czym robot orientuje się, że dojazd do celu jest możliwy. Sytuacja ta przedstawiona jest na Rysunku 2.9. Następnie robot kontynuuje bezpieczną jazdę aż do zadanego celu końcowego.



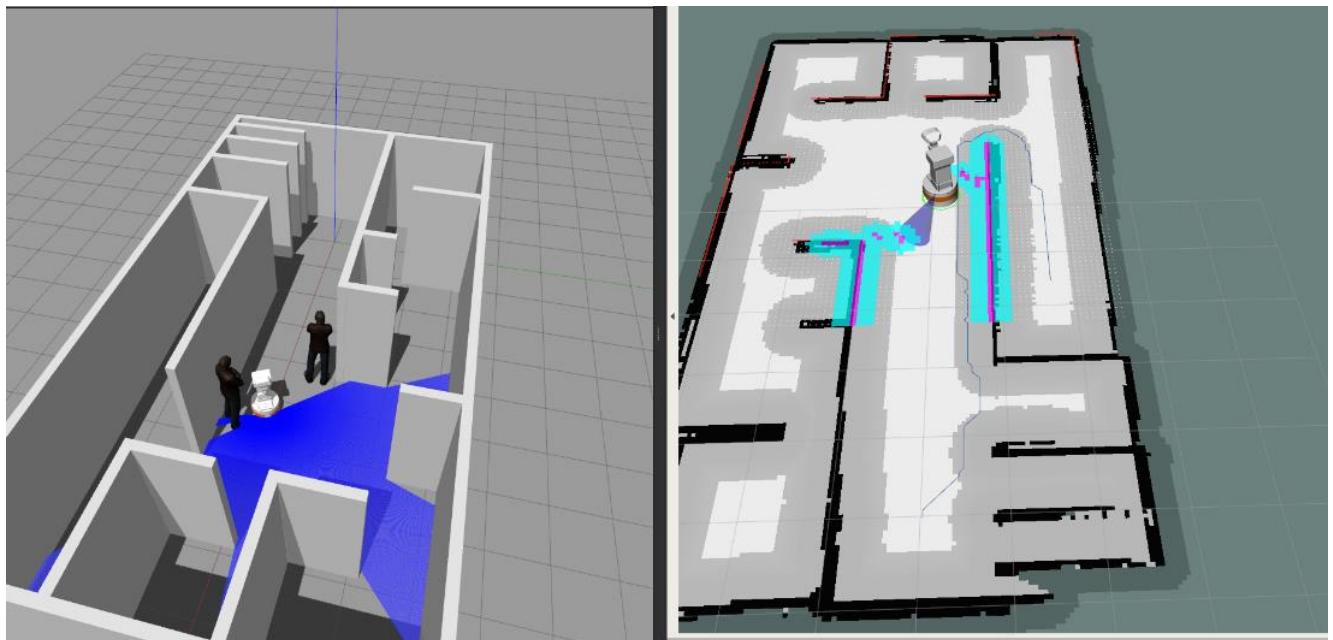
Rysunek 2.9 Robot Tiago wychodzi z trybu Recovery i kontynuuje jazdę

5. Podsumowanie:

Mimo iż robot wydaje się nieźle spełniać postawione mu zadanie, podczas pracy nad projektem napotkaliśmy wiele problemów. Jednym z pierwszych napotkanych problemów była kiepska mapa która została stworzona na podstawie modelu świata – powodowało to niedokładności w sterowaniu, oraz częste kolizje Tiago ze ścianami.

Kolejnym problemem jest czas potrzebny planerowi lokalnemu na wyznaczenie trajektorii. Zdarzają się sytuacje w których Tiago zatrzymuje się i bardzo długo „myśli” zanim kontynuuje ruch. Niestety mimo wielu prób z różnymi parametrami nie udało nam się całkowicie wyeliminować tego zjawiska.

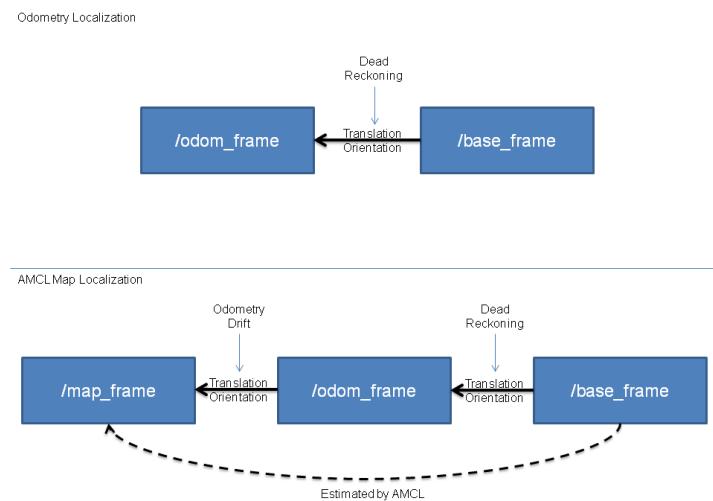
Na koniec przedstawiamy jeszcze Tiago jeżdżącego po dosyć skomplikowanej trajektorii zawierającej „prawdziwych” ludzi. Na szczęście dobrze dostrojony robot poradził sobie nawet z tak poważnym zadaniem i nie zranił (a nawet nie dotknął) żadnego z nich.



Rysunek 2.10 Tiago pokonuje skomplikowaną trajektorię

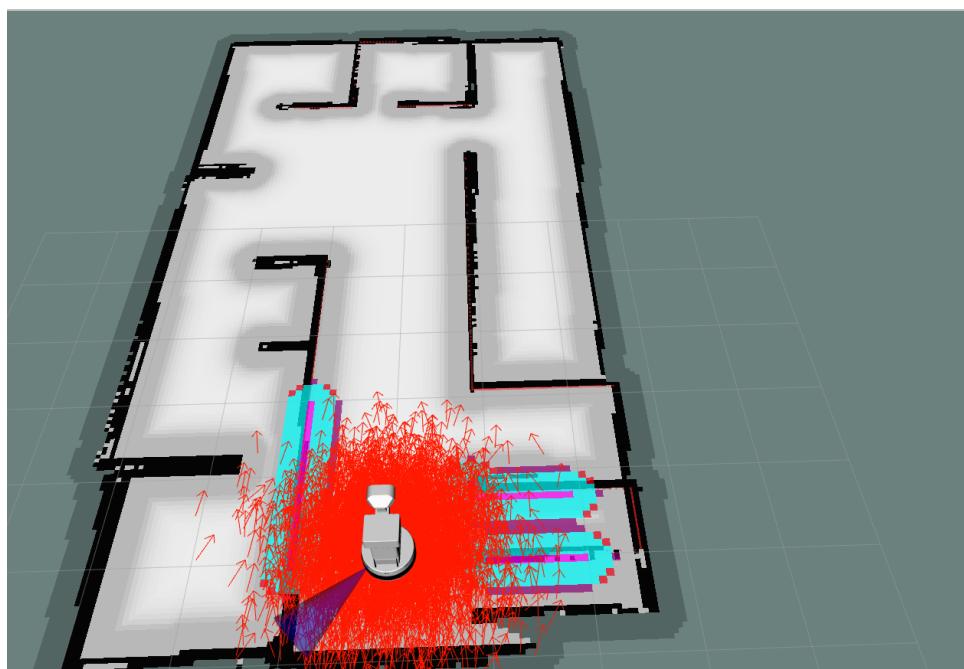
LABORATORIUM 3

Laboratorium 3 polegało na użyciu algorytmu AMCL (Adaptive Monte Carlo Localization) do lokalizacji globalnej Tiago, oraz zbadaniu wpływu parametrów na jego działanie. Sposób działania algorytmu *amcl* przedstawiony jest na poniższym diagramie (źródło-dokumentacja ROS amcl).

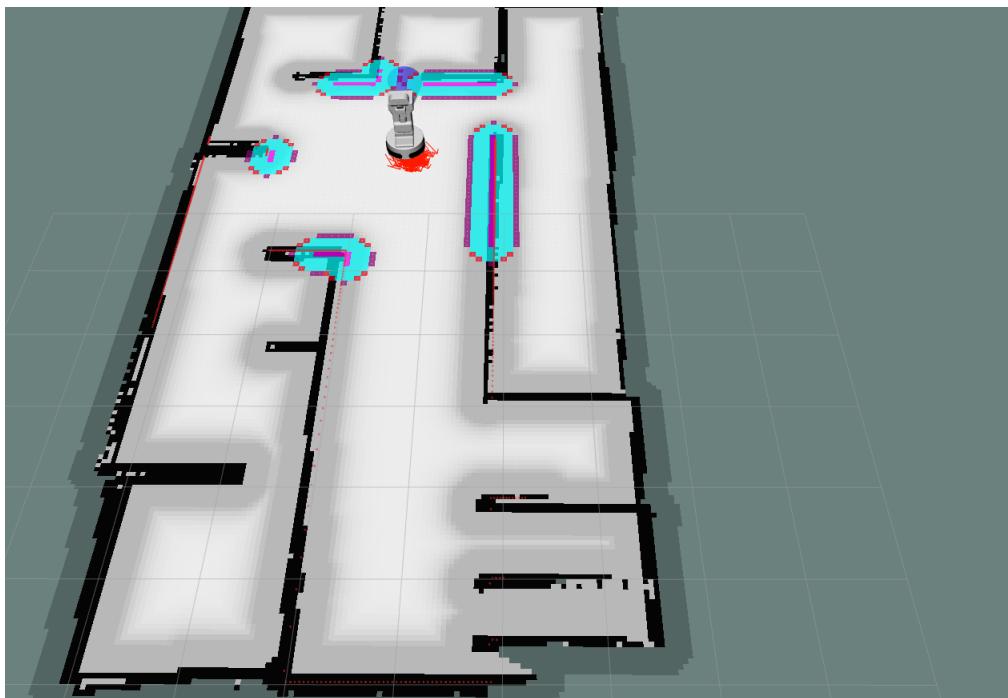


Rysunek 3.1 Schemat działania lokalizacji w oparciu o AMCL

W trakcie laboratorium, zmienialiśmy kolejne parametry *amcl* wykorzystując *rqt_reconfigure* i obserwowaliśmy ich wpływ na działanie algorytmu w obydwu światach: labiryntie i korytarzu. Rozpoczęliśmy od sprawdzenie działania algorytmu przy parametrach domyślnych. Zauważaliśmy że robot porusza się o wiele płynniej, co może świadczyć o większej dokładności lokalizacji globalnej z wykorzystaniem amlc niż odometrii.



Rysunek 3.2 Rozkład możliwych położen algorytmu AMCL przy starcie

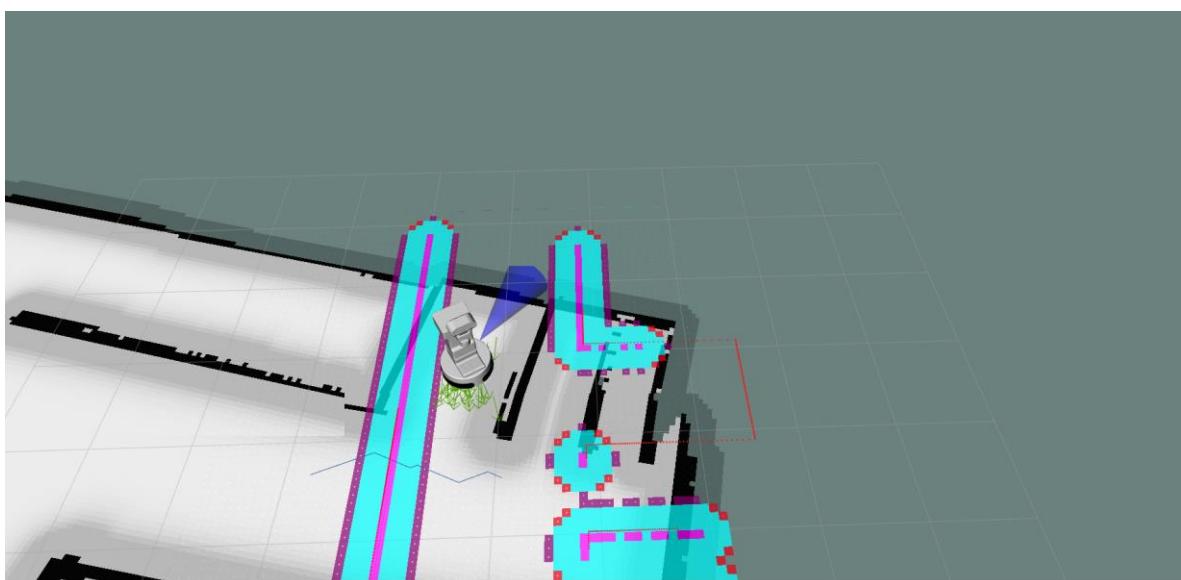


Rysunek 3.3 Rozkład możliwych położen algorytmu AMCL po przejechaniu pewnego dystansu

Jak widzimy na powyższych dwóch obrazkach, na początku chmura potencjalnych punktów jest spora, jednak wraz z przemieszczaniem się robota i wykrywaniem punktów charakterystycznych na mapie dokładność lokalizacji wzrasta.

Wpływ liczby cząsteczek:

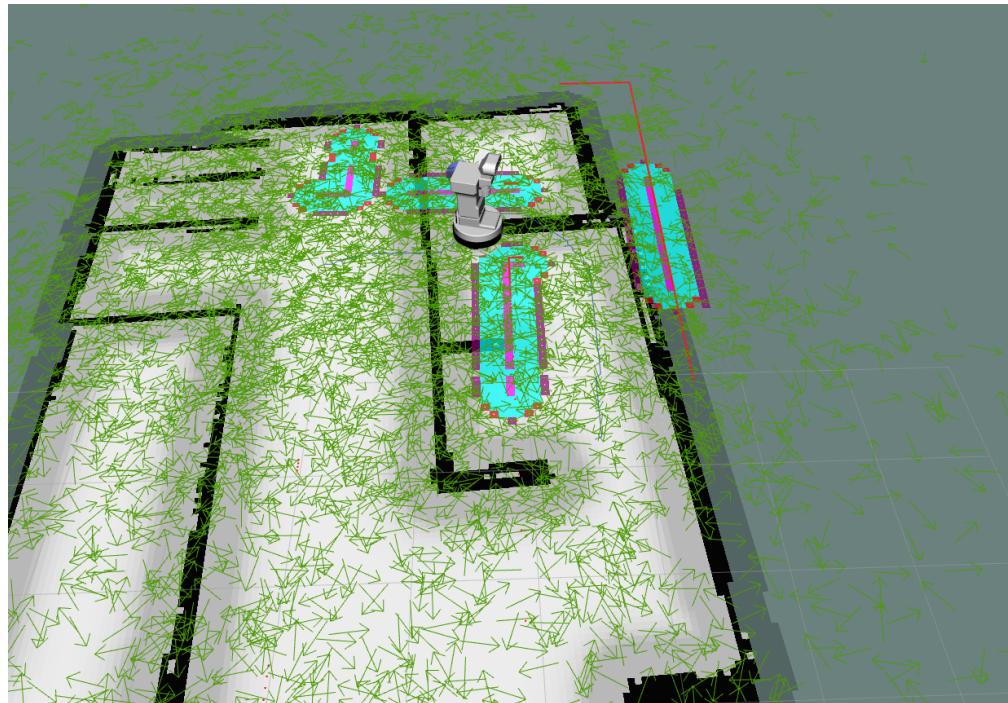
Zmniejszając max liczbę cząsteczek dostępnych w algorytmie zmniejsza się dokładność lokalizacji a tym samym wydłuża się czas po którym algorytm dokładnie „znajdzie” Tiago. Ponieważ algorytm opiera się o rozkłady losowe, przy małej liczbie strzałek dokładna samolokalizacja może nie być w ogóle możliwa.



Rysunek 3.4 Algorytm amcl przy małej liczbie prawdopodobnych położień

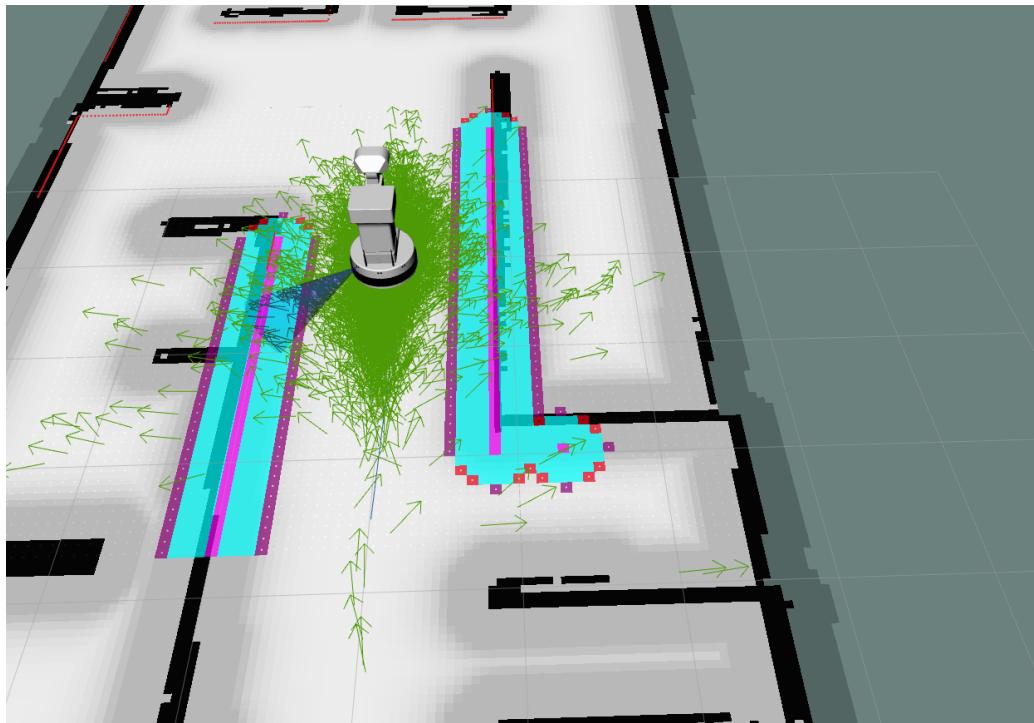
Zwiększenie spodziewanych błędów w estymacji odometrii:

Ustawiliśmy parametry *odom_alpha1-4* na 0.4. co spowodowało rozstrzał estymowanych pozycji robota na całą mapę – tak jak na Rysunku 3.5



Rysunek 3.5 Algorytm amcl przy zbyt dużej wartości parametru alpha - start

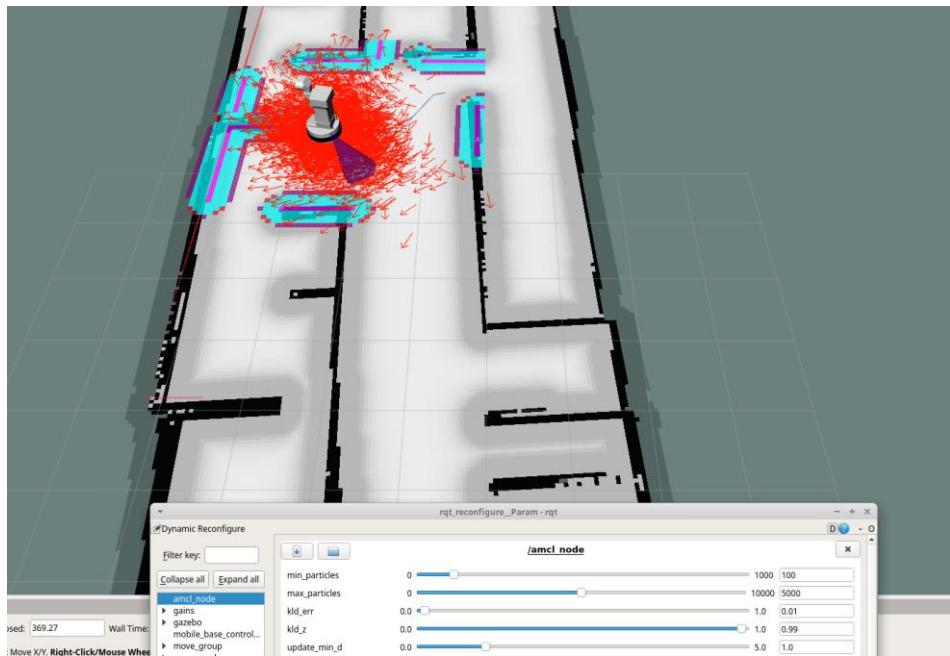
Na szczęście po przejechaniu pewnego dystansu wynik lokalizacji poprawia się (Rysunek 3.6), wciąż jest jednak dużo gorszy niż był przy wartościach domyślnych.



Rysunek 3.6 Algorytm amcl przy zbyt dużej wartości parametru alpha – po chwili jazdy

Zwiększenie parametru update_min_d:

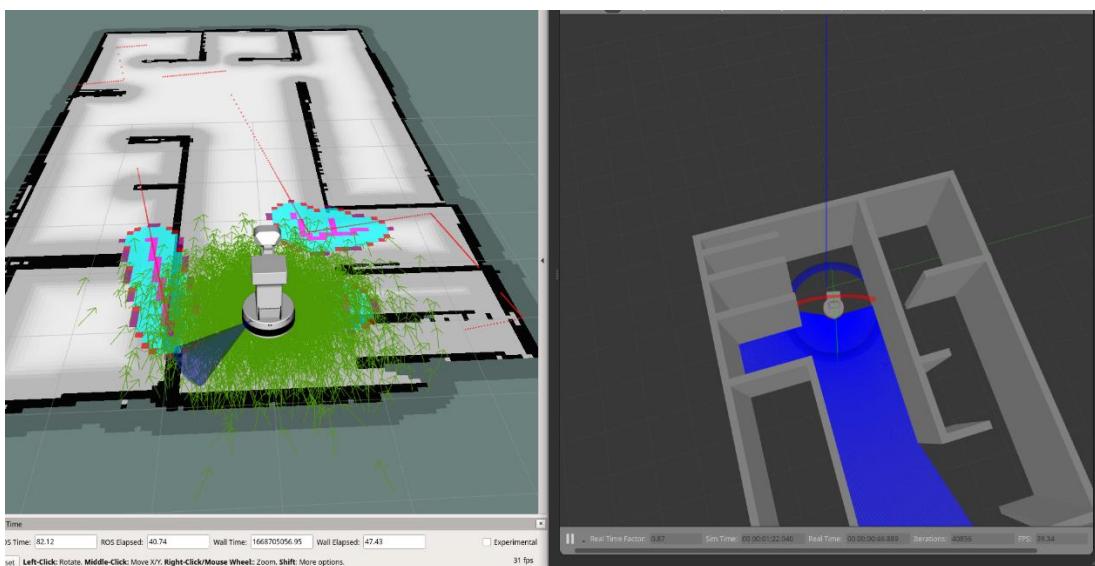
Parametr ten określa minimalny dystans po jakim algorytm generuje nową chmurę punktów. Zwiększyliśmy go z domyślnej wartości 0,2 metra do 1 metra. Wyniki eksperymentu przedstawione są na Rysunku



Rysunek 3.7 Algorytm amcl ze zwiększonym parametrem update_min_d

Zauważamy znaczące pogorszenie jakości estymacji pozycji robota.

Kolejnym eksperymentem jaki postanowiliśmy przeprowadzić była symulacja wystąpienia dużego błędu niesystematycznego – w postaci wrednego studenta który przestawił i obrócił Tiago wimulatorze świata Gazebo. Obraz świata z punktu widzenia robota (a więc ten w symulatorze Rviz) przedstawiony jest na Rysunku 3.8.

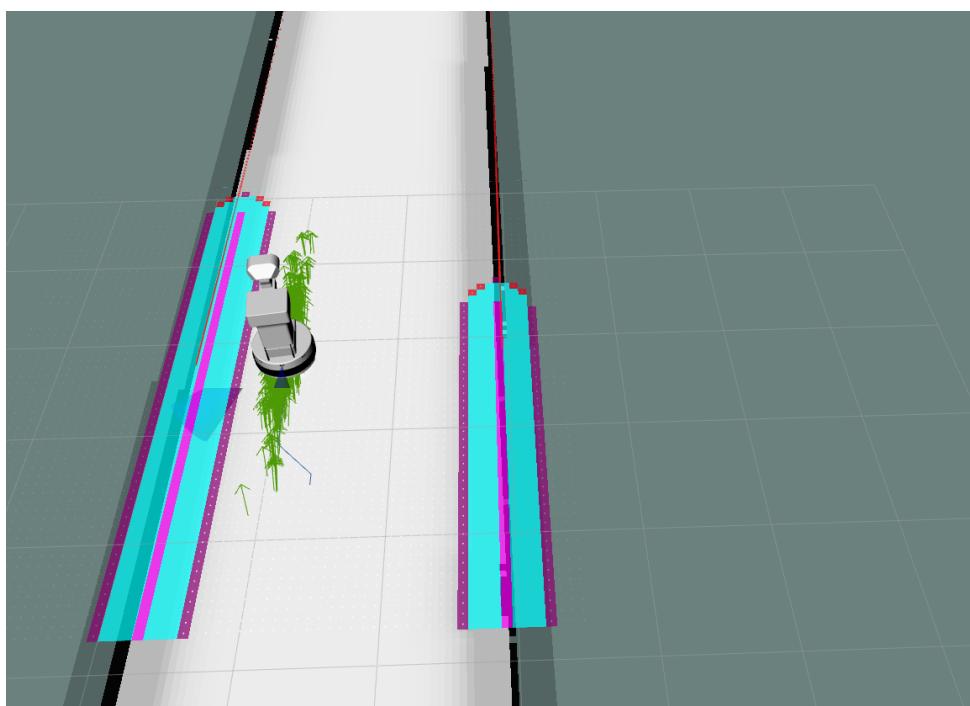


Rysunek 3.8 Widok z Rviz'a i Gazebo po przestawieniu robota

Na szczęście po przejechaniu niewielkiego dystansu algorytm *amcl* umożliwił robotowi Tiago wyznaczenie swojej rzeczywistej pozycji.

Zmiana mapy na korytarz:

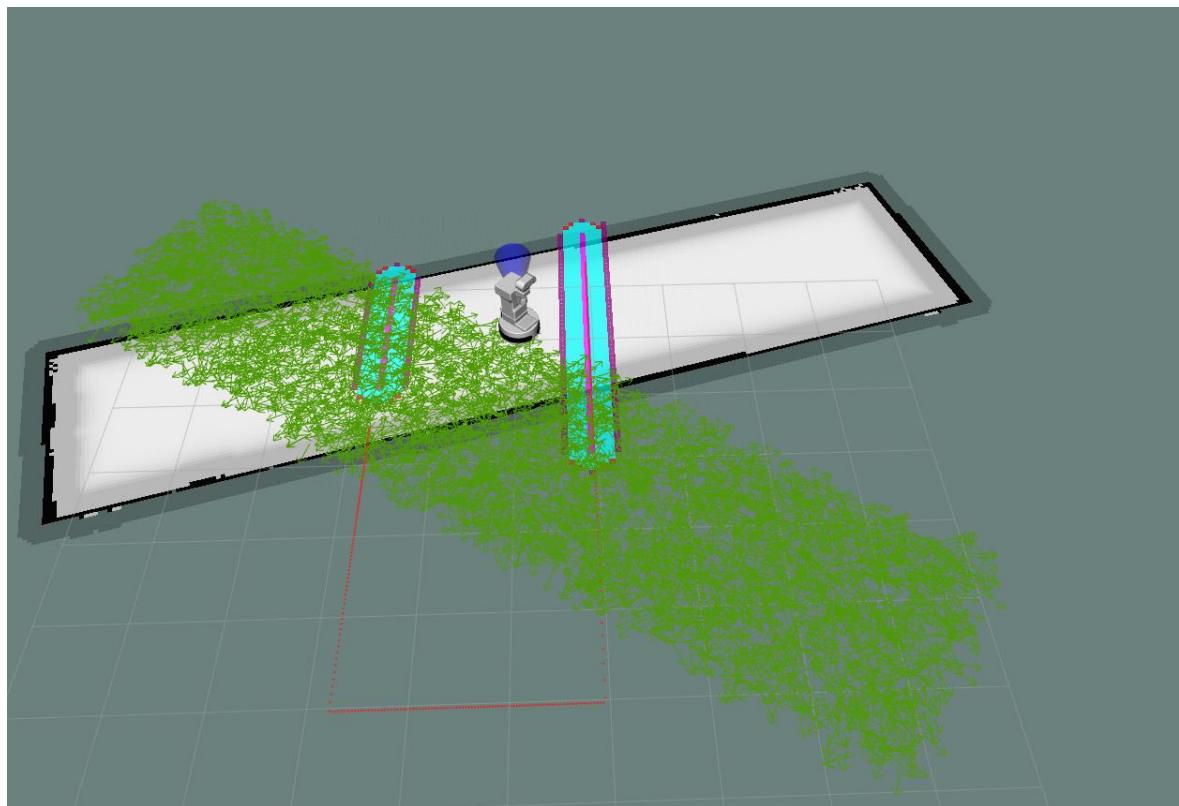
Teraz zajmiemy się analizą wpływu parametrów algorytmu *amcl* gdy Tiago znajduje się na mapie *korytarz*. Pierwszą rzeczą jaką zauważymy jest to że w korytarzu czas potrzebny na wyznaczenie lokalizacji jest znacznie dłuższy – wynika to z faktu że na tej mapie jest o wiele mniej punktów charakterystycznych w porównaniu do *labiryntu*. Mimo to po kilku przejazdach „w tą i z powrotem” jakość lokalizacji jest satysfakcjonująca.



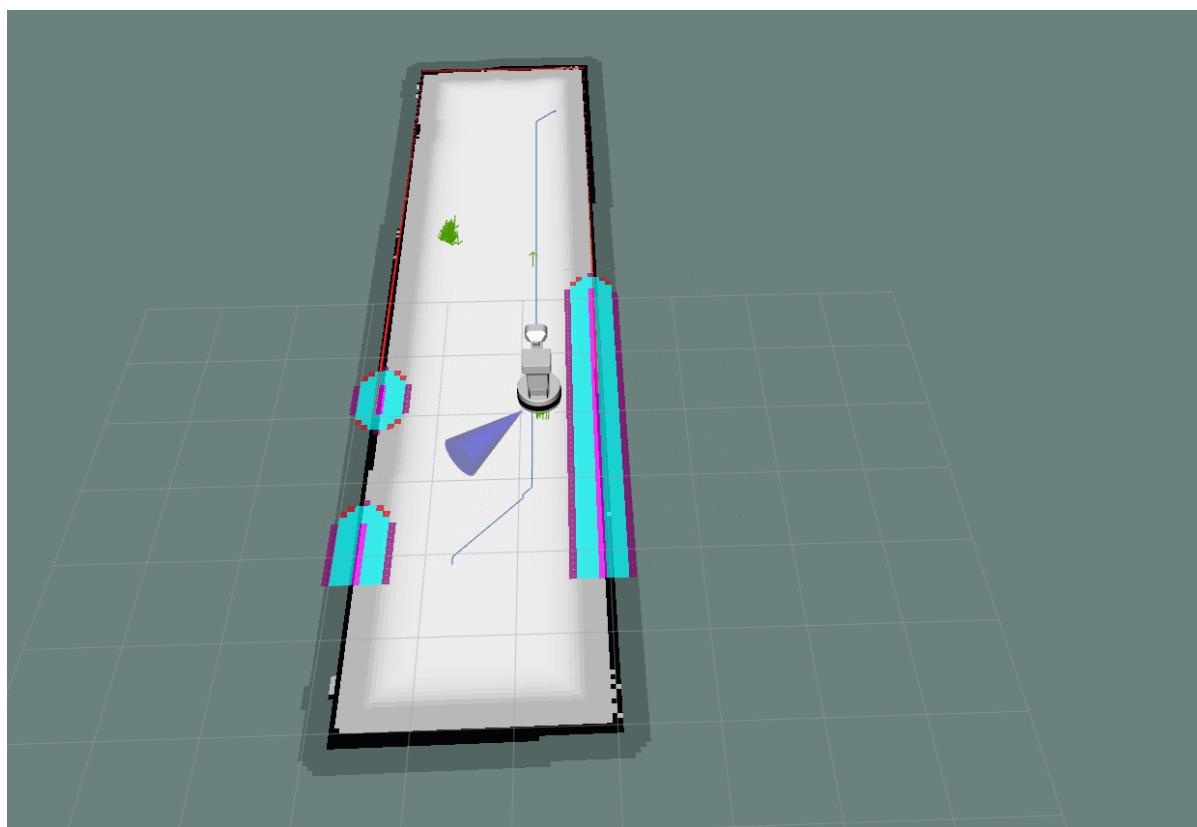
Rysunek 3.9 Algorytm *amcl* przy jeździe po korytarzu

Losowy rozkład losowej chmury punków:

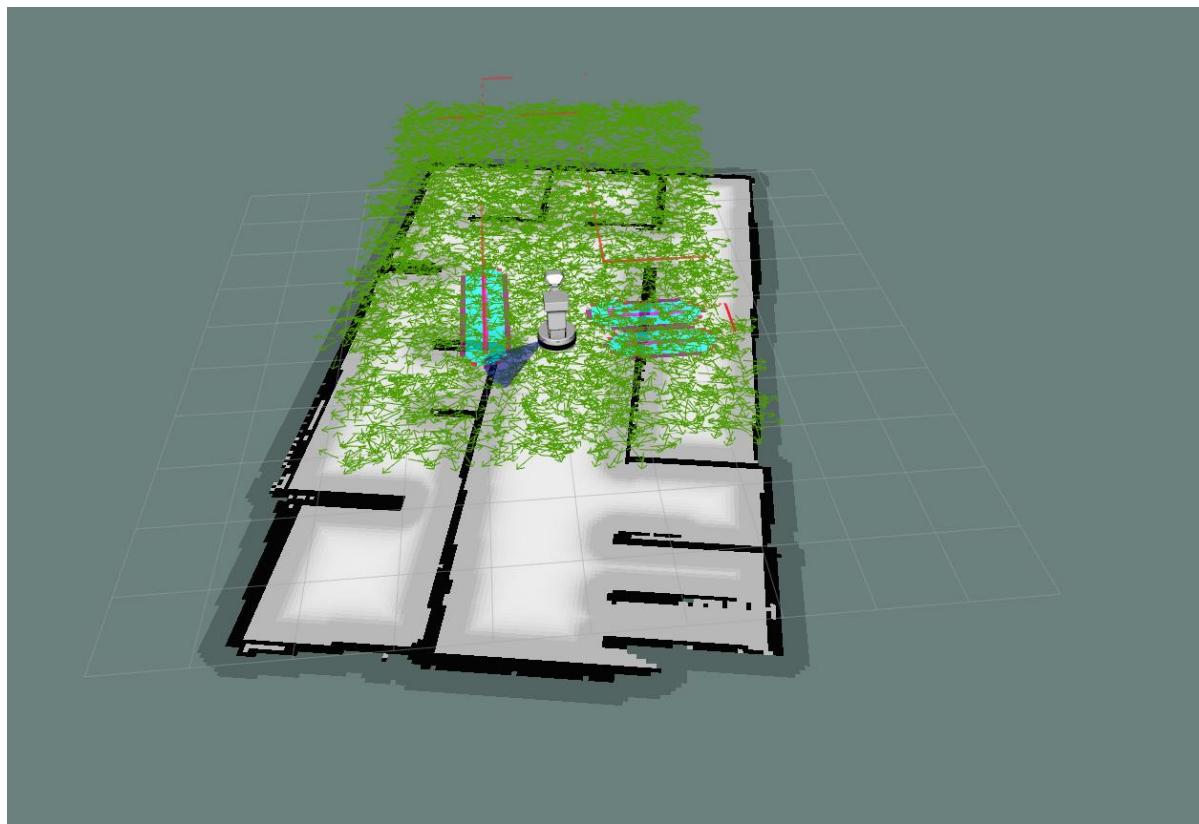
W pewnej chwili przy użyciu odpowiedniego rozservice'a resetujemy chmurę punków algorytmu *amcl*. Po resetie prawdopodobne punkty rozkładają się losowo (czyli mniej więcej równomiernie) po całej mapie. Po kilku obrotach wynikających z przejścia w tryb Recovery obserwujemy skupianie się strzałek wyznaczających prawdopodobną pozycję Tiago w dwóch punktach. Po chwili zastanowienia dochodzimy do wniosku, że jest to prawidłowe zachowanie się algorytmu wynikające z jego charakterystyki – obydwa te punkty są prawdopodobnymi miejscami znajdowania się robota.



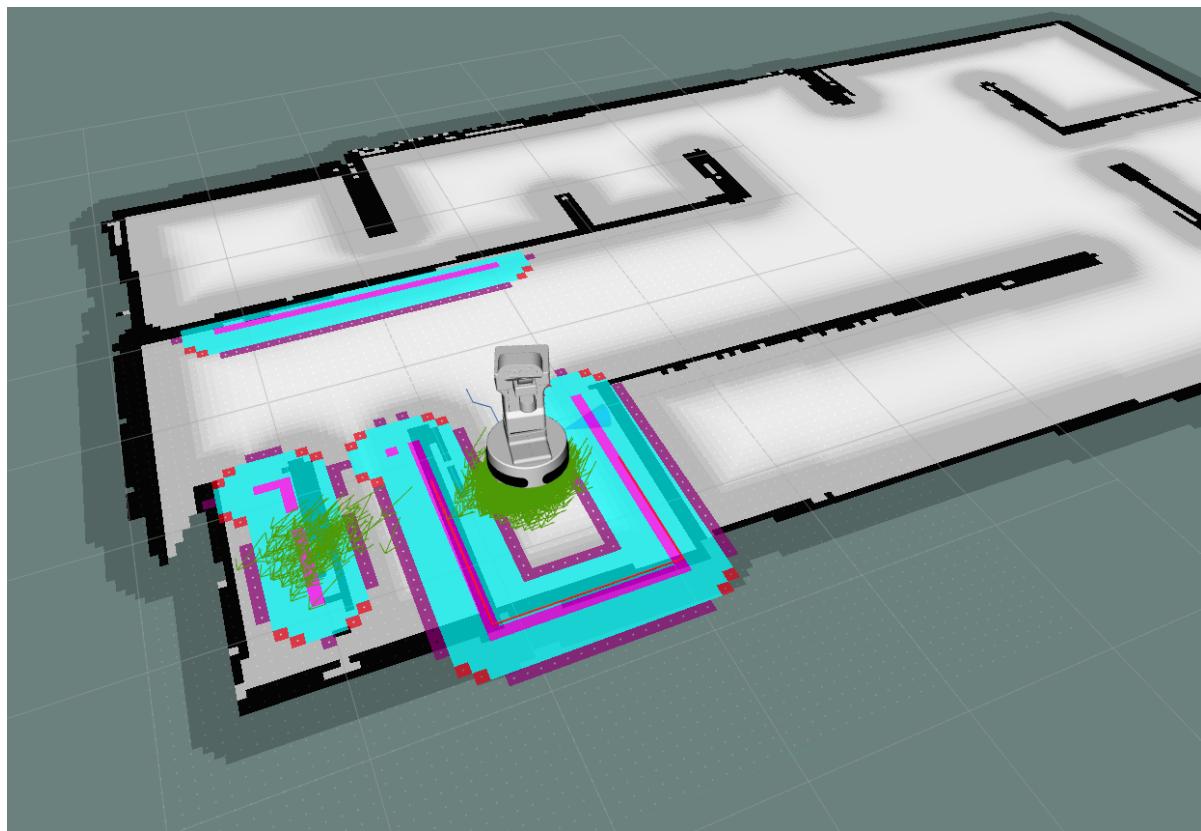
Rysunek 3.10 Rest chmury punktów algorytmu amcl - korytarz



Rysunek 3.11 Rest chmury punktów algorytmu amcl – dwie możliwe pozycje w korytarzu



Rysunek 3.12 Rest chmury punktów algorytmu amcl - labirynt



Rysunek 17 Rest chmury punktów algorytmu amcl – dwie możliwe pozycje w labiryncie