Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Sterowanie i symulacja robotów

Sprawozdanie końcowe z projektu i laboratorium
Blok mobilny

Kaniuka Jan, Krasnodębski Przemysław

Warszawa, 2021

Spis treści

1. Wstęp

Sprawozdanie zostało przygotowane w ramach zajęć z przedmiotu *Sterowanie i symulacja robotów* w semestrze zimowym roku 2021. Część laboratoryjna została zrealizowana podczas zajęć w dniu 19.10.2021. Główny przedmiot rozważań podczas projektu oraz laboratorium stanowiła analiza dokładności przemieszczania się robota mobilnego podczas względnego sterowania pozycyjnego. W pierwszym przypadku robot poruszał się na podstawie wyłącznie zadawanych prędkości, a w drugim wariancie wykorzystywał dane z odometrii. Programy zaimplementowane podczas prac nad projektem zostały również uruchomione na rzeczywistym robocie *Tiago* firmy PAL w Laboratorium Robotyki na Wydziale EiTI.

Rys. 1.1: Robot *Tiago* (*Rico*)

2. Część laboratoryjna

2.0.1. Zapoznanie ze środowiskiem

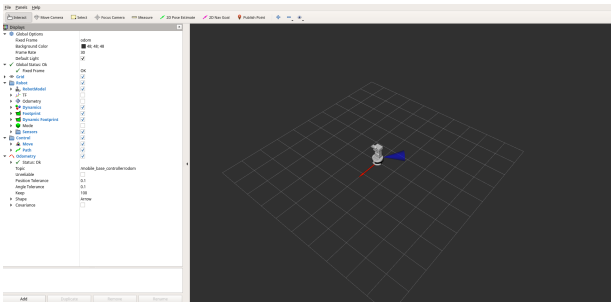
Pracę w laboratorium rozpoczęliśmy od odpowiedniego skonfigurowania stanowiska do pracy. Stworzyliśmy przestrzeń roboczą i podłączyliśmy się do pakietów robota *Tiago*. Następnie przeszliśmy do folderu źródeł i pobraliśmy stosowne repozytorium. W pierwszej kolejności uruchomiliśmy programy Gazebo oraz RViz w celu wizualizacji robota w pustym świecie (symulacji).

2.0.2. Sterowanie prędkościowe

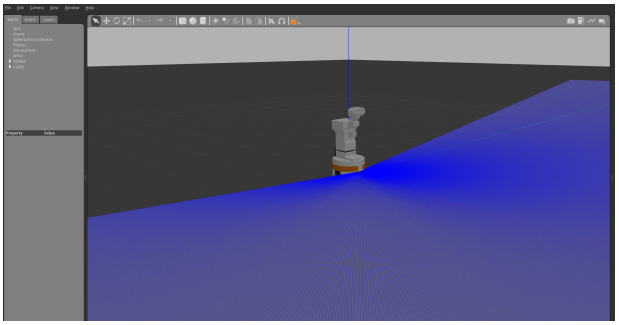
W naszym przypadku sterowanie to polega na inicjalizacji ruchu w punkcie (0,0,0) [x,y,theta]. Kolejne sterowania wyznaczane są na podstawie zrealizowanej (idealnie) poprzedniej akcji ruchu. Zakładamy więc, że znając prędkość i zadając konkretny czas ruchu, robot dojedzie do pożądanej pozycji. Testy mieliśmy wykonać dla trzech, różnych prędkości o następujących wartościach podanych przez Prowadzącego:

	linear	angular
low	0.04 m/s	0.11 rad/s
medium	0.096 m/s	0.264 rad/s
high	0.23 m/s	0.6336 rad/s

(testy sprawdzające poprawność działania programu wykonywaliśmy względem danych z tematu `/gazebo/model_states/pose[1]/position`)



Rys. 2.1: Widok z wizualizatora *RViz*



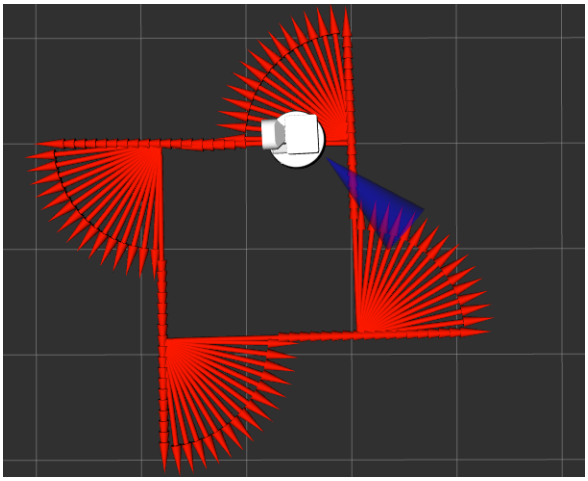
Rys. 2.2: Widok z symulatora *Gazebo*

2.0.3. Omówienie implementacji

Parametrami w naszym programie są: długość boku kwadratu oraz prędkość liniowa i obrotowa. Prędkość zadana jest stała, więc przekształcając równanie dla ruchu jednostajnego możemy wyliczyć czas. Zadajemy ruch liniowy przez wyznaczony okres czasu, a następnie ruch obrotowy o kąt 90 stopni. Zakładamy (możliwe, że naiwnie), że robot wykonał powierzone mu zadanie tzn. przebył dystans w odpowiednim czasie.

2.0.4. Test kwadratu z wizualizacją w Rviz

Podczas zajęć udało nam się sprawdzić poprawność działania zaimplementowanej struktury sterowania. Poniżej załączamy obraz ze ścieżką wykonaną przez robota podczas sterowania prędkościowego przy małej prędkości.

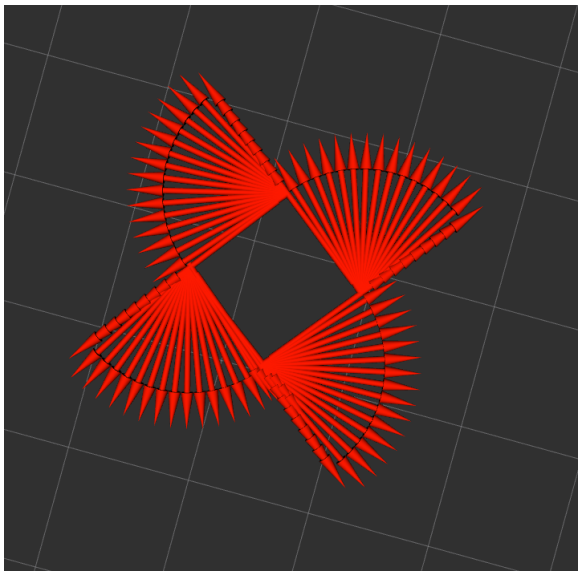


Rys. 2.3: Sterowanie prędkościowe (prędkość *mała*)

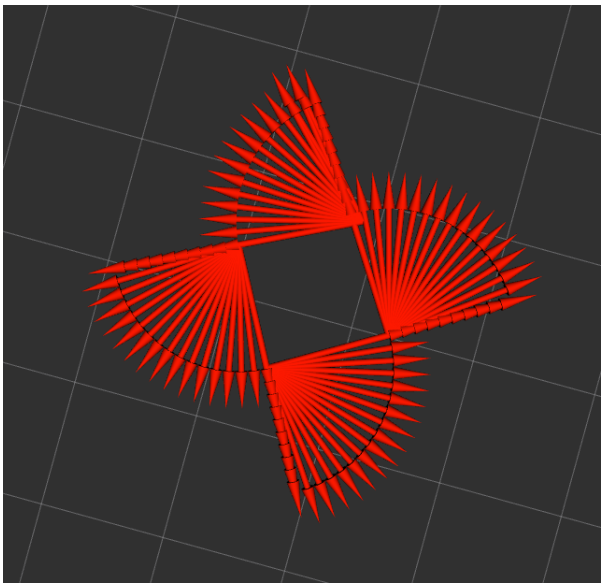
3. Część projektowa

3.0.1. Implementacja względnego sterowania pozycyjnego

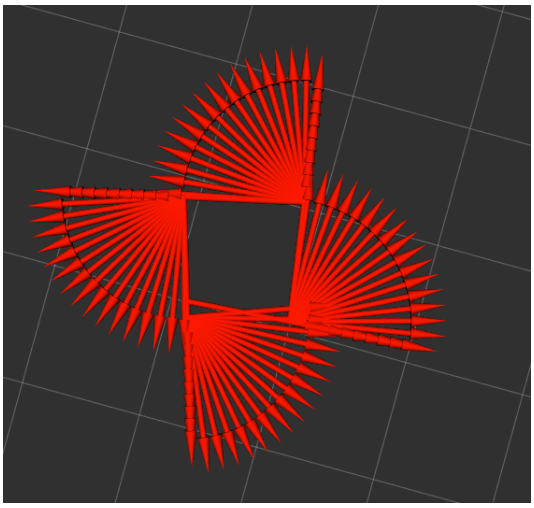
Poniżej zamieszczamy testy kwadratu dla trzech, różnych wartości prędkości: liniowej i kątowej:



Rys. 3.1: Prędkość *mała*



Rys. 3.2: Prędkość *średnia*

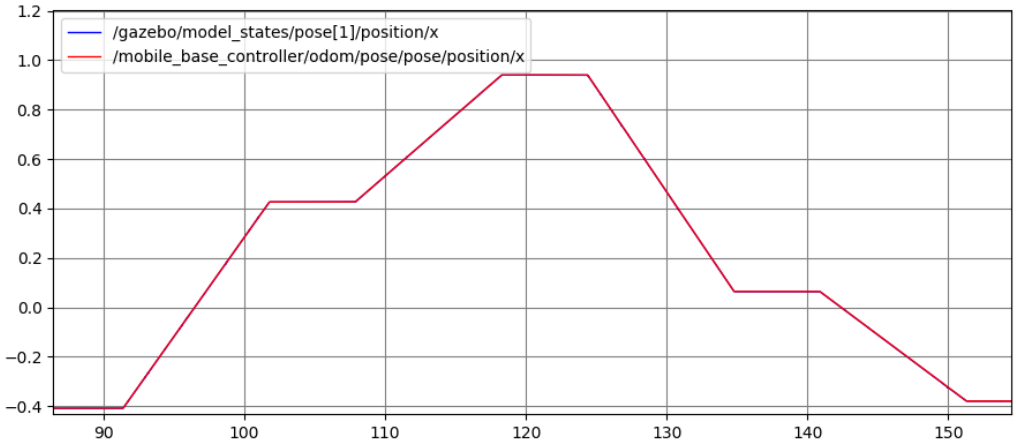


Rys. 3.3: Prędkość *duża*

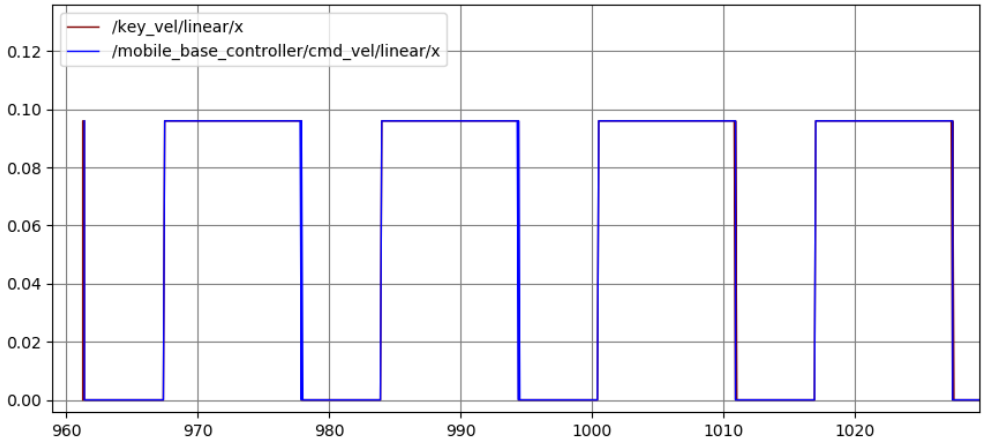
3.0.2. Porównanie dokładności osiągania pozycji z wykorzystaniem pozycji referencyjnej

Dokonaliśmy analizy działania wewnętrznych regulatorów robota podczas symulacji w reakcji na zadawane komendy ruchu. Przebiegi wartości zadanej i osiąganych parametrów ruchu przedstawiono na rysunkach poniżej. Z powodu bardzo podobnych wyników dla różnych prędkości i implementacji przedstawiono tylko niektóre wykresy.

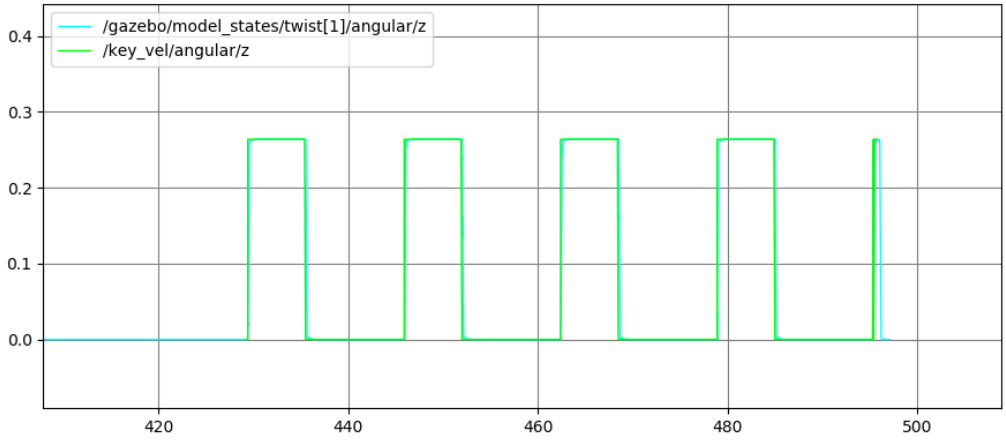
Sterowanie prędkościowe



Rys. 3.4: Pozycje liniowe x zadane i wykonane

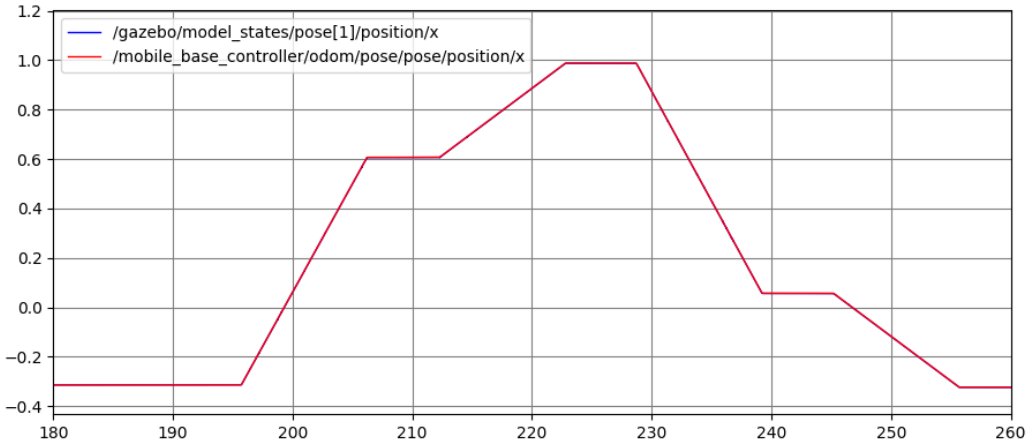


Rys. 3.5: Prędkości liniowe x zadane i wykonane

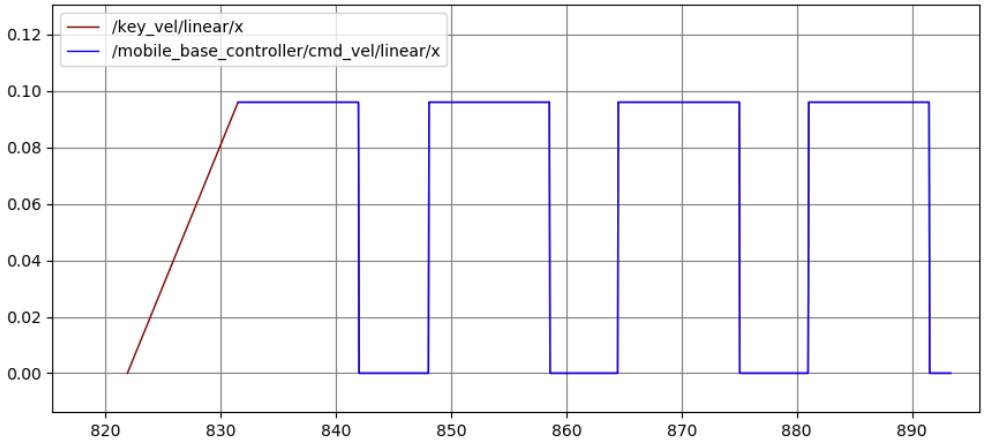


Rys. 3.6: Prędkości kątowe zadane i wykonane

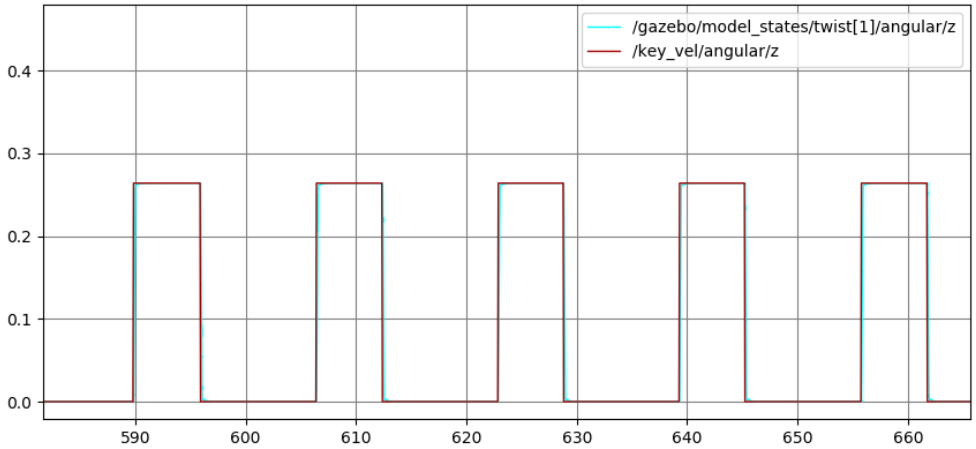
Sterowanie z odometrią



Rys. 3.7: Pozycje liniowe x zadane i wykonane



Rys. 3.8: Prędkości liniowe x zadane i wykonane



Rys. 3.9: Prędkości kątowe zadane i wykonane

Powyższe dane zebrano podczas ruchu robota ze średnią prędkością. Wyraźnie widać, że prędkości oraz pozycje zadane są równe referencyjnym. Małe rozbieżności można zauważyć podczas dużych zmian wartości prędkości. Wynikają one ze skończonej wartości przyśpieszenia robota.

3.0.3. Praca z rzeczywistym robotem

Mieliśmy możliwość uruchomienia naszych programów na rzeczywistym robocie *Tiago*. Celem zadania było zebranie danych do analizy porównawczej pomiędzy lokalizacją globalną, a lokalizacją wynikającą z odometrii. Uruchomiliśmy program zadający ruch po kwadracie z wykorzystaniem odometrii dla trzech wartości prędkości.

3.0.4. rosbag

Podczas pracy z robotem zebraliśmy dane, które zostały poddane późniejszej analizie. Przy użyciu narzędzia *rosbag* mogliśmy "nagrać" wiadomości nadawane na wybranych tematach, aby móc później np. zwizualizować dane z pozycji na wykresie.
Zebrane dane pozyskaliśmy z tematów:

— `/tf`
— `/mobile_base_controller/odom`
— `/robot_pose`

Poniżej załączyliśmy dane obrazujące przekształcenia pomiędzy układami `map` i `odom` w momencie startowym (na początku zbierania danych). Różnica w położeniu tych układów współrzędnych zostanie uwzględniona podczas analizy danych opisanej w dalszej części sprawozdania.

```
At time 1634830380.909
- Translation: [-0.434, 0.499, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.895, 0.446]
            in RPY (radian) [0.000, -0.000, 2.218]
            in RPY (degree) [0.000, -0.000, 127.074]
```

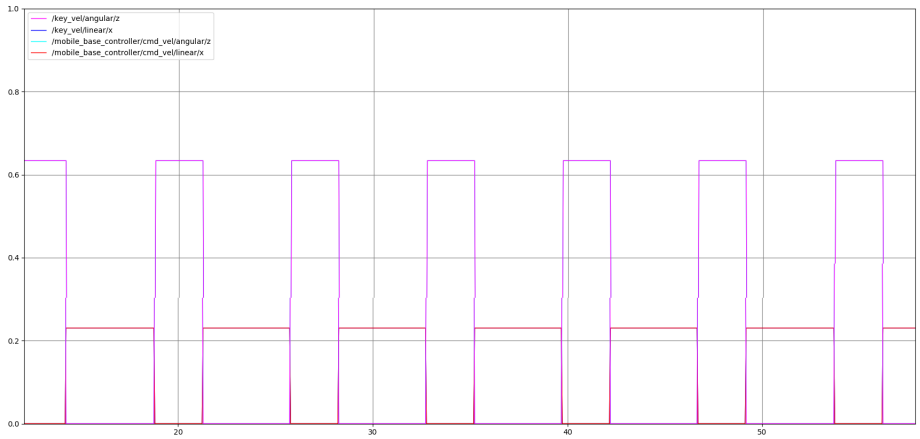
```
At time 1634830659.457
- Translation: [-0.431, 0.601, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.876, 0.482]
            in RPY (radian) [0.000, -0.000, 2.137]
            in RPY (degree) [0.000, -0.000, 122.432]
```

```
At time 1634830792.527
- Translation: [-0.425, 0.700, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.858, 0.514]
            in RPY (radian) [0.000, -0.000, 2.061]
            in RPY (degree) [0.000, -0.000, 118.104]
```

3.0.5. rqtplot

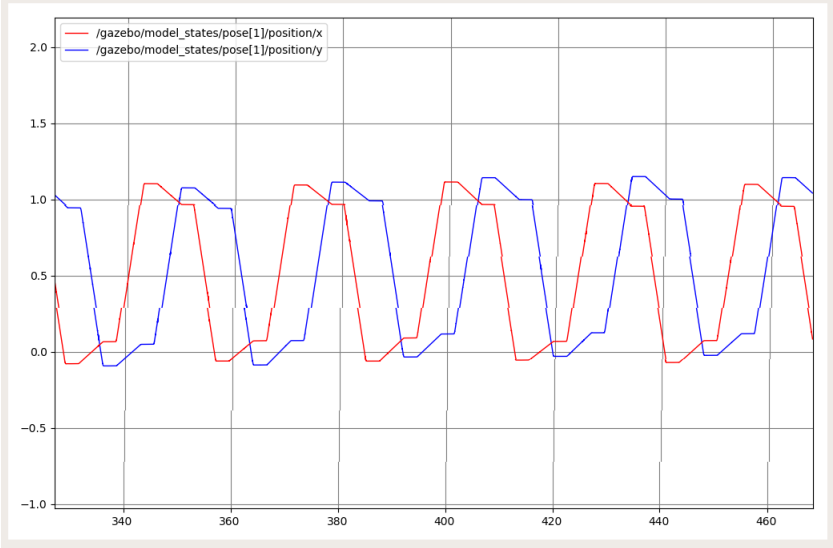
Narzędzie `rqt_plot` pozwoliło nam na wizualizację oraz porównywanie prędkości i/lub pozycji podczas przeprowadzania symulacji. Dzięki temu narzędziu mogliśmy zweryfikować, czy prędkość zadana jest zgodna z prędkością rzeczywistą.

Poniższy wykres pokazuje, że prędkości zadane oraz wykonane są zgodne.



Rys. 3.10: Prędkości zadane i wykonane

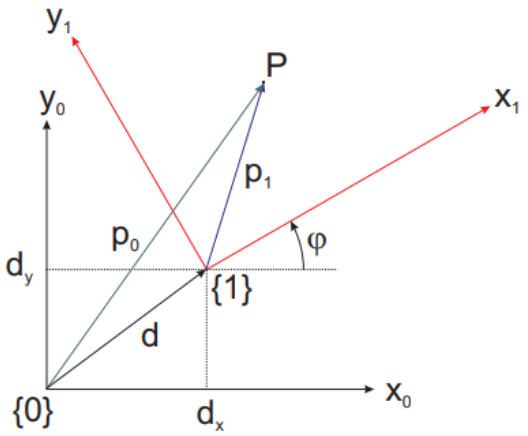
Narzędzie `rqt_plot` pozwoliło nam również na wizualizację danych obrazujacych pozycję referencyjną robota.



Rys. 3.11: Pozycja robota z *Gazebo*

3.0.6. Analiza danych z pliku *.bag*

Dane odometryczne w temacie `/mobile_base_controller/odom` są wyrażone w układzie `odom`, a dane z lokalizacji globalnej w układzie `map`. Transformacja między tymi układami jest zmienna w trakcie pracy robota i wynika z działania algorytmu globalnej lokalizacji. Zatem, aby porównać lokalizację robota na podstawie odometrii oraz lokalizację z fuzją innych danych sensorycznych należy porównywać dane z obu wskazanych tematów względem wspólnego układu współrzędnych. Za ten układ należy przyjąć układ `map`.
Przekształcenie wykonaliśmy zgodnie z poniższą grafiką (slajd z wykładu *Wstęp do robotyki* autorstwa dr hab. inż. Wojciecha Szynkiewicza, EiTI PW)

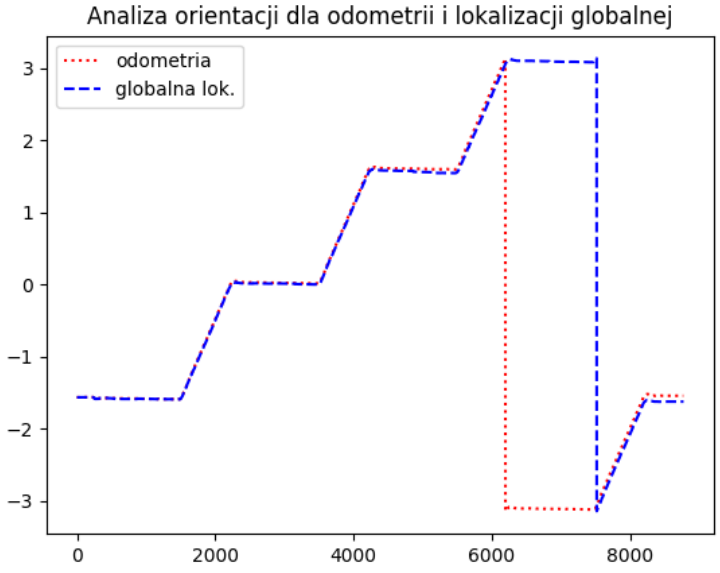


Rys. 2: Obrót i przesunięcie w $\mathbb{R}^2$

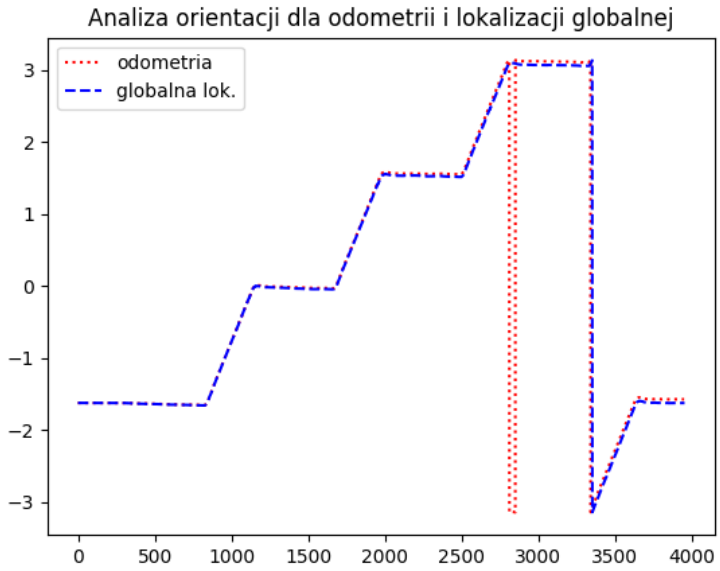
$$\begin{bmatrix} p_{0x} \\ p_{0y} \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} p_{1x} \\ p_{1y} \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

Rys. 3.12: Transformacja między układami współrzędnych na płaszczyźnie

Ruch rzeczywistego robota



Rys. 3.13: Test kwadratu z odometrią, prędkość niska



Rys. 3.14: Test kwadratu z odometrią, prędkość średnia

Rys. 3.15: Test kwadratu z odometrią, prędkość wysoka

Przedstawione przebiegi orientacji dla odometrii i lokalizacji globalnej są ze sobą zgodne. Widoczne rozbieżności wynikają z niewielkich błędów/szumów pomiarowych i faktu zawierania się miary kątowej w zakresie od -π do π.

Wyniki pomiarów

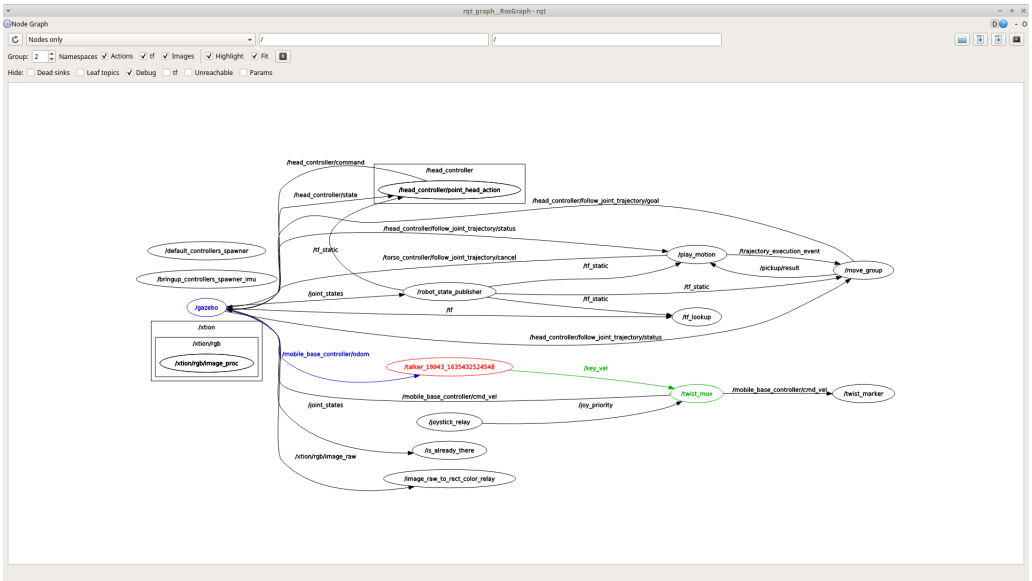
Tab. 3.1: Porównanie błędów pozycji dla różnych prędkości

Prędkość	Błąd x		Błąd y		Błąd theta	
	Skumulowany	Średni	Skumulowany	Średni	Skumulowany	Średni
low	61.17	0.0294	92.95	0.0447	521	0.2506
medium	48.06	0.0122	99.35	0.0252	6327	1.6018
high	276.52	0.0315	308.98	0.0352	19730	2.2498

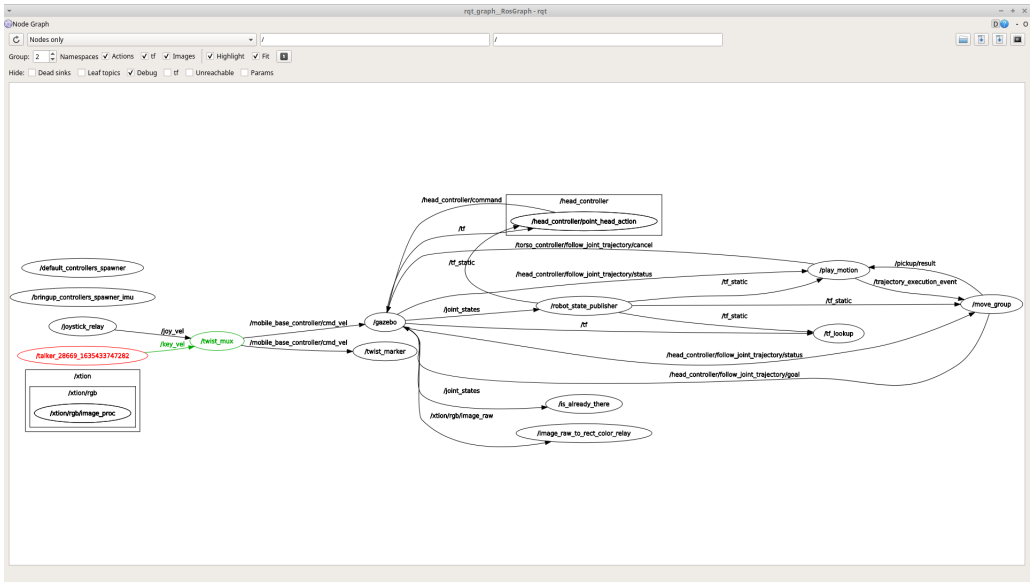
W przypadku zbyt szybkich ruchów robota tracimy dokładnośc pozycjonowania go w przestrzeni. Ograniczone przyśpieszenie robota wpływa szczególnie na niedokładnośc ruchu obrotowego, przez co dokładne wykonanie wyznaczonej trajektori może być awykonalne.

3.0.7. Struktura komunikacji

W celu zobrazowania struktury sterującej robotem *Tiago* w ramach ROS (*Robot Operating System*) załączmy poniżej grafy struktur komunikacji tzn. zależności pomiędzy poszczególnymi węzłami i tematami.



Rys. 3.16: Sterowanie z odometrią - struktura komunikacji



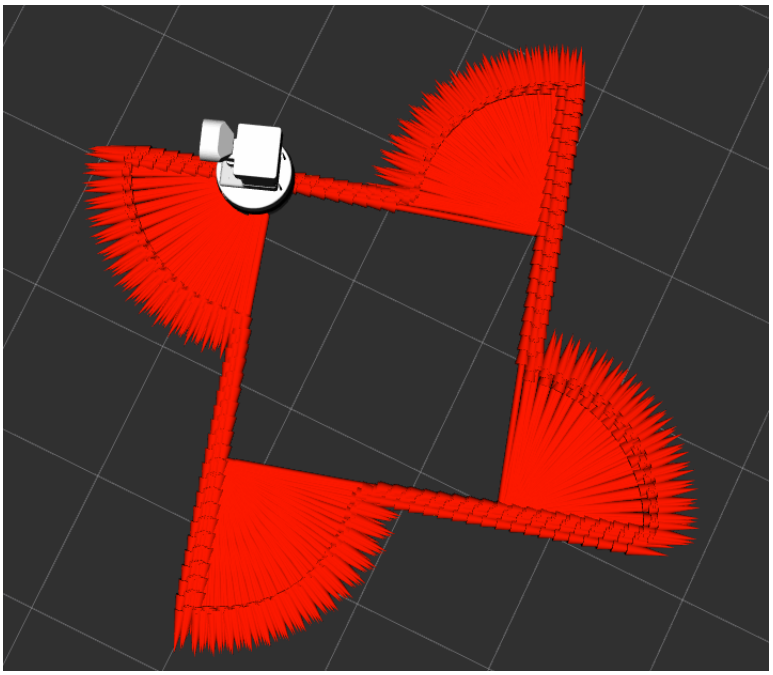
Rys. 3.17: Sterowanie wyłącznie prędkościowe - struktura komunikacji

4. Podsumowanie

Testy przeprowadzone dla sterowania prędkościowego wykazały zgodność prędkości zadanej z prędkością wykonaną. Mimo to ten rodzaj sterowania musi zakładać całkowity brak oddziaływań zewnętrznych na robota. Najmniejszy poślizg, czy też nierówna powierzchnia, mogą spowodować, że robot będzie "przekonany", że dojechał do celu - w rzeczywistości może być daleko od niego.

Odometria jest bardziej skomplikowana od samego sterowania prędkościowego. Korzystamy w niej z danych z enkoderów zamontowanych na wałach silników, które informują nas o dystansie przebytym przez robota oraz o kącie, o jaki wykonał obrót. Nierówności terenu, ściany, poślizgi, czy też manualne podniesienie i postawienie robota w innym miejscu powodują zagubienie informacji o lokalizacji. Odometria robota bazuje na fizycznych wielkościach takich jak rozstaw i średnica kół, czy też rozdzielczość enkoderów. Drobne niedokładności w wartościach tych parametrów powodują akumulację błędów. Zaobserwowaliśmy to podczas testów z prawdziwym robotem.

Na poniższej grafice widać ścieżkę wykreśloną przez robota. Po wykonaniu kilku, symulowanych przejazdów w ramach testu kwadratu widać już wyraźnie, że kolejne ścieżki nie pokrywają się idealnie.



Rys. 4.1: Akumulacja błędów odometrii

5. Wstęp

Sprawozdanie zostało przygotowane w ramach zajęć z przedmiotu *Sterowanie i symulacja robotów* w semestrze zimowym roku 2021. Część laboratoryjna została zrealizowana podczas zajęć w dniu 4.11.2021. Na laboratorium skupiliśmy się na budowie świata i jego mapy. W części projektowej mieliśmy za zadanie zaimplementować moduł planowania lokalnego.

Rys. 5.1: Robot *Tiago* (*Rico*)

6. Część laboratoryjna

6.0.1. Budowa parterowego świata

Pracę na laboratorium zaczęliśmy od budowy parterowego świata w symulatorze `Gazebo`. Zbudowaliśmy i zapisaliśmy dwa światy: parterowy budynek oraz długi korytarz bez drzwi. Model budynku będzie naszym środowiskiem testowym dla modułu planowania implementowanego w czasie projektu. Drugi świat zostanie wykorzystany przy pracy z systemem lokalizacji `amcl` podczas laboratorium nr 3.

6.0.2. Parterowy budynek z przejściami o różnej szerokości

Zbudowaliśmy budynek o prostym układzie pomieszczeń. Umieściliśmy w nim przejścia o różnej szerokości - odpowiednio 100%, 150%, 200% i 300% szerokości robota. W części projektowej będziemy sprawdzać, jak wartości parametrów map kosztów oraz planerów wpływają na zdolność robota do przejechania przed otwór danej szerokości.

6.0.3. Długi korytarz bez drzwi

Utworzyliśmy korytarz o jednolitych ścianach, bez drzwi oraz okien o długości 20 metrów.

6.0.4. Konfiguracja modułu SLAM

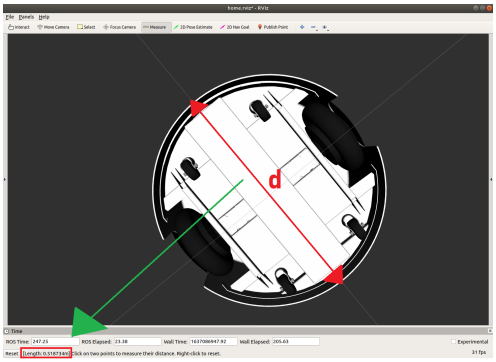
Korzystaliśmy z modułu `SLAM` (*Simultaneous Localization and Mapping*) wykorzystującego informacje z tematu */scan_raw*. Pozwoliło to na stworzenie dwuwymiarowej siatki zajętości.

6.0.5. Budowa mapy

Zbudowanie mapy sprowadzało się do uruchomienia węzła pozwalającego na sterowanie robotem za pomocą klawiatury i na przejechaniu przez robota całej powierzchni parterowego budynku.

6.0.6. Pomiar średnicy robota

Korzystający z narzędzia *Measure* w `RViZ` zmierzyliśmy średnicę bazy jezdnej robota, aby móc ją później uwzględnić przy konfiguracji map kosztów.



Rys. 6.1: Pomiar średnicy bazy jezndej robota TiaGo.

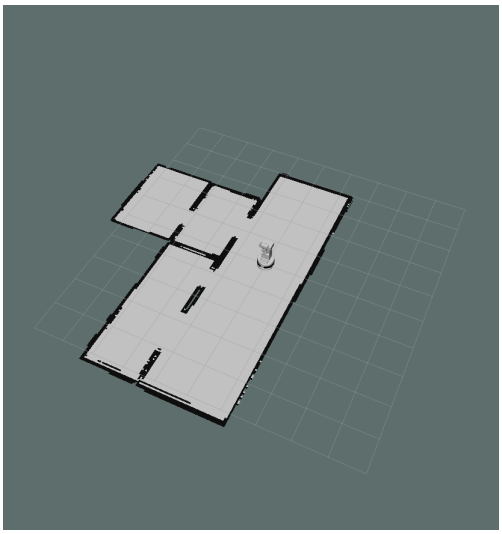
7. Część projektowa

7.0.1. Wstępna konfiguracja głównego pliku projektu

W pliku `listener.cpp` subskrybujemy topic */move_base_simple/goal*, aby mieć bieżącą informację o lokalizacji punktu docelowego. Istotne jest także stworzenie obiektu `tf2_ros::Buffer`, który wykorzystuje lokalna i globalna mapa kosztów. Podczas tworzenia takiej mapy kluczowe jest to, aby wzajemne transformacje pomiędzy układem globalnym, układem robota i układem związanym z położeniem sensorów były aktualne.

7.0.2. *global_costmap* z warstwą *static*

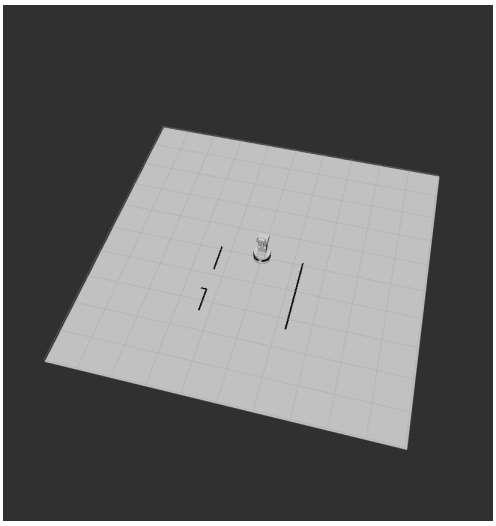
Następnie utworzyliśmy obiekt klasy `costmap_2d::Costmap2DROS`, w tym celu niezbędne było także zdefiniowanie parametrów mapy z warstwą static w pliku `global_costmap_params.yaml` oraz odpowiednia konfiguracja wspomnianego pliku w pliku startowym `tiago_home.launch`. Dokładne znaczenie wykorzytsanych parametrów zostało wyjaśnione w pliku konfiguracyjnym *.yaml*. Po uruchomieniu węzła została wygenerowana mapa kosztów.



Rys. 7.1: Globalna mapa kosztów z warstwą static

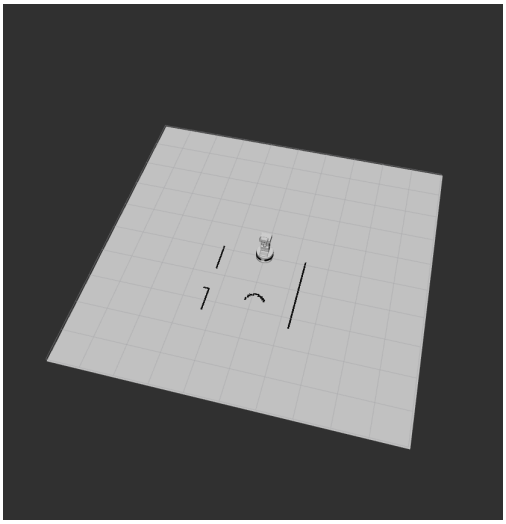
7.0.3. *local_costmap* z warstwami *static+obstacle*

Stworzenie lokalnej mapy kosztów wymagało wykonania podobnych czynności jak dla globalnej mapy kosztów. Dodatkowo w pliku konfiguracyjnym *.yaml* zdefinowano warstwę obstacle, która służy do wykrywania lokalnych przeszkód. Po uruchomieniu węzła powstała lokalna mapa kosztów.



Rys. 7.2: Lokalna mapa kosztów z warstwą static

W trakcie testów umieścilismy przed robotem przeszkodę, lokalna mapa kosztów została zaktualizowana.



Rys. 7.3: Lokalna mapa kosztów z warstwą static oraz obstacle

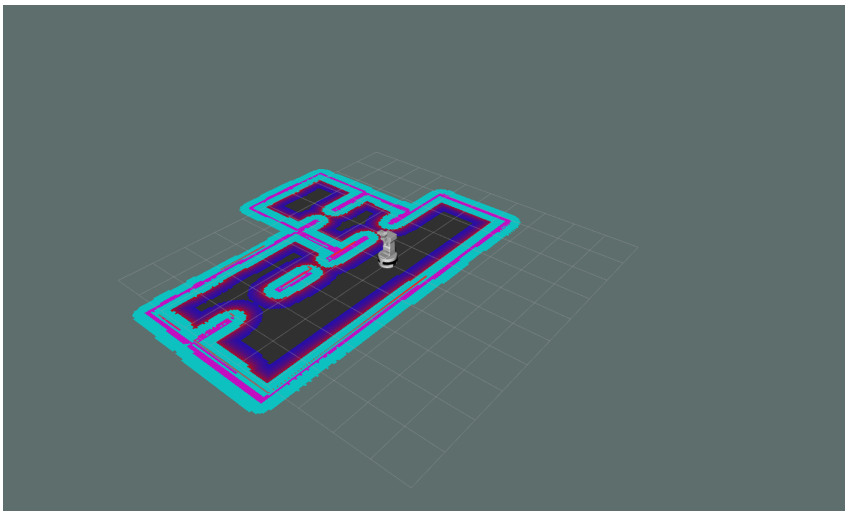
Dla porównania dokonano też testów z przeszkodami dla globalnej mapy kosztów, tak jak oczekiwaliśmy, globalna mapa nie uwzględnia lokalnych przeszkód.

obstacle.pdf obstacle.png obstacle.jpg obstacle.mps obstacle.jpeg obstacle.jbig2 obstacle.jb2 obstacle.PDF obstacle.PNG

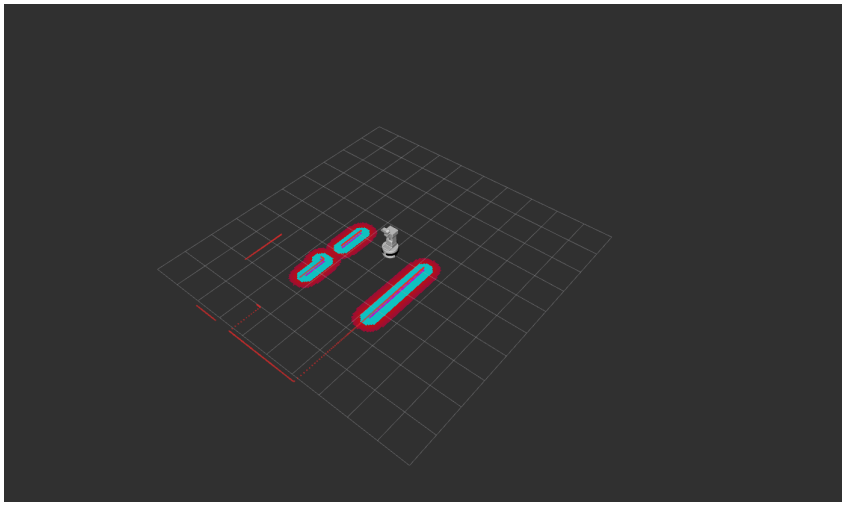
Rys. 7.4: Lokalna mapa kosztów z warstwą static

7.0.4. Dodanie warstwy *infalation* do obydwu map kosztów

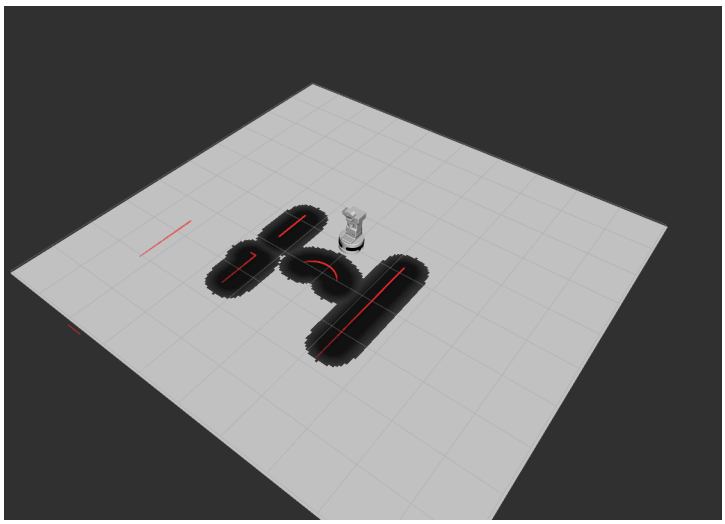
Po pomyślnych testach dla statycznych map kosztów przeszliśmy do konfiguracji warstw inflation. Polegało to głównie na dodaniu. w pliku konfiguracyjnym, parametrów: promienia inflacji oraz funkcji kosztu w zależności od odległości. Wygląd warstwy inflacji przedstawiono na rysunkach poniżej.



Rys. 7.5: Globalna mapa kosztów z warstwą static oraz inflation



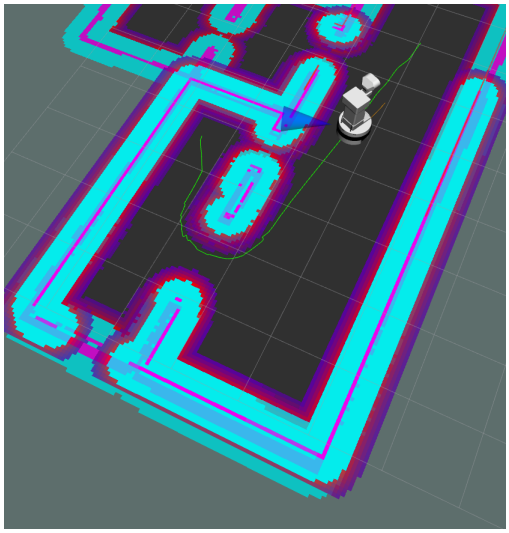
Rys. 7.6: Lokalna mapa kosztów z warstwą static oraz inflation



Rys. 7.7: Lokalna mapa kosztów z warstwą static, inflation oraz obstacle

7.0.5. *global_planner*

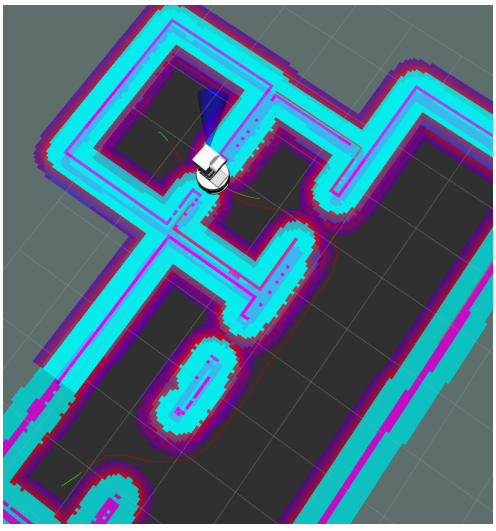
W celu wyznaczenia trasy do zadanego punktu zaimplementowano obsługę globalnego plannera przy użyciu klasy `global_planner::GlobalPlanner`. Jego działanie oparte jest o globalną mapę kosztów oraz parametry zdefiniowane w pliku `global_planner_params.yaml`. Ścieżka defionowana jest na podstawie algorytmu Dijkstry.Wywołanie glonalnego planera odbywa się jednokrotnie przy wyznaczeniu celu ruchu.



Rys. 7.8: Trasa wyznaczona przez planer globalny - na rysunku zieloną linią.

7.0.6. *local_planner*

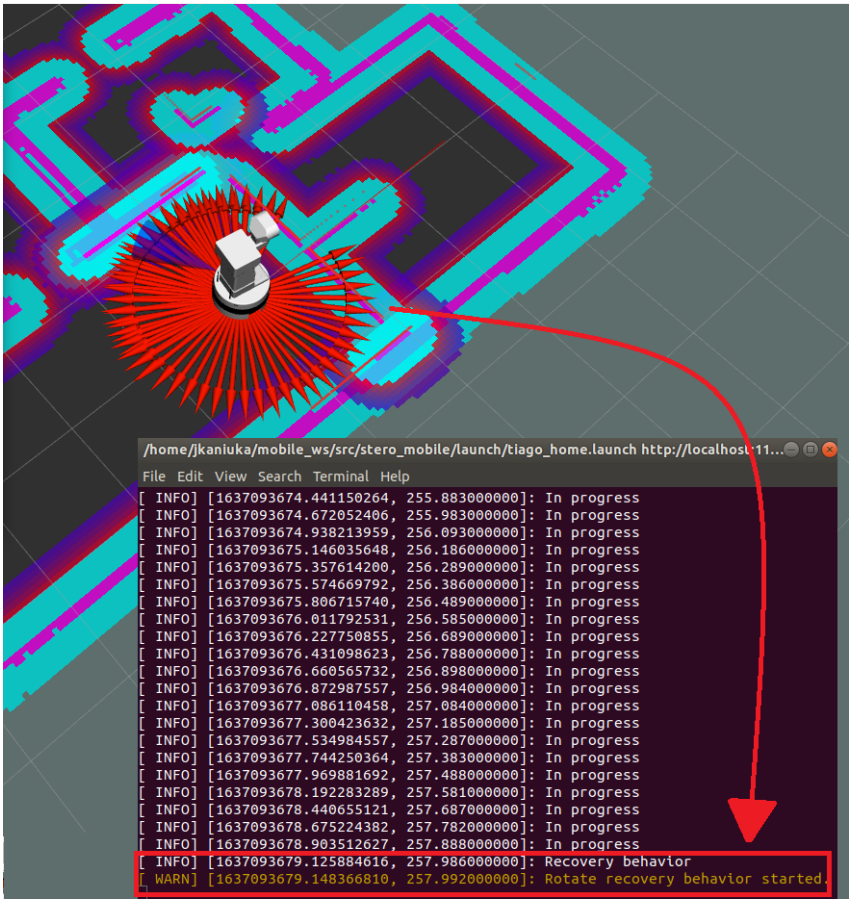
Wyznaczenie lokalnej trajektorii odbywa się poprzez algorytmy klasy `base_local_planner::TrajectoryPla`
Odpowiada ona za generowanie lokalnej ścieżki metodą DWA, na podstawie lokalnej mapy kosztów(mapa statyczna oraz lokalne przeszkodzy z warstwą inflation) oraz za wyznaczenie wektora prędkości. Planer lokalny pracuje w pętli do czasu, aż robot osiągnie swój cel, lub wyznaczenie trajektorii nie będzie możliwe.



Rys. 7.9: Trajektoria wyznaczona przez planer lokalny - na rysunku żółtą linią.

7.0.7. *recovery behavior*

Kiedy robot nie potrafi wygenerować ścieżki lub znajdzie się w położeniu z którego nie potrafi się wydostać rolę sterowania przejmuje algorytm klasy `rotate_recovery::RotateRecovery`. Jego działanie polega na wywołaniu obrotów robota do czasu, aż lokalny planer wyznaczy dostępną trajektorię. Dodatkowo wprowadzono czyszczenie lokalnej mapy kosztów, aby zniwelować problem "śladów" po usuniętych elementach. Poniżej widoczna jest grafika przedstawiająca robota w wyżej opisanej sytuacji (strzałki to wektory prędkości robota z temetu */odometry*).



Rys. 7.10: Recovery behavior.

Bardzo istotną kwestią podczas implementacji `rotate_recovery::RotateRecovery` jest mapowanie tematów, na którym publikowana jest prędkość robota.

```
<remap from="cmd_vel" to="key_vel"/>
```

Mapowanie można uwzględnić w pliku *.launch* lub w konsoli - każdorazowo przy uruchamianiu węzła sterujacego. `RotateRecovery` jest komponentem `NavigationStack` i z tego wynika fakt, że wszystkie prędkości są nadawane na temacie */cmd_vel* (a TiaGo przyjmuje wiadomości typu `Twist` na temacie */key_vel*).

7.0.8. Informacja o stanie robota

W celu informacyjnym oraz w celu ułatwienia debugowania zaimplementowaliśmy komunikaty informujące o stanie, w którym aktualnie znajduje się robot. Wykorzystaliśmy w tym celu mechanizm `ROS_INFO()`. Komunikaty są następujące:

— Goal passed successfully

— Planning
— In progress
— Recovery behavior
— Finish

8. Podsumowanie

8.0.1. Obserwacje i wnioski

Zaimplementowany moduł planowania został poddany licznym testom w środowisku symulacyjnym. Początkowo wyznaczanie ścieżki przez planer globalny nie działało idelanie - wynikało to z konfiguracji parametrów w plikach *.yaml*. Mnogość tychże parametrów była podstawową trudnością podczas wstępnych konfiguracji. Z czasem jednak udało nam się dobrać parametry, przy których planowanie ścieżki oraz jej wykonanie przez robota można było ocenić jako "zadowalające". Przy finalnie ustalonych wartościach parametrów robot był wstanie przejechać przez przejście o każdej z szerokości zaimplementowanych podczas laboratorium.

Na etapie testów badaliśmy również działanie planera lokalnego w reakcji na rekonfigurację środowiska. Podczas wykonywania ścieżki przez robota dodawaliśmy na trasie przeszkody w postaci brył (kula, sześcian). Robot omijał przeszkodę wyznaczając trasę przy wykorzystaniu modułu planowania lokalnego.