Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Sterowanie i symulacja robotów

Sprawozdanie z projektu i laboratorium nr 2
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 4.11.2021. Na laboratorium skupiliśmy się na budowie świata i jego mapy. W części projektowej mieliśmy za zadanie zaimplementować moduł planowania lokalnego.

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

2. Część laboratoryjna

2.0.1. 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.

2.0.2. 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.

2.0.3. Długi korytarz bez drzwi

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

2.0.4. Konfiguracja modułu SLAM

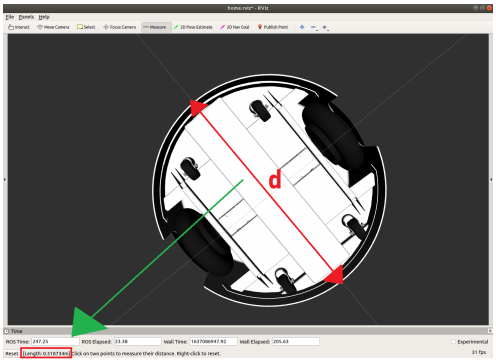
Korzystaliśmy z modułu `SLAM` (*Simultaneous Localization and Mapping*) wykorzystującego informacje z tematu */scan_raw*. Pozwoliło to na stworzenie dwuwymiarowej siatki zajętości.

2.0.5. 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.

2.0.6. Pomiar średnicy robota

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



Rys. 2.1: Pomiar średnicy bazy jezndej robota TiaGo.

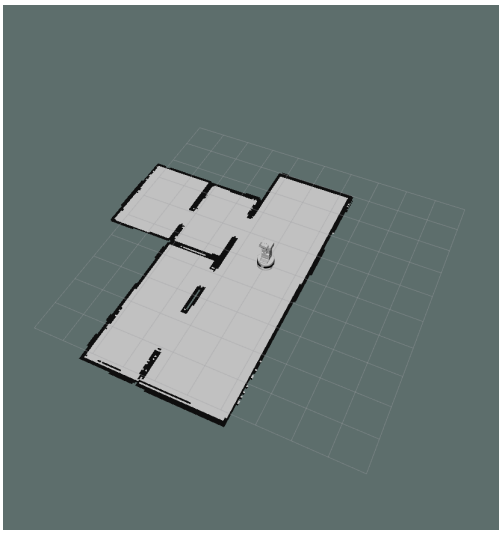
3. Część projektowa

3.0.1. 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.

3.0.2. *global_costmap* z warstwą *static*

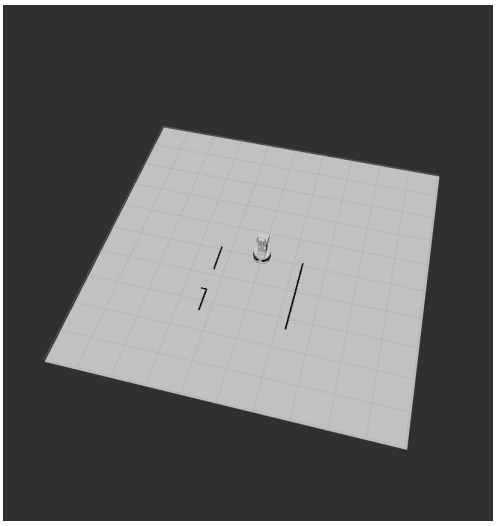
Następnie utworzyliśmy obiekt klasy `costmap_2d::Costmap2DROS`, w tym celu niezbędne było także zdefiniowanie parametrów mapy z warstwą static w pliku `global_costmap_params.yaml` oraz odpowiednia konfiguracja wspomnianego pliku w pliku startowym `tiago_home.launch`. Dokładne znaczenie wykorzytsanych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



Rys. 3.1: Globalna mapa kosztów z warstwą static

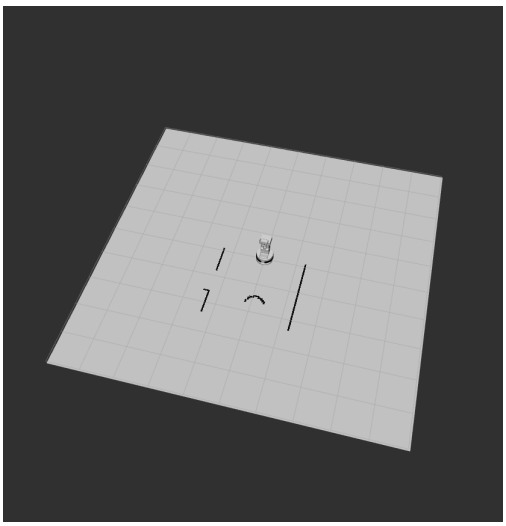
3.0.3. *local_costmap* z warstwami *static+obstacle*

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.



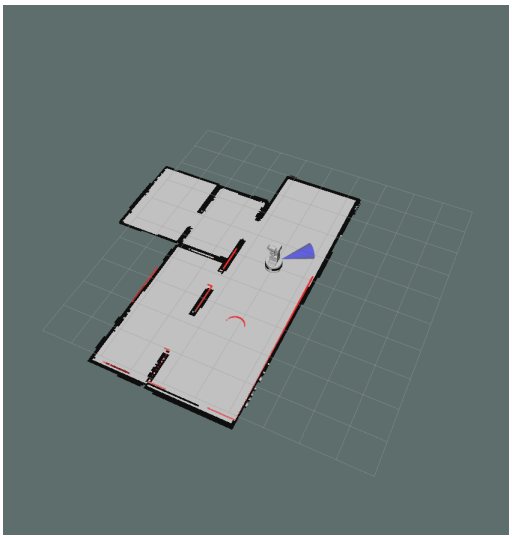
Rys. 3.2: Lokalna mapa kosztów z warstwą static

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



Rys. 3.3: Lokalna mapa kosztów z warstwą static oraz obstacle

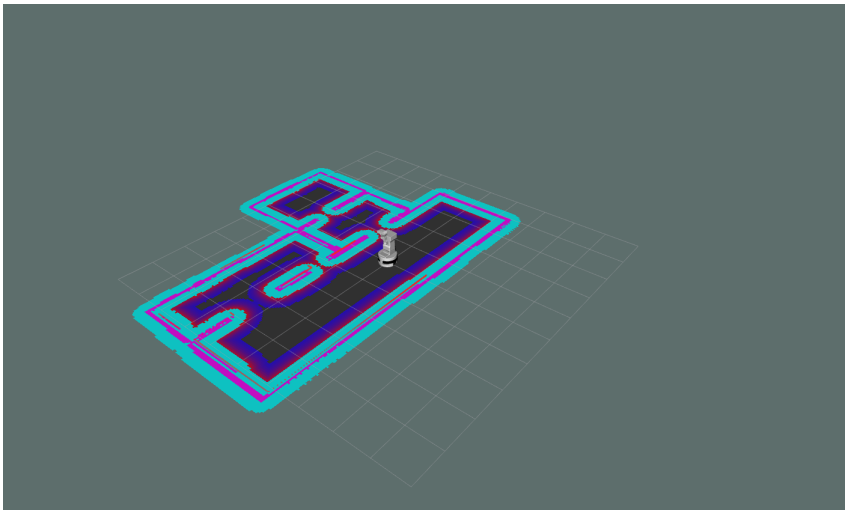
Dla porównania dokonano też testów z przeszkodami dla globalnej mapy kosztów, tak jak oczekiwaliśmy, globalna mapa nie uwzględnia lokalnych przeszkód.



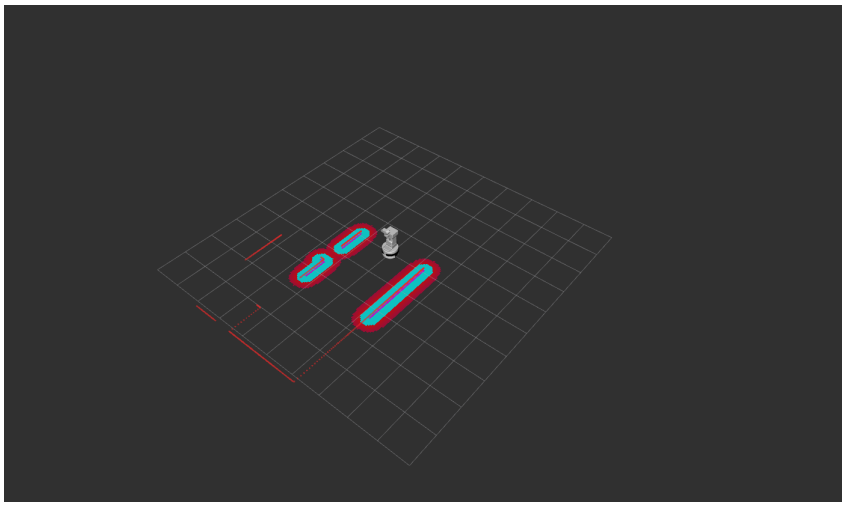
Rys. 3.4: Lokalna mapa kosztów z warstwą static

3.0.4. Dodanie warstwy *infalation* do obydwu map kosztów

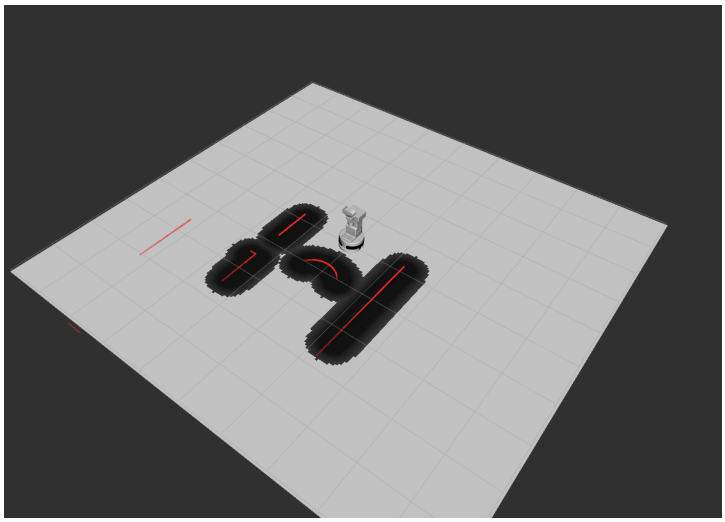
Po pomyślnych testach dla statycznych map kosztów przeszliśmy do konfiguracji warstw inflation. Polegało to głównie na dodaniu. w pliku konfiguracyjnym, parametrów: promienia inflacji oraz funkcji kosztu w zależności od odległości. Wygląd warstwy inflacji przedstawiono na rysunkach poniżej.



Rys. 3.5: Globalna mapa kosztów z warstwą static oraz inflation



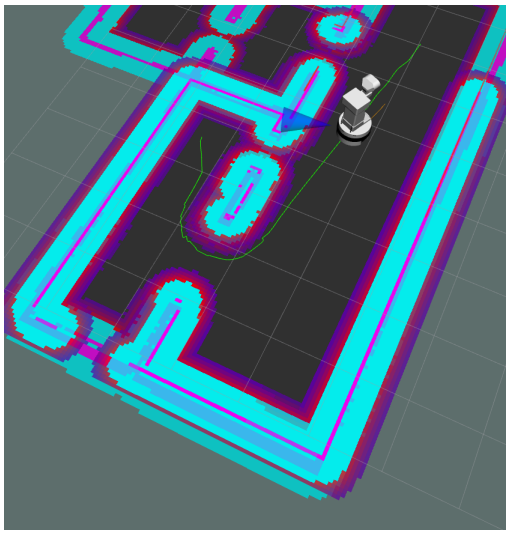
Rys. 3.6: Lokalna mapa kosztów z warstwą static oraz inflation



Rys. 3.7: Lokalna mapa kosztów z warstwą static, inflation oraz obstacle

3.0.5. *global_planner*

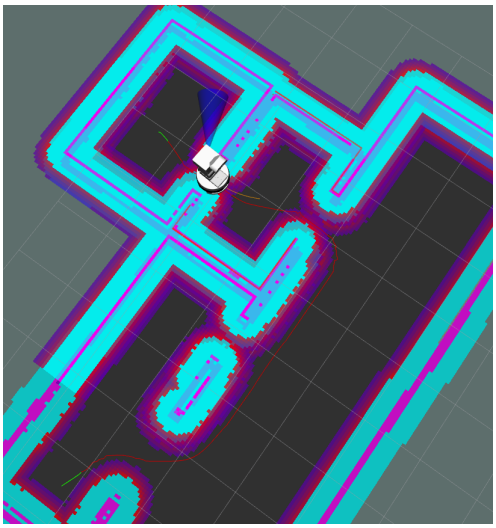
W celu wyznaczenia trasy do zadanego punktu zaimplementowano obsługę globalnego plannera przy użyciu klasy `global_planner::GlobalPlanner`. Jego działanie oparte jest o globalną mapę kosztów oraz parametry zdefiniowane w pliku `global_planner_params.yaml`. Ścieżka defionowana jest na podstawie algorytmu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



Rys. 3.8: Trasa wyznaczona przez planer globalny - na rysunku zieloną linią.

3.0.6. *local_planner*

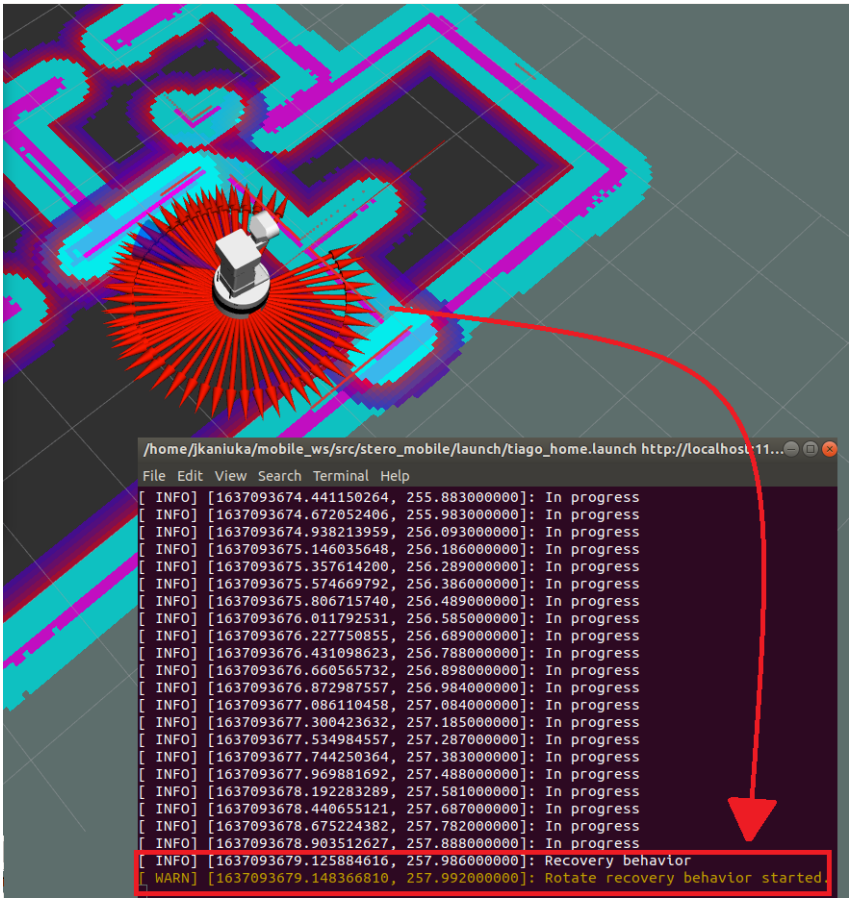
Wyznaczenie lokalnej trajektorii odbywa się poprzez algorytmy klasy `base_local_planner::TrajectoryPlannerROS`. Odpowiada ona za generowanie lokalnej ścieżki metodą DWA, na podstawie lokalnej mapy kosztów(mapa statyczna oraz lokalne przeszkodzy z warstwą inflation) oraz za wyznaczenie wektora prędkości.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



Rys. 3.9: Trajektoria wyznaczona przez planer lokalny - na rysunku żółtą linią.

3.0.7. *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. 3.10: 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*).

3.0.8. 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

4. Podsumowanie

4.0.1. 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.