POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY
INSTYTUT STEROWANIA I ELEKTRONIKI PRZEMYSŁOWEJ
PRACA DYPLOMOWA INŻYNIERSKA
na kierunku Automatyka i robotyka
specjalność: Automatyka

Wojciech Adam DUDEK
Nr albumu: 238879

Rok akademicki: 2013/2014
Warszawa, 15-10-2013

Wykorzystanie czujnika Kinect i systemu ROS do sterowania ruchem robota mobilnego

Zakres pracy:

1. *Wstęp*
2. *Opis wykorzystywanego oprogramowania, czujnika i robota*
3. *Konfiguracja niezbędnych bibliotek i pakietów w systemie ROS*
4. *Projekt i implementacja schematu sterowania robotem*
5. *Eksperymentalna weryfikacja działania robota*
6. *Podsumowanie i wnioski*

Kierujący pracą:

Termin złożenia pracy: 03-02-2014

Praca wykonana i obroniona pozostaje własnością Instytutu, Katedry i nie będzie zwrócona wykonawcy

Streszczenie

Niniejsza praca dotyczy zagadnień związanych z nawigacją i ruchem robota mobilnego. Głównym jej celem było przygotowanie i uruchomienie systemu autonomicznej nawigacji z wykorzystaniem czujnika **Kinect**. Założono, że robot poruszał się będzie do wskazanego punktu na mapie, a trajektoria ruchu będzie wyznaczana tak, aby platforma mobilna omijała przeszkody zarówno statyczne, jak i dynamiczne(np. człowiek ruchu).

Niezbędnymi krokami do realizacji zadania było poznanie **Robot Operating System - ROS**, czujnika Kinect oraz połączenie ich w funkcjonalną całość. Praca zawiera opis zaimplementowanego, skonfigurowanego i zmodyfikowanego otwartego oprogramowania. Zaprezentowano przebieg i wyniki weryfikacji sterowania robotem. Stwierdzono istotne trudności podczas: podłączania robota do komputera, przetwarzania danych czujnika oraz tworzenia mapy otoczenia. Następnie opisano rozwiązania powyższych problemów.

Projekt spełnia wszystkie założenia postawione w fazie koncepcyjnej. Zbudowano aplikację realizującą manualne sterowanie ruchem robota, wyznaczanie trajektorii ruchu oraz samodzielne omijanie napotkanych przeszkód. Dodatkową funkcjonalnością, w ramach pracy dyplomowej była budowa mapy 2D otoczenia robota. Wszystkie postawione cele pracy zostały zrealizowane.

Abstract

Topic: Application of the Kinect sensor and the ROS system to the control of the mobile robot movement

This thesis addresses problems of mobile robot navigation and movement. The main objective of this work was to create a system of autonomous navigation using a Kinect sensor. The idea was to make the robot move to the designated point on the map avoiding both static and dynamic (e.g. a moving human) obstacles.

Necessary step to accomplish the task was to combine the Robot Operating System - ROS and a Kinect sensor into a functional whole. The dissertation contains a description of implemented, configured and modified open source software. The process and verification results of robot control is presented. There were siginificant difficulties in process of: connecting the robot to the computer with ROS, transforming the Kinect sensor input stream and environment mapping. Solutions to the above issues are explained precisely throughout the dissertation.

The project meets all the assumptions in the conceptual stage. An application that allows not only manual control of a robot but also automatic path generation and autonomous obstacle evasion was implemented as well. The application provides additional functionality for creating a 2D map of the environment that was not included in the paper's proposal. All of the goals of the thesis have been met.

Politechnika Warszawska
Wydział Elektryczny

Warszawa, dnia 3 lutego 2014 roku.

OŚWIADCZENIE

Świadom/a odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska/magisterska pt.

Wykorzystanie czujnika Kinect i systemu ROS do sterowania ruchem robota mobilnego

- została napisana przeze mnie/nas samodzielnie
- nie narusza niczyich praw autorskich
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej.

Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Imię i Nazwisko dyplomanta: **Wojciech Dudek**

Podpis dyplomanta:

Spis Treści

1. Wstęp

W ostatnich dekadach zanotowano znaczące przyspieszenie rozwoju w wielu dziedzinach nauki. Powstawały nowe koncepcje, algorytmy i prototypy urządzeń, które zapoczątkowały postęp technologiczny i dały możliwość produkcji coraz tańszych i efektywniejszych podzespołów robotów. Wraz ze spadkiem cen rósł popyt, co doprowadziło do znacznego wzrostu liczby tych urządzeń. Znajdowano dla nich coraz więcej możliwości zastosowań i bez przerwy udoskonalano ich podzespoły. Roboty stawały się coraz szybsze, dokładniejsze, ekonomiczniejsze oraz bardziej autonomiczne.

Szybki rozwój robotyki doprowadził do powstania wielu grup robotów, różniących się parametrami, zastosowaniami i możliwościami. Ze względu na rodzaj przeznaczenia, roboty można podzielić na:

- ***roboty w służbie prawa:*** Najpopularniejszym zastosowaniem tych robotów jest rozbrajanie bomb. Charakteryzują się dużą stabilnością i mobilnością. Są to platformy mobilne z kamerą, silnym źródłem światła oraz niekiedy z ramieniem manipulującym. Zazwyczaj są one zdalnie sterowany przez operatora. Roboty te mogą również pełnić rolę sanitariusza w misjach wojskowych.
- ***roboty – zwiadowcy:*** Urządzenia te często są stosowane do eksploracji środowisk z pewnych przyczyn niedostępnych dla człowieka. Wzorem takiego robota może być platforma autonomiczna, zaopatrzona w kamery i manipulatory przeznaczona do pracy pod wodą. Do tej grupy zaliczamy również roboty przemierzające przestrzeń kosmiczną oraz penetrujące środowiska o wysokim skażeniu.
- ***roboty w rozrywce:*** Do tej grupy zaliczamy urządzenia imitujące pluszowe zabawki czy też interaktywne autonomiczne roboty przypominające zwierzęta. Są one wyposażone w zmysł wzroku, słuchu, dotyku i równowagi. Do tej kategorii zaliczamy również roboty występujące w różnych konkursach takich jak: BattleBots (roboty walczące) i RoboCup (roboty grające w piłkę nożną).
- ***roboty w nauce:*** Jest to dynamicznie rozbudowywana grupa robotów wykorzystywanych do eksperymentów naukowych. Jedną z bardziej zaawansowanych konstrukcji tej grupy jest robot zbliżony budową do człowieka.

- ***roboty w gospodarstwie domowym:*** W tej kategorii znajdują się roboty codziennego użytku, wykonujące prace w gospodarstwie domowym. Odkurzają, zmywają podłogi, czyszczą baseny, rynny, okna oraz koszą trawę.

Jak przedstawiono powyżej, robotyka ma szerokie zastosowanie w wielu branżach oraz korzysta z rozwiązań nowych technologii. Biorąc to pod uwagę, postanowiono zgłębić problem jakim jest ruch i nawigacja platform autonomicznych. Jest to jedna z podstawowych funkcji robotów, więc prowadzony projekt będzie mógł być rozwijany w wielu kierunkach.

1.1 *Cele pracy*

Głównym celem pracy było przygotowanie i uruchomienie systemu autonomicznej nawigacji robota mobilnego z wykorzystaniem czujnika Kinect. Aby zrealizować ten cel należało zapoznać się z zasadami działania systemu ***Robot Operating System – ROS*** oraz sposobami komunikacji między robotem a czujnikiem.

ROS jest podstawowym narzędziem, dzięki któremu można poznać zagadnienia i problemy związane ze sterowaniem robotami mobilnymi. Niniejsza praca miała więc również zweryfikować funkcjonalność systemu ***ROS*** oraz możliwości jakie niesie ze sobą jego połączenie z czujnikiem ***Kinect***.

1.2 *Założenia projektowe*

Oczekuje się, że robot będzie poruszał się w pełni autonomicznie, po wyznaczonej przez siebie trajektorii, do wskazanego punktu na mapie. Platforma ma omijać przeszkody, zarówno statyczne, jak i dynamiczne (np. człowiek w ruchu), których wysokość minimalna wynosi 30 cm.

Komunikacja między robotem, a komputerem sterującym ma odbywać się poprzez port USB 2.0 komputera. To założenie jest w pełni uzasadnione, gdyż USB 2.0 jest standardowym i powszechnie używanym portem komunikacyjnym.

Dodatkową funkcjonalnością projektu ma być generowanie mapy środowiska zewnętrznego robota oraz zapisywanie jej w standardowym dla systemu formacie. Należy też zoptymalizować zużycie procesora i innych zasobów komputera, aby umożliwić ewentualny, dalszy rozwój projektu.

2. Opis wykorzystywanego oprogramowania, czujnika i robota

2.1 *Oprogramowanie – Wprowadzenie do systemu ROS*

Robot Operating System (ROS) jest elastycznym środowiskiem do pisania i uruchamiania oprogramowania dla robotów. Zawiera wiele narzędzi, bibliotek i algorytmów. System upraszcza pisanie zarówno złożonych funkcji, jak i budowanie całych projektów oprogramowania, na różnego rodzaju roboty. Tworzenie potężnego oprogramowania dla robota ogólnego użytku jest trudne. Odruchy komunikacyjne i poznawcze człowieka stają się złożonym procesem technologicznym i informatycznym w przypadku próby implementacji ich w środowisku robota. Zakładając tylko zmienną intensywność światła, można napotkać poważne problemy przy sterowaniu optycznym robota. Oprogramowanie dla robotów jest tak skomplikowane i zawiera tyle zmiennych, że stworzenie dobrego środowiska przez jedną instytucję byłoby pracochłonne, nieopłacalne, lub nawet nieosiągalne.

Związku z powyższym, zbudowano ROS od podstaw, udostępniono jego źródło oraz opublikowano szczegółową dokumentację, aby wszyscy twórcy oprogramowania dla robotów mogli wziąć udział w rozbudowie tego systemu. Zaproponowano współpracę przy aktualizacji i rozbudowie systemu wszystkim twórcom oprogramowania dla robotów. Biorąc pod uwagę zróżnicowanie grup tworzących oprogramowanie, tak skonfigurowano system, aby specjaliści mogli dobudowywać kolejne pakiety, współpracujące z już istniejącymi. Dzięki pracy z ROS student może zapoznać się z możliwościami jakie daje otwarte oprogramowanie udoskonalane przez wielu specjalistów z całego świata.

System ROS posiada największe wsparcie na platformie Ubuntu, jednakże istnieją również wersje eksperymentalne działające na następujących platformach: *Debian, Windows, OS X OpenEmbedded/Yocto, UDOO.*

Instalacja systemu

Aby zainstalować ROS w wersji Hydro na platformę Ubuntu należy wejść na stronę http://wiki.ros.org/hydro/Installation/Ubuntu. Znajduje się tam dokładny opis kolejnych kroków niezbędnych do przeprowadzenia poprawnej instalacji systemu.

Po złożonej konfiguracji środowiska należy pobrać i zainstalować system ROS. Rekomendowane jest pobranie wersji Desktop-Full, ponieważ zawiera ona najwięcej niezbędnych, dla tego projektu, pakietów i bibliotek.

W celu aktualizacji indeksu pakietu Debian oraz pobraniu środowiska należy wywołać poniższe komendy:

```
sudo apt-get update
sudo apt-get install ros-hydro-desktop-full
```

Powyższy krok należy powtarzać przed każdą instalacją dodatkowych pakietów i narzędzi w systemie ROS. Kolejnymi punktami są: instalacja, inicjalizacja i aktualizacja pakietu ***rosdep***. Pozwala on na kompilowanie zależności systemowych źródła ROS oraz jest niezbędny do uruchomienia niektórych składników podstawowych systemu. Aktualizację tego pakietu należy stosować, kiedy jest podejrzenie pojawienia się nowych definicji, których zainstalowany dotychczas rosdep nie posiada.

```
sudo apt-get install python-rosdep
sudo rosdep init
rosdep update
```

Ostatnim krokiem jest ustawienie zmiennych środowiska, aby powłoka systemowa Ubuntu – ***bash*** załadowała przy starcie terminalu odpowiednie pliki konfiguracyjne systemu. W tym celu należy wywołać komendę:

```
echo "source /opt/ros/hydro/setup.bash" >> ~/.bashrc
```

oraz następnie :

```
source ~/.bashrc
```

2.1.1 Zarządzanie środowiskiem

Środowisko ROS łatwo przystosowuje się do różnego rodzaju zestawów pakietów i daje możliwość łączenia różnych wersji ROS. Jest to wykonalne dzięki dołączaniu poszczególnych plików setup.*sh lub nawet dodania dołączania do skryptu startu powłoki.

Organizacja i budowa kodu systemu może być dwojakiego rodzaju – ***catkin,*** lub ***rosbuild***. Zważając na większą uniwersalność organizacji ***catkin*** niż r***osbuild***, w pracy wykorzystano pierwszy z wymienionych. Poniżej znajduje się krótka charakterystyka wybranej organizacji:

Catkin – oficjalny system służący do kompilacji ***ROS***. Jest następcą podstawowego systemu budowania ***ROS – rosbuild.*** Łączy makra CMake oraz skrypty Python aby dostarczyć kilka dodatkowych funkcji. Catkin charakteryzuje się większą funkcjonalnością niż rosbuild, ponieważ pozwala na lepszą dystrybucję pakietów, lepsze wsparcie skrośnego kompilowania

oraz jest przenośny na inne środowiska. System ten jest bardzo zbliżony do CMake, ale dodatkowo wspiera automatyzację infrastruktury ***package find*** oraz budowę wielu zależnych od siebie projektów w tym samym czasie.

Środowisko pakietów jest to miejsce, w którym będą przechowywane dodatkowe pakiety i biblioteki systemu ROS. Aby zainicjować i stworzyć to środowisko należy wywołać komendy:

```
catkin_init_workspace
```

- zostaje stworzony plik konfiguracyjny ***CMakeLists.txt***

```
catkin_make
```

- tworzy podfoldery ***build*** i ***devel***

Nawigacja w systemie plików ROS

Poniższe komendy pozwalają szybko poruszać się między pakietami i lokalizacjami w systemie ROS:

```
rospack find [nazwa_pakietu]
```

- Zwraca ścieżkę do pakietu

```
roscd [nazwa_lokalizacji[/podfolder]]
```

- Przemiezczenie się do lokalizacji pakietu

```
roscd log
```

- Pliki logów uruchomionych programów ROS

```
rosls [nazwa_lokalizacji[/podfolder]]
```

- Wylistowanie plików pakietu

```
Klawisz Tabulatora
```

- Uzupełnienie nazw pakietów
- Listowanie nazw, jeśli jest ich więcej i rozpoczynają się na dany ciąg znaków

Tworzenie przykładowego pakietu catkin

Pierwszym krokiem do stworzenia pakietu jest przemieszczenie się do źródła środowiska:

```
cd ~/catkin_ws/src
```

Następnie tworzymy przykładowy pakiet ***pierwszy_pakiet*** zależny od pakietów ***std_msgs***, ***roscpp*** oraz ***rospy***:

```
catkin_create_pkg pierwszy_pakiet std_msgs roscpp rospy
```

Dostosowanie pakietów - opis i edycja plików wygenerowanych przez catkin_create_pkg

Plik package.xml:

- **Description(opis)** – krótki opis zakresu pakietu,
- **Maintainer(opiekun)** – email i nazwa opiekuna,
- **Licence(licencja)** – typ licencji,
- **URL(linki)** – atrybuty: type=”webside” lub „bugtracker” lub „repository”,
- **Author(twórca)** – email i nazwa
- **Build Tool Dependencies** – narzędzia potrzebne do zbudowania pakietu, istnieją dla architektury na której przebiega kompilacja,
- **Build Dependencies** – pakiety potrzebne do budowy pakietu dla kierunkowej architektury,
- **Run Dependencies** – pakiety potrzebne do działania kodu tego pakietu lub do budowy bibliotek o tym pakiecie.

Plik CMakeLists.txt:

- **Cmake_minimum_required -** Wymagana wersja CMake,
- **Project()** - Nazwa pakiet,
- **Find_package() -** Znajdywanie innych pakietów CMake lub Catkin,
- **Add_message_files(), add_service_files(), add_action_files()** - Inicjalizacja generatorów wiadomości, serwisów, lub akcji,
- **Generate_messages()** - Wywołanie generacji wiadomości, serwisu lub akcji,

- **Catkin_package()** - Specify package build info export,
- **Add_library()/add_executable()/target_link_libraries()** - Biblioteki lub pliki wykonacze, które trzeba wcześniej zbudować,
- **Catkin_add_gtest()** - Testy budowy,
- **Install()** - Zasady instalacji.

2.1.2 Pliki wykonawcze systemu i komunikacja między nimi

Węzeł - jest plikiem wykonawczym w pakiecie ROS lub procesem wykonującym obliczenia. Węzły są połączone i komunikują się ze sobą poprzez tematy, usługi RPC oraz serwery parametrów. Budowane są zazwyczaj do obsługi niewielkich części projektu - system sterowania robotami zazwyczaj składa się z wielu węzłów. Na przykład jeden węzeł jest odpowiedzialny za obsługę kamery, drugi wyznaczanie trasy, a jeszcze inny steruje pracą silników robota.

Należy zwrócić uwagę na to, że system zbudowany na podstawie komunikujących się ze sobą węzłów posiada wiele zalet. Jedną z nich jest odseparowanie awarii reszty kodu od błędu jednego węzła. Po uszkodzeniu jednego węzła, na przykład wizualizacji, reszta pracuje nadal bez zarzutu i robot osiągnie ustalony punkt w przestrzeni. Dodatkowo kod projektu jest mniej złożony, co ułatwia zapoznawanie się z jego funkcjami i zapewnia lepsze dostosowanie systemu do stawianych mu zadań.

Każdy węzeł w systemie charakteryzuje się nazwą i typem. Dzięki temu można się odwoływać do kolejnych części kodu poprzez wywołanie nazwy poszczególnego węzła i kolejno nazwy pliku wykonawczego. System ROS odnajduje najpierw żądany pakiet, przeszukuje go i uruchamia/zwraca plik wykonawczy o żądanej nazwie. Należy więc uważać, aby każdy plik wykonawczy w pakiecie miał swoją indywidualną nazwę.

Biblioteki klienckie (client libraries) ROS – wykorzystywane do pisania węzłów:

- ***rospy*** *-> pyton client library*
- ***roscpp*** *-> C++ client library*

Poniżej przedstawiono podstawowe narzędzia do analizy pakietów i węzłów z poziomu konsoli:

- **roscore** – pakiet-master który zarządza innymi uruchamianymi pakietami zapewnia komunikację między nimi,
- **rosnode** – wyświetla informacje o działających węzłach, zawiera polecenia:

- **list** – wyświetla działające pakiety
- **info [nazwa_węzła]** – wyświetla, które tematy są publikowane, subskrybowane oraz które usługi są uruchomione przez dany węzeł.
- **ping** - test
- **rosrun [nazwa_pakietu nazwa_węzła]** – załączenie węzła przez wywołanie nazwy jego pakietu.

Komunikacja między węzłami – tematy (topics)

Tematy – są to szyny danych charakteryzujące się indywidualną nazwą. Za ich pomocą przesyłane są wiadomości między węzłami - jeden z nich publikuje wiadomości w danym temacie, a drugi zaś subskrybuje dany temat i tym samym otrzymuje publikowane w nim wiadomości. Nie wnikając w szczegóły działania tematów można powiedzieć, że węzeł publikujący wiadomości nie wie czy jest zapotrzebowanie na wysyłane przez niego wiadomości.

Schemat działania tematów pozwala na istnienie wielu nadawców i wielu odbiorców poszczególnej informacji. Sposób działania tematów został stworzony jednak z myślą o jednokierunkowej komunikacji. Węzły, które potrzebują komunikować się ze sobą w obu kierunkach powinny korzystać z systemu usług. Dodatkowo istnieje serwer parametrów, który służy do gromadzenia i udostępniania niewielkiej ilości stanów/parametrów.

Generowanie grafu komunikacji:

Aby wygenerować graf komunikacji między uruchomionymi węzłami należy uruchomić węzeł rqt_graph:

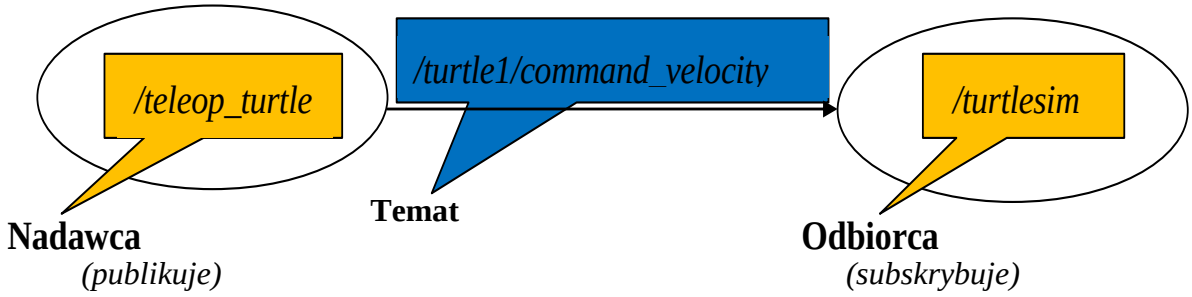
```
sudo apt-get install ros-hydro-rqt
```

- Pobranie pakietu węzła rqt_graph,

```
rosrun rqt_raph rqt_graph
```

- Uruchomienie węzła generującego plik PDF z grafem na szczycie środowiska catkin.

Przykładowy graf komunikacji między węzłami przedstawia rys. 1.



Rys. 1 *Przykładowy graf komunikacji między węzłami*

Nawigacja między tematami:

Do wyświetlania parametrów tematów w systemie ROS służy polecenie ***rostopic.*** Wywołuje się je z poniższymi parametrami:

- **-h** -> pomoc
- **bw** -> przepustowość tematu (szerokość pasma)
- **echo** -> wiadomości publikowane w tym temacie
- **hz** -> częstotliwość publikowania
- **list** -> aktywne tematy
- **type** -> typ tematu

Wiadomości – wprowadzenie i przykład:

Komunikacja w systemie przebiega przez wysyłanie wiadomości do odpowiednich tematów. Przesyłane informacje zazwyczaj mają postać prostych struktur danych. Poza danymi typu integer, floating point lub boolean wspierane są wiadomości w postaci tablic i struktur podobnych do tych w języku programowania C.

Oznacza to, że typ tematu zależy od typu wiadomości w nim publikowanych. Sprawdza się go poprzez:

```
rostopic type [nazwa_tematu]
```

Znając typ wiadomości można wyświetlić jej szczegóły na przykład:

```
rosmsg show [typ_wiadomości]
```

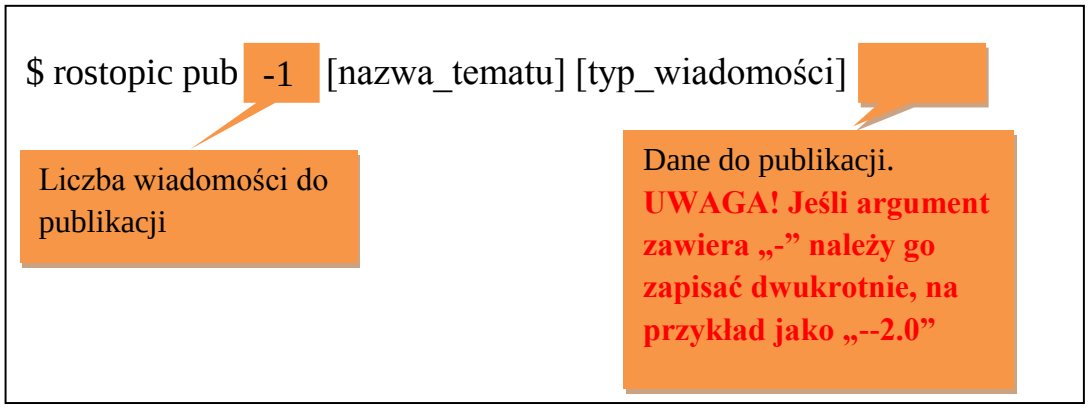
oraz otrzymać stosowną odpowiedź:

```
float32 linear
floar32 angular
```

Dzięki tej informacji wiadomo, jakiego rodzaju wiadomości oczekuje węzeł subskrybujący dany temat.

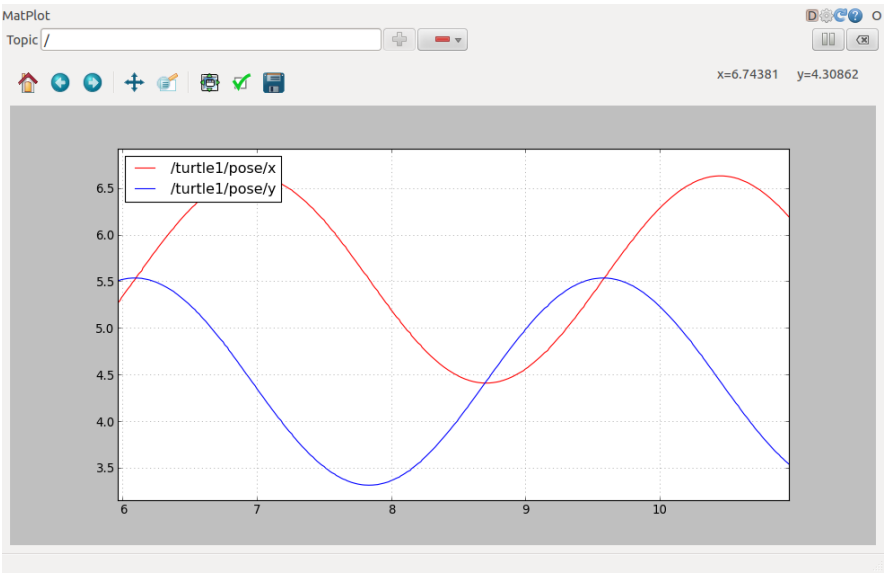
Przykładowe publikowanie wiadomości i wykres publikacji od czasu:

Poniżej zaprezentowano i objaśniono podstawową komendę publikującą wiadomości:



Rys. 2 *Komenda publikująca wiadomości*

Chcąc wyznaczyć wykres danych tematu w zależności od czasu należy uruchomić węzeł ***rqt_plot.*** Przykładowy wykres przedstawia ***rys. 3.***



Rys. 3 *Wykres pozycji x i y robota w zależności od czasu, źródło: [1]*

Usługi i parametry

Podstawowa komunikacja w systemie dokonywana jest przez wiadomości publikowane/pobierane z określonych tematów. Jest to jednak komunikacja jednokierunkowa. Często w projekcie niezbędnym jest udostępnienie usługi publikującej serię danych w odpowiedzi na żądanie innego węzła. Taką funkcję realizuje się przez uruchomienie węzła świadczącego usługę. Konfiguruje się go za pomocą plików srv. Węzeł zainteresowany informacją usługodawcy wysyła do niego żądanie i oczekuje na odpowiedź. Można ustawić stałe połączenie z usługą, co skutkuje zwiększeniem wydajności połączenia.

Nawigacja między usługami:

Aby zarządzać usługami należy użyć komendy ***rosservice*** z poniższymi parametrami:

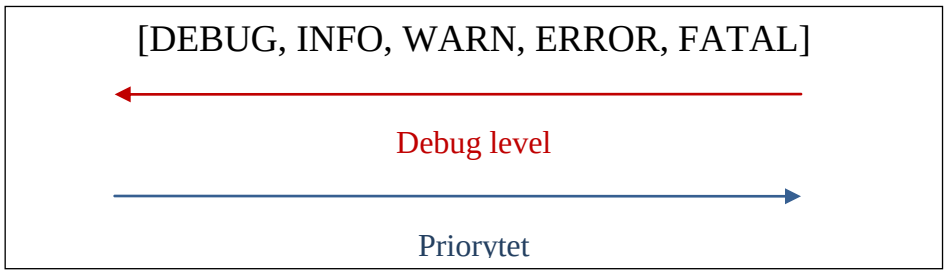
- **list** – aktywne usługi
- **call [paramtery usługi]–** wywołanie usługi z podanymi parametrami
- **type –** typ usługi
- **find –** znajdź po nazwie
- **uri –** ROSRPC uri usługi

Zmiana parametrów w systemie:

- **Rosparam –** komenda zarządzająca danymi w *ROS Parameter Server*
- **Ros Parameter Server –** przechowuje typy zmiennych:
  - integer (1)
  - float (1.0)
  - string (one)
  - booleans (true)
  - lists [1,2,3]
  - dictionary {a:b, c:d}
- **Parametry komendy *rosparam***:
  - **set ->** ustaw
  - **get ->** wyświetl wartość (rosparam get /) – cały Parameter Server
  - **load ->** pobierz z pliku
  - **dump ->** wyślij do pliku (params.yaml)
  - **delete ->** usuń
  - **list ->** wyświetl wszystkie nazwy parametrów

Debugging oraz uruchamianie jednocześnie

Aby wyświetlić informacje wyjścia uruchomionych węzłów należy wywołać komendę ***rqt_console.*** Natomiast, aby zmienić debug level (poziom ważności wyświetlanych informacji) należy zastosować komendę ***rqt_logger_level***. Na ***rys. 4*** zaprezentowano zależność pomiędzy debug level a priorytetem wyświetlanych informacji.



Rys. 4 *Układ wzrostu debug level oraz priorytetu*

Uruchomienie kilku węzłów jednocześnie

Aby uruchomić kilka węzłów jednocześnie, należy najpierw stworzyć plik o rozszerzeniu .launch, a następnie uruchomić go przy pomocy komendy:

```
roslaunch [nazwa_pakietu] [plik.launch]
```

Poniżej zaprezentowano i opisano przykładowy plik ***uruchom.launch***:

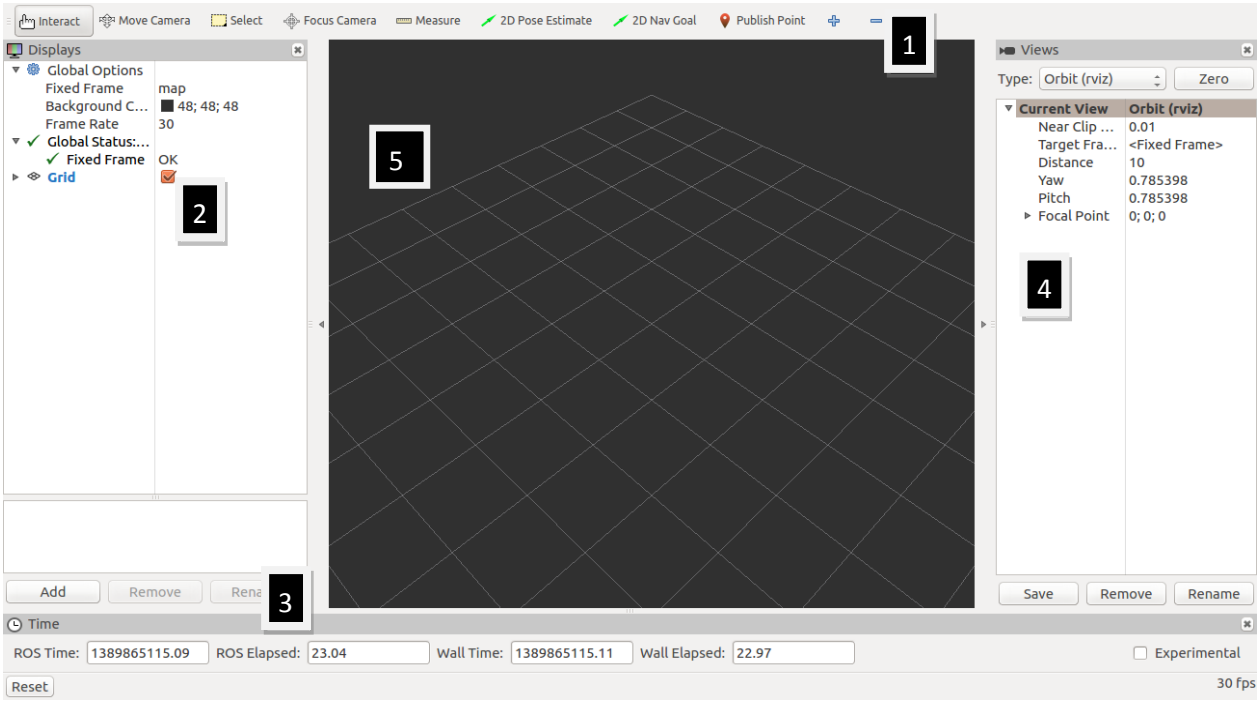
```
1 <launch>
3 <group ns="turtlesim1">
4 <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
5</group>
6
7<group ns="turtlesim2">
8 <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
9 </group>
10
11<node pkg="turtlesim" name="mimic" type="mimic">
12<remap from="input" to="turtlesim1/turtle1"/>
13<remap from="output" to="turtlesim2/turtle1"/>
14</node>
15</launch>
```

Źródło: http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch

Powyższy plik uruchamia dwie grupy: ***turtlesim1*** oraz ***turtlesim2***, zaś każda grupa uruchamia węzeł ***turtlesim***. Podział na grupy zastosowano, aby uniknąć konfliktu nazw przy równoczesnym uruchomieniu dwóch jednakowych węzłów. W linii numer 11 załączany jest węzeł ***mimic***, który ma za zadanie subskrybować temat ***turtlesim1/turtle1*** i publikować identyczne wiadomości na temat ***turtlesim2/turtle1.*** W praktyce powyższy plik realizuje funkcję naśladowania. Robot z grupy ***turtlesim2*** będzie zachowywał się identycznie jak ten z ***turtlesim1***.

2.1.3 Wizualizator Rviz

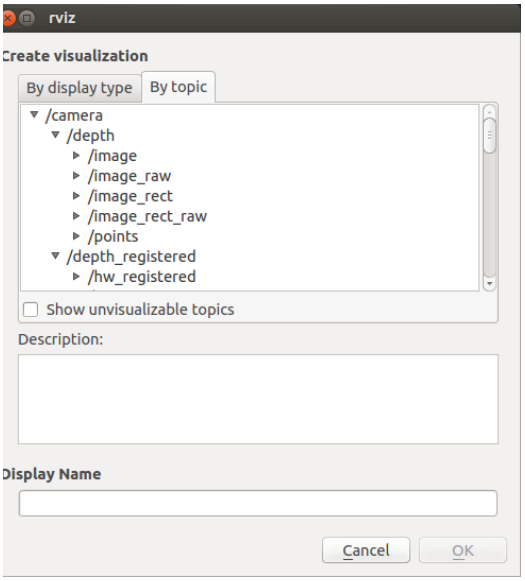
Rviz **(rys. 5)** jest to pakiet udostępniający program wizualizacyjny wbudowany w system ***ROS***. Został on użyty w projekcie do wyświetlania: danych z czujnika Kinect, map, zaplanowanych trajektorii ruchu robota, symbolicznego modelu robota oraz do wskazywania położenia i orientacji poszczególnych układów współrzędnych. Program umożliwia operatorowi również publikowanie wiadomości dotyczących ustalenia aktualnej pozycji i orientacji robota (narzędzie ***2D Pose Estimate***) oraz wyznaczenia punktu docelowego ruchu dla nawigacji (narzędzie ***2D Nav Goal***).



Rys. 5 *Główne okno wizualizatora Rviz,* **1** – Górna linijka wyboru narzędzi, **2** – Okno edycji wyświetlanych obiektów, **3** – Przyciski dodawania, usuwania i zmiany nazwy wyświetlanego obiektu, **4** – Okno edycji pozycji obserwatora, **5** – Przestrzeń wizualizacji

Program udostępnia różne sposoby manipulacji pozycją obserwatora. Przy włączonym **module orbitalnego manewrowania**, do poruszania się po aktualnej orbicie należy posługiwać się **LPM**, natomiast aby przemieścić punkt centralny orbity należy przytrzymać klawisz **Shift** oraz manewrować myszą z wciśniętym **LPM**.

Po naciśnięciu przycisku **Add** (panel [3]) otworzy się poniższe okno dialogowe:

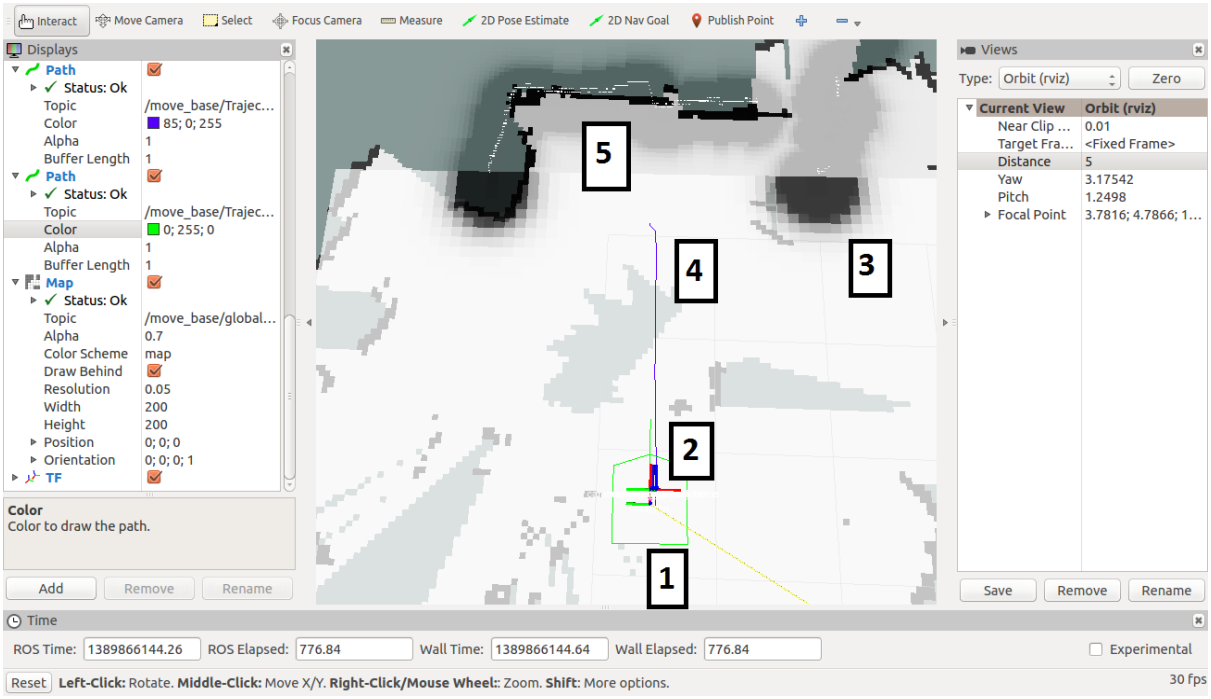


Rys. 6 *Okno dodawania obiektów wizualizacji*

Użytkownik ma możliwość dodawania obiektów w dwojaki sposób. Pierwsza zakładka – ***By display type*** daje możliwość dodania obiektu znając jego typ wyświetlania, natomiast druga zakładka – ***By topic*** grupuje obiekty, których dane są publikowane w tematach systemu oraz można je poddać wizualizacji.

Należy pamiętać jednak, że wybierając obiekty z zakładki ***By display type*** należy je dodatkowo poddać edycji w oknie wyświetlanych obiektów ***(rys. 5)***. Niezbędne na przykład jest wskazanie tematu, z którego obiekt ma pobierać dane do wizualizacji.

Poniżej (**rys. 7**) zamieszczono odpowiednio skonfigurowaną wizualizację na potrzeby prowadzonego projektu:



***Rys. 7** Okno skonfigurowanego wizualizatora,* **1** – sylwetka robota, **2** – układy współrzędnych robota, **3** – rzeszkoda zaznaczona na mapie lokalnej, **4** – planowana trajektoria ruchu, **5** – odczyt z czujnika i zaznaczona przeszkoda na mapie globalnej.

2.2 *Sprzęt – działanie czujnika Kinect oraz charakterystyka robota Pioneer P3-DX*

W tej sekcji omówiono działanie i scharakteryzowano użyte w projekcie elementy techniczne. Dobór tych elementów przeprowadzono na podstawie założeń projektowych opisanych w punkcie ***1.2*** niniejszej pracy. W pierwszej części podrozdziału zaprezentowano zastosowany czujnik pomiaru głębi Kinect, a następnie scharakteryzowano użytą w projekcie platformę mobilną.

2.2.1 Kinect – budowa i działanie czujnika

Kinect jest optyczno-akustycznym czujnikiem, który ze względu na swoje właściwości znalazł szerokie zastosowanie w robotyce. Główną zaletą tego czujnika jest niski koszt mapowania przestrzeni 3D i 2D. Kinect dzięki swoim właściwościom i niskiej cenie może zastąpić drogie czujniki laserowe. Urządzenie to zostało opracowane z myślą o analizowaniu i wyznaczaniu ruchu poszczególnych części ciała człowieka. W związku

z tym niektóre funkcjonalności tego czujnika nie zostały wykorzystane w opracowanym projekcie. W tym rozdziale skupiono się wyłącznie na opisie zastosowanych funkcjonalności.

Rys. 8 *Kinect, źródło: materiały producenta*

Sprzęt

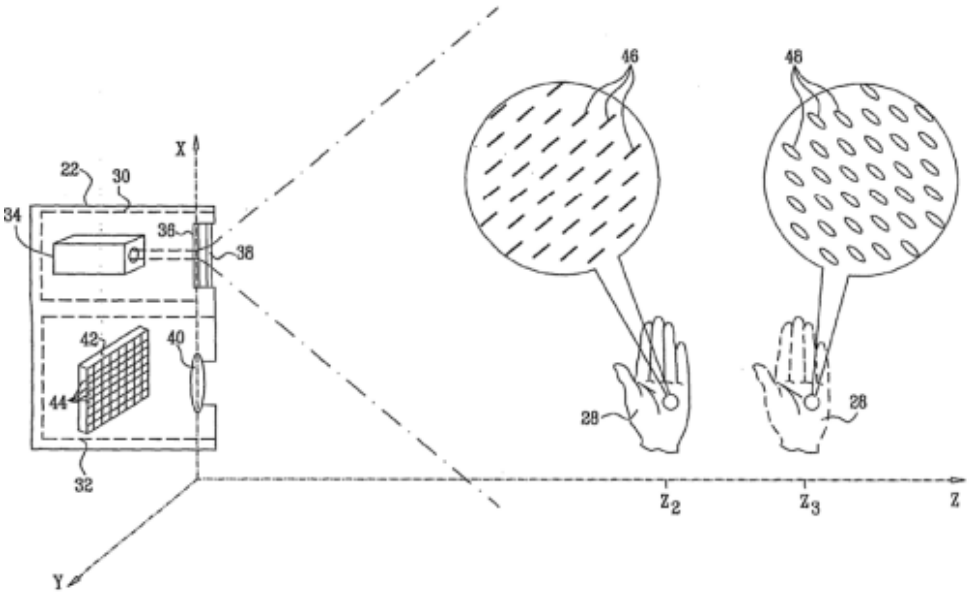
Podstawowym działaniem czujnika Kinect jest wykrywanie ruchu fizycznego i wysłanie odpowiednich danych do urządzenia odbiorczego. Funkcję wykrywania ruchu spełnia czujnik głębi. Jest on zbudowany jest z promiennika światła strukturalnego oraz z matrycy CMOS. Służy on do pomiaru odległości poszczególnych punktów od czujnika. W projekcie wykorzystano również wbudowany silnik zmieniający kąt nachylenia modułu czujnika.

Działanie

Wyznaczenie mapy odległości punktów środowiska jest dokonywane przez analizę chmury punktów. Są one generowane przez odbite światło lasera. Oświetlacz działający w paśmie podczerwieni wysyła znany wzór świetlny w przestrzeń. Następnie matryca zbiera odbicia, a oprogramowanie analizuje jego zniekształcenie w stosunku do pierwotnie wysłanego wzoru. Technika zastosowana w Kinect nazywana jest światłem strukturalnym. Czujnik analizuje zniekształcenia wzoru świetlnego z wykorzystaniem dwóch podstawowych komputerowych technik wizyjnych:

- **Wyznaczanie głębi przez zmianę ostrości –** wraz z odległością od czujnika maleje ostrość obrazu. W celu zwiększenia dokładności tej techniki w Kinect zastosowano astygmatyczne soczewki o różnych ogniskowych w kierunkach x i y. Przy korzystaniu z tej techniki rzucane wzory świetlne w kształcie okręgów stają się elipsami zmieniającymi swoją orientację

w zależności od odległości. Schemat działania przedstawia poniższy obraz (**rys. 9**).



Rys. 9 *schemat działania soczewek astygmatycznych, źródło:[2]*

- **Wyznaczanie głębi metodą stereo –** w tej technice wykorzystywany jest efekt paralaksy. Występuje on przy obserwowaniu tego samego obiektu z różnych miejsc w przestrzeni. Porównując dwa obrazy środowiska zebrane z różnych punktów obserwacji można wyznaczyć głębię, ponieważ obiekty znajdujące się bliżej zostają przesunięte znaczniej niż te znajdujące się w oddali. W czujniku Kinect wykorzystano wprawdzie pojedynczą kamerę na podczerwień (zdolną do rejestracji rzutowanych wzorów świetlnych), ale przesunięto ją wyraźnie w stosunku do projektora światła strukturalnego, co powoduje że wzór ten ulega innemu przesunięciu na obiektach bliskich, a innemu na odległych – głębię oświetlonych obiektów wyznacza się więc na podobnej zasadzie do stereowizji.

2.2.2 Charakterystyka robota Pioneer P3-DX

Zastosowany w projekcie robot o napędzie różnicowym jest jednym z najbardziej rozpowszechnionych robotów mobilnych stosowanych do celów badawczych. Wszechstronność i niezawodność wybranej platformy sprawiły, że wyznacza ona parametry odniesienia dla badań w robotyce. Platforma jest dostarczana z odpowiednio skonfigurowanym i wbudowanym kontrolerem. Robot zawiera silniki z enkoderami o 500

impulsach na obrót, 19 centymetrowe koła, mocną aluminiową obudowę oraz 8 ultradźwiękowych sonarów z przodu i z tyłu. Po podłączeniu komputera zewnętrznego robot jest gotowy do pracy.

Rys. 10 *Robot Pioneer P3-DX, źródło:[materiały producenta]*

Robot Pioneer P3-DX może osiągnąć prędkość 1,6 m/s, a jego maksymalna ładowność wynosi 23 kg. Robot zasilany jest trzema akumulatorami żelowymi hot-swap 12V/9Ah. Kontroler robota sprawdza prędkość robota i dostarcza informacje na temat stanu robota; wyznacza pozycję robota względem układu środowiska zewnętrznego (x, y, theta) oraz udostępnia podstawowe parametry stanu platformy.

Producent udostępnił środowisko programowania wysokopoziomowego Pioneer SDK. Powstały kod po kompilacji może być uruchamiany na komputerze pokładowym lub w trybie offline, na bezpłatnym symulatorze MobileSim. Pioneer P3-DX jest platformą uniwersalną, wykorzystywaną do badań związanych z:

- mapowaniem
- teleoperowaniem
- lokalizacją
- monitorowaniem
- rekonesansem
- wizją
- manipulacją
- autonomiczną nawigacją
- współpracą

Pioneer P3-DX sprawdza się najlepiej na twardych podłożach. Robot jest zdolny do pokonywania niskich progów oraz wspinania się na rampy dla niepełnosprawnych.

3. Konfiguracja i charakterystyka niezbędnych bibliotek i pakietów w systemie ROS

3.1 *Dobór bibliotek i pakietów*

ROS jest rozwijany przez społeczność wielu developerów, przez co można napotkać pakiety o podobnych (czasami też identycznych) właściwościach, które: działają w inny sposób, subskrybują i publikują inne tematy, współpracują z różnymi bibliotekami, są kompatybilne z odmiennymi wersjami systemu. Część z nich przestała być też aktualizowana i stała się niekompatybilna z najnowszymi wersjami systemu ROS.

Przy wyborze narzędzi należy zwrócić szczególną uwagę na dokumentację opublikowaną na stronie http://wiki.ros.org. Wspomniany portal internetowy zawiera niezbędne informacje na temat wspieranych elementów systemu, między innymi: krótką charakterystykę, możliwości zastosowania, typ licencji, adres do pobrania źródła, spis niezbędnych bibliotek i pakietów do działania elementu, przykłady lub tutoriale, adres raportów błędów oraz interfejs programowania aplikacji (ang. Application Programming Interface, API). Ostatnia z wymienionych informacji jest szczególnie pomocna w kontekście zrozumienia pobieranych i publikowanych przez dany pakiet wiadomości. W API każdego pakietu zaprezentowane są subskrybowane i publikowane tematy systemu ROS. Te informacje pozwalają developerowi na odpowiedni dobór elementów tworzonego projektu, tak aby nazwy i typy odpowiednich tematów były jednakowe. Niekiedy może być tak, że niezbędne pakiety nie posiadają zamienników i należy je dostosować do siebie. Drugim ważnym krokiem, istotnym głównie dla osób początkujących, jest sprawdzenie dostępności wsparcia społeczności developerów systemu. W tym celu zaleca się pobieżne przeszukać forum: http://answers.ros.org/, gdzie zamieszczane są pytania i rozwiązania problemów użytkowników systemu.

Wybierając biblioteki i pakiety do realizacji prowadzonego projektu kierowano się przede wszystkim: dobrze rozwiniętą dokumentacją, popularnością stosowania przez programistów oraz szerokim wsparciem ze strony użytkowników systemu.

Po dokładnej analizie problemu nawigacji robotem mobilnym i zapoznaniu się z dostępnymi narzędziami wyznaczono najlepsze rozwiązania dla projektu.

3.2 *Obsługa czujnika Kinect - Pakiet openni_launch*

Prawidłową pracę czujnika i ustalenie komunikacji dwustronnej między nim, a systemem ***ROS*** zapewnia sterownik – pakiet ***openni_launch***. Programy tego pakietu zapewniają konwersję danych czujnika (obrazów raw depth/RGB/IR) na obrazy głębi oraz chmury punktów. Wspomniany pakiet jest podstawą tego projektu i jest niezbędny do jego realizacji. Chcąc pobrać i zbudować jego pliki należy wykonać poniższe kroki:

- Zaktualizować narzędzie apt-get oraz zainstalować narzędzie portalu Github uruchamiając poniższe komendy:

```
sudo apt-get update
sudo apt-get install git
```

- Przejść do folderu źródła środowiska pakietów:

```
roscd catkin_ws/src
```

- Pobrać źródło pakietu ***openni_launch***:

```
git clone https://github.com/ros-drivers/openni_launch.git
```

- Zainstalować wszystkie zależności pakietu:

```
sudo rosdep update
sudo rosdep install openni_launch
```

Po instalacji trzeba zbudować na nowo środowisko pakietów catkin_ws. W tym celu należy:

- Wrócić na szczyt środowiska:

```
roscd catkin_ws
```

- Ponownie zbudować środowisko pakietów:

```
catkin_make
```

Po udanej instalacji pakiet jest gotowy do użycia i można uruchomić sterownik komendą:

```
roslaunch openni_launch openni.launch
```

3.3 *Imitacja czujnika laserowego przy użyciu Kinecta – pakiet turtlebot_bringup*

W projekcie zastosowano pakiet przekształcający strumień danych z czujnika Kinect na strumień danych imitujący laser. Ten zabieg musiał być zrealizowany, gdyż przyspieszył on znacznie wyznaczanie miejsc niedostępnych dla robota. Bez jego pomocy moc obliczeniowa zastosowanego komputera nie byłaby wystarczająca na płynne przetwarzanie danych czujnika i wyznaczanie trasy ruchu. Taka optymalizacja nie przynosi żadnych skutków ubocznych, gdyż z założeń robot poruszał się będzie na płaszczyźnie.

Pakiet ***turtlebot_bringup***, który między innymi realizuje wspomnianą wcześniej konwersję danych, jest częścią projektu ***Turtlebot***. W celu pobrania pakietu ***turtlebot_bringup*** i zainstalowania wszystkich jego zależności należy:

- Przejść do źródła środowiska pakietów:

```
roscd catkin_ws/src
```

- Pobrać projekt ***Turtlebot***:

```
sudo git clone https://github.com/turtlebot/turtlebot.git
```

- Zainstalować wszystkie zależności pakietu ***turtlebot_bringup:***

```
sudo rosdep update
sudo rosdep install turtlebot_bringup
```

Po instalacji należy poddać edycji plik **3dsensor.launch**, tak aby wyglądał on zgodnie z **załącznikiem nr 1**. Następnie trzeba zbudować na nowo środowisko pakietów catkin_ws. W tym celu należy:

- Wrócić na szczyt środowiska:

```
roscd catkin_ws
```

- Ponownie zbudować środowisko pakietów:

```
catkin_make
```

Po udanej instalacji i konfiguracji można uruchomić narzędzie komendą:

```
roslaunch tortlebot_bringup 3dsensor.launch
```

3.4 *Komunikacja z robotem – pakiet RosAria*

Standardowym środowiskiem do komunikacji z robotem ***Pionieer P3-DX*** jest program ***Aria***. Chcąc w pełni korzystać z możliwości wspomnianej platformy mobilnej, należy zastosować pakiet ***RosAria.*** Komunikuje się on z robotem przekazując polecenia zmiany prędkości publikowane w temacie systemu ***ROS*** oraz publikuje w odpowiednim temacie wiadomości odometryczne otrzymane od robota.

W celu pobrania tego pakietu należy:

- Upewnić się czy repozytoria *„universe”* są odkomentowane w pliku */etc/apt/sources.list*, następnie zainstalować narzędzie portalu Mercurial uruchamiając poniższą komendę:

```
sudo apt-get install mercurial
```

- Przejść do źródła środowiska pakietów:

```
roscd catkin_ws/src
```

- Pobrać pakiet ***RosAria***:

```
sudo hg clone http://code.google.com/p/amor-ros-pkg/
```

- Zainstalować wszystkie zależności pakietu ***RosAria:***

```
sudo rosdep install ROSARIA
```

Po instalacji trzeba zbudować na nowo środowisko pakietów catkin_ws. W tym celu należy:

- Wrócić na szczyt środowiska:

```
roscd catkin_ws
```

- Na nowo zbudować środowisko pakietów:

```
catkin_make
```

Po udanej instalacji można uruchomić pakiet komendą:

```
rosrun ROSARIA RosAria
```

Poniżej zaprezentowano **właściwości pakietu RosAria:**

- **Subskrybowane tematy:**
  - ***cmd_vel*** (typ wiadomości: geometry_msgs/Twist) – otrzymuje polecenia prędkości. Używa wartości liniowych podanych jako zmienna x i rotacyjnych podanych jako zmienna z. Pobrana wiadomość jest archiwizowana, więc nie musi być ponownie publikowana aż do zmiany swojej wartości.
- **Publikowane tematy:**
  - ***pose*** (typ wiadomości: nav_msgs/Odometry) – publikuje wiadomości odometryczne, z częstotliwością zależną od modelu robota. W przypadku robota Pioneer P3-DX częstotliwość ta wynosi 10 Hz.
  - ***bumper_state*** (typ wiadomości: ROSARIA/BumperState) – publikuje stany czujników zderzeniowych
  - ***sonar*** (typ wiadomości: sensor_msgs/PointCloud) – publikuje odczyty sonarów. Jeśli temat nie jest subskrybowany, sonary zostają wyłączone.
- **Ważny parametr:**
  - ***port*** (typ wiadomości: string, domyślny: /dev/ttyUSB0) - Seryjny port urządzenia do którego podłączany jest robot, lub nazwa hosta i port TCP oddzielony dwukropkiem, na przykład: 12.0.111.22:8011

3.5 *Węzeł transformacji układów współrzędnych – pakiet własny*

Do prawidłowej pracy nawigacji należało zbudować własny pakiet, który będzie odpowiedzialny za dostarczanie informacji o różnicy w przesunięciu i orientacji między odpowiednimi układami współrzędnych. Pierwszym etapem do stworzenia własnego pakietu jest jego budowa w środowisku. Dokonuje się tego podążając krokami zaprezentowanymi w rozdziale ***2.1.1***. Należy jednak pamiętać o wprowadzeniu odpowiednich pakietów, od których budowany pakiet jest zależny. W przypadku prowadzonego projektu są nimi: ***geometry_msgs***, ***roscpp***, ***tf***. Wygenerowane pliki zaleca się poddać edycji według schematu zaprezentowanego w rozdziale ***2.1.1***.

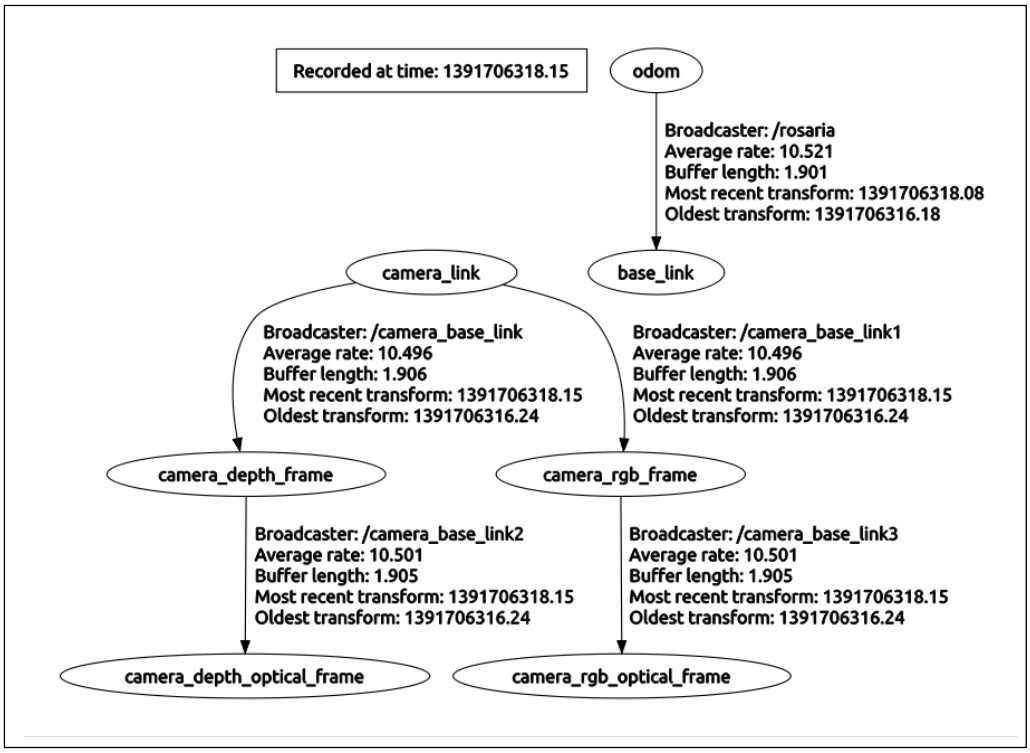
Mając przygotowany wcześniej pakiet, można rozpocząć analizę niezbędnych właściwości węzła. Dla uproszczenia można wykorzystać drzewo transformacji, które jest generowane przez jeden z podstawowych węzłów systemu. Chcąc poznać obecny stan drzewa w systemie podążano następującymi krokami:

- Uruchomiono wszystkie wykorzystywane w projekcie węzły

- Włączono węzeł tworzący drzewo transformacji:

```
rosrun rqt_tf_tree rqt_tf_tree
```

- Dokonano analizy otrzymanego schematu:



Rys. 11 *Gałęzie transformacji układów*

Z powyższych gałęzi należy zbudować jedno drzewo, w taki sposób aby transformacje i orientacje były zgodne z rzeczywistością. W przypadku prowadzonego projektu należało dostarczyć systemowi informację na temat przekształcenia układu ***base_link*** do ***camera_link*** oraz układu ***map*** do ***odom***. Podstawowymi narzędziami do przedstawiania przekształceń układów w systemie ***ROS*** są kwaterniony i wektory. Chcąc dokładnie odwzorować transformacje, wyznaczono odpowiednie współrzędne rzeczywistego położenia czujnika względem środka podstawy robota oraz napisano kod węzła w pliku ***tf_broadcaster.cpp***.

Kod węzła tf_broadcaster

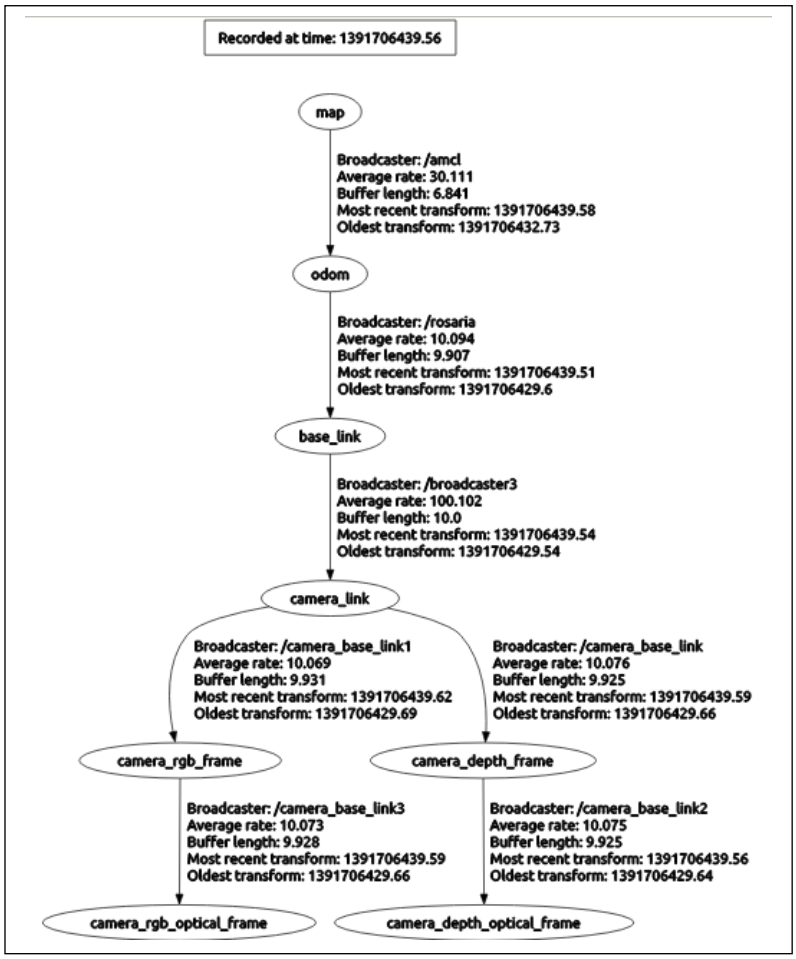
```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "robot_tf_publisher");
  ros::NodeHandle n;

  ros::Rate r(100);

  tf::TransformBroadcaster broadcaster;
```

```
 while(n.ok()){
   broadcaster.sendTransform(
     tf::StampedTransform(
       tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.3)),
       ros::Time::now(),"base_link", "camera_link"));
   r.sleep();
}
while(n.ok()){
   broadcaster.sendTransform(
     tf::StampedTransform(
       tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(3.0, 3.0, 0.0)),
       ros::Time::now(),"map", "odom"));
   r.sleep();
}

}
```

Po napisaniu kodu zmodyfikowano plik ***CMakeLists.txt*** pakietu węzła, tak aby kompilator dołączył plik ***tf_broadcaster.cpp***. Niezbędnych zmian dokonano w poniższych liniach pliku ***CmakeLists.txt***:

```
## Declare a cpp executable
add_executable(tf_broadcaster src/tf_broadcaster.cpp)

## Specify libraries to link a library or executable target against
 target_link_libraries(tf_broadcaster
   ${catkin_LIBRARIES} )
```

Następnie ponownie uruchomiono wszystkie wykorzystywane w projekcie węzły i wygenerowano nowy graf transformacji. Schemat po poprawnie zastosowanych przekształceniach zamieszczono poniżej:



Rys. 12 *Drzewo transformacji układów po konfiguracji*

3.6 *Algorytmy i programy nawigacyjne – stos navigation stack*

W tym podrozdziale została opisana implementacja stosu ***navigation stack*** w systemie ***ROS***. Jest to niezbędny element zawierający pakiety, węzły i biblioteki sterujące ruchem robota, analizujące środowisko oraz wyznaczające trajektorię ruchu. Zasada działania stosu jest dość prosta. Pobiera on wiadomości odometryczne i strumień informacji z czujnika, analizuje je, buduje mapę przestrzeni i ostatecznie publikuje żądania zmian prędkości. Bardziej skomplikowane jest dostosowanie i zastosowanie nawigacji. Aby mieć możliwość zastosowania Navigation stack robot musi: być sterowany przez ROS, mieć kompletne drzewo transformacji układów oraz publikować dane czujników w odpowiednim systemowym formacie.

W celu pobrania i zbudowania pakietu należy wykonać następujące kroki:

- Przejść do źródła środowiska pakietów:

```
roscd catkin_ws/src
```

- Pobrać stos ***navigation stack***:

```
sudo git clone https://github.com/ros-planning/navigation
```

- Zainstalować wszystkie zależności ***navigation stack:***

```
sudo rosdep update
sudo rosdep install navigation
```

Po instalacji trzeba zbudować na nowo środowisko pakietów catkin_ws. W tym celu należy:

- Wrócić na szczyt środowiska:

```
roscd catkin_ws
```

- Ponownie zbudować środowisko pakietów:

```
catkin_make
```

Po wykonaniu powyższych instrukcji pakiet jest gotowy do użycia.

3.7 *Mapowanie pomieszczenia – pakiet gmapping*

Budowa mapy i dołączenie jej do nawigacji w systemie ROS są opcjonalne. Dokładne odwzorowanie i zbadanie możliwości nawigacji, w prezentowanym systemie, wymusza tworzenie mapy środowiska. W stosie ***Navigation stack*** mapa statyczna służy do dokładnej aproksymacji położenia robota. Tę funkcję spełnia algorytm ***amcl*** oparty o ***adaptacyjną aproksymację lokalizacji Monte Carlo*** lub ***KLD-sampling***. Wspomniana procedura zwiększa skuteczność filtru cząsteczkowego optymalizując wielkość próbki. Algorytm pobiera małą liczbę punktów publikowanych przez laser, jeśli gęstość ich rozmieszczenia jest na małym obszarze przestrzeni i dużą liczbę próbek, gdy niepewność stanu próbek jest wysoka.

Jednym z podstawowych pakietów obsługujących tworzenie mapy środowiska jest pakiet ***gmapping***. Aby go pobrać i zbudować, należy:

- Przejść do źródła środowiska pakietów:

```
roscd catkin_ws/src
```

- Pobrać projekt ***gmapping***:

```
sudo git clone https://github.com/ros-perception/slam_gmapping.git
```

- Zainstalować wszystkie zależności pakietu ***gmapping:***

```
sudo rosdep update
sudo rosdep install gmapping
```

Po instalacji trzeba zbudować na nowo środowisko pakietów catkin_ws. W tym celu należy:

- Wrócić na szczyt środowiska:

```
roscd catkin_ws
```

- Ponownie zbudować środowisko pakietów:

```
catkin_make
```

Po realizacji powyższych kroków można przystąpić do tworzenia własnej mapy. Jak zapisano w dokumentacji pakietu na stronie: http://wiki.ros.org/gmapping, mapa tworzona jest w oparciu o dane w dwóch tematach:

- ***tf*** (tf/tfMessage) – transformacje układów współrzędnych
- ***scan*** (sensor_msgs/LaserScan) – dane z czujnika laserowego

Przed uruchomieniem programu tworzącego mapę należy przygotować sterownik czujnika Kinect (pakiet ***openni_launch***) oraz dostarczyć systemowi wiadomości na temat transformacji układu podstawy robota do układu ogólnego (pakiet ***własny***). Po niezbędnych przygotowaniach:

- Włączono mapowanie wpisując w terminalu:

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

- Skonfigurowano odpowiednio wizualizator ***Rviz*** w celach obserwacyjnych,
- Uruchamiając program map saver pakietu map_server ze stosu ***navigation stack*** zapisano mapę do pliku o nazwie ***moja_mapa***:

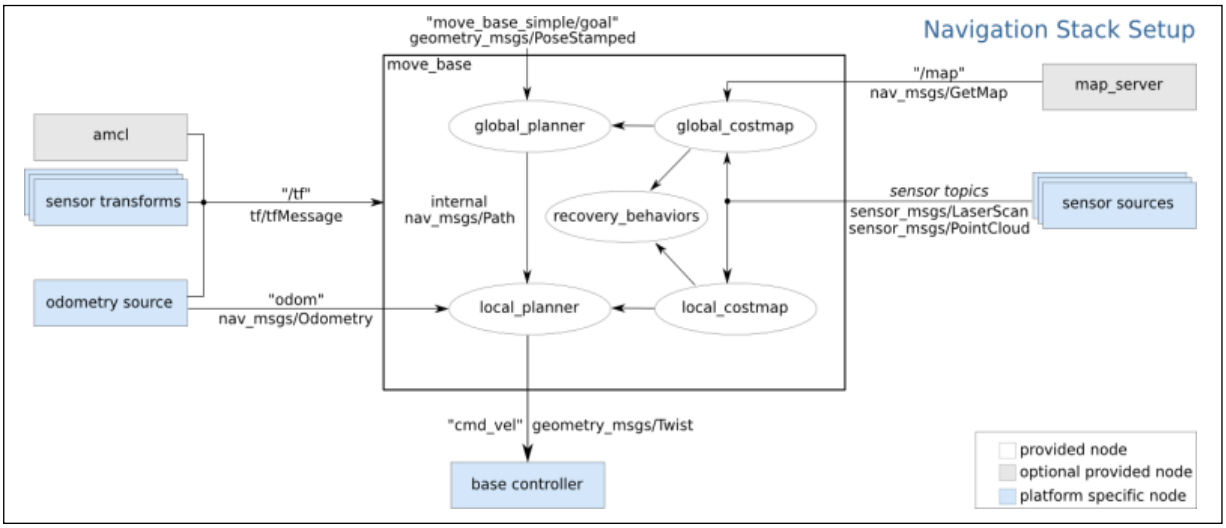
```
rosrun map_server map_saver -f moja_mapa
```

4. Projekt, implementacja schematu sterowania robotem oraz eksperymentalna weryfikacja jego działania

W systemie ***ROS*** do nawigacji używany jest projekt ***Navigation stack.*** Jest to duży projekt zawierający wiele bibliotek i pakietów, który do prawidłowego działania potrzebuje: odometryczne wiadomości robota, strumienie danych z czujników, miejsce docelowe ruchu. Po analizie wyżej wymienionych danych algorytm publikuje w odpowiednim temacie odpowiednie żądania zmiany prędkości ruchu robota.

4.1 *Schemat działania stosu Navigation stack oraz opis jego elementów*

Poniżej zaprezentowało graf wyjaśniający drogi komunikacji między elementami projektu ***Navigation stack*** oraz opisano elementy niezbędne do uruchomienia nawigacji.



Rys. 13 *Graf przesyłu informacji w projekcie Navigation stack, źródło: [1]*

Elementy zawarte w sekcji ***move_base***, są podmiotami dostępnymi po instalacji ***Navigation stack.*** Małe szare prostokąty symbolizują składowe systemu nawigacji, które można opcjonalnie dołączyć, zaś niebieskie prostokąty przedstawiają niezbędne węzły, które nie należą do projektu ***Navigation stack***.

W cudzysłowach nad strzałkami widnieją nazwy tematów, które są subskrybowane przez podmioty ***move_base,*** zaś pod strzałkami można odczytać typy wiadomości publikowanych w powyższych tematach. Podsumowując, projekt ***Navigation stack*** subskrybuje wiadomości:

- **odometryczne robota i czujników** – temat: „odom”
- **transformacji układów współrzędnych –** temat: „/tf”
- **miejsca docelowego ruchu** – temat: „move_base_simple/goal”
- **mapy statycznej (jeśli dostępna) –** temat: „/map”
- **strumienie danych od czujników –** tematy których wiadomości mają typ: „sensor_msgs/LaserScan”, lub „sensor_msgs/PointCloud”

Poniżej omówiono charakterystykę każdego z jej elementów.

- **amcl –** system probabilistycznej lokalizacji dla poruszającego się robota w przestrzeni 2D. Realizuje adaptacyjne lokalizowanie za pomocą algorytmu przybliżenia Monte Carlo (lub próbkowanie KLD), który wykorzystuje filtr do śledzenia pozycji robota względem statycznej mapy,
- **sensor transforms –** zbiór węzłów publikujących wiadomości na temat przekształceń układów współrzędnych czujników względem układu robota (zazwyczaj ***base_link***),
- **odometry source –** zbiór węzłów publikujących wiadomości odometryczne w temacie ***odom***,
- **map_server –** węzeł uruchamiający usługę wysyłania danych map. Dostarcza też narzędzie ***map_saver***, które pozwala na zapisywanie dynamicznie generowanych map,
- **sensor sources –** zbiór węzłów publikujące strumienie danych z czujników,
- **base controller –** węzeł lub zbiór węzłów subskrybujących temat ***cmd_vel*** i wysyłających pobrane żądania zmian prędkości do robota,
- **global_costmap –** element tworzący siatkę globalną całej przestrzeni w celu wybrania najbardziej optymalnej trajektorii ruchu. Przechowuje informację o przeszkodach,
- **global_planner –** biblioteka raz z węzłem, wyznacza trasę ruchu robota,
- **local_costmap –** element odpowiedzialny za tworzeni, lokalnie wokół robota, siatki wartości. Są one dobierane wg funkcji kosztów ruchu w danym kierunku. Przechowuje informację o przeszkodach,
- **local_planner –** pakiet uruchamiający kontroler, który porusza robotem po mapie. Kontroler ten zapewnia połączenie między planowaną trajektorią ruchu a robotem. Zadaniem regulatora jest

wyznaczenie, przy pomocy mapy, takich zmian wartości x, y i theta prędkości robota, aby osiągnął on miejsce docelowe.

- **recovery_behaviors –** element węzła ***move_base***. Odpowiadający za wykonanie odpowiednich czynności wyswobadzających robota z różnego rodzaju nieplanowanego zatrzymania lub zakleszczenia.

4.2 *Budowa pakietu konfigurującego nawigację*

Pierwszym krokiem do konfiguracji ***Navigation stack*** jest stworzenie pakietu w źródle środowiska zależnego od odpowiednich elementów zewnętrznych. Analizując powyższy schemat można zaobserwować pakiety zewnętrzne, których informacje muszą być dostarczone do nawigacji. W przypadku prowadzonego projektu są nimi:

- ***move_base*** – główny pakiet nawigacji zawierający m.in. algorytmy naprowadzania
- ***RosAria*** – pakiet wysyłający wiadomości odometryczne i odbierający od nawigacji wiadomości zadanych prędkości
- ***robot3_tf*** – pakiet udostępniający niezbędne transformacje układów współrzędnych
- ***openni_launch*** – pakiet przechowujący dane czujnika
- ***turtlebot_bringup*** – pakiet generujący końcowy strumień analizowanych danych (imitacja skanów laserowych)

W celu stworzenia takiego pakietu postąpiono wg schematu z punktu ***2.1.1*** podstawiając do komendy powyższe zależności i skonfigurowano pliki ***CMakeLists.txt*** oraz ***package.xml***. W efekcie otrzymano poniższą strukturę:

```
- catkin_ws
  -   …
  -   mojrobot (stworzony pakiet)
      -   CMakeLists.txt
      -   package.xml
```

Rys. 14 *struktura-1 pakietu konfiguracyjnego Navigation stack*

Pliki powstałe po procesie edycji, to jest: CMakeLists.txt oraz package.xml zostały załączone do pracy.

Następnie napisano właściwe pliki konfiguracyjne dla nawigacji:

- ***my_robot_configuration.launch*** – załącza węzły zewnętrzne współpracujące z nawigacją
- ***costmap_common_params.yaml*** – plik konfiguracyjny dla strumieni danych czujników
- ***global_costmap_params.yaml*** – plik konfiguracjny parametrów global_costmap
- ***local_costmap_params.yaml*** – plik konfiguracyjny parametrów local_costmap
- ***base_local_planner_params.yaml*** – plik konfiguracyjny przetwarzania żądań zmian prędkości
- ***move_base.launch*** – plik uruchamiający utworzone pliki konfiguracyjne, algorytm nawigacji oraz serwer mapy

Wszystkie powyższe pliki znajdują się w załączniku do pracy – ***załącznik – pliki konfiguracji nawigacji***.

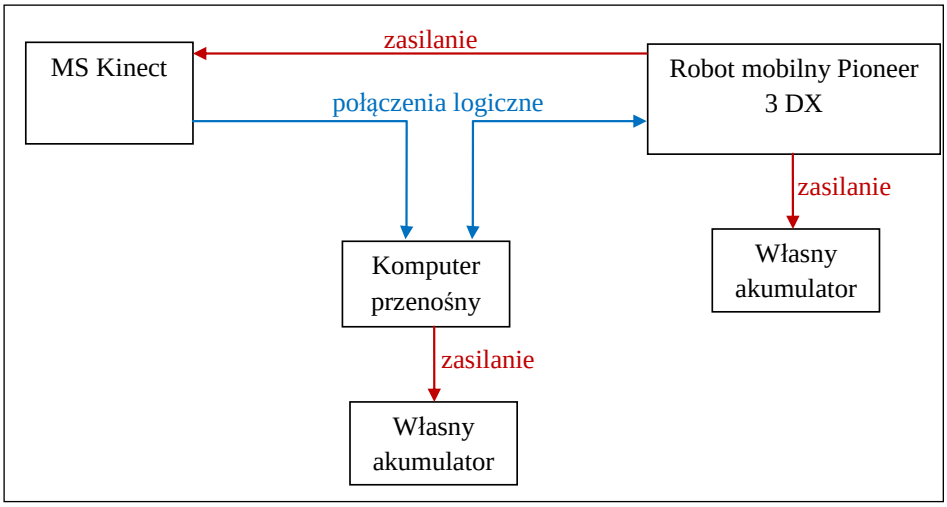
W celu uruchomienia nawigacji należy wywołać kolejno pliki: ***my_robot_configuration.launch*** oraz ***move_base.launch.*** Następnym krokiem jest uruchomienie wizualizatora ***Rviz.*** Program ten umożliwia wyznaczenie punktu docelowego ruchu robota oraz udostępnia wizualizację postrzeganego przez system środowiska zewnętrznego. Szczegółowe możliwości programu oraz sposób jego konfiguracji przedstawiono w punkcie ***2.1.3*** niniejszej pracy.

4.3 *Komunikacja między urządzeniami*

Do projektu wykorzystano trzy urządzenia, którym należało zapewnić zasilanie oraz odpowiednie połączenia logiczne. W tym rozdziale opisano rodzaje połączeń, ich schemat oraz przedstawiono praktyczne rozwiązania zastosowanej komunikacji.

4.3.1 Schemat połączeń

Schemat blokowy wykonanych połączeń oraz uzasadnienie zastosowanej komunikacji zamieszczono poniżej.



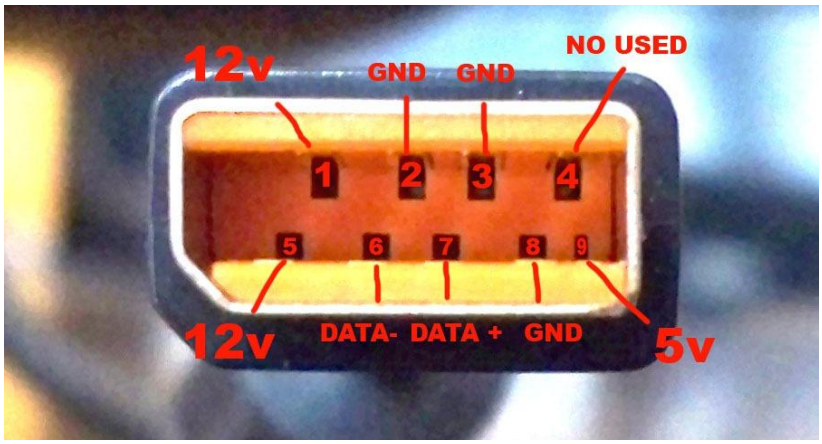
Rys. 15 *Schemat połączeń elektrycznych i logicznych*

Robot Pioneer P3-DX jest platformą mobilną, dlatego zasilanie do wszystkich urządzeń musi być doprowadzone z przenośnego źródła. Chcąc ograniczyć koszty projektu postanowiono, że czujnik będzie pobierał zasilanie z akumulatora wbudowanego w robota oraz wykorzystano komputer z własnym zasilaniem akumulatorowym.

Tworząc schemat połączeń logicznych kierowano się zapotrzebowaniem urządzeń na odpowiednie dane. Komputer, na którym uruchamiany jest system, potrzebuje danych zebranych przez czujnik. Dlatego też zastosowano komunikację jednokierunkową. Połączenie logiczne między robotem a komputerem musi przebiegać w obie strony, ponieważ platforma mobilna wysyła do komputera dane na przykład na temat aktualnej prędkości obrotowej kół, zaś komputer wysyła do robota informację z żądaną prędkością obrotową.

4.3.2 *Podłączenie i mocowanie czujnika*

W związku z tym, że Kinect został skonstruowany głównie do pracy z nowym typem konsoli, posiada on niestandardowe złącze logiczno-zasilające, które zaprezentowano, wraz z opisem, poniżej:



Rys. 16 *złącze logiczno-zasilające czujnika Kinect,*
źródło: http://suhastech.com/homemade-kinect-hack-usb-ac-power-adapter-connector-for-the-xbox-360/

Najlepszym sposobem podłączenia czujnika okazał się zakup kabla rozdzielającego zasilanie od danych. Jest on ogólnie dostępny i niedrogi, więc jego zakup nie kłócił się z poczynionymi założeniami projektowymi.

Poniżej zaprezentowano użyty kabel rozdzielający:

Rys. 17 *Kabel do czujnika rozdzielający dane od zasilania,*
źródło: http://www.fasttech.com/product/1240804-power-supply-ac-usb-adapter-cable-for-xbox-360-kin

Po zastosowaniu kabla rozdzielającego dane czujnika są przesyłane przez port USB 2.0, natomiast kabel zasilający zakończony jest wtyczką do gniazdka ściennego. Chcąc zachować pełną mobilność platformy należało zasilić Kinecta ze źródła znajdującego się na robocie.

Przemyślano różne rozwiązania tego problemu, między innymi: zakup przetwornicy 12V/230V, zakup dodatkowego źródła z gniazdem ściennym oraz przerobienie kabla rozdzielającego. Przetwornica 12V/230V nie jest drogim urządzeniem i mieściłaby się w założeniach projektowych. Biorąc jednak pod uwagę straty energii wiążące się z tym rozwiązaniem (mała sprawność tanich przetwornic), postanowiono znaleźć inny sposób na zasilania czujnika. Ucięto więc wtyczkę gniazda ściennego kabla rozdzielającego i przez stabilizator napięcia 7812 – 12V/1A podłączono przewody do akumulatora. Zastosowanie dodatkowego zabezpieczenia w postaci stabilizatora jest niezbędne, ponieważ podczas ruchu robota mogą wystąpić znaczne spadki napięcia na zaciskach źródła zasilania robota.

Jako że projekt opiera się między innymi na wyznaczaniu współrzędnych przeszkód na trasie ruchu, ważnym aspektem pracy było odpowiednie mocowania czujnika. W trakcie eksperymentalnej weryfikacji pracy robota, opisanej w punkcie ***4.4.4*** niniejszej pracy, zweryfikowano różne możliwości położenia czujnika na platformie mobilnej.

Robot Pioneer P3-DX standardowo komunikuje się za pomocą złącza RS-232. Zastosowany w projekcie komputer przenośny nie posiadał portu RS-232, należało zastosować adapter złącza USB do portu szeregowego RS-232.

4.4 *Eksperymentalna weryfikacja działania robota*

Przedstawiony w niniejszej pracy system sterowania robotami mobilnymi jest rozwijany na licencji Open Source, BSD, dlatego też podczas implementacji i konfiguracji, zarówno pakietów, jak i narzędzi systemu napotkano liczne trudności i błędy. W tym rozdziale zostały opisane rozwiązania zastosowane w projekcie na bazie doświadczeń po eksperymentalnej weryfikacji działania robota oraz spostrzeżenia na temat najczęściej popełnianych błędów na etapie realizacji projektu.

4.4.1 Podłączenie robota i podstawowe sterowanie

Po implementacji niezbędnych bibliotek i zbudowaniu projektu przystąpiono do części praktycznej pracy, z wykorzystaniem schematu połączeń zaprezentowanego na ***rys. 6***. Uruchomiono pakiet ***ROSARIA*** w celu ustanowienia połączenia z robotem Pioneer P3-DX. Test połączenia przebiegł poprawnie czego potwierdzeniem był sygnał dźwiękowy.

Następnie przeprowadzono testu ruchu robota. Dokonano tego za pomocą pakietu ***turtlebot_teleop***, który między innymi pozwolił na wydawanie poleceń zmian prędkości za pomocą klawiatury komputera. Po uruchomieniu programu nie zaobserwowano reakcji robota na wydawane polecenia. Przystąpiono do analizy i wyznaczenia rozwiązania problemu.

Błędem okazało się przeoczenie niezgodności pakietu ***turtlebot_teleop*** ze sterownikiem robota – pakietem ***ROSARIA***. Przeanalizowano ponownie dokumentację obu elementów. Program wydający polecenia zmian prędkości publikował wiadomości w temacie ***turtlebot_telop_keyboard/cmd_vel***, natomiast sterownik subskrybował wiadomości z tematu ***/cmd_vel***. W celu naprawienia błędu poddano edycji plik pakietu ***turtlebot_teleop***, do którego prowadzi ścieżka: turtlebot_teleop/launch***/keyboard_teleop.launch***. Szóstą linię kodu:

```
<remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
```

Zastąpiono następującą:

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

Dzięki tej edycji uzyskano przeniesienie wiadomości publikowanych w temacie ***turtlebot_teleop_keyboard/cmd_vel*** do tematu ***cmd_vel***.

Kolejne testy nie przyniosły oczekiwanych rezultatów. Po konsultacjach z promotorem stwierdzono, że błędna może być wartość parametru ***TicksMM***. Jest to parametr odpowiedzialny za przełożenie obrotów kół na odległość. Podczas uruchomienia programu jego wartość ustalana była na ***0***. Działo się tak z powodu błędu kodu programu. Wspomnianemu parametrowi była przypisywana wartość innego parametru – ***RevCount***. Aby pozbyć się tego błędu należało w pliku ***RosAria.cpp*** w linii 125 zamienić

```
n_.setParam( "TicksMM", RevCount);
```

na:

```
n_.setParam( "TicksMM", TicksMM);
```

W końcowym etapie realizacji pracy opublikowana została nowa wersja pakietu, która nie zawiera wyżej omówionego błędu.

Po zastosowaniu powyższych zmian w kodach plików i ponownemu uruchomieniu programów ***RosAria*** i ***keyboard_teleop.launch*** z pakietu ***turtlebot_teleop***, robot zaczął poruszać się zgodnie z oczekiwaniem.

4.4.2 Test dwóch trybów pracy nawigacji – PointCloud oraz LaserScan

Pakiet ***Navigation stack*** jest konfigurowany wieloma istotnymi parametrami. Zdecydowana większość z nich została ustawiona na bazie wartości podstawowych, a następnie była eksperymentalnie regulowana. Dla jednego z najważniejszych parametrów, jakim niewątpliwie jest typ strumienia danych wejściowych, przeprowadzono odrębny test.

Zastosowana w projekcie nawigacja ma możliwość pracować na danych pobranych z czujnika laserowego lub z czujnika publikującego chmury punktów ***(PointCloud)***.

Zdecydowano, że jako pierwsza testowi poddana będzie nawigacja w trybie analizy danych typu ***PointCloud***, ponieważ jest to standardowy typ danych użytego czujnika. Aby uruchomić test wprowadzono odpowiednie zmiany do pliku ***costmap_common_params.yaml***. Zamiast parametru ***laser_scan_sensor*** użyto ***point_cloud_sensor*** i skonfigurowano go w następujący sposób:

```
point_cloud_sensor: {sensor_frame: camera_link, data_type: PointCloud, topic: /camera/depth/points, marking: true, clearing: true}
```

Po uruchomieniu nawigacji oraz przygotowaniu wizualizacji, nastąpił znaczny wzrost procentowego użycia CPU. Komputer został tak bardzo obciążony, że podstawowa nawigacja kursorem była mocno ograniczona. Postanowiono usunąć z wizualizacji dane publikowane, przez sterownik czujnika (***PointCloud***), co pomogło zredukować obciążenie procesora. W kolejnym kroku wyznaczono punkt końcowy ruchu. Robot stał w punkcie startu przez około 5 sekund, a następnie ruszył w kierunku wyznaczonego celu. Ruch był po chwili przerwany. Następnie robot się obrócił dookoła i ruszył w przód uderzając w przeszkodę. Powtórzono próbę, jednak efekt był bardzo zbliżony do poprzedniego.

Następnie poddano testowi drugi zaproponowany tryb nawigacji. Dokonano zmian w pliku ***costmap_common_params.yaml***, tak aby nawigacja analizowała dane typu ***LaserScan***. Poniżej rozpisano dokładną konfigurację parametru ***laser_scan_*sensor:

```
laser_scan_sensor: {sensor_frame: camera_link, data_type: LaserScan, topic: /scan, marking: true, clearing: true }
```

Po uruchomieniu nawigacji oraz przygotowaniu wizualizacji w programie ***Rviz***, szybkość pracy komputera nie uległa zmianie. Użycie tej metody zniwelowało obciążenie procesora, co przełożyło się na dobrą responsywność aplikacji sterującej. Platforma ruszyła bez zauważalnego opóźnienia i osiągnęła cel bez zbędnych postojów i obrotów. Przy kolejnych próbach osiągnięto podobne efekty.

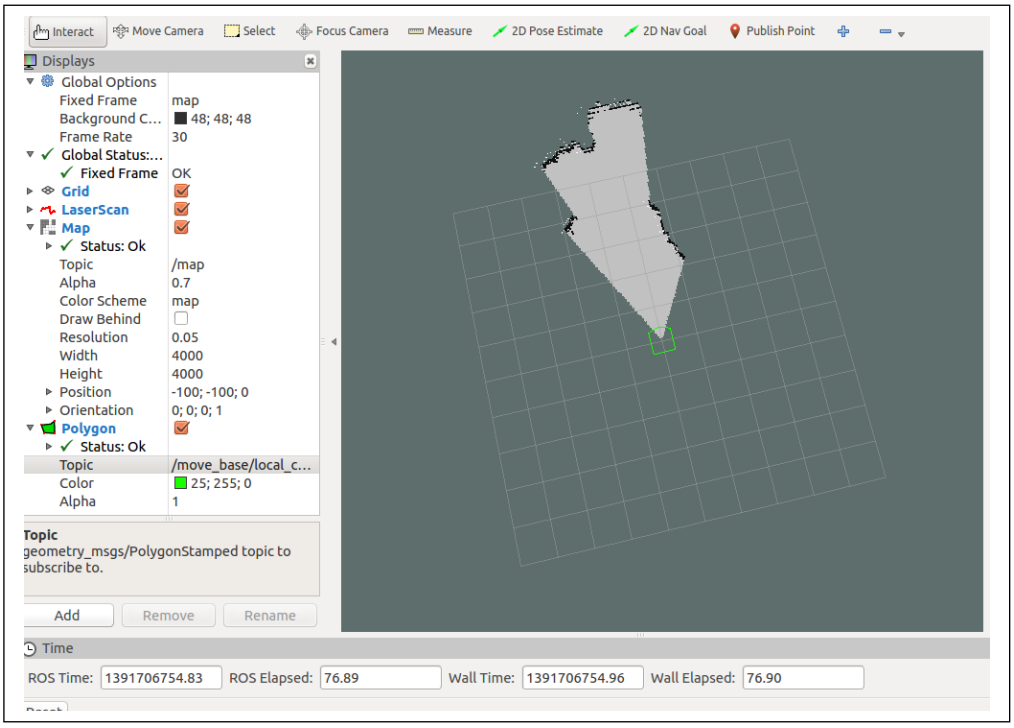
Na podstawie powyższych eksperymentów można wyciągnąć następujące wnioski:

- Wizualizacja danych typu ***PointCloud*** zużywa dużo pamięci używanego procesora, więc powinna być wyłączana.
- Algorytm nawigacji korzystając z chmury punktów potrzebuje zdecydowanie więcej czasu na analizę środowiska i wyznaczenie trajektorii ruchu.
- Zaobserwowany postój i obrót robota może być spowodowany zbyt długim czasem oczekiwania algorytmu kalibrującego położenie robota na mapę środowiska.

- Nawigacja działa zdecydowanie płynniej w trybie pracy z czujnikiem laserowym. Może to być spowodowane ograniczeniami użytego komputera.
- W dalszej realizacji projektu postanowiono korzystać z konwersji strumienia danych typu ***Pointcloud*** na typ ***LaserScan***.

4.4.3 Tworzenie mapy środowiska

Po uruchomieniu programu mapującego, według punktu ***3.7*** niniejszej pracy oraz programu sterowania ręcznego ***telep_keyboard***, należącego do pakietu ***turtlebot_teleop***, uzyskano zaprezentowany poniżej widok w oknie wizualizatora:



Rys. 18 *Wizualizacja mapowania*

Zastosowanie standardowej procedury tworzenia mapy skutkowało długim czasem analizy kolejnych odczytów czujnika i było nieporęczne dla użytkownika. Usprawniono tę czynność poprzez:

- Włączenie nagrywania danych z dwóch tematów:

```
rosbag record -O mylaserdata /scan /tf
```

- Odpowiednie poruszanie robotem:
    - Zmiana parametrów prędkości maksymalnej publikowanej za pomocą pakietu ***turtlebot_teleop***:

```
rosparam set turtlebot_teleop_keyboard/scale_linear 0.06
rosparam set turtlebot_teleop_keyboard/scale_angular 0.2
```

    - Ograniczenie szybkich rotacji
    - Zmiana położenia robota po upływie całej pętli program (pobór danych, dopasowanie pomiaru do poprzedniego odczytu, zapis i publikacja nowej mapy)
- Uruchomienie wizualizacji danych czujnika oraz analiza dostrzeżonych przez robota przeszkód
- Po objechaniu żądanej przestrzeni wyłączenie nagrywania i wszystkich innych terminali
- Przestawienie parametru systemu, tak aby czas w ***ROS*** był identyczny z nagranym:

```
rosparam set use_sim_time true
```

- Uruchomienie ***roscore*** oraz program mapującego:

```
roscore
rosrun gmapping slam_gmapping scan:=scan
```

- Odtworzenie nagranego pliku w zwolnionym 0.4 razy tempie:

```
rosbag play -r 0.4 mylaserdata.bag
```

- Po ukończeniu mapowania zapisano mapę:

```
rosrun map_server map_saver
```

Plik mapy – ***map.pgm*** znajduje się w folderze ***Home*** systemu Linux. Zastosowane spowolnienie odtwarzania nagranego pliku, pozwala algorytmowi dopasowującemu kolejne dane czujnika, przetworzyć punkty z poprzedniej serii zanim uruchomi on kolejną pętlę analizy środowiska. Należy jednak dobrać optymalną wartość tego parametru, gdyż zbyt duże spowolnienie może skutkować, przy mało różnorodnym środowisku, nadmiernym nakładaniem kolejnych odczytów czujnika. Przyczyną może być ograniczony kąt widzenia w płaszczyźnie poziomej czujnika lub niedostateczna kalibracja sterownika robota - pakietu RosAria, który odpowiada za wysyłanie żądań zmian prędkości.

4.4.4 Wyznaczenie optymalnego położenia czujnika

Początkowo umieszczono czujnik na robocie w miejscu wskazanym na poniższym zdjęciu:

Rys 19 *Podstawowe położenie czujnika*

Podczas testów w małym pomieszczeniu zauważono znaczenie ograniczenie czujnika. Kinect nie przesyła informacji o punktach przestrzeni znajdujących się w odległości mniejszej niż 30 cm. Jest to poważne utrudnienie w prowadzonym projekcie, ponieważ przeszkody znajdujące się w niedostępnej przestrzeni nie są uwzględniane w procesie tworzenia mapy statycznej, mapy przeszkód i tworzenia trajektorii ruchu. Robot rozpoczynający ruch z miejsca gdzie w niewielkiej odległości przed nim znajduje się przeszkoda porusza się w przód i uderza w nią. Problem występuje również podczas w trakcie trwania ruchu, ponieważ robot poruszając się aktualizuje mapę i usuwa przeszkody znajdujące się w niedalekiej odległości od niego.

Problem postanowiono rozwiązać ustawiając czujnik w zaprezentowanym poniżej miejscu oraz obracając go, za pomocą wbudowanego silnika, o -25 stopni.

Rys. 20 *Nowe położenie czujnika*

Należało jednak dodatkowo zmienić transformacje układów czujnika względem układu bazowego robota. Dokonano tego edytując plik wykonawczy przygotowanego wcześniej pakietu własnego. Wyznaczono nowe położenie czujnika wg zmiennych typu yaw, pitch, roll oraz zaimplementowano funkcję konwertującą te zmienne do kwaternionów. Poniżej zaznaczono przekształconą część pliku **tf_broadcaster.cpp**.

tf_broadcaster.cpp

```
.
.
.
  while(n.ok()){
    broadcaster.sendTransform(
      tf::StampedTransform(

tf::Transform(tf::createQuaternionFromRPY(0,20*0.01745329251,0),
tf::Vector3(-0.13, 0.0, 0.5)),

// tf::Transform(tf::Quaternion(0,0,0,1), tf::Vector3(0.1, 0.0,
0.3)),
        ros::Time::now(),"base_link", "camera_link"));
    r.sleep();
}
.
.
.
```

Należy pamiętać o ponownym zbudowaniu środowiska pakietów po każdej serii zmian plików wykonawczych środowiska.

Uruchomiono nawigację, przeprowadzono testy i zebrano poniższe wnioski:

- Podczas pracy w trybie czujnika laserowego mapa przestrzeni 2D jest budowana w oparciu o środkową, horyzontalną linię chmury punktów. Ustawiając czujnik w zaproponowanym miejscu nawigacja bierze pod uwagę jedynie przeszkody w odległości 30 cm od robota.
- Dzięki temu zabiegowi poprawiono dolną granicę zasięgu czujnika, ale stracono możliwość poprawnego wyznaczania trajektorii ruchu.
- W trybie czujnika głębi mapa 2D budowana jest w oparciu o pełną chmurę punktów wyznaczonych przez Kinect. Po zastosowaniu transformacji układów współrzędnych otrzymano wystarczająco szeroki zakres działania czujnika. Zaobserwowano jednak problem opóźnienia publikowania strumienia danych, który szczegółowo został opisany w podrozdziale ***4.4.2.***

Po analizie powyższych wniosków postanowiono pozostawić czujnik w pierwotnym położeniu zaprezentowanym na początku tego punktu na zdjęciu ***nr 21***.

5. Podsumowanie i wnioski

Projekt okazał się bardzo pracochłonny i rozwijający. Głównym problemem było zrozumienie zasad działania systemu **Robot Operating System**. Jest to bardzo złożone narzędzie do programowania i sterowania robotami. Jego podstawa została zbudowana na licencji Open Source, dzięki czemu uzyskano prężnie rozwijające się środowisko programowania oraz zjednoczono społeczność naukową wokół problemu sterowania robotami.

Wszystkie postawione cele pracy zostały zrealizowane. Szczegółowo zapoznano się z budową i działaniem systemu ROS, co pozwoliło na poprawne zbudowanie projektu i wyznaczenie koncepcji dalszego jego rozwoju. System poznawano wyłącznie na podstawie dokumentacji spisanej w języku angielskim, dzięki czemu dyplomant ma otwartą drogę do dyskusji na tematy sterowania robotami w środowisku akademickim i naukowym.

W trakcje realizacji niniejszej pracy zapoznano się także z działaniem czujnika Kinect, który okazał się bardzo dobrym narzędziem do nawigacji robotem autonomicznym. Charakteryzuje się on wystarczającą częstotliwością pracy i zasięgiem pomiarowym, lecz niestety obarczony jest też pewnymi ograniczeniami. Jednym z nich jest brak dookólnej obserwacji przestrzeni, co znacznie utrudnia tworzenie mapy przestrzeni. Kolejnym zaś jest stosunkowo duża minimalna odległość detekcji przeszkód.

W ramach niniejszej pracy dyplomowej zweryfikowano możliwości połączenia ROS z czujnikiem Kinect. Okazało się, że system zawiera już gotowe biblioteki i sterowniki dedykowane do tego typu czujników. Są one wydajne i zapewniają poprawną komunikację z urządzeniem, jednak ich prawidłowa konfiguracja jest czasochłonna. Wymaga ona szczegółowej wiedzy na temat konfiguracji pakietów ROS oraz komunikacji między elementami tego systemu.

Wykonany projekt spełnia wszystkie założenia postawione w fazie koncepcyjnej jego realizacji. Udało się zbudować aplikację realizującą zarówno manualne sterowanie ruchem robota, jak i autonomiczne omijanie przeszkód. Oprogramowanie poprawnie wyznacza trajektorię, wzdłuż której później realizowany jest ruch robota. Platforma mobilna potrafi też omijać przeszkody zarówno statyczne (nawigacja w pomieszczeniu), jak i dynamiczne (spontanicznie pojawiające się obiekty). Podczas testów sprawdzono przypadek przejścia człowieka przez wyznaczoną przez robota trajektorię i otrzymano pozytywny wynik - robot osiągnął założony cel z wystarczającą dokładnością.

Dodatkową funkcjonalność, jaką zrealizowano w ramach pracy dyplomowej była budowa mapy 2D otoczenia robota. Pozytywnie przeprowadzono konfigurację

oprogramowania i przygotowano system, jednak podczas testów okazało się, że wspomniane wyżej ograniczenia zastosowanego czujnika w znaczący sposób utrudniają budowę mapy.

Podczas realizacji projektu zmniejszono też zużycie zasobów komputera poprzez przekształcenie obrazu typu PointCloud na LaserScan. W ten sposób zwiększono efektywność sterowania robotem i osiągnięto płynną pracę wizualizatora. Ograniczając zużycie procesora przygotowano projekt do dalszego rozwoju.

Reasumując, projekt zrealizowany w ramach niniejszej pracy dyplomowej okazał się być bardzo ciekawy i jednocześnie wymagający. Wiedza i umiejętności dotyczące dynamicznie rozwijającej się dziedziny wiedzy, jaką jest autonomiczna nawigacja robotów mobilnych, z pewnością będą pomocne w dalszym rozwoju naukowym i zawodowym autora pracy. Ponadto, tematyka projektu jest na tyle aktualna, rozwijająca i multidyscyplinarna, że może stać się przedmiotem ćwiczenia laboratoryjnego dla studentów, którzy będą mieli możliwość wykorzystania swej wiedzy i umiejętności do zaprogramowania jednej z podstawowych funkcji robotów mobilnych, jaką jest lokalizacja i nawigacja.

Bibliografia

[1] Dokumentacja systemu ROS [online]. [Dostęp 1 lutego 2014 r.]: http://wiki.ros.org

[2] John MacCormick: **How does the Kinect work?** [online]. [Dostęp 1 lutego 2014 r.]: http://pages.cs.wisc.edu/~ahmad/kinect.pdf

[3]Dieter Fox: **KLD-Sampling: Adaptive Particle Filters**. Department of Computer Science & Engineering University of Washington [online]. [Dostęp 1 lutego 2014 r.]: http://books.nips.cc/papers/files/nips14/AA62.pdf

[4]Pioneer P3-DX [online]. [Dostęp 1 lutego 2014 r.]: http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx#

Załącznik nr 1

- ***CMakeLists.txt***:

```
cmake_minimum_required(VERSION 2.8.3)
project(nowy_robot)

## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  ROSARIA
  move_base
  openni_launch
  robot3_tf
  turtlebot_bringup
)
catkin_package(
CATKIN_DEPENDS ROSARIA move_base openni_launch robot3_tf turtlebot_bringup
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

- ***package.xml***:

```
<?xml version="1.0"?>
<package>
  <name>nowy_robot</name>
  <version>0.0.0</version>
  <description>The nowy_robot package</description>
  <maintainer email="wojciech@todo.todo">wojciech</maintainer>
  <license>TODO</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>ROSARIA</build_depend>
  <build_depend>move_base</build_depend>
  <build_depend>openni_launch</build_depend>
  <build_depend>robot3_tf</build_depend>
  <build_depend>turtlebot_bringup</build_depend>
  <run_depend>ROSARIA</run_depend>
  <run_depend>move_base</run_depend>
  <run_depend>openni_launch</run_depend>
  <run_depend>robot3_tf</run_depend>
  <run_depend>turtlebot_bringup</run_depend>
  <export>
  </export>
  </package>
```

- ***my_robot_configuration.launch*** – załącza węzły zewnętrzne współpracujące z nawigacją

```
<launch>
<include file="$(find openni_launch)/launch/openni.launch" />
<include file="$(find turtlebot_bringup)/launch/3dsensor.launch" />
<node pkg="robot3_tf" type="tf_broadcaster3" name="broadcaster3" output="screen">
  </node>
<node pkg="ROSARIA" type="RosAria" name="rosaria" output="screen">
  </node>
</launch>
```

- ***costmap_common_params.yaml*** – plik konfiguracyjny dla strumieni danych sensorów

```
obstacle_range: 6.0
raytrace_range: 2.0
inflation_radius: 2.0
footprint: [[0.3,-0.3], [-0.3, -0.3], [-0.3, 0.3], [0.3,0.3],[0.4,0]]

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: camera_link, data_type: LaserScan, topic: /scan, marking:
true, clearing: true}
```

- ***global_costmap_params.yaml*** – plik konfiguracjny parametrów global_costmap

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
```

- ***local_costmap_params.yaml*** – plik konfiguracyjny parametrów local_costmap

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.1
map_type: costmap
```

- ***base_local_planner_params.yaml*** – plik konfiguracyjny przetwarzania żądań zmian prędkości

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_rotational_vel: 1.0
  min_in_place_rotational_vel: 0.4

  acc_lim_th: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: false
```

- ***move_base.launch*** – plik uruchamiający utworzone pliki konfiguracyjne, algorytm nawigacji oraz serwer mapy

```
<launch>
  <master auto="start"/>

  <!-- Run the map server  -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
mapy)/map.yaml"/>
  <!--- Run AMCL    -->
  <include file="$(find amcl)/examples/amcl_omni.launch" />

  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">

    <rosparam file="$(find nowy_robot)/costmap_common_params.yaml" command="load"
ns="global_costmap" />

    <rosparam file="$(find nowy_robot)/costmap_common_params.yaml" command="load"
ns="local_costmap" />

    <rosparam file="$(find nowy_robot)/local_costmap_params.yaml" command="load" />

    <rosparam file="$(find nowy_robot)/global_costmap_params.yaml" command="load" />

    <rosparam file="$(find nowy_robot)/base_local_planner_params.yaml" command="load"
/>
  </node>
</launch>
```

- ***3dsensor.launch* –** konwersja wiadomości typu PointCloud do LaserScan

```
<launch>

  <arg name="camera"      default="camera"/>

 <arg name="publish_tf"  default="false"/>

 <!-- Factory-calibrated depth registration -->

 <arg name="depth_registration"            default="true"/>

 <!-- Processing Modules -->

 <arg name="rgb_processing"                default="true"/>

 <arg name="ir_processing"                 default="true"/>

 <arg name="depth_processing"              default="true"/>

 <arg name="depth_registered_processing"     default="true"/>

 <arg name="disparity_processing"          default="true"/>

 <arg name="disparity_registered_processing" default="true"/>

 <arg name="scan_processing"               default="true"/>

 <!-- Worker threads for the nodelet manager -->

 <arg name="num_worker_threads" default="4" />

 <!-- Laserscan topic -->

 <arg name="scan_topic" default="scan"/>

<!—Laserscan This uses lazy subscribing, so will not activate until scan is requested.

  -->

 <group if="$(arg scan_processing)">

   <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan" args="load
depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg camera)/$(arg
camera)_nodelet_manager">

     <!-- Pixel rows to use to generate the laserscan. For each column, the scan will
```

```
        return the minimum value for those pixels centered vertically in the image. -->
    <param name="scan_height" value="1"/>
    <param name="output_frame_id" value="/$(arg camera)_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="$(arg camera)/depth/image_raw"/>
    <remap from="scan" to="$(arg scan_topic)"/>
    <!-- Somehow topics here get prefixed by "$(arg camera)" when not inside an app
namespace, so in this case "$(arg scan_topic)" must provide an absolute topic name (issue
#88).  Probably is a bug in the nodelet manager: https://github.com/ros/nodelet_core/issues/7 -
->
    <remap from="$(arg camera)/image" to="$(arg camera)/depth/image_raw"/>
    <remap from="$(arg camera)/scan" to="$(arg scan_topic)"/>
   </node>
  </group>
</launch>
```