Robotyka Mobilna
Laboratorium

Pioneer – ćwiczenie I

Opracowanie:
Łukasz Chechliński

Uaktualnienie:
Daniel Koguciuk

Źródło:
www.ros.org

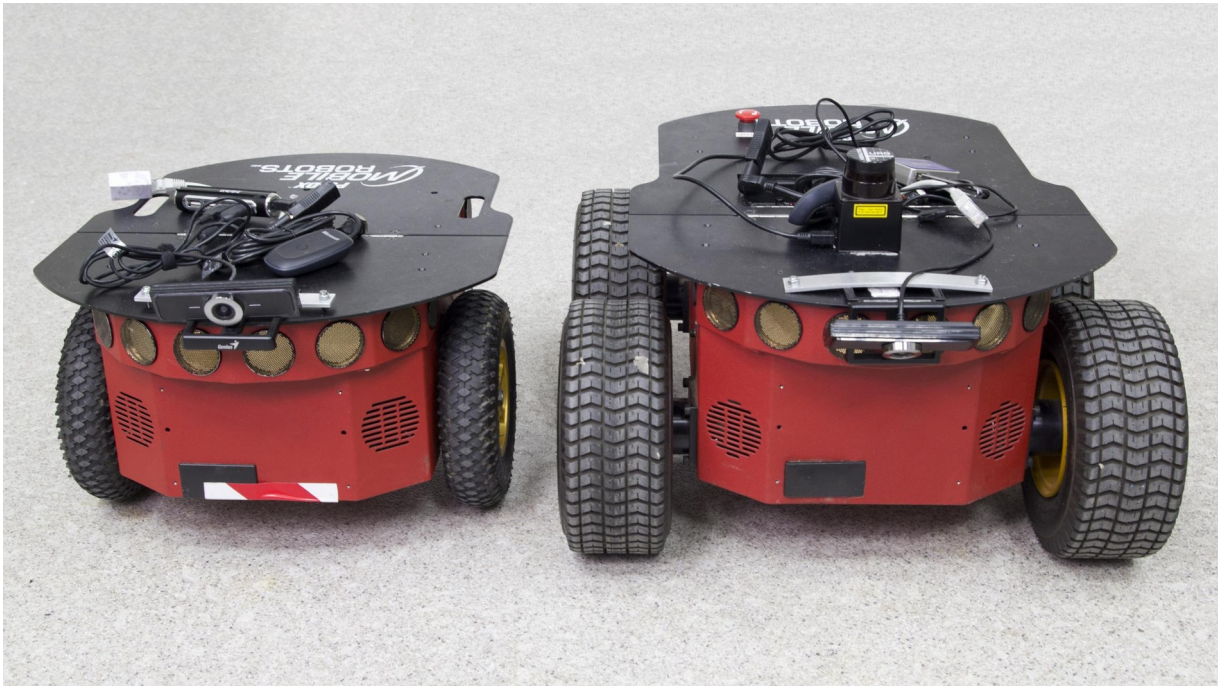
Wstęp

Celem niniejszego ćwiczenia jest napisanie pierwszego prostego programu sterującego w sposób autonomiczny robotem typu Pioneer. Zadaniem robota jest przejechanie po trasie wyznaczonej przez znajdującą się na podłodze taśmę, wykrywaną przez robota za pomocą kamery USB. Działanie robota ma charakter behawioralny – jego ruch wynika wprost z danych sensorycznych, robot nie gromadzi danych o świecie ani nie planuje swoich działań.

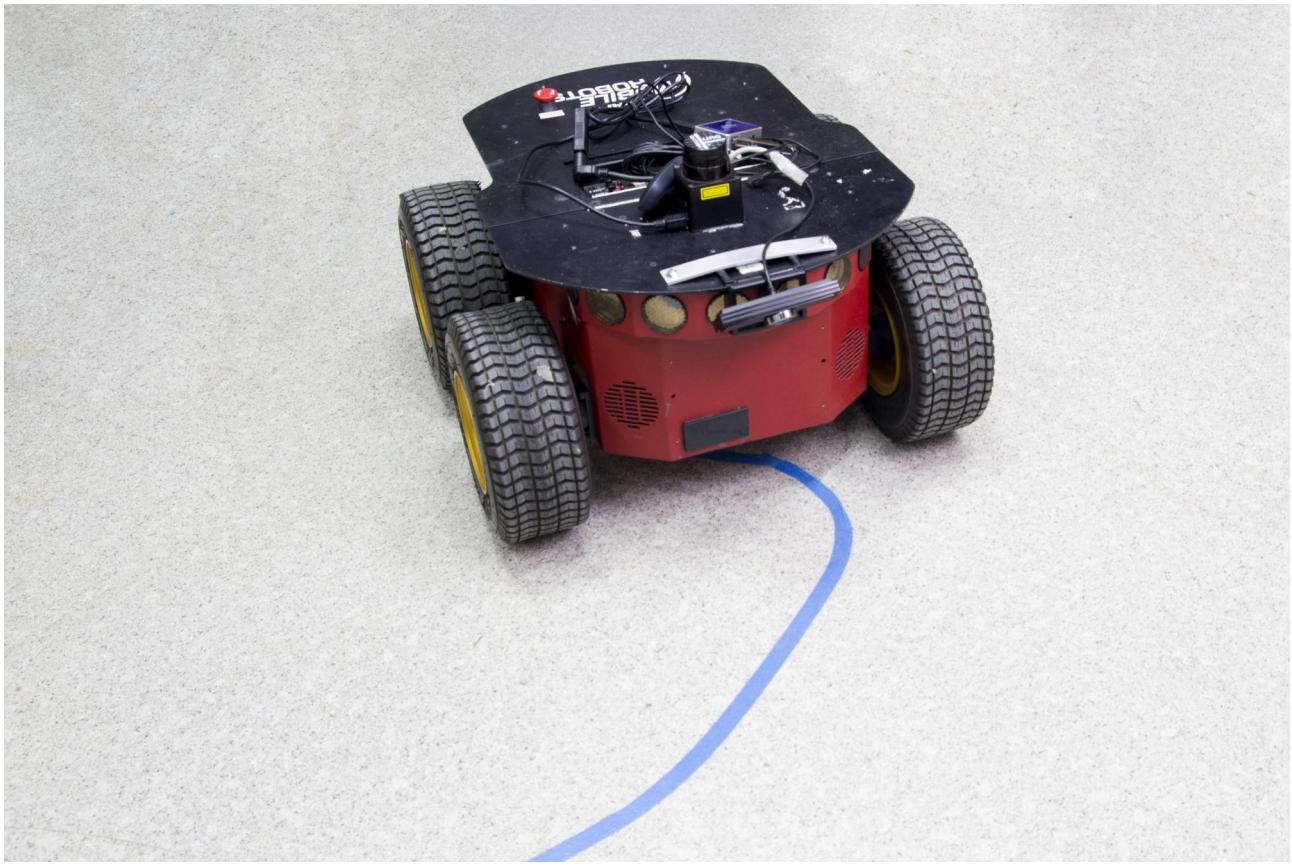
Poniższa instrukcja zorganizowana jest w taki sposób, by przeprowadzić partycypanta przez cały proces tworzenia oprogramowania dla robota mobilnego na potrzeby laboratorium z przedmiotu robotyka mobilna. Jeżeli zatem student ma jakieś pytania, to sugerowana kolejność poszukiwania odpowiedzi na takowe jest następująca:
1) Przeczytać instrukcję.
2) Przeczytać instrukcję.
3) Zapytać prowadzącego.
Dokument jest napisany możliwie rozwlekle komentując każdy etap pracy z systemem linux oraz z robotem, tak, by uzdolniony student IAiR'u był w stanie z sukcesem przeprowadzić laboratorium. Dla bardziej zaawansowanych użytkowników linuxa lub osób doświadczonych z pracą z robotem, czy ROS'em przewidziane są wyszarzone elementy instrukcji, szerzej tłumaczące dane zagadnienie.

Konfiguracja sprzętowa

W ćwiczeniu tym wykorzystywany jest robot typu Pioneer. Jest on wyposażony w sonary, czyli sensory ultradźwiękowe mierzące odległości do przeszkód w różnych kierunkach oraz różnicowy układ napędowy składający się, w zależności od modelu, z dwóch kół napędzanych oraz trzeciego podporowego lub czterech kół napędowych. Roboty te zostały zmodernizowane i obecnie mają na swoim pokładzie komputer jednoukładowy NVIDIA Jetson TK1. Umożliwia on podłączenie do robota kamery USB oraz analizę w czasie rzeczywistym odbieranego obrazu. Na zdjęciu poniżej, po lewej znajduje się robot **Robin**, zaś po prawej **na na na na na Batman**.



Komputer pokładowy jest w stanie realizować dosyć złożone obliczeniowo zadania, jednak nie jest on podłączony do monitora i klawiatury (co zresztą nie jest zbyt wygodne przy pracy z robotem mobilnym). Dlatego też jako terminal służący do dostępu do niego wykorzystywany jest komputer stacjonarny podłączony do tej samej sieci wi-fi. Poniższy rysunek przedstawia **Batmana** znajdującego się na torze ćwiczeniowym.



Uruchomienie robota

Ćwiczenie to wykonywane jest na dwóch pokazanych wyżej robotach. W zależności od wybranego superbohatera (i przyporządkowanego mu komputera-terminalu) dane do logowania są następujące:

Użytkownik: mr_labs
Hasło: heurystyka

Włącznik robota znajduje się z tyłu, na jego lewej stronie obok gniazda zasilania. Po jego włączeniu powinien nastąpić automatyczny start komputera pokładowego Jetson TK1, co zajmuje kilkanaście sekund. Aby sprawdzić, czy system jest online można wywołać komendę ping oraz nazwę robota, przykładowo dla Batmana komenda ping będzie wyglądać następująco:

```
ping batman
```

Nazwa batman jest znana komputerowi PC, ponieważ została wpisana do pliku /etc/hosts, który może wyglądać następująco:

```
127.0.0.1       localhost
127.0.0.1       tales
192.168.0.130   batman
192.168.0.131   robin
```

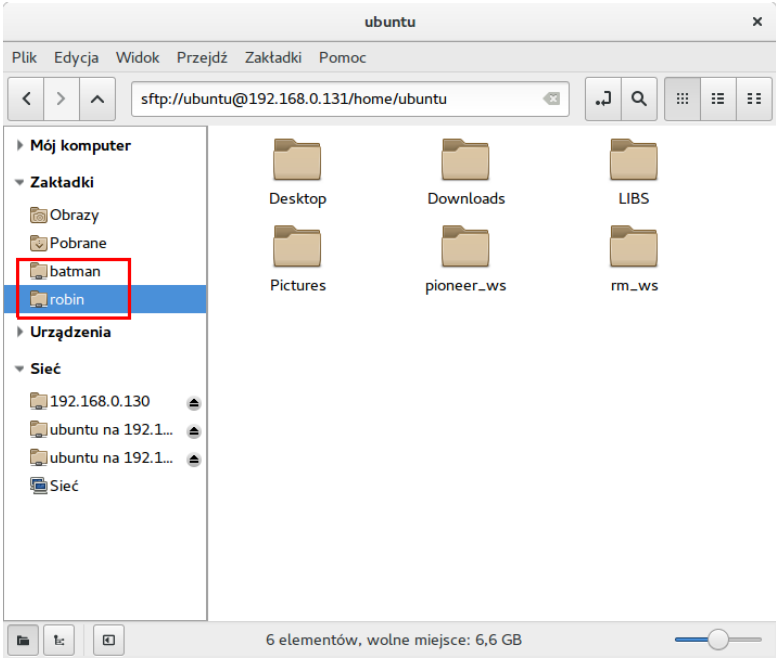
Jeśli jakość połączenia jest słaba (występują duże opóźnienia między pakietami lub da się zauważyć znaczny odsetek pakietów utraconych), to warto w czasie pisania oprogramowania podłączyć komputer pokładowy robota do routera za pomocą kabla ethernetowego. Zapewni to wyższy komfort pracy podczas pisania kodu, zaś po rozpoczęciu testów możemy robota odłączyć – jakość połączenia powinna być wystarczająca do obserwacji jego pracy. Po sprawdzeniu połączenia możemy zalogować się zdalnie na komputer pokładowy. Wpisujemy w tym celu komendę (w zależności od nazwy robota):

```
batman_ssh
```

Jest to alias rozpoczęcia sesji SSH z opcją wykorzystania powłoki graficznej, czyli powyższa komenda jest równoważna takiemu wpisowi w ~/.bashrc:

```
alias pioneer='ssh -X ubuntu@batman'
```

Powyżej opisano procedurę łączenia z robotem poprzez terminal, pracę jednak rozpoczniemy od uruchomienia przeglądarki plików robota z poziomu komputera PC. W tym celu należy wcisnąć klawisz home, czyli „okienko” na klawiaturze oraz wybrać przeglądarkę plików **nemo**. Po lewej stronie będzie dodana zakładka do połączenia się przeglądarką z systemem plików na robocie:



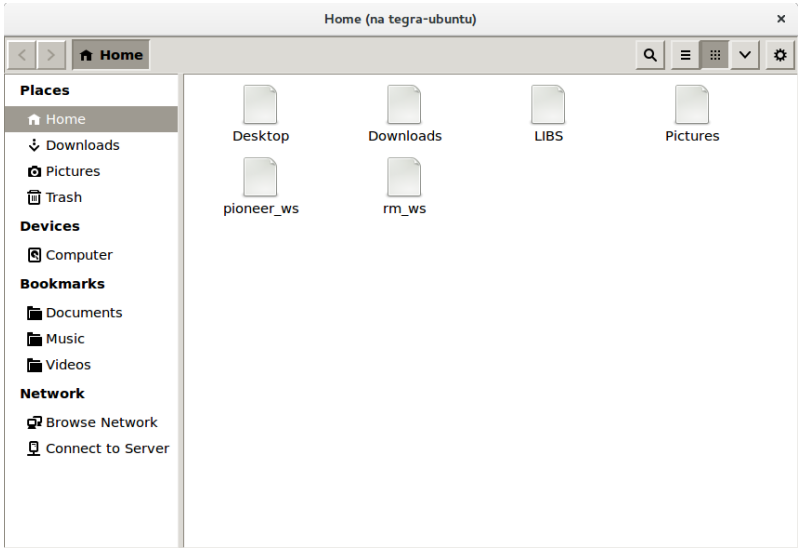
Aby przygotować sobie wygodną zakładkę, jak pokazano wyżej można w adresie przeglądarki plików (wywołuję się ją poprzez skrót CTRL+L) wpisać poniższy adres, a następnie dodać lokalizację do zakładek (oczywiście jest to już zrobione):

```
sftp://ubuntu@batman/home/ubuntu/
```

Na obu robotach wykorzystujemy domyślne konto użytkownika:

Użytkownik: ubuntu
Hasło: ubuntu

O podanie tego hasła zostaniemy poproszeni zarówno podczas pierwszego połączenia z wykorzystaniem nemo jak i podczas otwierania sesji SSH. Po wskazaniu w przeglądarce nemo zakładki z robotem dostaniemy okno podobne do tego poniżej:



W katalogu tym znajdują się dwa katalogi robocze ROSa – *pioneer_ws* oraz *rm_ws*. Katalogiem wykorzystywanym podczas laboratorium Robotyki Mobilnej będzie oczywiście *rm_ws*. Korzystanie z programu *nautilus* umożliwia pracę z katalogami wygodniejszą niż poprzez polecenia *cd* i *ls*, co nie znaczy jednak że może on zastąpić je we wszystkich przypadkach.

Przechodząc do zasadniczej części pracy należy uruchomić na komputerze pokładowym rdzeń ROSa. W tym celu należy utworzyć nową sesje SSH i wpisać w niej polecenie:

```
roscore
```

Następnie należy uruchomić sterowniki robota. W tym celu otwieramy kolejną sesję SSH i wpisujemy w niej komendę:

```
pioneer_base_launch
```

Jest to alias do pełnej komendy z parametrami (paczka pioneer_driver nie jest standardową paczką ROS, tylko zestawem programów przygotowanych przez nas w naszym laboratorium):

```
alias pioneer_base_launch='roslaunch pioneer_driver pioneer_driver.launch
robot_name:=robin aria:=true urdf:=false imu:=false scanner_ip:=false scanner_usb:=false
joy:=true joy_control:=true --screen'
```

W tym momencie robotem możemy już sterować ręcznie za pomocą pada. W tym celu należy go włączyć, a następnie poruszać gałką sterującą (usunęliśmy funkcjonalność przycisku zezwolenia).

Kolejnym krokiem jest zapoznanie się z kanałami komunikacji, jakie oferują nam sterowniki robota. Podglądu istniejących topic'ów nie musimy jednak dokonywać w sesji SSH, wystarczy zwykłe okno terminalu. Jest tak, ponieważ ROS jest systemem który może działać w sieci kilku komputerów, w architekturze monomaster. W tym ćwiczeniu *master* znajduje się na komputerze pokładowym robota (tam uruchomiliśmy komendę *roscore*), natomiast komputer stacjonarny skonfigurowany jest jako *slave*. Wymaga to odpowiedniego ustawiania informacji o adresach IP, co nie leży w zakresie niniejszego ćwiczenia. Aby podejrzeć listę topiców na robocie należy wpisać komendę (jak już dobrze wiadomo z ogólnej instrukcji o ROS):

```
rostopic list
```

Zobaczymy wtedy odpowiedź zbliżoną do poniższej:

```
/batman/aria/battery_recharge_state
/batman/aria/battery_state_of_charge
/batman/aria/battery_voltage
/batman/aria/bumper_state
/batman/aria/cmd_vel
/batman/aria/motors_state
/batman/aria/parameter_descriptions
/batman/aria/parameter_updates
/batman/aria/pose
/batman/aria/sonar
/batman/aria/sonar_pointcloud2
/batman/joy
/diagnostics
/rosout
/rosout_agg
/tf
```

Jak łatwo się domyślić, dla robota Robin nazwy topiców będą odpowiednio inne. Aby zobaczyć odczyty z sonarów, wpisujemy komendę:

```
rostopic echo /batman/aria/sonar
```

Usłyszymy dźwięk pracujących sonarów. Analogicznie możemy zobaczyć publikowane komendy

prędkości:

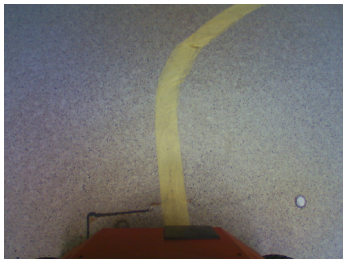
```
rostopic echo /batman/aria/cmd_vel
```

Możemy wtedy zobaczyć, że komendy prędkości publikowane są tylko wtedy, gdy pad jest wykorzystywany do sterowania. Pozwala nam to na publikowanie komend z innego źródła, które nie będą zakłócane przez zerowe komendy prędkości wysyłane przez nieużywanego aktualnie pada. Oglądanie odczytów sonarów w postaci listy cyfr jest jednak niewygodne. Aby zobaczyć ich wizualizację, uruchamiamy program *rViz*:

```
rosrun rviz rviz
```

Algorytm detekcji linii

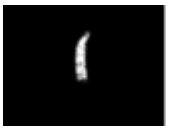
W tym ćwiczeniu napiszemy dwa programy. Jeden z nich będzie odpowiadał za detekcję linii na obrazie z kamery. Informację wyjściową będzie przesyłał on w postaci listy wykrytych punktów do programu sterującego. Wejściem dla pierwszego programu jest obraz z kamery:



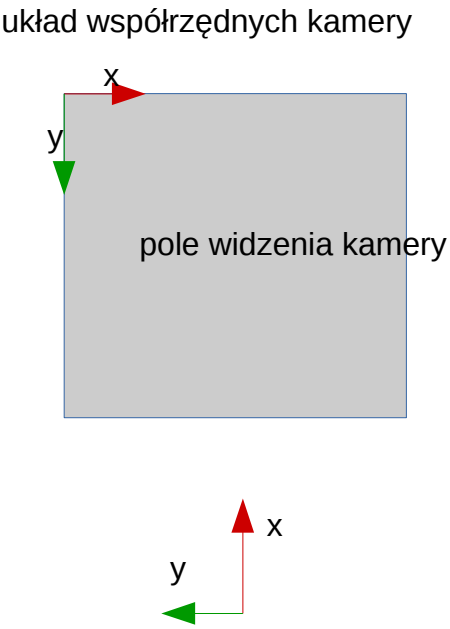
Widać na nim linię, za którą podążać ma robot. Podobne kolory występują również w tle, co utrudnia zadanie. Aby lepiej odróżniać te kolory, należy przekonwertować obraz z przestrzeni barw BGR do HSV. Następnie należy określić, jaki kolor ma śledzona linia. Najszybszym sposobem na ustalenie tego jest zrobienie zrzutu ekranu i otworzenie go w programie graficznym, np. ***KolourPaint***. W programie używamy na wybranym fragmencie linii narzędzia pobierania koloru, a następnie włączamy podgląd na palecie barw (poprzez dwukrotne kliknięcie na pobrany kolor). Program podaje wartości składowych S i V w skali od 0 do 255, tak samo jak OpenCV. Różnica występuje przy wartości H – tutaj program poda nam wartość w stopniach (od 0 do 360), podczas gdy OpenCV przyjmuje wartości H w przedziale od 0 do 180. Tak więc odczytaną wartość należy podzielić przez dwa.

Odczytana wartość jest wartością średnią, musimy jeszcze określić przedział wartości zapewniający odpowiednią tolerancję na zmiany warunków oświetlenia. Doboru tego najlepiej dokonać poprzez ocenę poprawności zaznaczenia linii (czyli obszaru pomiędzy dolnym a górnym progiem zakresu koloru). Poprawny rezultat wycięcia powinien wyglądać mniej więcej następująco:

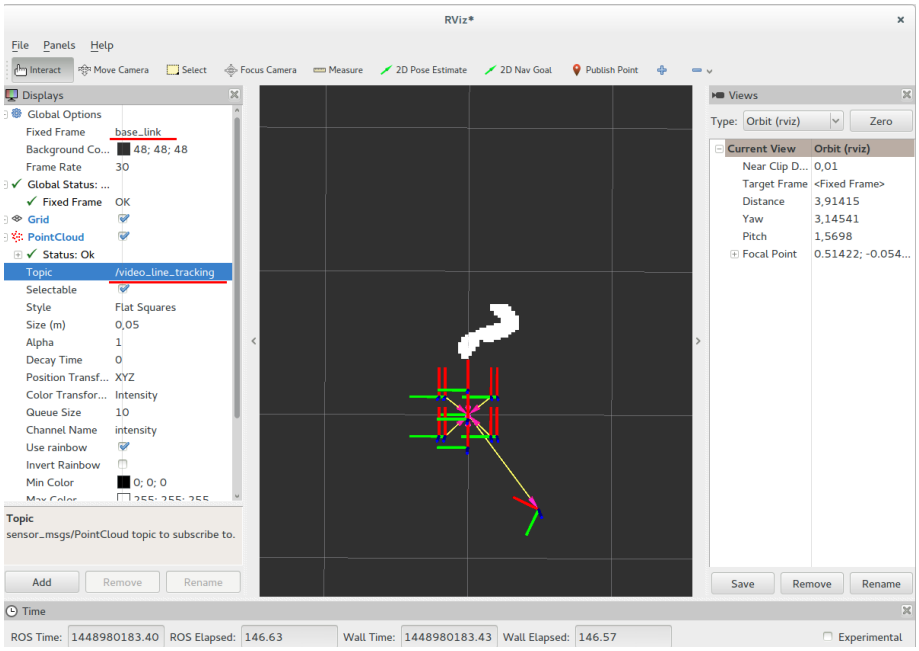
Jak widać jest on zaszumiony, warto więc poddać uzyskany obraz erozji. Ponadto widać, że wykryta linia jest dosyć gruba, nie warto więc publikować wszystkich jej punktów. Dlatego też obraz można pomniejszyć, np. ośmiokrotnie. Da to następujacy wynik:



Pozostaje dokonanie zamiany obrazu na listę punktów linii. Aby to wykonać, przechodzimy po wszystkich pikselach obrazu za pomocą podwójnej pętli *for*, a następnie dla każdego nieczarnego punktu (czyli o wartości różnej od zera) dokonujemy przeliczenia z układu współrzędnych kamery na układ współrzędnych podstawy robota. Musimy pamiętać, że układy te nie tylko są względem siebie przesunięte i obrócone, ale i wyrażone w innych jednostkach – musimy dokonać więc zamiany z pikseli na metry. Na szczęście nie jest tutaj wymagana metrologiczna dokładność, wystarczy że oszacujemy szerokość pola widzenia kamery z dokładnością do 10 cm. Wzajemne ułożenie układów pokazano na schemacie poniżej.



Opublikowaną chmurę punktów można zobaczyć w programie *rViz*, zmieniając nazwę topic'u z wyświetlaną chmurą punktów z odczytów sonarów na publikowana przez nas linię. Obie te chmury można też oczywiście oglądać jednocześnie – wtedy trzeba dodać nową instancje pluginu wyświetlającego te dane. Widok wizualizacji przedstawia rysunek poniżej (miejsca istotne konfiguracyjnie podkreślono czerwoną linią):



Uwaga: po zakończeniu rozwoju aplikacji do detekcji linii należy usunąć z niej wyświetlanie wszelkich okien z podglądem, a do podglądu rezultatu końcowego wykorzystywać program *rViz*. Wynika to z faktu, że podgląd okiem z obrazem w sesji SSH będzie wprowadzał znaczne

opóźnienia do pracy robota i może uniemożliwić mu poprawne wykonanie zadania.

Algorytm sterowania robotem

Program sterujący robotem subskrybuje chmurę punktów opisaną wyżej i na jej podstawie wyznacza sterowanie robota. Będzie się on składał z trzech funkcji, dwóch sterujących i jednej analizującej pojawiające się dane.

Podstawową funkcja sterowania jest funkcja stopu. Wysyła ona zerowe komendy prędkości, co skutkuje zatrzymaniem robota. Należy przy tym pamiętać, że aby robot przestał się poruszać nie wystarczy zaprzestać wysyłać nowe komendy prędkości – sterowniki robota Pioneer będą realizowały zawsze ostatnio otrzymaną komendę.

Drugą funkcją jest podążanie robota w wyznaczonym kierunku. W wersji podstawowej funkcja ta przyjmuje wyznaczony kierunek ruchu robota, a następnie publikuje komendę prędkości, składającą się ze stałej prędkości liniowej (na początku przyjąć wartość 0.1) oraz prędkości kątowej proporcjonalnej do otrzymanego kierunku ruchu. Układ taki działa jak regulator proporcjonalny.
Funkcja obsługująca przychodzące wiadomości analizuje je dwustopniowo. Najpierw sprawdza, czy wiadomość zawiera odpowiednio wiele punktów. Gdy nie ma ich wcale oznacza to, ze robot zjechał z trasy lub dojechał do końca linii. Gdy jest tylko kilka, mogą stanowić one szum, więc również nie powinny być brane pod uwagę.

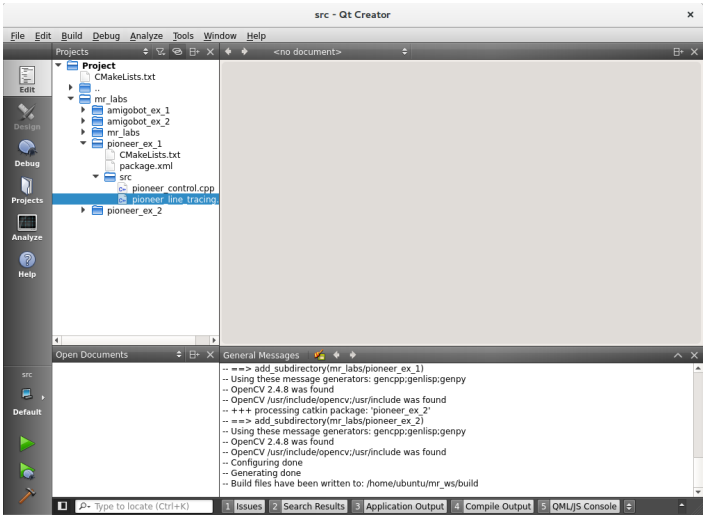
Jeżeli otrzymaliśmy odpowiednią liczbę punktów, to musimy w jakiś sposób na ich podstawie wyznaczyć kierunek ruchu robota. W tym ćwiczeniu sugerowaną metodą jest znalezienie środka ciężkości otrzymanej chmury punktów. Jest to kolejny punkt docelowy robota, a więc kierunkiem który należy podać do funkcji realizującej ruch jest składowa y tego punktu.

Implementacja algorytmów

Aby przystąpić do implementacji kodu należy w nowym oknie sesji SSH (**czyli na robocie**) przejść do katalogu src w aktualnym workspace oraz wpisać komendę :

```
cd ~/mr_ws/src
qtcreator CMakeLists.txt
```

Uruchomiony zostanie program *QT Creator*. Po otworzeniu drzewo projektu będzie wyglądało podobnie jak na rysunku poniżej.



Edytować będziemy dwa pliki. Algorytm detekcji linii zapisać należy w pliku *pioneer_ex_1/src//pioneer_line_tracking.cpp*. Z kolei algorytm sterowania robotem zapisany będzie w pliku *pioneer_ex_1/src//pioneer_control.cpp*. Aby uruchomić program do detekcji linii wpisujemy w terminalu sesji SSH komendę:

```
rosrun pioneer_ex_1 pioneer_line_tracing
```

Natomiast aby uruchomic program sterujący robotem, wpisujemy:

```
rosrun pioneer_ex_1 pioneer_control
```

Szkielet programu do detekcji linii wygląda następująco:

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud.h>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

/**
 * @brief robot_name   Current robot name.
 */
const string robot_name="batman";

/**
 * @brief fromXYWH      Calculate 2D point in robot frame coordinate based on
its position on the image.
 * @param x             X coordinate on image in pixels.
 * @param y             Y coordinate on image in pixels.
 * @param width         Image width.
 * @param height        Image height.
 * @return              Calculated 2D point.
 */
geometry_msgs::Point32 fromXYWH(int x, int y, int width, int height){
}

/**
```

```
 * @brief main          Main routine.
 * @param argc          Used as default in ros.
 * @param argv          Used as default in ros.
 * @return
 */
int main(int argc, char** argv)
{
    //ROS node initialization
    ros::init(argc, argv, "pioneer_line_tracking");
    ros::NodeHandle n;
    ros::Rate rate(30);

    //Subscriber
    ros::Publisher pub_cloud =
n.advertise<sensor_msgs::PointCloud>("/pioneer_line_tracking", 10);

    // Video open
    VideoCapture capture;
    capture.open(-1);
    if(!capture.isOpened()){
        cout << "Unable to open camera" << endl;
        return 1;
    }

    while(n.ok())
    {
        // Get frame
        Mat input;
        capture >> input;
        if(input.empty()) break;

        // Calculate points in robot frame
        // ...

        //Spin
        ros::spinOnce();

        //Sleep
        rate.sleep();
    }
    return 0;
}
```

Do napisania kodu przydatne mogą być następujace funkcje biblioteki OpenCV:

- operator >> wczytujący obraz z obiektu cv::VideoCapture do cv::Mat
- Mat::empty() - sprawdza czy macierz obrazu nie jest pusta
- cvtColor() - konwersja koloru
- inRange() - binaryzacja obrazu poprzez sprawdzenie, czy piksel należy do danego przedziału kolorów
- getStructuringElement() - funkcja zwracająca kernel do operacji morfologicznych
- erode() - erozja morfologiczna
- pyrDwon() - funkcja zmniejszająca obraz dwukrotnie
- resize() - funkcja zmieniająca rozmiar obrazu
- imshow() - wyświetlenie obrazu
- waitKey() - oczekiwanie na wciśniecie przycisku

Więcej informacji o tych funkcjach można znaleźć w dokumentacji biblioteki OpenCV.

Szkielet programu sterującego robotem wygląda przedstawiono poniżej:

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud.h>
#include <geometry_msgs/Twist.h>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

/**
 * @brief robot_name            Current robot name.
 */
const string robot_name="batman";

/**
 * @brief sub_cloud             Pointcloud data subscriber.
 */
ros::Subscriber sub_cloud;

/**
 * @brief pub_vel               Robot velocity publisher.
 */
ros::Publisher pub_vel;

/**
 * @brief getPointL2Norm        Get L2 (cartesian) distance from the point to
the coordinate system origin.
 * @param p                     Given point.
 * @return                      Calculated L2 distance.
 */
float getPointL2Norm(geometry_msgs::Point32 p){
    return sqrt(pow(p.x, 2) + pow(p.y,2) + pow(p.z,2));
}

/**
 * @brief stop                  Send stop command to the robot
 */
void stop(){
}

/**
 * @brief move                  Send move command to the robot. This is a line
follower, so it always has
 *                              linear velocity, with additional angular
velocity.
 * @param rotate                Angular velocity factor.
 */
void move(float rotate){
}

/**
 * @brief lineCallback          Process cloud of detected line points.
 * @param msg                   ROS point cloud message.
 */
void lineCallback(const sensor_msgs::PointCloudPtr& msg){
```

```
    // If 5 or less points in the pointcloud end the program
    const int min_num_points = 5;
    if(msg->points.size() < min_num_points){
        stop();
        exit(0);
    }

    //Move the robot!
}

/**
 * @brief main                      Main routine.
 * @param argc                      Used as default in ros.
 * @param argv                      Used as default in ros.
 * @return
 */
int main(int argc, char** argv)
{
    //ROS node initialization
    ros::init(argc, argv, "pioneer_control");
    ros::NodeHandle n;
    ros::Rate rate(30);

    // Subscriber and publisher init
    sub_cloud = n.subscribe("/pioneer_line_tracking", 10, &lineCallback);
    pub_vel = n.advertise<geometry_msgs::Twist>(string("/") + robot_name +
string("/aria/cmd_vel"), 10);

    while(n.ok())
    {
       //Spin
        ros::spinOnce();

        //Sleep
        rate.sleep();
    }

    return 0;
}
```

W wypadku nieprawidłowego działania robota jego ruch można zatrzymać poprzez zatrzymanie programu, lub szybciej poprzez jednoczesne sterowanie za pomocą pada – wysyłanie komend z prędkościa zerową z pada zagłuszy komendy ruchu wysyłane przez program.