EPD 2: Robot control. Introduction to Gazebo and RViz

Objectives

- First contact with the robot simulator Gazebo
- First contact with the tool RViz in ROS
- First program to control a robot (Turtlebot) to reach a goal

Assignments

1 Materials

In this EPD we will use two new tools integrated in ROS:

1. The 3D simulation engine Gazebo
2. The visualization tool RViz

2 The robot simulator Gazebo

We will use the robot Turtlebot to perform the lab. assignments. But in order to allow working at home and also perform experiments without the robot, we will use a robot simulator. In particular, we will use the simulator Gazebo.

The first step is to install the Turtlebot simulator under Gazebo by commanding:

```
sudo apt-get install ros-indigo-turtlebot-simulator
```

Install the ROS package with the simulator of the turtlebot robot.

Once installed, we can start the Gazebo simulator. Gazebo is a tool for robot simulation. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.

To launch Gazebo with the turtlebot robot already included we command:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

We have first to set up the environment variable TURTLEBOT_GAZEBO_WORLD_FILE to contain the path to the file empty.world in the ROS package turtlebot_gazebo

```
export TURTLEBOT_GAZEBO_WORLD_FILE=/opt/ros/indigo/share/turtlebot_gazebo/worlds/empty.world
```

We can see the robot in a scenario without obstacles. The *models* tab show the models that are represented. We can insert more models by using Insert Model. We can try to insert a box model, car model, etc.

Gazebo implements a physical simulator, and can emulate sensors and motions of the robot. This simulator is integrated into ROS by creating several nodes and topics on which you can read the robot sensors or publish commands to the simulated robot. In the case of the simulator, we can now write code that works the same with the simulator and the real robot.

In order to test this, we will install the turtlebot_teleop package.

```
sudo apt-get install ros-indigo-turtlebot-apps ros-indigo-turtlebot-rviz-
                                launchers
```

It provides launch files for the teleoperation for the turtlebot robot (either simulated or real) with different input devices. We will see how do it for a keyboard teleoperation. For that we command in a new terminal:

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Move the robot around with the teleop package

To know more:

You can find a description on the turtlebot packages in ROS at:
http://wiki.ros.org/Robots/TurtleBot

3 The visualization tool RViz

As commented, after running Gazebo there are several ROS nodes and topics with which you can communicate. The tool ***rqt*** from ROS allows us to see the current nodes, and how they are connected:

Launch the tool rqt by typing:

```
rqt
```

in a separate terminal. Then, select, from the Plugins menu, Introspection, Node Graph

We can also see the Publications, Subscriptions, Services and Connections of each node (also commanding rostopic list).

But also ROS has a visual tool to see the sensor data published into a ROS system. This tool is called RViz, and can be launched by typing:

```
rosrun rviz rviz
```

We will explain how you can show information into this visualization tool. We will use it to see how the Gazebo simulator is able to simulate as well the Kinect sensor and the robot.

This visualization tool, RViz, is just this. It is not a simulator. It only displays information. The same tool can be used with the real and the simulated robot.

There is a pre-configuration to show the most relevant topics from the Turtlebot. It can be run by commanding:

```
roslaunch turtlebot_rviz_launchers view_robot.launch
```

4 Coordinate frames

The simulated robot uses many coordinate frames to operate. All of them are published through the ROS /tf topic, and can be used by means of the TF library that we learned in EPD1.

Among them, we will be using two main frames:

- /odom, which is the world frame. The origin (0,0,0) is set at the first position of the robot when Gazebo is launched
- /base_link is a local frame attached to the robot (that is, it moves with the robot), with the X axis aiming to the direction of motion of the robot.

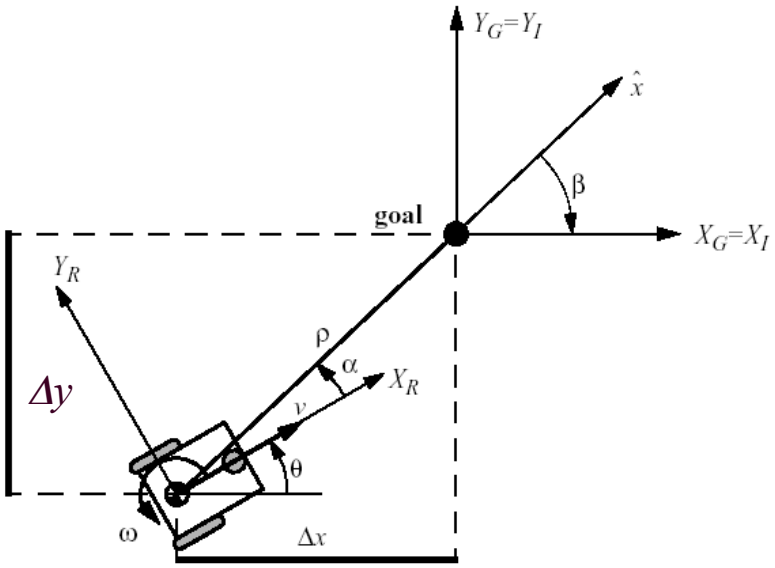

Figure 1: Coordinate frames. The world frame /odom (denoted as X_G and Y_G) and the robot local frame /base_link (denoted as X_R and Y_R)

Our objective will be to reach goals expressed in the world frame, in odom. One way to do it is to transform those points to the local frame, and then reasoning about the linear (v) and angular (ω) velocities that we have to command the robot.

For the conversion, we will use, from the tf library, the TransformListener class. It allows to ask for the transformation between any two given coordinate frames.

For instance, the following code transforms a point expressed in the odom frame to the base_link frame:

```
geometry_msgs::PointStamped goal;
goal.header.frame_id = "odom";

geometry_msgs::PointStamped base_goal;

[…]

tf::TransformListener listener;
listener.transformPoint("base_link", goal, base_goal);
```

We use the PointStamped class, that represents a point in a coordinate frame:

geometry_msgs/PointStamped Message
File: geometry_msgs/PointStamped.msg
Raw Message Definition

```
# This represents a Point with reference coordinate frame and timestamp
Header header
Point point
```

Compact Message Definition

std_msgs/Header header
geometry_msgs/Point point

In the field header, it is possible to indicate the coordinate frame of the point.

To know more:
You can find a tutorial on the transformListener at:
http://wiki.ros.org/tf/Tutorials

And more information on geometry data types at:
http://wiki.ros.org/geometry_msgs

5 A first node controlling Turtlebot

The main objective of this EPD is to be able make an initial node that controls the robot. The code of the node can be downloaded from BlackBoard.

Download the code and uncompress it into you ROS sandbox. Create a new package called epd2 depending on roscpp and tf

During the class the code will be explained. The main objective is to control the robot in order to reach a goal. The coordinates of the goal are passed as parameters to the node. Then, you have to complete the code for, given the goal coordinates and the robot coordinates, control the robot to reach the goal.

We will use a **proportional controller**. The idea is that the velocities (linear and angular) to be commanded to the robot will be proportional to the error between the robot position and the goal position. In particular, to the distance (ρ in Fig. 1) and the angular difference (α in Fig. 1) between the robot and the goal.

In order to do that, you have to understand the code and modify the function command.

Complete the function command so that the robot reach the goal

The important data to control the robot is the message to control the robot.

To control the robot, your program should publish messages at the topic /cmd_vel. The messages through this topic are of the type:
http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html

geometry_msgs/Twist Message

File: geometry_msgs/Twist.msg

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

6 A second node tracking paths with Turtlebot

The main objective of this part is to improve the initial node that controls the robot by adding other functionalities. Concretely, a follower of a path will be implemented. The code of the new node can be downloaded from BlackBoard, and corresponds to the executable c2.

Download the code c2.cpp and add it to the package. Include the adequate lines in CMakeLists.txt to compile and generate a new executable called c2.

During the class the code will be explained. Firstly, the path will be loaded from the file **plan.txt**. The path is given by a sequence of waypoints. The main objective is to control the robot in order to reach each waypoint of the plan and then command the next waypoint up to the final waypoint.

The plan is a `vector<geometry_msgs::Pose>` variable. This allows us to use implemented functions in the vector library.

The function `command` has been modified for the robot to be able follow a waypoint within the path. A output variable is added (type bool) to check if the current waypoint has been reached (bool command(double goal_x, double goal_y)).

Then, you have to complete the code to reach a waypoint and decide when the current waypoint is reached. In this case, the next waypoint of the plan should be commanded.

In order to do that, you have to understand the code and modify the functions command and main.

Complete the function command so that the robot can decide if the current waypoint has been reached
Complete the function main to command the corresponding waypoint in each instant

Finally, ROS_INFO messages will be used (ROS_INFO("XXXXXX.");). They are equivalent to std::cout <<"XXXXXX" << std::endl; but ROS_INFO messages provides more information.

The messages of the vector variable are of the following type:

geometry_msgs/Pose Message

File: geometry_msgs/Pose.msg

Raw Message Definition

```
# A representation of pose in free space, composed of postion and orientati
on.
Point position
Quaternion orientation
```

Compact Message Definition

geometry_msgs/Point position
geometry_msgs/Quaternion orientation