EPD3: Robot control. Collision avoidance using Kinect sensor.

Objectives

- First program to control a robot (Turtlebot) in a scenario with obstacles to reach a goal
- First program using the Kinect sensors to detect obstacles and command the robot up to the goal

Assignments

1 Materials

In this EPD we will continuous using the known tools integrated in ROS:

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

Moreover, we will add the Kinect sensor to avoid detected collisions from the obtained data. We will use a 2D laser scan readings because the point cloud of the Kinect sensor can be converted.

2 The robot simulator Gazebo

First, we start the Gazebo simulator by commanding:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Our goal is to avoid existing obstacles in the environment. We can insert obstacles by using Insert Model tab in Gazebo. For example, we can insert Cube 20k, Gas Station, Grey Wall, Office Building, and more in the scenario.

We need to use a sensor to detect the obstacles. Particularly, we will need to launch a node to obtain the 2D laser scan readings from the Kinect. This node and its topics are launched when Gazebo is run. Therefore, we will be able read the 2D laser scan readings. We can see with the tool *rqt* the publications. In this case we are interested in /scan[sensor_msgs/LaserScan].

Thus, we will subscribe to the topic /scan and receive this type of messages: sensor_msgs::LaserScan. The data structure is presented next:

sensor_msgs/LaserScan Message

File: sensor_msgs/LaserScan.msg

Raw Message Definition

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Compact Message Definition

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

autogenerated on Thu, 27 Mar 2014 00:22:28

We can know the first angle (*angle_min*) up to the last angle (*angle_max*). *ranges* provides the distance (in meters) for an given angle. Knowing this distance, its angle from *angle_min* and *angle_increment* and the maximum range (*range_max*) we can detect obstacles. Note that *angle_increment* provides the distance angular between measurements; therefore we can know the angle for each measured.

3 A first node controlling Turtlebot to avoid the detected collisions

The main objective of this EPD is to be able make an initial node that reaches a goal avoiding detected collisions. The code of the node can be downloaded from BlackBoard.

Download the package and uncompress it into you ROS sandbox.

During the class the code will be explained.

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

Complete the function command so that the robot reach the goal avoiding the obstacles

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
```

Finally, to detect obstacles, your program should subscribe to the topic /scan. The messages through this topic are of the type sensors_msgs::LaserScan.

4 Annex

In order to work at home on the detection of obstacles, several data files (extension bag) and the corresponding scenario can be downloaded from BlackBoard. Each scenario indicates both the location of the robot and the obstacles. Bag file contains the position of the robot (*x, y, theta*) and the reading obtained from the laser (*data_scan*). Therefore, you can detect the obstacle.

This table shows the scenario and corresponding bag file:

SCENARIOS	FILES	Obstacles
S1	scenario1.bag	5
S2	scenario2.bag	1
S3	scenario3.bag	1
S4	scenario4.bag	1
S5	scenario5.bag	1
S6	scenario6.bag	1
S7	scenario7.bag	1
S8	Scenario8.bag	2
S9	scenario9.bag	2

The steps to run your node (c3) are the next:

1. roscore
2. *rosbag play scenarioX.bag*
3. *rosrun epd3 c3 plan.txt*

Additionally, during playing, you can pause at any time by pressing *space*. When paused, you can step through messages by pressing *s*. Thus, you can check if the obstacles are detected correctly.