4 On-board sensors: depth camera, laser scan, bumpers, cliff sensors

In the previous package, a random robot controller has been developed. The controller did not use input data from sensors to assign velocity values. In a sense, the robot was operating blindly, or, in other words, in open-loop modality. However, sensory inputs are fundamental to cope with the challenges of real-world scenarios. To support safe and effective navigation, the TurtleBot2 is equipped with an RGB-D *depth camera*, an *array of bumpers*, and an *array of cliff sensors*, which are described in the two sections that follow.

4.1 Depth camera (a first introduction)

The depth camera returns a *point cloud* that can be used to assert the presence or not of objects within the 3D field of view of the camera.

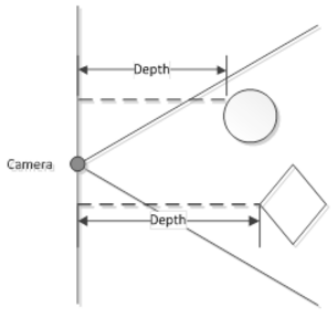

Figure 7: Depth of an object within the FOV of the camera. Each pixel gets a depth value.

In our real TurtleBot2, the RGB-D camera is an ORBBEC Astra, that has a declared FOV of 60° horizontal and 49.5° vertical, a depth image of 640 × 480 pixels updated at 30 Hz. The operational range is between 0.6 to 8 meters. The Microsoft Kinect, the sort of original RGB-D camera, has a FOV of 57° horizontal and 43° vertical, with an operational range of 0.9 to 3.5 meters (and same depth image size). The model which is include as a default RGB-D camera for the TurtleBot2 since ROS Indigo, is the *Asus Xtion Pro Live 3D Sensor*, shown in the figure. The working of sensor is based on an infrared emitter, an infrared camera, and an RGB camera. The FOV is 58° H, 45° V, 70° D (Horizontal, Vertical, Diagonal). The distance of use is between 0.8 and 3.5 meters.

Figure 8: The Asus Xtion Pro Live sensor which is the default mounted on the TurtleBot2 in ROS and Gazebo models.

For the time being, we will not use the depth camera, but rather we will use a 2D slice of its data, that would be equivalent of the output returned from a *planar laser scanner*

(range finder). A virtual sensor exists that precisely returns this information, which would tell about the presence of obstacles on a plane at the height of the depth camera sensor. This is performed by the `depthimage_to_laserscan` package.

☛ However, before moving to check what kind of messages the laser scan sensor provides, it is instructive to check what the camera "sees". At this aim first starts gazebo (if you don't have yet a real robot), for instance with the maze world from the homework:

```
$ export TURTLEBOT_GAZEBO_WORLD_FILE=~/catkin_ws/worlds/funky-maze.world;
> roslaunch turtlebot_gazebo turtlebot_world.launch
```

then, in two different shell, execute the following rosrun commands, that will run the `image_view` package, which is a viewer for ROS image topics (`http://wiki.ros.org/image_view`):

```
$ rosrun image_view image_view image:=/camera/rgb/image_raw
$ rosrun image_view image_view image:=/camera/depth/image_raw
```

where the first command shows what the robot sees with the RGB camera, while the second shows the depth images, where the grayscale encodes depth. In gazebo, you can manually move the robot and/or add objects to the scene, in front of the robots, and you will see how the images do change. If you have a real robot, you can keep track of what the robot is seeing in the real world. In the figure, the maze world is shown as from `Gazebo`, together with the RGB and depth images from `image_view`. We can also use `rviz` for an enhanced

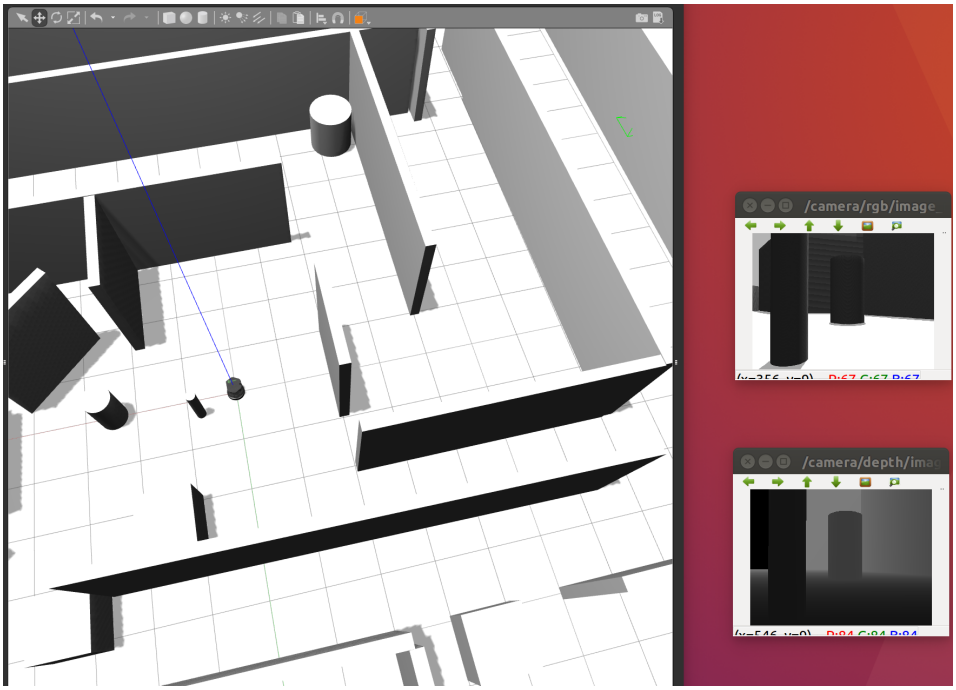

Figure 9: The maze world in Gazebo, and the RGB and depth images as seen from the robot camera.

visualization:

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

check the `PointCloud` button and you will see something similar to what shown in the figure below (Left). If you uncheck `PointCloud` and check `LaserScan`, you will visualize the output from the laser scan sensor, that will show the obstacle in range of the FOV of the robot, as shown in the figure (Right).

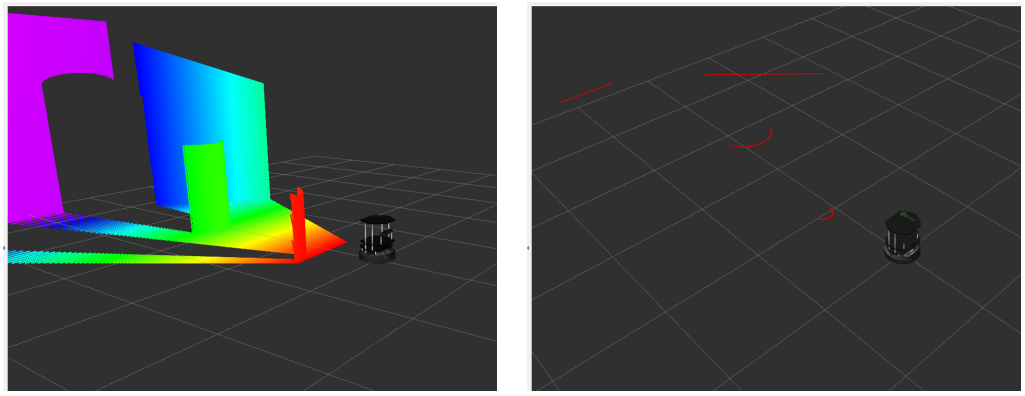

Figure 10: Visualization of the point cloud (Left) and of the laser scan (Right) in `rviz` for the same scenario of the previous figure.

4.2 Laser scan sensor

Going back to the 2D laser scan sensor, the output of the scan readings are published on the topic `/scan`:

```
$ rostopic info /scan
```

that makes use of `sensor_msgs/LaserScan` as type of topic messages:

```
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
```

with the following meaning:

```
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]
                         # (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.
```

☛ In order to subscribe and read the messages from the `/scan` topic, the above message type must be imported in the python code, with a statement:

```
from sensor_msgs.msg import Laserscan
```

The run-time value of the fields of the range finder messages (e.g., the *ranges* of the objects in the FOV) can be read as usual from the command line:

```
$ rostopic echo /scan/ranges
```

The sensor has a number of parameters (`http://wiki.ros.org/pointcloud_to_laserscan`):

```
-min_height (double, default: 0.0)          The minimum height to sample in the point cloud in meters.
-max_height (double, default: 1.0)          The maximum height to sample in the point cloud in meters.
-angle_min (double, default: -90)           The minimum scan angle in radians.
-angle_max (double, default: 90)            The maximum scan angle in radians.
-angle_increment (double, default: pi/360)  Resolution of laser scan in radians per ray.
-scan_time (double, default: 1.0/30.0)      The scan rate in seconds.
-range_min (double, default: 0.45)          The minimum ranges to return in meters.
-range_max (double, default: 4.0)           The maximum ranges to return in meters.
-target_frame (str, default: none)          If provided, transform the pointcloud into this frame before
                                              converting to a laser scan. Otherwise, laser scan will be
                                              generated in the same frame as the input point cloud.
-concurrency_level (int, default: 1)        Number of threads to use for processing pointclouds. If 0,
                                              automatically detect number of cores and use the equivalent
                                              number of threads. Input queue size is tied to this parameter.
-use_inf   (boolean, default: true)         If disabled, report infinite range (no obstacle) as range_max -+ 1.
                                              Otherwise report infinite range  as +inf.
```

☛ Parameter values can be retrieved or can be set from command line or from inside a program using the `rosparam` service:

```
$ rosparam list
$ rosparam get /depthimage_to_laserscan/range_min
```

☛ In addition, the package `/depthimage_to_laserscan` publishes the topic `/depthimage_to_laserscan/parameter_descriptions` that precisely describes the parameters and their values in use:

```
$ rostopic echo /depthimage_to_laserscan/parameter_descriptions
```

Many complex packages include such a `parameter_descriptions` topic precisely to ease the access to relevant parameters. Also an `parameters_updates` topic is available for dynamic updates. As a general rule, it is necessary to check what are the specific parameters of a sensor before using it!

In particular, for the laser scan sensor, the `ranges[]` arrays returns a depth value for the presence (or not) of an object along n radial directions, where n depends on the max and min scan angles (`angle_min` and `angle_max`), and on the angle increment parameter, `angle_increment`. If an object is detected at a distance d_i along the i-th radial direction, the distance value is reported in `ranges[i]`. If nothing is detected (up to the maximum range), a `NaN` is reported (or any arbitrary value over the declared ranges). `ranges[1]` corresponds to the rightmost scan direction with respect to the heading of the robot, while `ranges[n]` corresponds to the leftmost one. `ranges[n/2]` corresponds to the measure along the heading of the robot. Since the size of the depth image is 640 × 480 pixels, the number n is equal to 640 (an horizontal slice). The difference between max and min ranges divided by n must be then equal to the angular resolution.

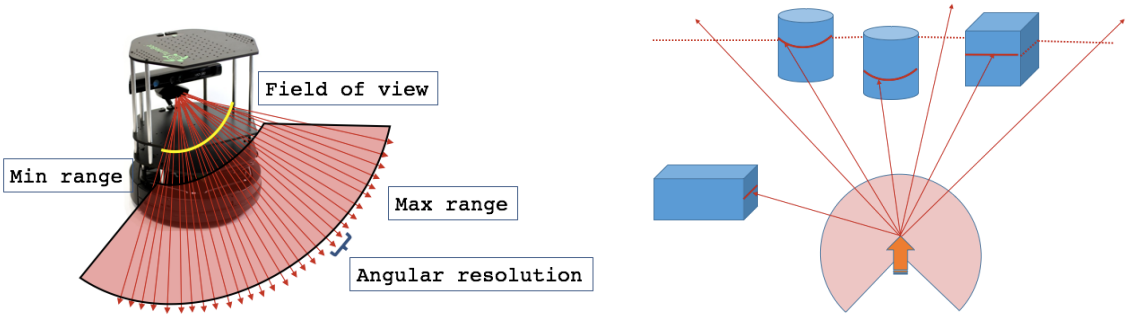

Figure 11: Illustration of laser scan operations.

4.3 Bumpers

In the TurtleBot2 there are *three bumper sensors* in the base of robot (which is the *Kobuki* robot):

- one in the middle-front;
- one in the left front;
- one the right front.

☛ The bumper sensor is a mechanical sensor that helps the robot *detecting collision with obstacles*. When the robot touch an external object one bumper sensor (if in correct position) gets pressed and throws a *bumper event*. A bumper event is also generated when the bumper is released.

Bumper messages following a bumper event are published in the topic:

```
/mobile_base/events/bumper
```

As usual, if we check the info of the topic with:

```
$ rostopic info /mobile_base/events/bumper
```

we find that topic messages are of type:

```
kobuki_msmgs/BumperEvent
```

which means that a statement

```
from kobuki_msmgs.msg import BumperEvent
```

will need to be part of the initial part of a python code using bumpers.

To check what is the structure of the `BumperEvent` messages:

```
$ rosmsg info kobuki_msmgs/BumperEvent
```

which shows the following:

```
# bumper
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2

# state
uint8 RELEASED = 0
uint8 PRESSED  = 1

uint8 bumper
uint8 state
```

The `state` field says what actions has triggered the event, while the `bumper` field says to which bumper the event refers to.

The state of the bumpers can be check at any time from the command line echoing the topic:

```
$ rostopic echo /mobile_base/events/bumper
```

For instance, if in Gazebo you manually move the robot in order to let it touch an obstacle, the above topic should report it.

4.4 Cliff sensors

Cliff sensors are responsible for *detecting cliffs and altitude changes* when the robot is moving, especially to prevent crashes when reaching stairs. They are located in the bottom of the Kobuki base. The behavior is very similar to the bumper sensor. Likely, we are not going to use cliff sensors, however, they are described below for completeness.

There are *three cliff sensors*:

- one in the center,
- one in the left side,
- one in right side.

Cliff events are published in the topic:

```
/mobile_base/events/cliff
```

Again, getting the info from `rostopic` and the type of message from `rosmsg` we get that type is:

```
kobuki_msmgs/CliffEvent
```

which means that a statement

```
from kobuki_msmgs.msg import CliffEvent
```

will need to be part of the initial part of a python code using cliff sensors.

To check what is the structure of the `CliffEvent` messages:

```
$ rosmsg info kobuki_msmgs/CliffEvent
```

which shows the following:

```
# cliff sensor
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2

# cliff sensor state
uint8 FLOOR = 0
uint8 CLIFF = 1

uint8 sensor
uint8 state

# distance to floor when cliff was detected
uint16 bottom
```

5 Setting the robot pose for simulation experiments

When running a simulation experiment, it might be appropriate to set he initial pose of the robot at a specific location of the environment and at a specific orientation. This can be realized in different ways.

5.1 Using `world` files

First, the pose can be flexibly specified in the `world` file which is given as input to a `roslaunch` command. We will study later how to create or modify a world file, therefore, let's skip this for the time being.

5.2 Using environment variables

A way which is related to the changing a world file, is by *exporting the environment variable holding the robot pose* when calling gazebo, for instance:

```
$ export ROBOT_INITIAL_POSE="-y 2 -x 3 -z 0 -R 0 -P 0 -Y 2";
> roslaunch turtlebot_gazebo turtlebot_world.launch
```

where the pose is set in terms of position variables (`x,y,z`) and orientation ones (`Roll Pitch, and Yaw`). Any subset of the pose variables can be set, with the remaining ones taking their default values. In a similar way, also the initial `Twist` (linear and angular velocity vectors) can be set.

In Gazebo, after selecting the robot, in the left panel it is possible to check in `[Property - Value]` that the value of the pose has been assigned correctly (as well as what is the pose of the robot or of other objects at any time).

5.3 Manually

☛ Of course, an easy, yet not automatic way of changing the pose is to do it *manually*, moving the robot inside Gazebo (first select, and then move). The robot can be manually moved at any time.

5.4 Using a set model `service`

In a more general way, from the inside of a simulation, it is possible to set the robot pose using a *service*, analogous to what done for reading the ground truth. In this way the service allows to set a value, rather than returning one. How to use a service for the purpose is shown in the code below.

One issue with this way of proceeding consists in the so-called problem of the *kidnapping robot*: in pratice the robot is being teleported, which means that its *state* knowledge is not valid anymore (e.g., its odometry information would be totally wrong after the teleportation act).

The code below is the same as in `pose-monitor.py`, except for a modification in the `__init__` function and in the `import` statements (lines 15–16, 40–52). The code for the functions other than the `__init__` one are not reported being the same. The node gets connected to the service for setting the *state of an entity* in the Gazebo world and then sets the pose of the `mobile_base` entity, that physically moves the TurtleBot2, by assigning `position.x=2` (all the other parameters stay from default). If Gazebo is launched first as usual, and then `rosrun` this node, the robot will be "teleported" in the new coordinate position (from the default one at `0,0,0`).

```
1 #!/usr/bin/env python
2
3 import rospy
4
```

```
5  # Import the Odometry message
6  from nav_msgs.msg import Odometry
7
8  # Import the Twist message
9  from geometry_msgs.msg import Twist
10
11 import tf
12
13 # for the ground truth
14 from gazebo_msgs.srv import GetModelState
15 from gazebo_msgs.srv import SetModelState
16 from gazebo_msgs.msg import ModelState
17
18 class PoseMonitor():
19
20     def __init__(self):
21         # Initiate a named node
22         rospy.init_node('pose_monitor', anonymous=True)
23
24         self.odom_sub = rospy.Subscriber('/odom', Odometry, self.callback_odometry)
25
26         self.vel_change_sub = rospy.Subscriber('/change', Twist, self.callback_velocity_change)
27
28         self.rate = rospy.Rate(1)
29
30         self.report_pose = False
31
32         print("Wait for GET service ....")
33         rospy.wait_for_service("gazebo/get_model_state")
34         #rospy.wait_for_service("gazebo/spawn_model")
35
36         print(" ... Got it!")
37
38         self.get_ground_truth = rospy.ServiceProxy("gazebo/get_model_state", GetModelState)
39
40         print("Wait for SET service ....")
41         rospy.wait_for_service("gazebo/set_model_state")
42         #rospy.wait_for_service("gazebo/spawn_model")
43
44         print(" ... Got it!")
45
46         self.model_state = ModelState()
47
48         self.model_state.model_name = "mobile_base"
49         self.model_state.pose.position.x = 3
50
51         self.set_model_state = rospy.ServiceProxy("gazebo/set_model_state", SetModelState)
52         self.set_model_state(self.model_state)
53
54     .... (the same as in pose-monitor.py)
```