EPD 2: ROS concepts and Introduction to robot sensing

Objectives

- Introduction to the ROS parameter server.
- Introduction to launch files.
- Use the parameter server to customize our node developed in EPD1.
- Use of topic subscription to receive the navigation goal.
- First contact with robot sensors to develop a simple collision checker.

Assignments

1 Introduction to the parameter server of ROS

The main idea of ROS is that it is composed by basic packages with a simple mission given to them. In this way there are several standard packages that are able to solve a basic robotic problem. Here we give you some examples:

- Localization of the robot: i. e. where a robot is in a map.
- Mapping: sensor measures are accumulated to generate a map.
- SLAM: simultaneous localization and mapping.
- Navigation: which path the robot has to follow in order to reach a goal in a map?
- Control: how the robot should follow a path.
- Obstacle avoidance: robot should safely follow a path without colliding with dynamic (moving) obstacles.
- Sensing: there are packages for communicating with the sensors onboard the robot (cameras, lasers, GPS…)
- Drivers: these modules communicate with the low level part of the robot so that we can give it commands by using the ROS middleware (topics and services).

In order to customize the behavior of each module, ROS has a parameter server that the modules can read to configure themselves. In this EPD, we will first see how to configure the parameters of our node, developed in EPD1, by using the parameter server.

Definition: parameter server

A parameter server is a shared, multi-variate dictionary that is accessible via network APIs. The ROS nodes can use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify it if necessary.

Definition: parameters

Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to protect parameter names from colliding. The hierarchical scheme also allows parameters to be accessed individually or as a tree. For example, for the following parameters.

Types of parameters:

The Parameter Server uses XMLRPC data types for parameter values, which include:

- 32-bit integers
- booleans
- strings
- doubles
- iso8601 dates
- lists
- base64-encoded binary data

Private parameters:

For the internal configuration of a node, they usually read their private parameters. For example: a camera can have as configuration option its resolution or color mode. Similarly, our node that reaches a goal can have as internal parameter its maximum speed.

Using rosparam in command line

The ROS parameter server has a command line interface that allows us to get and set parameters. Examples:

```
$ rosparam get /<param_name>
$ rosparam set /<param_name> <value>
```

We can also watch a list of the parameters for all the running nodes:

```
$ rosparam list
```

We can also load a set of parameters stored in a YAML file (https://yaml.org/) with the option "load":

```
$ rosparam load <file_name>.yaml
```

Example of yaml file:

goal:

x: 5.0

y: 3.0

2 Introduction to launch files

ROS provides the roslaunch utility that can be used to easily execute several ROS nodes. It parses launch files (.launch) that are written in XML format. Inside this files we can define nodes to be executed with the tag *<node>* (see example). We can define external arguments to a launch file with the *<arg>* tag. Finally, we can include other launch files with the *<include>* tag, we can specify some of the parameters of an included launch file by adding *<arg>* tags to it. For more information, please refer to http://wiki.ros.org/roslaunch

Setting parameters in launch files

roslaunch also provides a way to define parameters of a node, it is to define them inside a launch file. The *<param>* tag inside a node can be used to define a private parameter of that node (info: http://wiki.ros.org/roslaunch/XML/param).

<param name="param_name" type=""str|int|double|bool|yaml" value="value" />

Finally, we can also load a YAML of parameters by using a tag similar to *<rosparam command="load" file="PATH/TO/FILE.yaml">*. For more information please refer to http://wiki.ros.org/roslaunch/XML/rosparam.

An example of a launch file (epd1_control.lauch):

```
<launch>
        <env name="TURTLEBOT3_MODEL" value="burger" />
        <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_empty_world.launch"/>
        <node pkg="epd1" type="control.py" name="epd1_control" output="screen">
                <!-- we could place some param tags here -->
        </node>
</launch>
```

To be detected by ROS, we need to place the launch files inside a directory call "launch" inside our ROS packages. For instance, the previous launch file, should be placed in "*...rva_ws/src/epd1/launch/epd1_control.launch*".

To run launch files we must use the command "roslaunch". From the container we could do:

```
$ roslaunch epd1 ep1_control.launch
```

* Before that, do not forget to source the ROS ecosystem, compile your workspace and source it:

```
$ source opt/ros/noetic/setup.bash
$ catkin_make                  # from you rva_ws directory
$ source devel/setup.bash      # from you rva_ws directory
```

Getting parameters from the parameter server in a node

We have seen how to set parameter by the command line and by a launch file. Now, we show how to read a parameter from a node.

Getting a parameter is as simple as calling rospy.get_param(param_name):

```
# get a global parameter
rospy.get_param('/global_param_name')

# get a parameter from our parent namespace
rospy.get_param('param_name')

# get a parameter from our private namespace
rospy.get_param('~private_param_name')
```

Futher information: http://wiki.ros.org/rospy_tutorials/Tutorials/Parameters

3 Use the parameter server to customize a node

- Let's create a new ROS package called epd2 in our workspace (check the EPD1 to remember how to do it). The dependencies of this package must be *rospy, tf, tf2_ros, geometry_msgs* and *sensor_msgs*.
- Create the directories "scripts" and "launch" inside you new package epd2.
- Download the files "controlCollisionCheck.py" and "epd2.launch" from the EPD2 directory in the Aula Virtual. Place them inside your "scripts" and "launch" directories respectivelly.
- Compile your workspace and source it.

The provided code will be explained in the class. You will see how we can configure some parameters of our robot controller, and how to launch some nodes by using the launch file "epd2.launch". After that, you must complete the following assignment:

1. ***Extend the node controlCollisionCheck.py by adding the necessary code to set a maximum linear velocity value and a maximum angular velocity value.***
2. ***Extend the launch epd2.launch to set the maximum velocity parameters.***
3. ***Use the maximum velocity parameters obtained to saturate your robot velocity commands.***

4 Receiving the navigation goal through a topic

The provided node "controlCollisionCheck.py" includes a new topic subscription:

```
rospy.Subscriber("move_base_simple/goal", PoseStamped, self.goalCallback)
```

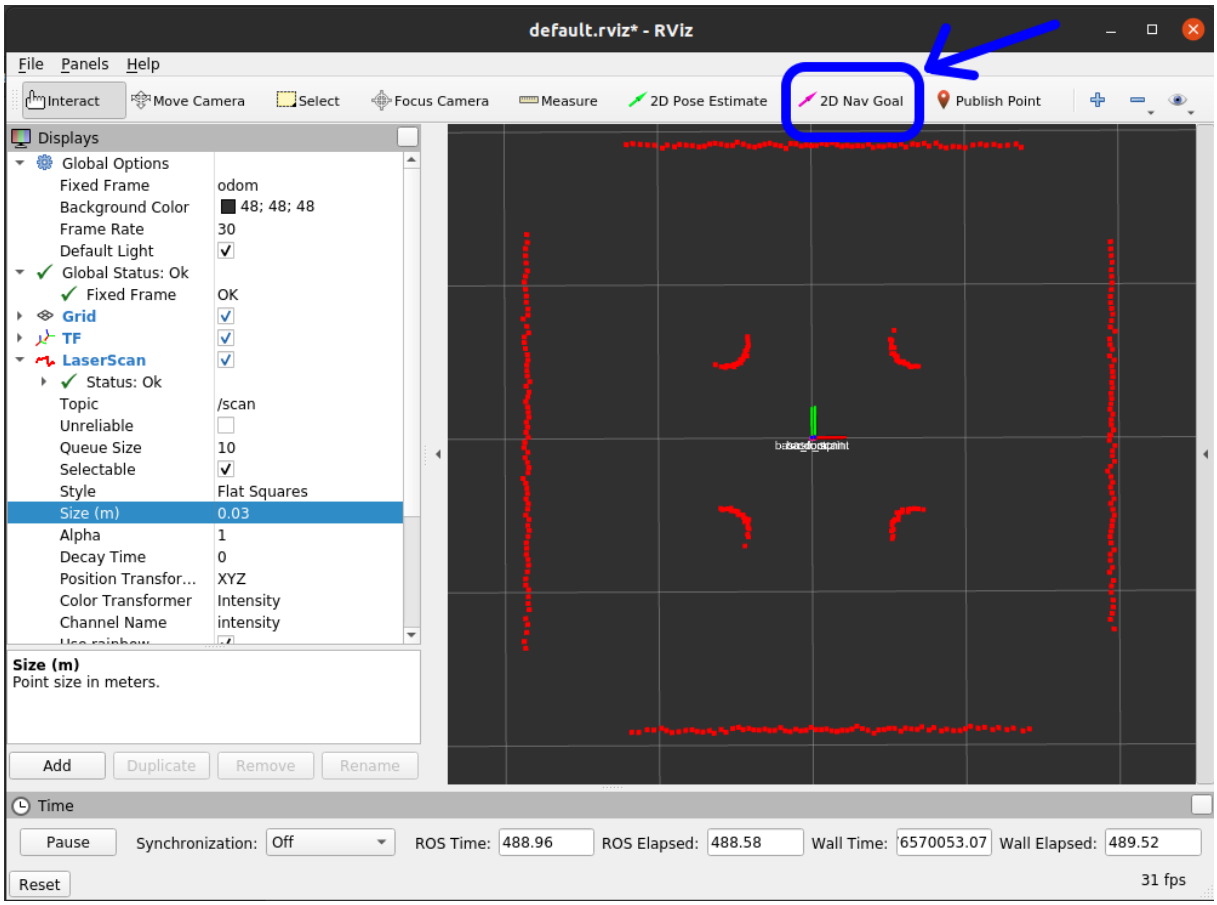
Our node will be listening to the topic "*move_base_simple/goal"* in which we will receive messages of type geometry_msgs/PoseStamped (http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/PoseStamped.html). Every time a new message arrives, the function *goalCallback* is triggered. This way, we (or others) can publish navigation goals and our node will proccess them.
We can send navigation goals in two ways:

- by command line and filling the fields with the desired values:

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped "header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: ''
    pose:
      position:
        x: 0.0
        y: 0.0
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 0.0"
```

- Or a simpler one, by using Rviz. Altough the purpose of Rviz is visualization, it also includes some tools to publish some particular data in ROS. It includes a tool that allows us to publish navigation goals by clicking on the position that we want to be our goal. First, We must press the "2D Nav Goal" button (see image below), and then click in the desired position in the main visualization screen. Then, the tool will publish a PoseStamped message through

the topic "*move_base_simple/goal*". The goal will be published regarding the coordinates frame indicated in the field "Fixed frame" of the right menu. This way we can send goals easily to our robot controller.



Try to send different goals by using Rviz to your controller node

5 Introduction to sensing: developing a simple collision checker

If we want to avoid obstacles, we need to use a sensor to detect the obstacles. Particularly, our turtlebot3 robot includes a 2D laser range finder: https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/ .
The Gazebo simulator is able to simulate this sensor among many others. Therefore, we will be able read the 2D laser scan readings, since our simulator is publishing this data. We can see with the tool *rqt* the publications. In this case we are interested in /scan topic [sensor_msgs/LaserScan].

From a 2D laser scan we obtain a set of distances (ranges) to the obstacles around the sensor. Each distance position in the vector of distances corresponds to a known angle, so we have a set of polar coordinates of the obstacles regarding the center of the sensor.

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*). R*anges* 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 angular distance between measurements. Therefore we can know the angle for each measurement.

The main objective of this EPD is to develop a robot local controller which is able to lead the robot to a given goal, and to check possible collision with obstacles by using the data provided by a 2D laser scan sensor.

Fill the function checkCollision, in the node controlCollisionCheck.py, in order to detect a possible collision if the robot get close to an obstacle. The method should return True in that situation and the robot will stop.