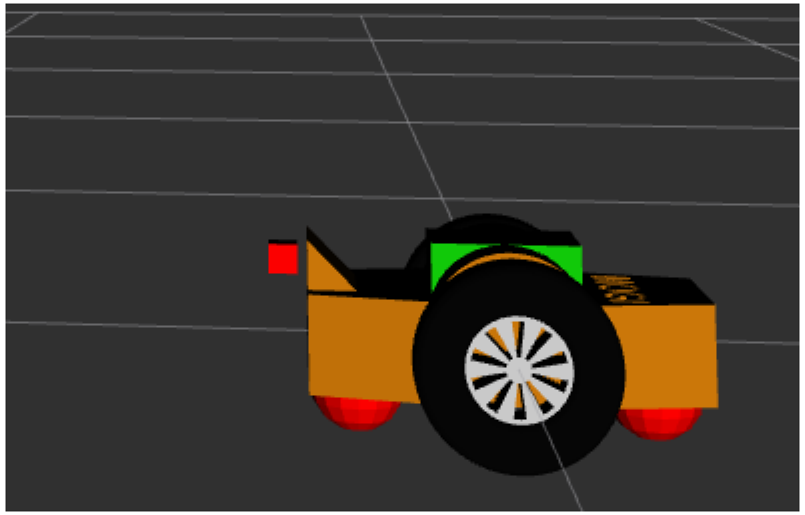
Experimental Robotics Laboratory – Vision Sensors and CV

Carmine Tommaso Recchiuto

Camera sensors



```
<gazebo reference="camera_link">
  <sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.1</near>
        <far>15</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <!-- Noise is sampled independently per pixel on each frame.
             That pixel's noise value is added to each of its color
             channels, which at that point lie in the range [0,1]. -->
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
      <optical_frame_id>camera_link_optical</optical_frame_id>
      <camera_info_topic>camera/camera_info</camera_info_topic>
    </camera>
    <always_on>1</always_on>
    <update_rate>20</update_rate>
    <visualize>true</visualize>
    <topic>camera/image</topic>
  </sensor>
</gazebo>
```

Camera sensors

We can see that both /camera/camera_info and /camera/image topics are forwarded. But this is still not the ideal way to forward the camera image from Gazebo. Without compression the 640x480 camera stream consumes almost 20 MB/s network bandwith which is unacceptable for a wireless mobile robot!!

ROS has a very handy feature with it's image transport protocol plugins, it's able to automatically compress the video stream in the background without any additional work. This feature however doesn't work with the parameter_bridge

To activate this feature, we need to run an additional node, the *image_bridge node,* which is defined in the the ros_gz_image package.

Camera sensors

We remove #"/camera/image@sensor_msgs/msg/Image@gz.msgs.Image", from the parameter bridge, since we do not need to forward directly that topic for the reason seen before.

We add the image_bridge node:

```
gz_image_bridge_node = Node(
        package="ros_gz_image",
        executable="image_bridge",
        arguments=[
            "/camera/image",
        ],
        output="screen",
        parameters=[
            {'use_sim_time': LaunchConfiguration('use_sim_time'),
             'camera.image.compressed.jpeg_quality': 75},
        ],
    )
```

Very important: remember to add the new node to the launchDescription:

launchDescriptionObject.add_action(gz_image_bridge_node)

Camera sensors

While you cannot see compressed images from rviz, you can check the differences between the bandwidth of the two topics (raw and compressed) and the related images by using rqt.

If compressed images are not visible in rqt, you have to install the plugins you want to use:

```
sudo apt install ros-jazzy-compressed-image-transport: for jpeg and png compression
sudo apt install ros-jazzy-theora-image-transport: for theora compression
sudo apt install ros-jazzy-zstd-image-transport: for zstd compression
```

Camera sensors

We already set up the jpeg quality in the image_bridge node with the following parameter:

'camera.image.compressed.jpeg_quality': 75

But how do we know what is the name of the parameter and what other settings do we can change? To see that we will use the rqt_reconfigure node:

ros2 run rqt_reconfigure rqt_reconfigure

We can play with the parameters here, change the compression or the algorithm as we wish and we can monitor it's impact with rqt.

LIDAR

LIDAR (an acronym for “Light Detection and Ranging” or “Laser Imaging, Detection, and Ranging”) is a sensing technology that uses laser light to measure distances. LIDAR sensor typically emits pulses of laser light in a scanning pattern (2D or 3D) and measures how long it takes for the light to return after hitting nearby objects. From this, the system computes distances to obstacles or surfaces in the environment.

By continuously scanning the surroundings, the LIDAR provides a 2D or 3D map of distances to any objects around the robot. Lidars are simple and important sensors of almost every mobile robot applications, it's widely used in Simultaneous Localization and Mapping (SLAM) algorithms which use LIDAR scans to build a map of the environment in real time while also estimating the robot’s pose (position and orientation) within that map.

LIDAR

As usual we need to modify: - URDF

```
<joint type="fixed" name="scan_joint">
  <origin xyz="0 0 0.15" rpy="0 0 0"/>
  <child link="scan_link"/>
  <parent link="base_link"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
</joint>


<link name='scan_link'>
  <inertial>
    <mass value="1e-5"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia
        ixx="1e-6" ixy="0" ixz="0"
        iyy="1e-6" iyz="0"
        izz="1e-6"
    />
  </inertial>
  <collision name='collision'>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size=".1 .1 .1"/>
    </geometry>
  </collision>


  <visual name='scan_link_visual'>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
       <mesh                    filename                    =
"package://bme_gazebo_sensors/meshes/lidar.dae"/>
    </geometry>
  </visual>
</link>
```

Gazebo file

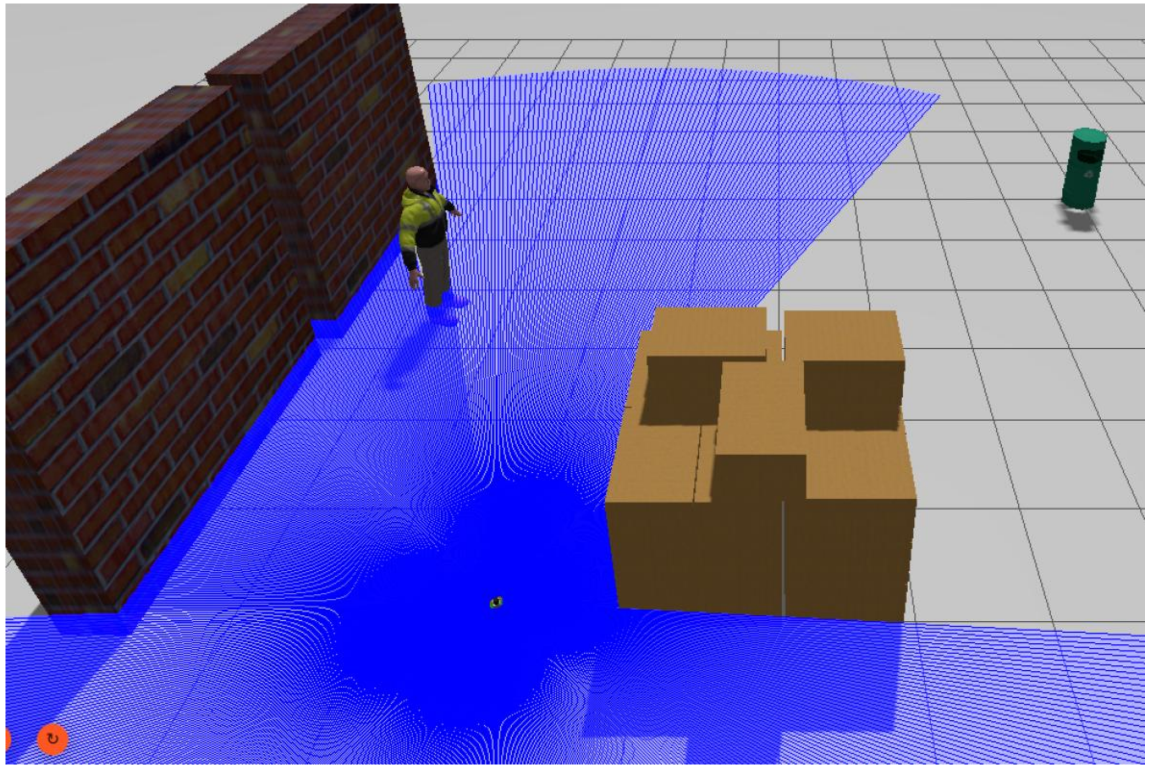
```
<gazebo reference="scan_link">
   <sensor name="gpu_lidar" type="gpu_lidar">
     <update_rate>10</update_rate>
     <topic>scan</topic>
     <gz_frame_id>scan_link</gz_frame_id>
     <lidar>
       <scan>
         <horizontal>
           <samples>720</samples>
           <!--(max_angle-min_angle)/samples * resolution -
->
           <resolution>1</resolution>
           <min_angle>-3.14156</min_angle>
           <max_angle>3.14156</max_angle>
         </horizontal>
         <!-- Dirty hack for fake lidar detections with
ogre 1 rendering in VM -->
         <!-- <vertical>
             <samples>3</samples>
             <min_angle>-0.001</min_angle>
             <max_angle>0.001</max_angle>
         </vertical> -->
       </scan>
       <range>
         <min>0.05</min>
         <max>10.0</max>
         <resolution>0.01</resolution>
       </range>
       <noise>
           <type>gaussian</type>
           <mean>0.0</mean>
           <stddev>0.01</stddev>
       </noise>
       <frame_id>scan_link</frame_id>
     </lidar>
     <always_on>1</always_on>
     <visualize>true</visualize>
   </sensor>
</gazebo>
```

LIDAR

…and add the topic in the parameter bridge:

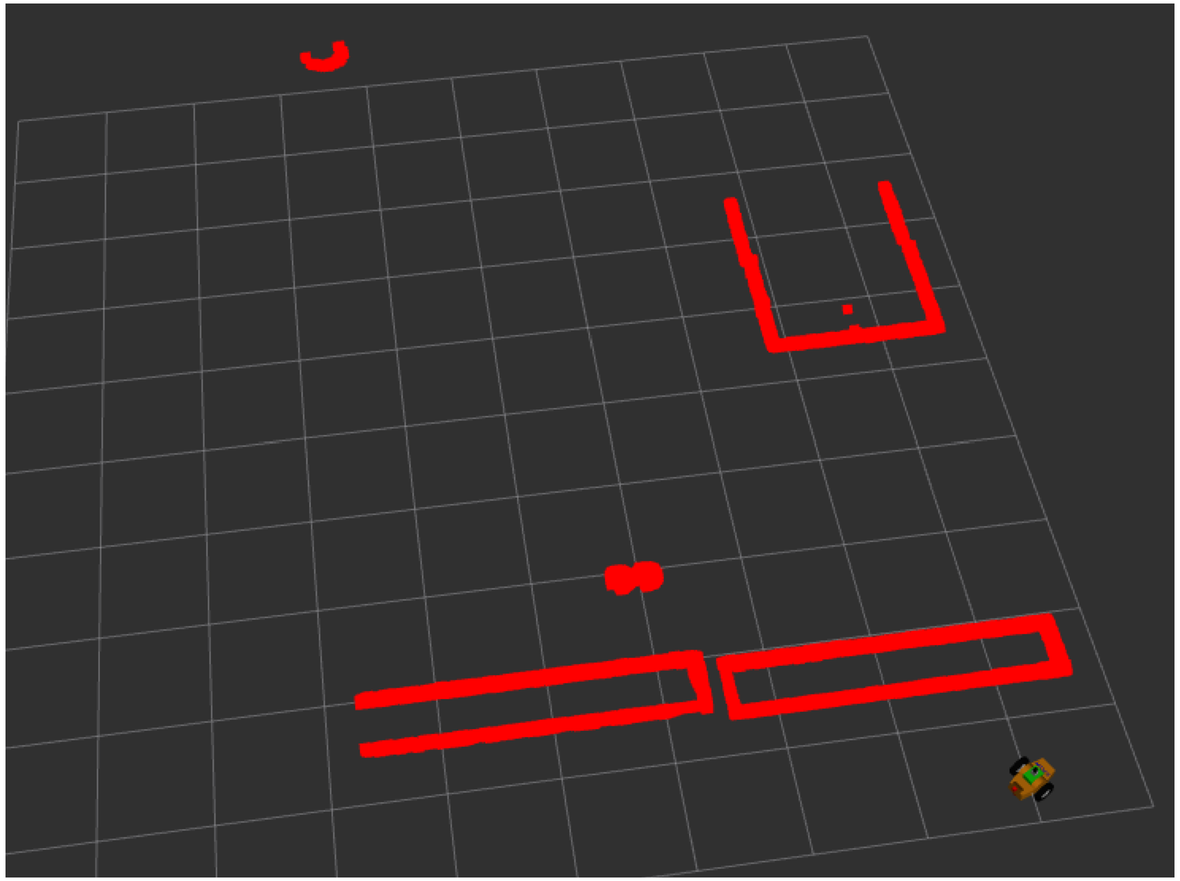
```
"/scan@sensor_msgs/msg/LaserScan@gz.msgs.LaserScan",
```

We can also verify the rendering of lidars in Gazebo with the Visualize Lidar tool



LIDAR

If we increase decay time of the visualization of lidar scans and we drive around the robot we can do a very simple "mapping" of the environment. Although in the course we will see that mapping algorithms are more complicated, usually this a good quick and dirty test on real robots if odometry, lidar scan and the other components are working well together



3D LIDAR

If we want to simulate a 3D lidar we only have to increase the number of vertical samples together with the minimum and maximum angles. For example the following vertical parameters are matching a Velodyne VLP-32 sensor:

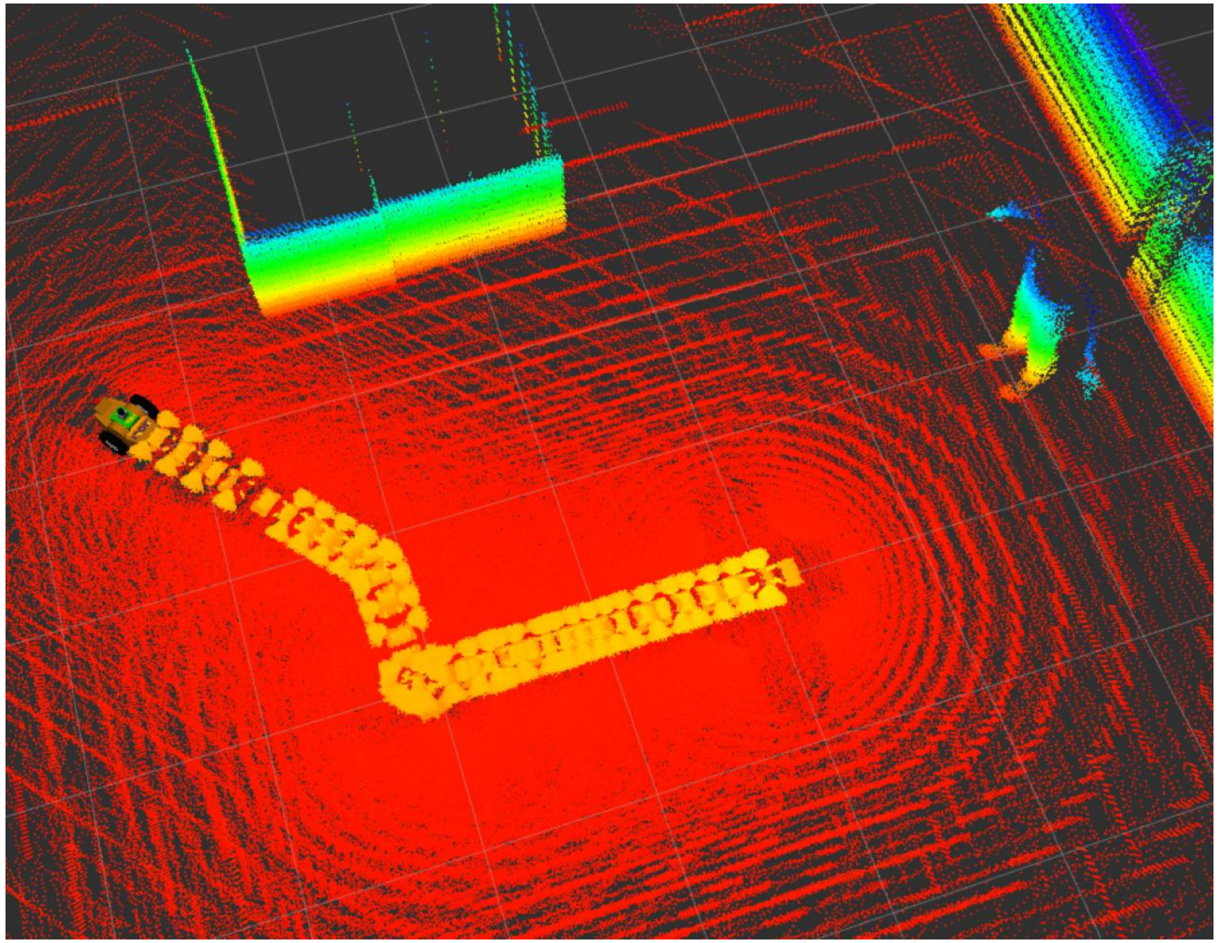
```
<vertical>
     <samples>32</samples>
     <min_angle>-0.5353</min_angle>
     <max_angle>0.1862</max_angle>
</vertical>
```

Again, we need to add one more topic in our launch file:

```
"/scan/points@sensor_msgs/msg/PointCloud2@gz.msgs.PointCloudPacked",
```

3D LIDAR

As before, we could increase the decay time just as we did with the 2D points.



RGBD Camera

Another way to get 3D pointclouds around the robot is using an RGBD camera which can tell not just the color but also the depth of every single pixel. To add an RGBD camera let's replace the conventional camera with this one

```
<gazebo reference="camera_link">
 <sensor name="rgbd_camera" type="rgbd_camera">
  <camera>
   <horizontal_fov>1.25</horizontal_fov>
   <image>
    <width>320</width>
    <height>240</height>
   </image>
   <clip>
    <near>0.3</near>
    <far>15</far>
   </clip>
   <optical_frame_id>camera_link</optical_frame_id>
  </camera>
  <always_on>1</always_on>
  <update_rate>20</update_rate>
  <visualize>true</visualize>
  <topic>camera</topic>
  <gz_frame_id>camera_link</gz_frame_id>
 </sensor>
</gazebo>
```

Notice that differently from before, we need to use camera_link here as well (the orientation here is already correct)

RGBD Camera

As you can see from the plugin, this publishes on the topic “camera”. Actually, we have two topics:

/camera/depth_image , which provides a grayscale camera stream where the grayscale values correspond to the distance of the individual pixels

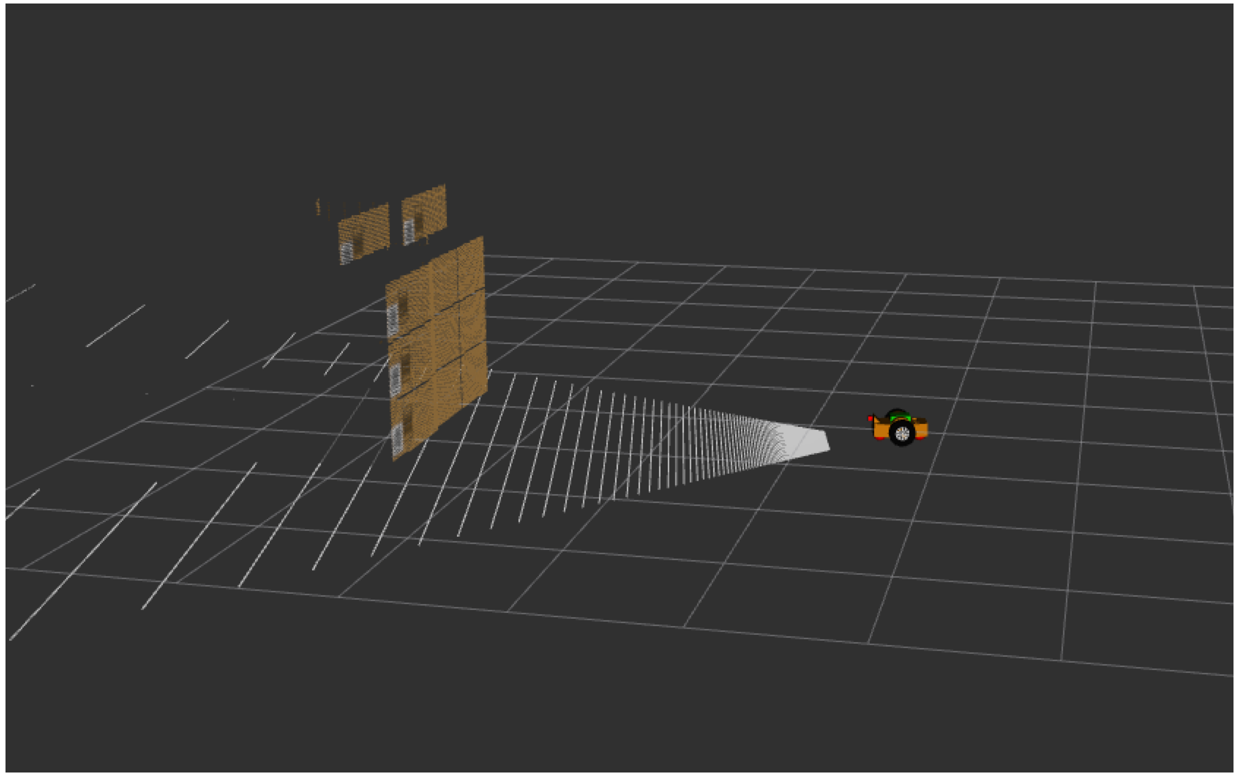
/camera/points , which is a 3D pointcloud, the same type as the 3D lidar's point cloud. We can visualize it in RViz just as any point clouds.

Let's add the topics to the parameter bridge:

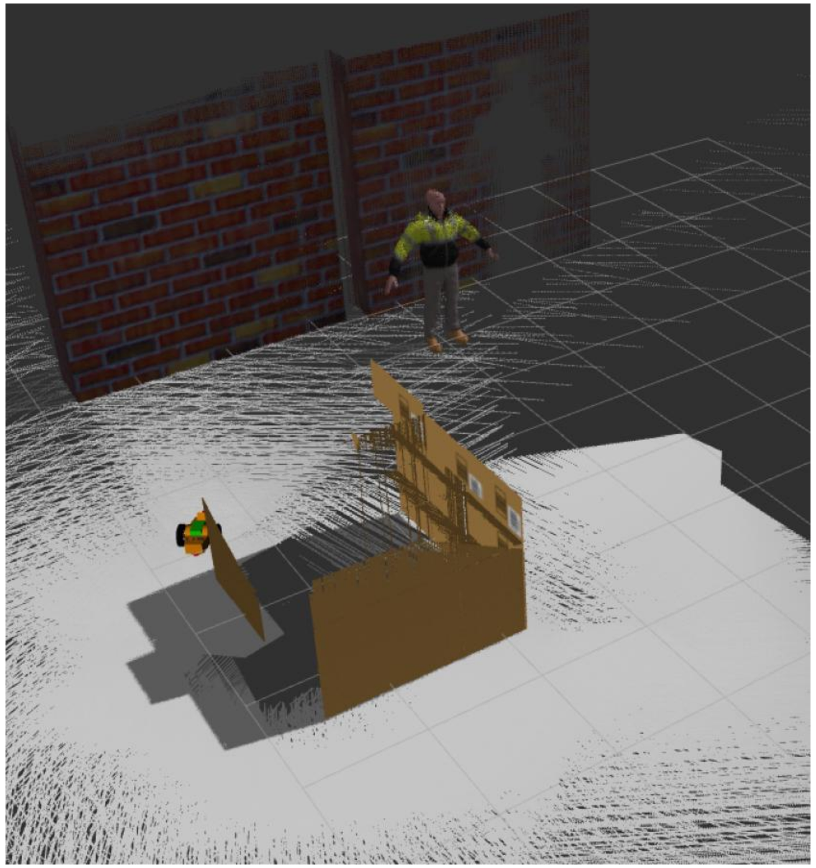
```
"/camera/depth_image@sensor_msgs/msg/Image@gz.msgs.Image",
"/camera/points@sensor_msgs/msg/PointCloud2@gz.msgs.PointCloudPacked",
```

What is the outcome now?

RGBD Camera



Also in this case, we can increase the decay time, to implement a sort of rough mapping approach



Camera sensors

We have seen how we could use imu or gps for localization. But how can I use cameras?

In the following, we are going to use a library for classical computer vision: OpenCV, to understand how we can integrate it in ROS: https://github.com/CarmineD8/my_opencv

- ✓ OpenCV: Open Source Computer Vision & Machine Learning software library
  - Created in 1999 by Intel.
  - Supported from 2008 by WillowGarage (a SME dedicated to hardware & open source software for personal robotics applications, e.g. ROS).
- ✓ Cross-platform Windows | Linux | Android | Mac OS | iOS …
- ✓ Free under BSD License Commercial & non-commercial applications.

OpenCV Modules

■ core	▶ Base data structures & core routines
■ imgproc	▶ Image processing routines
● Linear / nonlinear image filtering	
● Geometric image transforms	
● Shape descriptors	
● Basic image operators	
● Basic feature detection	
■ video	▶ Video analysis routines
● Motion estimation	
● Motion segmentation	
● Background subtraction	
● Object tracking	

OpenCV Modules

- calib3D ► Basic multiple-view geometry algorithms
  - Single/stereo camera calibration
  - Object pose estimation
  - Stereo correspondence
  - 3D reconstruction
- features2D ► 2D image features routines
  - Feature detectors
  - Descriptor extractors
  - Descriptor matchers
  - Object categorization
- objdetect ► object detection routines
  - Detection of objects and instances of predefined classes

OpenCV – ROS

- ✓ How could we use opencv with ROS?

Let’s see how to write a simple subscriber:

```
#include "cv_bridge/cv_bridge.hpp"
#include "image_transport/image_transport.hpp"
#include "opencv2/highgui.hpp"
#include "rclcpp/logging.hpp"
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/image.hpp"
```

These headers will allow us to subscribe to image messages, display images using OpenCV's simple GUI capabilities, and log errors.

OpenCV – ROS

```
#include "cv_bridge/cv_bridge.hpp"
#include "image_transport/image_transport.hpp"
#include "opencv2/highgui.hpp"
#include "rclcpp/logging.hpp"
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/image.hpp"
```

cv_bridge is a **ROS package** that provides an interface between **ROS image messages** (sensor_msgs/Image) and **OpenCV images** (cv::Mat in C++ or numpy.ndarray in Python).

In other words, it lets you easily **convert images from ROS topics to OpenCV format**, and vice versa

image_transport is a **ROS package** that makes it easier and more efficient to **publish and subscribe to images** in ROS.
It’s built on top of the regular ROS publisher/subscriber system (sensor_msgs/Image), but it adds **support for multiple transport options**, such as compression

OpenCV – ROS

```
void imageCallback(const sensor_msgs::msg::Image::ConstSharedPtr & msg)
{
  try {
    cv::Mat img = cv_bridge::toCvShare(msg, "bgr8")->image;
    cv::imshow("Received Image", img);
    cv::waitKey(1);
  } catch (cv_bridge::Exception & e) {
    RCLCPP_ERROR(rclcpp::get_logger("subscriber"), "cv_bridge exception: %s", e.what());
    return;
  }
}
```

The imshow function displays the image provided as the second argument.

cv_bridge is a ROS library that converts between ROS image messages (sensor_msgs::Image) and OpenCV image formats (cv::Mat).

toCvShare() returns a shared pointer to a CvImage object, which has a member called image of type cv::Mat.
image is the actual OpenCV image matrix you can process or display.

OpenCV – ROS

```
int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("image_subscriber");
  auto subscription = node->create_subscription<sensor_msgs::msg::Image>(
    "/camera/image", 10, imageCallback);
  RCLCPP_INFO(rclcpp::get_logger("subscriber"), "Subscribed to /camera/image topic");
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

Here we are subscribing to the “raw” topic, without using image_transport functionalities (i.e., the capability of using compressed images)

OpenCV – ROS

In the CMakeLists, to use OpenCV and related libraries, we need to add the following dependencies:

```
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(cv_bridge REQUIRED)
find_package(OpenCV REQUIRED)
find_package(image_transport REQUIRED)

add_executable(image_subscriber
src/subscriber.cpp)

ament_target_dependencies(image_subscriber
rclcpp sensor_msgs cv_bridge OpenCV
image_transport)

install(TARGETS
  image_subscriber
  DESTINATION lib/${PROJECT_NAME}
)
```

Sensor_msgs is the package where the message used for images is described

Image_transport is needed only if we are actually going to use compressed images, otherwise we could remove the header from our file (and consequently, the dependency in our CMakeLists)

OpenCV – ROS

We have now a node that subscribe to a ROS image and transform it in a CV:Mat Image. But as mentioned, we are not using the image_transport package properly. Let’s do that!

Now let's start up a new subscriber, this one using compressed transport. The key is that image_transport subscribers check the parameter _image_transport for the name of a transport to use in place of "raw". Let's set this parameter:

```
ros2 run my_opencv it_subscriber --ros-args -p image_transport:=compressed
```

All other compressed codecs could be used.

OpenCV – ROS

We have now a node that subscribe to a ROS image and transform it in a CV:Mat Image. But as mentioned, we are not using the image_transport package properly. Let’s do that!

We need just to change the main function:

```
int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::NodeOptions options;
  rclcpp::Node::SharedPtr node = rclcpp::Node::make_shared("image_listener", options);
  node->declare_parameter<std::string>("image_transport", "raw");
  cv::namedWindow("view");
  cv::startWindowThread();
  image_transport::ImageTransport it(node);
  image_transport::TransportHints hints(node.get());
  image_transport::Subscriber sub = it.subscribe("camera/image", 1, imageCallback, &hints);
  rclcpp::spin(node);
  cv::destroyWindow("view");

  return 0;
}
```

NodeOptions lets you configure a node at creation, e.g., for parameters, namespaces

Declares a ROS 2 parameter called "image_transport" with default value "raw".This allows you to later change the transport type (e.g., "compressed") at runtime or via a launch file.

TransportHints let the subscriber choose which transport to use (raw, compressed, etc.), possibly reading from parameters.

OpenCV – ROS

By now, we are able to subscribe to ROS images (either compressed or not) and transform them into CV:mat images. That's ok, but what we can do with these images?

At first, we could try to slightly modify them, and republish them on another ROS topic!

That's not complex. We can just add a publisher (an image_transport publisher!)

```
image_transport::Publisher pub;
```

In the callback, we modified the image with cv functions, and after that we transform it again in a ROS message:

```
auto out_msg = cv_bridge::CvImage(msg->header, "bgr8", img).toImageMsg();
```

ready to be published on a ROS topic.

OpenCV – ROS

Great! Could we do the same in python?

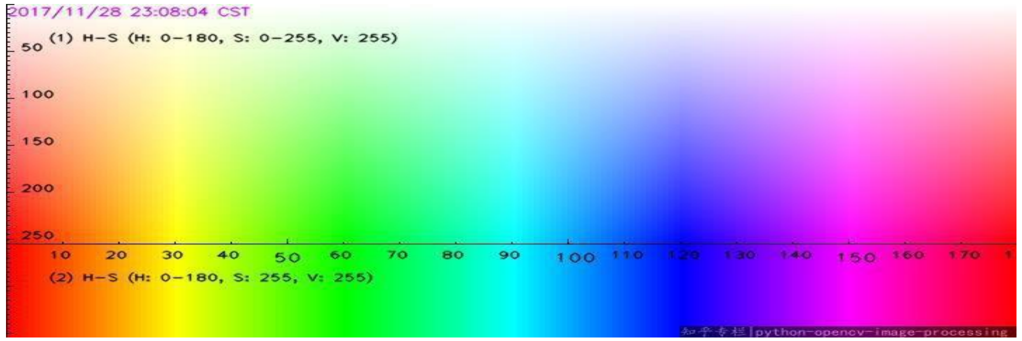
Check the package: https://github.com/CarmineD8/my_opencv_py

- How you can notice (modifier.py), python does not have the image_transport module.
- To use the compressed images, you should directly subscribe (or publish) to them
- Another difference is that you can directly publish the compressed image, without the need of publishing also the raw one. In python, there are some opencv function that are able to transform the image from a compressed one, to a OpenCv one.

OpenCV – ROS

Ok, but any practical application?

- Object tracker: example_ball.py



- GaussianBlur -> We blur the frame to reduce high frequency noise and focus on the structural objects inside the frame , such as the ball.
- HSV -> We convert the image in the HSV color space (**Hue Saturation Brightness)**
- Mask -> We supply the lower HSV color boundaries for the color *yellow*, followed by the upper HSV boundaries. The output of cv2.inRange is a binary mask. A series of erosions and dilations remove any small blobs that may be left on the mask.
- Contours and Center

Image-Based Visual Servoing

The error is computed directly on the values of the features extracted on the 2D image plane, without going through a 3D reconstruction and the robot should move so as to bring the current image features (what it “sees” with the camera) to their desired values

How to get the object's position in space?

- Camera_calibration package

- The function estimates the following parameters of a monocular camera from several views of a calibration pattern. The geometry of this pattern is usually known (i.e. it can be a chessboard):

    - ➢ The linear intrinsic parameters: the focal lengths in terms of pixels (these are basically scale factors), the principal point which would be ideally in the center of the image, and sometimes a skew coefficient between the x and the y axis (but this is often zero).
    - ➢ The non-linear intrinsic parameters: the previously mentioned parameters are forming the linear camera matrix, but there are also some non-linear parameters in the transformation from the 3D camera to the 2D image plane, i.e. the lens distortion.
    - ➢ The extrinsic parameters: the transformation matrix between the 3D world and 3D camera coordinate systems.

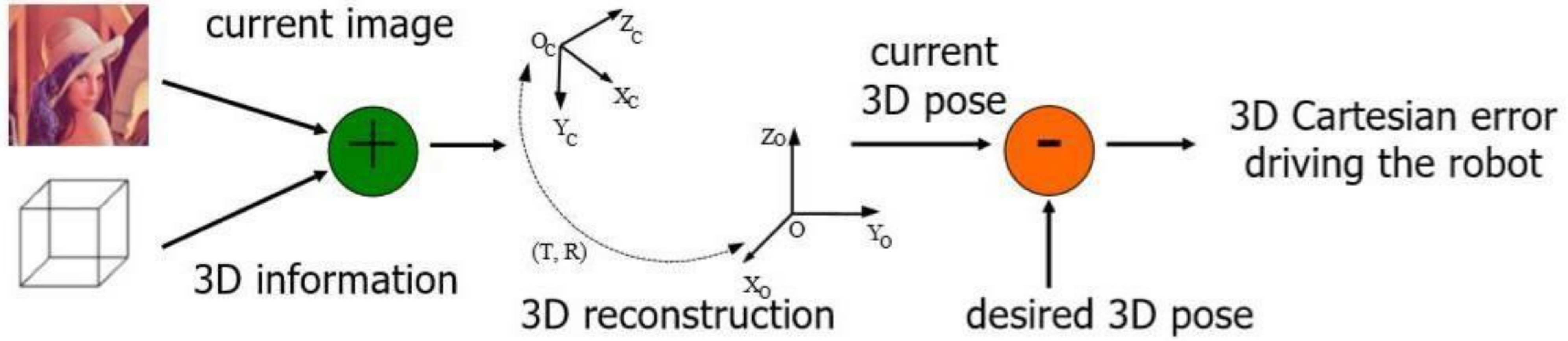
Position-Based Visual Servoing

Once you have the intrinsics of a camera, you can assume that these will never change (except you change the optics or zooming). On the other hand the extrinsics can be changed, i.e. you can rotate the camera or put it to another location. You should see that the scenario of changing an object's pose to the camera is very similar in this case. And this is what the *cv::solvePnP(...)* is used for.
The function estimates the object pose given:

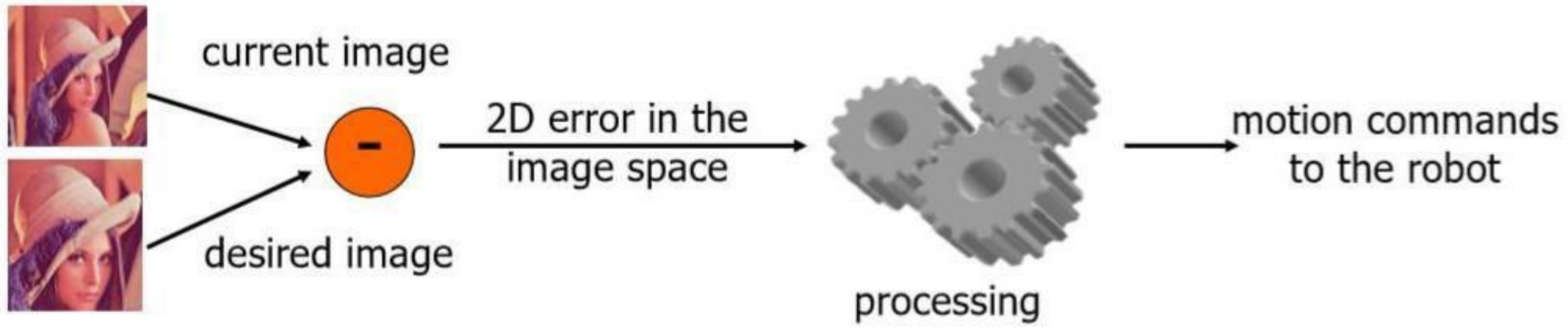
- A set of 3D object points in a model coordinate system,
- Their 2D projections on the image plane,
- The linear and non-linear intrinsic parameters.

The output of cv::solvePnP(...) is given as a rotation vector (rvec) together with a translation vector (tvec) that bring the 3D object points from the model coordinate system to the 3D camera coordinate syste

- position-based visual servoing (**PBVS**)

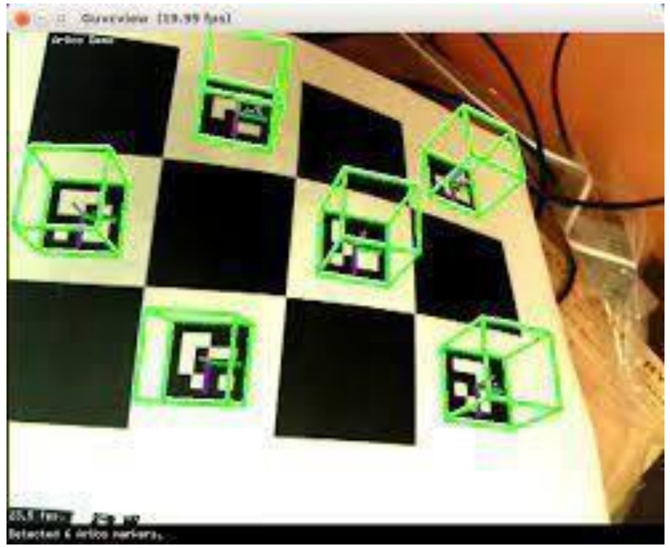


- image-based visual servoing (**IBVS**)



ArUco

- ArUco is an OpenSource library for detecting squared fiducial markers in images. Additionally, if the camera is calibrated, you can estimate the pose of the camera with respect to the markers.
- The library is written both in C++ and python.



ArUco

Let’s see the most simple example for Marker detection:

```
#include "aruco.h"
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
 int main(int argc,char **argv){
   if (argc != 2 ){ std::cerr<<"Usage: inimage"<<std::endl;return -1;}
   cv::Mat image=cv::imread(argv[1]);
   aruco::MarkerDetector MDetector;
   //detect
   std::vector<aruco::Marker> markers=MDetector.detect(image);
   //print info to console
   for(size_t i=0;i<markers.size();i++){
       std::cout<<markers[i]<<std::endl;
      //draw in the image
       markers[i].draw(image);
   }
   cv::imshow("image",image);
   cv::waitKey(0);
}
```

In the above example, you can see that the task of detecting the markers is implemented by the class MarkerDetector. It is prepared to detect markers of any of the Dictionaries allowed. By default, the MarkerDetector will look for squares and then will analyze the binary code inside.

For the code extracted, it will compare against all the markers in all the available dictionaries. So, if you explicitly indicate the dictionary you are using, you avoid undesired errors and confusions with other Dictionaries. Thus, before calling detect, you should call:

```
MDetector.setDictionary("ARUCO_MIP_36h12");
```

ArUco

Let's see the most simple example for Marker detection:

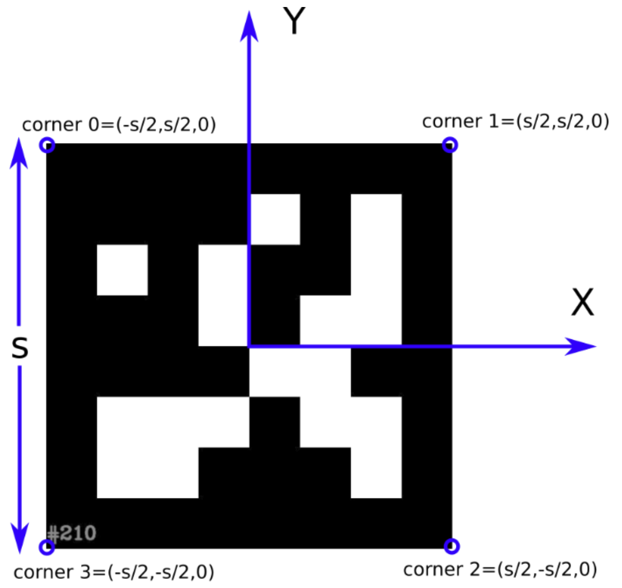
```
#include "aruco.h"
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
 int main(int argc,char **argv){
   if (argc != 2 ){ std::cerr<<"Usage: inimage"<<std::endl;return -1;}
   cv::Mat image=cv::imread(argv[1]);
   aruco::MarkerDetector MDetector;
   //detect
   std::vector<aruco::Marker> markers=MDetector.detect(image);
   //print info to console
   for(size_t i=0;i<markers.size();i++){
       std::cout<<markers[i]<<std::endl;
      //draw in the image
       markers[i].draw(image);
   }
   cv::imshow("image",image);
   cv::waitKey(0);
}
```

ArUco uses the class Marker, representing a marker observed in the image. Each marker is a vector of 4 points (representing the corners in the image), a unique id, its size (in meters), and the translation and rotation that relates the center of the marker and the camera location.

Markers

Markers are composed by an external black border and an inner region that encodes a binary pattern. The binary pattern is unique and identifies each marker. Depending on the dictionary, there are markers with more or fewer bits. The more bits, the more words in the dictionary, and the smaller the chance of confusion. However, more bits means that more resolution is required for correct detection.

Markers can be used as 3D landmarks for camera pose estimation. We denote s the size of the marker once it is printed on a piece of paper. The following image shows the coordinate system employed in the library.



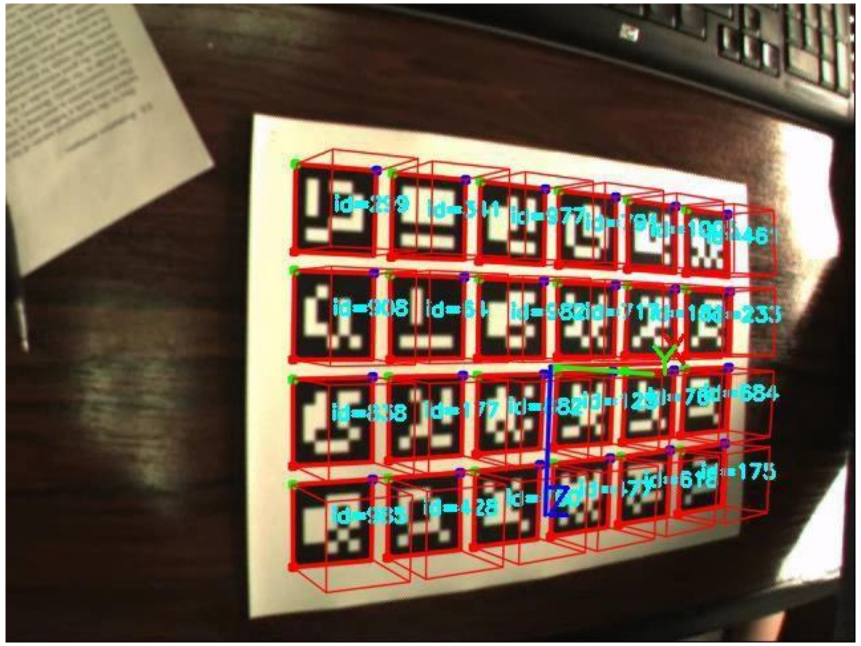
s= marker size

Marker Maps

Detecting a single marker might fail for different reasons such as poor lighting conditions, fast camera movement, occlusions, etc.

To overcome that problem, ArUco allows the use of maps of markers. Marker map is composed by several markers in known locations.

Marker maps present two main advantages. First, since there have more than one markers, it is less likely to lose them all at the same time. Second, the more markers are detected, the more points are available for computing the camera extrinsics. As a consequence, the accuracy obtained increases.



Practical implementation with ROS

- Two main packages:

ROS2-gazebo-aruco: https://github.com/SaxionMechatronics/ros2-gazebo-aruco

can be used for generating markers to be placed in the gazebo environment.

Once created a box (create_marker_tile_image.py in the src folder), you can create a folder called gazebo_models, and add the path in your .bashrc file

Es: export GZ_SIM_RESOURCE_PATH=/home/ubuntu/gazebo_models

This will give you the possibility to add boxes with markers in your simulation, by adding the corresponding model to the gazebo_models directory

Suggestion: always use DICT_ORIGINAL_ARUCO!

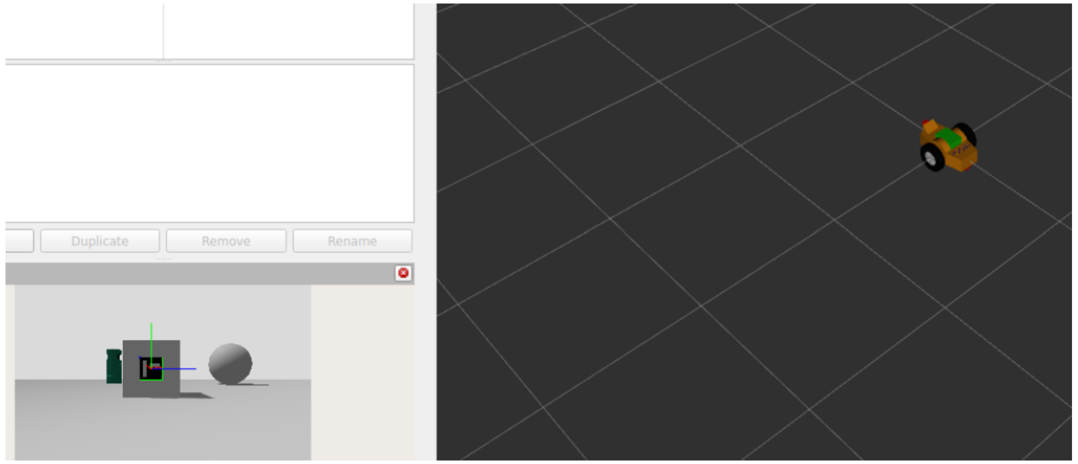
Practical implementation with ROS

- Two main packages:

https://github.com/fictionlab/ros_aruco_opencv for actual marker detection.

Please notice that you have to adapt the config file:
-> adapt it to the correct topic (camera/image)
-> adapt it to the correct dictionary (ARUCO_ORIGINAL)

You can see that the robot is able to find the marker in the environment, and also to estimate its relative distance


Duplicate
Remove
Rename