Worksheet 5 – Images on ROS Network

Vision is a crucial sensorial input that allows the robot to perform several common tasks. In this worksheet we'll see how image data should be represented, published and subscribed over the ROS network, and how ROS nodes can be further extended with a third-party library for computer vision.

OpenCV and ROS Image Types

Digital images are represented in OpenCV library as ***cv::Mat*** data structures, while the ROS framework uses the ***sensor_msgs::Image*** data type to define them. Therefore, when developing a ROS node that handles and processes images using OpenCV functions, only cv::Mat images can be used. When we need to transmit our processed images over the ROS network to other nodes, we need to convert them from ***cv::Mat*** to ***sensor_msgs::Image*** data type. These data types are not interchangeable (i.e., cannot be casted automatically to each other's data type). Therefore, there is a specific package, **cv_bridge**, in the ROS distribution that exposes the **cv_bridge::CvImage** class. This class converts to and from these data types and is defined as follows:

```
namespace cv_bridge
{
  class CvImage
  {
    public:
      std_msgs::Header header;
      std::string encoding;
      cv::Mat image;
   };

   typedef boost::shared_ptr<CvImage> CvImagePtr;
   typedef boost::shared_ptr<CvImage const> CvImageConstPtr;
}
```

As you can see, we have at our disposal the class attribute "*image*" of the same data type used in the OpenCV library (cv::Mat). The typical procedure to convert from ROS to OpenCV images is the following:

```
#include <cv_bridge/cv_bridge.h>


sensor_msgs::Image rosImage;

cv_bridge::CvImagePtr cvImagePtr;
```

```
try
{
   // toCvCopy method copies the image data and returns a mutable CvImage
   cvImagePtr = cv_bridge::toCvCopy( rosImage );
   // toCvShare method shares the image data, returning a const CvImage
   // cv_bridge::CvImageConstPtr cvImagePtr = cv_bridge::toCvShare( rosImage );
} catch (cv_bridge::Exception &e) {
   ROS_ERROR( "cv_bridge exception: %s", e.what() );
}

// shallow copy to a new cv::Mat
cv::Mat opencv_image = cvImagePtr->image;
// or make a hard copy
cv::Mat opencv_image = cvImagePtr->image.clone();
```

Using the resulting cv_bridge::CvImagePtr, we can use cvImagePtr->image to get an object of type cv::Mat, containing our image data. Similarly, to convert from OpenCV to ROS images:

```
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <image_transport/image_transport.h>

cv::Mat opencv_image;
sensor_msgs::Image ros_image;

// … some code here

cv_bridge::CvImage out_msg;
out_msg.header.stamp = ros::Time::now() ; // timestamp from the system clock
out_msg.header.frame_id  = "/camera_link";     // camera position in the world
out_msg.encoding = sensor_msgs::image_encodings::BGR8; // or other types
out_msg.image     = opencv_image; // your cv::Mat
out_msg.toImageMsg( ros_image );
// now we can publish the image…
ros_img_pub.publish(ros_image);  // image_transport::Publisher
```

NOTE *- For more detailed information about the usage of* ***cv_bridge*** *package please read the following page: http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages.*

If we use a different computer vision library or a different class than cv::Mat for representing an image, then we need to manually construct the sensor_msgs::Image by using the following ROS function:

```
sensor_msgs::Image ros_image;
sensor_msgs::fillImage( ros_image, std::string encoding, int rows, int cols, int step, void* ptrData );
```

where the encoding parameter can be "bgr8", "mono8", etc. The step is defined as image's columns times the number of channels.

Publishing and Subscribing to ROS Images

Transmitting uncompressed (raw) images over a network can be quite expensive on data bandwidth. Data compression techniques and carefully selected rates of transmission are transport strategies that have to be considered when we are limited to low network bandwidth as in small robotic systems.

We already know that to publish or subscribe to data over a ROS network we use the common and generic **ros::Publisher** and **ros::Subscriber** classes, respectively. However in order to transmit image data (sensor_msgs::Image) while coupling with the requirements of image compression, the **image_transport::ImageTransport** class provided by the **image_transport** package is used instead.

Therefore the student is advised to avoid the use of ros::Publisher to publish images as follows:

```
// ---------- AVOID THIS APPROACH ----------
#include <ros/ros.h>

void imageClbk (const sensor_msgs::ImageConstPtr& msg)
{
  // Some code here
}

ros::NodeHandle nh;
ros::Subscriber sub = nh.subscribe("in_image", 1, imageClbk);
ros::Publisher pub = nh.advertise<sensor_msgs::Image>("out_image", 1);
```

Use the *image_transport* classes instead:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>

void imageClbk (const sensor_msgs::ImageConstPtr& msg)
{
  // Some code here
}

ros::NodeHandle nh;
image_transport::ImageTransport it(nh);
image_transport::Subscriber sub = it.subscribe("in_img", 1, imageClbk);
image_transport::Publisher pub = it.advertise("out_img", 1);
```

To use **cv_bridge** and **image_transport** packages as described, we need to include the respective header files in our code and write in the CMakeLists.txt file that our package depends on the **cv_bridge** and **image_transport** package:

```
find_package( catkin REQUIRED COMPONENTS roscpp cv_bridge image_transport <etc> …)
```

The next exercises will be accomplished by using computer vision routines provided by OpenCV library. Luckily for us the latter is already distributed with ROS framework and, thus, there is no need to compile and install this library from its source files.

Create a video stream from a camera

Let's imagine that we want to extend a ROS-enabled robot, equipped with a vision camera, with the capability to search and identify a particular object placed on robot's environment. Before starting to develop a ROS node for object identification, we need to first create a more generic and useful node to the robot's ROS ecosystem: the camera driver that creates a live stream by publishing the camera's frames (images) over the ROS network. That's what we are going to do as our first exercise.

Interfacing with video devices

ROS nodes can interface with several USB cameras by using the OpenCV's VideoCapture class. This class provides C++ API for capturing video from cameras or for reading video files. For example, the following code shows a very simple (non-ROS) C++ program to open and read the frame from a camera:

```
#include "opencv2/opencv.hpp"

int main( int argc, char** argv )
{
   cv::VideoCapture cap(<device_index>);  // open the camera with <device_index>
   if ( !cap.isOpened() )        // check if we succeeded
      return -1;

   cv::Mat frame;        // create just an empty image without any
                         // information regarding its type and dimensions
   for( ; ; )
   {
      cap >> frame;      // get a new frame from camera
      if ( !frame.data )  // check if a new frame was successfully retrieve
         break;          // this can fail if the camera is not available anymore
                         // or, in the case of a video file, we have reached the end of file.
      cv::imshow( "camera", frame );   // opens a named window to display the image
      cv::waitKey(1);                  // waits 1 millisecond – useful to allow the display
                                       // refresh from cv::imshow. It can also capture
                                       // pressed keys.
  }
}
```

where <device_index> is the identification number of the camera device. You can define commonly as "0" to open your integrated webcam (if your computer has one) or other integer number if you have more

than one USB camera connected. Instead of a device index, we can pass a std::string with the full path to a video file that we wish to open and read.

Alternatively we can create the **cv::VideoCapture** class without any arguments and, later, we can call its open method:

```
bool VideoCapture::open( const std::string& filename )
bool VideoCapture::open( int device )
```

where **filename** is the name of the opened video file or image sequence (eg. img_%.jpg, which will read samples like img_00.jpg, img_01.jpg, img_02.jpg, ...). The **device** parameter is the identification number of the opened video capturing device (i.e. a camera index). Note that these methods first call **VideoCapture::release()** to close an already opened file or camera.

NOTE – For more detailed information regarding the API provided by OpenCV library, please take a look at: http://docs.opencv.org/. Further reading material with examples can be also found at the following bibliography: OpenCV 2 Computer Vision Application Programming Cookbook, Robert Laganière, May 2011.

Exercise

Now let's get to work:

- Create a new package named as "t5_package" in your ROS workspace.
- Create a ROS node named "camera_driver" and apply what you have learned about the OpenCV library in order to interface with your camera. Don't forget to use the image_transport::Publisher class to transmit the live frames over the ROS network. These images must be published under the topic "/camera/image_raw".

To check if your camera_driver node is working as expected, we must open a new terminal and execute the following ROS node:

```
rosrun image_view image_view image:=/camera/image_raw
```

We should see now what **camera_driver** is publishing on topic "/camera/image_raw". Finally, to finish this exercise the student must do the following:

- In the same package, create the ROS node "**image_reader**" that must behave like the **image_view** node. It must subscribe to topic "/camera/image_raw" and pop out a window to display the published images using the OpenCV's cv::imshow() and cv::waitKey() functions.

Camera Calibration

Several cheap cameras introduce image artifacts such as radial distortion that can hamper the effectiveness of computer vision algorithms. Commonly, these cameras are first calibrated to minimize the effects of distortion. The calibration process is based on the knowledge of a regular pattern's dimensions, such as a chessboard, that needs to be captured by the camera at several angles and positions. Then, this pattern's aspect must be evaluated and (omitting the mathematics behind it) we can obtain a calibration matrix that rectifies the image in order to obtain the correct aspect of the pattern.

Proposed ROS Architecture

Let's begin by learning how ROS framework can ease our life when we need to calibrate a vision camera. Check the following scheme (Fig.1):

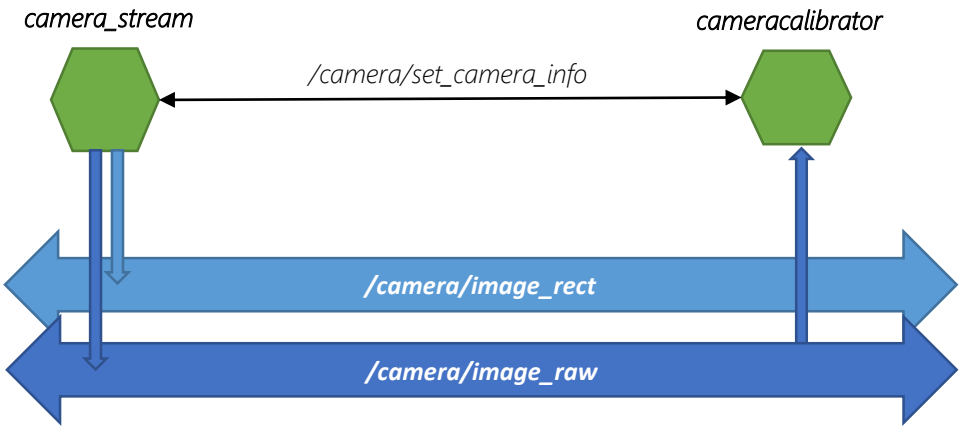


Fig. 1 – Proposed ROS ecosystem for the camera calibration procedure.

The camera_stream node publishes frames incoming from our robot's camera at topic *"/camera/image_raw"*. These are the distorted frames. Then, we have the cameracalibrator node that subscribes to the previous topic and checks for a chessboard pattern on the image stream. This node is already distributed in the ROS framework.

To safely achieve a good calibration, the human operator needs to move the chessboard up, down, left, right, to and from the camera and also skew it. The cameracalibrator indicates the progress of these movements at its graphical user interface (see Fig. 2).

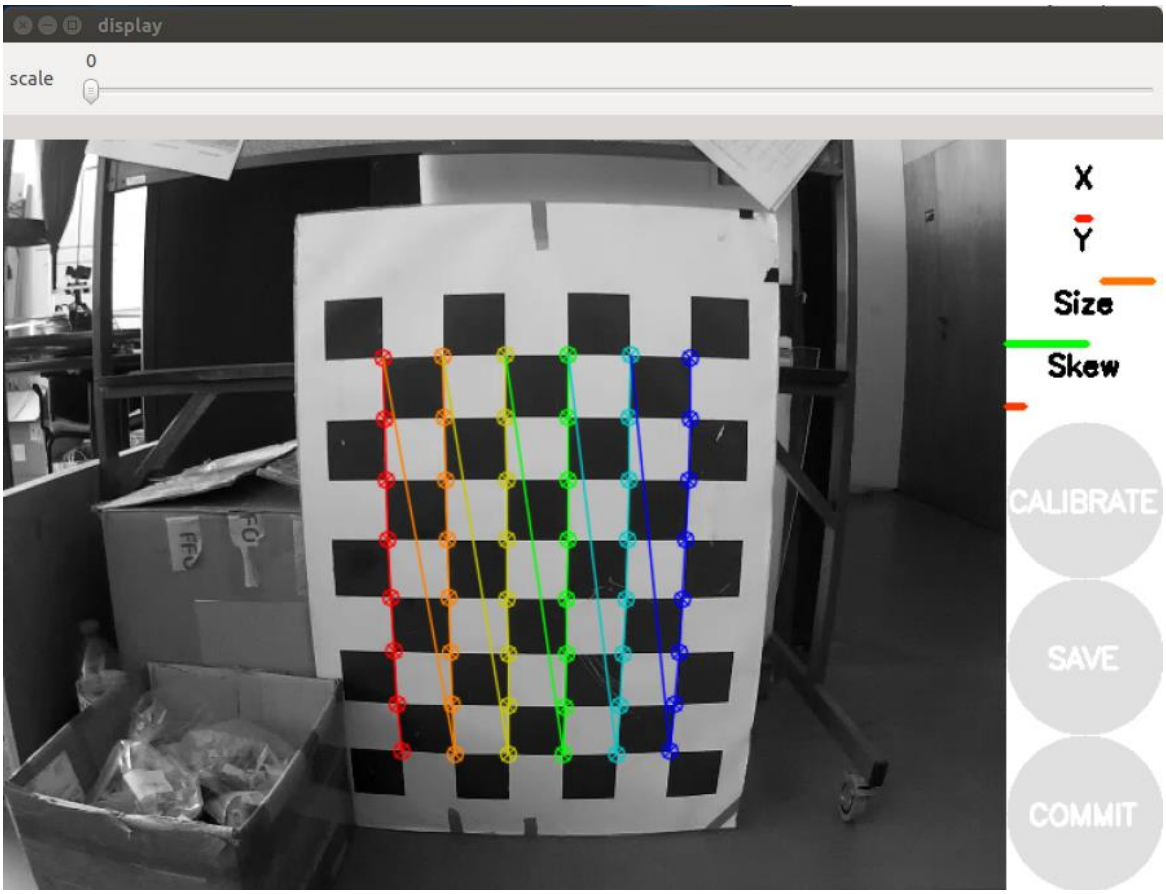
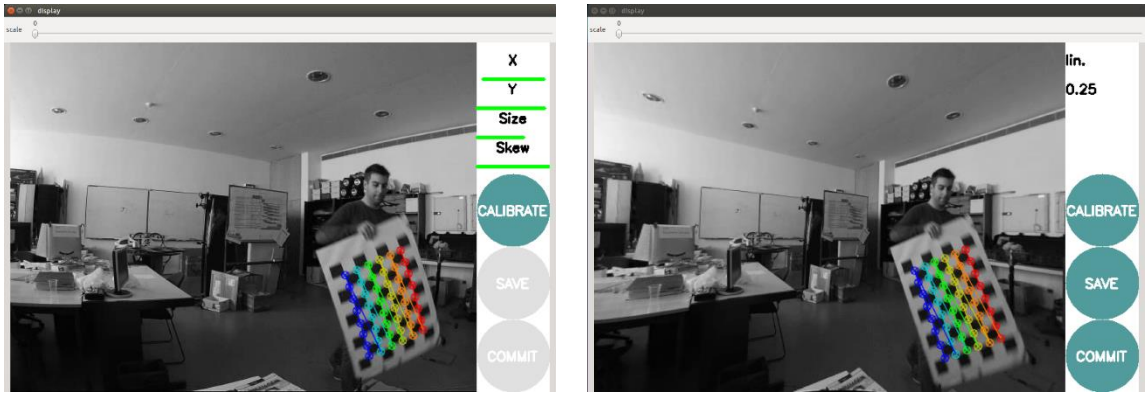


Fig. 2 – Graphical user interface of **cameracalibrator** node. At the upper right corner of this image we can see several progress bars that indicate us what chessboard's positions have been presented or not to the camera's field of view. For instance, the progress bar for the camera's X axis shows a small red bar at its center. This means that the **cameracalibrator** detected the chessboard at the center of the image, but it's still requires that the chessboard must be presented at the left regions and also right regions of the image. While the chessboard is moving to these regions, the X's progress bar increases its size and changes its color from red to green. Please note that "Size" refers to the Z axis, which commonly points outwards the camera. Therefore, the distance from chessboard to the camera must also vary. Finally "Skew" relates to changes in pitch, yaw and roll angles.

When a satisfactory amount of data has been gathered by cameracalibrator, it'll notify the user by enabling the CALIBRATE button on the GUI. After clicking CALIBRATE, we have to wait (a few minutes on the worst case scenario) while the calculations are performed to obtain the calibration matrix. Then, the COMMIT button is activated (see Fig. 3) and it must be clicked for this information to be sent back to the camera_stream node in a *CameraInfo* format. For the interested readers, check the following webpage: http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html).

In particular, the cameracalibrator calls a service at topic *"/<camera_name>/set_camera_info"* that in our case is provided by the camera_stream node. The latter node has now all the required information to start rectifying (undistort) the images and publishing them at a new topic: *"/camera/image_rect"*.



(a) (b)

Fig. 3 – Calibration procedure at different stages: a) Green colored progress bars are almost filled. The CALIBRATE button is now active for the user to click on it and begin the computation of the calibration matrix. b) The calibration is finished (after a few minutes!) and both SAVE and COMMIT buttons are now active. The SAVE button is used to create and save to the disk a YAML file that contains the calibration matrix and parameters. The COMMIT button can be used instead to call a ROS service server on topic "/<camera_name>/set_camera_info". This service accepts the calibration matrix in the ROS CameraInfo format.

Now let's check the actual implementation of this scheme for our practical class.

Hands-on-Approach

Due to the limited number of chessboards patterns and cameras available for the students, the camera_stream node was programmed to fake the streaming of a real camera that needs to be calibrated. To accomplish this behavior, several snapshots of the chessboard pattern were already taken, at different angles and positions, by the real camera. These images are published by this node at topic *"/camera/image_raw"*, just like it was streaming them from the real camera. The cameracalibrator node will use these images to generate the calibration matrix and send it to the camera_stream node as described previously.

At this point, the camera_stream node knows how to calibrate its camera. Therefore, in our case a recorded video from the same real camera is then opened by the camera_stream node, which publishes the original frames with distortion (at topic *"/camera/image_raw"*) and the rectified ones (at topic *"/camera/image_rect"*).

Both calibration images and test video are included in the package of this worksheet. In this exercise the OpenCV library is used to load the calibration images from the disk and also to open and read frames from the test video when the time is right. Please note that camera_stream node automatically switches from publishing the calibration images to publishing the frames from the test video when the node finds that there is a calibration matrix already saved in the "t5_package" (from a previous calibration) or when the "/camera/*set_camera_info*" service is called.

Let's test for ourselves the calibration procedure. Launch the roscore process and open a new terminal to execute the cameracalibrator node:

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.0745 camera:=/camera image:=/camera/image_raw
```

The "size" parameter represents the structure of the chessboard's corners (see Fig. 4 a)). The size of a chessboard square (in meters) is specified in the "square" parameter and the camera name is set with the "camera" parameter. This name will used as a prefix for the "/set_camera_info" service topic required by the cameracalibrator node. Finally, the "image" parameter defines the image topic at which this node subscribes to.

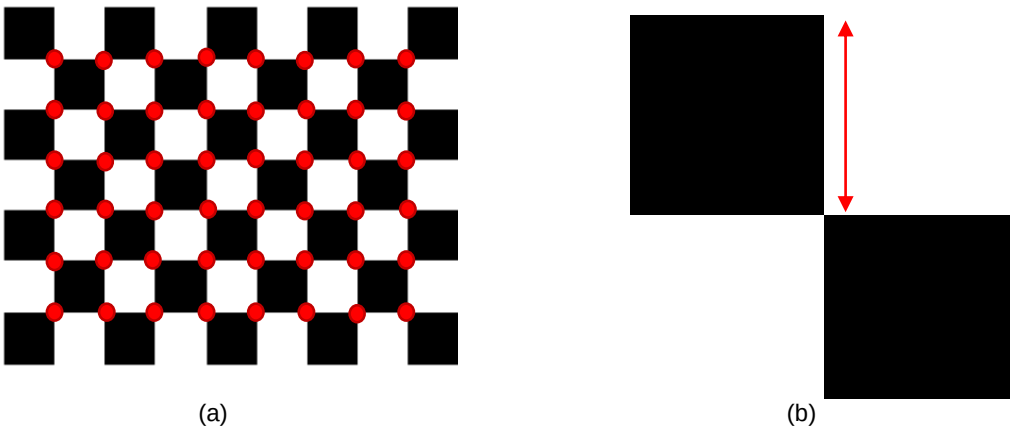


Fig. 4 – Example of a chessboard pattern used in the calibration procedure. a) Pattern's effective size definition highlighted by an overlay of red corners: 8x6. b) Size of the chessboard's square in meters (red arrow).

Open now another terminal and launch the camera_stream node:

```
rosrun t5_package camera_stream
```

We'll see a fast streaming of the calibration images that are processed by the cameracalibrator. When possible click on the CALIBRATE button to begin the calculations.

While we wait for this process to finish, let's execute the ROS Visualization Tool (RVIZ) as depicted in Fig. 5. This process is the main tool to see what's going on over the ROS network. It provides a set of plugins to parse several data types in order to display the published data for the user.

Open a new terminal and launch RVIZ process:

```
rosrun rviz rviz
```

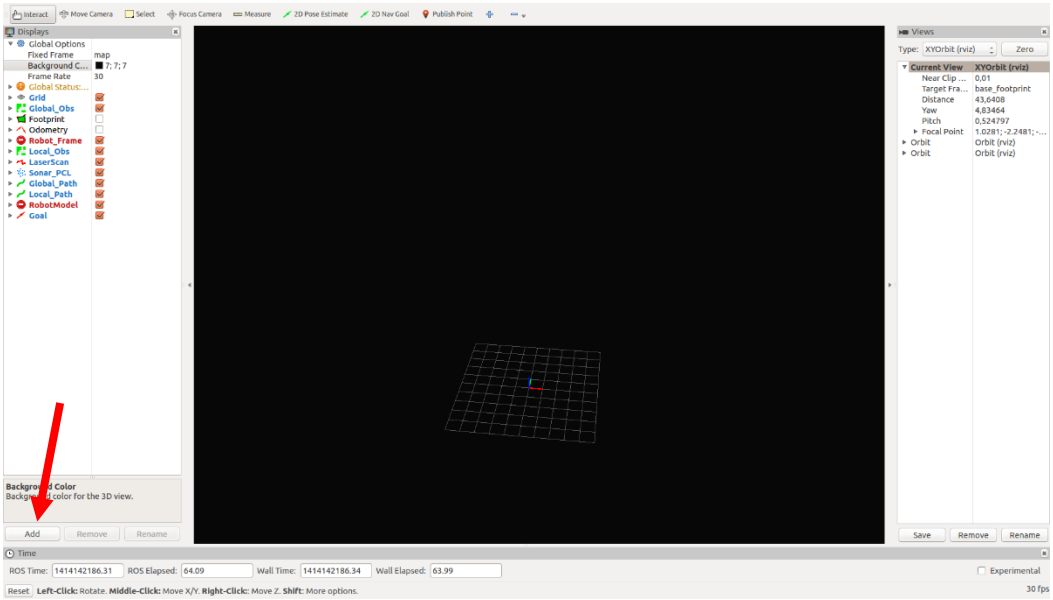


Fig. 5 – ROS Visualization Tool (RVIZ).

Now in RVIZ's graphical user interface add two windows for ROS images, as depicted in Fig. 6 (a). Then you must set the topic to which you want to subscribe to see what contains on sensor_msgs::Image messages (see Fig. 6 (b)). In our case, the image topics are: "/camera/image_raw" and "/camera/Image_rect". At this moment, we are ready to simultaneously display both distorted and rectified images in the RVIZ.

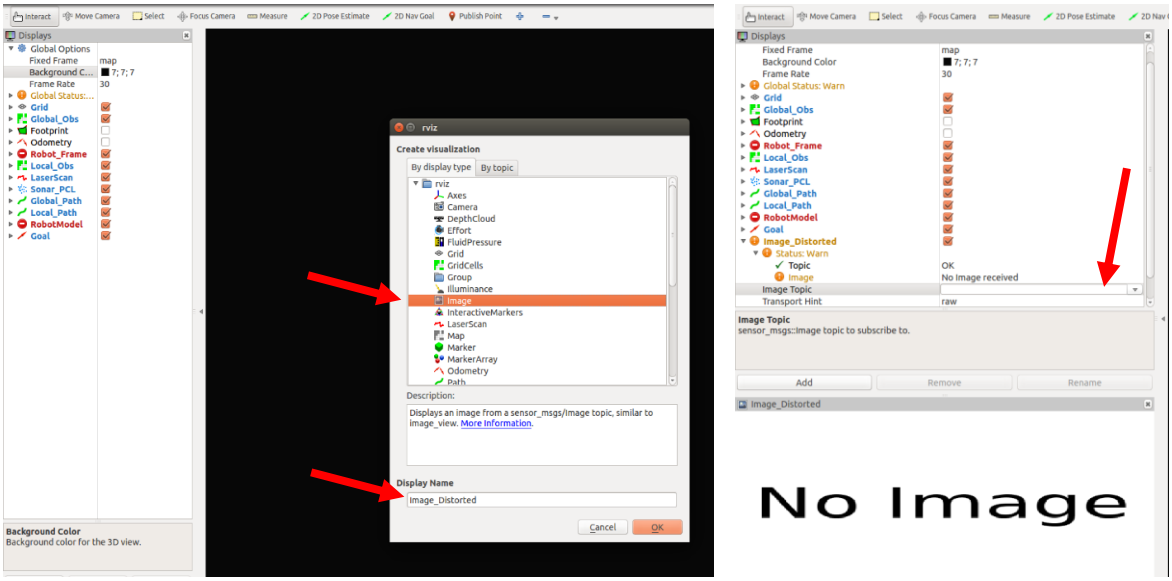


Fig. 6 – Adding a new display window (a) and setting the topic parameter (b) in RVIZ.

Let's go back now to cameracalibrator node. When the calculations are finished we will see something similar to what is depicted at Fig. 3 b)). Note that we can define the scale of the calibration at the scale's slide bar (see Fig. 7). Just leave to the default value 0.

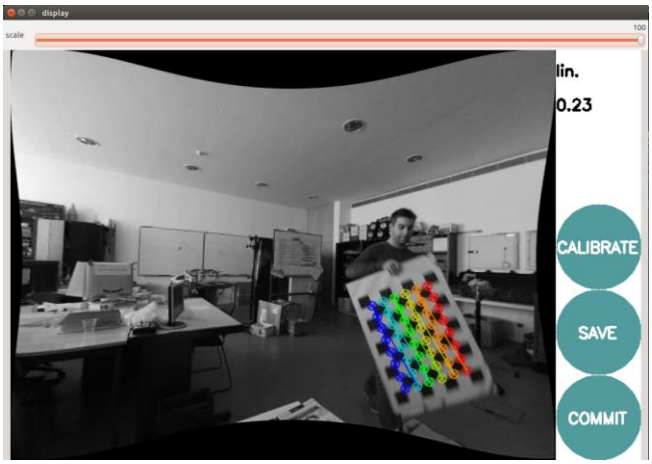


Fig. 7 – Effect of changing the scale of the calibration matrix. At scale 100 we can see clearly the transformation applied to the distorted image. However, at all sides of the image there are black pixels. This is undesired to post image processing. Hence, if we set the scale at value 0 the image is cropped to remove these artifacts (black regions) and zoomed in to comply with the resolution at which it was calibrated.

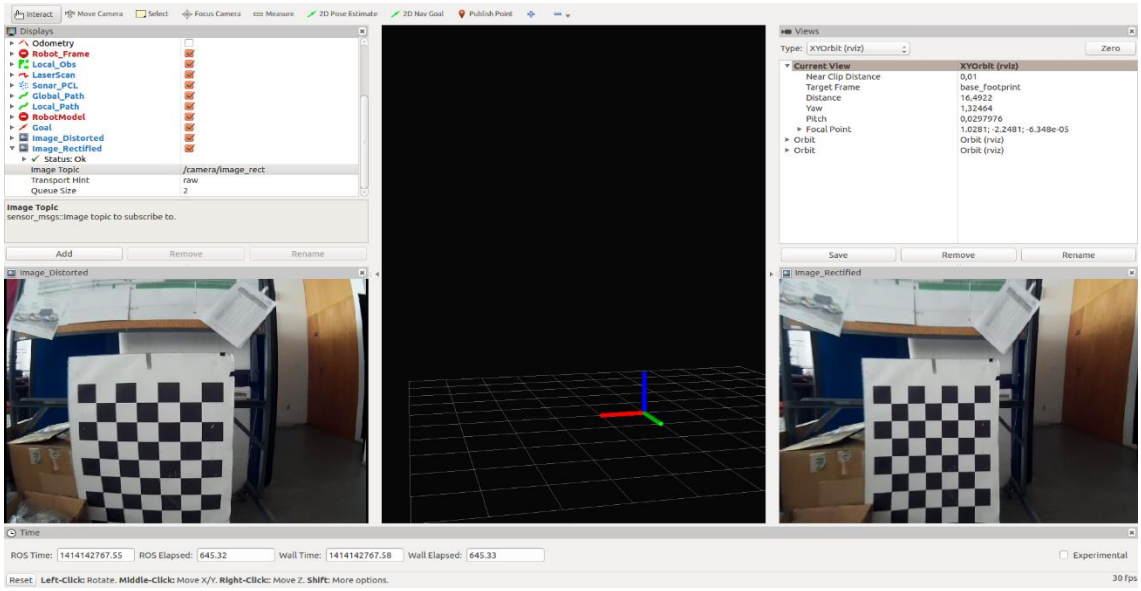
Click now on the COMMIT button to call the service at topic "/camera/set_camera_info", and switch to RVIZ to view the streaming of the test video. The student will be able to check the differences of distorted *versus* rectified frames, as depicted in Fig. 8. If you miss any footage, you can replay the video by re-running the camera_stream again from the terminal. This node will skip the calibration process because it detects that it already has the calibration matrix (.yaml file) to calibrate the camera and, thus, publishes right away the recorded test video.

Final Exercise

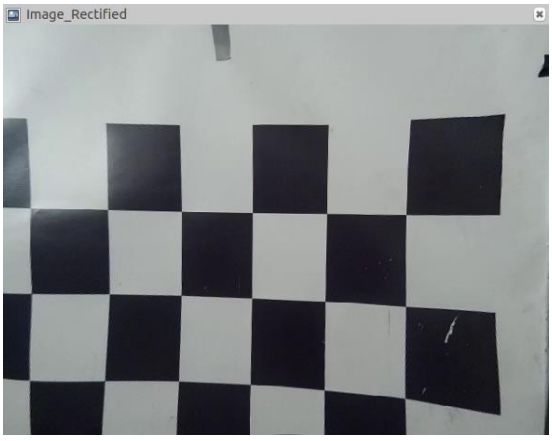
1. Create an "image_convert" ROS node in "t5_package" that subscribes to image topic "/camera/image_raw". This node must convert images from RGB color space to grayscale (see cv::cvtColor() function on OpenCV API documentation).Then it must smooth the grayscale image by applying gaussian filter with a 21x21 kernel and zero standard deviation in x and y axis (search for cv::GaussianBlur() function). Finally, the processed image must be published on topic "/camera/image_processed". Note: All operations must be performed on 8-bit images. You can use the camera_driver node done in this practical class that publishes the frames from your web camera to this topic.

2. Change now the camera_reader node to also subscribe to topic "/camera/image_processed".

The resulting "t5_package" package from this exercise must be zipped into a file named as "T5_<NUMBER1>_<NUMBER2>_<NUMBER3>.zip", where NUMBER is your student's number (i.e., T5_44500_43261_46312.zip). This file must be submitted at the Moodle platform no later than October 20th, 23:55.



(a)



(b) (c)

Fig. 8 – Displays for both distorted and rectified images in RVIZ (a). Distorted image (b). After the calibration, the result was a nice rectification in the center region of the image, as depicted in (c). However, at the edges of the image there is an "over-rectification" (still another distortion). More angles and positions of the chessboard pattern are required in order to achieve better results.