## Tasks and results

### 1. Basic

For C++, the EdgeDetector.hpp and EdgeDetector.cpp files should be filled in. If any libraries are required, add them to the CMakeLists.txt. For Python, the edge\_detector.py file should be filled in. Instructions must also be provided to download and install any required Python packages.

It should be possible to provide different images as input to your code and show the output as the image with the edges detected as green lines. Also, provide a Readme.md file detailing instructions on installation, implementation steps, concepts used, and possible improvements.

\*\*Result\*\*

Edge detection is detecting rapid changes of pixel intensity in a locality.   
The implemented edge detection solutions are

1. Laplacian edge detector:

This detector identifies edges as the ‘Zero-Crossing points’ in the second derivative (2-D Laplacian) of the Image function

2. Canny edge detector

This detector uses a gradient operator (Sobel filter) to compute the gradient at every pixel and then applies a 1-D laplacian along the gradient orientation, as computed from the result of applying the gradient operator previously. If we get a strong zero crossing at the end of above operation, we declare it as an edge.

*NOTE: gradient edge detector ignored as it provided a noisy edge detected image when tested in this use case*

We convert the image from RBG space to grayscale as it simplifies the computation for edge detection

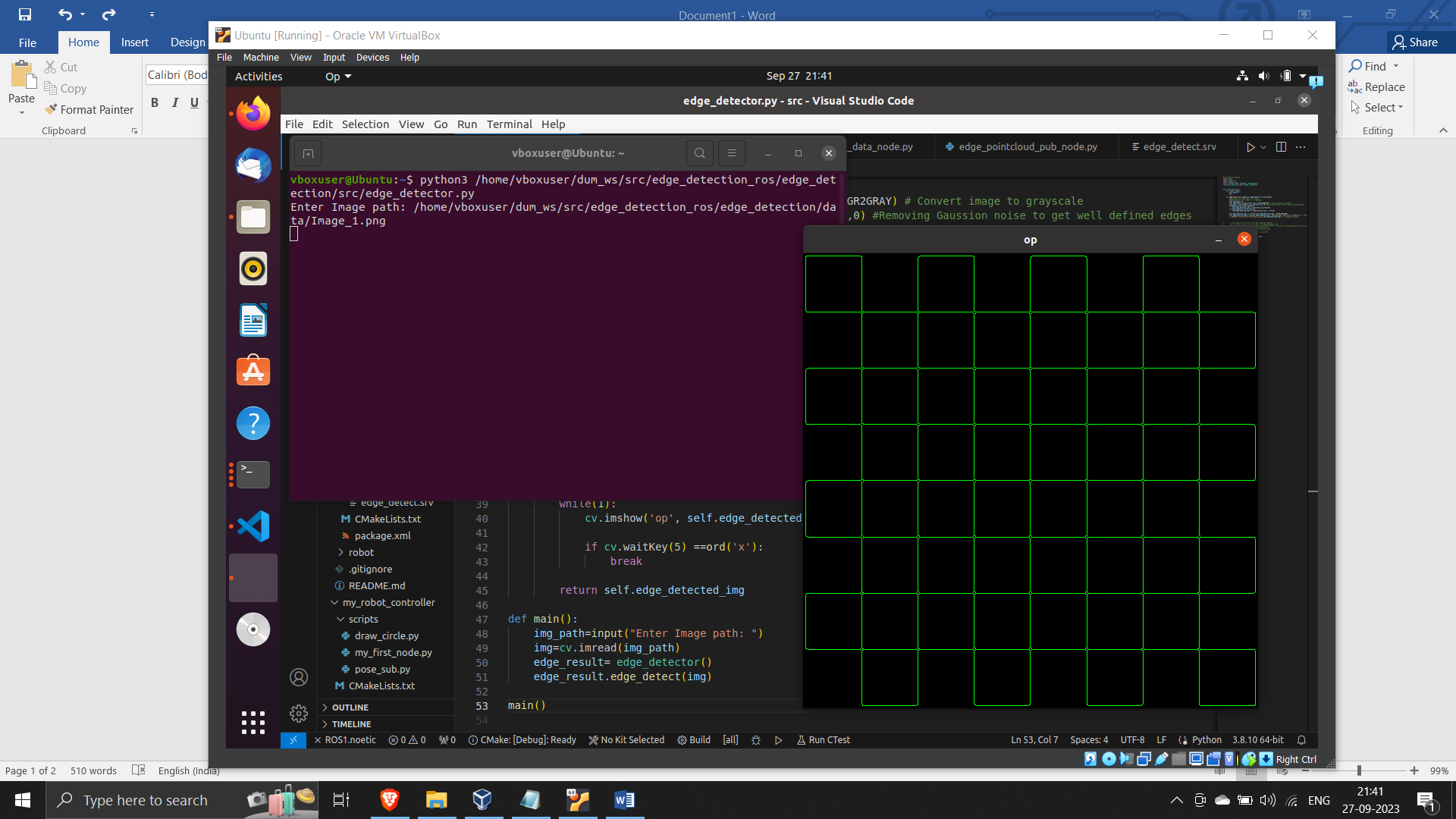
We utilise a Gaussian blur to de-noise the images. Noises are nothing but instantaneous/rapid spikes in our image which would produce false zero crossings when edge detectors are applied, this leads to detecting false edges

**The above is implemented in edge\_detector.py as function edge\_detect,** takes 2 parameter image\_raw 🡪 Image data

edge\_detector\_filter\_type 🡪 0 : Canny edge detector [Default]; 1: Laplacian

I have also written a piece of code which is commented which overlays the result of canny and laplacian over each other to provide the combined result of both algorithms.

Sample O/P:



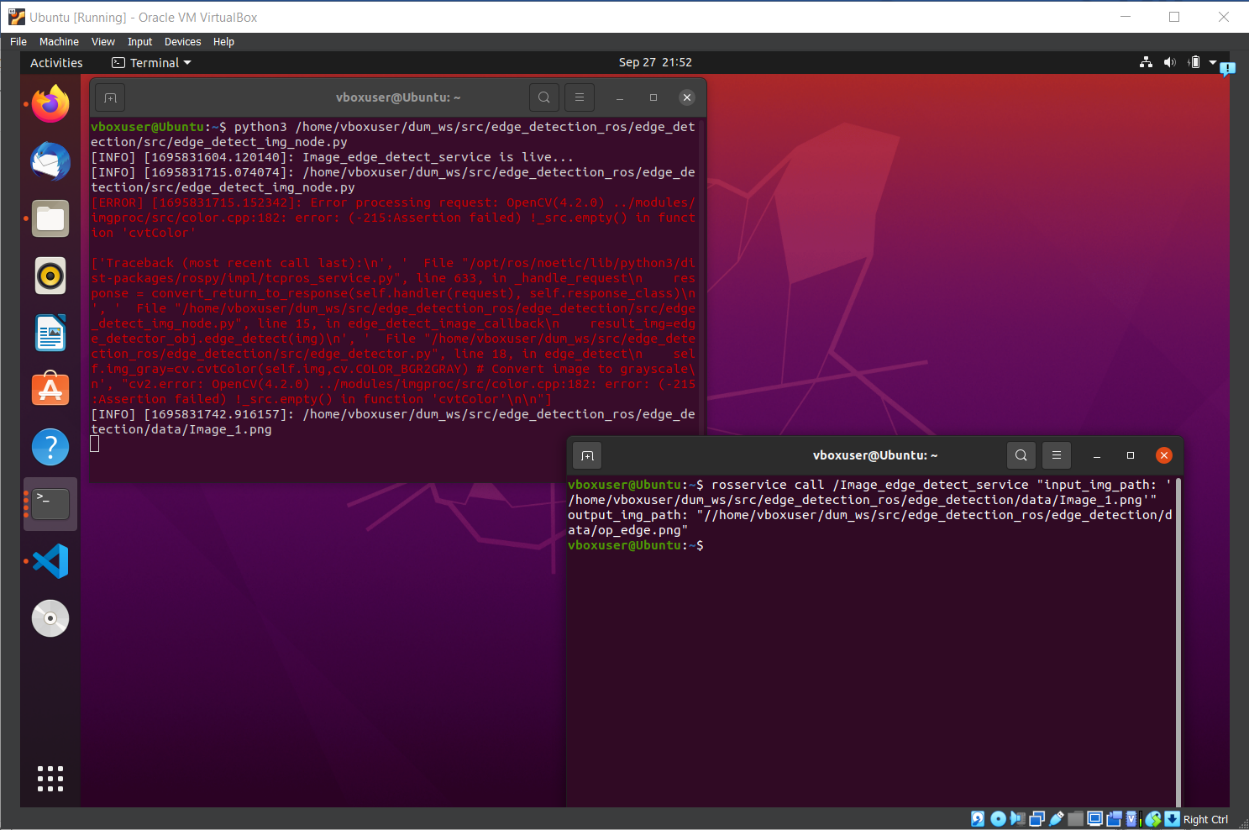
### 2. Vision\_ROS:

Provide ROS .srv and .msg files required to create a ROS service for edge detection. Give example usage of this service with a client to detect edges for image files in a directory.

\*\*Result\*\*

A rosservice was created with takes in the ‘input\_img\_path’ as a string and returns the file path ‘output\_img\_path’, where the edge detected image result is stored.

To use this service, run “../src/edge\_detect\_img\_node.py “ file to initiate the service “Image\_edge\_detect\_service”.

Sample O/P:

Additionally, detect edges for the images subscribed from an image topic that is available when the given ROS bag (.bag) file is played. The input image and detected edges should be visualised on RViz.

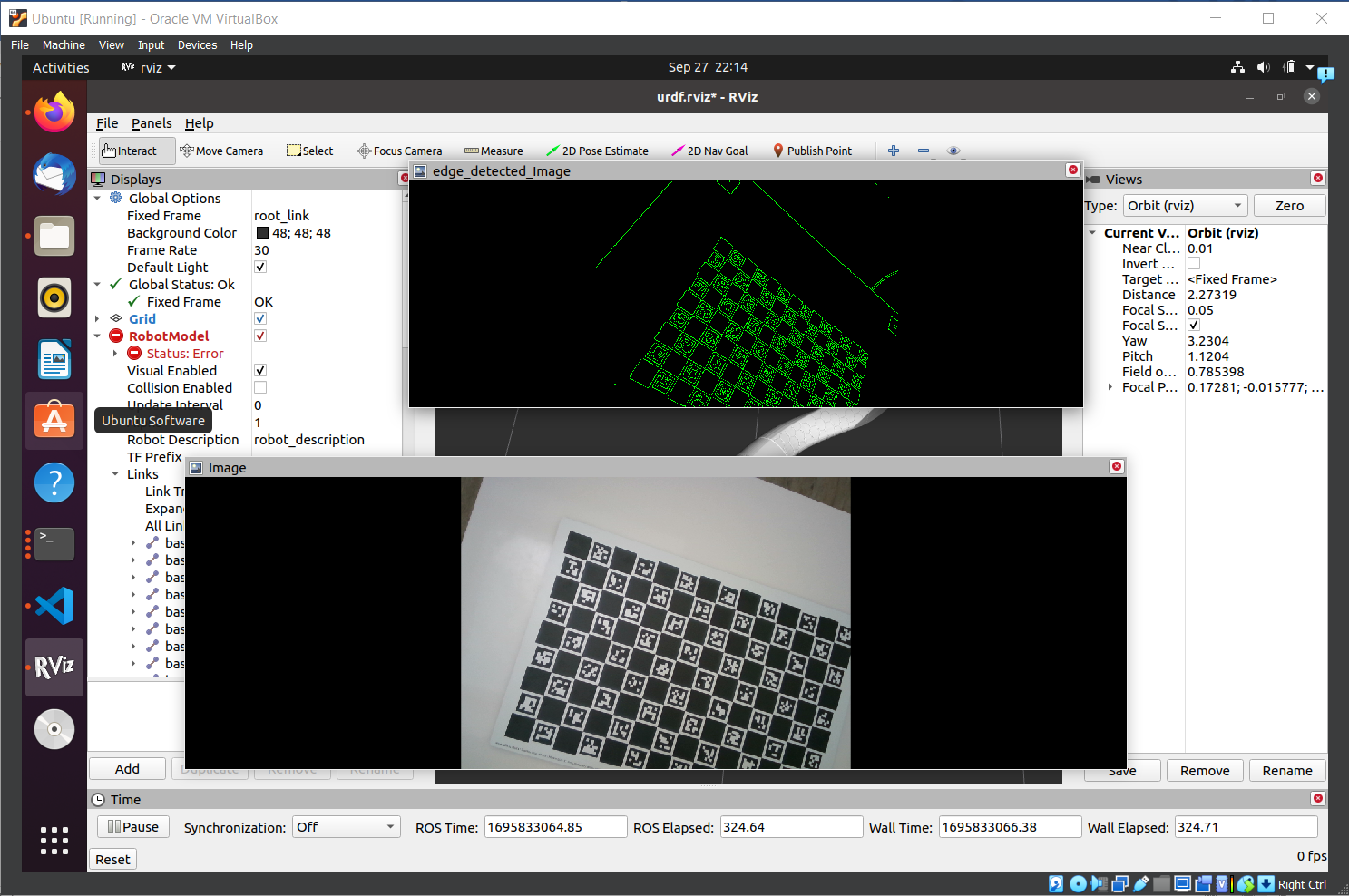
\*\*Result\*\*

In order to accomplish this, we need to ‘image\_raw\_listener’ node that subscribes to the topic *‘/camera/color/image\_raw’* to get sensor\_msgs/Image and publish the edge detected images as sensor\_msgs/Image to the topic *‘edge\_detected\_image’.*

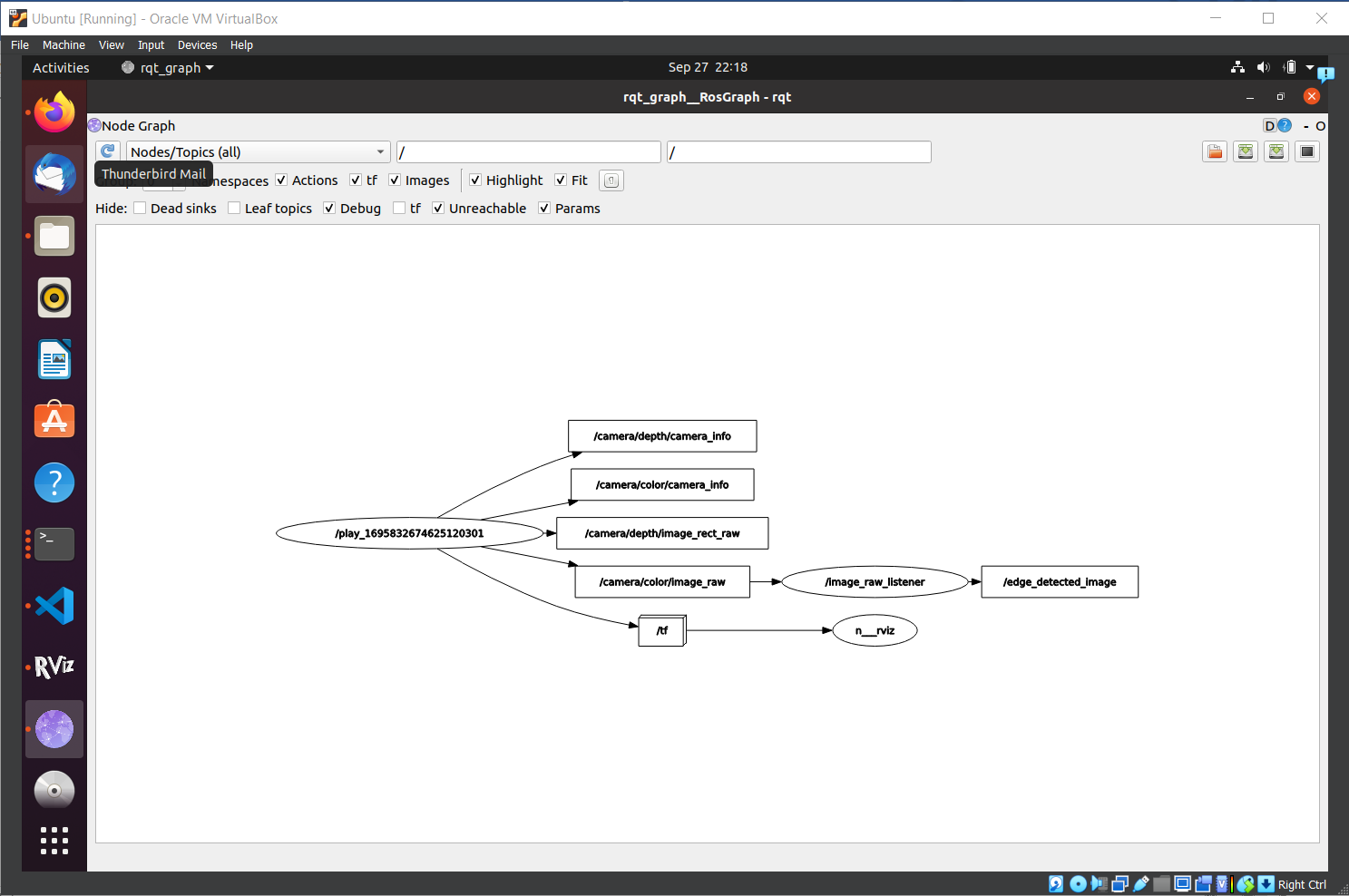
The Image data from sensor\_msgs/Image message is converted to a numpy array (so that function from opencv can be used) and sent to an edge\_detector object to extract and returned edge detected Image back to be published as a sensor\_msgs/Image message.

To perform this,

* run one of the provided bag file. ( *rosbag play --clock -l <path to bagfile> )*
* Then, run ../src/edge\_detect\_bag\_data\_node.py
* Once the topic *‘edge\_detected\_image’* is getting published, go to rviz and add two ‘displays’ which are connected to the topics *’edge\_detected\_image’* and *‘/camera/color/image\_raw’*



rqt\_graph:



Convert the detected edge pixels from pixel coordinates (u, v) to 3D data (x, y, z) using the depth images and camera parameters also available in the .bag file. Publish the 3D data to a ROS topic (suggestion: of type sensor\_msgs/PointCloud), with a topic name edge\_points.

\*\*Result\*\*

From the edge detected 2D image data provide by the topic ‘edge\_detected\_image’ , we can find the 3D coordinates and generate the point cloud with the help of Intrinsic camera matrix (provided by topic ‘/camera/color/camera\_info’) and depth image data (provided by ‘'/camera/depth/image\_rect\_raw'’ ).

The conversion formula is as below

* x=depth\*(u-cx)/fx
* y=depth\*(y-cy)/fy
* z=depth

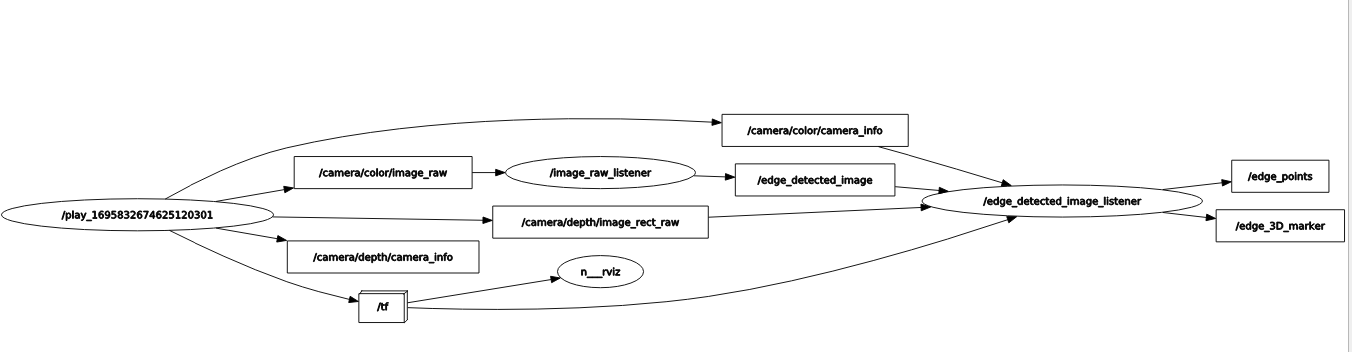
,where (fx, fy) is focal lengths and  (cx, cy) is principal point of the camera, provided by ‘/camera/color/camera\_info’

The 3d-coordinates are then published as pointcloud to the topic ‘edge\_points’ as ‘sensor\_msgs/PointCloud’ type.

To perform this,

* run one of the provided bag file. ( *rosbag play --clock -l <path to bagfile> )*
* Then, run ../src/edge\_detect\_bag\_data\_node.py
* Once the topic *‘edge\_detected\_image’* is getting published, run ../src/edge\_pointcloud\_pub\_node.py
* To verify if the point could is generated run “rostopic echo /edge\_points ” in a new terminal

rqt\_graph:



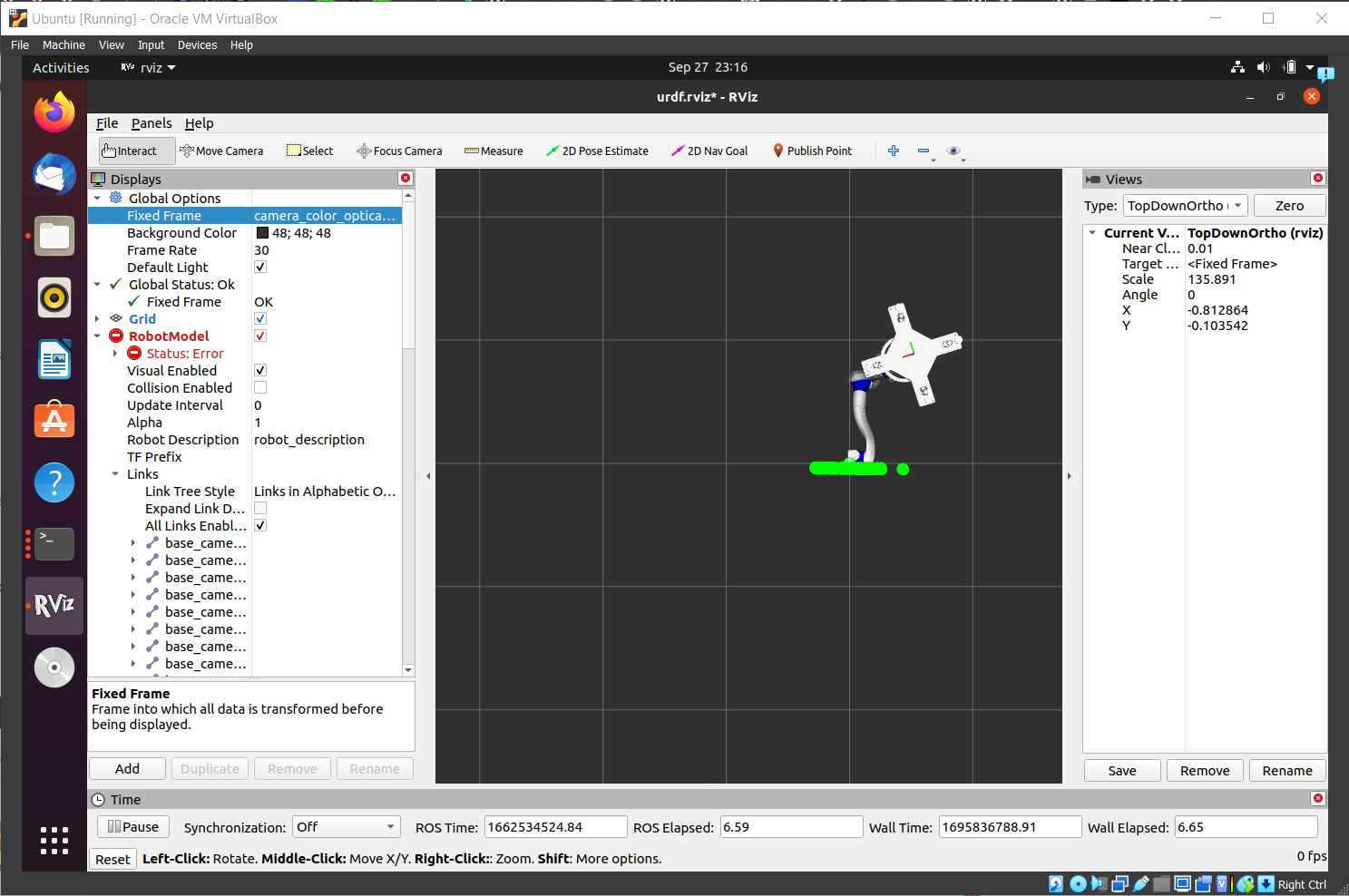
### 3. Robot\_ROS:

Extend the code further by visualizing the 3D edge points for each frame as RViz markers together with the visualization of a robot.

\*\*Result\*\*  
The rviz markers, with the help of the 3D coordinates generated previously, are being sent to the frame with id ‘camera\_color\_optical\_frame’. This is because the image data provide by the ‘/camera/..’ topics are with the above mentioned frame\_id. The makers are published to the topic ‘edge\_3D\_marker’ as ‘visualization\_msgs/Marker’ messages.

To view the markers

* run one of the provided bag file. ( *rosbag play --clock -l <path to bagfile> )*
* Then, run ../src/edge\_detect\_bag\_data\_node.py
* Once the topic *‘edge\_detected\_image’* is getting published, run ../src/edge\_pointcloud\_pub\_node.py
* Open rviz and add ‘display’ of type ‘Marker’ by the topic ‘/edge\_3D\_marker’



*PS: Apologies as I was unable to record a video of the markers populating as my VM kept crashing when trying to record*

You can use the robot URDF model provided to visualize the robot and multiple frames of reference. Please provide a video of the markers and the robot on Rviz for a duration of one loop of the given .bag file as part of the submission.

\*\*Result\*\*

This is accomplished by using the TransformListener package. By using TransformListener.transformPoint() function we can transfer the 3D coordinate data from ‘camera\_color\_optical\_frame’ to other frames based on the transformation matrix details extracted messages we receive from the /tf and /tf\_static topics.

The part of code that does the above is commented in ../src/edge\_pointcloud\_pub\_node.py as I was not able to test it.

**Room for improvement:**

Though all python scripts can be made into classes for better modularity and ease of execution

The reason for utilising rospy was because I was able to iterate and the test code faster. The logic used can be directly translated to roscpp also