Institute of Architecture of Application Systems (IAAS)

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Research Project

Computer Vision in Robotics

Vinayak Vishwas Patil.
(Matriculation Number: 3507267)

Course of Study: Information Technology, Embedded systems.

Examiner: Prof. Dr. Marco Aiello

Supervisor: Mr. Nasiru Aboki, M.Sc.

Commenced: October 1, 2021

Completed: March 21, 2022

Abstract

Vision in robots or machines plays a vital role. It allows the robot to visually see and interact with its environment. Robot vision expands computer vision approaches to meet the needs of robots and robotic systems. Navigation towards a specific target while avoiding obstacles, finding a person, and reacting to the person's directions or detecting, recognizing, grasping, and delivering objects are all common tasks. Robot vision harnesses the power of visual sensing to observe and perceive the environment and react to it.

The aim is to add more functionalities to the telepresence bot. We are extending the functionalities with the help of built-in sensors and actuators of the robot. We study ORB-SLAM and implement a solution for creating a map of the unknown environment with the help of a Monocular camera. ORB-SLAM has three major threads, that are tracking, local mapping, and loop closing.

We have implemented object detection with the help of YOLOv3 where it can classify 80 different objects based on the COCO dataset. It's a pre-trained model where the structure of the neural network is defined with the weights and biases of the network. It uses 53 convolution layers in its single neural network to process the image frame and gives the result with good accuracy.

Face recognition has been implemented with the help of the dlib python library. It consists of four major processes that are face detection, face landmark estimation, face encoding, and providing the result. In-depth information about the implementation of all work done is discussed in further sections.

Contents

List of Figures

1 Introduction

Robots are expected to become commonplace in the future. Robots must observe and comprehend their surroundings to gain context awareness and respond correctly to interact with the world robustly and safely. In general, we aim to provide robots with minimum knowledge in advance and have them gather and interpret the information they need for new task execution through interaction and online learning. One of the key drivers in the field of artificial cognitive systems development has been to achieve this long-term aim. A service robot, for example, must be able to learn about items and object categories to execute tasks in a human setting. However, being a passive observer of the world will not allow the robots to create useful categories or object representations. They, like humans, should learn about items and their representations by interacting with them.

Numerous companies now provide robotic solutions with qualities that make them ideal for the aforementioned applications. Despite the wide range of robotic platforms available, it is still difficult to modify and swiftly tailor the behavior of an assistant robot from one scenario to the next based on customer requirements. Furthermore, to perform meaningful interactions, the out-of-the-box robot's functionalities frequently need to be supplemented by external services.

The robot “Butler” available at the IAAS department is a Telepresence robot designed by Ohmni Lab. Ohmni Lab is a robotics firm situated in Silicon Valley that focuses on providing businesses with demand-driven robotics solutions and its robotic platform is used for development. It is specifically designed for video conferencing, where you can move around freely and interact with others as you drive the robot through a remote space. Butler already has capabilities that enable demo deployment in a real-world setting as well as the construction of educational applications. However, the specifications of the application we envision require the robot to detect the objects, identify people and create a map of the unknown environment, under partial difficult conditions and with greater accuracy.

For example, to efficiently create a map of the environment the image frame captured from the robot has to go through various steps so that we can estimate the different parameters used for Tracking, Local Mapping, and Loop closing. For object detection and face recognition, the image frame has to be processed to get the desired results. To implement these functionalities, we need an external processing system that improves the capabilities of the Butler robot concerning the tasks of creating the map, object detection, and face recognition. These functionalities require a complex interaction of the robot with our local host system, where all the processing is done and the required commands are feedbacked to the robot by establishing a communication between them.

The main aim of the research project is to use its camera sensor and actuators to add functionalities to the robot. The task of this robot is to provide a virtual presence for people, but this task has not been used in this project. The camera sensor is used to implement Visual SLAM specifically ORB-

SLAM for creating a 3D point cloud map of the environment. These 3D points are mapped into 2D points to get a map of the environment. Concepts and knowledge from Deep learning are used for detecting objects and face recognition. We have implemented these functionalities to help the people in the lab where the robot can be used for assisting them in certain specific tasks.

1.1 Motivation

Visual perception plays an important role in the behavior of many living animals, including humans. Our ability to track objects, that is, to keep an object in the field of view for an extended period using our oculomotor system as well as head and body motions, is one of the most significant components. Humans can do this swiftly and reliably without putting in a lot of effort. As a result, it's only normal to assume that the artificial cognitive systems we're working on will be able to display similar capacities to some level.

State-of-the-art computer vision techniques are now being developed and used in a variety of industries, including not only the industry but also everyday life. Cameras have become an integral part of personal computers, cell phones, and even robots. It's a huge plus if a telepresence robot can support computer vision techniques because there are so many things a robot and a camera can do.

Robot vision necessitates a system-level approach, in which vision is one of the numerous sensory components that collaborate to complete specific tasks. This aspect of the robotic system is known as embodiment, and it is analogous to biological systems in that the properties of the body shape perceptual tasks. A robot system perceives to act and acts to perceive–vision is employed as a means for the robot to act in and interact with the world. As a result, visual processing is not a standalone entity, but rather a component of a larger system. For more than three decades, the vision has been used in robotic applications. Industrial robotics, service robotics, medical robotics, and underwater robotics are just a few examples.

We want to use the telepresence robot sensors and actuators to expand its capabilities and increase its use cases where it will assist the people of the lab in their activities and will be helpful for the society in some or other way.

1.2 Problem Statement

Robots may interact with their surroundings in a variety of ways thanks to visual feedback. Navigation and obstacle avoidance are examples of visual feedback, whereas more complicated examples include interaction with the user and object manipulation. We need vision systems that can deliver enough information, whether they're manipulating an object or interacting with a human. We need systems that can interpret what they "see" using known or self-acquired models: these systems must perceive to act, and act to perceive.

As mentioned in the previous section, this project aims to create a map of the environment with the help of VSLAM, implementing real-time object detection and face recognition. All the above objectives can be achieved with the robot's sensor and actuators. There is no need for an external device except a local host system.

The following are the research questions this project has addressed:

- What is the necessity of having a vision as it relates to robotics?
- How do we connect to the robot's internal operating system and then use an external host system to give and receive commands to operate it?
- How can we use an in-built monocular camera to implement VSLAM?
- What strategy will we use to detect and classify as many objects as possible?
- How can we put real-time face recognition into practice?

1.3 Research Methodology

The research work has used the ORB-SLAM algorithm where a Monocular camera is used to create the environment map. The YOLOv3 method is used to detect objects in real-time. Real-time facial recognition is accomplished using a Python built-in package that employs deep learning methods. All of the algorithms listed above are implemented in the ROS framework.

The following is the strategy used to answer the aforementioned research questions:

- For the Butler Robot, the VSLAM used is ORB-SLAM. It has previously been proven that VSLAM is a solved issue and there is a substantial amount of material and resources available. Similarly, for YOLO (you only look once) and Face recognition, there is extensive literature available. VSLAM, object detection, and face recognition has not been implemented before on the robot and hence it was important to refer to past work to complete the task. It increases the credibility of the study by assisting in the adoption of a more appropriate research approach. It highlights the advantages and disadvantages of the previous study in the same subject.

- It was important to understand the working of the Robot Operating System framework and knowledge about the Docker environment. Ohmni runs on Ohmni OS, which is a bespoke version of Android-x86 developed by OhmniLabs. It's a one-of-a-kind Android-Ubuntu hybrid that combines Android's appealing characteristics, such as a familiar UI and a streamlined boot process, with the power and flexibility of a full Ubuntu system. The Ohmni Developer Edition includes a sophisticated Docker virtualization layer, which ROS developers use to run the ROS framework on top of Ohmni.

- The Butler robot is the main hardware component and it is important to assess the tasks, so they can be implemented on the supported operating system and algorithms on the middleware framework. The next stage was to set up the program and decide the appropriate algorithms, ROS, and software packages to use in the implementation.

- In the real process of development, the essential features are divided into smaller tasks. First the implementation of VSLAM, then using YOLOv3 for object detection, and finally face recognition with the help of machine learning algorithms. Techniques for verification and validation are employed.

2. Background

2.1 Vision System

In recent years, there has been a rising trend toward greater work in robotics and computer vision systems. The International Conference on Computer Vision Systems (ICVS) promotes this by bringing together work that isn't widely known in the computer vision community and, to a lesser extent, in robotics conferences. Human-centered vision systems and vision for human-robot interaction (HRI) are gaining popularity alongside robot vision (Vincze).

Apart from affordability, accuracy, ease of integration, modularity, and flexibility, it has been recognized that there are two main prerequisites for commercialization in real-world scenarios:

(1) For effective dynamic performance, vision and control must be combined. Fast movements are required to justify the commercial application of vision-based control.

(2) Vision must be stable and consistent. Perception must be able to assess the condition of the environment to react to changes and ensure the robot's and its surroundings' safety.

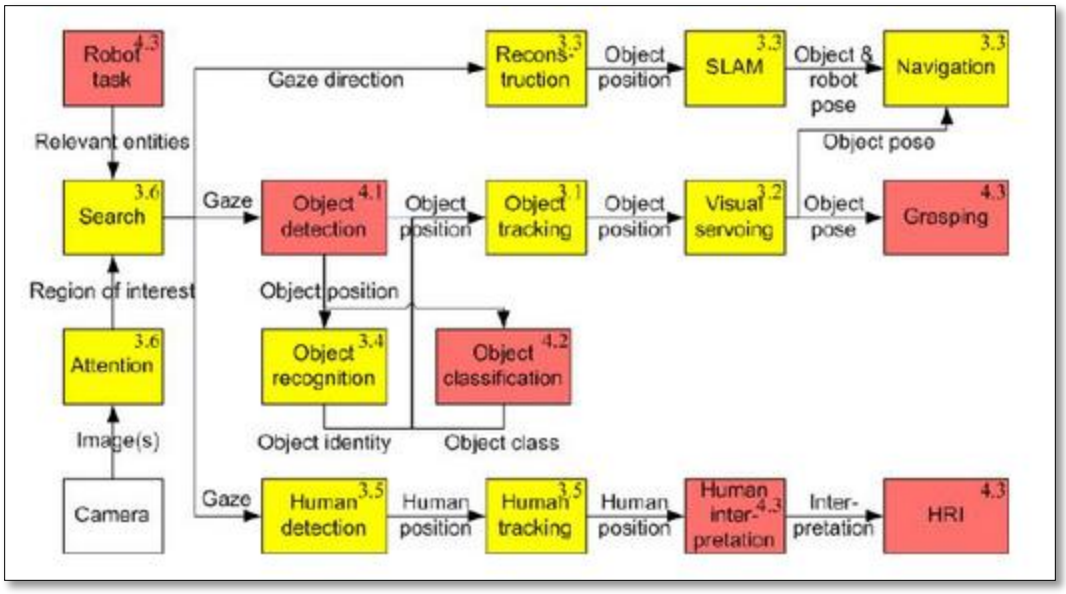

Figure 1 Robot Vision System

Figure 1 depicts a rough model of a robot vision system. At this relatively abstract level of explanation, the overview seeks to show that a robot vision system performs three key functions: navigation, grasping, and Human-Robot Interaction (HRI). The way these functions interact is determined by the job. For example, navigation is now considered a completely solved problem with application-ready approaches and advanced research subjects. The yellow boxes in figure 1 represent that robust performance for these aspects of robot vision has been achieved. The red boxes are primarily concerned with formalizing the semantics of robot tasks and connecting them to grasping and HRI (Vincze).

Both biological and robotic systems must find an appropriate balance between the width of the visual field and its resolution due to limited memory storage and processing capability. The amount of visual data is too vast for the system to process efficiently otherwise. This equilibrium is also determined by the tasks that the systems must complete. A broad field of view is preferable to an animal that must remain vigilant to identify an approaching predator. If the same animal also functions as a predator, the situation is reversed. A broad field of view is also advantageous to a robotic system to avoid colliding with objects when navigating through a congested environment. Although there are systems that illustrate the use of monocular vision for visual serving, binocular setups are used by the majority of robot systems that move around in the environment and interact with people and things. (Vincze)

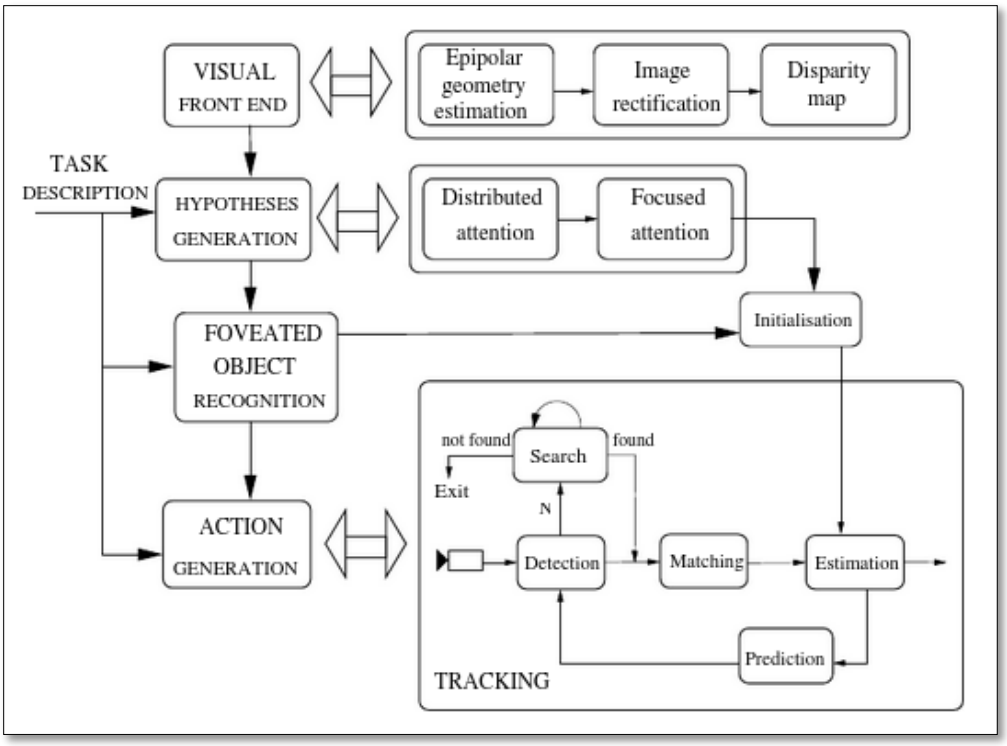

Figure 2 General Description of Robot Vision.

Figure 1 shows a fairly general depiction of a robot vision system.

It is made up of the following modules, as indicated in Figure 2:

•Visual Front-End: extracts visual data for figure-ground segmentation and other higher-level operations.

•Generation of Hypotheses: creates hypotheses about the things in the scene that are important to the work at hand.

•Recognition: determines the relevance of seen items using either picture characteristics or color histograms.

•Action Generation: based on the results of the recognition and the current task specification, initiates actions such as visual tracking and posture estimation. (Vincze)

2.2 Robot Operating System

2.2.1 ROS

The Robot Operating System (ROS or ros) is a robotics middleware suite that is open-source. Although ROS is a collection of software frameworks for robot software development rather than an operating system, it provides services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management for a heterogeneous computer cluster. A graph architecture is used to describe running sets of ROS-based processes, with nodes receiving, posting, and multiplexing sensor data, control, status, planning, actuator, and other communications. Despite the significance of responsiveness and low latency in robot control, ROS is not a real-time operating system in and of itself (RTOS). However, it is feasible to connect ROS with real-time programming. The absence of support for real-time systems was addressed in the development of ROS 2, a significant redesign of the ROS API that will make use of contemporary libraries and technologies for core ROS functionality while also adding support for real-time code and embedded hardware. (source O. , ROS Wiki documentation open source, n.d.) (source O. , ROS Wiki open source)

2.2.2 Design

2.2.2.1 Philosophy

ROS was created with open source in mind, to allow users to customize the tools and libraries that interface with the core of ROS, allowing them to tailor their software stacks to their robot and application area. As a result, aside from the broad structure within which applications must live and interact, there is very little that is genuinely essential to ROS. In some ways, ROS is the underlying infrastructure that allows nodes and messages to communicate. In truth, ROS is more than just plumbing; it's also a robust and mature set of tools, a wide range of robot-agnostic capabilities given by packages, and a larger ecosystem of ROS add-ons. (source O. , ROS Wiki open source)

2.2.2.2 Computation graph model

ROS processes are represented as nodes in a network with edges called topics connecting them. ROS nodes may communicate with one another via topics, make service calls to other nodes, provide a service to other nodes, and set or receive shared data from the parameter server, which is a community database. All of this is made possible by the ROS Master, which registers nodes to itself, sets up node-to-node communication for topics, and controls parameter server changes.

Messages and service calls do not go through the master; instead, when all node processes register with the master, the master sets up peer-to-peer communication between them. (source O. , ROS Wiki open source)

1. Node

A single process executing the ROS graph is represented by a node. Every node has a name that it has to register with the ROS master before it can do anything else. Under separate namespaces, several nodes with different names can exist, or a node can be designated as anonymous, in which case it will create an extra identifier at random to add to its provided name. Because most ROS client code is in the form of a ROS node that performs actions based on information received from other nodes, communicates information to other nodes, or sends and receives requests for actions to and from other nodes, nodes are at the heart of ROS programming.

2. Topics

Buses are the names given to the topics over which nodes transmit and receive messages. Within their namespace, topic names must also be unique. A node must publish to a topic in order to send messages to it, whereas it must subscribe in order to receive messages. No node knows which nodes are transmitting or receiving on a topic under the publish/subscribe architecture; all it knows is that it is sending/receiving on that topic. The sorts of messages passed on a topic can be user-defined and vary considerably. Sensor data, motor control commands, condition information, actuator commands, and anything else can be included in these messages.

3. Services

Services may also be advertised by a node. A service is an action that a node may do that has a single outcome. As a result, rather than processing velocity orders to a wheel motor or odometer data from a wheel encoder, services are frequently utilized for tasks with a clear beginning and conclusion, such as taking a single-frame photograph. Services are advertised and called from nodes to nodes.

2.2.2.3 Tools

The primary capability of ROS is supplemented by a series of tools that enable developers to observe and record data, navigate ROS package hierarchies quickly, and write scripts that automate complicated configuration and setup operations. The integration of these tools significantly improves the capabilities of ROS-based systems by simplifying and solving a variety of typical robotics development issues. Instead of offering implementations of hardware drivers or algorithms for diverse robotic tasks, these packages provide task and robot-agnostic utilities that come with the core of most recent ROS installations.

1. Rviz

Rviz is a three-dimensional visualizer for robots, their working environs, and sensor data.
It's a highly customizable tool with a variety of visuals and plugins to choose from.

2. Catkin

Catkin is comparable to CMake in that it is cross-platform, open-source, and language-independent.

3. Roslaunch

Roslaunch is a command-line program for launching numerous ROS nodes locally and remotely, as well as configuring the ROS parameter server. A complicated starting and setup procedure may be readily automated using roslaunch configuration files, which are written in XML. Other roslaunch scripts may be included in roslaunch scripts, nodes can be launched on specific computers, and processes can be restarted if they fail during execution.

2.2.2.4 Packages

Many open-source implementations of standard robotics functionality and algorithms may be found in ROS. Packages are used to arrange the open-source implementations. Many packages are pre-installed with ROS releases, while others are created by individuals and released through code sharing sites like github. (source O. , ROS Wiki open source)
The following are some noteworthy packages:

1. Systems and tools

- actionlib is a standardised interface for working with preemptable tasks.
- Multiple algorithms can be performed in a single process with nodelet.
- For non-ROS apps, rosbridge provides a JSON API to ROS features.

2. Mapping and localization

- The slam toolbox is a complete 2D SLAM and localization system.
- For simultaneous localization and mapping, gmapping offers a wrapper for OpenSlam's Gmapping method.
- Google's cartographer delivers real-time 2D and 3D SLAM techniques.
- amcl is a library that implements adaptive Monte-Carlo localization.

3. Navigation

- The capacity to navigate a mobile robot in a planar environment is provided by navigation.

4. Perception

- Vision opencv is a meta-package that contains ROS and OpenCV integration packages.

5. Coordinate frame representation

- After Hydro, tf2 is the second iteration of the tf library, which provides the same functionality for ROS versions.

6. Simulation

- The meta-package gazebo ros pkgs contains packages for integrating ROS with the Gazebo simulator.

2.3 Docker

Docker is a group of platforms as a service (PaaS) solution that provide software in containers using OS-level virtualization. There are two tiers to the service: free and premium. Docker Engine is the program that runs the containers.

2.3.1 Images

A Docker image is a file that a Docker container uses to run programs. Docker images, like a template, serve as a collection of instructions for constructing a Docker container. When utilizing Docker, Docker images also serve as a starting point.
Dockerenv is the default command to run, and it may be run from any adb or ssh shell. The ohmnilabs/ohmnidev image, which is an Ubuntu 18.04 base image with some extra utilities included, will be pulled from the Docker Hub repository

2.3.2 Containers

A container is a standard software unit that encapsulates code and all of its dependencies so that the program may be moved from one computer environment to another fast and reliably. A Docker container image is a small, independent software package that contains everything needed to execute a program, including code, runtime, system tools, system libraries, and settings.

2.4 Camera Calibration

Geometric camera calibration, also known as camera re-sectioning, calculates the characteristics of an image or video camera's lens and image sensor. These characteristics can be used to adjust for lens distortion, estimate the size of an item in world units, or find the camera's position in the picture. These tasks are used to recognize and measure objects in applications such as machine vision. They're also employed in robotics, navigation systems, and reconstruction of 3-D scenes.

2.4.1 Camera Calibration Parameters

The camera matrix is calculated by the calibration process utilizing extrinsic and intrinsic characteristics. Extrinsic parameters describe a hard translation from a three-dimensional world coordinate system to a three-dimensional camera coordinate system. The intrinsic parameters reflect a projective translation from the coordinates of the 3-D camera to the coordinates of the 2-D picture. (source O. , camera calibration – ROS Wiki, n.d.)

1. Extrinsic Parameters

A rotation, R, and a translation, t, are the extrinsic parameters. The picture plane is defined by the x- and y-axis of the camera's coordinate system, which has its origin in the optical center.

2. Intrinsic Parameters

The intrinsic parameters include the focal length, the optical center, also known as the principal point, and the skew coefficient.

2.4.2 Distortion in Camera Calibration

Because an ideal pinhole camera does not have a lens, the camera matrix does not account for lens distortion. The camera model contains radial and tangential lens distortion to correctly simulate a genuine camera. (source M. o., n.d.)

1. Radial distortion

   When light rays bend more along the borders of a lens than they do in the optical center, radial distortion results. The bigger the distortion, the smaller the lens.

2. Tangential distortion

   When the lens and the picture plane are not parallel, tangential distortion develops. This sort of distortion is represented by tangential distortion coefficients.

2.5 ORB SLAM

2.5.1 Three Threads: Tracking, Local Mapping and Loop Closing

Every frame, the tracking is responsible for localizing the camera and determining when to insert a new keyframe. The algorithm starts with a feature match with the previous frame and then uses motion-only BA to optimize the posture. The location recognition module is used to do a global re-localization whenever the tracking is lost (for example, owing to occlusions or sudden movements). Following an initial estimate of the camera posture and feature matchings, a local visible map is recovered using the system's co-visibility graph of keyframes. After that, reprojection is used to look for matches with the local map locations, and the camera posture is

optimized again with all matches. Finally, the tracking thread determines whether or not to insert a new keyframe. (ORB-SLAM2)
To obtain an appropriate reconstruction in the surroundings of the camera posture, the local mapping processes new keyframes and conducts local BA. To triangulate new points, new correspondences for unmatched ORB in the new keyframe are searched in related keyframes in the co-visibility graph. An arduous point culling policy is implemented sometime after creation, depending on the information obtained throughout the tracking, in order to keep only high-quality points. Local mapping is also in charge of removing keyframes that are no longer needed.
With each new keyframe, the loop closing looks for loops. The algorithm calculates a similarity transformation that informs us about the drift collected in the loop if a loop is discovered. The loop's two sides are then aligned, and the duplicated points are fused. Finally, to establish global consistency, a pose graph optimization over similarity constraints is done. The primary innovation is that we optimize the co-visibility graph using the Essential Graph, a sparse subgraph. The algorithm uses the Levenberg-Marquardt algorithm implemented in g2o to carry out all optimizations. (Raúl Mur-Artal, 2015)

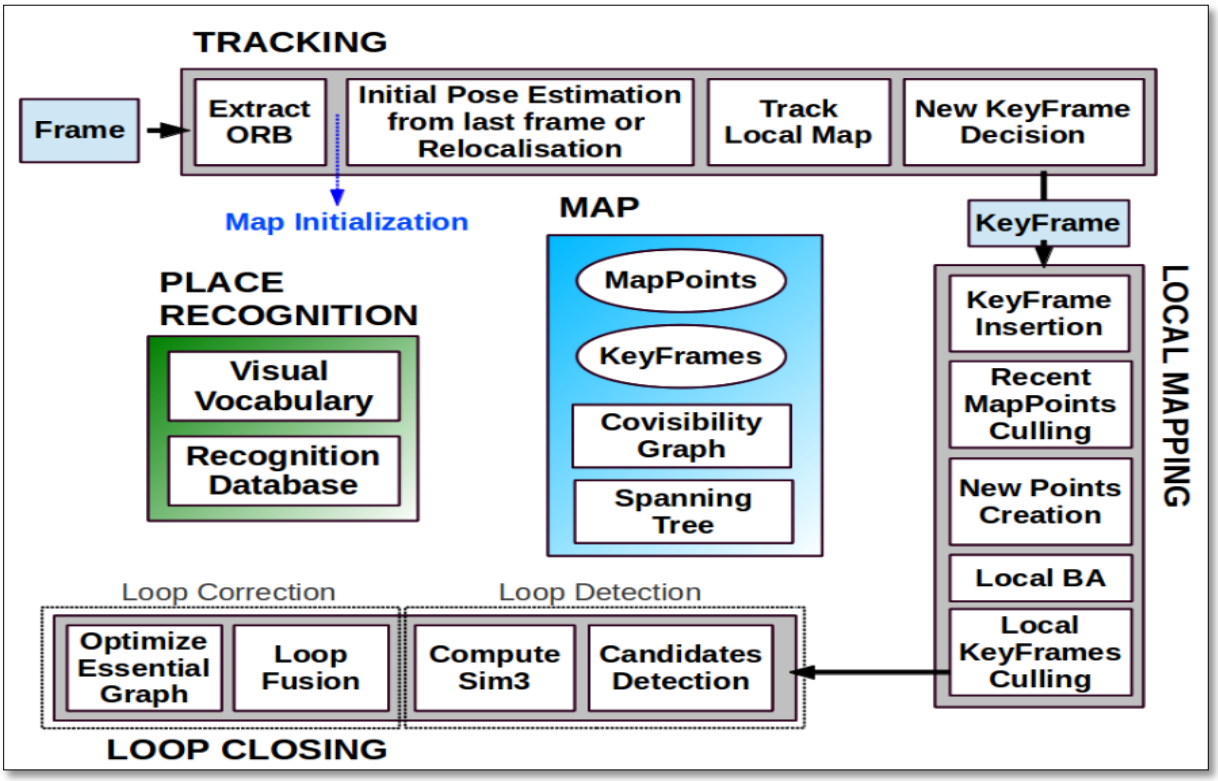

Figure 3 Threads of ORB SLAM

2.5.2 Map Points, Key-Frames and their Selection.

• Its 3D position Xw, i in the world coordinate system is stored in each map point pi.

• The mean unit vector of all its viewing directions (the rays that connect the point to the optical center of the keyframes that witness it), which is ni.

• A representative ORB descriptor Di, which is the associated ORB descriptor with the smallest hamming distance among all associated descriptors in the keyframes where the point is detected.

• The maximum dmax and lowest dmin distances at which the point may be viewed, as determined by the ORB features' scale invariance restrictions.

• The camera posture Tiw, which is a rigid body transformation that converts points from the world to the camera coordinate system, is stored in each keyframe Ki.

• All ORB features retrieved in the frame, whether or not related to a map point, whose coordinates are undistorted if a distortion model is specified.

A liberal policy is used to construct map points and keyframes, while a more stringent culling process is in charge of finding superfluous keyframes and incorrectly matched or un-trackable map points. This allows for a flexible map extension during exploration, which improves tracking resilience under difficult situations (e.g., rotations, quick movements), while keeping the map's size limited in repeated visits to the same area, i.e., lifetime operation.

2.5.3 Co-visibility Graph and Essential Graph

Co-visibility information between keyframes is quite valuable in a number of tasks in our system, and it is represented as an undirected weighted graph. Each node is a keyframe, and an edge exists between two keyframes if they share observations of the same map points (at least 15), with the number of shared map points determining the edge's weight.
To fix a loop, the algorithm uses pose graph optimization, which spreads the loop closing error throughout the graph. To avoid including all of the edges supplied by the co-visibility graph, which may be rather thick, the algorithm suggests creating an Essential Graph that keeps all of the nodes (keyframes) but fewer edges, while still maintaining a robust network that produces correct results. The approach generates a spanning tree progressively from the initial keyframe, resulting in a linked subgraph of the co-visibility graph with the fewest possible edges. (Raúl Mur-Artal, 2015)

2.6 YOLOv3

2.6.1 Bounding Box Prediction

Using dimension clusters as anchor boxes, the method predicts bounding boxes. For each bounding box, the network predicts four coordinates: tx, ty, tw, and th. If the cell is displaced from the image's top left corner by (cx, cy) and the enclosing box has width and height of pw, ph. We utilize the sum of squared error loss during training. If the ground truth for a coordinate prediction is t*, then our grain is the ground truth value (calculated from the ground truth box) minus our prediction: t* t*. By inverting the equations above, we can simply calculate the ground truth value. Using logistic regression, YOLOv3 predicts an objectness score for each bounding box. If the bounding box prior overlaps a ground truth object by more than any other bounding box prior, this value should be 1. We disregard the prediction if the bounding box prior is not the best but overlaps a ground truth object by more than a threshold. The algorithm uses the value of as a threshold 0.5. Unlike the system, each ground truth object is only given one bounding box in advance. There is no loss in coordinate or class predictions if a bounding box prior is not given to a ground truth object; only objectness is lost. (pjreddie)

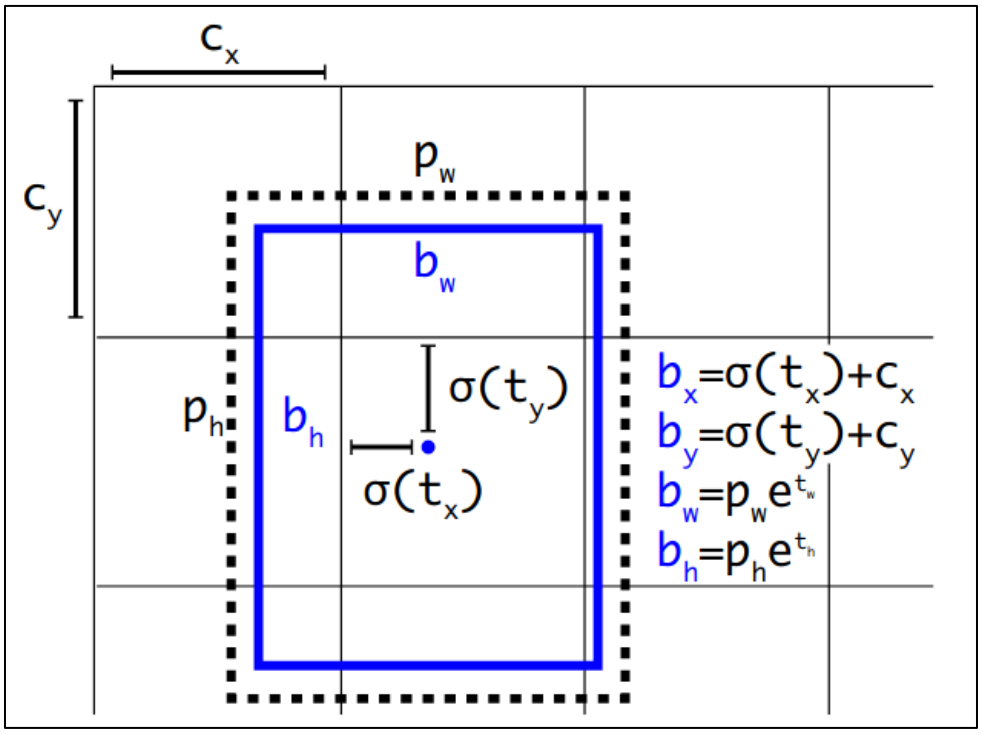

Figure 4 Bounding Box YOLOv3

2.6.2 Class prediction

Using multilabel classification, each box predicts the classes that the bounding box may include. The approach here does not employ a SoftMax because it has been discovered that it is not essential

for optimal performance; instead, it relies on independent logistic classifiers. The approach uses binary cross-entropy loss for class predictions during training. (pjreddie)

2.6.3 Predictions Across Scales

At three distinct scales, YOLOv3 predicts boxes. Using a concept akin to feature pyramid networks, the system collects features from those scales. It adds many convolutional layers to the underlying feature extractor. Last but not least, a 3-d tensor encapsulating bounding box, objectness, and class predictions are predicted.

2.6.4 Feature Extractor

The new network is a cross between the YOLOv2 network, Darknet-19, and the newfangled residual network stuff. The network employs consecutive 3*3 and 1*1 convolutional layer, but it now includes shortcut connections and is much bigger. There are 53 convolutional layers in total.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 5 Neural Network structure for YOLOv3

2.6.5 Training

The system continues to be trained on whole pictures, with no hard negative mining or other such techniques. It employs multi-scale training, a large amount of data augmentation, batch normalization, and other common techniques. For training and testing, the Darknet neural network framework is also employed. (pjreddie)

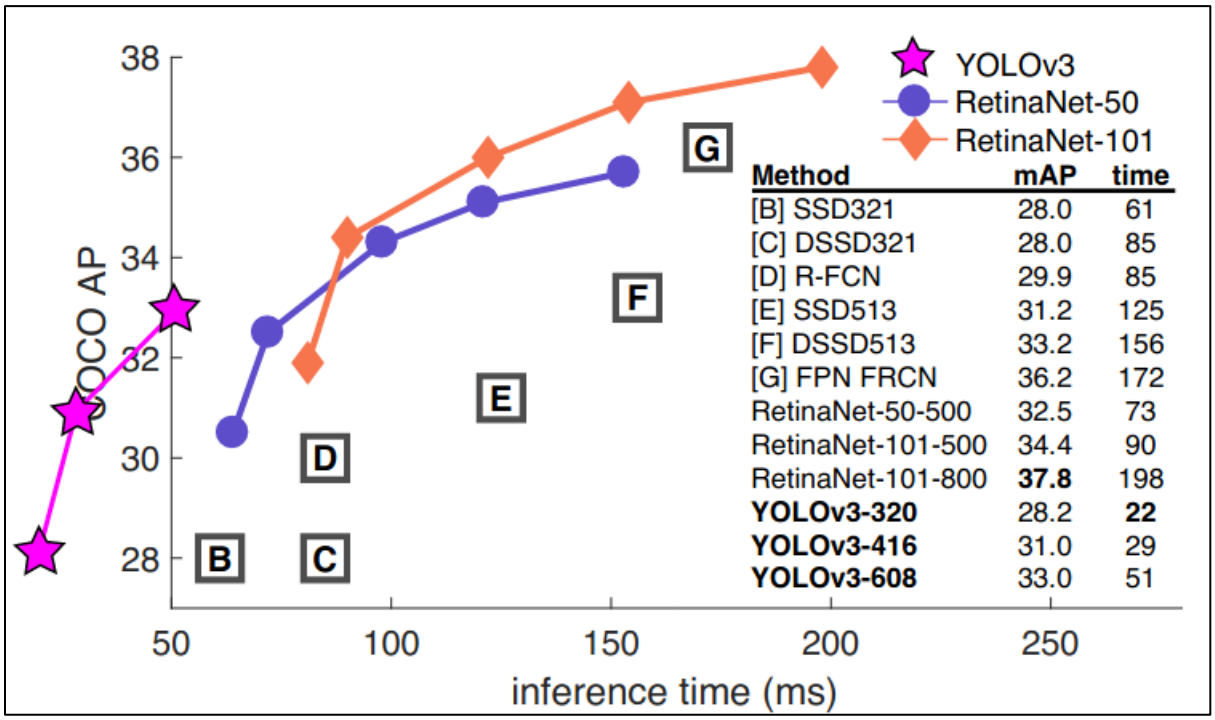

Figure 6 Performance of YOLOv3

2.7 Face Recognition

Face recognition is actually a collection of related issues:

- First, look at a photograph and identify all of the people in it.
- Second, concentrate on each face and recognize that it is still the same person even if it is turned in an unusual way or lit poorly.
- Third, be able to distinguish distinctive aspects of the face from other individuals, such as the size of the eyes, the length of the face, and so on.
- Finally, ascertain the person's name by comparing the distinctive qualities of that face to all of the persons you already know.

Step 1: Finding all the Faces

To construct a reduced version of an image, use the HOG method to encode it. Find the portion of the picture that most closely resembles a general HOG encoding of a face using this simplified image. Grey scale image is enough to detect the faces in the image. Independent of the brightness of the image. Gradient: it shows the flow from light to dark across the entire image depending upon the surround pixels for each pixel. All we have to do to locate faces in this HOG image is find the region of our image that looks the most like a known HOG pattern that was taken from a bunch of other training faces. (recognition) (Geitgey)

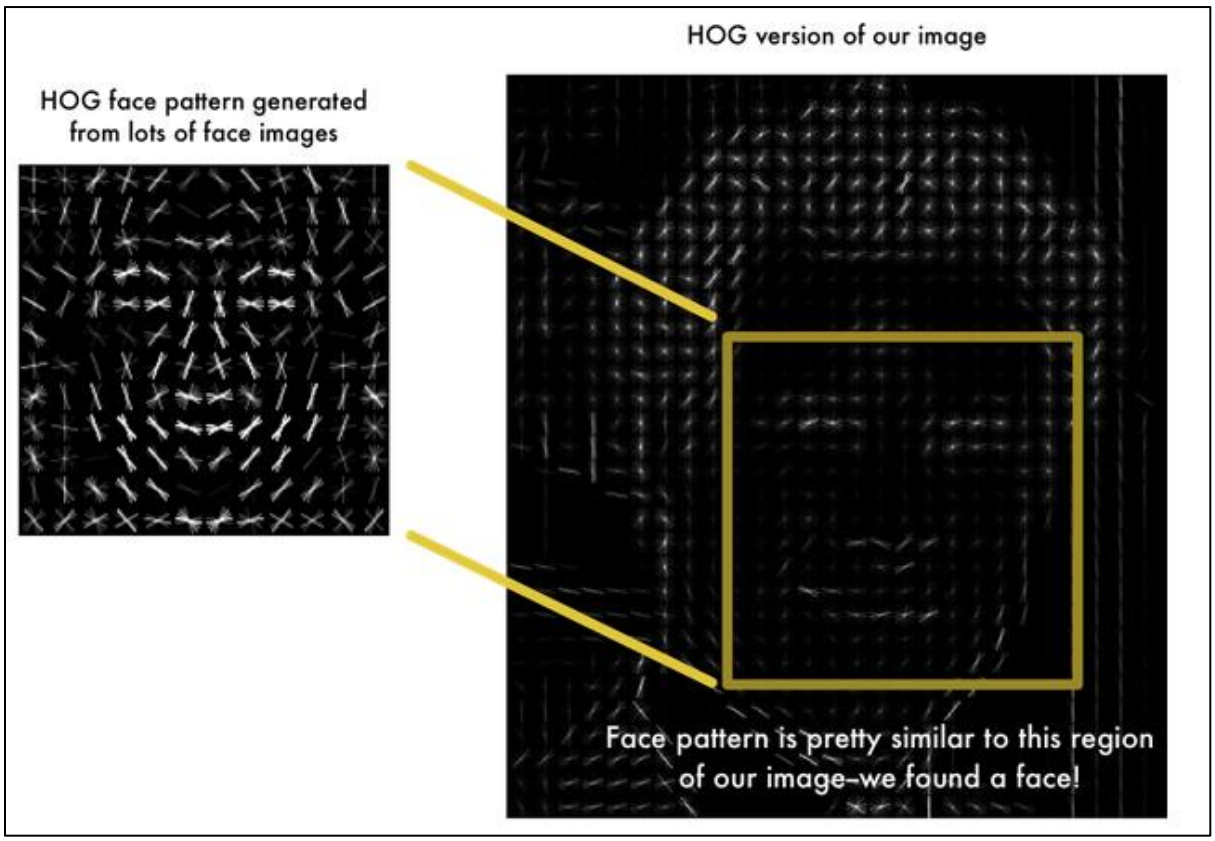

Figure 7 HOG pattern for face detection

Step 2: Posing and Projecting Faces

Find the primary landmarks in the face to determine the face's position. Once we've located those landmarks, we can utilize them to distort the picture to center the eyes and mouth. Face landmark estimation -The main concept is that we will identify 68 distinct places (known as landmarks) on every face, such as the top of the chin, the outside edge of each eye, the inside edge of each brow, and so on. Then, on any face, we'll train a machine learning algorithm to locate these 68 specific points. (Geitgey)

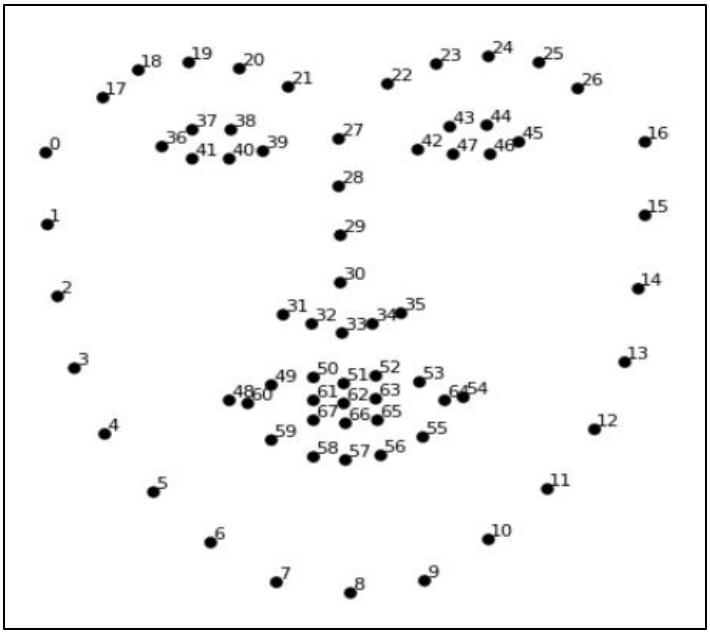

Figure 8 Sixty-Eight distinct landmarks for face recognition.

Step 3: Encoding Faces

Run the centered face picture through a neural network that understands how to measure facial traits. Keep track of the 128 measurements.

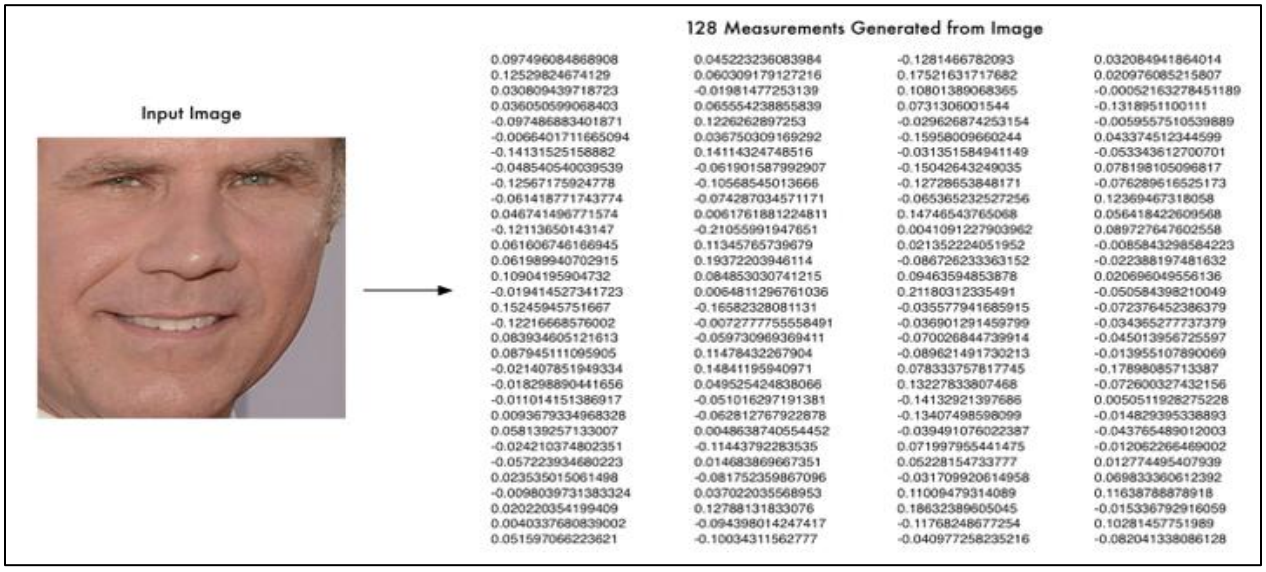

Figure 9 128 measurements - face encoding process

Step 4: Finding the person's name from the encoding

This final step is actually the simplest of the entire procedure. All we have to do now is look through our database of known persons to discover the person with the most similar dimensions to our test image. (Geitgey)

3 State of Art

3.1 Literature Survey

3.1.1 ORB SLAM

Simultaneous localization and mapping is a well-studied issue in robotics, and there is a substantial volume of information in the form of books and scholarly articles available. Implementing SLAM on the Ohmni robot was a prerequisite for the research project. We looked at some of the implementations to see if we could apply the same concepts to the Butler robot.

The work done by appliedAI-Initiative on ORB SLAM was helpful, giving the whole idea about the implementation of algorithms on the robot. This is the ROS implementation of the ORB-SLAM2 real-time SLAM library, which computes the camera trajectory and a sparse 3D reconstruction for Monocular, Stereo, and RGB-D cameras (in the stereo and RGB-D case with true scale). It can recognize loops and reposition the camera in real-time. (ORB-SLAM2)

3.1.2 YOLOv3

The topic of object detection is broadly researched in Artificial Intelligence. We got a considerable number of references and literature on this topic in form of articles and web pages. To implement the object detection on the Butler robot for this research project we had to take some references where this algorithm has been implemented.

Yolo ROS: Real-time Object detection for ROS gives detailed information about the YOLO algorithm and elaborates the information on the programming robot with ROS. This is a ROS module for detecting objects in camera photos. YOLO (You Only Look Once) is a cutting-edge, real-time object detection technology. You may utilize YOLO (V3) on GPU and CPU using this ROS package. The convolutional neural network's pre-trained model may recognize pre-trained classes, such as the VOC and COCO data sets, or you can build your network using your detection objects. (source O. , COCO data set paper) (pjreddie)

3.1.3 Face Recognition

The problem of face recognition has received a lot of attention in the Artificial Intelligence community. In the form of articles, we gathered a substantial quantity of reference and literature on this issue. To develop face recognition on the Butler robot for this research project, we needed to look at several examples of this technique in action.

The (pjreddie) provides in-depth details on the facial recognition algorithm and expands on the programming robot using ROS. This face detector combines a linear classifier, an image pyramid, and a sliding window detection approach with the now-classic Histogram of Oriented Gradients (HOG) feature. In addition to human faces, this sort of object detector is fairly generic and capable of identifying a wide range of semi-rigid objects.

4 Solution Design

4.1 System Components

4.1.1 Hardware Setup

4.1.1.1 Monocular USB Camera

Implementation of ORB SLAM with our local system required a USB camera and Robot Camera. We have issue with robot camera, so we have applied the SLAM with the help of USB camera. We have discussed the issue in further chapter in detail. The cost of the camera is 10Euro. Whenever working with a Butler robot, it doesn't need any external hardware except the local system for processing algorithms.

4.1.2 Software Setup

4.1.2.1 ROS

ROS (Robot Operating System) is a framework for operating robotic components from a computer in this project. The ROS system consists of a collection of free nodes that connect via a subscription/subscription order approach. The Robot Operating System (ROS) is a platform for developing robot software. It's a suite of tools, libraries, and protocols for designing robust and resilient robot behavior on a range of robot platforms. ROS was built from the ground up to promote the development of collaborative robotic software.

4.1.2.2 Camera Calibration Parameters

After the camera calibration is done, we get all the required calibration parameters. These parameters have to be mentioned in all of the launch files of ORB SLAM. (source O. , camera calibration – ROS Wiki, n.d.)

Figure 10 Camera parameters after calibration of camera is done.

4.1.2.3 ORB SLAM

The task of simultaneous localization and mapping of the environment is done with the help of a visual slam that is ORB SLAM. ORB-SLAM is a versatile and accurate Monocular SLAM system that can calculate the camera trajectory and a sparse 3D reconstruction of the scene in real-time in a broad range of situations, from short hand-held sequences to a car driving around many city blocks. It can complete enormous loops and execute global re-localization from a wide range of baselines in real-time. It comes with a robust and automated initialization from both planar and non-planar scenarios. Because keyframes are introduced relatively quickly during exploration, a new survival of the fittest keyframe selection allows for the preservation of a small map while boosting tracking resilience. It's a monocular SLAM system with a thorough breakdown of its components. The system can process sequences from both indoor and outdoor settings, as well as robot and hand-held gestures. In tiny inside circumstances, the system's precision is often less than 1 cm, while in big outside scenarios, it's a few meters. (ORB-SLAM2) (Raúl Mur-Artal, 2015)

Install the packages required for ORB SLAM, in the launch folder we have a launch file named robot_ohmni.launch , urdf_ohmni.launch, and usb_cam.launch. robot_ohmni.launch file is used with the transform frames of the robot tf's publishers is tb_control. urdf_ohmni.launch is used with the URDF of the Robot where tf's publisher is robot_state_publisher. The usb_cam.lanuch file is used for usb_cam while performing ORB SLAM. The camera calibration parameters should be added to all launch files.

In the robot_ohmni.launch file we have added static transforms between map frame and Odom frame, between footprint frame and base_link frame, and between base_link and main_cam. When the urdf_ohmni.launch file is considered here, we have added only one static transform between base_link and main_cam. In the usb_cam.lanch file there is no requirement of a static transform. We are using octomap_server and map_server for our creation of maps, so these two ROS packages must be installed on the local system. We have to provide the camera rostopic in MonoNode.cc

4.1.2.4 Robot Setup

A. Docker Environment

To begin working with the robot or to deploy custom code, you must first connect to it.

The procedures to allow communication with the robot is outlined below.

• To get to the shell on your smartphone, go to the Ohmni app's settings button (on the blue screen), and then hit the "Version Name" item seven times. This will disclose all objects that have been concealed.

• Scroll to the bottom of the page and choose "Enable ADB." This will launch the ADB daemon, which will allow you to connect through the local network.

We as a ROS developer can run the ROS frame work on the top of Ohmni OS using docker virtualization layer,

i. tb_control_docker_image

In bot cli> **setprop ctl.stop tb-node**

In bot cli> **docker pull ohmnilabsvn/ohmni_ros:ohmni_ros_tbcontrol_0.0.8.1**

In bot cli> **docker run -it --network host --privileged -v /dev:/dev -e ROS_IP=192.168.178.104 ohmnilabsvn/ohmni_ros:ohmni_ros_tbcontrol_0.0.8.1 bash**

In your local system terminal, export and provide the required IP addresses:

export ROS_IP=[local_system_IP]

export ROS_MASTER_URI=http://[butler_robot_ip]:11311

The rostopic command should give all the ros topics published by the butler robot on the local system.

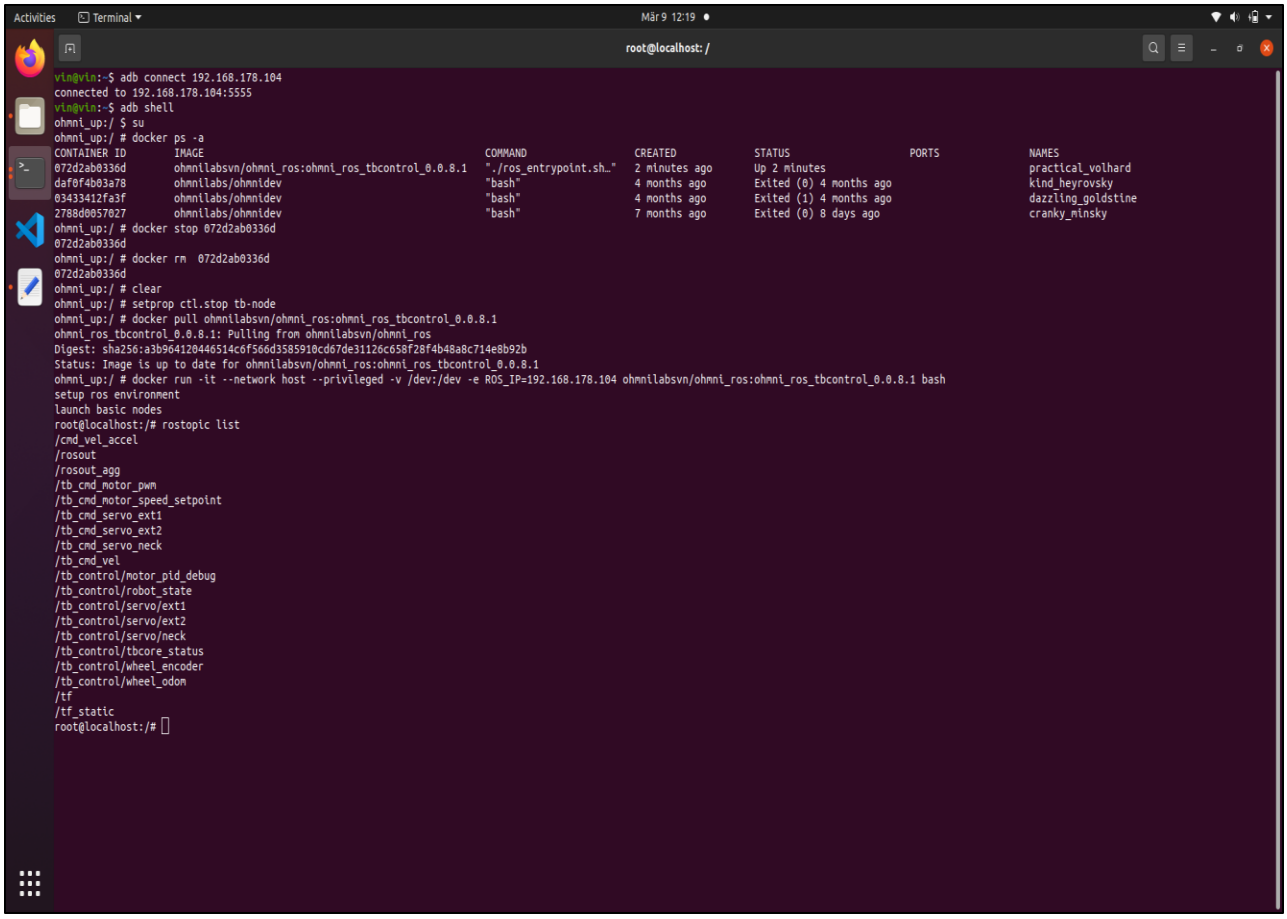

Figure 11 Running an image for tb_control docker

ii. RGB_docker_image

In bot cli> **docker pull baoden/ohmni_rgbcam_ros:launch_ros**

In bot cli> **docker run -it --network host --privileged -v /dev:/dev -e ROS_IP=192.168.178.104 baoden/ohmni_rgbcam_ros:launch_ros bash**

In your local system terminal, export and provide the required IP addresses:

export ROS_IP=[local_system_IP]

export ROS_MASTER_URI=http://[butler_robot_ip]:11311

The rosoptic command should give all the ros topics published by the butler robot on the local system.

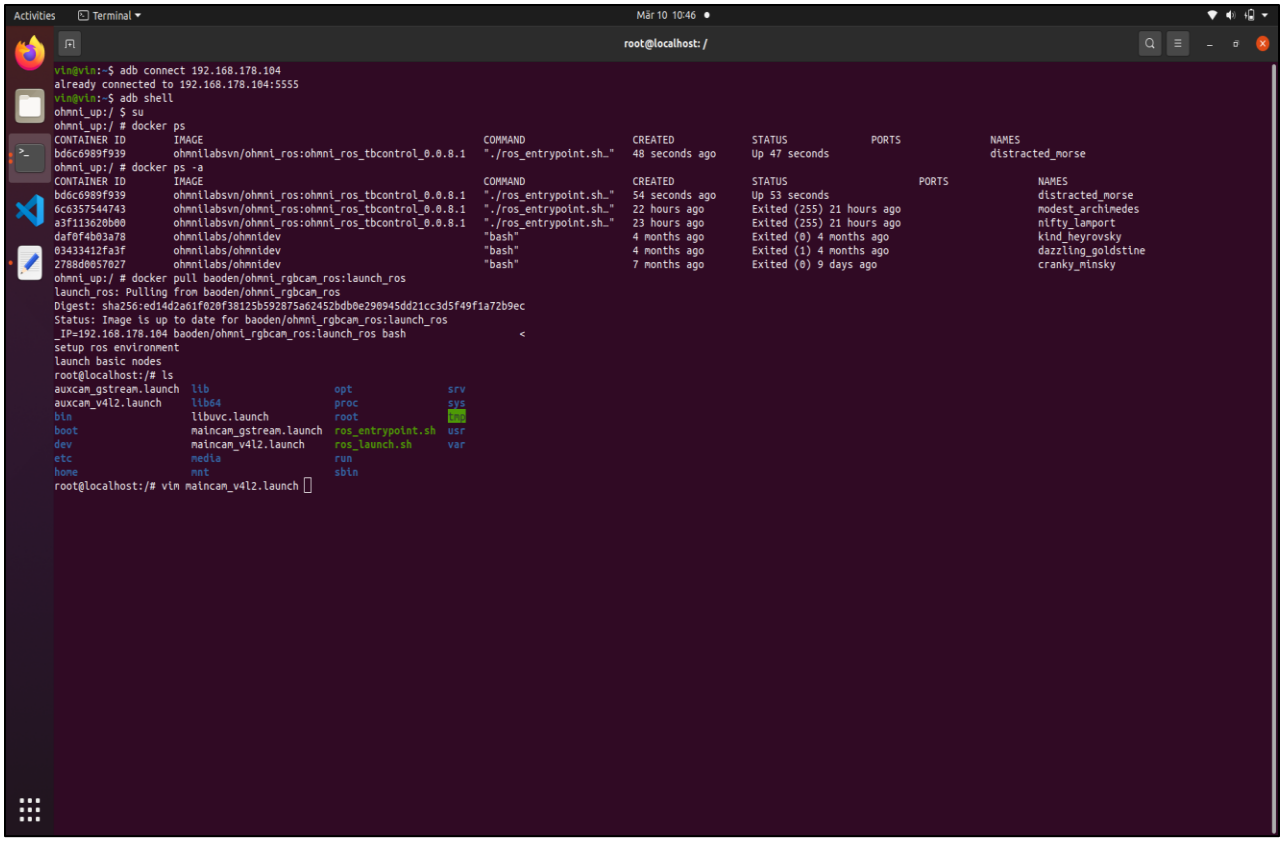

Figure 12 Running image for RGB camera of Robot.

B. Pixel format of Image received by the butler robot.

The color code of the image is not RGB, we have to change the pixel format in the launch file of the main_cam. The default pixel format is uyvy, we must change it to **yuyv**.

Make the changes in the file as mentioned below:

In root@localhost:/ we have the file named maincam_v4l2.launch

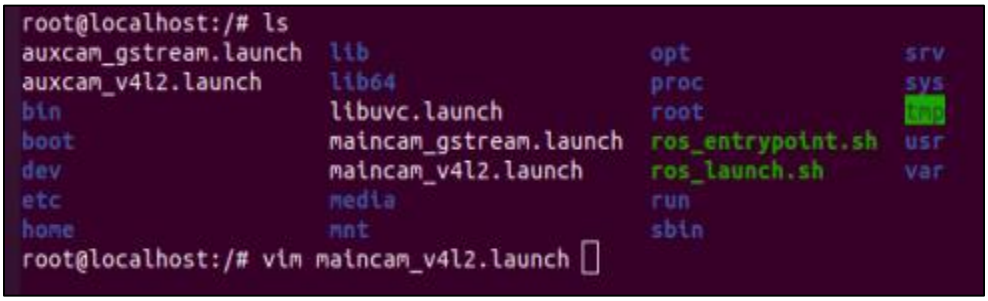

Figure 13 Launch file of RGB camera.

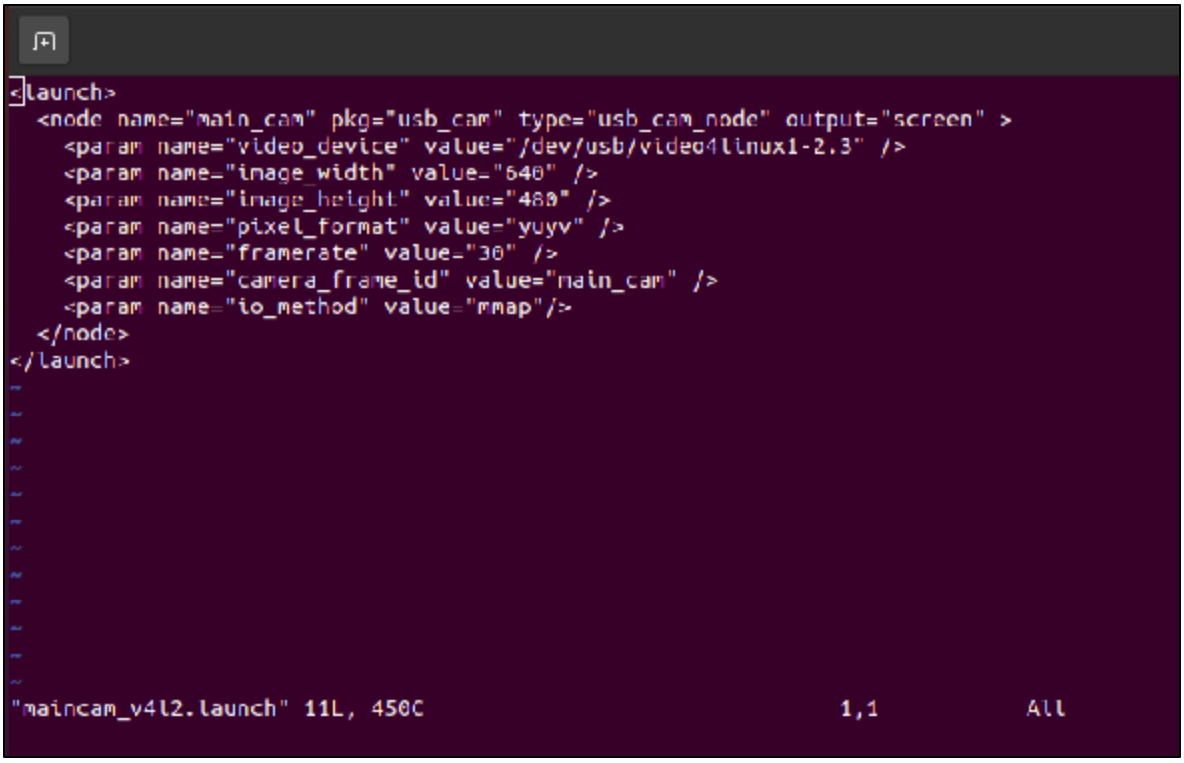

Figure 14 Pixel format in launch file. (yuyv)

C. Launch files and Transform Frames (location and coordinates)

We have provided the location of the main_cam (visual sensor) with the help of static transform between base_link and main_cam. It gives the idea about the location and coordinates the visual sensor concerning other transform frames. It is required as the robot needs to locate all the functioning parts used while executing tasks. (source O. , map_server - ROS Wiki, n.d.) (source O. , octomap_server - ROS Wiki, n.d.)

```
<?xml version="1.0"?>
<launch>
       <!--  <node pkg="tf" type="static_transform_publisher" name="map_to_odom" args="0 0 0 0 0 0 1 map odom 27"/>
       <node pkg="tf" type="static_transform_publisher" name="footprint_to_base_link" args="0 0 0.075 0 0 0 1 footprint base_link 27"/>
       <node pkg="tf" type="static_transform_publisher" name="base_link_to_camera_link" args="-0.069 0 1.363 0 0 0 1 base_link main_cam 27"/> -->
       <node name="orb_slam2_mono" pkg="orb_slam2_ros" type="orb_slam2_ros_mono" output="screen">
       <!-- localize has to be false while creating a new map -->
      <param name="publish_pointcloud" type="bool" value="true" />
      <param name="publish_pose" type="bool" value="true" />
      <param name="localize_only" type="bool" value="false" />
      <param name="reset_map" type="bool" value="false" />
      <!-- static parameters -->
      <!-- save the map, then load it here by value ="true" -->
      <param name="load_map" type="bool" value="false" />
      <param name="map_file" type="string" value="map.bin" />
      <param name="voc_file" type="string" value="$(find orb_slam2_ros)/orb_slam2/Vocabulary/ORBvoc.txt" />
      <param name="pointcloud_frame_id" type="string" value="map" />
      <param name="camera_frame_id" type="string" value="main_cam" />
      <param name="min_num_kf_in_map" type="int" value="5" />
      <!-- ORB parameters -->
      <param name="/ORBextractor/nFeatures" type="int" value="2000" />
      <param name="/ORBextractor/scaleFactor" type="double" value="1.2" />
      <param name="/ORBextractor/nLevels" type="int" value="8" />
      <param name="/ORBextractor/iniThFAST" type="int" value="20" />
      <param name="/ORBextractor/minThFAST" type="int" value="7" />
      <!-- Camera parameters -->
      <!-- Camera frames per second -->
      <param name="camera_fps" type="int" value="30" />
      <!-- Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale) -->
      <param name="camera_rgb_encoding" type="bool" value="true" />
       <!-- Camera calibration parameters -->
       <!--If the node should wait for a camera_info topic to take the camera calibration data-->
      <param name="load_calibration_from_cam" type="bool" value="false" />
      <!-- Camera calibration and distortion parameters (OpenCV) -->
     <param name="camera_fx" type="double" value="205.805216" />
     <param name="camera_fy" type="double" value="205.003250" />
     <param name="camera_cx" type="double" value="318.620705" />
     <param name="camera_cy" type="double" value="238.230326" />
      <!-- Camera calibration and distortion parameters (OpenCV) -->
     <param name="camera_k1" type="double" value="0.493141" />
     <param name="camera_k2" type="double" value="-0.27386" />
     <param name="camera_p1" type="double" value="-0.66150" />
     <param name="camera_p2" type="double" value="0.378325" />
     <param name="camera_k3" type="double" value="0.0" />
  </node>
</launch>
```

Figure 15 Launch file for ORB SLAM with Robot tb_control tf's.

```
<?xml version="1.0"?>
<launch>

       <node name="orb_slam2_mono" pkg="orb_slam2_ros" type="orb_slam2_ros_mono" output="screen">
       <!-- localize has to be false while creating a new map -->
      <param name="publish_pointcloud" type="bool" value="true" />
      <param name="publish_pose" type="bool" value="true" />
      <param name="localize_only" type="bool" value="false" />
      <param name="reset_map" type="bool" value="false" />
      <!-- static parameters -->
      <!-- save the map, then load it here by value ="true" -->
      <param name="load_map" type="bool" value="false" />
      <param name="map_file" type="string" value="map.bin" />
      <param name="voc_file" type="string" value="$(find orb_slam2_ros)/orb_slam2/Vocabulary/ORBvoc.txt" />
      <param name="pointcloud_frame_id" type="string" value="map" />
      <param name="camera_frame_id" type="string" value="main_cam" />
      <param name="min_num_kf_in_map" type="int" value="5" />
      <!-- ORB parameters -->
      <param name="/ORBextractor/nFeatures" type="int" value="2000" />
      <param name="/ORBextractor/scaleFactor" type="double" value="1.2" />
      <param name="/ORBextractor/nLevels" type="int" value="8" />
      <param name="/ORBextractor/iniThFAST" type="int" value="20" />
      <param name="/ORBextractor/minThFAST" type="int" value="7" />
      <!-- Camera parameters -->
      <!-- Camera frames per second -->
      <param name="camera_fps" type="int" value="30" />
      <!-- Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale) -->
      <param name="camera_rgb_encoding" type="bool" value="true" />
       <!-- Camera calibration parameters -->
       <!--If the node should wait for a camera_info topic to take the camera calibration data-->
      <param name="load_calibration_from_cam" type="bool" value="false" />
      <!-- Camera calibration and distortion parameters (OpenCV) -->
     <param name="camera_fx" type="double" value="205.805216" />
     <param name="camera_fy" type="double" value="205.003250" />
     <param name="camera_cx" type="double" value="318.620705" />
     <param name="camera_cy" type="double" value="238.230326" />
      <!-- Camera calibration and distortion parameters (OpenCV) -->
     <param name="camera_k1" type="double" value="0.493141" />
     <param name="camera_k2" type="double" value="-0.27386" />
     <param name="camera_p1" type="double" value="-0.66150" />
     <param name="camera_p2" type="double" value="0.378325" />
     <param name="camera_k3" type="double" value="0.0" />
  </node>
  <node pkg="rviz" name="rviz" type="rviz" args="-d $(find orb_slam2_ros)ros/config/rvizconfig.rviz"/>
  <!-- send urdf to param server -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find orb_slam2_ros)ros/urdf/ohmni_bot.urdf.xacro'" />
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
       <param name="use_gui" value="false"/>
  </node>
  <!-- Send robot states to tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen"/>
</launch>
```

Figure 16 Launch file for ORB SLAM using URDF of Butler Robot.

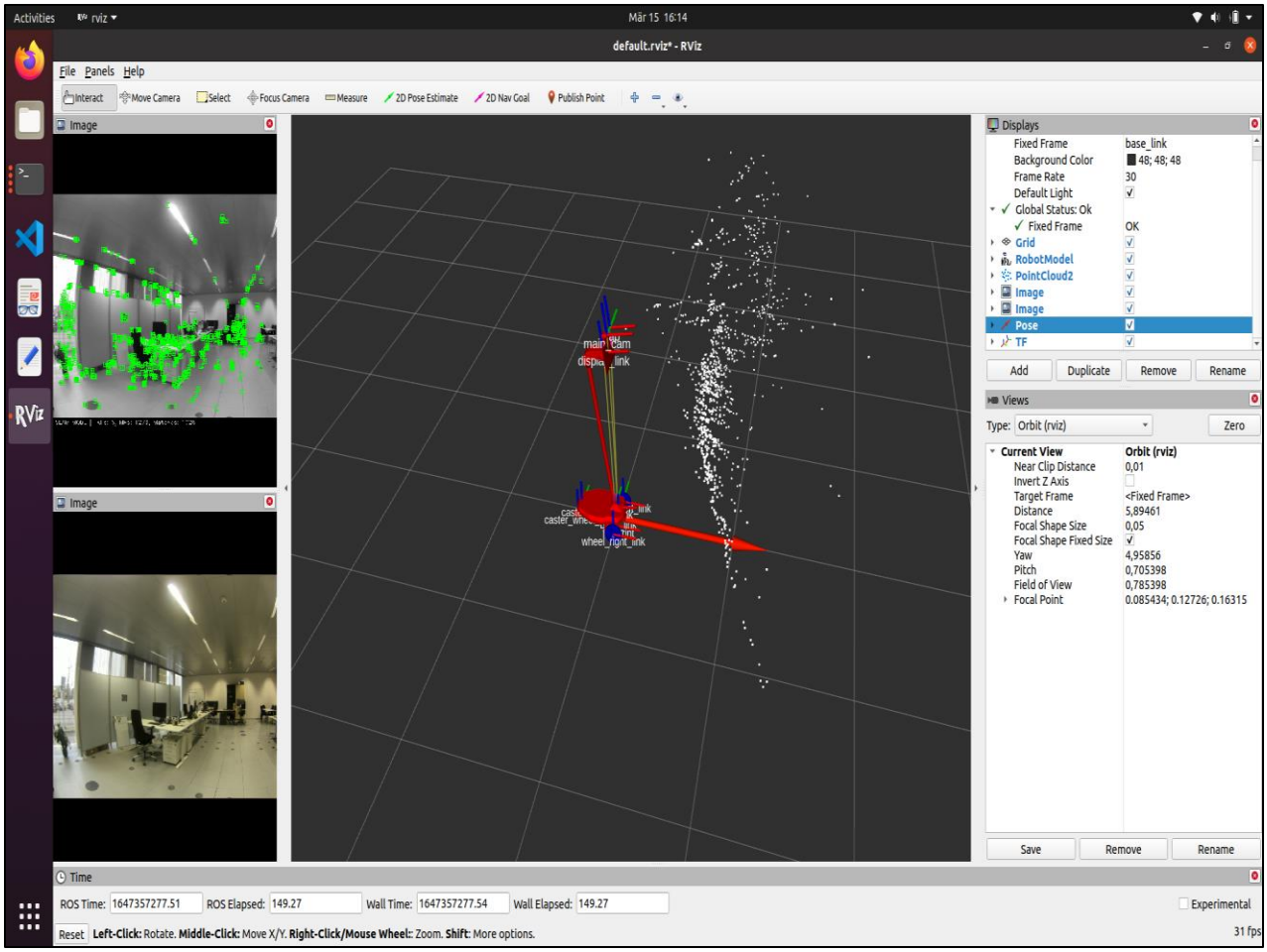

Figure 17 Rviz visualization with URDF of Butler Robot.

```
<link name="main_cam">
  <visual>
    <geometry>
      <box size="0.01 0.01 0.01"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <box size="0.01 0.01 0.01"/>
    </geometry>
  </collision>
 <inertial>
    <origin
      xyz="-0.00036659 0.00064155 -0.00027052"
      rpy="0 0 0" />
    <mass
      value="0.25482" />
    <inertia
      ixx="8.9194E-05"
      ixy="1.8116E-07"
      ixz="3.6435E-08"
      iyy="0.00018279"
      iyz="2.423E-07"
      izz="0.00011103" />
  </inertial>
</link>
<joint name="camera_mount_joint" type="fixed">
  <origin rpy="0.0 -0.183 0.0" xyz="-0.069 0 1.363"/>
  <parent link="base_link"/>
  <child link="main_cam"/>
</joint>
```

Figure 18 Static link between base_link and main_cam in URDF file of Butler Robot.

D. Octomap_server package

In the launch file of the octomap_server we have to provide the ros topic for 3D cloud point, in the ORB SLAM case its /orb_slam2_mono/map_points. This package is used to get the /ocuppancy_grid and /projected_map where it is further used by the map_server to save the map in .pgm and .yaml format. It gives us the visualization of the map in a convenient way. (source O. , octomap_server - ROS Wiki, n.d.) (source O. , map_server - ROS Wiki, n.d.)

```
<?xml version="1.0"?>
<launch>
        <node pkg="octomap_server" type="octomap_server_node" name="octomap_server">
                <param name="resolution" value="0.005" />

                <!-- fixed map frame (set to 'map' if SLAM or localization running!) -->
                <param name="frame_id" type="string" value="map" />

                <!-- maximum range to integrate (speedup!) -->
                <param name="sensor_model/max_range" value="5.0" />

                <!-- data source to integrate (PointCloud2) -->
                <remap from="cloud_in" to="/orb_slam2_mono/map_points" />

        </node>
</launch>
```

Figure 19 Launch file for octo_map_server.

4.1.2.5 YOLOv3 - Object Detection.

Implementation of YOLOv3 requires a launch file, where the rostopic of the camera used has to be given. In this launch file, we have to provide the .yaml file path. Another launch file is launched where we have to provide configuration file path and network weight path.

Open

darknet_ros.launch
~/catkin_ws/src/darknet_ros/darknet_ros/launch

```
<?xml version="1.0" encoding="utf-8"?>

<launch>
  <!-- Console launch prefix -->
  <arg name="launch_prefix" default=""/>
  <arg name="image" default="/main_cam/image_raw" />

  <arg name="yolo_weights_path"          default="$(find darknet_ros)/yolo_network_config/weights"/>
  <arg name="yolo_config_path"           default="$(find darknet_ros)/yolo_network_config/cfg"/>

  <!-- ROS and network parameter files -->
  <arg name="ros_param_file"             default="$(find darknet_ros)/config/ros.yaml"/>
  <arg name="network_param_file"         default="$(find darknet_ros)/config/yolov2-tiny.yaml"/>

  <!-- Load parameters -->
  <rosparam command="load" ns="darknet_ros" file="$(arg ros_param_file)"/>
  <rosparam command="load" ns="darknet_ros" file="$(arg network_param_file)"/>

  <!-- Start darknet and ros wrapper -->
  <node pkg="darknet_ros" type="darknet_ros" name="darknet_ros" output="screen" launch-prefix="$(arg launch_prefix)">
    <param name="weights_path"          value="$(arg yolo_weights_path)" />
    <param name="config_path"           value="$(arg yolo_config_path)" />
    <remap from="camera/rgb/image_raw"  to="$(arg image)" />
  </node>

<!--<node name="republish" type="republish" pkg="image_transport" output="screen"   args="compressed in:=/front_camera/image_raw raw out:=/main_cam/image_raw" /> -->
</launch>
```

Figure 20 Main Launch for ORB SLAM. (darknet_ros.launch)

Open

yolo_v3.launch
~/catkin_ws/src/darknet_ros/darknet_ros/launch

```
1 <?xml version="1.0" encoding="utf-8"?>
2
3 <launch>
4
5   <!-- Use YOLOv3 -->
6   <arg name="network_param_file"         default="$(find darknet_ros)/config/yolov3.yaml"/>
7   <arg name="image" default="/main_cam/image_raw" />
8
9
10  <!-- Include main launch file -->
11  <include file="$(find darknet_ros)/launch/darknet_ros.launch">
12    <arg name="network_param_file"    value="$(arg network_param_file)"/>
13    <arg name="image" value="$(arg image)" />
14  </include>
15
16 </launch>
```

Figure 21 yolo_v3.launch file

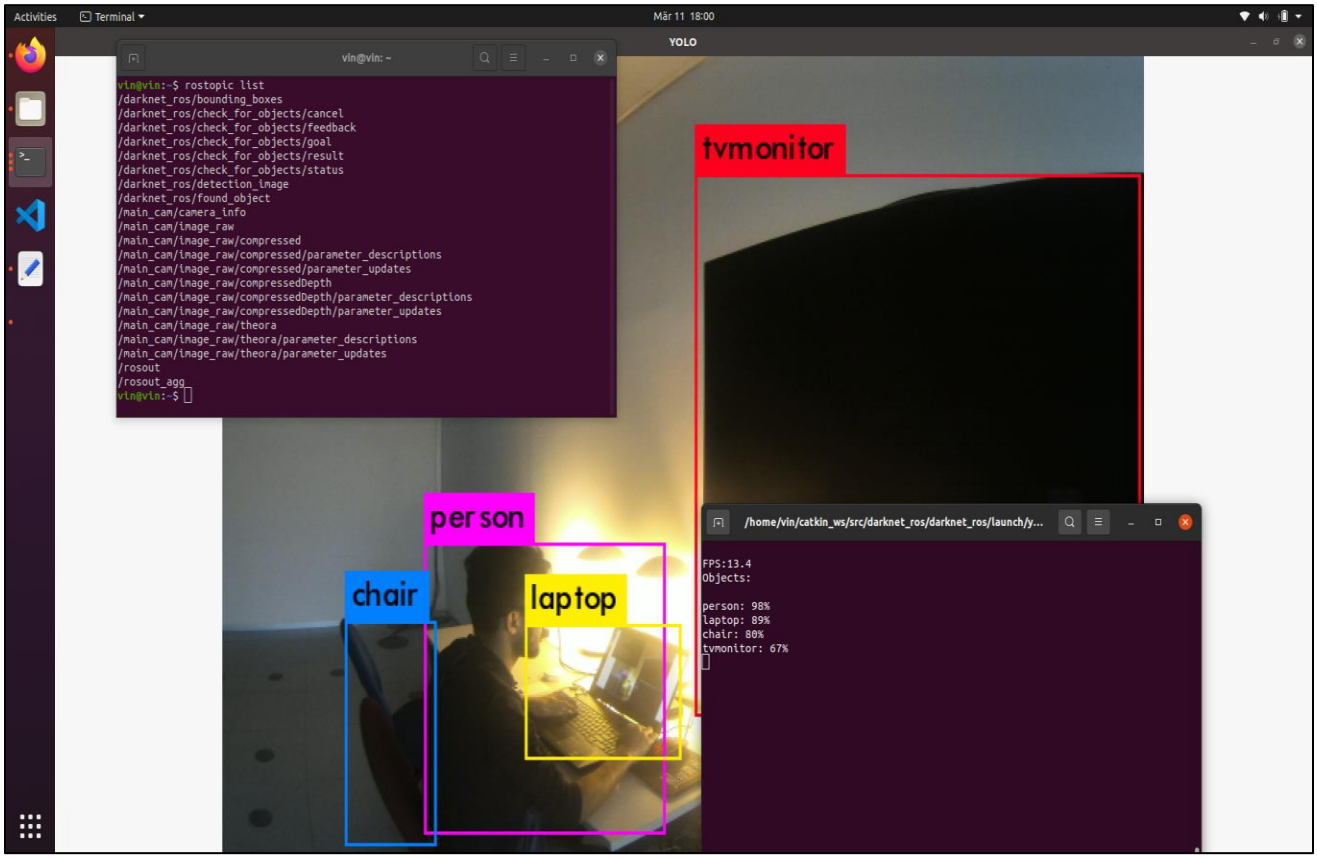

Figure 22 ros topic list.

4.1.2.6 Face Recognition

Install the package robocup_msgs before launching the launch file for face recognition. In the launch file face_detection.launch we have to provide the paths for labeled_people, auto_labeled_people, img_tests, and config file.

We have provided the image ros topic in the PeopleFaceIdentification_simple.py python script itself so, no need of providing ros topic in the launch file.

```
<?xml version="1.0" encoding="utf-8"?>
<launch>

  <rosparam file="$(find people_face_identification)/config/common_face.yaml" />
  <param name="/PeopleFaceIdentificationSimple/face_folder" value="$(find people_face_identification)/data/labeled_people"/>
  <param name="/PeopleFaceIdentificationSimple/face_folder_auto" value="$(find people_face_identification)/data/auto_labeled_people"/>
  <param name="/PeopleFaceIdentificationSimple/imgtest_folder" value="$(find people_face_identification)/data/img_tests"/>
  <param name="/PeopleFaceIdentificationSimple/config_folder" value="$(find people_face_identification)/config"/>

  <node pkg="people_face_identification" type="PeopleFaceIdentification_simple.py" name="PeopleFaceIdentificationSimple"  output="screen" >
  </node>

  <node name="image_view_face" pkg="image_view" type="image_view" respawn="false" output="screen">
     <remap from="image" to="/face_detection/face_image"/>
     <!-- <remap from="image" to="/face_detection/all_faces_image"/> -->
     <param name="autosize" value="false" />
  </node>

</launch>
```

Figure 23 face_detection.launch file.

5. Experimental Set up and Results

5.1 ROS master configuration

It's crucial to export data so that the commands are executed and use ROS tools like rviz on the workstation. We must set a variable "ROS MASTER URI " and "ROS IP" with the correct IP address in the bashrc file on both the Ohmni's and the Linux PC that runs the rviz.

In localhost terminal window export the IP addresses

export ROS_IP= [IP of Local_Host]

export ROS_MASTER_URI=http://[IP_of_Robot]:11311

Note: Whenever a new window terminal is opened, these two commands should be executed without fail. If it's not done, the ros topics from the Butler robot won't be observed on the localhost.

5.2 Creating 2D map of the Environment

Before implementation of ORB SLAM, camera calibration has to be done. Use the camera_calibration ROS package to get the camera calibration parameters. This package uses OpenCV camera calibration. We must have a chess board image to calibrate the camera.

Run the cameracalibrator.py node for a monocular camera using a 7x6 chessboard with 240mm squares. Choose a fisheye camera for the process of calibration.

rosrun camera_calibration cameracalibrator.py --size 7x6 --square 0.024 image:=/main_cam/image_raw camera:=/main_cam

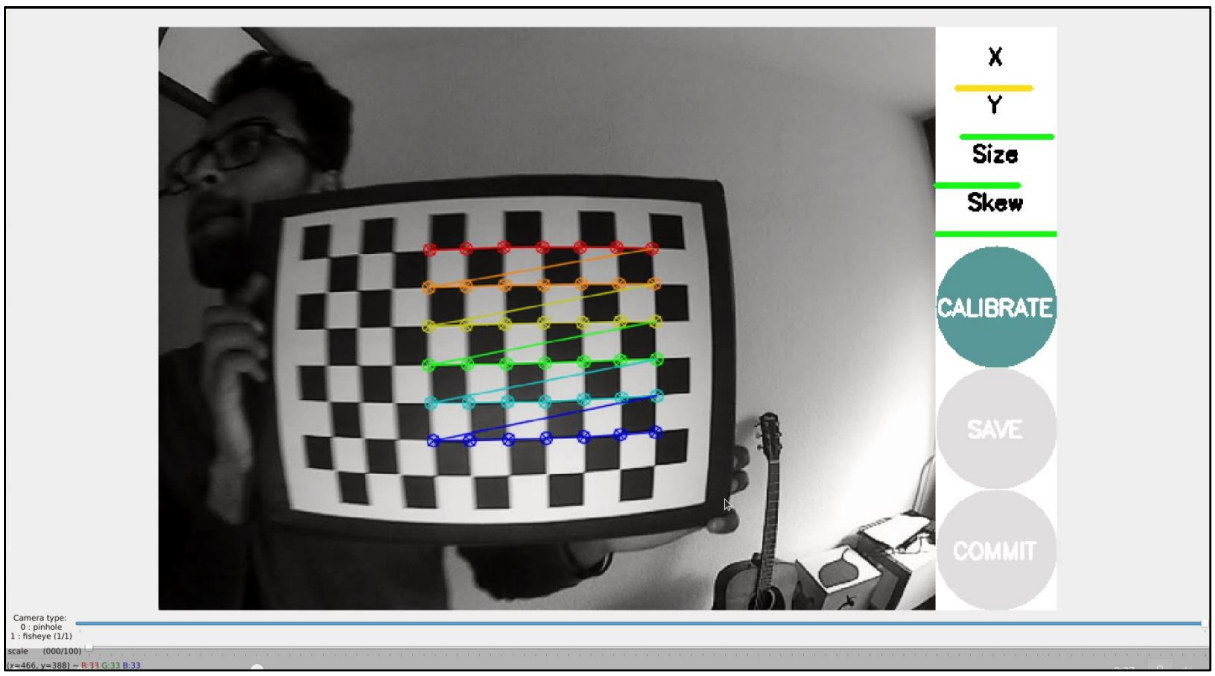

Figure 24 Camera Calibration.

cameracalibrator.py displays a calibration window and subscribes to ROS raw image topics. It may be used in monocular or stereo mode. The calibration window displays the most recent photos from the cameras, with the checkerboard highlighted. The node computes the camera calibration parameters when the user hits the CALIBRATE button. When the user selects COMMIT, the node uses a service call to provide these new calibration settings to the camera driver.

We have created the map of the environment with the help of USB_camera, as the robot camera has issues while creating the Map. The issue is discussed later in detail. We launch the ORB SLAM launch file,

roslaunch orb_slam2_ros usb_cam.launch
roslaunch orb_slam2_ros robot_ohmni.launch

In the terminal window we launch rviz to visualise the ORB SLAM image and 3D point clouds formed during the SLAM. (source O. , ROS Wiki documentation open source, n.d.)

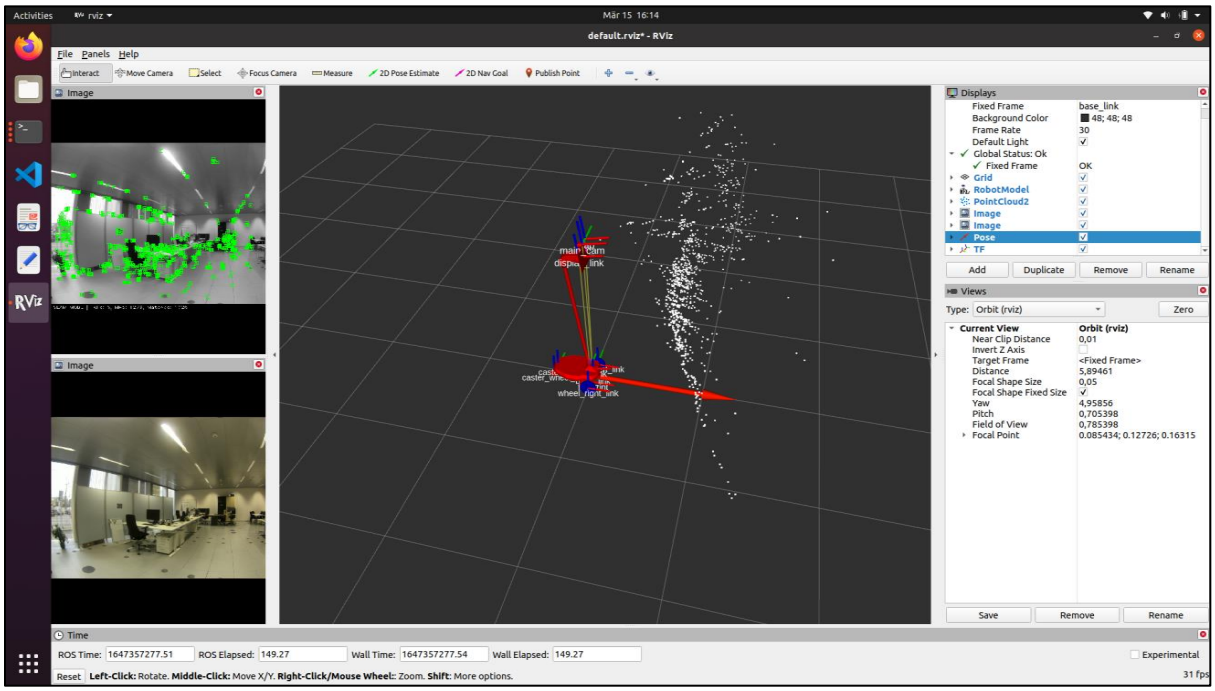

Figure 25 3D cloud points, ORB SLAM implementation with Butler Robot camera.

To save the map created by the ORB SLAM in binary format, run the command in the terminal,

rosservice call /orb_slam2_mono/save_map map.bin

The map is saved at the /.ros directory.

We can also load the saved map, by setting the load_map parameter in the launch file as “true”.

Movement of the Robot can be controlled by the python script. We have to run the below node for it:

rosrun orb_slam2_ros tb_cmd_vel.py

Note: The docker container/image (tb_control) should be running so, we can control the robot with the help of A, S, D and W keys of the keyboard.

We are able to get the 3D cloud point data from ORB SLAM. Octomap_server is used where it subscribes to the 3D point cloud topic and publishes /Occupancygrid and /projected_map. The

occupancy grid and projected map are important ros topics for map_server for creation of the map in .pgm format. We also get a .yaml file where map parameters are defined.

roslaunch octomap_server by_me_octomap_mapping.launch

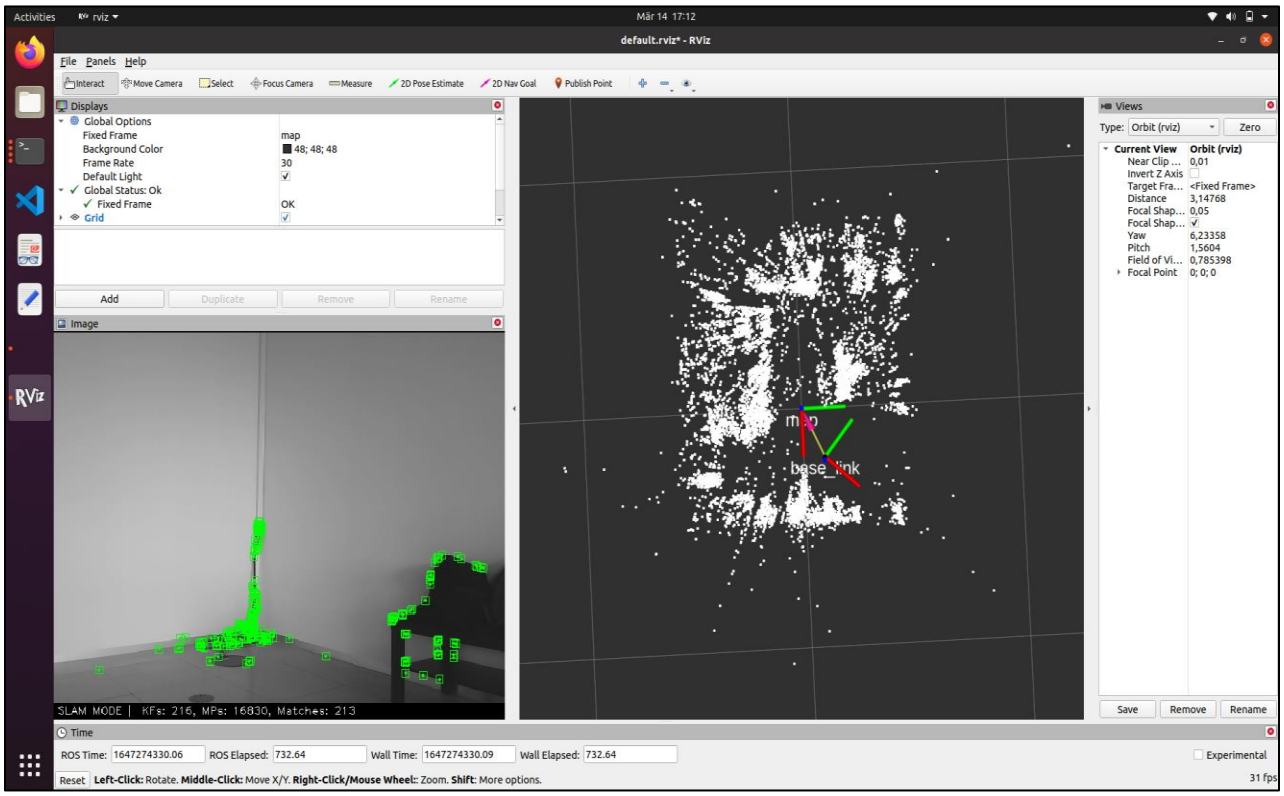

Figure 26 3D cloud points ORB SLAM with USB camera.

We can also save this map with help of Octomap_server with help of below command,

rosrun octomap_server octomap_saver -f my_map.bt

To load the map and visualize it on Rviz we have to run the below node

rosrun octomap_server octomap_server_node my_map.bt

We now run the node for the map_server to save the map into 2D, the occupancy grid and projected map makes the 3D point into 2D by projected it on the base plane. (source O. , map_server - ROS Wiki, n.d.)

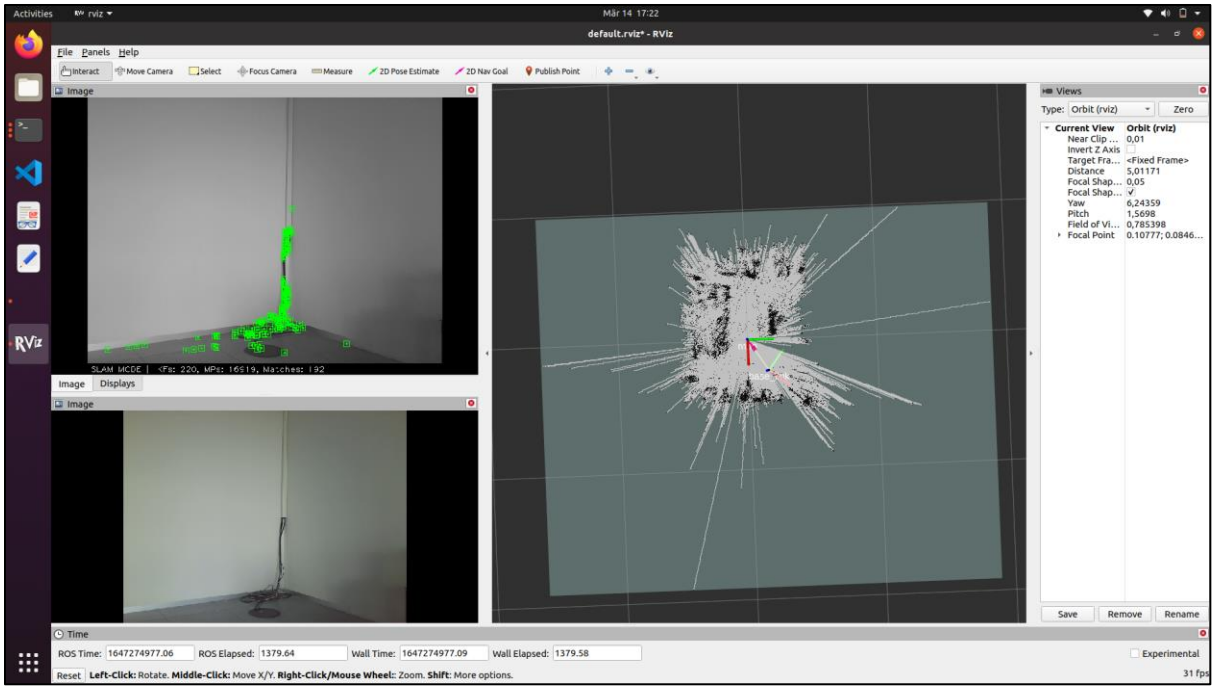

Figure 27 2D map created with octo_map_server.

rosrun map_server map_saver --occ 90 --free 10 -f map_name map:=/projected_map

We get the .pgm and .yaml file of the map after successful execution of this node.

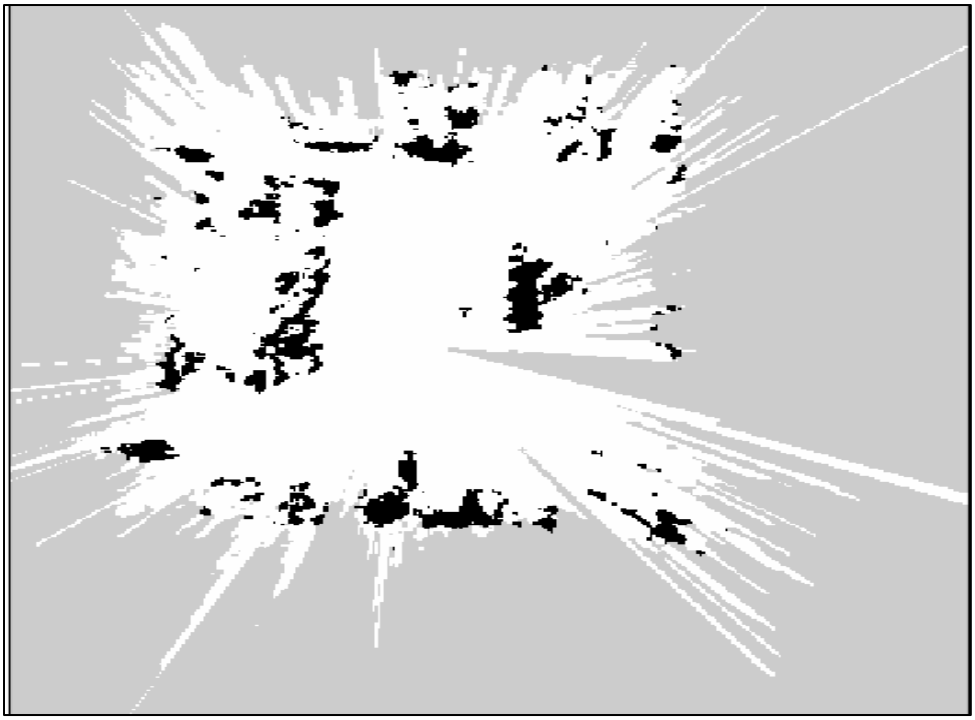

Figure 28 2D map with map_server package.

We have applied a median filter on the .pgm file of the map, just run the filter.py python script and we will get the smooth image with less noise. (source O. , map_server - ROS Wiki, n.d.)

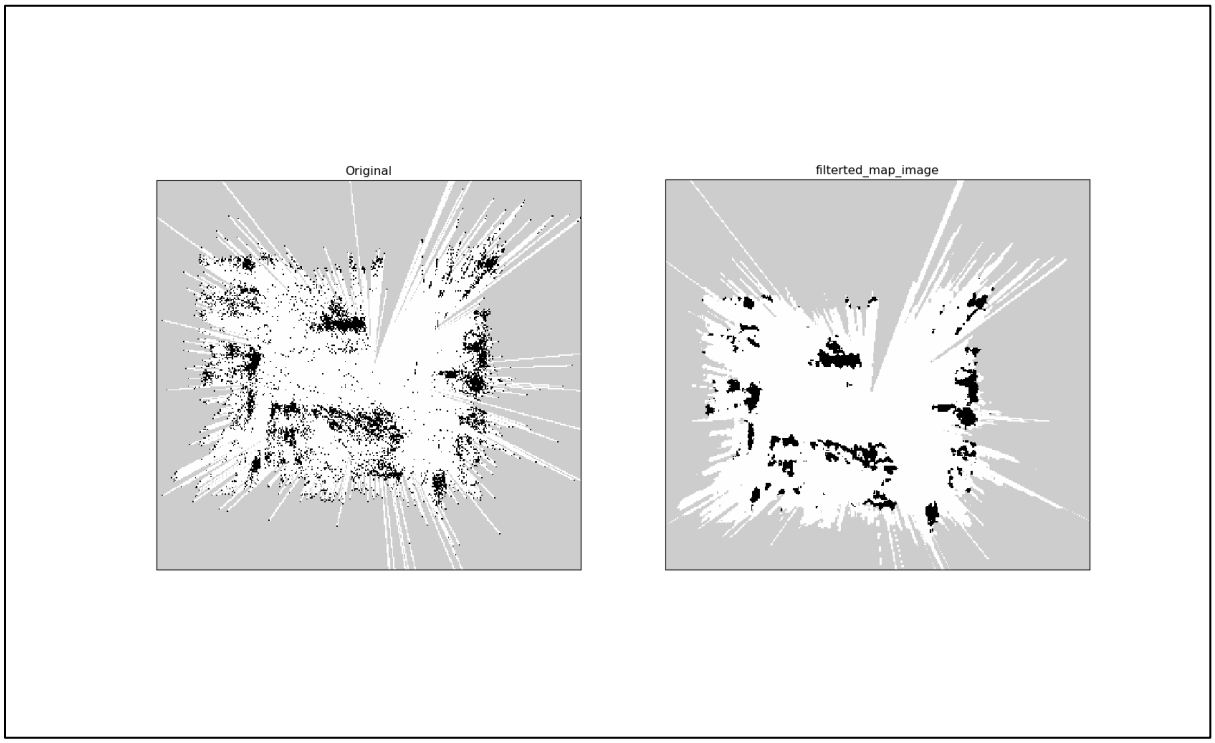

Figure 29 Median filter applied on 2D map to get denoised map.

The rqt_graph (relation between nodes - subscribing and publishing) (source O. , ROS Wiki open source)

rosrun rqt_graph rqt_graph

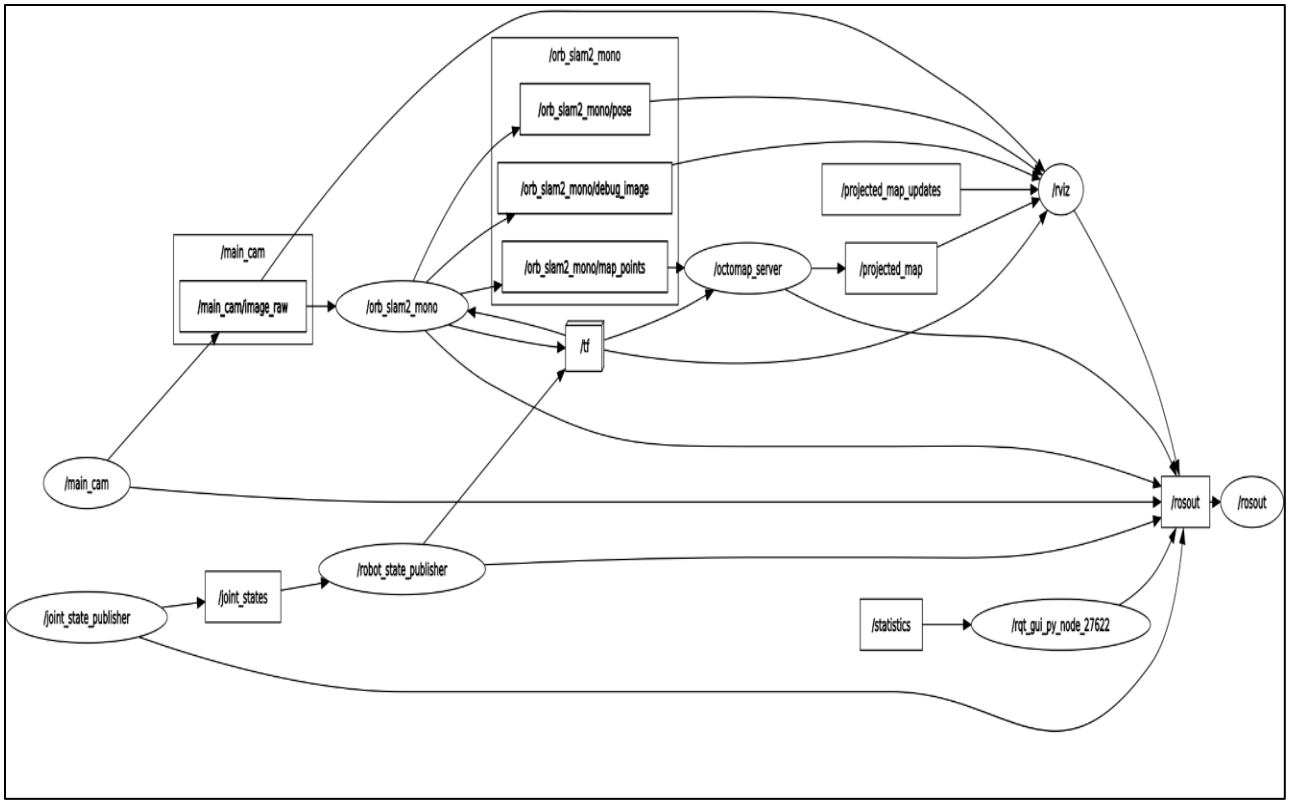

Figure 30 rqt graph with robot URDF.

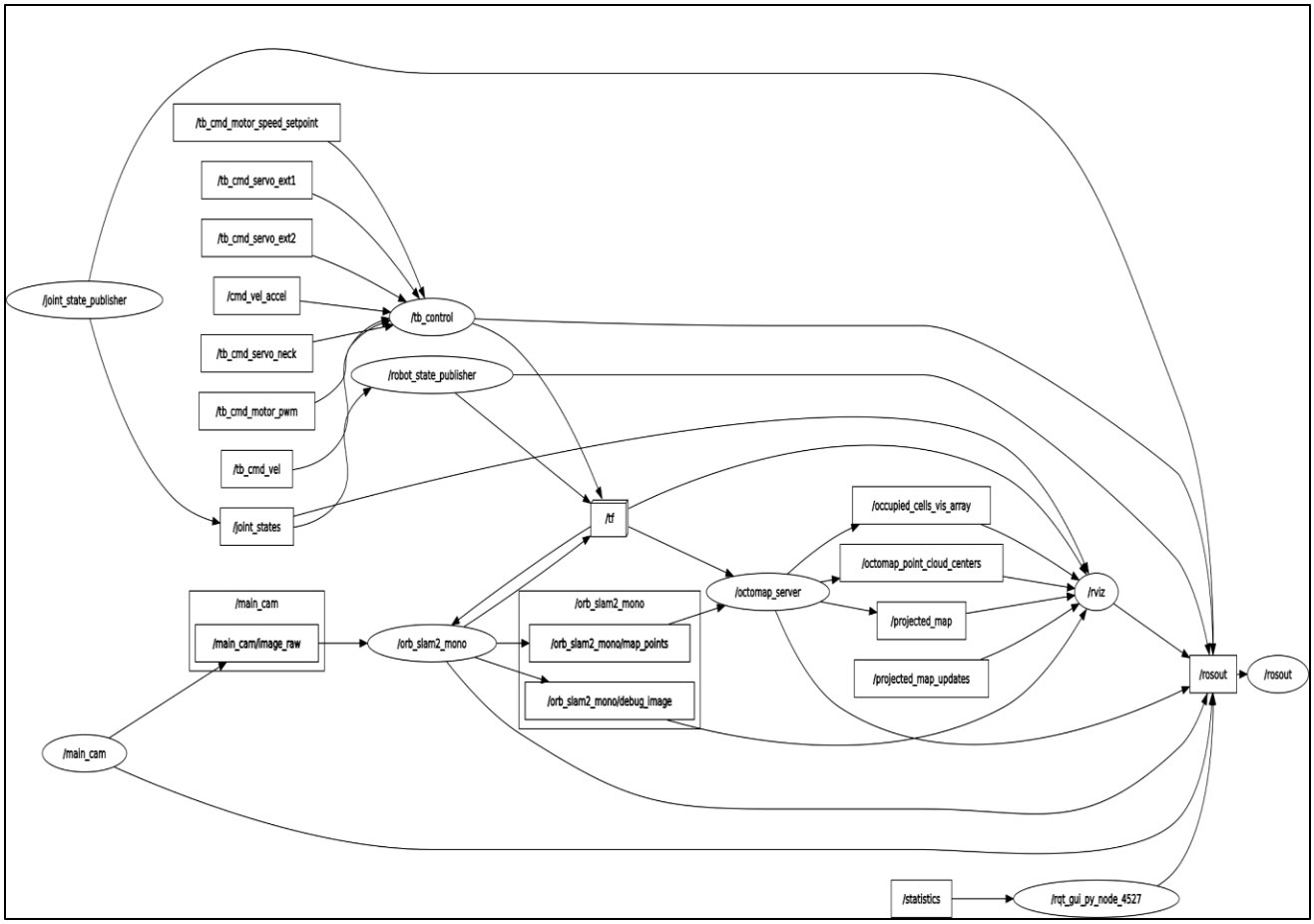

Figure 31 rqt graph with robot tf's.

We get the transform frame tree with the rosrun command:

rosrun tf2_tools view_frames.py

1. USB camera.

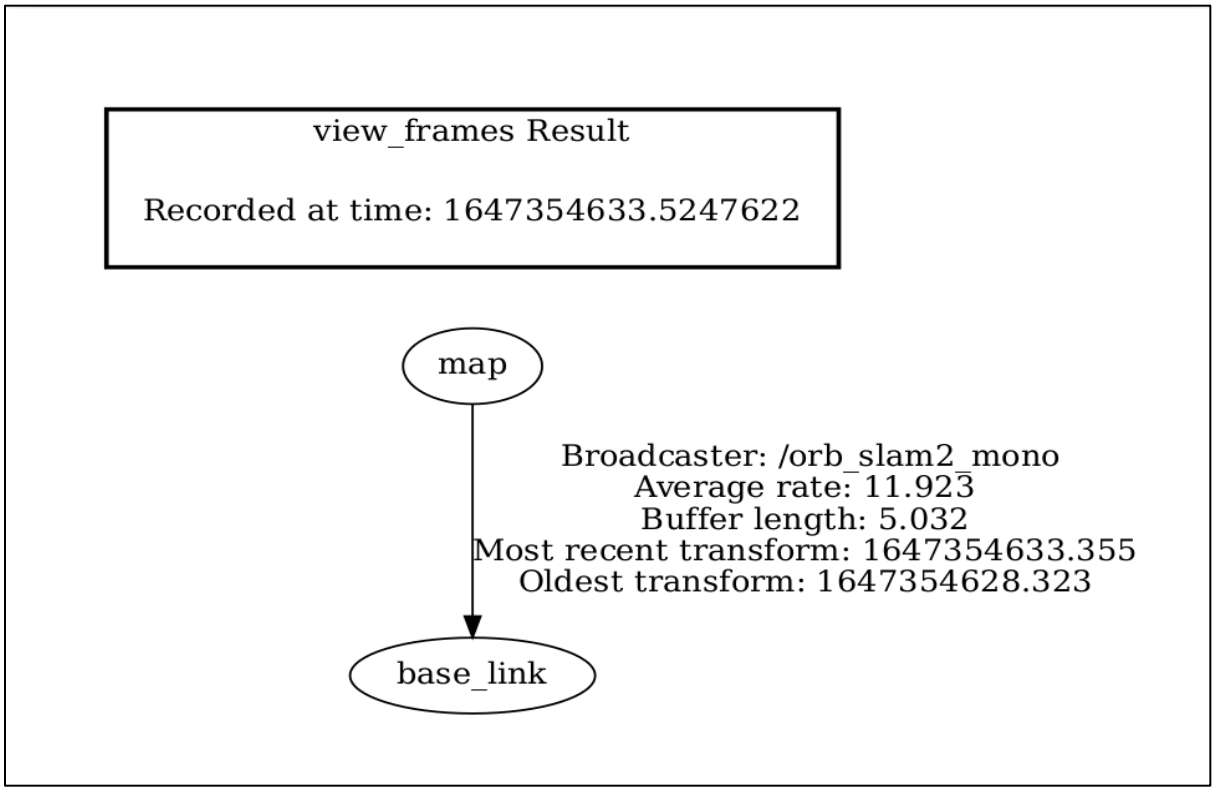

Figure 32 transform frame for USB cam ORB SLAM.

2. Robot camera

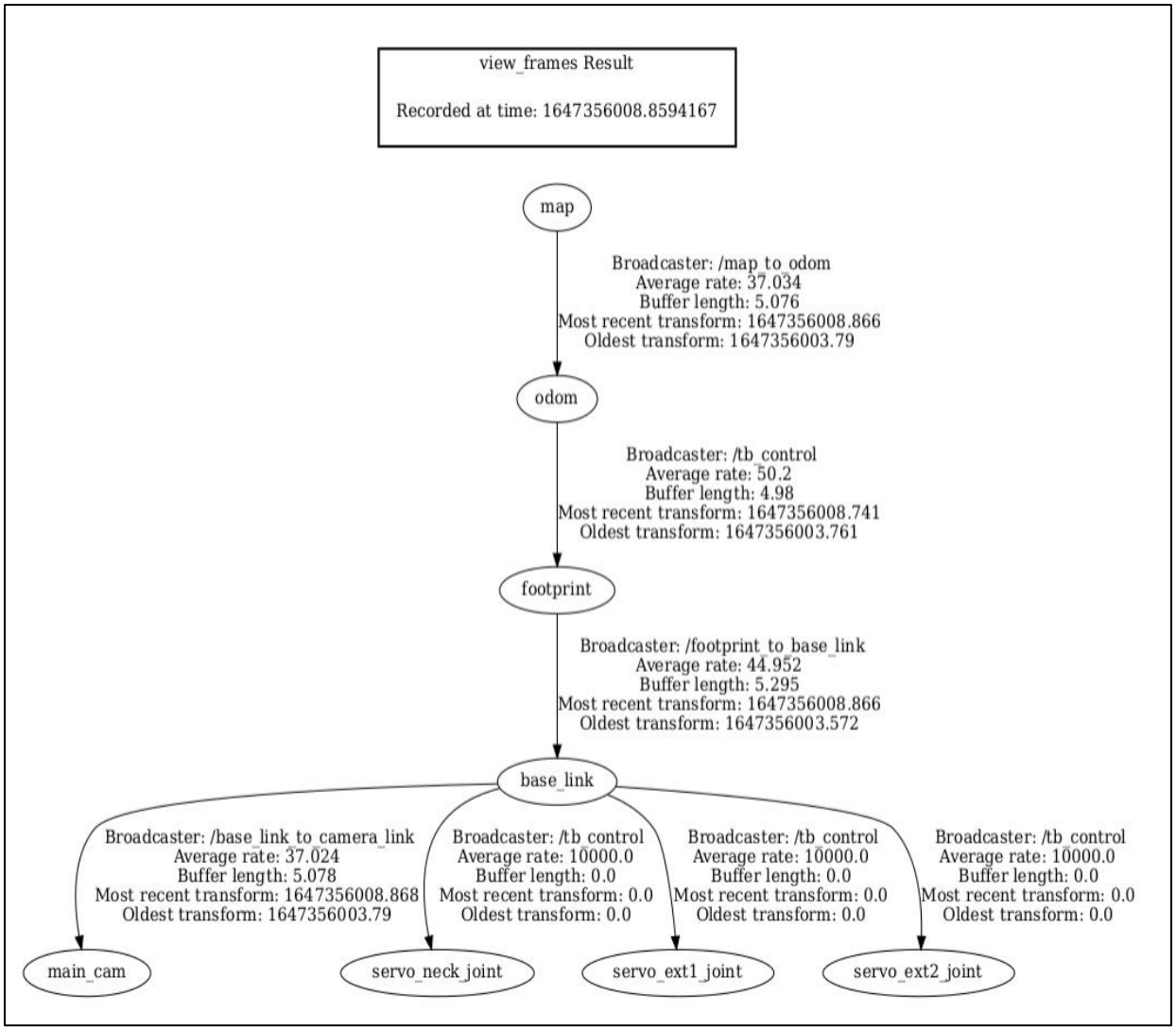

Figure 33 transform frame with robot camera - tb_control.

3. Robot Camera with URDF

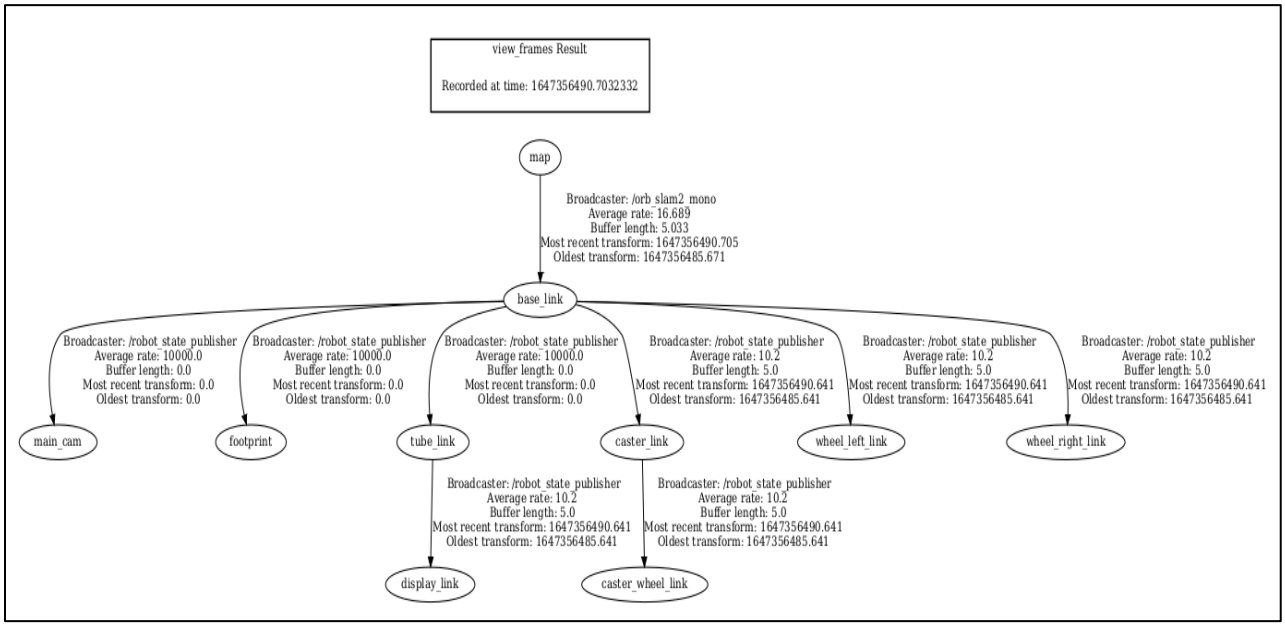

Figure 34 transform frame with URDF of Robot.

Comparison between VSLAM and Lidar SLAM

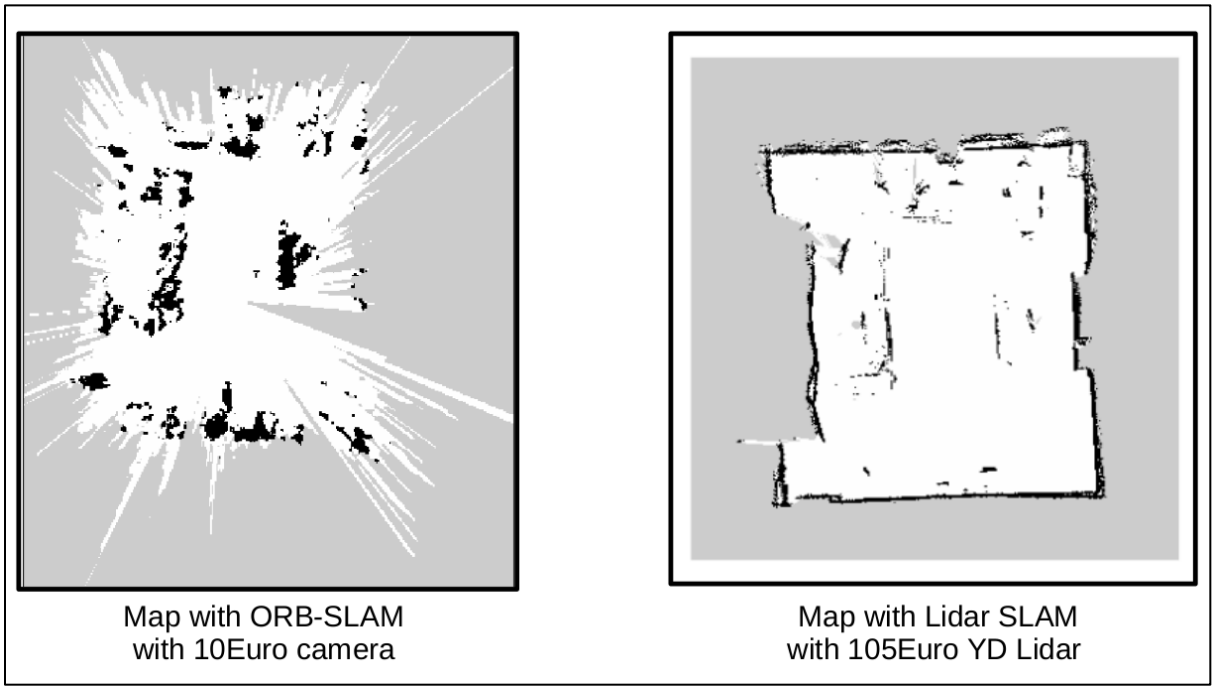

Figure 35 VSLAM vs Lidar SLAM.

5.3 ORB SLAM Issue

We observed, whenever there is a static transform between base_link and main_ cam link, the creation of the map is not processed properly. The 3D points accumulate into one region; it does not depend upon the motion of the robot.

We tried six different ways to solve this issue but none of them gave us the desired result. The outcome of all solutions was the same.

We have created two launch file for the Robot:

i) with tb_control tf's

roslaunch orb_slam2_ros robot_ohmni.launch

ii) with URDF of Robot.

roslaunch orb_slam2_ros urdf_ohmni.launch

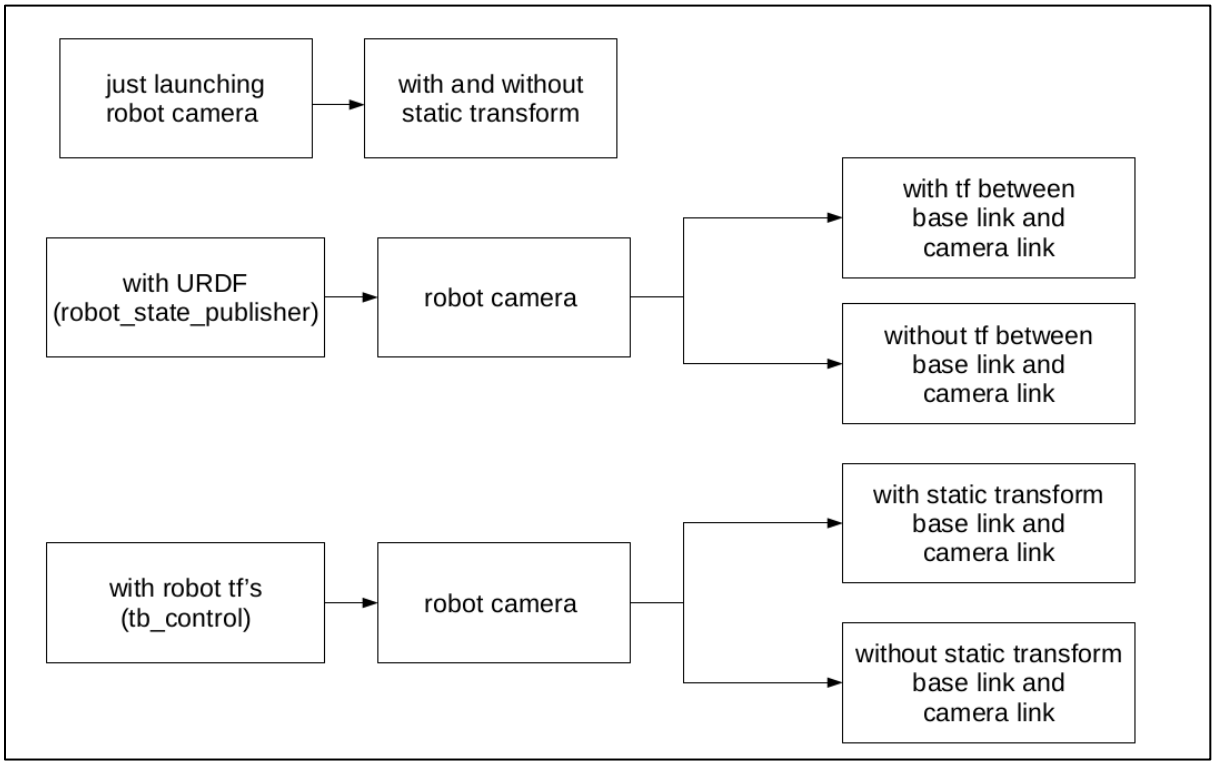

Figure 36 Implemented solutions for issue.

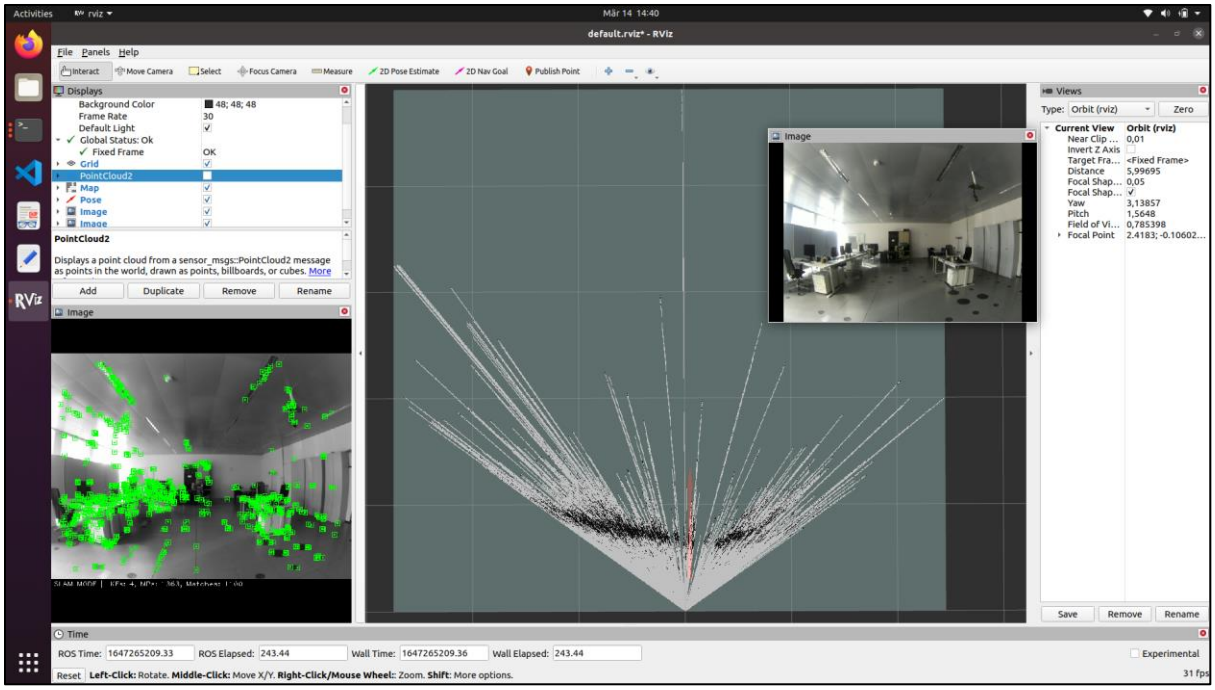

Figure 37 Issue of accumulation of 3D points in one plane.

5.4 YOLOv3 Object Detection

The algorithm uses GPU for its processing, so the local system must have CUDA installed. Use the below command into the terminal for launching the launch file.

Note: The docker RGB image/container must be running.

roslaunch darknet_ros yolo_v3.launch

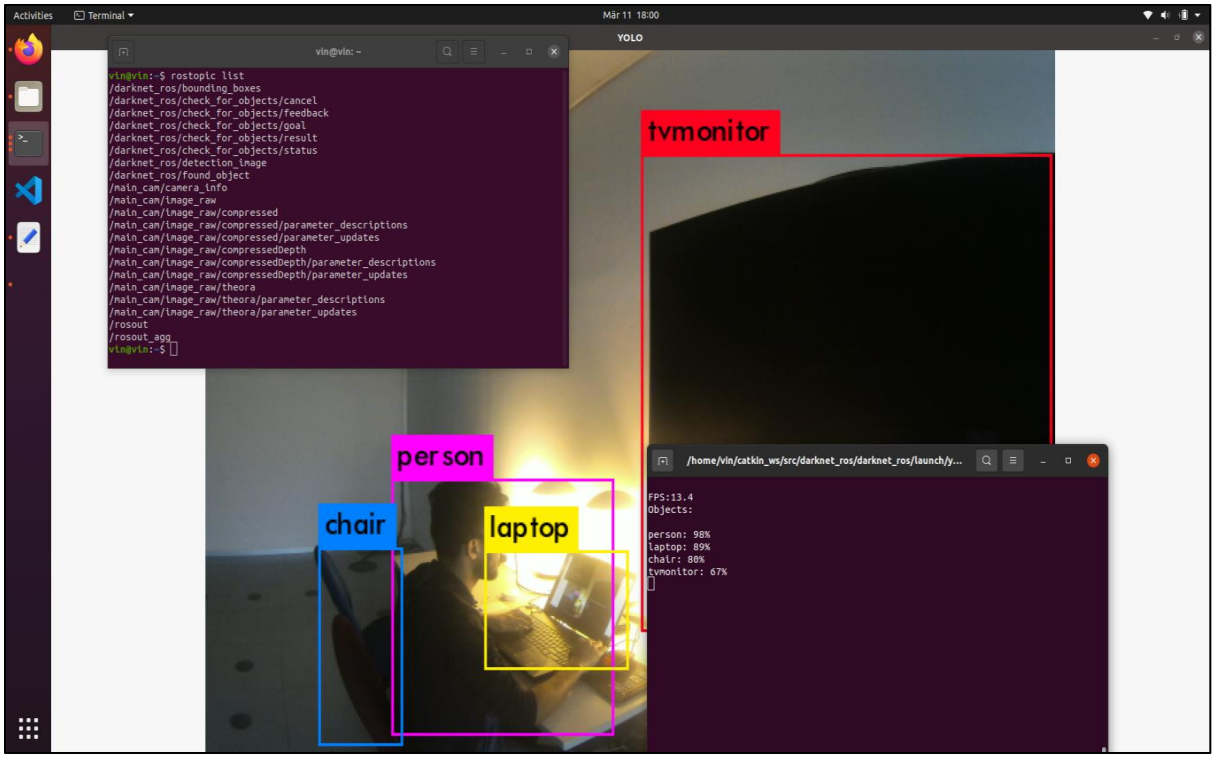

Figure 38 YOLOv3 output.

At the output we get the detected objects with their confidence.

5.4 Face Recognition

This algorithm uses GPU for its processing so CUDA has to be installed on the local system. In runtime itself we are able to give an image a label and it hardly takes 2 minutes to do it. We have to launch the ros launch file to get the desired output window,

roslaunch people_face_identification face_identification.launch

The algorithm will check if any face belongs to in its labelled database. If yes, it will give the ID of the person at the output window, otherwise it will assign a random ID and the image will get stored into an unlabelled dataset.

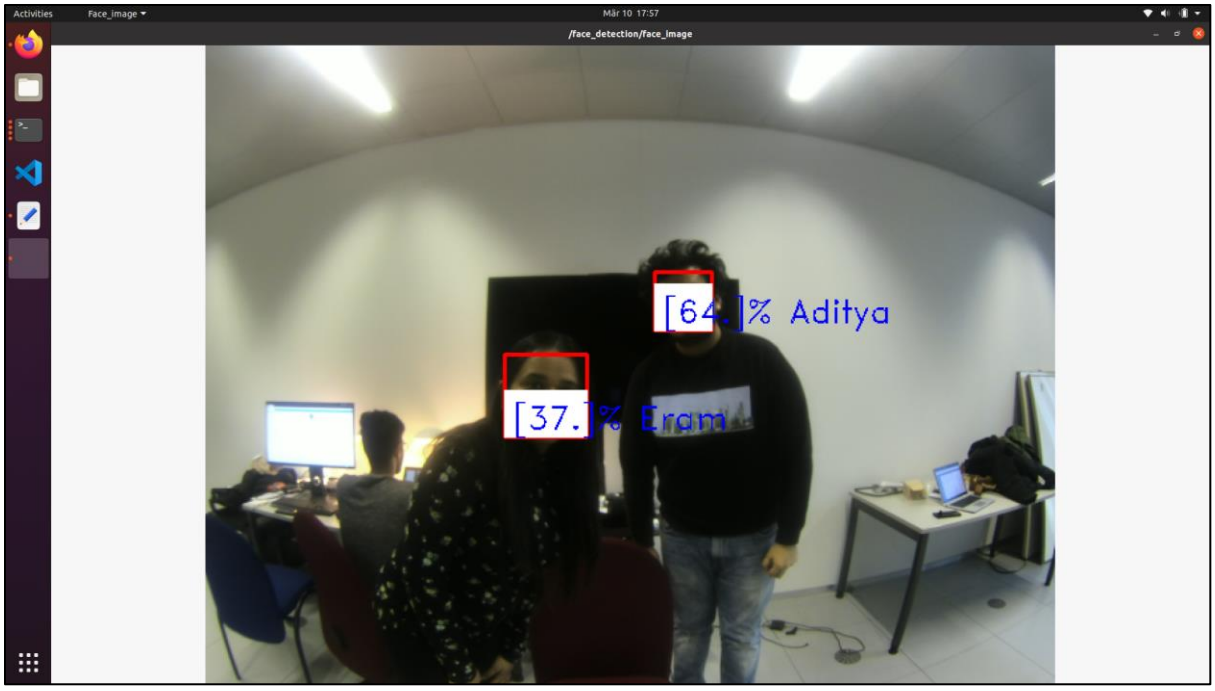

Figure 39 Face recognition output.

If we want to stop the auto learn process (the algorithm wont give us the ID for new person face rather it will display as Unknown in the output window) call the rosservice as mentioned below,

rosservice call /toogle_auto_learn_face false

If we want to start giving ID to new detected faces we have to make this rosservice call to “true” value

rosservice call /toogle_auto_learn_face true

5.5 Results for Object Detection and Face recognition.

We observed 100 samples to calculate the lag/latency in the frames received by the Butler Robot to the local system.

	Number of samples	Latency/Lagging (min-max)	Average value
Object detection	100	690ms-780ms	731ms
Face recognition	100	850ms-940ms	886ms

Figure 40 Latency.

5.6 Aux Camera

We can access the Aux camera of the Butler robot by launching the auxcam_v4l2.launch file in the RGB image/container.

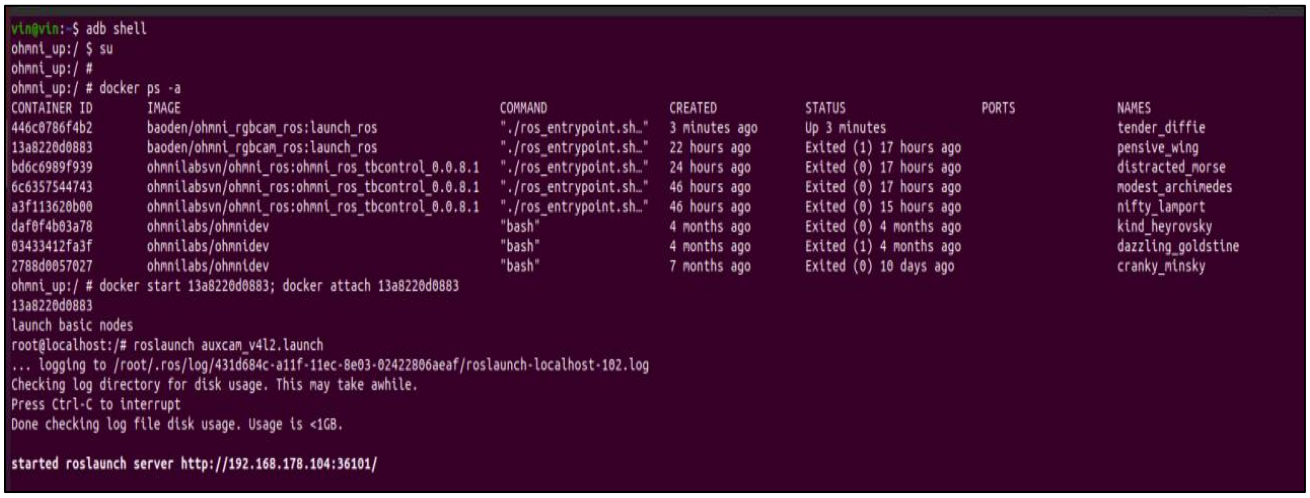

Figure 41 Launch File for Aux Camera of Butler Robot.

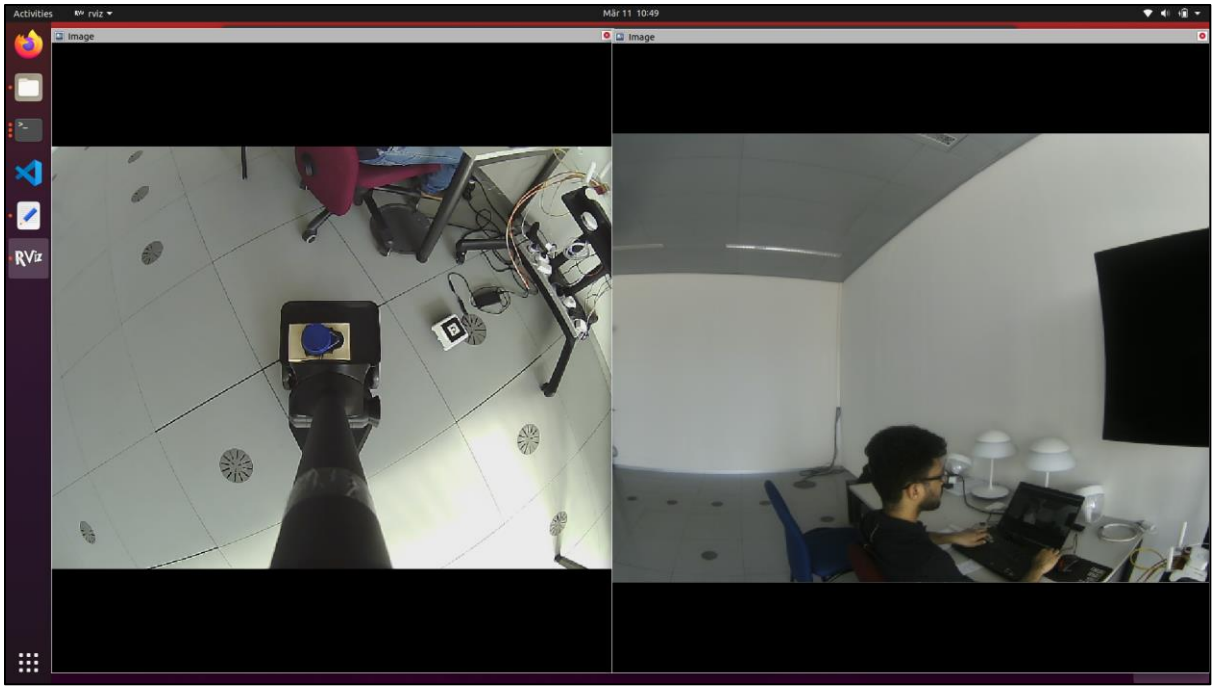

Figure 42 Output with aux and main camera of Butler robot.

6. Conclusion and Future Scope

In this research project, we have presented a low-cost solution to address the SLAM problem for creating a map of the environment in 2D with the help of ORB SLAM using a monocular camera. The project discusses the technical aspects of robot mapping and localization in an unknown environment using ROS Master and Rviz for visualization as a tool. We compared the results of VSLAM and Lidar SLAM and concluded that it's better to use a camera for SLAM for the indoor environment where there is no drastic change in the environment. We can get a good accuracy for creating maps of the environment by using an RGB Depth camera.

We have implemented the YOLOv3 algorithm for the detection of 80 different objects (as per COCO dataset) and the delay/latency we got for the image frame is 731ms. The accuracy of the algorithm is pretty good and it can be used for a real-time system where the deadline is not violated. Face recognition was another objective where it was successfully implemented. It used machine learning algorithms to detect the person's face and the accuracy of this algorithm is good. We can label the unknown face in run time which hardly takes two minutes. The lag/delay we got for the image frame on our local system is 886ms.

Further, we would like to extend this project by building a Navigation stack for the ORB-SLAM where we can have autonomous movement of the robot. A depth camera for ORB SLAM can be used for creating a map of the environment. Also, another VSLAM technique which is RTAB can be implemented where it uses both Lidar and Camera for SLAM process and Navigation. The SLAM can be implemented as Lidar is already integrated with the Butler Robot. Higher versions

of the YOLO algorithm can be implemented to get more accuracy and good performance for processing time. As Raspberry Pi is integrated with the robot, we can have more sensors and actuators to have more applications.

References

Geitgey, A. o. (n.d.). Retrieved from https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78.

ORB-SLAM2, R. M.-A. (n.d.). Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras.

pjreddie. (n.d.). *YOLOv3 Paper.* Retrieved from https://pjreddie.com/media/files/papers/YOLOv3.pdf.

Raúl Mur-Artal, J. M. (2015). *ORB-SLAM: A Versatile and Accurate Monocular SLAM System.*

recognition, O. s. (n.d.). Retrieved from https://pypi.org/project/face-recognition/.

source, M. o. (n.d.). *Matlab open source*. Retrieved from https://www.mathworks.com/help/vision/ug/camera-calibration.html.

source, O. (n.d.). *camera calibration – ROS Wiki*. Retrieved from http://wiki.ros.org/camera_calibration.

source, O. (n.d.). *COCO data set paper* . Retrieved from https://cocodataset.org/.

source, O. (n.d.). *map_server - ROS Wiki*. Retrieved from http://wiki.ros.org/map_server.

source, O. (n.d.). *octomap_server - ROS Wiki*. Retrieved from http://wiki.ros.org/octomap_server.

source, O. (n.d.). *ROS Wiki documentation open source*. Retrieved from http://wiki.ros.org/Documentation.

source, O. (n.d.). *ROS Wiki open source.* Retrieved from http://wiki.ros.org/.

Vincze, D. K. (n.d.). *Vision for Robotics.*

[1] “Vision for Robotics book by Danica Kragic1 and Markus Vincze” https://www.researchgate.net/publication/220666540_Vision_for_Robotics (accessed November 15th, 2021).

[2] [Monocular] Raúl Mur-Artal, J. M. M. Montiel and Juan D. Tardós. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, 2015. (2015 IEEE Transactions on Robotics Best Paper Award).

[3] [Stereo and RGB-D] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics, vol. 33, no. 5, pp. 1255-1262, 2017.

[4] "octomap_server - ROS Wiki." http://wiki.ros.org/octomap_server (accessed October 10th, 2021).

[5] "map_server - ROS Wiki." http://wiki.ros.org/map_server (accessed January 10th, 2021).

[6] "camera calibration – ROS Wiki." http://wiki.ros.org/camera_calibration (accessed October 20th, 2021)

[7] "Matlab open source" https://www.mathworks.com/help/vision/ug/camera-calibration.html (accessed October 20th, 2021)

[8] "ROS Wiki documentation open source" http://wiki.ros.org/Documentation (accessed November 3rd, 2021)

[9] "YOLOv3 Paper" https://pjreddie.com/media/files/papers/YOLOv3.pdf (accessed December 15th, 2021)

[10] "COCO data set paper" https://cocodataset.org/ (accessed 19th December, 2021)

[11] "Open source dlib library Face recognition" https://pypi.org/project/face-recognition/ (accessed January 20th, 2022)

[12] "Article on face recognition by Adam Geitgey" https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78 (accessed 21st January, 2022)

[13] "ROS Wiki open source" http://wiki.ros.org/ (accessed 15th October 2021)