EE492 Senior Project

Target Localization and Relative Navigation Using RGB-D on ROS

Mehmet Yaşar Alıcı

Department of Electrical and Electronic Engineering

Boğaziçi University

Bebek, Istanbul 34342

Project Advisor: H. Işıl Bozma

Evaluation Committee 1. Member: Yağmur Denizhan

Evaluation Committee 2. Member: Mehmet Akar

15.02.2021

to Johann Sebastian Bach,
whose tunes thoroughly accompanied this project.

Acknowledgements

Throughout the implementation of the project and writing of its report, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor H. Işıl Bozma, for her guidance through each stage of the process. Her expertise was invaluable in formulating the research questions and methodology in this project. Throughout, I enjoyed her sincerity, openness and care. She listened to my findings meticulously and was always interested in learning new technologies that I applied in the project. Moreover, I am grateful that she introducing me the research problems addressed at the Intelligent Systems Lab. Under her initiative, I had the joy to dive in to the insightful team discussions, collaborate with other lab members and test their work within my project.

Finally, I would like to thank the team members at Intelligent Systems Lab, Mirhan Urkmez, Meric Durukan, Kemal Bektas and Haluk Zorluoglu, for their support and collaboration.

Abstract

Object recognition is a hard problem to solve in robotics [10]. Recently, deep learning (DL) based approaches using vision data performed well in this domain [14]. Motivated by this, we propose a DL-based method for the object recognition problem on RGB-D cameras to Segway RMP220 robot on Gazebo. To realize our method, our project involves three stages. First, a simulation environment was prepared for the problem. Then, a neural network model and tracker were employed sequentially to detect and track an object based on the RGB-D input. Finally, our robot is tasked to navigate towards the detected object based on the object detection.

Contents

List of Figures

List of Tables

1 Introduction

For robots to successfully operate in real world, it is very crucial for them to perceive its environment correctly [19]. While humans can efficiently and effortlessly detect objects, it is still a challenging task for robots due to the complexity of the real world. Traditionally, this problem was addressed by researchers by algorithms based on images captured by RGB cameras [22, 11]. However, this technique was lacking depth information and therefore inadequate in a 3-dimensional world [16].

Recently, RGB cameras with depth sensors, called RGB-D cameras, have become widely available due to their low-cost [12]. Depth information obtained from these cameras are commonly used to construct depth maps or 3D point clouds. In depth maps, a distance value (Z) is stored with every pixel (X,Y). The distance is expressed in metric units (meters for example) and calculated from the back of the left eye of the camera to the scene object [1]. On the contrary to the depth maps, 3D point clouds represent a scene in direct (X,Y,Z) coordinates instead of adding a distance to each pixel.

Moreover, convolutional neural networks (CNN) have achieved great success in many fields, computer vision being the most prominent [23]. This is as a result of the emergence of large, high-quality, publicly accessible labelled datasets and the enablement of parallel GPU computing, which enables notable acceleration in model training [9, 17]. A CNN is made up of three different types of neural layers, namely (i) convolutional layers, (ii) pooling layers, and (iii) fully connected layers, as shown in Figure 1. When a CNN reaches its final fully connected layers, the input data is mapped to a 1D feature vector after each layer converts the input volume to an output volume of neuron activation [23].

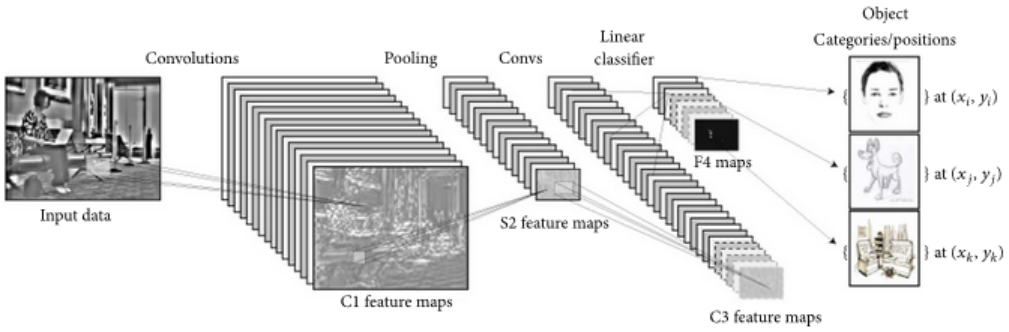


Figure 1: Example architecture of a CNN for a computer vision task (object detection) [23]

.

Deep learning approaches on RGB-D data have been successfully proposed for the problem of object detection. For example, in [20] authors converted RGB-D scans to 3D point clouds and proposed a neural network that operates on these points to detect objects. It was demonstrated that their method outperforms the state-of-the-art on KITTI and SUN RGB-D 3D detection benchmarks. However, object detection based on deep-learning in real-time suffers from slowness and, therefore, is not suitable for real-time context such as mobile robotics.

Recently, visual tracking methods have gained significant attention from literature [17]. Given initial object's coordinates, tracking methods are able to evaluate online the quality of the location of the target in the new frame, without ground truth. The algorithms are less computationally expensive than the deep learning based object detection methods, therefore, favorable against applying detection at each time frame. For example, in [8] authors proposed Multi Instance Learning (MIL) algorithm for object tracking and demonstrated that their algorithm can run at real-time speeds.

Various object detection and tracking algorithms are available in leading machine learning and image process-

ing libraries. For example, TensorFlow features an Object Detection API which contains various successful deep learning algorithms pre-trained on COCO 2017 Dataset [2]. Moreover, OpenCV implements various object tracking algorithms as a part of its Tracking API [3]. These implementations can be employed for application-oriented projects as "out-of-the-box" localization to dramatically speed up the development process.

In real world mobile robot scenarios, it is difficult to collect continuous vision information while the robots are in motion due to the unstable vision information [18]. An efficient method of obtaining visual information is using pan/tilt apparatus. It effectively photographs an object by adjusting its mounted camera to a desired location through controlling a pan movement unit for horizontal rotation and a tilt movement unit for vertical rotation [13] (See Figure 3). To locate a target at the image plane, an actuation module consists of pan/tilt motion servo motors with motor controllers to steer the camera.

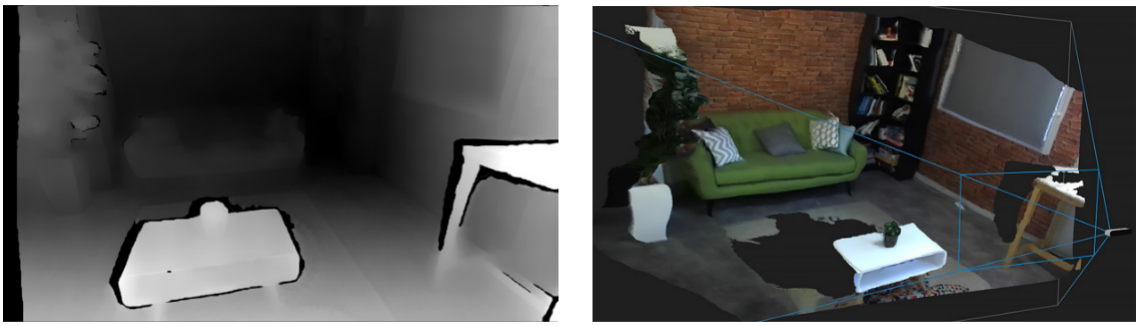


Figure 2: Depth information obtained by ZED Camera: (a) Depth Map, (b) 3D Point Cloud [1].

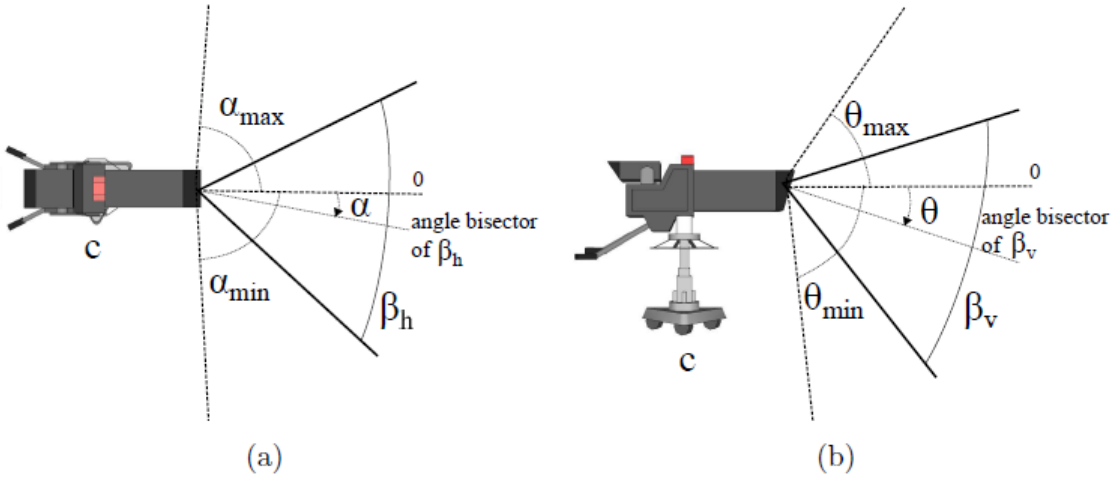


Figure 3: Modeling a camera: (a) pan (horizontal plane) , and (b) tilt (vertical plane) [24]

Last but not least, it is challenging to develop software to address high-level tasks, such as mapping or reasoning, on robotics. Since there are wildly warying hardware, one has to start from writing low-level device drivers up to high-level tasks [21]. To address these challenges, The Robot Operating System (ROS) was developed, giving the researchers the freedom to focus only on high-level areas. On the other hand, a robot simulator, Gazebo, was developed for ROS to mimick the real-world environments in a computer. However, while it eliminates the requirement for hardware and leads to fast experiments, one has to keep in mind that the real-world introduces much more complexity than simulation and successful simulations may suffer from catastrophic failures on transfer to real-world.

2 Problem Statement

The goal of this project is (i) localization of an object using RGB-D sensor data through detection and tracking and (ii) navigation relative to the object, i.e towards it, of the Segway robot on ROS Gazebo simulator.

3 Method

To solve our problem defined at Section 2, the project goes through three major stages. In the first stage, we get our simulation environment, robot and components, i.e. sensors, pan-tilt apparatus, up and running in Gazebo. Next, we employ a CNN model and MIL algorithm to detect and track an object, respectively, using the RGB-D sensor data. Based on the object localization, we develop a navigation framework where our Segway robot moves towards the object using APF algorithm.

3.1 Robot Bring-up on Gazebo

3.1.1 Introduction to Reference Model

Firstly, we were introduced to the robot and its essential components that we would bring up on our Gazebo through the reference design on Solidworks, which is observed in Figure 4.

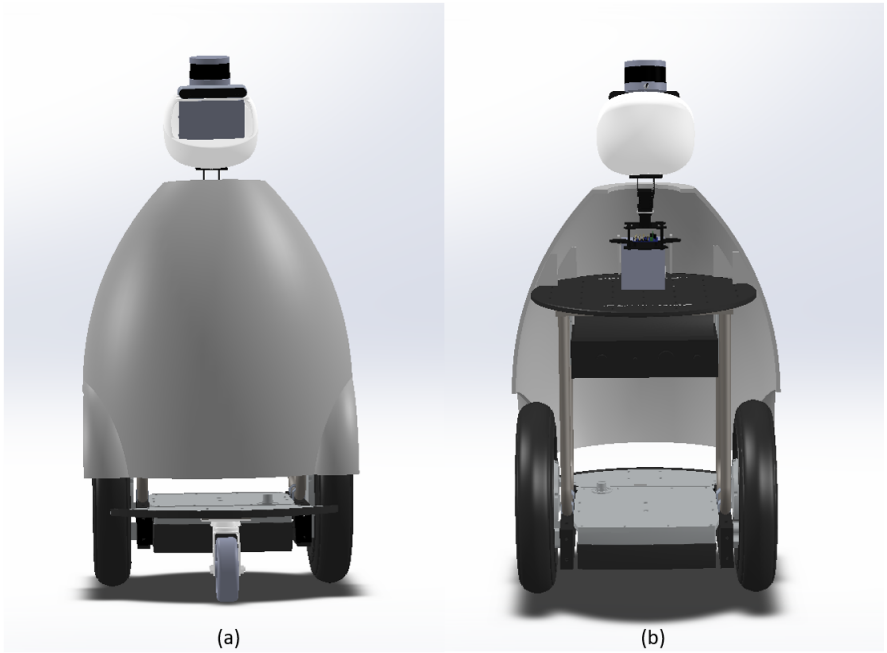


Figure 4: Reference Model of the Robot drawn in Solidworks: (a) Front view. (b) Rear view.

Then, we identified the main components of the reference design as observed from Figure 5. Our next goal was to bring-up these components one by one in Gazebo.

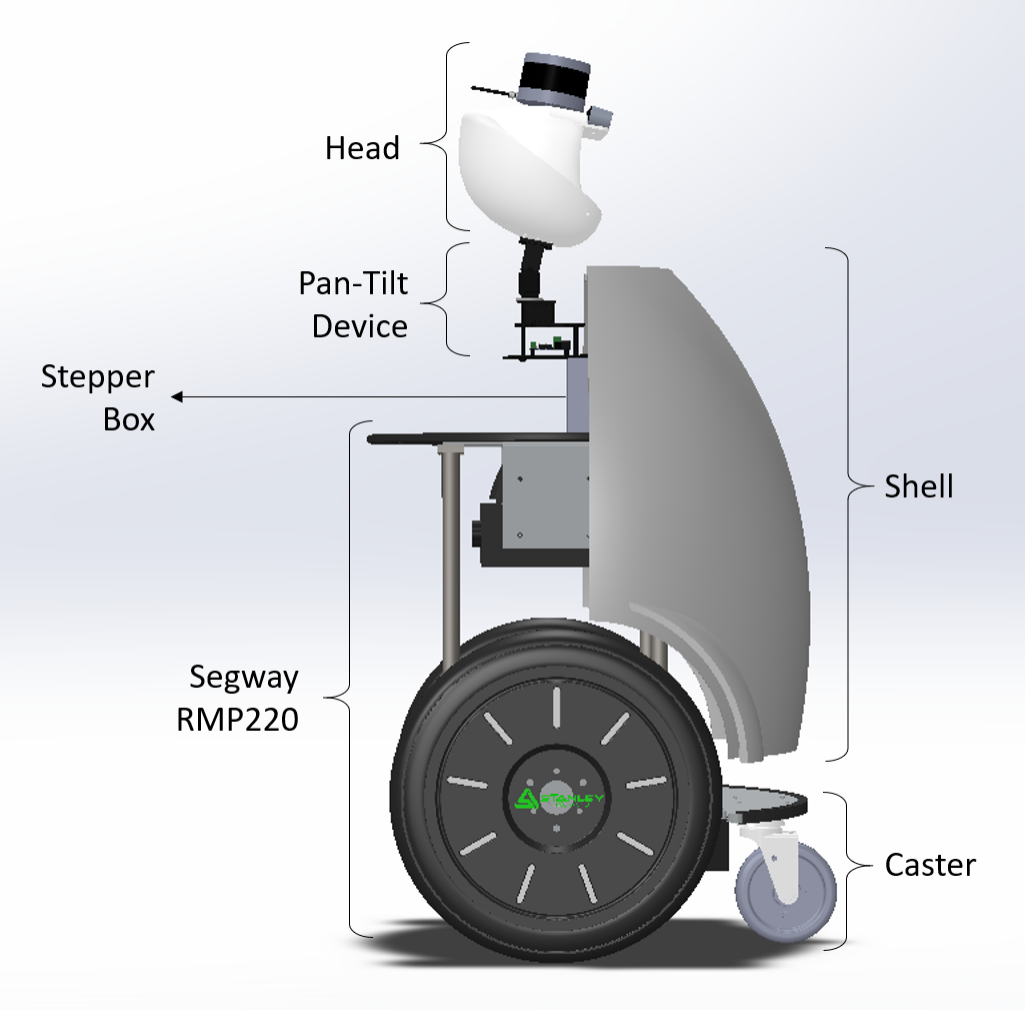


Figure 5: Reference model's side view and its *parts* in Solidworks.

3.1.2 Segway RMP220 Bring-up

We started off by bringing-up Segway RMP220 on Gazebo. We investigated the component's Github repositories offered by its manufacturer, which ideally should provide packages to visualize and control the component on Gazebo. Unfortunately, the packages we found were supporting only ROS version Indigo, which precedes our version Melodic by two releases. Therefore, after cloning the repositories to our workspace, a set of packages failed to build.

We removed the failing packages since we didn't need at that time. In the case of any need to those packages in future, it is possible to build these by building their ROS Indigo dependencies directly in the workspace.

Afterwards, we had to let the packages know which model of the component that we desire to use. In our case, it was the model RMP220; and we satisfied this by sourcing our model's bash file as,

```
cd {WORKSPACE_ROOT}/src/segway_v3/segway_v3_config/std_configs/
```

```
source segway_config_RMP_210.bash
```

Then, we tested the installation by running,

```
roslaunch segway_gazebo segway_empty_world.launch
```

After this command, we obtained the RMP220 as observed in Figure 6 (a). Moreover, to test navigation, we run,

```
rostopic pub -r 10  /segway/navigation/cmd_vel \
geometry_msgs/Twist \
"linear:
  x: -1.5
  y: -1.5
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0"
```

As a result of this command, we successfully moved the RMP220, as observed in Figure 6 (b).

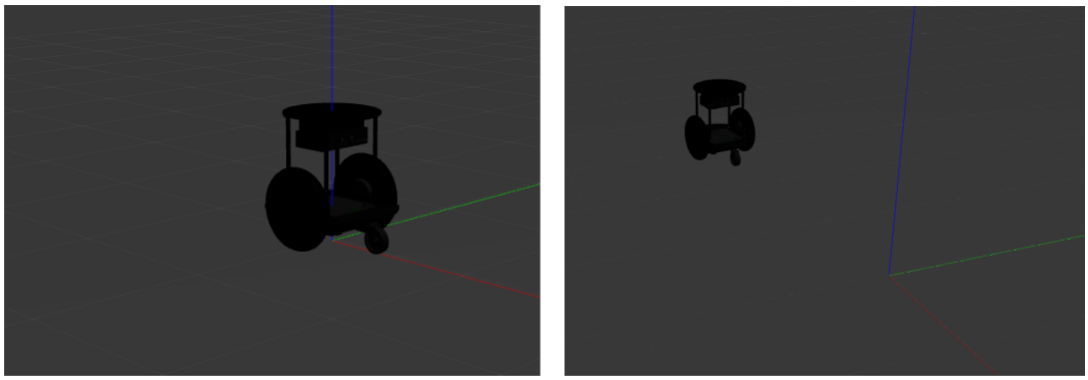


Figure 6: Segway RMP220 Bring-up on Gazebo: (a) Spawn, (b) Navigation

Last but not least, we have documented our bring-up and workarounds for failing packages for ROS Melodic publicly on Github to facilitate other researchers' work. The documentations are available in Appendices B, C to this paper.

3.1.3 Widowx Turret Bring-up

After bringing-up Segway RMP220, we moved to get our pan-tilt device WidowX Turret up and functional in Gazebo. We started by cloning the manufacturer's Github repository to our workspace. Unfortunately, we realized that the manufacturer did not provide a Gazebo package for simulation and instead provided an RViz package, which is another simulation software to be used within ROS.

Upon an unfruitful Github search to find a Gazebo package for WidowX Turret, we have decided to create our own. Toward this end, we had to fulfill robot's compatibility and functionality on Gazebo. To investigate

compatibility, we checked whether the robot description fulfills the Gazebo requirements described in [4]. We were lucky that it already did. Therefore, creating a *widowx_turret_gazebo* package with the launch file *widowx_turret_empty_world.launch* to spawn the robot, we run,

```
roslaunch widowx_turret_gazebo widowx_turret_empty_world.launch
```

and obtained the device on Gazebo as observed in Figure 7 (a).

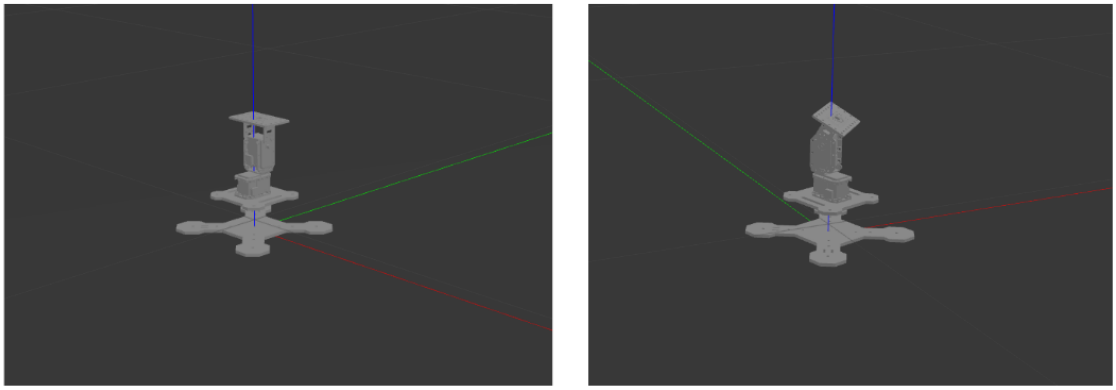


Figure 7: WidowX-Turret Bring-up on Gazebo: (a) Spawn (b) Control of Joints

Now, our purpose was to gain functionality, i.e ability to control pan and tilt joints, to WidowX Turret. To this end, we first created a PID Controller with the gains, $K_p = 20, K_d = 0.1, K_i = 1$ for both joints. Reference input $r(t)$ and system's output $y(t)$ are the angles that the joint makes with the normal to the axis of rotation. On the other hand, the controller's aim was to remove the steady-state tracking error, i.e make the $r(t) - y(t) = 0$ as $t \to \infty$.

After creating controllers, we added ROS Gazebo Control plugins to the robot's description following the tutorial in [5] to associate controllers with the joints. After spawning the component, we run,

```
rostopic pub /pan_controller/command std_msgs/Float64 \
"data: -0.5"
rostopic pub /tilt_controller/command std_msgs/Float64 \
"data: -0.5"
```

to rotate the tilt and pan controllers -0.5 rads each. As a result, we obtained the Figure 7 (b). As in the case of Segway RMP220, we documented our Gazebo bring-up publicly on Github for other researchers and it is also available under Appendix D to our paper.

3.1.4 Bringing up Other Components

Next, our focus was to bring up uncontrollable components in Gazebo. For this purpose, we used a third-party SW2URDF add-in to Solidworks to migrate those components from the reference design in Solidworks to Gazebo.

We defined the reference axes and coordinate systems for links following the tutorial [6]. Then, we run SW2URDF and specified the links and joints manually. The add-in generated ROS packages which involves all files to visualize the exported components in Gazebo.

We obtained the whole robot via this technique, however, our strategy was replacing the generated components that we wanted to control, i.e RMP220 and Pan-Tilt, with those we have already brought up in Gazebo. Those components are based on the packages by their manufacturers, therefore, the replacement would give us the opportunity to make our robot compatible with them.

3.1.5 RGB-D and LIDAR Sensors Bring-up

We use the model ZED by Stereolabs as RGB-D camera and model Puck by Velodyne as LIDAR sensor as described in the reference design. Even though we will not use LIDAR in our project, we decided to bring it up as well for future works.

We started by investigating publicly available URDF models of the sensors. Luckily, the manufacturers provide the URDF models in their Github repositories. We obtain them and incorporate to our project without much effort. However, we had to add Gazebo sensor plug-ins to the URDFs of the sensors so that they publish their sensor data obtained from Gazebo to ROS. Toward this end, we followed the tutorial in [7] and equipped the URDFs with Gazebo plug-ins.

To test our work, we have placed a few objects in front of the sensors and run the RViz tool which visualizes ROS topics. In RViz, we selected the topics, `/zed/color/image` and `/zed/depth/image` for RGB and depth map, respectively. As a result we obtained Figure 8. Moreover, selecting `/velodyne_points` topic for LIDAR, we obtained Figure 9. With these results, we successfully brought up RGB-D and LIDAR Sensors.

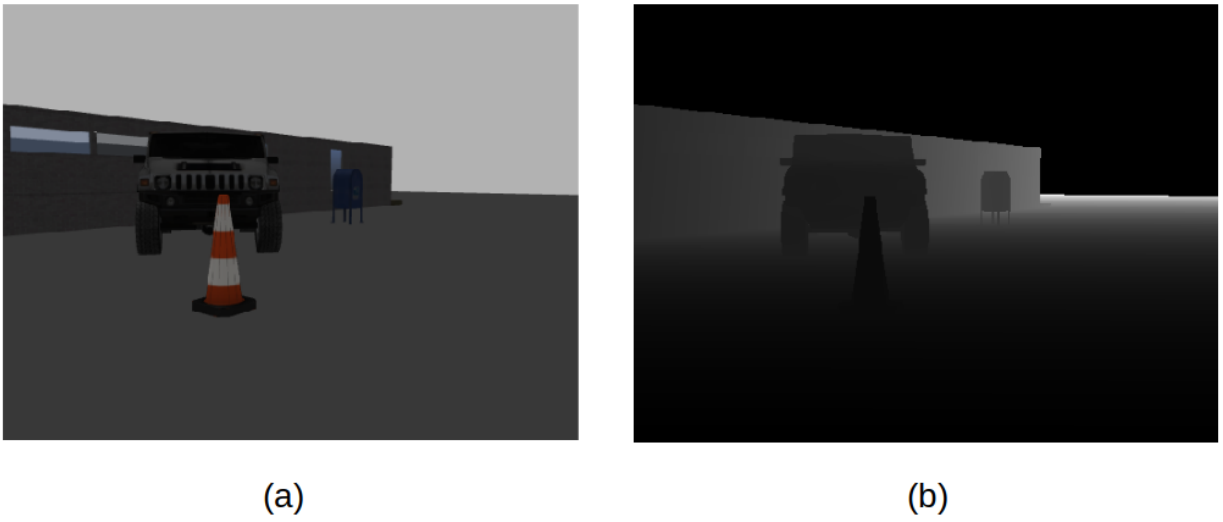


Figure 8: RGB-D Bring-up: (a) RGB image, (b) Corresponding depth map.

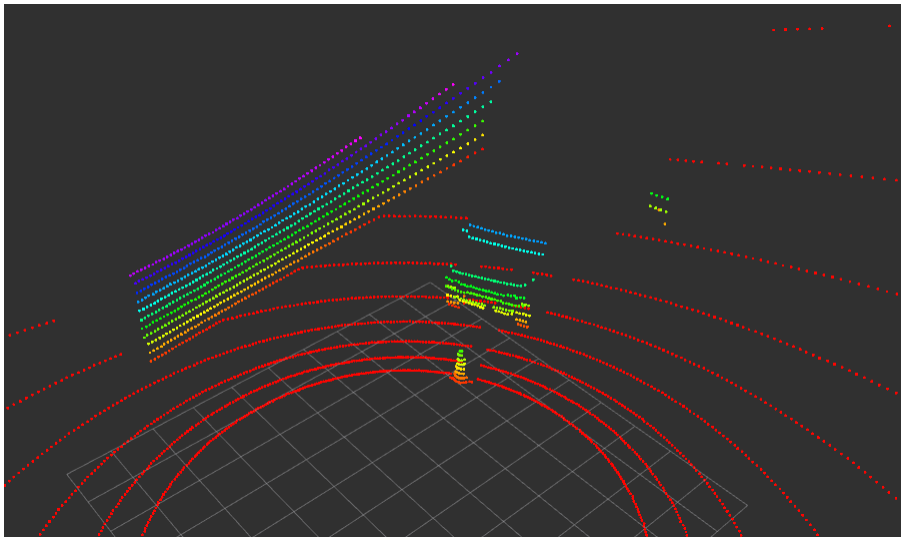


Figure 9: LIDAR Bring-up.

3.1.6 System Integration

Having brought up all components individually, our purpose was now to integrate them together to obtain a fully functional robot.

We started off integrating our robot by selecting RMP220's URDF as the base because the remaining components' URDFs were relatively simple to add. We collected all the other components in RMP220's URDF and defined their joints with each other. Then, we run the RMP220's launch file to test our URDF.

Unfortunately, initial results were far from desired. The robot components appeared hugely misaligned and far away from each other. The most possible cause of this error was that we did not correctly configured the reference axes and planes on Solidworks before running the SW2URDF tool.

Nevertheless, the error was recoverable in this phase without shifting our focus back to previous work. We adjusted the link and joint locations direcly from URDF and observed the results on Gazebo. Upon a series of trial and error, we successfully built the robot as observed in Figure 10 (a).

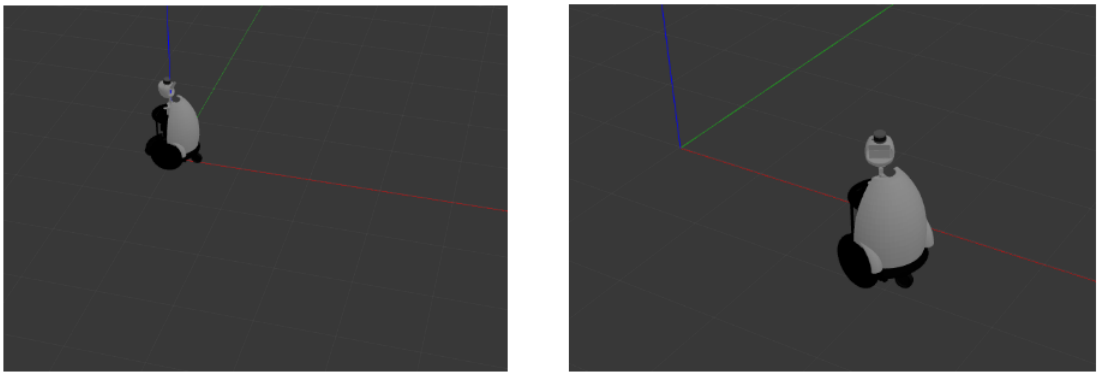


Figure 10: Final robot in Gazebo after system integration: (a) Spawn, (b) Control

However, the control part were not as successful as visualization. While we could control Pan-Tilt Device, the commands for RMP220 were unable to move the robot. Upon studying the Gazebo logs, we detected the error. The *ros_gazebo_plugin*, which is a required plugin to control the robot on Gazebo, was defined for both RMP220 and Pan-Tilt device, which must defined only once for the whole robot. This explains why we could control Pan-Tilt and not the RMP220. In fact, while Gazebo was reading the URDF of the robot, it deactivates the current plugin when it reads a new one, which in our case was the pan-tilt device after RMP220.

Upon fixing this error, we could navigate our robot by commanding RMP220 and Pan-Tilt device as observed on Figure 10 (b). Moreover, RGB-D and LIDAR sensors were working as expected. This is to say that our robot was ready to acquire *visual intelligence* by a neural-network-based object detection scheme at this point.

Lastly, we documented the installation and a how-to-use tutorial of our complete system under Appendix A.

3.2 Object Localization: Detection and Tracking

In this stage, we aimed to localize an object in the robot's viewing range. Since the operation runs in real-time and there would be no GPU support in the real robot, it required to run on CPU and be fast.

To meet the requirements, we employed detection and tracking in a sequential manner. We decided that the detection runs initially and once after every T tracking operation. Our aim was to keep T high as much as possible. Considering the speed and accuracy of the used models and algorithms used, we chose $T = 10$ to maximize the localization performance.

In the following sections, we will detail detection and tracking parts.

3.2.1 Detection

In detection, our aim is to obtain a bounding box around the object in the RGB image. Toward this end, we use the TensorFlow 2 Object Detection API. It features a "Model Zoo" which contains state-of-the-art deep learning models pre-trained on COCO 2017 Dataset. To view the complete list, refer to [15]. We selected two fastest candidates among the list as shown in Table 1.

Model Name	**Speed (ms)**	**COCO mAP**	**Outputs**
SSD MobileNet v2 320x320	19	20.2	Boxes
CenterNet Resnet50 V1 FPN 512x512	27	31.1	Boxes

Table 1: Candidate model characteristics in TF Model Zoo

Following the tutorial in [2], we installed the API and then downloaded our candidate models. Our first aim was to measure the speed of CPU-based inference in our system. The accuracy would be evaluated through empirical observations during the simulations. We constructed a test environment shown in Figure 11 and obtained Table 2.

(a) (b)

Figure 11: Bounding boxes for a target object using SSD MobileNet v2 320x320: (a) RGB image, (b) Depth map.

Model Name	Model Speed (s)		Confidence %
	Loading	Inference	
SSD MobileNet v2 320x320	19	0.89	64
CenterNet Resnet50 V1 FPN 512x512	28.94	2.38	95

Table 2: Candidate models' characteristics in actual environment to detect a person

We favor the first model in Table 2 to use as our detector because the second model was too slow that its inference dramatically lags behind the incoming RGB Stream. Even though we compromise from accuracy with the first model, we chose it since its superior speed was the prioritized concern for us.

3.2.2 Tracking

We employed OpenCV's Tracking API to implement the tracking part in our project. The API exposes various tracking algorithms and we selected the MIL algorithm due to its superior results with real-time performance.

We developed a minimal setup to test the tracker with a human as our target object in the environment. Then, we initialized the tracker with the bounding box coordinates of the human. As a result, the tracker worked as expected as shown in Figure 12 and obtained an average of 0.27 seconds as tracking processing time.

Figure 12: Tracking of a target using MIL algorithm

3.2.3 Tracking with Head

Up until now, the tracking is realized only on the software level. However, we were interested in migrating to hardware tracking, i.e tracking the object with the head. With this setting, we always center the target object to our RGB image, therefore never lose sight of the target.

To realize our goal, we defined a function f,

$$f(v, y(t)) = a(t) \tag{1}$$

where,
$v = (x_T, y_T)$ is the coordinates of center of the gravity of the target's bounding box,
$a(t)$ is the desired reference input to the pan-tilt.

However, it is very important to wait for the current command to finish until a new command is given to pan-tilt. Therefore, we define $r(t)$ as,

$$r(t) = \begin{cases} r(t) + a(t) & |r(t) - y(t)| < M_{ss} \\ r(t) & \text{o.w} \end{cases} \tag{2}$$

where,
M_{ss} is a threshold limit to consider that the pan-tilt is not moving anymore,
$y(t)$ is the output position of the pan-tilt.

We will now explain how we implemented our mathematical solution in ROS. For the sake of simplicity, consider only pan controller, i.e, $v(t) = x_T$ instead of (x_T, y_T).

Consider Eq. 1. We easily calculate $v = x_T$ because we already know the target's bounding box via tracking. Moreover, we obtain $y(t)$ by subscribing to `\pan_controller\state\position_value` topic which provides the current output state of the pan apparatus. Next, f is a function that maps from RGB range in pixels to viewing range in radians. More explicitly, we define f as,

$$f(x_T) = \frac{x_T}{W}\Theta - \frac{\Theta}{2} \tag{3}$$

where,
W is the maximum RGB width,
Θ is the maximum horizontal viewing angle.

Having found $a(t)$, we now consider Eq. 2. We only add $a(t)$ to current $r(t)$ and publish it to pan controller if and only if the pan apparatus is settled, i.e is not moving anymore. If we didn't consider this condition, the pan would move more multiple times by the desired distance and fail dramatically.

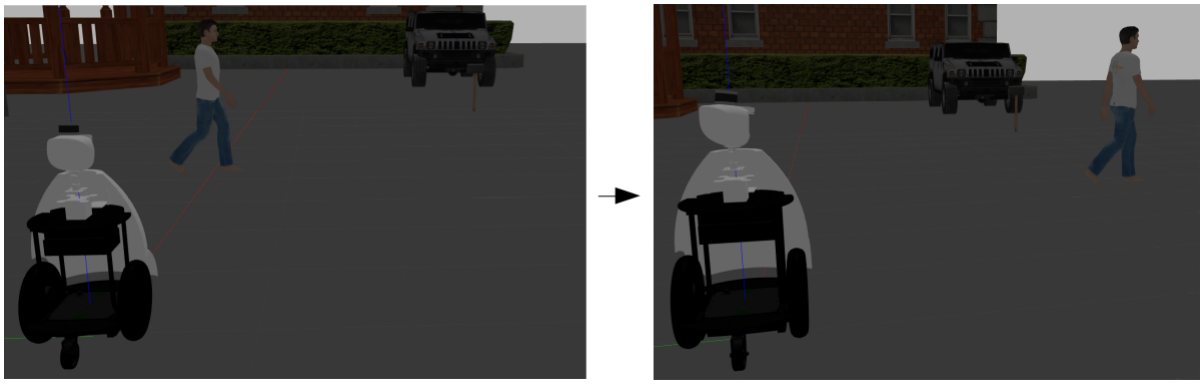


Figure 13: Tracking with head as target moves to the right.

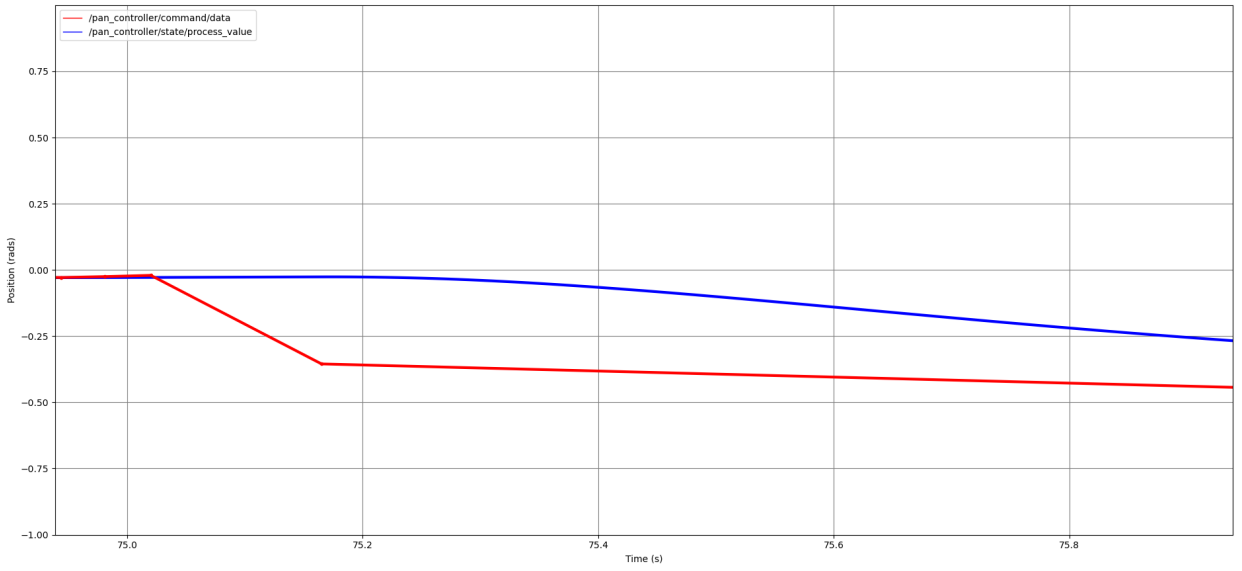


Figure 14: Reference input (red) and output (blue) plots of the controlled pan system while the head moves to the right in Figure 13.

3.2.4 System Integration

After developing detection and tracking individually, we integrated them together to our project. For this purpose, we developed a ROS node that receives a stream of RGB images from the sensor and process them sequentially within tracking and detection modules. Our end-purpose in this node is to eventually obtain the bounding box of the target object in the image.

To achieve detection and tracking in a sequential manner, we implemented a state-machine in the node that acts as a main loop as shown in Figure 15. At each iteration in the tracking module, the node publishes the bounding box of the object.

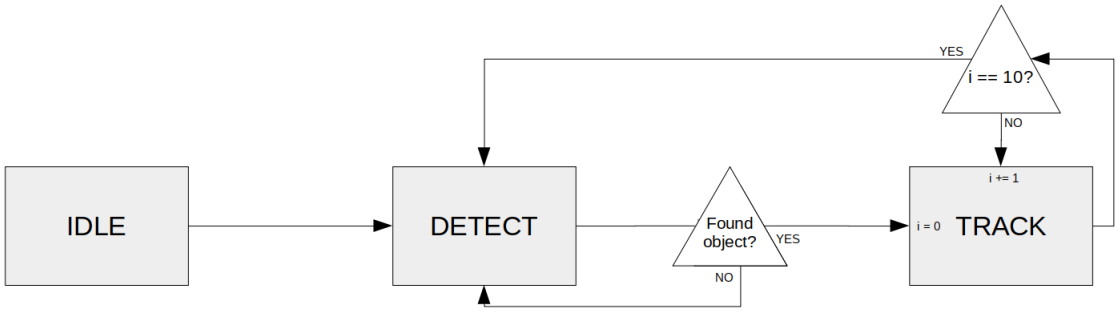


Figure 15: Detection-Tracking node main logic

The target object is specified as a command line argument to the node. There are 90 object categories for the pre-trained neural network and therefore we can run our detection-tracking scheme for any of these objects. For example, to select the target object as "Person", we run the node by,

```
python detect_track.py person
```

On the other hand, we developed another node for head-tracking of the target object and integrated to our system. The node subscribes to the bounding box of the object and obtain the reference angles for the pan-tilt controller using Equations 1, 2 and 3. The reference angles are then published to the pan-tilt controller to center the target to the image.

3.3 Autonomous Navigation Relative to Object

In this section, our purpose is to navigate towards the object that we have been tracking. Toward this end, we employed a variation of APF algorithm that was developed by BOUN ISL. The algorithm requires us to specify the target location in PoseStamped format, which is,

```
Header header
    uint32 seq
    time stamp
    string frame_id
Pose pose
    geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
    geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

Since our robot moves on the terrain, it is sufficient to specify the position `x` and orientation `z` and keep the remaining as default 0. Now, our purpose is the find the `x` and `z`, which are the distance and angle to the target, respectively, relative to the target.

From the previous section we know,

i. target's bounding box coordinates in the RGB image,

ii. pan's angle with the axis of rotation when the target is in the center in the RGB image.

We see that ii. directly correlates to `z`. To find `x`, we utilize the corresponding depth map of the RGB image. We extract the target's box in the depth map using i. and calculate its histogram, which lays out the distribution of distances in the box. We, then, select the most repeated distance in the box as our target's distance and equate it to `z`.

Having found the target's location, we publish the PoseStamped message to the APF algorithm. APF algorithm, then, navigates our robot to the target as shown in Figure 16.

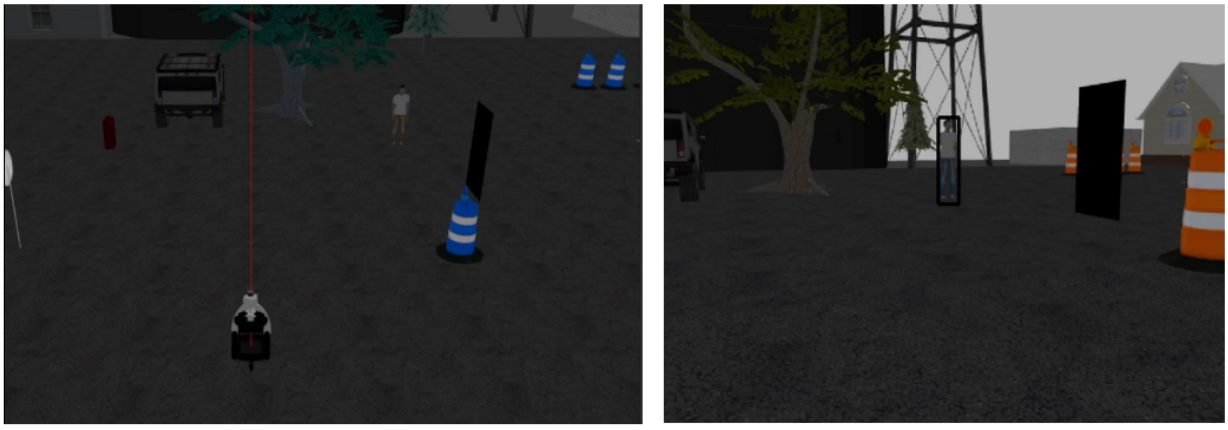
4 Conclusion

Robots require successful object localization schemes to achieve their navigation tasks. Toward this end, we proposed a sequential detection and tracking scheme using RGB-D sensor data for our Segway RMP220 based robot. In Gazebo environment, we show that we successfully detect an object, track it with head and navigate towards it.

4.1 Future Work

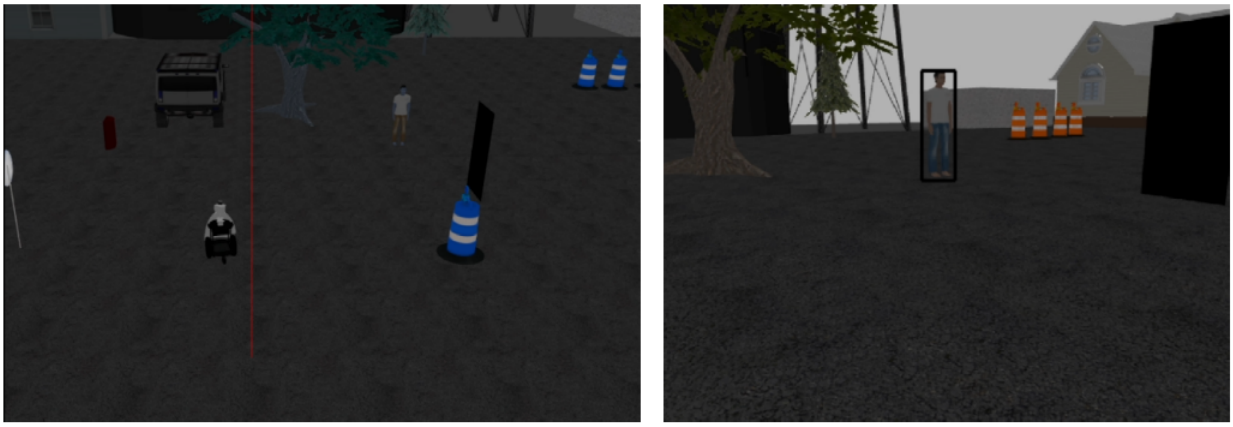
A promising area of future work is LIDAR sensor incorporation to object localization. A major drawback of our project is that the robot's detection is limited on its RGB-D sensor viewing angle. With LIDAR, the robot will sense its 360° environment and therefore can locate and navigate objects that are found, for example, on its back as well.

Related to the first point, it would also be a exciting direction of development to address the problem of point cloud analysis. Currently, our method relies on RGB image as well as depth map. Constructing a point cloud environment using LIDAR or RGB-D sensor might improve the detection and tracking performance.



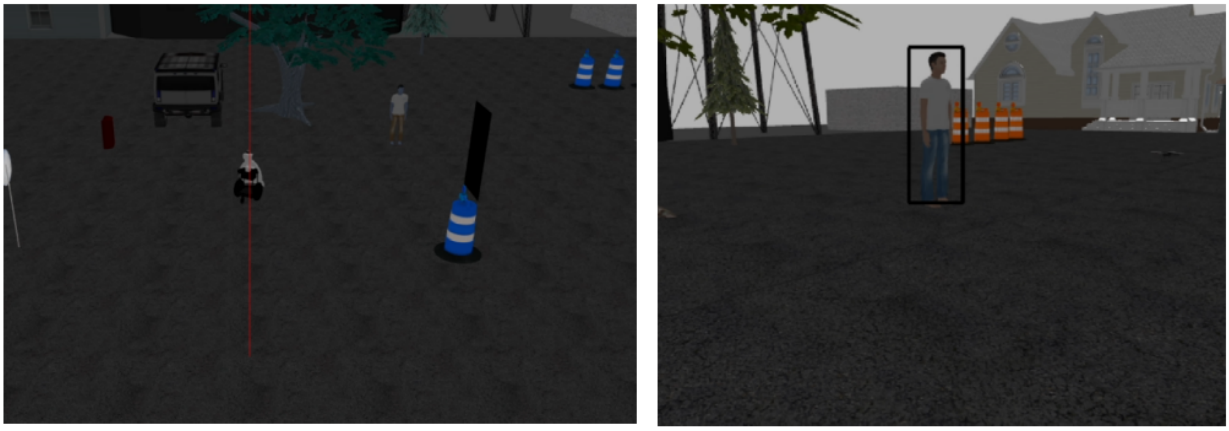
(a) (b)

Start



(a) (b)

Midway



(a) (b)

End

Figure 16: Autonomous navigation towards the human: (a) Gazebo Views, (b) RGB Sensor Views

References

[1] URL: https://www.stereolabs.com/docs/depth-sensing/.

[2] URL: https://github.com/tensorflow/models/tree/master/research/object_detection.

[3] URL: https://docs.opencv.org/3.4/d9/df8/group__tracking.html.

[4] URL: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[5] URL: http://gazebosim.org/tutorials/?tut=ros_control.

[6] URL: https://www.youtube.com/watch?v=7pjogRqbmIk.

[7] URL: http://gazebosim.org/tutorials?tut=ros_gzplugins#Tutorial:UsingGazebopluginswithROS.

[8] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. "Visual tracking with online multiple instance learning". In: *2009 IEEE Conference on computer vision and Pattern Recognition*. IEEE. 2009, pp. 983–990.

[9] Yu Cheng et al. "Model compression and acceleration for deep neural networks: The principles, progress, and challenges". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136.

[10] Sven J Dickinson et al. *Object categorization: computer and human vision perspectives*. Cambridge University Press, 2009.

[11] Wei Ji et al. "Automatic recognition vision system guided for apple harvesting robot". In: *Computers & Electrical Engineering* 38.5 (2012), pp. 1186–1195.

[12] Changjuan Jing et al. "A comparison and analysis of RGB-D cameras' depth performance for robotics application". In: *2017 24th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*. IEEE. 2017, pp. 1–6.

[13] Woon Yong Kim. *Pan/tilt camera*. US Patent 6,503,000. Jan. 2003.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[15] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[16] Tien Thanh Nguyen et al. "Apple detection algorithm for robotic harvesting using a RGB-D camera". In: *International Conference of Agricultural Engineering, Zurich, Switzerland*. 2014.

[17] Sankar K Pal et al. "Deep learning in multi-object detection and tracking: state of the art". In: *Applied Intelligence* 51 (2021), pp. 6400–6429.

[18] Jaehong Park et al. "Pan/tilt camera control for vision tracking system based on the robot motion and vision information". In: *IFAC Proceedings Volumes* 44.1 (2011), pp. 3165–3170.

[19] Cristiano Premebida, Rares Ambrus, and Zoltan-Csaba Marton. "Intelligent robotic perception systems". In: *Applications of Mobile Robots*. IntechOpen London, UK, 2018, pp. 111–127.

[20] Charles Ruizhongtai Qi et al. "Frustum PointNets for 3D Object Detection from RGB-D Data". In: *CoRR* abs/1711.08488 (2017). arXiv: 1711.08488. URL: http://arxiv.org/abs/1711.08488.

[21] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[22] Amy L Tabb, Donald L Peterson, and Johnny Park. "Segmentation of apple fruit from video via background modeling". In: *2006 ASAE Annual Meeting*. American Society of Agricultural and Biological Engineers. 2006, p. 1.

[23] Athanasios Voulodimos et al. "Deep learning for computer vision: A brief review". In: *Computational intelligence and neuroscience* 2018 (2018).

[24] Roberto Yus et al. "MultiCAMBA: a system for selecting camera views in live broadcasting of sport events using a dynamic 3D model". In: *Multimedia Tools and Applications* 74 (June 2015), pp. 4059–4090. DOI: 10.1007/s11042-013-1810-4.

A General Installation and Demo Tutorial

An object detection and relative navigation scheme for our custom robot on Gazebo.

A.1 Getting Started

A.1.1 Prerequisites

1. ROS Melodic

   Install following the instructions on http://wiki.ros.org/melodic/Installation.

2. ROS Package Dependencies

   Install them via,

   ```
   sudo apt-get install ros-melodic-costmap-2d ros-melodic-robot-localization

   ros-melodic-yocs-cmd-vel-mux ros-melodic-effort-controllers ros-melodic-navigation

   ros-melodic-geometry2 ros-melodic-nmea-msgs ros-melodic-bfl ros-melodic-arbotix-python
   ```

3. Tensorflow Object Detection API

   Install following the tutorial under the link*

4. Python Dependencies

   Activate the Conda environment created in Step 3 and run,

   ```
   conda install opencv-python
   ```

A.1.2 Installing

A.1.2.1 Robot and Environment

5. Create a new Catkin workspace,

   ```
   mkdir -p ~/navi_ws/src
   ```

6. Clone the repository to the workspace,

   ```
   cd ~/navi_ws/src
   git clone https://github.com/mehmetalici/localize-and-navigate.git
   ```

7. Build the packages in the workspace,

   ```
   cd ~/navi_ws
   catkin_make
   ```

8. Source the workspace and the robot,

   ```
   cd ~/navi_ws
   source devel/setup.bash
   source src/localize-and-navigate/segway_v3/segway_v3_config/std_configs/segway_config_RMP_220.bash
   ```

9. Test the robot and environment running,

   ```
   roslaunch segway_gazebo segway_empty_world.launch
   ```

   You should see our robot and world in the Gazebo simulation software.

*<https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install>

A.1.2.2 Detection

1. Download a model

   Activate your environment, and run,

   ```
   cd ~/navi_ws/src/localize-and-navigate/obj_detection/src
   python download_model.py 20200711 ssd_mobilenet_v2_320x320_coco17_tpu-8
   ```

   to download the model with name and date `ssd_mobilenet_v2_320x320_coco17_tpu-8` and `20200711`, respectively.

   You can change the model name and date according to the models in the list[†], which involves all pre-trained models.

A.1.2.3 Tracking

1. Download a specific version of OpenCV Bridge.

   Apply,

   ```
   cd ~
   git clone https://github.com/mehmetalici/opencv-bridge-err-resolver.git
   cd opencv-bridge-err-resolver
   catkin clean -b
   catkin build
   ```

A.2 Robot Control

A.2.1 Control of Head

A pan-tilt device rotates the head of the robot. The joints of Pan-Tilt are controlled by their PID controllers.

The head features a ZED RGM-D camera and it is possible to control the head by providing (x, y) coordinates of the target in the RGB image. To test it, first run,

```
roscd robot
rosrun rviz rviz -d rviz/rviz_cfg.rviz
```

and to move to a position of 0.5 rads positive to the axis of rotation for both joints, run,

```
rostopic pub /pan_controller/command std_msgs/Float64 "data: 0.5"
rostopic pub /tilt_controller/command std_msgs/Float64 "data: 0.5"
```

A.2.2 Control of Navigation

The robot can be navigated by commanding to RMP220.

For instance, to move the robot through a circle, run,

```
rostopic pub -r 10 /segway/teleop/cmd_vel geometry_msgs/Twist "linear:
  x: 10.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0"
```

[†] <https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md>

A.2.3 Control of Sensors

The robot is equipped with a Velodyne Puck LIDAR and a ZED RGB-D Camera. The sensors are always active and publishing under their respective topics.

Sensor information can be visualized using `RViz`. To do this, follow the instructions here[‡].

A.3 Detection and Tracking

1. Activate your environment and source your workspace.
2. Run,

   ```
   roscd obj_detection/src
   source ~/opencv-bridge-err-resolver/install/setup.bash
   ```

3. To detect a `person`, run,

   ```
   python detect_track.py person
   ```

   To see a complete list of avaliable objects you can pass to the script, go to the directory in which you downloaded the model and open `mscoco_label_map.pbtxt`.

4. To visualize detection, run,

   ```
   rosrun rviz rviz
   ```

   Then select the topics `/rgb_with_det` or `/rgb_with_det`, which produces images with boxes of target object, on RViz GUI.

A.4 Navigation

1. Source the workspace and run,

   ```
   roscd obj_detection/src
   python navigate_goal.py
   ```

2. Run the APF algorithm,

   ```
   roscd apfrl/src
   python apf_rl_0707.py
   ```

[‡]<http://gazebosim.org/tutorials?tut=drcsim_visualization&cat=drcsim>

B Segway Robot Ros Melodic Bringup

A tutorial to bring-up Segway robot in ROS Melodic.

B.1 Getting Started

B.1.1 Prerequisites

1. Ros Melodic

   Install following instructions on this link[§].

2. ROS Package Dependencies.

   Install them via,

   ```
   sudo apt-get install ros-melodic-costmap-2d ros-melodic-robot-localization

   ros-melodic-yocs-cmd-vel-mux ros-melodic-effort-controllers
   ```

B.1.2 Installation

1. Clone the repositories,

   ```
   mkdir -p ~/segway_ws/src
   cd ~/segway_ws/src
   git clone https://github.com/StanleyInnovation/segway_v3_simulator.git
   git clone https://github.com/StanleyInnovation/segway_v3.git
   git clone https://github.com/StanleyInnovation/segway_v3_robot.git
   ```

2. Normally at this point you should be building the packages with `catkin_make`. However, for various reasons, e.g. unsupported ROS Melodic version, the following packages will fail to build:

   - `segway_assisted_teleop`
   - `ublox`
   - `but_velodyne`
   - `velodyne`
   - `nmea_comms`

3. You may select one of following:

   a. Remove Failing Packages

   If you won't need those packages, delete them from the repository by,

   ```
   cd ~/segway_ws/src
   rm -rf segway_navigation/segway_assisted_teleop
   rm -rf segway_v3_robot/third_party_sensors/nmea_comms
   rm -rf segway_v3_robot/third_party_sensors/velodyne
   rm -rf segway_v3_robot/segway_sensor_filters/but_velodyne
   rm -rf segway_v3_robot/third_party_sensors/ublox/
   ```

   b. Building Packages by Workarounds

   Current workarounds are listed below. We will update this section as much as possible.

   - `segway_assisted_teleop`: Follow the instructions here[¶].

[§] <http://wiki.ros.org/melodic/Installation>

[¶] <https://gist.github.com/mehmetalici/3964f139bd5316b8d0c4098dd15e0326>

4. Build packages with `catkin_make` and source the workspace by,

   ```
   cd ~/segway_ws
   catkin_make
   source devel/setup.bash
   ```

5. Source the desired robot's config.

   First, See the available robots by,

   ```
   cd ~/segway_ws/src/segway_v3/segway_v3_config/std_configs/
   ls
   # Prints the available robots.
   ```

   For `RMP210`, run:

   ```
   source segway_config_RMP_210.bash
   ```

6. Test the installation by,

   ```
   roslaunch segway_gazebo segway_empty_world.launch
   ```

C Segway Package Workarounds for ROS Melodic Bringup

Segway ROS packages does not support Melodic. In this tutorial, we will provide step-by-step instructions on how to build various SEGWAY packages on ROS Melodic. The tutorial is not exhaustive and we will expand our coverage with time.

C.1 `segway_assisted_teleop` Package

C.1.1 Common Steps

1. Create a Catkin workspace.
   `mkdir -p ~/segway_ws/src`
2. Clone Segway V3[‖] repository to `/src`
   `git clone https://github.com/StanleyInnovation/segway_v3.git`
3. Build the Segway packages within the repository by,
   ```
   cd ~/segway_ws
   catkin_make
   ```
4. You will face errors while CMake is building the `segway_assisted_teleop` package. Their fixes are explained below:

C.1.2 B. Errors and Fixes

ERROR: `tf` does not name a type

Following is the first error that you will be facing,

```
In file included from /home/malici/test2_ws/src/segway_v3/segway_navigation/
segway_assisted_teleop/src/segway_assisted_teleop.cpp:37:0:
/home/malici/test2_ws/src/segway_v3/segway_navigation/segway_assisted_teleop/
include/segway_assisted_teleop/segway_assisted_teleop.h:56:7: error: âtfâ does not name a type;
did you mean ‘tm‘?
    tf::TransformListener tf_;
    ^~
    tm
```

FIX

1. Fix this error by adding the line

   `#include <tf/transform_listener.h>`

   to

   `segway_assisted_teleop/src/segway_assisted_teleop.cpp`.

2. You should be able to build packages by,

   ```
   cd ~/segway_ws
   catkin_make
   ```

ERROR: no matching function for call to Costmap2DROS

Next, you will face the following error,

```
/home/malici/test2_ws/src/segway_v3/segway_navigation/
segway_assisted_teleop/src/segway_assisted_teleop.cpp:
In constructor "assisted_teleop::AssistedTeleop::AssistedTeleop()"
/home/malici/test2_ws/src/segway_v3/segway_navigation/
```

[‖] <https://github.com/StanleyInnovation/segway_v3>

```
segway_assisted_teleop/src/segway_assisted_teleop.cpp:92:95:
error: no matching function for call to
"costmap_2d::Costmap2DROS::Costmap2DROS(const char [14], tf::TransformListener&)"
   AssistedTeleop::AssistedTeleop() : costmap_ros_("local_costmap", tf_),
   planning_thread_(NULL)
```

due to ROS version mismatch.

FIX

To fix this, we have to build `ros-kinetic` version of `ros-navi` package (which includes `costmap-2d`) from source in the workspace.

1. Clone the package to workspace,
   ```
   cd ~/segway_ws/src
   git clone https://github.com/ros-planning/navigation.git
   ```
2. Checkout to `ros-kinetic` branch,
   ```
   cd navigation
   git branch -f kinetic-devel origin/kinetic-devel
   git checkout kinetic-devel
   ```
3. You should be able to build packages by,
   ```
   cd ~/segway_ws
   catkin_make
   ```

ERROR:: 'tf2*buffer*' was not declared in this scope

Afterwards you will face this error,

```
/home/malici/test2_ws/src/navigation/amcl/src/amcl_node.cpp:
In member function "tf2_ros::Buffer& AmclNode::TransformListenerWrapper::getBuffer()":
/home/malici/test2_ws/src/navigation/amcl/src/amcl_node.cpp:133:51:
error: "tf2_buffer_" was not declared in this scope
       inline tf2_ros::Buffer &getBuffer() {return tf2_buffer_;}
```

FIX

Fix this error by building `ros-indigo` version of `tf` from source in the workspace.

1. Clone the package to workspace,

   ```
   cd ~/segway_ws/src
   git clone https://github.com/ros/geometry.git
   ```

2. Checkout to `ros-indigo` branch,

   ```
   cd geometry
   git branch -f indigo-devel origin/indigo-devel
   git checkout indigo-devel
   ```

3. Lastly, since now you have two `tf` packages within your CMake include path,

   i. one coming from your system-level `ros-melodic`,

   and,

   ii. one coming from your workspace-level `ros-indigo`,

   we have to prioritize ii. over i. when we let CMake know about the include search paths.

   To do this, open `~/segway_ws/src/geometry/tf/CMakeLists.txt` and find the following code segment,

   ```
   include_directories(
       ${Boost_INCLUDE_DIR}
   ```

```
    ${catkin_INCLUDE_DIRS}
    include)
```

and prioritize the local include directory by,

```
include_directories(
    include
    ${Boost_INCLUDE_DIR}
    ${catkin_INCLUDE_DIRS})
```

4. You should be able to build packages by,

```
cd ~/segway_ws
catkin_make
```

D Widowx Turret Bring-up on Gazebo

A tutorial to bring-up Pan-Tilt Turret on ROS Melodic and Gazebo.

D.1 Getting Started

D.1.1 Prerequisites

1. Ros Melodic Desktop Full

   Install following instructions on http://wiki.ros.org/melodic/Installation.

2. ROS Package Dependencies.

   Install it via,

   ```
   sudo apt-get install ros-melodic-arbotix-python
   ```

D.1.2 Installation

1. Clone the repository,
   ```
   mkdir -p ~/pantilt_ws/src
   cd ~/pantilt_ws/src
   git clone https://github.com/mehmetalici/Widowx-Turret-gazebo.git
   ```
2. Build packages with `catkin_make` and source the workspace by,
   ```
   cd ..  # cd to workspace root dir.
   catkin_make
   source devel/setup.bash
   ```

D.1.3 Simulate the Robot

```
roslaunch widowx_turret_gazebo widowx_turret_empty_world.launch
```

D.1.4 Command the Joints

```
rostopic pub /pan_controller/command std_msgs/Float64 "data: -0.5"
rostopic pub /tilt_controller/command std_msgs/Float64 "data: 0.5"
```