Autonomous Mapping, Localization and Navigation using Computer Vision

Siddharth Saha
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA, 92093
sisaha@ucsd.edu

Jay Chong
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA, 92093
jac269@ucsd.edu

Youngseo Do
Halıcıoğlu Data Science Institute
University of California, San Diego
La Jolla, CA, 92093
y1do@ucsd.edu

Abstract

The focus of this paper is on the application of computer vision in mapping, localization and navigation of an autonomous vehicle. Primarily we are dealing with the problems of

- How to map and localize in an environment where 2D Lidars are not useable
- How to use computer vision to enable navigation in varying light conditions
- How to avoid obstacles and stay within lanes using computer vision

We aim to achieve this using the following methods

- Use of the RTABMAP[1] package which uses Camera, Lidar and Odometry to map and localize. It uses the depth information from the camera as well which allows a 3D Lidar view in at least one direction
- Use of the rqt_reconfigure dynamic GUI to tune camera parameters to reduce sensitivity to light
- Use of Facebook AI Research's Detectron2 Deep Learning network[2] to segment images and detect objects based off captions

1 Introduction

One of the main tools used in autonomous mapping and navigation is a 3D Lidar. A 3D Lidar provides various advantages. It is not sensitive to light conditions, it can detect color through reflective channels, it has a complete 360 degree view of the environment and does not require any ”learning” to detect obstacles. One can use the reflective channel to detect the color of lanes as well as a regular 2D axis view to avoid obstacles. The pointcloud information from the Lidar can also easily enable mapping and localization as the vehicle will know where it is at all points. It is easy to see why so many large scale autonomous vehicle units invest in expensive and bulky Lidars. However, this is not accessible to all due to it's price. A camera (even depth) is much more affordable. However it comes with it's own slew of disadvantages. It can see color but programming for the color is hard due to varying light conditions. Unless you use multiple cameras you often can't see all around you. These factors together are a hindrance to autonomous mapping and navigation and we thus aim to resolve it through this paper

2 Environment

To carry out experiments we choose to use our field track at UCSD. It provides an environment where the 2D Lidar sometimes comes in use (through cones) but still cannot completely depend on the Lidar to navigate safely. There are white lane markings and central yellow lanes that can be used as a guide. We can also spread cones around to use as obstacles. During the daytime the sunlight appears heavily on end while being darker on the other side which creates issues in doing even 1 lap safely in the track. In the night the light conditions are much more stable. But as seen in the image below there are still areas where the light is strong and other areas where the light is much weaker

Figure 1: Environment for testing

3 Experiment Design

To meet the 3 targets described in the abstract we have set up 3 different experiments:

3.1 Mapping

In SLAM, we are driving the car around to construct a map of the environment while simultaneously localizing itself relative to the map[3]. For this experiment, we will be generating our own data. This will be done by controlling a car and have it drive around the track while the car maps and localizes itself. Previously, we ran this experiment in a simulated environment and it generated 3 different datasets, the ground truth, the odometry path, and the SLAM path. However, in the real world, we don't have a ground truth. So, we will be driving the car on the yellow line of the track as a substitution. Driving it on the track's yellow line will improve testing stability and give us more consistent results. The goal here is to increase the number of greens we see in figure B. Getting a green means that the algorithm has successfully found a loop closure and localized itself. Getting a yellow means that the loop closure has been rejected, but the data is still saved for further analysis and processing. An example can be seen below

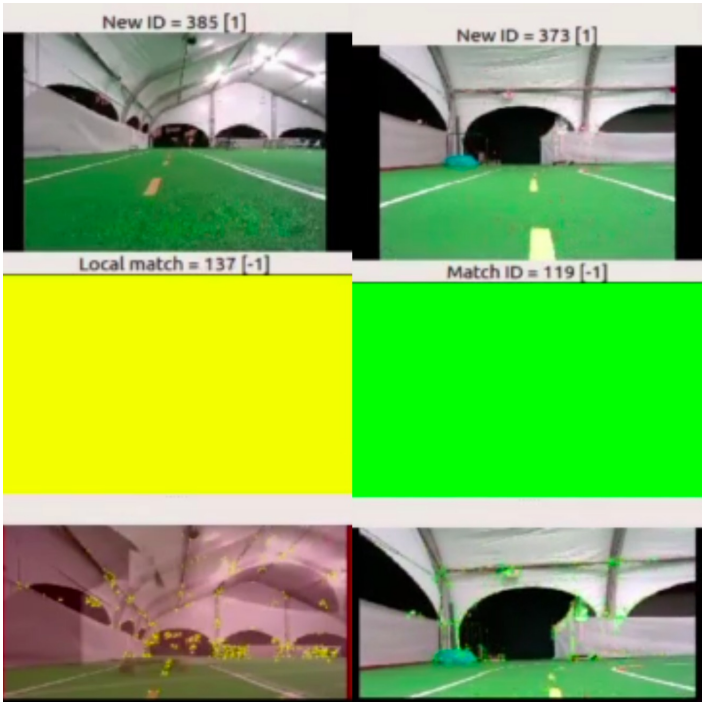


Figure 2: Green and Yellow hypothesis on the track

Driving speed is crucial when running the RTABMAP algorithm. When the car goes too fast or makes sudden movements, the car will fail to localize itself or detect any loop closures. In figure A, there is red ghosting at the bottom square and this is caused by a sudden movement while turning. In order to fix this issue, we had to drive over the same area. Furthermore, an environment in which the vehicle can return to a previously visited spot is beneficial because this allows the car to map and locate new images to older ones. When we mapped in an indoor environment without any set path, the mapping found very little loop closures. However, when mapping on the track, where we were able to drive the car in a consistent path, the car is able to find a significant number of loop closures.

3.2 Light Conditions

The problem we want to address is: how do we figure out a way for the car to navigate robustly using computer vision under varying light conditions? When driving the car in the tent, we initially noticed that the car had some challenges in navigating under sunlight conditions, and realized that Intel Realsense D455 camera configuration tuning is necessary in order to alleviate the sensitivity of the camera to such bright conditions. Specifically, we want to test different camera parameters (brightness, depth gain, contrast, auto exposure, hue, white balance etc.) dynamically to see which

configuration setting could best set off the camera's sensitivity to bright conditions and give us a similar image layout to the normal, default setting (non-bright conditions).

Our initial idea was to find a way to incorporate Intel Realsense software development kit (SDK) directly to ROS so that we can use the SDK sliders to adjust camera configuration settings. However, this was quite hard to implement and debug within our workspace. As a more realistic approach, we integrated Realsense camera settings to a rqt plugin called 'rqt_reconfigure' so that we can easily view and edit parameters that are accessible with the dynamic_reconfigure package. With this package, we are able to use its command line tools to produce a variety of .yaml files that will be read in the launch file to make the data collection process easier and reproducible. After executing the launch file, we can simultaneously use this rqt plugin tool and refer to Intel's tuning guidance[4] to address our question. Specific steps to our experiment are as follows.

- Use dynamic reconfigure package to tune parameters
- Take several images of the same area with different lighting conditions
- Once data under different configuration settings is collected with the recording procedure, compare between same images at the same position using the evaluation metric SSIM
- Repeat the process and choose configuration with the best SSIM

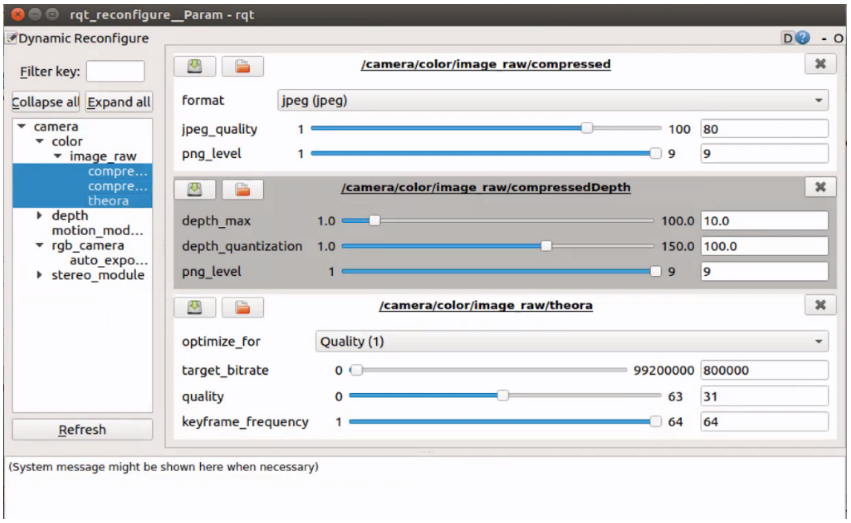


Figure 3: rqt_reconfigure GUI

3.3 Object Segmentation

The experiment design here is relatively simple compared to the previous two.

3.3.1 Data Collection

We run some laps in our track and collect image data. This poses a slight issue in that the images are collected at 60FPS(frames per second). Meaning we have 60 images of nearly the same moment. In the Light Conditions section this didn't pose a problem since we would just record the first image and stop. To combat this issue we diverted the images into to a new ROS topic that published the images at a throttled down 1FPS. This allowed for a lot more distinct images to be collected that made the data collection and storage process a lot more efficient

3.3.2 Data Preparation

This image data is then labelled with the help of the MakeSense AI tool[5]. This tool enables annotations in various formats in a precise manner. We used the COCO JSON format to get segmentations and boundary boxes. The images are then manually split into a train, validation and test data.

3.3.3 Training

We built a repository that is able to take this data and provide us back metrics on how the Detectron2 model performed. This repository allows for several model files to be provided so that we can run several experiments at once without constantly monitoring. This enables us to get results back much more efficiently as well. We can run experiments using different base models(Mask RCNN, Faster RCNN etc), different learning rates, different epochs and different batch sizes. The model which performs best on the validation dataset is returned. This model is then used to evaluate the test data as well as return a sample video of all the predictions on the test. Normally a video on test data would not make sense since each test image would not be sequential. However, in our case, since the image data for train, validation and test are collected sequentially in laps they tie together as well. By providing a custom framerate we are able to control the output of the video as well for legible outputs

4 Dataset

Each of the 3 experiments provide their own dataset which is used in different ways:

4.1 Mapping

Our experiment generated a .db file which consisted of positional data and image data. Positional data comes in 2 files and each file has the following format:

Column Name	Meaning
timestamp	Time stamp at which the position was recorded
tx	Translational X component position
ty	Translational Y component position
tz	Translational Z component position
rx	Imaginary X component of Quaternion or pitch
ry	Imaginary Y component of Quaternion or roll
rz	Imaginary Z component of Quaternion or angle of wheels
rw	Real W component of Quaternion or yaw

Both datasets will have the same timestamp for each data entry. This means for any given timestamp, we have 2 points of information on the location of the robot.

Since both datasets are already aligned, we don't have to do any preprocessing of the data. We will do a basic sanity test on the data by plotting the positions of each dataset and seeing if the plotted trajectory of the datasets are similar. An example is shown below:

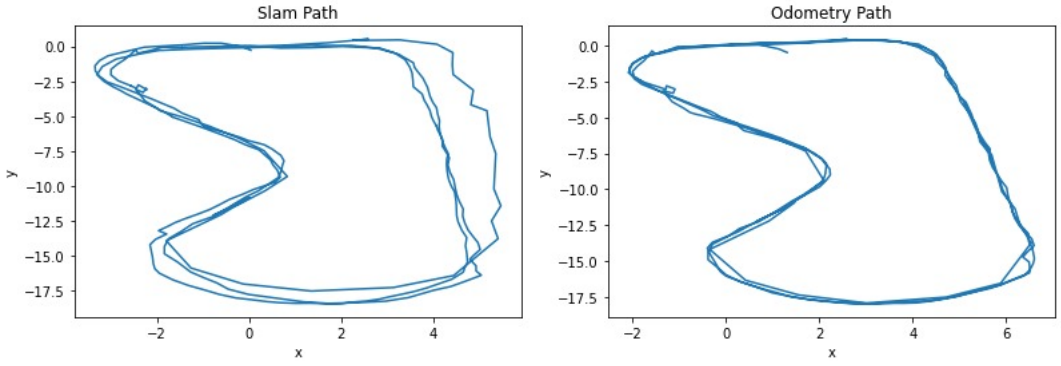


Figure 4: Plotted trajectory of datasets

Using RTABMAP's built-in database, we are able to view image data that was recorded during the mapping and localization process. Below are two images from our dataset.

Figure 5: Incorrect mapping b/w 2 images

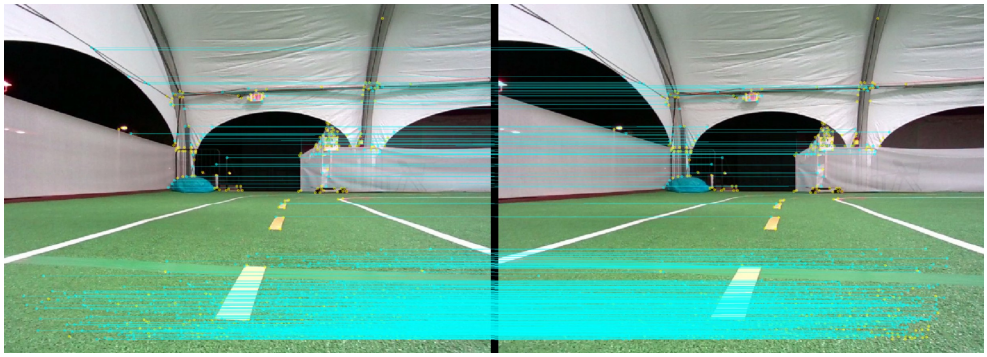


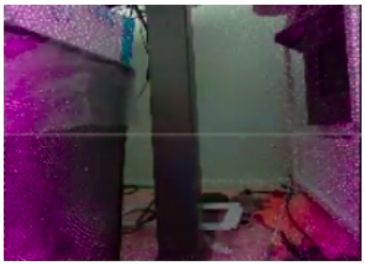
Figure 6: Correct mapping b/w 2 images

In each image data, there are 2 key elements that RTABMAP labels, the first is a yellow dot, which represents a unique key point in that particular image, the second is a blue line, which represents when RTABMAP is able to match key points between two images. There are two images in every index of image data, one from an initial mapping and another which is used to compare it with the old image. In figure 5, we see that since they are different frames, there are no blue lines. However, in figure 6, it was able to match a majority of the yellow points. With this, a loop closure is detected at the given image data.

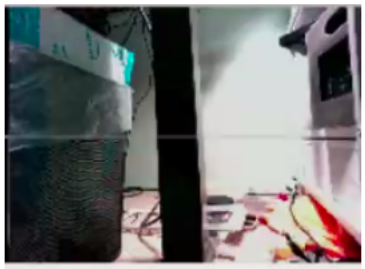
4.2 Light Conditions

Our experiment will record and generate a collection of .bag files, with one image to be exported from each .bag file. Since we have previously witnessed that the car navigates robustly under conditions with less sunlight, we will mainly use the image data under default settings (non-bright) as a standard baseline to be compared with images with other configuration settings under brighter conditions.

1. Image under default settings (non-bright)

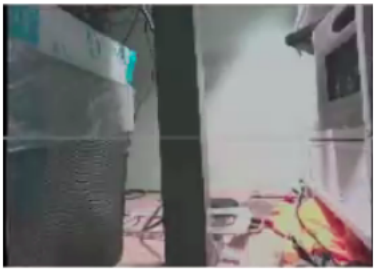


2. Image under default settings with lamp above the camera (brighter conditions)



3. Image under lower brightness (-26) setting with lamp above camera

4. Image under lower gain (30) and higher brightness (57) setting with lamp above camera



4.3 Object Segmentation

The dataset will be a set of images. Primarily a ROS bag file containing images from the throttled down topic we made. We extract the images from the bag file, label and split into train, validation and test data. Each of the 3 contains the following information

- A COCO JSON dictionary containing the annotations for each image
- A set of images (whose file names are included in the JSON dictionary above)

The COCO JSON dictionary (after being read into Detectron2) is formatted as follows

- Images: Contains a list of dictionaries. Each dictionary contains information on a specific image. We have details like the Image ID(used to link with annotations and segmentations later), File Name, Width and Height for each image
- Annotations: Contains a list of dictionaries. Each dictionary contains information on a specific annotation instance. We have details like the annotation ID, image ID, category ID, iscrowd (refers to whether we are annotating a single object or a bunch of objects together. For the purposes of this research paper all our objects are annotated separately so iscrowd is 0), and segmentation (a list of x,y vertices since our labelling is in polygon format)
- Categories: Contains a list of dictionaries. Each dictionary contains information on the categories our dataset is labelled for. We have details like the category name and category ID. For the purposes of this experiment we have categories lane and cone. The category ID is used to reference these category names

A sample image can be viewed below

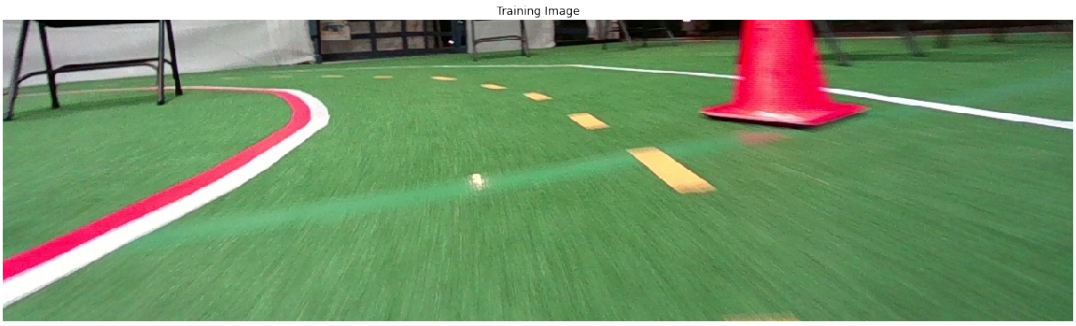


Figure 7: Raw training image

The image is a 360 x 1280 image taken from the Research environment mentioned earlier. As mentioned we can see the white lane and orange cone in the image.

The boundary boxes are generated from a max computation using the list of vertices in segmentation (basically most extreme at each end). With this we have a clear idea of where the objects are on the map and can compute the position we should move towards to avoid the objects detected (thus staying within the lane and avoiding the cones)

5 Evaluation

Each of the 3 experiments have their dataset evaluated in different ways:

5.1 Mapping

We will be using the Absolute Trajectory Error (ATE) as our main metric to accurately evaluate our SLAM algorithm. Below is a visual representation of how the ATE is calculated[6]

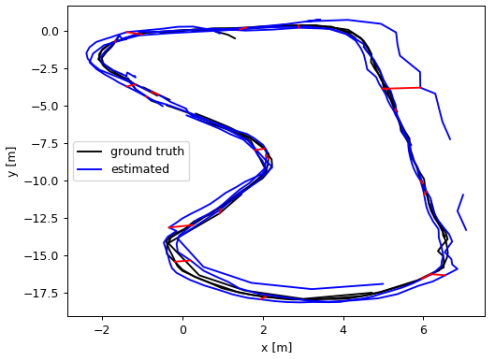


Figure 8: Baseline ATE

The path is created through driving around the track multiple times. The blue lines represent what the algorithm thinks the path is, the black lines represent the ground truth, and the red lines represent the distance between the black and blue lines

5.1.1 Integration

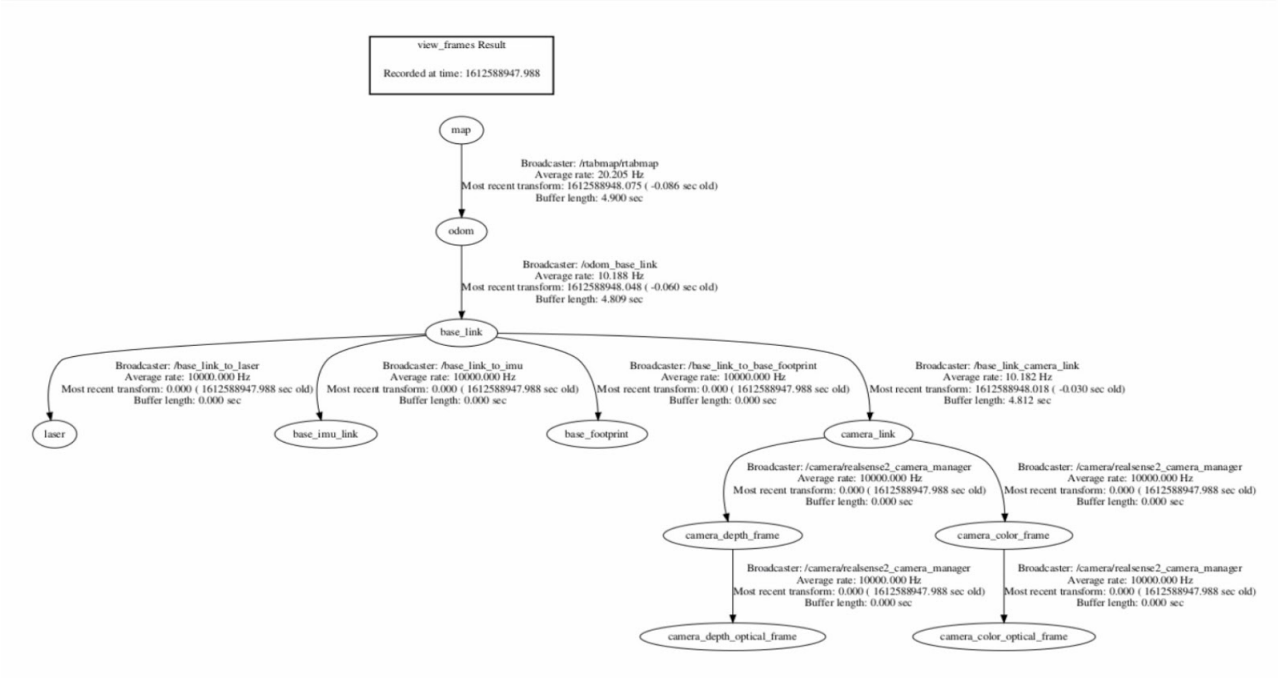


Figure 9: Transform Tree

Prior to running the RTABMAP algorithm in the real world, we needed to make sure all of our sensors were working. This includes the lidar, vesc, and RGB-D camera. After making sure that all sensors are working, we created a launch file that sends sensor data to RTABMAP for processing. The launch file included dummy nodes so that the correct transformation tree can be created. RTABMAP requires that all sensors are under the same transformation tree for any data to be processed.

5.2 Light Conditions

The metric we will be using to evaluate the sensitivity of different camera configurations to lights is Structural Similarity Index (SSIM) and Mean Squared Error (MSE). First, MSE is a fundamental metric that calculates the difference in surface, or pixels of two compared images. SSIM is used as a metric to measure the similarity between two images at the same position. Structural Similarity Index between two images is calculated as a value between -1 and +1[7]. A value of +1 indicates that the 2 images are very similar, while a value of -1 indicates that they are very different[7]. In our case, we will compute the SSIM between the image under default configuration setting and image under adjusted configuration settings with brighter conditions (study lamp or phone lights over camera). The higher the SSIM between the image under default setting and tuned camera configuration setting, the lower the sensitivity of the camera is.

5.2.1 Why we specifically chose SSIM:

Structural Similarity Index (SSIM) is fit to reflect the human visual perception system, which identifies the differences between the information extracted from two images[8]. SSIM adopts the assumption that the human visual perception system (HVS) is highly adapted for extracting structural information, and considers image degradations as perceived changes in structural information variation. Relating this to our case, we can best identify the structural information that changed between the image under default (non-bright conditions) setting and tuned setting under bright conditions (lamp, phone light). This quality assessment is a more enhanced form of measurement as compared

to metrics like mean squared error, which computes error between two images by simply quantifying the difference in the values of each of the corresponding pixels of images. So what makes SSIM better in quality than other metrics? SSIM extracts 3 key features from an image: 1) luminance 2) contrast 3) structure[8].

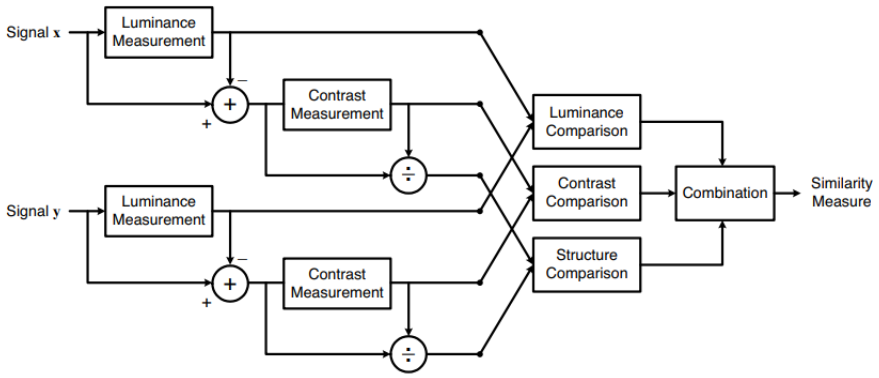


Figure 10: SSIM illustration

$$\text{Luminance}(x,y) = l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$
$$\text{Contrast}(x,y) = c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$
$$\text{Structure}(x,y) = s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

Two images are represented as x and y, with μ and σ representing the mean and standard deviation of the given images. C_1, C_2, and C_3 are constants to ensure stability in case the denominator becomes zero. SSIM score is given as the multiplication of these components with the relative importance of each metric: α, βandγ

$$\text{SSIM}(x,y) = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma$$

5.3 Object Segmentation

To evaluate the model we choose to use the Average Precision[9] which is the official metric used in the Microsoft COCO paper as well as the main metric returned by the Detectron2 network. Unlike SSIM and ATE mentioned earlier, Average Precision is a much more simpler metric that is used in several classification problems. Let's first start with Precision and Recall. The formulas are as follows

$$\text{Precision} = \frac{\text{Number of True Positives}}{\text{Number of True Positives} + \text{Number of False Positives}}$$
$$\text{Recall} = \frac{\text{Number of True Positives}}{\text{Number of True Positives} + \text{Number of False Negatives}}$$

These are simple metrics used in class imbalance problems as it is more revealing than accuracy. Precision is basically a measure of how much we can trust the model when it says positive (in this case positive is it classified an object in our image). Recall on the other hand is a measure of how good our model is at predicting positives overall. These 2 together share an inverse relationship. If you have a really high precision your recall tends to fall down. On the other hand a high recall means your precision will fall down. These 2 together form a curve known as the precision recall curve which is decreasing in nature. The area under this curve is what we call "Average Precision". This is taken a step further in most research papers such as AP50, AP75 and mAP. In our own paper we use

mAP. The numbers here (50 and 75) are the thresholds at which we determine what is a true positive and what is a false positive. When our model classifies an object it comes with a confidence level. In the case of AP50, anything with a confidence of 50% is considered positive and below negative. This is then compared to the actual labels to see if it is falsely classified or not. AP75 follows a similar principle. mAP is the mean of the average precision curve at all levels of confidence. In the COCO paper this is 10 splits from 5% to 95%. It also averages the precision across all categories. This is the most common metric used in papers and is what we will be using to serve as a baseline comparison for future research

Each model will be evaluated on the above metrics in a certain priority order (inputted into our repository pipeline). Whichever model performs best is used for final inference and video generation. Once it passes this stage we will shift the model weights to our physical car. This is the final test phase. If our car is able to navigate safely on our track with the help of the model then it means it was successful and we were able to achieve autonomous navigation and obstacle avoidance using the camera

6 Results

Each of the 3 experiments resulted in different things

6.1 Mapping

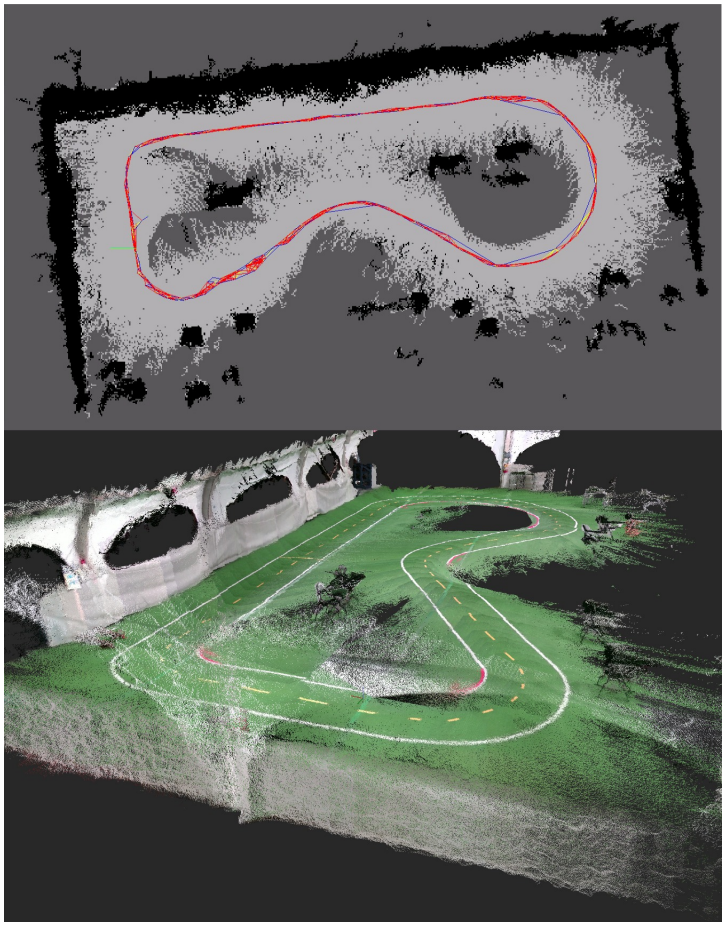


Figure 11: 2D and 3D maps

These are the 2D and 3D visualizations of the maps created of our track. In the 2D image, the black areas represent objects, light gray areas represent areas where the car has seen and dark gray areas represent areas where the car has not seen.

6.2 Light Conditions

6.2.1 Baseline

The baseline SSIM and MSE we want to improve from are 0.5922 and 19243, where we have compared images under non-bright and bright conditions, both using default camera settings.

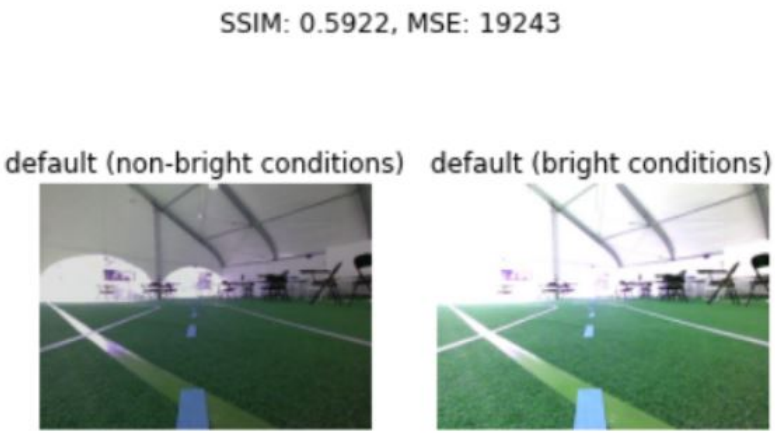


Figure 12: Baseline image comparison

6.2.2 Tuned

Configuration File	SSIM	MSE
Param1	0.7248	7231
Param2	0.6173	7981
Param3	0.6832	8435
Param4	0.6369	9066
Param5	0.4652	12903
Param6	0.3387	18349
Param7	0.7856	893
Param8	0.7456	746
Param9	0.8598	1823
Param10	0.7531	2598
Param11	0.8322	1398
Param12	0.5894	18094
Param13	0.9245	49
Param14	0.6419	3459
Param15	0.4255	16772
Param16	0.6047	17388
Param17	0.3249	19834
Param18	0.6757	5663
Param19	0.6821	6285

On seeing the top 5 configurations

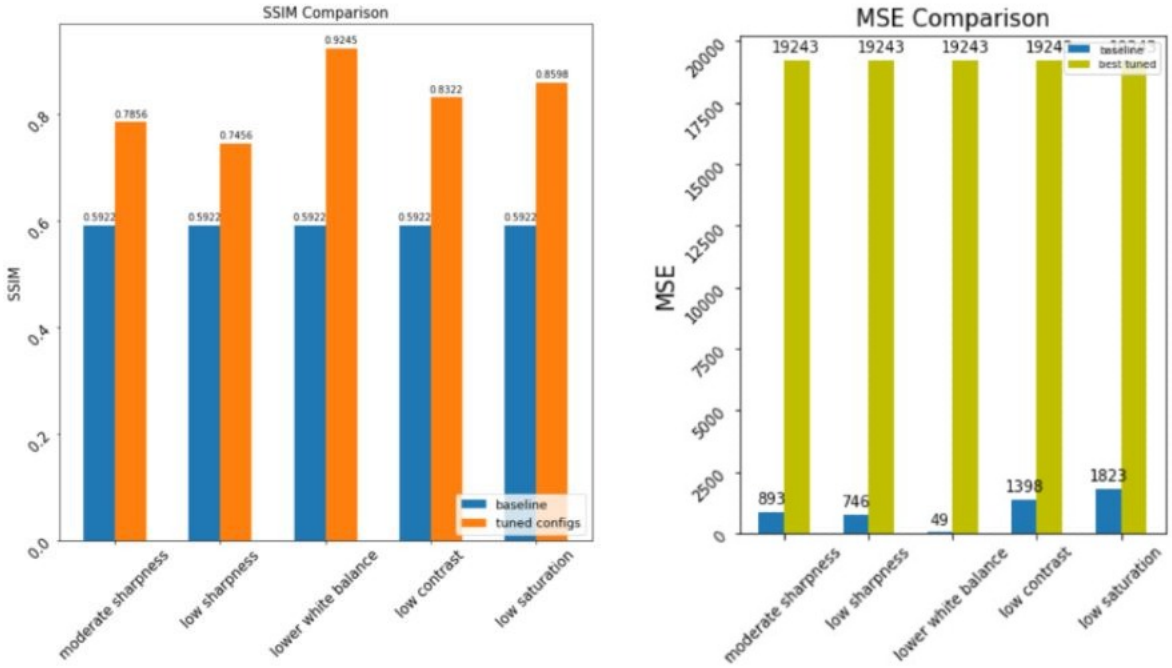


Figure 13: Top Configurations SSIM and MSE

Overall, lower levels of parameters were helpful in alleviating light intensity. Luminance, contrast and structure were appropriately balanced out while exposing the details of the image, which led to a noticeable rise in SSIM and reduction of pixel difference (MSE) to under 1000. The five configurations were successful in compensating for the loss of content caused by light, especially on the front left side of the tent that surrounds our track. The best tuned configuration we found through setting at 3500 (compared to the initial value at 4600), and we got SSIM to reach 0.9245 and MSE down to 49.

The following were the results of the best tuned configuration

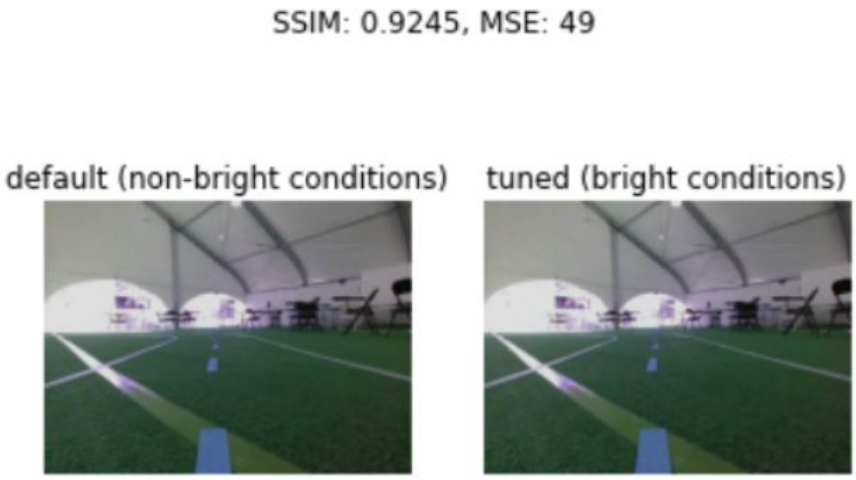


Figure 14: Best Configurations Image Comparison

On the other hand, the bottom 5 configurations

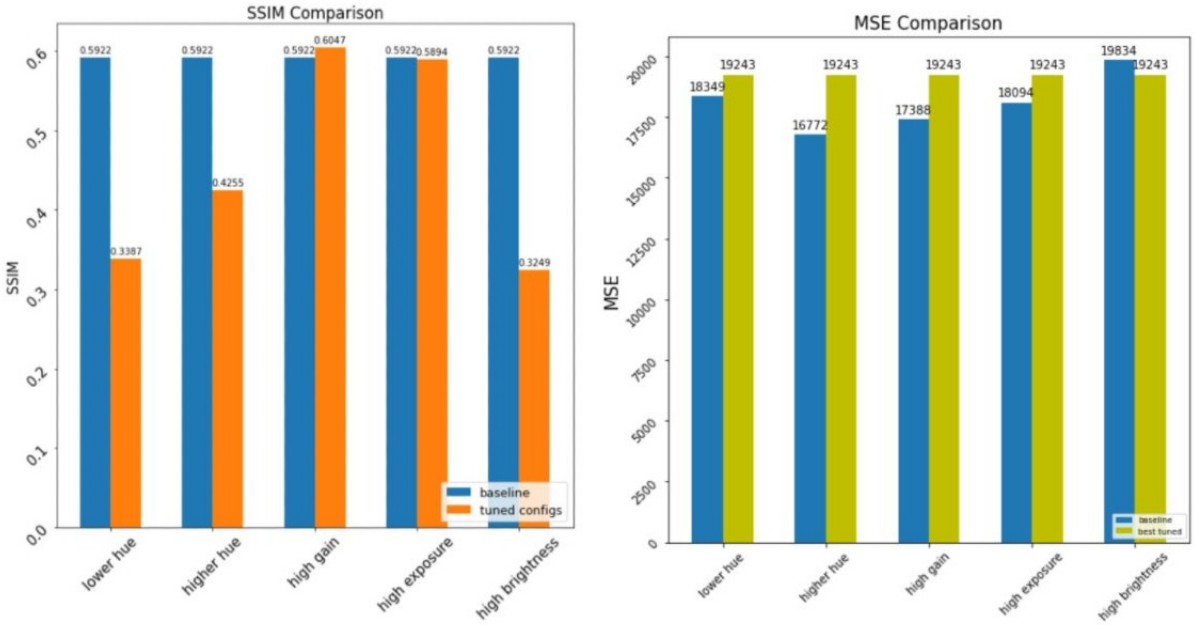


Figure 15: Bottom Configurations SSIM and MSE

Generally higher levels of parameters displayed minor improvements or made things worse, since the altered images deviated even more from the image with default configurations (under non-bright conditions).

6.2.3 Analysis

The top 5 parameters that were useful were: lower white balance, lower contrast, lower sharpness, lower contrast and lower saturation. Other minor parameter adjustments were used alongside each of these parameters, but these were the features that brought about dominant alterations to the non-tuned image under daylight conditions.

Lower white balance improved the results by compensating for the "color cast" imposed by the light (ground looking light-greenish to slight yellow)[10]. Moderate to low sharpness increased content and detail of the image while preventing bad artifacts in the image. Lower contrast minimized the difference in pixel (MSE) by offsetting the image's liveliness induced by light sensitivity of the camera. In terms of saturation, the difference in the levels of dominant color hues such as green (ground) and blue (middle, dotted lines) and remaining hues decreased with low saturation levels, once that led to shutting off dominant, colored hues that were being emphasized and intensified due to light.

Notable parameters that were not useful were: lower/higher hue, high gain, high exposure and high brightness.

Both lower and higher levels of hue parameter merely led to changes in color, unlike how the contrast parameter reduced color gradients to set off liveliness of the image. Increasing gain introduced electronic noise and degradation in depth quality (checked through Realsense Depth camera node), which also made it difficult for the tuned image to become similar in layout to the default image [4].

6.2.4 Realtime Performance Evaluation

Now that we have found the best tuned configuration that maximizes SSIM and minimizes MSE, how do we actually determine if this configuration is useful in terms of dynamic movement? The tuning of Intel D455 camera nodes was done with single, static images since that was the effective way to solely compare between the effects of parameter adjustments themselves. However, we also believed that it was important to integrate the viability of our tuned configuration to the car's entire runtime, which in our case is one lap around the track.

The dataset we use is the same, as we have already exported images from .bag collections using the image_view ROS package. Instead of selecting a single image to compare with another single image, we take the whole set of images recorded in one lap.

We evaluate the consistency of our best-tuned configuration's ability to offset light in realtime by observing the variability of pure luminescence across the set of images. If this configuration is able to offset light at one area of the track but does so in a relatively weaker amount in another area of the track, that demonstrates the variability of pure luminescence. The ultimate goal is to test whether or not the similarity level between realtime performance of default configuration images (non-bright conditions) and tuned configuration images (bright conditions) is high.

We measure that by taking RGB color code information using Python's ImageStat library, in order to compute the perceived brightness of all images in the set. We then simply calculate the standard deviation (measures variability) of this numeric figure to get the realtime performance. The returned values are shown below:

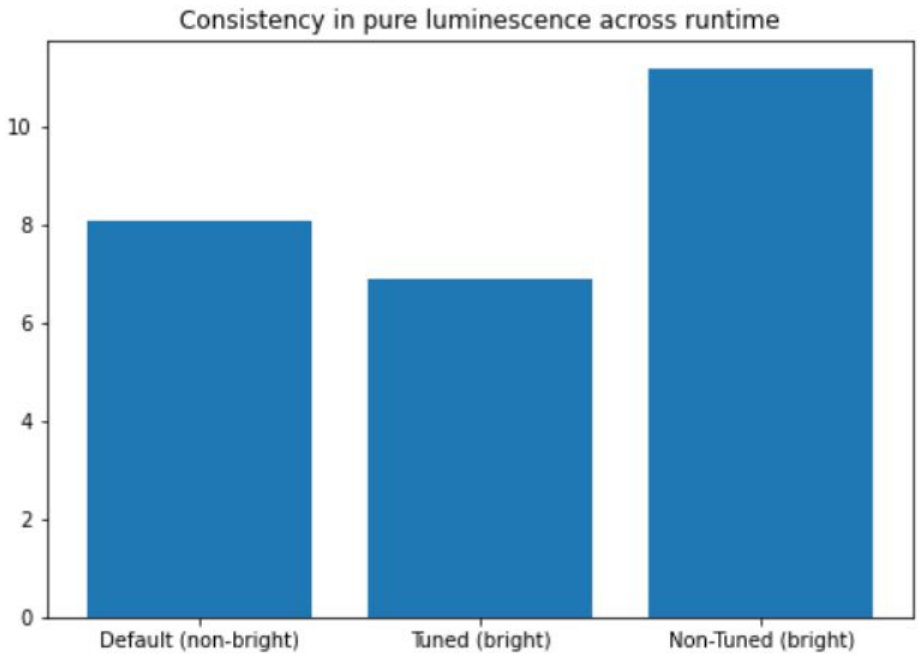


Figure 16: Runtime/realtime performance in light sensitivity

As expected, the image dataset with non-tuned configuration (under bright conditions) displayed the highest value, since images were exposed to different levels of light at different areas of the track. We found that the realtime performance between default configuration and tuned configuration is highly similar at 86%, compared to the initial realtime performance between default configuration (non-bright conditions) and non-tuned configuration (bright conditions) being 69% similar. This showed that we are 86% confident in deploying our best-tuned configuration in maintaining a consistent amount of light that it would offset across the given realtime.

Although our best-tuned configuration demonstrated a noticeable improvement in realtime performance across the car's runtime, the camera was still sensitive to light in certain areas of the track when the car drove autonomously in bright daylight conditions. This shows that there is still work that needs to be done to actually allow our camera or car to be more robust to lighting conditions

6.3 Object Segmentation

After experimenting with the hyper parameters we got

Epochs	Batch Size	Learning Rate	Train mAP	Validation mAP
100	64	0.02	83.7	81.3
100	16	0.02	84.9	81.8
100	8	0.02	86.3	83.9
200	8	0.02	90.6	79.9
200	8	0.01	89.9	87.4

Our baseline model was that of 100 epochs, batch size of 64 and a learning rate of .02 which was quickly dropped down due to a scheduler. Since our dataset was quite small we decided to drop down the batch size for more stochastic weight updates. This worked and we found 8 to be the optimal batch size. We next aimed to target epochs. However increasing the 200 epochs quickly led to overfitting on the data. We reduced the learning rate to compensate and the performance stabilized near 90%

This was a surprisingly good performance and when tested on track it provided near perfect inferences for the lane and cone objects

Using the Detectron2 Framework we can draw the segmentation and even a boundary box around each object in the image

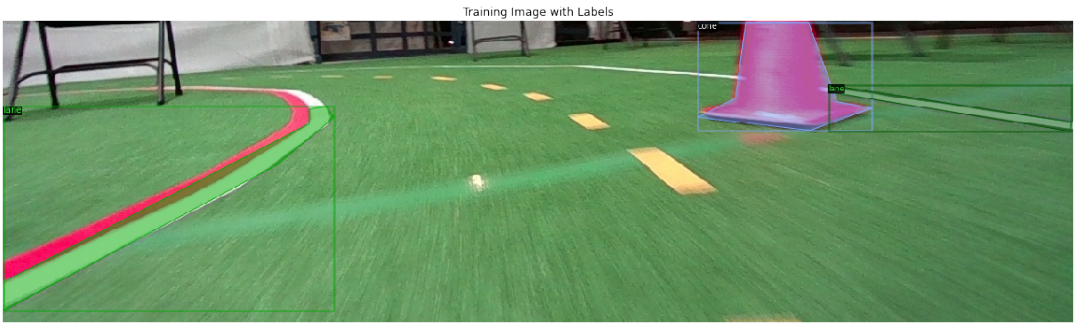


Figure 17: MaskRCNN Labelled image

However the model itself runs extremely slowly at around 5FPS since the inference takes 200 milliseconds. This required us to drive really slowly to adjust to the latency in predictions

6.3.1 Usage

The inference results were used to generate a white polygon. The white polygon was basically the area between the lanes and symbolized the areas we are allowed to drive in. The boundary box vertices of the lanes were used in this calculation since they precisely mapped the endpoints of the lane segmentation. The segmentation information from the cones was then overlayed. In cases of overlap between the cones and the white polygon we generated earlier we blacken out that section of the image. This means we are not allowed to go there

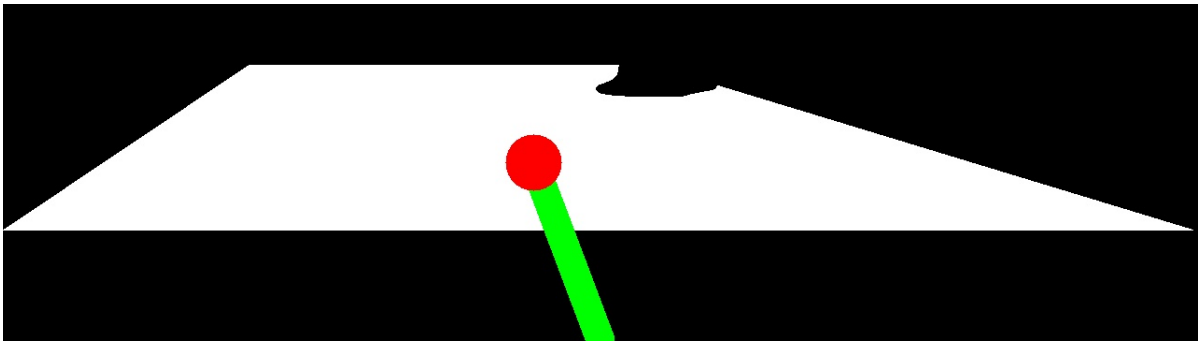


Figure 18: Processed driving image

The centroid of the image is then taken to choose a point to navigate towards in the white region. This point is converted into a throttle and a steering angle which is sent into our VESC to drive towards

Here is a flowchart of how this process looks

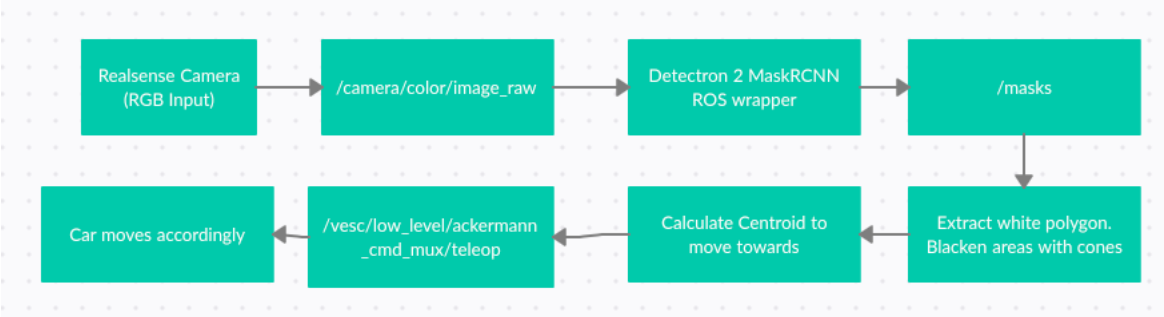


Figure 19: Driving ROS Flowchart

Here the /camera/color/image_raw and /masks are ROS topics that are used to communicate b/w the several nodes. We run the Detectron2 MaskRCNN wrapper separately to prevent re-initalizing of the model constantly

7 Conclusion

Overall we had some successed and some failures in our entire setup. The car was able to localize well but required a lot of laps to map effectively. Our car is able to drive safely in the track but it requires extremely slow speeds due to the MaskRCNN model having high enough inference times to not be usable in real world. The camera tuning was able to reduce the effect of light but not to the extent that it was reliable to deploy in the real world. Future improvements would be

- Use of additional sensory information and better parameters during mapping
- Use of cheaper models and offloading model inference to another chip (Like the OpenCV AI Kit Depth camera)
- Use of domain randomization to reduce the effect of lights

8 Team Contributions

Siddharth Saha: Wrote sections Abstract, Introduction, Environment and Conclusion. Wrote up section Object Segmentation under Experiment Design, Dataset, Evaluation and Result. Conducted research, designed and distributed research targets among team, Wrote object segmentation coding repository to generate and test models for inference in car. Wrote RTABMAP tuning repository to test several configurations at scale. Developed Dockerized container to run said repositories. Created navigation module based off Detectron2 results in ROS. Provided data collection instructions for collecting data for Object Segmentation, integrated and edited writeups to NIPS style report and assisted Jay and Youngseo in debugging problems in their topics. Working on OpenCV AI Kit Depth Camera and how to incorporate depth assisted segmentations at scale using their neural chips

Jay Chong: Data Collection for all 3 experiments. Lidar debugging, Camera debugging and VESC debugging of car. Integration of Mapping and Localization from last quarter to car. Debugging Transforms issue. Running tests with said mapping in real world track. Helped Youngseo debug and setup his testing environment, Wrote up section Mapping under Experiment Design, Dataset and Evaluation and Result. Set up remote connection to allow team members to work on car. Collaborated with Siddharth on using OpenCV AI Kit with Depth camera and set up camera on mount in real car

Youngseo Do: Researched and designed a pipeline of testing Intel Realsense camera configuration settings. Researched and came up with an evaluation metric to judge the sensitivity of different camera configurations against light and self-implemented an evaluation method for runtime performance of best-tuned configuration. Modified the launch file to handle multiple camera

configuration settings, Coordinated with Jay in collection of .bag files and results from exported data. Wrote the light tuning portions of the report (Experiment Design, Dataset, Evaluation, Results), website and presentation. Created the whole structure of code artifact (ETL, EDA, comparison, evaluate) for camera tuning

References

[1] M. Labbé and F. Michaud. Rtab-map as an open-source lidar and visual slam library for large-scale and long-term online operation. https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/7/7a/Labbe18JFR_preprint.pdf, 2019. in Journal of Field Robotics, vol. 36, no. 2, pp. 416–446, (Wiley).

[2] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2.

[3] Sagarnil Das. Simultaneous localization and mapping (slam) using rtab-map. https://arxiv.org/pdf/1809.02989.pdf, 2018.

[4] John Woodfill Anders Grunnet-Jepsen, John N. Sweetser. Best-known-methods for tuning intel® realsense™ d400 depth cameras for best performance. https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/BKMs_Tuning_RealSense_D4xx_Cam.pdf, 2020.

[5] Piotr Skalski. Make Sense. https://github.com/SkalskiP/make-sense/.

[6] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[7] Pranjal Datta. All about structural similarity index (ssim): Theory + code in pytorch. https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e, 2020.

[8] H.R. Sheikh E.P. Simoncelli Zhou Wang, A.C. Bovik. Image quality assessment: From error visibility to structural similarity. https://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf, 2004.

[9] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context. https://arxiv.org/pdf/1405.0312.pdf.

[10] Chiou-Shann Fuh Po-Min Wang. Automatic white balance with color temperature estimation. https://www.researchgate.net/publication/224693906_Automatic_White_Balance_with_Color_Temperature_Estimation, 2007.