Integration of the camera depth image

3EY4 - Lab 10 - Bonus

AV#23

Danial Noori Zadeh

400367734

Hassan Al Masmoum

400330500

Kevin Le

400385350

April , 2024

Table of Contents

[Problem Identification 3](#_Toc163556464)

[Solution Overview 3](#_Toc163556465)

[Design steps 4](#_Toc163556466)

[Step 1 – launching the camera node & listening to the camera messages. 4](#_Toc163556467)

[Step 2 – collecting camera information data 5](#_Toc163556468)

[Step 3 – receive and de-projecting camera depth data. 6](#_Toc163556469)

[Verification of deprojection algorithm in MATLAB 6](#_Toc163556470)

[Step 4 – Integration into LiDAR callback function 8](#_Toc163556471)

[Testing & Verification 9](#_Toc163556472)

[Static test 9](#_Toc163556473)

[AEV in action 9](#_Toc163556474)

# Problem Identification

We were tasked with integrating the depth cloud data published by the camera with the LiDAR to detect obstacles that are not on the plane of the LiDAR. Implied in this problem is the requirement of efficiency and rate matching between the two sources of information (LiDAR and depth camera). A mismatch of sampling time or the resolution of can cause mis-operation of the self-driving algorithm.

# Solution Overview

The camera publishes two topics that are key to our solution. One is the camera info that publishes information needed to convert depth (d) and pixel location (u, v) to xyz coordinates and later to polar coordinate to refine the LiDAR published ranges. The second important message topic is /camera/depth/image\_rect\_raw that contains the actual depth readings. We use CVBridge library to convert the image raw data to a matrix, but we were unable to pip install pyrealsense2 python wrapper to extract the data, so, we manually computed them from the camera intrinsic data.

The solution uses a range of interest (ROI) over both u and v axis of the image pixels to reduce the unnecessary calculations. Another method improve efficiency is to filter the extracted points based on a x and y window to avoid reading ground points and detection of objects that are out of the trajectory of the car. The results are confirmed by publishing the refined scan ranges and the performance was significant improved by optimizing the ROI (from 6 Hz publish rate to 12 Hz).

Figure 4-1. 
Depth Field of View to Depth Map illustration 
id Depth Band 
I Depth gand 
HF0v 
Va lid Depth 
Z, valid Depth 
Depth rov 
Depth FOV 
As the scerp"s distance from the ckpth module increases, the invalid depth band 
decreases in the overall depth image. Overall depth image is invalid depth band plus 
valid depth I-nap. 

# Design steps

## Step 1 – launching the camera node & listening to the camera messages.

We used the rs\_camera.launch file provided by the camera to realsense2 package, enabled the IR for extra precision and reduced the publish rate to 15 fps. However, it is recommended by the realsense documentation that both the depth and RGB camera operate at the same frequency and as high as possible, and if needed apply “decimation” filter to bring down the rate. We have tThe launch file is called in the self-driving package under experiment.launch as shown below.

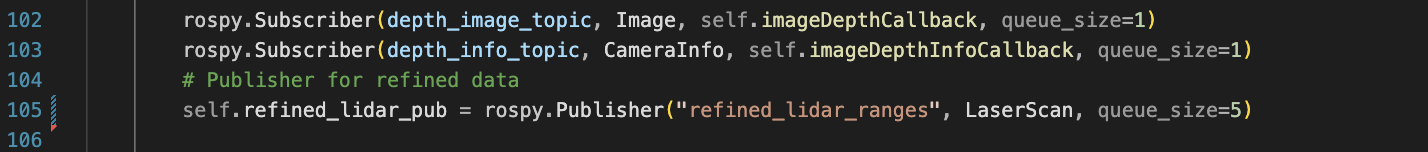
A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

The important camera depth data are published the highlighted topics, which we set up in the initialization of the navigation.py file as shown below. We need to import **Image** and **CameraInfo** datatypes from sensors.msg to launch these publishers and subscribers. The “refined\_lidar\_ranges” is the new topic published to debug the corrected ranges from the program.



A computer screen with white text

Description automatically generated

## Step 2 – collecting camera information data

Looking into a sample message from the camera\_info topic we can see the following fields which contain many useful information such intrinsic matrix (K) of the camera.

A screenshot of a computer

Description automatically generated

we then extracted the important information from the K field and stored them globally for the deprojection in the imageDepthCallback function.

A screen shot of a computer program

Description automatically generated

A close-up of a math problem

Description automatically generated

## Step 3 – receive and de-projecting camera depth data.

The xyz coordinates of the points in the camera depth image can be computed from the depth and pixel location (u and v). We used a reduced ROI to improve the efficiency of the deprojection process. We filtered out the ground points by filter the points based on their y-value. These settings can be changed in the params.yaml file.

|  |  |
| --- | --- |
| A blackboard with math equations  Description automatically generated |  |
| A computer screen shot of a program  Description automatically generated | |

The z value is compensated by self.cam\_baselink\_offset to translate the points into the LiDAR frame and then the points are expresses in polar coordinates. an important step before conversion to polar using numpy square root and inverse tangent is that the arrays have to be set as float type otherwise the function will exhibit erroneous behavior.

### Verification of deprojection algorithm in MATLAB

To verify the correctness of the deprojection function we made a MATLAB script that converts the image message to an image matrix, iterates over the depth points within the ROI and then computes the xyz coordinates. We used experimental data collected by placing a block that is out of the sight of the LiDAR at different distances away from the AEV. We monitored the output point cloud and tuned the ROI and y limits.

A screenshot of a computer

Description automatically generated

The red square displays the ROI setting. The advantage of performing the test in MATLAB is that we can use image processing tools such as hole filling filters to improve the readings.

A screenshot of a computer

Description automatically generated

## Step 4 – Integration into LiDAR callback function

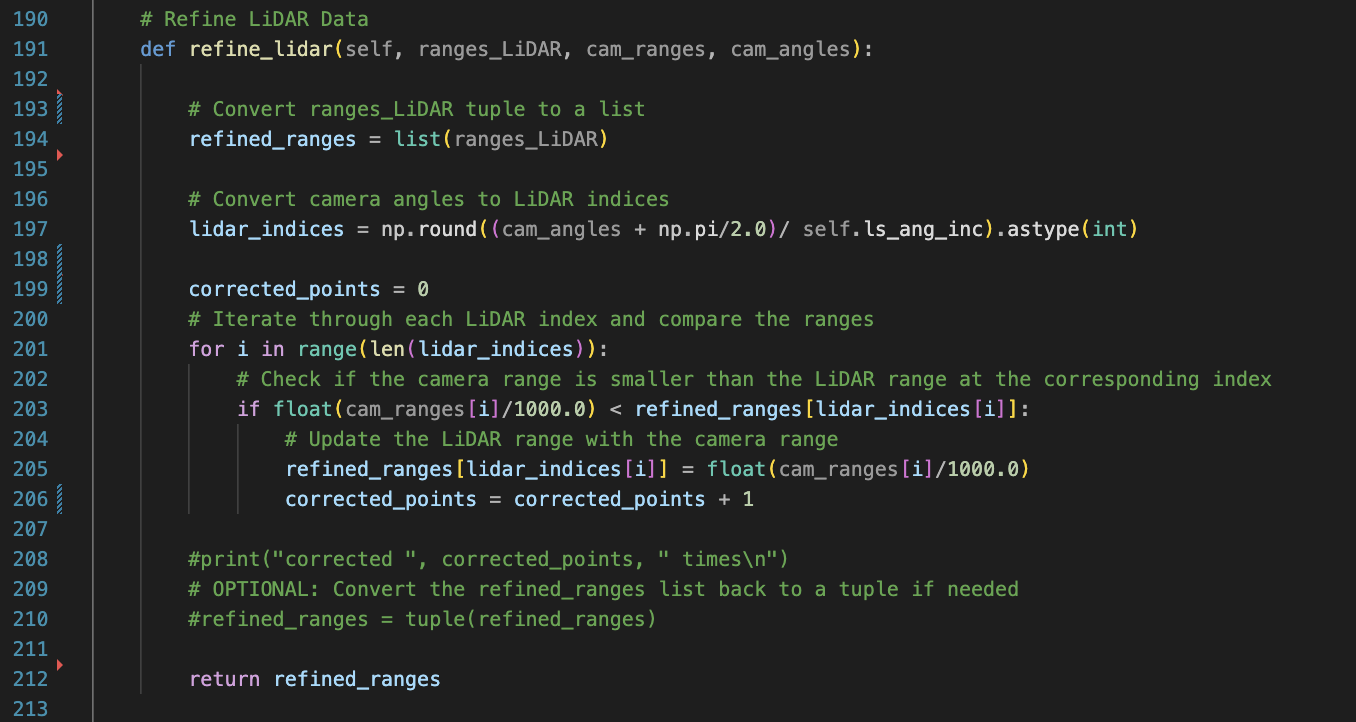
The LiDAR data needs to be refined before any pre-processing or navigation algorithm is run on it. We would only integrate the camera info if the global use\_camera variable is set to true, the camera intrinsic data is collected and the cam\_ranges have been extracted. We then pass the LiDAR ranges, camera ranges and corresponding angles to the refine\_lidar method and create a new LaserScan message for the refined ranges.

A computer screen shot of a program code

Description automatically generated

We first cast the ranges from LiDAR to a list, so it becomes mutable. We then find the closest lidar index that the cam\_angles correspond to by dividing the angle by the angle increment of the LiDAR. It’s important we add pi/2 to the angles to compensate for the LiDAR x axis being 90 degrees rotated with respect to the camera frame. We have already compensated for the translation between the two frames by adding the offset to the z-values.

The program will then iterate over the ranges in the camera, access the corresponding angle in the LiDAR frame and only overwrite the ranges if the camera suggests a smaller value. The corrected\_point is a debugging variable used to store the number of corrections made to the LiDAR ranges.



# Testing & Verification

## Static test

The published refined LiDAR was compared to the original LiDAR in rviz. We further verify the corrected points are coming from the camera’s depth cloud by enabling the depth cloud topic of the camera in rviz.

## AEV in action

The vehicle is able to avoid stationary objects assuming they are already in the camera’s line of sight; however, due to the difference in timing between LiDAR and the refined camera info the vehicle cannot avoid the objects that it did not have enough time to recognize.