Project WriteUP Final Sensor Fusion

This project deals with Extended Kalman Filter (EKF) and Sensor Fusion. This approach includes 4 different steps culminating in the fusion of lidar sensor data and camera sensor data. Those steps are filter implementation, track management, nearest neighbor data association, and camera and lidar sensor fusion.

Step 1: Implementation of EKF filter

For step 1, the goal was to implement an EKF to track a single real-world target with lidar measurement input over time. In student/filter.py, I have implemented the predict() function for an EKF.

The EKF is an adaptation of the original Kalman Filter that deals with linear motion.

For linear motion, the equation is: $x_k = f(x_{k-1}) + \nu = \mathbf{F}x_{k-1} + \nu$

However, because we are dealing with lidar data, we have to take into consideration that we now have to estimate a 2D position and 2D velocity. So, we have a 4D state vector: x = (px,py,vx,vy). Thus, the motion equation for EKF in 2D space is:

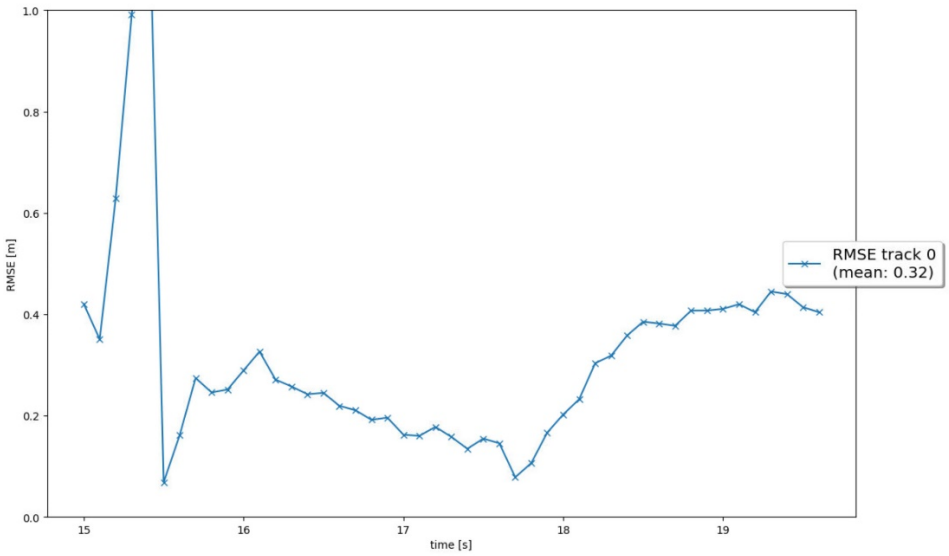
$$\begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} + \begin{pmatrix} \nu_{p_x} \\ \nu_{p_y} \\ \nu_{v_x} \\ \nu_{v_y} \end{pmatrix}$$

Therefore to calculate prediction, we will need to implement the state transition function F() as well as the covariance for the noise Q(). With those two functions implemented, implementing the predict() function becomes trivial.

An update must come after a prediction. Therefore, I needed to implement a function that would update the sensor measurements. To implement that update() function, I need to get the values for the measurement matrix (H), the residual (gamma), the covariance of the residual (S), the Kalman gain (K), the state update (x), and the covariance (P). While some of the measurements come directly from sensor readings, others have to be calculated. That is what I have done in the update() function. To complete the function, I had to implement the residual gamma() function and the covariance of the residual S.

Following the successful implementation of the update() function, I saved the updated value of the state x and covariance P by calling the functions set_x() and set_P() implemented in student/trackmanagement.py

Following a successful implementation of the predict() and update() functions, the RMSE yielded a mean of 0.32:



Step 2: track management

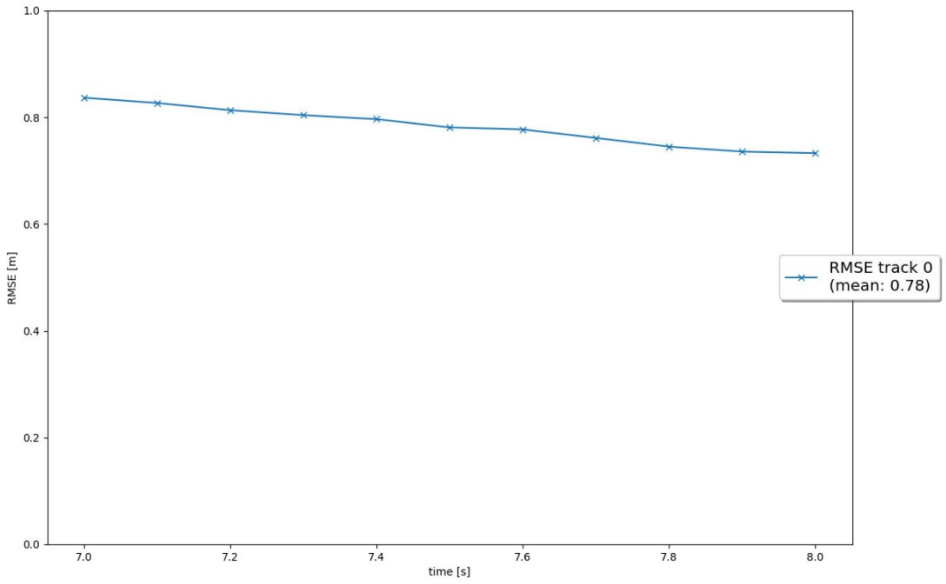
For this step, I needed to implement a track management class. I did so in the file student/trackmanagement.py. This file is where I have implemented classes that initialize and delete tracks, set track states, and assign track scores.

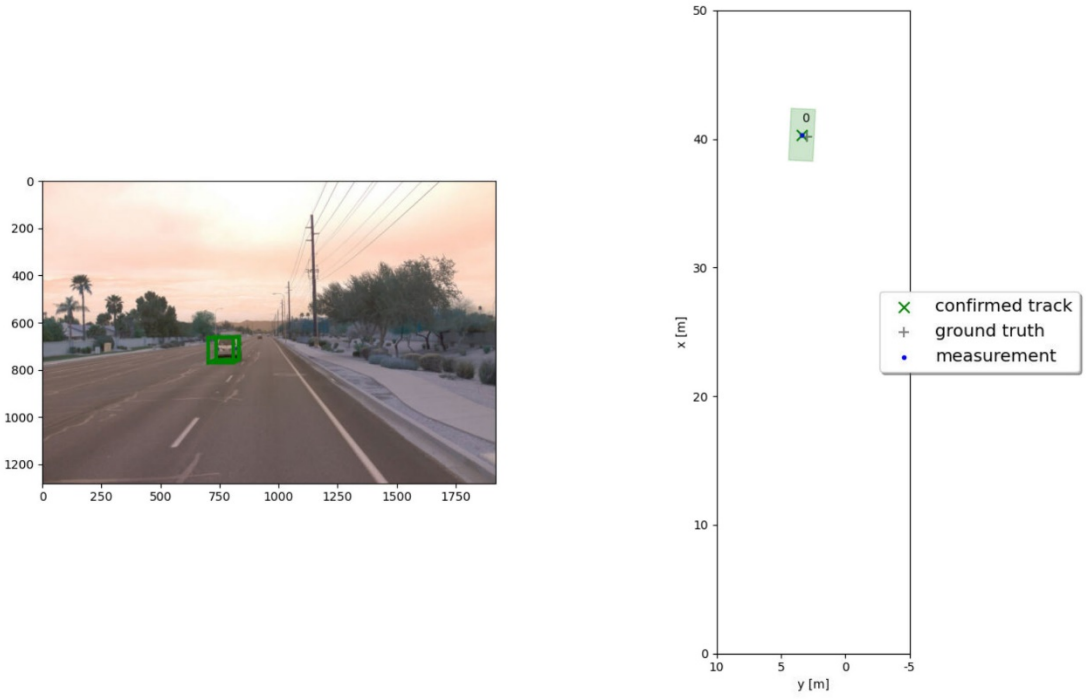
In the Track class, I initialized track.x and track.P based on the input (meas). This input is an unassigned lidar measurement object of type Measurement. Since unassigned measurements are in sensor coordinates, it is necessary to multiply the sensor coordinates by the transpose matrix sens_to_veh implemented in the Sensor class. This operation converts the initial coordinates into homogeneous vehicle coordinates. In that same function, I then initialize the track with a score of 1/params.window where window is the window size parameter.

I then proceeded to implement the manage_tracks() function where I decrease track scores for unassigned tracks, and delete tracks if the score is too low or P is too big by calling the delete_track() function that removes tracks from the track_list.

For complete track management, I implemented a function called handle_updated_track() that increases the track score for the input track and sets the track state to 'tentative' or 'confirmed' depending on the track score.

As seen below, the RMSE is quite high (0.78). In addition, the green boxes don't fit the car in the image very well.





This is because the lidar detections contain a y-offset. If the input has a systematic offset, the Kalman filter cannot compensate for it because we assume zero-mean data. We can, however, compensate for this offset through sensor fusion once we include other sensors.

Step 3: nearest neighbor data association

With the implementation of the nearest neighbor data association, I am able to implement multi-target tracking. The code for this step can be found within the file student/association.py. In the Association

class, I implemented the associate() function. In this function, I initialized an association matrix with infinity values and then looped over all tracks and measurements to set up the association matrix.

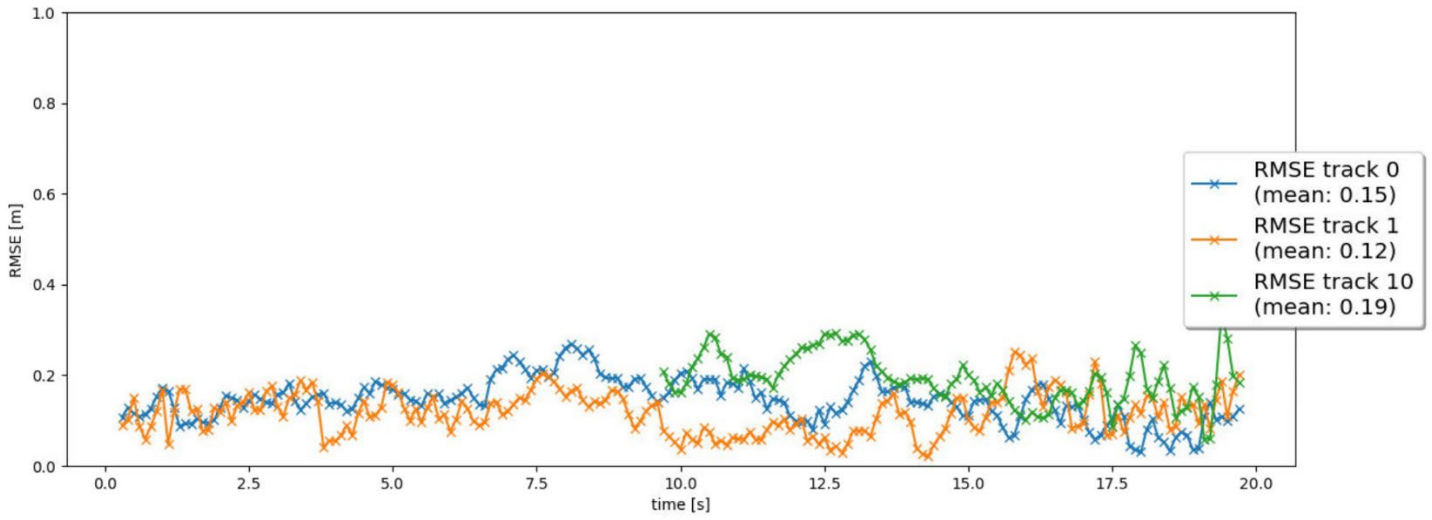
The process of associating the tracks in the track_list with the correct measurements requires calculating the Mahalanobis distance between a track and a measurement. Once the Mahalanobis distance is calculated, it's important to check if a measurement lies inside a track's gate. Since those calculations are constantly being repeated, I implemented the MHD() function to calculate the Mahalanobis distance and the gating() function that determines if a measurement lies inside the track gate and returns False if it doesn't.

When the gating function returns False, the value in the association matrix remains infinity. However, when the returned value is valid, the infinity value in the association matrix is replaced with the mhd_dist.

For our multi-target tracking to work, we need to continuously associate and update new measurements as they come in. This is the purpose of the associate_and_update() function. We are constantly searching for the next association between a track and a measurement and thus, we need to continuously determine which track is closer to a specific measurement. For this purpose, I have implemented the get_closest_track_and_meas() function.

The get_closest_track_and_meas() function finds the minimum entry in the association matrix, deletes the corresponding row and column from the association matrix, removes the corresponding track and measurement from the list of unassigned_tracks and the list of unassigned_meas and returns the minimum entry track and measurement. It returns np.nan (NaN == Not a Number) where we find infinity in the association matrix.

We loop through the association matrix until there are no more associations. While multiple tracks are updated with multiple measurements, each measurement is used at most once. As can be seen in the RMSE below, there is no confirmed ghost track.



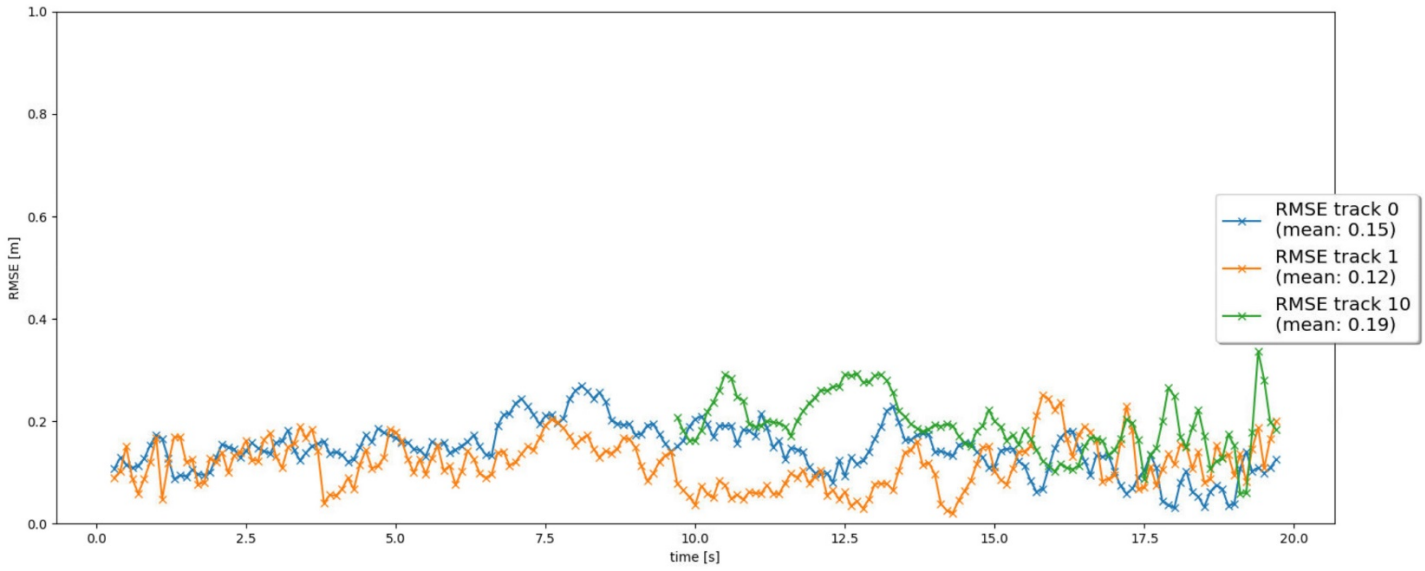
Step 4: fuse measurements from lidar and camera

In this step, I implemented a nonlinear camera measurement model and completed the sensor fusion module for camera-lidar fusion. The code for this module is found in student/measurements.py

In the Sensor class, I implemented the function in_fov() that checks if the input state vector x of an object can be seen by this sensor. The function returns True if x lies in the sensor's field of view. Otherwise, it returns False. I start by converting the sensor coordinates to homogeneous coordinates then, I exclude zero and the negative range before calculating the angle alpha between the object and the x-axis. Since the returned alpha always falls between [-pi/2, pi/2], there is no need to normalize. I determine the visibility of the object by checking if the returned alpha falls in the given field of view of this particular camera. In this particular case, the angle of the field of view in radians is: fov = [-0.35, 0.35].

I then implemented the calculation of the nonlinear measurement expectation value h(x) for the camera in the aptly named get_hx() function. Since data from the camera sensor is received in camera coordinates, I first had to convert it to homogeneous coordinates. Then, I did a projection from camera to image coordinates making sure not to divide by zero. The resulting hx was then returned for the camera sensor. In the measurement class, I initialized the camera measurement objects including z, R, and the sensor object sensor. Even though they could be measured with a camera, I have left out some attributes such as width and yaw from the camera measurements. In our implementation of sensor fusion, they are only used in lidar measurements.

Following the fusion, the resulting RMSE showed improvement in the speed of track confirmation and the elimination of ghost tracks.

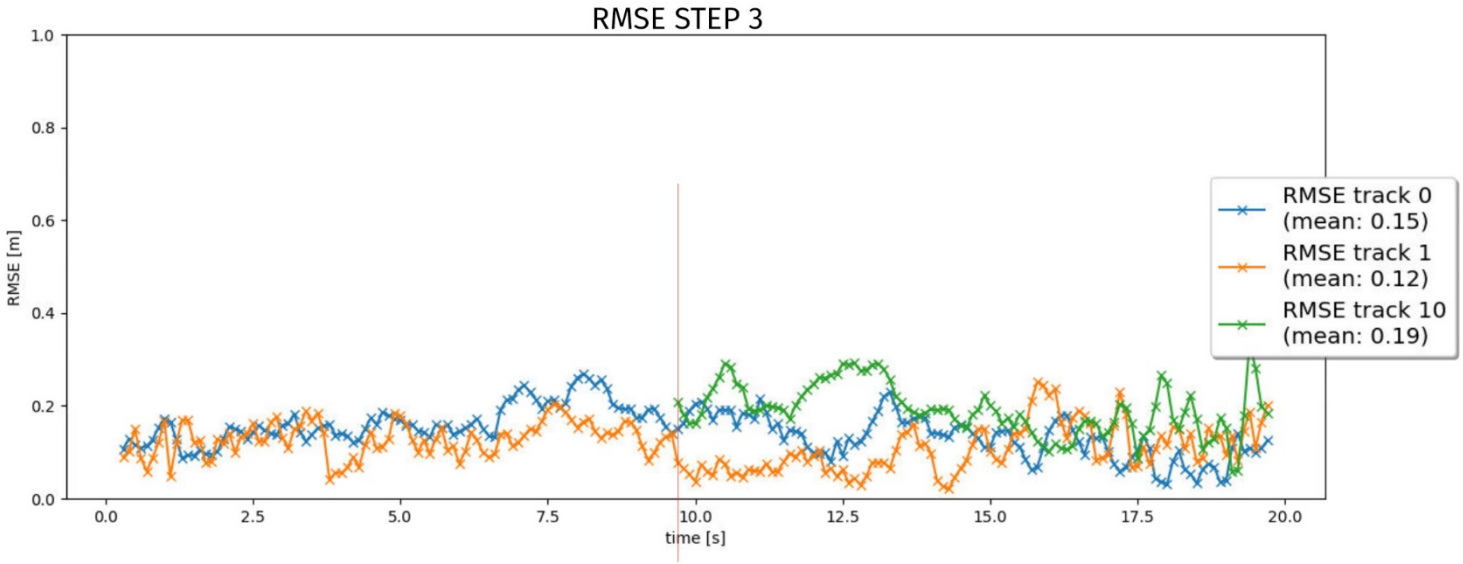


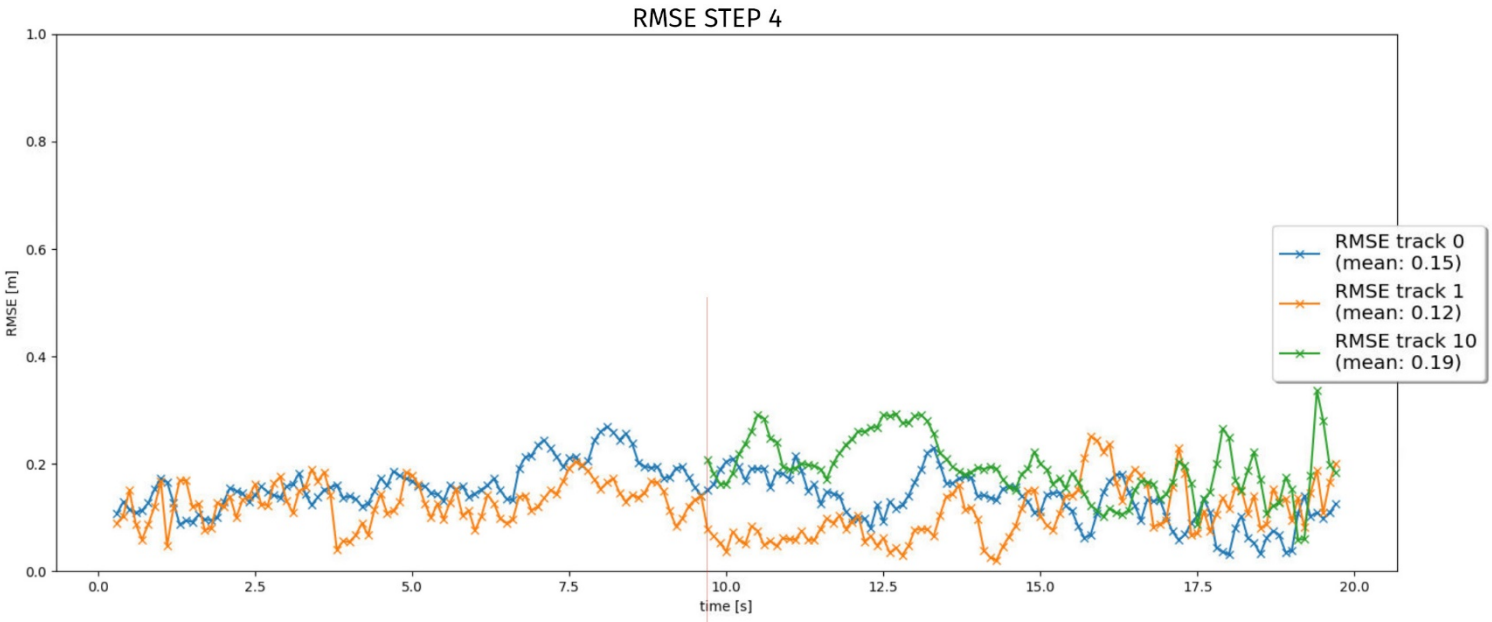
Are there any benefits in the fusion of Lidar sensor data and Camera sensor data?

Absolutely yes! Each has its own strengths and weaknesses. Unlike cameras, Lidars function independently of ambient lighting. They can achieve great results day and night without any loss of performance. On the other hand, Lidars cannot detect color or read text. Consequently, it would be

impossible for a Lidar-only system to detect traffic lights or read road signs. On the other hand, cameras excel at both of those tasks.

I have also noticed that cameras speed up the track confirmation process while making the elimination process faster for ghost tracks. I have included a comparison between the RMSE of section 3 where only Lidar data was used and the one in section 4 which included a fusion of Lidar and camera data. Around the 10-second mark, I have drawn a line going through the location where track 10 is initially confirmed. You will notice that for the RMSE in step 3, the line goes through the m in time[s] while for the RMSE in step 4, it goes through the I in time[s]. This means that the track for step 4 was confirmed a few milliseconds faster than the track in step 3. It might not seem like much but a difference of a few milliseconds could make a big difference in avoiding an accident.





Which challenges will a sensor fusion system face in real-life scenarios?

In definition, sensor fusion is relatively simple. It's just a matter of getting reading from multiple sensors and combining them together. However, that's easier said than done. Each sensor provides a different type of data that must be handled accordingly. For example, camera images require complex computations. Once an image is received, the system must first identify which of the data within an image is relevant. That takes a lot of CPU cycles. Just look at the following code that calculates the Jacobian H.

In the case of lidar data, we have

```
if self.name == 'lidar':
        H[0:3, 0:3] = R
```

Calculating H for camera data is a lot more involved as evidenced by the code below

```
 elif self.name == 'camera':
        # check for division by zero
        if R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0] == 0:
           raise NameError('Division by 0')
        else:
           H[0,0] = self.f_i * (-R[1,0] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,0] * (R[1,0]*x[0] + R[1,1]*x[1] + R[1,2]*x[2] + T[1]) \
                          / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
           H[1,0] = self.f_j * (-R[2,0] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,0] * (R[2,0]*x[0] + R[2,1]*x[1] + R[2,2]*x[2] + T[2]) \
                          / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
           H[0,1] = self.f_i * (-R[1,1] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,1] * (R[1,0]*x[0] + R[1,1]*x[1] + R[1,2]*x[2] + T[1]) \
                          / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
           H[1,1] = self.f_j * (-R[2,1] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,1] * (R[2,0]*x[0] + R[2,1]*x[1] + R[2,2]*x[2] + T[2]) \
                          / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
           H[0,2] = self.f_i * (-R[1,2] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,2] * (R[1,0]*x[0] + R[1,1]*x[1] + R[1,2]*x[2] + T[1]) \
                          / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
```

```
        H[1,2] = self.f_j * (-R[2,2] / (R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])
                        + R[0,2] * (R[2,0]*x[0] + R[2,1]*x[1] + R[2,2]*x[2] + T[2]) \
                        / ((R[0,0]*x[0] + R[0,1]*x[1] + R[0,2]*x[2] + T[0])**2))
    return H
```

All throughout this sensor fusion project, I have had to deal with the complications of dealing with camera data, transforming the coordinates, and of course, the use of object detection through resnet. With sensor fusion, there is also the matter of calibrating the sensors and determining which one should be prioritized. Furthermore, let's not forget that each sensor requires computing power for the interpretation of the data sent by that sensor; the more sensors we integrate, the more computing power we will need. In spite of the aforementioned challenges, we cannot eliminate sensor fusion if we hope to one day achieve Level 4 and 5 of autonomous driving.

Can you think of ways to improve your tracking results in the future?

Without a doubt, adding more sensors will improve the tracking results and improve safety. For this project, we are reading data from only one Lidar and one camera. Adding additional sensors would make it possible for the vehicle to "see" better, even under adverse conditions, and with a complete 360 degrees field of view. That's why most Level 3 vehicles integrate lidar with visual and IR cameras, ultrasonic sensors, and radar arrays since camera sensors are severely impaired by adverse weather conditions while the range of Lidars dramatically decreases under such conditions.