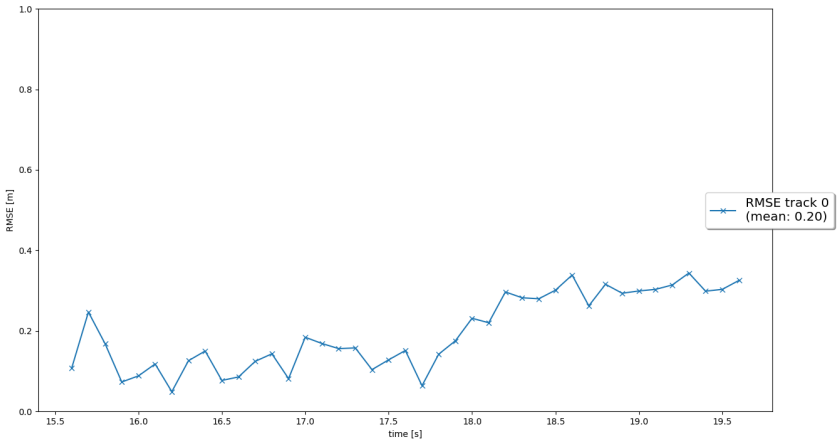
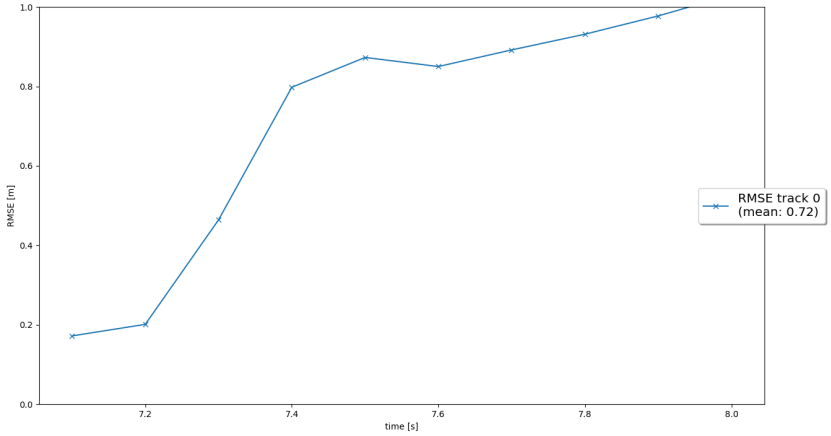
• In the second part of this project we have gone through these four steps:

1- Implementation of a EKF for sensors which include linear (lidar) and non-linear measurement models (camera). The EKF filter consists of a *Predict* stage, in which a track state vector is estimated via a linear motion model (**F**() and **Q**() functions), and an *Update* stage, in which data from sensor(s) refines the predicted state vector. For non-linear measurement models it is necessary to linearize the model around the current track position by obtaining the Jacobian of the non-linear transformation at the given point. The following plot was obtained in Step 1:

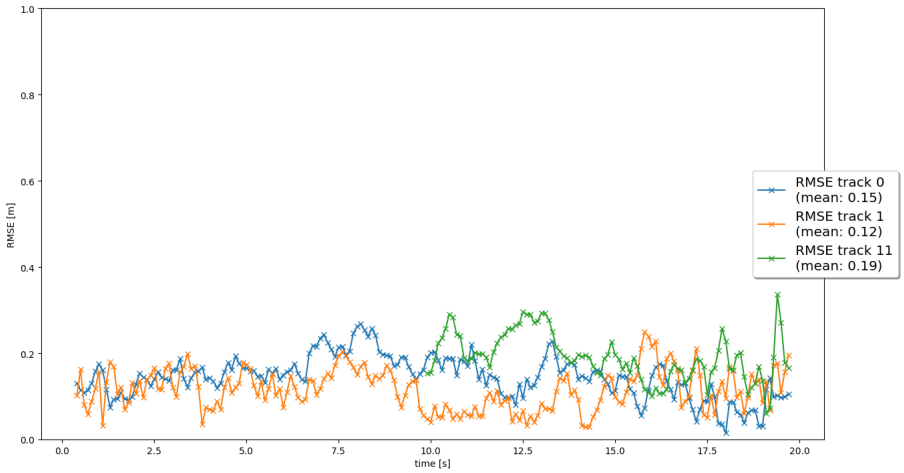


2- Implementation of a track management system to manage track states. This system takes care of:
   a. Track initialization: creates and associates a brand-new track to an unassociated measurement.
   b. Track score update: based on the number of consecutive detections.
   c. Track state transition: based on the score, the track can transition from **tentative** to **confirmed** if it goes above a threshold, and from **confirmed** to **deleted** if it drops below another threshold.
   d. Track deletion: the track is removed when transitioning to state **deleted.**

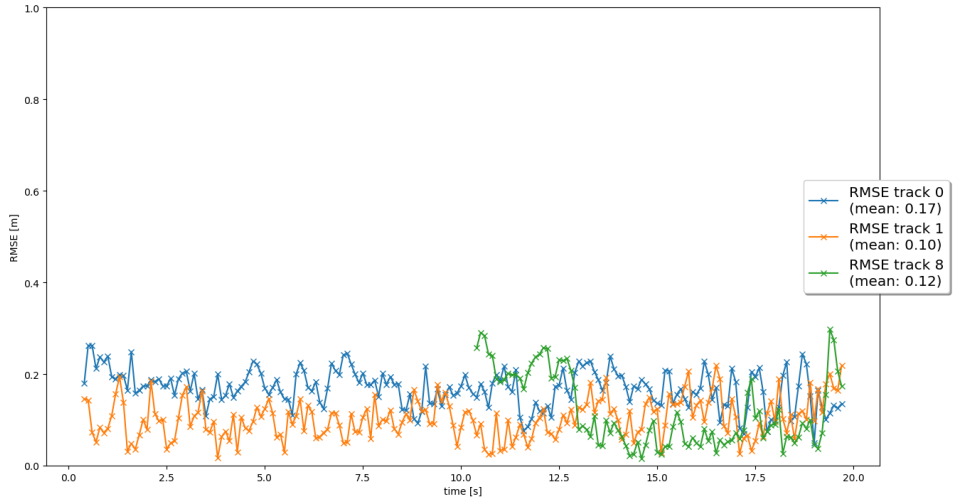
   The following plot was obtained at Step 2:



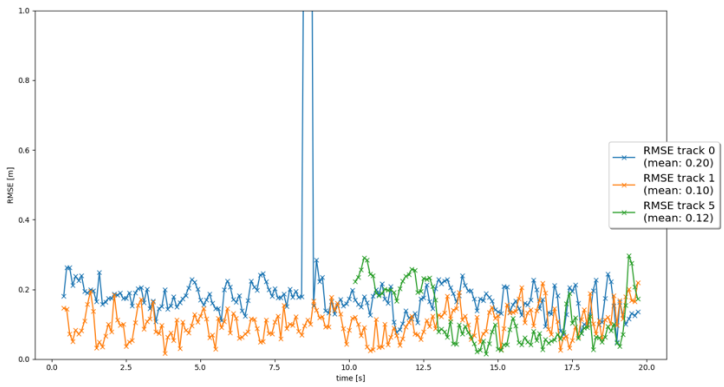
3- Implementation of a track-to-measurement association system, via Nearest Neighbor according to Mahalanobis distance. To achieve this, a matrix of track-to-measurement Mahalanobis distance is constructed, and then the smallest distances associations are extracted from this matrix. Gating to the accumulated probability of 0.995 of the chi squared distribution derived from track and measurement covariance matrices (which is the Mahalanobis distance) is applied to discard very unlikely associations. These confirmed tracks are obtained in the sequence, yielding these **RMSE**s (At this point, only the LIDAR data is being used):



4- Introduce Sensor Fusion by taking into account data from the camera sensor into the EKF filter loop. Implemented the functions Sensor.in_fov() to check if a point in physical 3D space can be seen by a given sensor. Also, the Sensor and Measurement classes were extended to include camera functionality. Three confirmed tracks were obtained as well, with generally slightly lower RMSE. The exception is track 0 (belonging to the car in front to the left)



• The part I found the greatest difficulty was step 4, in the sense that, with all the mechanisms and pieces up and running, it took me a good two hours to track and correct a bug in my code that prevented a track with big uncertainty from being deleted. This bug caused the tracks to be mixed up at some point, causing this big spike in **RMSE** in one frame:



• The benefit of using sensor fusion over using just the lidar data is that the overall **rmse** is decreased (albeit only slightly in the given scenario). We should also obtain a system that is more robust and less susceptible of false or clutter positives.

• A few challenges I can think of are:

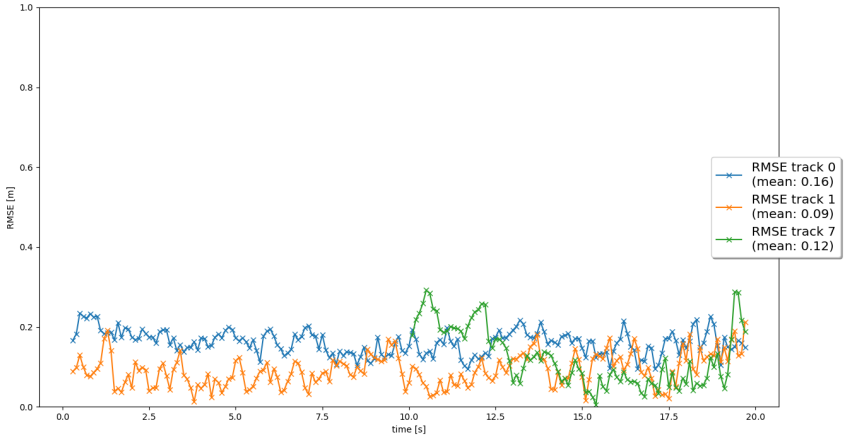
- Sensor obstruction: dirt in the lens or heavy rain can interfere with camera accuracy, introducing a greater uncertainty than the one reported by the maker.
- Sudden changes in the estimation when the detected objects get in and out sensor fov's due to slight sensor misalignments.

• Ways to improve tracking:

- Use something more sophisticated than an EKF, like an UKF or a Particle Filter (although processing time is a critical factor; it is essential to find a balance between performance and quality)
- Use additional sensor information (all lidars and all cameras available in the dataset)
- Further train the CNN models to improve detection accuracy.

MAKING MY PROJECT STAND OUT

- After spending a few cycles trying to find better parameters, I got a slight reduction of average **RMSE** of to 12.333 from 13. Set the flag FINE_TUNE in params.py to True to use these settings.



- Global Nearest Neighbor implemented in association.py, instead of Local Nearest Neightbor. Was implemented by leveraging the Hungarian Algorithm provided by the Munkres library. Enable by setting GLOBAL_NEAREST_NEIGHBOR = True In params.py. The key pieces of code are shown here:

```
if GLOBAL_NEAREST_NEIGHBOR:
            # Using the Hungarian Algorithm to compute the best associations globally
            self.associations = m.compute(self.association_matrix.A.tolist())
            self.last_association_index = 0
```

```
def get_closest_track_and_meas(self):

        if GLOBAL_NEAREST_NEIGHBOR:
            if self.last_association_index == len(self.associations):
                return np.nan, np.nan
            update_track = self.associations[self.last_association_index][0]
            update_meas = self.associations[self.last_association_index][1]
            self.last_association_index += 1
            self.unassigned_tracks.remove(update_track)
            self.unassigned_meas.remove(update_meas)
            return update_track, update_meas
```

- Detection from part 1 of the project were used. Video file is included (part_1_detections.mp4)

- To estimate the dimensions of detected objects (width, length and height), three new variables have been added to the state vector, making it 9-dimensional. The matrix for the linear motion model has been changed to fit this new size:

```
return np.matrix([
                [ 1,  0,   0,   params.dt,  0,          0,          0, 0, 0 ],
                [ 0,  1,   0,   0,          params.dt,  0,          0, 0, 0 ],
                [ 0,  0,   1,   0,          0,          params.dt,  0, 0, 0 ],
                [ 0,  0,   0,   1,          0,          0,          0, 0, 0 ],
                [ 0,  0,   0,   0,          1,          0,          0, 0, 0 ],
                [ 0,  0,   0,   0,          0,          1,          0, 0, 0 ],
                [ 0,  0,   0,   0,          0,          0,          1, 0, 0 ],
                [ 0,  0,   0,   0,          0,          0,          0, 1, 0 ],
                [ 0,  0,   0,   0,          0,          0,          0, 0, 1 ],
            ])
```

These three new quantities will be treated in a similar manner to the velocities: the model assumes they are preserved (thus the identity in the 6x6 right-bottom block), and the initial guess (0,0,0 for velocities and 1,1,1 for wight, height and length) will be updated by the filter at the *Update* step. A few more changes have been needed to accommodate this new 9D state vector (the shapes of the Q process noise matrix, Track.x and Track.P, and now the lidar sensor **dim_meas** is 6 instead of 3: it also reports measured width, height and length).

This can be activated by setting the flag ESTIMATE_DIMENSIONS in params.py to True

- As non-linear motion model, I have used a very simplified model for the motion of a bicycle. Given timestep dt:

$$
\begin{aligned}
x' &= x + v \cdot \cos(\theta)\ \Delta t \\
y' &= y + v \cdot \sin(\theta)\, \Delta t \\
z' &= z \\
\theta' &= v \cdot \sin(\delta) \\
v' &= v \\
\delta' &= \delta
\end{aligned}
$$

  This model presents a non-holonomic constraint which prevents the vehicle from sliding to the sides (can only roll forward or backward), and assumes that the z coordinate (height), speed and the front wheel angle are going to remain constant.

  Implementing this step allowed me to gain a deeper understanding on how the Kalman filter updates variables that are kept constant by the motion model (like velocities in the linear model used in the project, which are initialized to 0).

  This model can be used by setting USE_BICYCLE_MODEL in params.py to True

  The functions F (both the non-linear and the linearized matrix form), have been added to the code base, as shown here:

```
    def bicycle_non_linear_F(self, state_vector):
        '''Non-linear bicycle model with vector state [x, y, z, theta, v, delta]
            Where x, y, z is the position in 3D
            x and y are updated from bicycle velocity v, body angle theta, and
front wheel angle delta
            z is assumed to be preserved throughout the trajectory in this simple
model
        '''
        x = state_vector[0, 0]
        y = state_vector[1, 0]
        z = state_vector[2, 0]
        theta = state_vector[3, 0]
        v = state_vector[4, 0]
        delta = state_vector[5, 0]
        return np.matrix([
            [ x + v*np.cos(theta) * params.dt ],  # x
            [ y + v*np.sin(theta) * params.dt ],  # y
            [ z ],                                # z
            [ v*np.sin(delta) ],                  # theta
            [ v ],                                # v
            [ delta ]                             # delta
        ])
```

```
def bicycle_linear_F(self, x):
        '''Non-linear bicycle model linearized around point [x_0, y_0, z_0,
theta_0, v_0, delta_0]'''
        theta = x[3, 0]
        v = x[4, 0]
        delta = x[5, 0]
        return np.matrix([
            [1, 0, 0, -v*np.sin(theta)*params.dt, np.cos(theta)*params.dt, 0],
            [0, 1, 0,  v*np.cos(theta)*params.dt, np.sin(theta)*params.dt, 0],
            [0, 0, 1,  0,                         0,                       0],
            [0, 0, 0,  0,                         np.sin(delta), v*np.cos(delta)],
            [0, 0, 0, 0,                          1,                       0],
            [0, 0, 0, 0,                          0,                       1]
        ])
```

Along with its corresponding process noise Q matrix (very messy here, sorry!):

```
def bicycle_Q(self, state_vector):
        '''Linearized and integrated process noise Q around point [x_0, y_0, z_0,
theta_0, v_0, delta_0]'''
        self.q_v = 50
        self.q_d = 1
        # x: state_vector[0], y: state_vector[1], z: state_vector[2], theta:
state_vector[3], v: state_vector[4], delta: state_vector[5]
        theta = state_vector[3, 0]
        v = state_vector[4, 0]
        delta = state_vector[5, 0]
        return np.matrix([
            [ 1/3 * v**2 * self.q_v * np.sin(theta)**2 * params.dt**3,
-1/3 * v**2 * self.q_v * np.sin(theta) * np.cos(theta) * params.dt**3,    0,     -
1/2 * v * self.q_v * np.sin(delta) * np.sin(theta) * params.dt**2,
-1/2 * v * self.q_v * np.sin(theta) * params.dt**2,     0 ],
            [ -1/3 * v**2 * self.q_v * np.sin(theta) * np.cos(theta) *
params.dt**3,    1/3 * v**2 * self.q_v * np.cos(theta)**2 * params.dt**3,
0,     1/2 * v * self.q_v * np.sin(delta) * np.cos(theta) * params.dt**2,
1/2 * v * self.q_v * np.cos(theta) * params.dt**2,      0 ],
            [ 0,
0,                                                                    0,     0,
0,                                                  0 ],
            [ -1/2 * v * self.q_v * np.sin(delta) * np.sin(theta) * params.dt**2,
1/2 * v * self.q_v * np.sin(delta) * np.cos(theta) * params.dt**2,       0,
(v**2 * self.q_d * np.cos(delta)**2 + self.q_v * np.sin(delta)**2) * params.dt,
self.q_v * np.sin(delta) * params.dt,                   v * self.q_d *
np.cos(delta) ],
            [ -1/2 * v * self.q_v * np.sin(theta) * params.dt**2,
1/2 * v * self.q_v * np.cos(theta) * params.dt**2,                      0,
self.q_v * np.sin(delta) * params.dt,
self.q_v * params.dt,                                   0 ],
            [ 0,
0,                                                                    0,     v *
self.q_d * np.cos(delta) * params.dt,
0,                                                  self.q_d * params.dt ]
        ])
```