Writeup: Track 3D-Objects Over Time

1. Write a short recap of the four tracking steps and what you implemented there (filter, track management, association, camera fusion). Which results did you achieve? Which part of the project was most difficult for you to complete, and why?

In this project, I worked on four different modules of sensor fusion algorithm including filter, track management, association and camera fusion. The first part of the project dealt with the calculation of Extended Kalman Filter matrices. The aim was to track a single target. The first action was to update the matrices of 2D vehicle kinematic model to 3D model.

The state vector of 3D model is:

$$x = \begin{Bmatrix} p_x \\ p_y \\ p_z \\ v_x \\ v_y \\ v_z \end{Bmatrix}$$

And the state transition matrix is:

$$\begin{Bmatrix} p_x \\ p_y \\ p_z \\ v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} p_x \\ p_y \\ p_z \\ v_x \\ v_y \\ v_z \end{Bmatrix} + \begin{bmatrix} v_{px} \\ v_{py} \\ v_{pz} \\ v_{vx} \\ v_{vy} \\ v_{vz} \end{bmatrix}$$

If we assume the noise through acceleration in x, y and z to be equal, $v_x = v_y = v_z$, the continuous process noise covariance Q can be modelled as:

$$Q = \begin{bmatrix} q_1 & 0 & 0 & q_2 & 0 & 0 \\ 0 & q_1 & 0 & 0 & q_2 & 0 \\ 0 & 0 & q_1 & 0 & 0 & q_2 \\ q_2 & 0 & 0 & q_3 & 0 & 0 \\ 0 & q_2 & 0 & 0 & q_3 & 0 \\ 0 & 0 & q2 & 0 & 0 & q_3 \end{bmatrix}$$

Where:

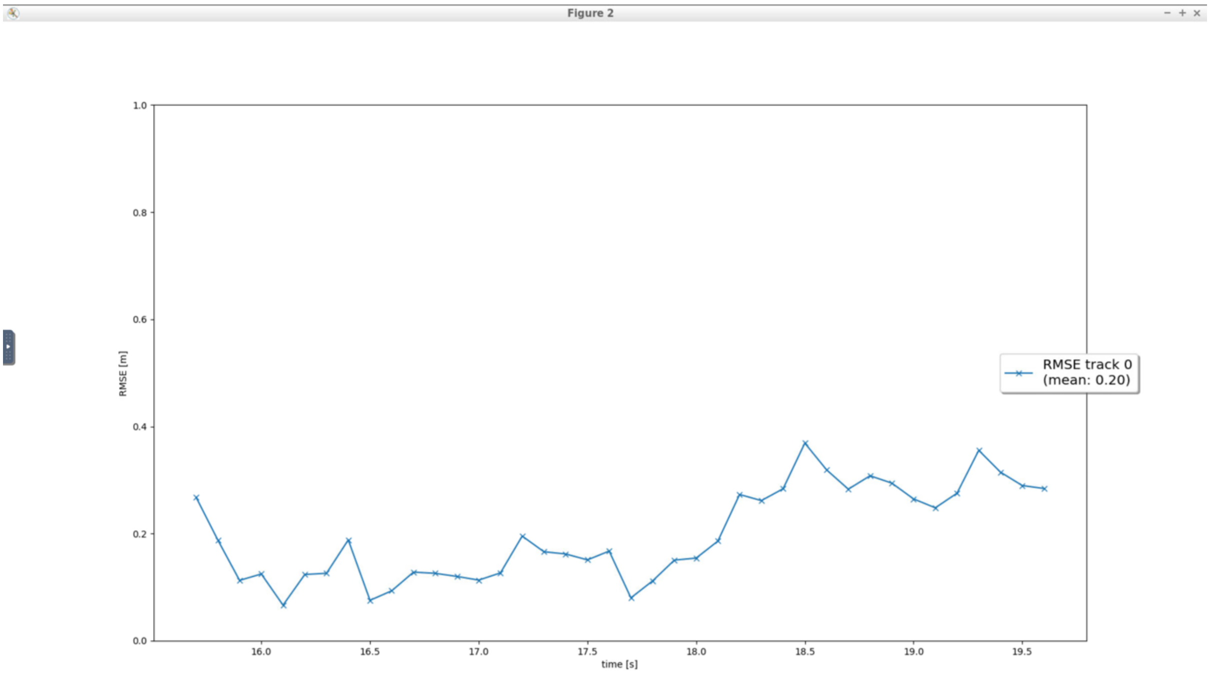
$$q_1 = \frac{q}{3} \times (\Delta t)^3$$
$$q2 = \frac{q}{2} \times (\Delta t)^2$$
$$q_3 = q \times \Delta t$$

Then Predict and update functions were defined as below codes:

```
def predict(self, track):
    ############
    F = self.F()
    x = F.dot(track.x)  # state prediction
    P = F * track.P * F.transpose() + self.Q()  # covariance prediction
    track.set_x(x)
```

```
    track.set_P(P)

    ############
    # END student code
    ############


def update(self, track, meas):
    ############
    H = meas.sensor.get_H(track.x)  # measurement matrix
    gamma = self.gamma(track, meas)  # residual
    S = self.S(track, meas, H)  # covariance of residual
    K = track.P * H.transpose() * np.linalg.inv(S)  # Kalman gain
    x = track.x + K * gamma  # state update
    I = np.identity(self.n)
    P = (I - K * H) * track.P  # covariance update
    track.set_x(x)
    track.set_P(P)
    ############
    # END student code
    ############
    track.update_attributes(meas)


def gamma(self, track, meas):
    ############
    Gamma = meas.z - meas.sensor.get_hx(track.x)  # residual
    return Gamma
    ############

    ############
    # END student code
    ############


def S(self, track, meas, H):
    ############
    cov_residual = H * track.P * H.transpose() + meas.R  # covariance of
residual
    ############

    return cov_residual
```

The results obtained are as below:



In the rest of the project, I implemented the track management to initialize and delete tracks, set a track state and a track score. A multi-target tracking system has to fulfil the following tasks in addition to a single-target tracking (part 1 of the project): i) Data Association which includes Associate measurements to tracks and ii) Track Management which includes Initialize new tracks, Delete old tracks and Assign some confidence value to a track.

An unassigned lidar measurement $z = (z_1, z_2, z_3)^T$ first has to be converted from sensor to vehicle coordinates and then the state of a new track will be initialized. The 3x3 Matrix for the position estimation error covariance P_{pos}can be initialized from the measurement covariance R by rotating from sensor to vehicle coordinates as below:

$$P_{pos} = M_{rot} \cdot R \cdot M_{rot}^T P_{pos} = M_{rot} \cdot R \cdot M_{rot}^T$$

The 3x3 Matrix for the velocity estimation error covariance P_{vel} can be initialized with a diagonal matrix containing large diagonal values, since we cannot measure velocity and therefore have a huge initial velocity uncertainty.
The overall estimation error covariance can then be initialized as:

$$P_0 = \begin{pmatrix} P_{pos} & 0 \\ 0 & P_{vel} \end{pmatrix}$$

A simple track score can be defined as:

$$score = \frac{number\ of\ detections\ in\ last\ n\ frames}{n}$$

The data association assigns measurements to tracks and decides which track to update with which measurement. As a distance measure for this decision, the Mahalanobis distance is used:

$$d(x,z) = \gamma^T S^{-1} \gamma = \big(z - h(x)\big) S^{-1} \big(z - h(x)\big)$$

Then the association matrix A that contains the Mahalanobis distances between each track and each measurement is constructed. Finally, Gating technique is used to reduce the association complexity by removing unlikely association pairs.

The function written for track management is:

```
class Track:
    '''Track class with state, covariance, id, score'''

    def __init__(self, meas, id):
        print('creating track no.', id)
        M_rot = meas.sensor.sens_to_veh[0:3, 0:3]  # rotation matrix from
sensor to vehicle coordinates

        ############
        pos_sens = np.ones((4, 1))  # homogeneous coordinates
        pos_sens[0:3] = meas.z[0:3]
        pos_veh = meas.sensor.sens_to_veh * pos_sens
        self.x = np.ones((6, 1))
        self.x[0:3] = pos_veh[0:3]
        print('x', self.x)

        P_pos = M_rot * meas.R * np.transpose(M_rot)

        P_vel = np.matrix([[params.sigma_p44 ** 2, 0, 0],
                           [0, params.sigma_p55 ** 2, 0],
                           [0, 0, params.sigma_p66 ** 2]])

        # overall covariance initialization
        self.P = np.zeros((6, 6))
        self.P[0:3, 0:3] = P_pos
        self.P[3:6, 3:6] = P_vel

        self.state = 'initialized'
        self.score = 1. / params.window

        ############
        # END student code
        ############
```

```
        # other track attributes
        self.id = id
        self.width = meas.width
        self.length = meas.length
        self.height = meas.height
        self.yaw = np.arccos(M_rot[0, 0] * np.cos(meas.yaw) + M_rot[0, 1] *
np.sin(
            meas.yaw))  # transform rotation from sensor to vehicle
coordinates
        self.t = meas.t

    def set_x(self, x):
        self.x = x

    def set_P(self, P):
        self.P = P

    def set_t(self, t):
        self.t = t

    def update_attributes(self, meas):
        # use exponential sliding average to estimate dimensions and
orientation
        if meas.sensor.name == 'lidar':
            c = params.weight_dim
            self.width = c * meas.width + (1 - c) * self.width
            self.length = c * meas.length + (1 - c) * self.length
            self.height = c * meas.height + (1 - c) * self.height
            M_rot = meas.sensor.sens_to_veh
            self.yaw = np.arccos(M_rot[0, 0] * np.cos(meas.yaw) + M_rot[0,
1] * np.sin(
                meas.yaw))  # transform rotation from sensor to vehicle
coordinates


###################

class Trackmanagement:
    '''Track manager with logic for initializing and deleting objects'''

    def __init__(self):
        self.N = 0  # current number of tracks
        self.track_list = []
        self.last_id = -1
        self.result_list = []

    def manage_tracks(self, unassigned_tracks, unassigned_meas, meas_list):
        ############

        # decrease score for unassigned tracks
        for i in unassigned_tracks:
            track = self.track_list[i]
            # check visibility
            if meas_list:  # if not empty
                if meas_list[0].sensor.in_fov(track.x):
                    track.state = 'tentative'
                    if track.score > params.delete_threshold + 1:
                        track.score = params.delete_threshold + 1
                    track.score -= 1. / params.window

        for track in self.track_list:  # delete old track
```

```
            if track.score <= params.delete_threshold:
                if track.P[0, 0] >= params.max_P or track.P[1, 1] >=
params.max_P:
                    self.delete_track(track)

        # END student code
        ############

        # initialize new track with unassigned measurement
        for j in unassigned_meas:
            if meas_list[j].sensor.name == 'lidar':  # only initialize with
lidar measurements
                self.init_track(meas_list[j])

    def addTrackToList(self, track):
        self.track_list.append(track)
        self.N += 1
        self.last_id = track.id

    def init_track(self, meas):
        track = Track(meas, self.last_id + 1)
        self.addTrackToList(track)

    def delete_track(self, track):
        print('deleting track no.', track.id)
        self.track_list.remove(track)

    def handle_updated_track(self, track):
        ############
        track.score += 1. / params.window
        if track.score > params.confirmed_threshold:
            track.state = 'confirmed'
        else:
            track.state = 'tentative'  ############

        # END student code
        ############
```

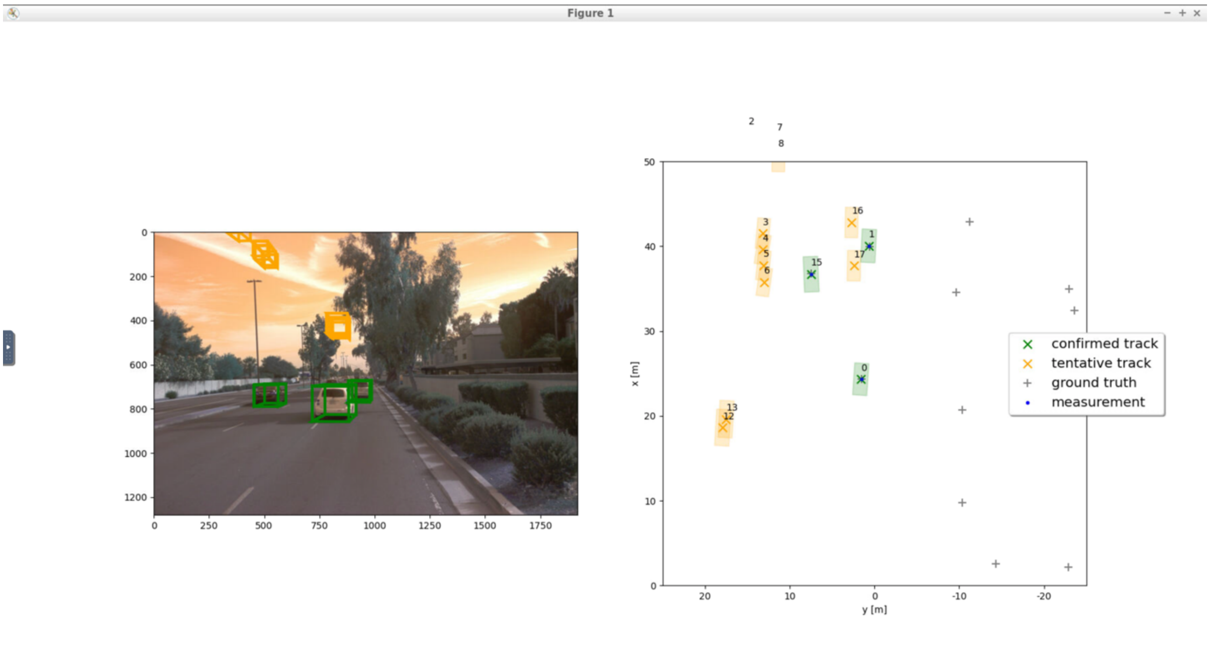
The code written for the association and gating is here:

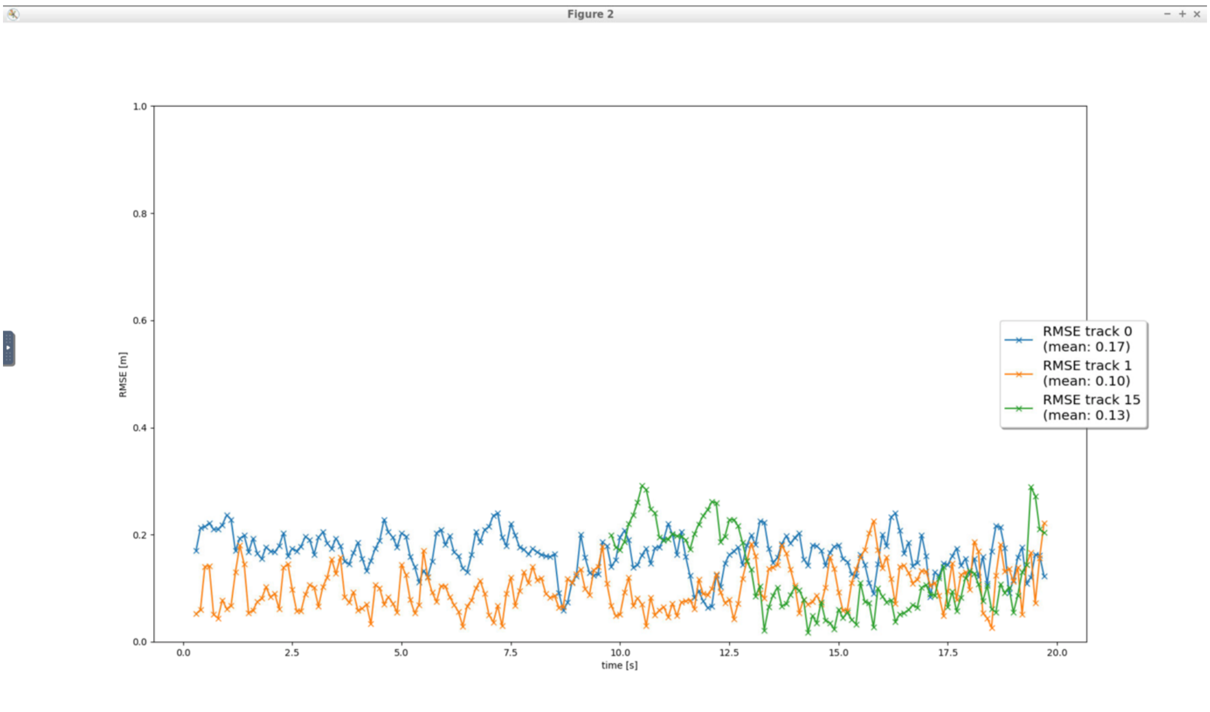
```
class Association:
    '''Data association class with single nearest neighbor association and
gating based on Mahalanobis distance'''

    def __init__(self):
        self.association_matrix = np.matrix([])
        self.unassigned_tracks = []
        self.unassigned_meas = []

    def associate(self, track_list, meas_list, KF):

        # the following only works for at most one track and one
measurement
        No_tracks = len(track_list)
        No_meas = len(meas_list)
        # initialize unassigned tracks and unassigned measurements lists to
be updated in the "associate_and_update" method
        self.unassigned_tracks = list(range(No_tracks))
        self.unassigned_meas = list(range(No_meas))
```

```python
        # initialize association matrix
        self.association_matrix = np.inf * np.ones((No_tracks, No_meas))
        for i in range(No_tracks):
            track = track_list[i]
            for j in range(No_meas):
                meas = meas_list[j]
                dist = self.MHD(track, meas, KF)
                if self.gating(dist, meas.sensor):
                    self.association_matrix[i][j] = dist
        return
        # END student code
        ############

    def get_closest_track_and_meas(self):
        A = self.association_matrix
        if np.min(A) == np.inf:
            return np.nan, np.nan

        # get indices of minimum entry
        ij_min = np.unravel_index(np.argmin(A, axis=None), A.shape)
        ind_track = ij_min[0]
        ind_meas = ij_min[1]

        # delete row and column for next update
        A = np.delete(A, ind_track, 0)
        A = np.delete(A, ind_meas, 1)
        self.association_matrix = A

        # update this track with this measurement
        update_track = self.unassigned_tracks[ind_track]
        update_meas = self.unassigned_meas[ind_meas]

        # remove this track and measurement from list
        self.unassigned_tracks.remove(update_track)
        self.unassigned_meas.remove(update_meas)

        # END student code
        ############
        return update_track, update_meas

    def gating(self, MHD, sensor):
        ############
        if sensor.name == "lidar":
            DOF = 2   # LiDAR DOF, features = (x,y,z)
        elif sensor.name == "camera":
            DOF = 1   # camera DOF,features = (x,y)
        # calculate the Inverse Cumulative Distribution Function (I-CDF)
for certain percentile (gating_threshold)
        limit = chi2.ppf(params.gating_threshold, df=sensor.dim_meas)
        if MHD < limit:
            return True
        else:
            return False

            # END student code
        ############

    def MHD(self, track, meas, KF):
        ############
        H = meas.sensor.get_H(track.x)
        gamma = meas.z - meas.sensor.get_hx(track.x)
```

```
        S = H * track.P * H.transpose() + meas.R
        MHD = gamma.transpose() * np.linalg.inv(S) * gamma  # Mahalanobis
distance formula
        return MHD

        ############
        # END student code
        ############
```

Here are the results for multi-object tracking:





2. Do you see any benefits in camera-lidar fusion compared to lidar-only tracking (in theory and in your concrete results)?

It is hard to justify the benefit of camera-lidar fusion to lidar-only tracking. In theory, fusion of sensors will improve the performance. In my understanding combining both sensors will not help to improve the accuracy of tracking however, it will support the safety enhancement of AV driving. In case one sensor ca not provide reliable data due to light conditions or weather conditions, the other sensor can support the functionality.

3. Which challenges will a sensor fusion system face in real-life scenarios? Did you see any of these challenges in the project?

The complexity of the algorithm, computational load and synchronisation are the main challenges that I can think of.

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

Better tuning of parameters and using more accurate detection algorithms are possible ways.