CS231A: Computer Vision, From 3D Reconstruction to Recognition Homework #4 Solution

(Winter 2021)

Shubham Shrivastava email: shubhams@stanford.edu

1 Extended Kalman Filter with a Nonlinear Observation Model (60 points)

Consider the scenario depicted in Figure 1 where a robot tries to catch a fly that it tracks visually with its cameras. To catch the fly, the robot needs to estimate the 3D position $\mathbf{p}_t \in \mathbb{R}^3$ and linear

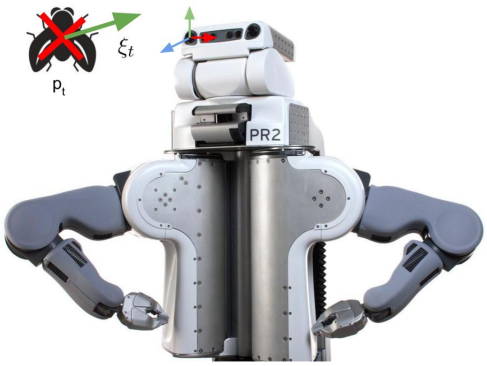

Figure 1

velocity $\xi_t \in \mathbb{R}^3$ of the fly with respect to its camera coordinate system. The fly is moving randomly in a way that can be modelled by a discrete time double integrator:

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \Delta t \xi_t \tag{1a}$$
$$\xi_{t+1} = 0.8\xi_t + \Delta t \mathbf{a}_t \tag{1b}$$

where the constant velocity value describes the average velocity value over Δt and is just an approximation of the true process. Variations in the fly's linear velocity are caused by random, immeasurable accelerations $\mathbf{a}_t$. As the accelerations are not measurable, we treat it as the process noise, $\mathbf{w} = \Delta t \mathbf{a}_t$, and we model it as a realization of a normally-distributed white-noise random vector with zero mean and covariance Q: $\mathbf{w} \sim N(0, Q)$. The covariance is given by $Q = \text{diag}(0.05, 0.05, 0.05)$

The vision system of the robot consists of (unfortunately) only one camera. With the camera, the robot can observe the fly and receive noisy measurements $\mathbf{z} \in \mathbb{R}^2$ which are the pixel coordinates (u, v) of the projection of the fly onto the image. We model this projection mapping of the fly's 3D location to pixels as the observation model h:

$$\mathbf{z}_t = h(\mathbf{x}_t) + \mathbf{v}_t \tag{1c}$$

where $\mathbf{x} = (\mathbf{p}, \xi)^T$ and $\mathbf{v}$ is a realization of the normally-distributed, white-noise observation noise vector: $\mathbf{v} \sim N(0, R)$. The covariance of the measurement noise is assumed constant and of value, $R = \text{diag}(5, 5)$.

We assume a known 3x3 camera intrinsic matrix:

$$K = \begin{bmatrix} 500 & 0 & 320 & 0 \\ 0 & 500 & 240 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1d}$$

a. (8 pts) Let $\Delta t = 0.1s$. Find the system matrix A for the process model, and implement the noise covariance functions (Implement your answer in the *system_matrix*, *process_noise_covariance*, and *observation_noise_covariance* functions in Q1.py).

Movement of the fly can be modeled as:

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \Delta t \xi_t$$
$$\xi_{t+1} = 0.8\xi_t + \Delta t \mathbf{a}_t$$

$\mathbf{a}_t$ is not observable

$\mathbf{w}_t$

State-Space model:

Noise: $w \sim N(0, Q)$

Q: Diag(0.05, 0.05, 0.05)

$$x_{t+1} = A_t x_t + B_t u_t + w_t \longrightarrow \text{Linear trasition model}$$

$$z_{t+1} = C_t x_{t+1} + \delta_t \longrightarrow \text{Linear observation model}$$

where the model state, $x_t = [p_t^x, p_t^y, p_t^z, \xi_t^x, \xi_t^y, \xi_t^z]^T$

p_t (position) $\quad$ ξ_t (linear velocity)

$$\Rightarrow A_t = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0.8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.8 \end{bmatrix}$$

Implementations of system matrix A, process model, and noise covariance functions are given in Listing 1.

```python
@staticmethod
def system_matrix():
    """
    Output:
      A: 6x6 numpy array for the system matrix.
    """
    dt = 0.1
    A = np.array([[1., 0., 0., dt, 0., 0.],
                  [0., 1., 0., 0., dt, 0.],
                  [0., 0., 1., 0., 0., dt],
                  [0., 0., 0., 0.8, 0., 0.],
                  [0., 0., 0., 0., 0.8, 0.],
                  [0., 0., 0., 0., 0., 0.8]], dtype=np.float64)
    return A

    @staticmethod
    def process_noise_covariance():
    """ Covariance matrix Q for process noise.
    Output:
      Q: 6x6 numpy array for the covariance matrix.
    """
    Q = np.zeros((6,6), dtype=np.float64)
    Q[3:,3:] = np.diag([0.05, 0.05, 0.05])
    return Q

    @staticmethod
    def observation_noise_covariance():
    """ Covariance matrix R for observation noise.
    Output:
      R: 2x2 numpy array for the covariance matrix.
    """
    R = np.diag([5.0, 5.0])
    return R
```

Listing 1: Code Snippet

b. (8 pts) Define the observation model h in terms of the camera parameters (Implement your answer in the *observation* function in Q1.py).

Implementation of the observation model h in terms of the camera parameters is given in Listing 2.

```python
@staticmethod
def observation(state):
    """ Function h, from state to noise-less observation. (Q1B)
    Input:
      state: (6,) numpy array representing state.
    Output:
      obs: (2,) numpy array representing observation.
    """
    # camera intrinsics
    K = np.array([[500., 0.,   320.],
                  [0.,   500., 240.],
                  [0.,   0.,   1.]], dtype=np.float64)

    obs = np.dot(K, state[:3]) # get 2d projection
    obs = (obs / obs[-1])[:2] # homogeneous to euclidean conversion

```

```
    return obs
```

Listing 2: Code Snippet

c. (8 pts) Initially, the fly is sitting on the fingertip of the robot when it is noticing it for the first time. Therefore, the robot knows the fly's initial position from forward kinematics to be at $\mathbf{p}_0 = (0.5, 0, 5.0)^T$ (resting velocity). Simulate in Python the 3D trajectory that the fly takes as well as the measurement process. This requires generating random acceleration noise and observation noise. Simulate for 100 time steps. Attach a plot of the generated trajectories and the corresponding measurements. Please add your code to the report.

The simulated trajectory and corresponding 2D measurements are shown in Figure 2.

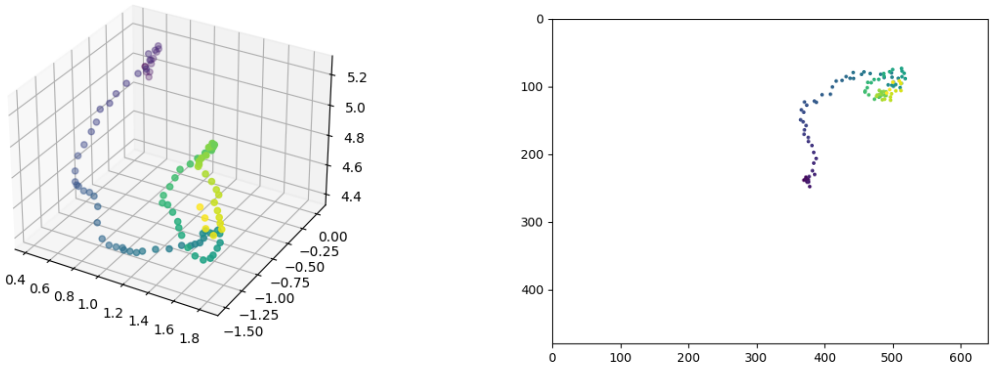

Figure 2: Left: Simulated trajectory, Right: 2D measurements

Code implementation is given in Listing 3.

```
def simulation(self, T=100):
    """ simulate with fixed start state for T timesteps.
    Input:
      T: an integer (=100).
    Output:
      states: (T,6) numpy array of states, including the given start state.
      observations: (T,2) numpy array of observations, Including the
    observation of start state.
    Note:
      We have set the random seed for you. Please only use np.random.
    multivariate_normal to sample noise.
      Keep in mind this function will be reused for Q2 by inheritance.
    """
    x_0 = np.array([0.5, 0.0, 5.0, 0.0, 0.0, 0.0])
    states = [x_0]
    A = self.system_matrix()
    Q = self.process_noise_covariance()
    R = self.observation_noise_covariance()
    z_0 = self.observation(x_0) + np.random.multivariate_normal(np.zeros((R.
    shape[0],)), R)
    observations = [z_0]
    for t in range(1,T):
        process_noise = np.random.multivariate_normal(np.zeros((Q.shape[0],)),
     Q)
        xt = np.dot(A, x_0) + process_noise
```

```
        zt = self.observation(xt) + np.random.multivariate_normal(np.zeros((R.
    shape[0],)), R)
        states.append(xt)
        observations.append(zt)
        x_0 = xt
    return np.array(states), np.array(observations)
```

Listing 3: Code Snippet

d. (8 pts) Find the Jacobian H of the observation model with respect to the fly's state $\mathbf{x}$. (Implement your answer of H in function *observation_state_jacobian* in Q1.py.)

$$K = \begin{bmatrix} 500 & 0 & 320 & 0 \\ 0 & 500 & 240 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(f_x, f_y, o_x, o_y)

Observation model h is defined such that,

$z_t = h(x_t) + v_t$

where, $x_t = (p, \varepsilon)^T$, v_t = normally-distributed white noise observation noise vector: $v \sim N(0, R)$

$R = \text{diag}(s, s)$

$$z(t) = h(x_t) + v_t$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = h\left(\begin{bmatrix} p_t^x \\ p_t^y \\ p_t^z \\ \varepsilon_t^x \\ \varepsilon_t^y \\ \varepsilon_t^z \end{bmatrix} \right) + v_t$$

(v_t → measurement noise)

$$u = f_x \frac{p_t^x}{p_t^z} + o_x \quad \text{— (i)} \qquad v = f_y \frac{p_t^y}{p_t^z} + o_y \quad \text{— (ii)}$$

Jacobian, $J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$; here $f \in [u, v]$

$x \in [p^x, p^y, p^z, \varepsilon^x, \varepsilon^y, \varepsilon^z]$

So,

$$J = \begin{bmatrix} \frac{\partial u_t}{\partial p_t^x} & \frac{\partial u_t}{\partial p_t^y} & \frac{\partial u_t}{\partial p_t^z} & \frac{\partial u_t}{\partial \varepsilon_t^x} & \frac{\partial u_t}{\partial \varepsilon_t^y} & \frac{\partial u_t}{\partial \varepsilon_t^z} \\ \frac{\partial v_t}{\partial p_t^x} & \frac{\partial v_t}{\partial p_t^y} & \frac{\partial v_t}{\partial p_t^z} & \frac{\partial v_t}{\partial \varepsilon_t^x} & \frac{\partial v_t}{\partial \varepsilon_t^y} & \frac{\partial v_t}{\partial \varepsilon_t^z} \end{bmatrix}$$

→ 0 (u & v are not a function of ε)

Let's compute other terms:

$$\frac{\partial u_t}{\partial p_t^x} = \frac{\partial}{\partial p_t^x}\left[f_x \frac{p_t^x}{p_t^z} + 0_x\right] = \frac{f_x}{p_t^z} \quad \text{— (iii)}$$

$$\frac{\partial u_t}{\partial p_t^y} = \frac{\partial}{\partial p_t^y}\left[f_x \frac{p_t^x}{p_t^z} + 0_x\right] = 0 \quad \text{— (iv)}$$

$$\frac{\partial u_t}{\partial p_t^z} = \frac{\partial}{\partial p_t^z}\left[f_x \frac{p_t^x}{p_t^z} + 0_x\right] = -f_x \frac{p_t^x}{(p_t^z)^2} \quad \text{— (v)}$$

Code implementation is given in Listing 4.

```
@staticmethod
def observation_state_jacobian(x):
    """
```

```
    Input:
      x: (6,) numpy array, the state we want to do jacobian at.
    Output:
      J: (2,6) numpy array, the jacobian of the observation model w.r.t state.
    """
    # camera intrinsics
    K = np.array([[500., 0.,   320.],
                  [0.,   500., 240.],
                  [0.,   0.,   1.  ]], dtype=np.float64)
    J = np.array([[K[0,0]/x[2],  0., -K[0,0]*x[0]/(x[2]**2), 0., 0., 0.],
                  [0.,  K[1,1]/x[2], -K[1,1]*x[1]/(x[2]**2), 0., 0., 0.]],
    dtype=np.float64)

    return J
```

Listing 4: Code Snippet

e. (20 pts) Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the camera. You can assume the aforementioned initial position and the following initial error covariance matrix: $P_0 = \text{diag}(0.1, 0.1, 0.1)$. The measurements can be found in `data/Q1E_measurement.npy`. Plot the mean and error ellipse of the predicted measurements over the true measurements. Plot the means and error ellipsoids of the estimated positions over the true trajectory of the fly. The true states are in `data/Q1E_state.npy`

These plots are given in Figure 3 and the corresponding code implementation for EKF is given in Listing 5

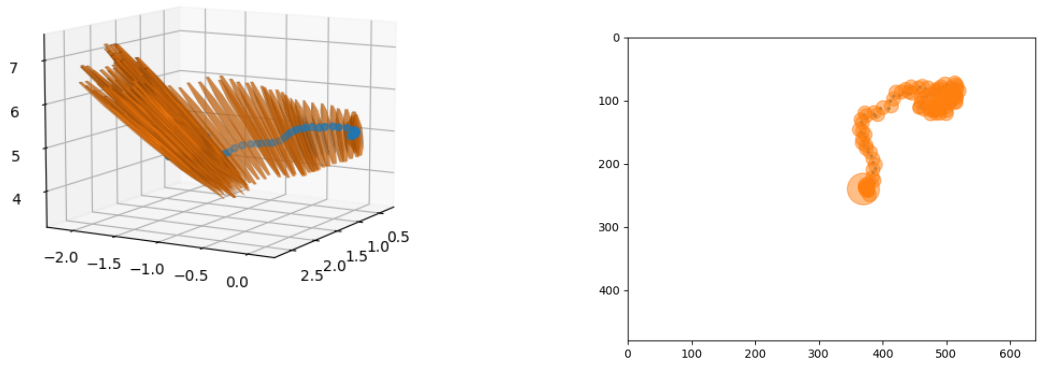

Figure 3: Left: Means and error ellipsoids of the estimated positions over the true trajectory of the fly, Right: Mean and error ellipse of the predicted measurements over the true measurements

```
def EKF(self, observations):
    """ Extended Kalman filtering
    Input:
      observations: (N,2) numpy array, the sequence of observations. From T=1.
      mu_0: (6,) numpy array, the mean of state belief after T=0
      sigma_0: (6,6) numpy array, the covariance matrix for state belief after
     T=0.
    Output:
      state_mean: (N,6) numpy array, the filtered mean state at each time step
    . Not including the
```

```
                starting state mu_0.
    state_sigma: (N,6,6) numpy array, the filtered state covariance at each time step. Not including
                the starting state covarance matrix sigma_0.
    predicted_observation_mean: (N,2) numpy array, the mean of predicted observations. Start from T=1
    predicted_observation_sigma: (N,2,2) numpy array, the covariance matrix of predicted observations. Start from T=1
    Note:
    Keep in mind this function will be reused for Q2 by inheritance.
    """
    mu_0 = np.array([0.5, 0.0, 5.0, 0.0, 0.0, 0.0])
    sigma_0 = np.eye(6)*0.01
    sigma_0[3:,3:] = 0.0
    A = self.system_matrix()
    Q = self.process_noise_covariance()
    R = self.observation_noise_covariance()

    # define initial mean and covariances
    state_mean = [mu_0]
    state_sigma = [sigma_0]
    predicted_observation_mean = []
    predicted_observation_sigma = []

    # iterate through each observation
    for ob in observations:
        # prediction step
        mu_bar_next = np.dot(A, state_mean[-1])
        sigma_bar_next = np.dot(A, np.dot(state_sigma[-1], A.T)) + Q
        # update step
        H = self.observation_state_jacobian(mu_bar_next) # compute jacobian
        kalman_gain_numerator = np.dot(sigma_bar_next, H.T)
        kalman_gain_denominator = np.dot(H, np.dot(sigma_bar_next, H.T)) + R
        kalman_gain = np.dot(kalman_gain_numerator, np.linalg.inv(kalman_gain_denominator)) # compute kalman gain
        expected_observation = self.observation(mu_bar_next)
        mu_next = mu_bar_next + np.dot(kalman_gain, (ob - expected_observation).T)
        sigma_next = np.dot((np.eye(6, dtype=np.float64) - np.dot(kalman_gain, H)), sigma_bar_next)
        # append new estimated mean and covariances to list
        state_mean.append(mu_next)
        state_sigma.append(sigma_next)
        predicted_observation_mean.append(expected_observation)
        predicted_observation_sigma.append(kalman_gain_denominator)

    return np.array(state_mean[1:]), np.array(state_sigma[1:]), np.array(predicted_observation_mean), np.array(predicted_observation_sigma)
```

Listing 5: Code Snippet

f. (8 pts) Discuss the difference in magnitude of uncertainty in the different dimensions of the state.

As it can be seen from Figure 3, the uncertainty in x and y dimensions are much smaller than the uncertainty in z dimension. Our measurement model loses information about the z dimension during 3D to 2D transformation, this introduces ambiguities in the determination of fly's position in the z dimension - and hence the uncertainty increases as time progresses.

2 Extra Credit - From Monocular to Stereo Vision (30 points)

Now let us assume that our robot got an upgrade: Someone installed a stereo camera and calibrated it. Let us assume that this stereo camera is perfectly manufactured, i.e., the two cameras are perfectly parallel with a baseline of $b = 0.2$. The camera intrinsics are the same as before in Question 1.

Now the robot receives as measurement $\mathbf{z}$ a pair of pixel coordinates in the left image (u^L, v^L) and right image (u^R, v^R) of the camera. Since our camera system is perfectly parallel, we will assume a measurement vector $\mathbf{z} = (u^L, v^L, d^L)$ where d^L is the disparity between the projection of the fly on the left and right image. We define the disparity to be positive. The fly's states are represented in the left camera's coordinate system.

a. (6 pts) Find the observation model h in terms of the camera parameters (Implement your answer in function *observation* in Q2.py).

Implementation of the observation model h in terms of the camera parameters is given in Listing 6.

```
@staticmethod
def observation(x):
    """ Observation function without noise.
    Input:
      x: (6,) numpy array representing the state.
    Output:
      obs: (3,) numpy array representing the observation (u,v,d).
    Note:
      we define disparity to be possitive.
    """
    # camera intrinsics
    K = np.array([[500., 0.,   320.],
                  [0.,   500., 240.],
                  [0.,   0.,   1.]], dtype=np.float64)

    # stereo camera baseline
    baseline = 0.2

    obs_uv = np.dot(K, x[:3]) # get 2d projection
    obs_uv = (obs_uv / obs_uv[-1])[:2] # homogeneous to euclidean conversion
    disparity = K[0,0] * baseline / x[2]
    obs = np.array([obs_uv[0], obs_uv[1], disparity], dtype=np.float64)

    return obs
```

Listing 6: Code Snippet

b. (6 pts) Find the Jacobian H of the observation model with respect to the fly's state x. (Implement H in function *observation_state_jacobian* in Q2.py)

$$K = \begin{bmatrix} 500 & 0 & 320 & 0 \\ 0 & 500 & 240 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(f_x, f_y, O_x, O_y)

Observation model h is defined such that,

$$z_t = h(x_t) + v_t$$

where, $x_t = (p, \varepsilon)^T$, v_t = normally-distributed white noise observation noise vector: $v \sim N(0, R)$

$R = \text{diag}(5, 5)$

$$z(t) = h(x_t) + v_t$$

$$\begin{bmatrix} u_t \\ v_t \\ d_t \end{bmatrix} = h\left(\begin{bmatrix} p_t^x \\ p_t^y \\ p_t^z \\ \varepsilon_t^x \\ \varepsilon_t^y \\ \varepsilon_t^z \end{bmatrix} \right) + v_t$$

v_t → measurement noise

$$u = f_x \frac{p_t^x}{p_t^z} + O_x \quad \text{— (i)} \qquad v = f_y \frac{p_t^y}{p_t^z} + O_y \quad \text{— (ii)}$$

baseline → b

$$d = f_x \cdot b \,/\, p_t^z \quad \text{— (iii)}$$

Jacobian, $J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$; here $f \in [u, v, d]$

$x \in [p^x, p^y, p^z, \varepsilon^x, \varepsilon^y, \varepsilon^z]$

So,

$$J = \begin{bmatrix} \frac{\partial u_t}{\partial p_t^x} & \frac{\partial u_t}{\partial p_t^y} \to 0 & \frac{\partial u_t}{\partial p_t^z} & \frac{\partial u_t}{\partial \varepsilon_t^x} & \frac{\partial u_t}{\partial \varepsilon_t^y} & \frac{\partial u_t}{\partial \varepsilon_t^z} \\ \frac{\partial v_t}{\partial p_t^x} \to 0 & \frac{\partial v_t}{\partial p_t^y} & \frac{\partial v_t}{\partial p_t^z} & \frac{\partial v_t}{\partial \varepsilon_t^x} & \frac{\partial v_t}{\partial \varepsilon_t^y} & \frac{\partial v_t}{\partial \varepsilon_t^z} \\ \frac{\partial d_t}{\partial p_t^x} \to 0 & \frac{\partial d_t}{\partial p_t^y} \to 0 & \frac{\partial d_t}{\partial p_t^z} & \frac{\partial d_t}{\partial \varepsilon_t^x} & \frac{\partial d_t}{\partial \varepsilon_t^y} & \frac{\partial d_t}{\partial \varepsilon_t^z} \end{bmatrix}$$

→ 0 (u, v, & d are not a function of ε)

Let's compute other terms:

$$\frac{\partial u_t}{\partial p_t^x} = \frac{\partial}{\partial p_t^x}\left[f_x \frac{p_t^x}{p_t^z} + 0_x\right] = \frac{f_x}{p_t^z} \quad \text{— (iv)}$$

$$\frac{\partial u_t}{\partial p_t^z} = \frac{\partial}{\partial p_t^z}\left[f_x \frac{p_t^x}{p_t^z} + 0_x\right] = -f_x \frac{p_t^x}{(p_t^z)^2} \quad \text{— (v)}$$

$$\frac{\partial v_t}{\partial p_t^y} = \frac{\partial}{\partial p_t^y}\left[f_y \frac{p_t^y}{p_t^z} + o_y\right] = \frac{f_y}{p_t^z} \quad \text{(vi)}$$

$$\frac{\partial v_t}{\partial p_t^z} = \frac{\partial}{\partial p_t^z}\left[f_y \frac{p_t^y}{p_t^z} + o_y\right] = -f_y \frac{p_t^y}{(p_t^z)^2} \quad \text{(vii)}$$

$$\frac{\partial d_t}{\partial p_t^z} = \frac{\partial}{\partial p_t^z}\left[f_x \frac{b}{p_t^z}\right] = -f_x \frac{b}{(p_t^z)^2} \quad \text{(viii)}$$

$$\Rightarrow J = \begin{bmatrix} \frac{f_x}{p_t^z} & 0 & -f_x \frac{p_t^x}{(p_t^z)^2} & 0 & 0 & 0 \\ 0 & \frac{f_y}{p_t^z} & -f_y \frac{p_t^y}{(p_t^z)^2} & 0 & 0 & 0 \\ 0 & 0 & -f_x \frac{b}{(p_t^z)^2} & 0 & 0 & 0 \end{bmatrix}$$

Implementation of the Jacobian Matrix is given in Listing 7.

```
@staticmethod
def observation_state_jacobian(x):
    """ The jacobian of observation function w.r.t state.
    Input:
      x: (6,) numpy array, the state to take jacobian with.
    Output:
      J: (3,6) numpy array, the jacobian H.
    """
    # camera intrinsics
    K = np.array([[500., 0.,   320.],
                  [0.,   500., 240.],
                  [0.,   0.,   1.]], dtype=np.float64)

    # stereo camera baseline
```

```
    baseline = 0.2

    J = np.array([[K[0,0]/x[2],  0., -K[0,0]*x[0]/(x[2]**2), 0., 0., 0.],
                  [0.,  K[1,1]/x[2], -K[1,1]*x[1]/(x[2]**2), 0., 0., 0.],
                  [0.,  0.,      -K[0,0]*baseline/(x[2]**2), 0., 0., 0.]],
    dtype=np.float64)

    return J
```

Listing 7: Code Snippet

c. (6 pts) What is the new observation noise covariance matrix R? Assume the noise on (u^L, v^L), and (u^R, v^R) to be independent and to have the same distribution as the observation noise given in Question 1, respectively. (Implement R in function *observation_noise_covariance* in Q2.py).

Implementation of the the new observation noise covariance matrix R is given in Listing 8.

```
@staticmethod
def observation_noise_covariance():
    """
    Output:
      R: (3,3) numpy array, the covariance matrix for observation noise.
    """
    sigma2_u = 5.0
    sigma2_v = 5.0
    R = np.array([[sigma2_u, 0.0, np.sqrt(sigma2_u)*np.sqrt(sigma2_v)],
                  [0.0, sigma2_v, 0.0],
                  [np.sqrt(sigma2_u)*np.sqrt(sigma2_v), 0.0, sigma2_u+sigma2_v
    ]])

    return R
```

Listing 8: Code Snippet

d. (6 pts) Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the left camera. You can assume the same initial position and the initial error covariance matrix as in the previous questions. Plot the means and error ellipses of the predicted measurements over the true measurement trajectory in both the left and right images. The measurements can be found in `data/Q2D_measurement.npy`. Plot the means and error ellipsoids of the estimated positions over the true trajectory of the fly. The true states are in `data/Q2D_state.npy` Include these plots here.

These plots are shown in Figures 4, 5, and 6.

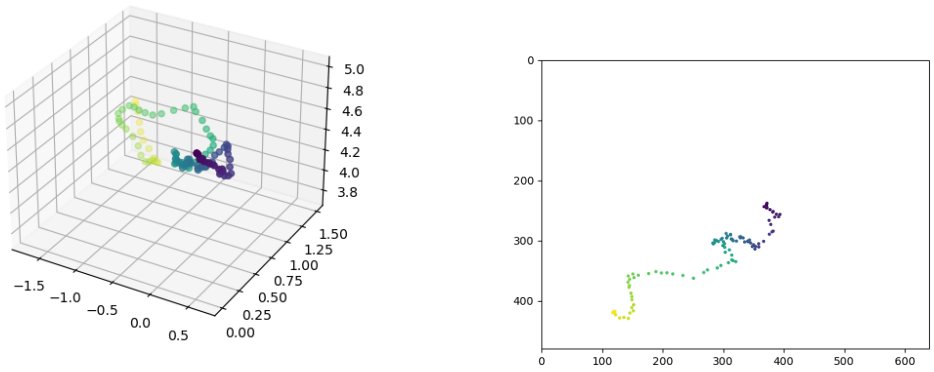

Figure 4: Left: Simulated true trajectory of the fly, Right: True measurements of the trajectory

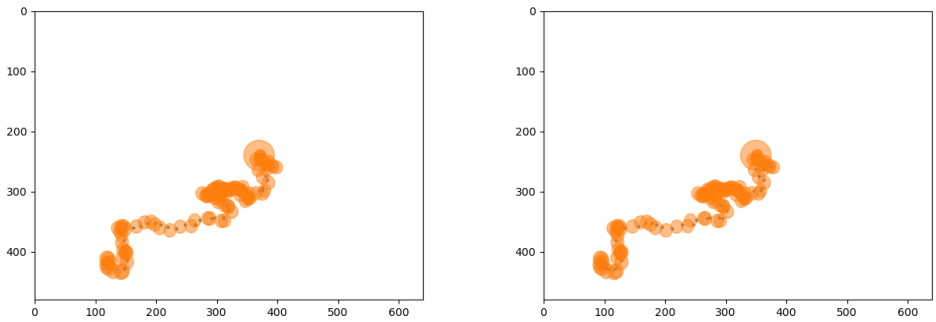

Figure 5: Plot the means and error ellipses of the predicted measurements over the true measurement trajectory in the left camera image (left) and right camera image (right)

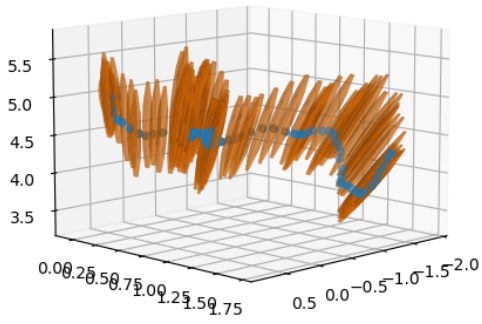

Figure 6: Means and error ellipsoids of the estimated positions over the true trajectory of the fly

e. (6 pts) In this Question, we are defining $\mathbf{z} = (u^L, v^L, d^L)^T$. Alternatively, we could reconstruct the 3D position $\mathbf{p}$ of the fly from its left and right projection (u^L, v^L, u^R, v^R) through triangulation and use $\mathbf{z} = (x, y, z)^T$ directly. Discuss the pros and cons of using (u^L, v^L, d^L) over (x, y, z)!

One could reconstruct the 3D position $\mathbf{p}$ of the fly from its left and right projection (u^L, v^L, u^R, v^R) through triangulation and use $\mathbf{z} = (x, y, z)^T$ directly which would have certain advantages over using $\mathbf{z} = (u^L, v^L, d^L)^T$. Using $\mathbf{z} = (x, y, z)^T$ directly will reduce the non-linear measurement function into a linear function and hence a simple Kalman Filter can be used to track the fly's 3D position and hence we would not need to compute Jacobian and get a linear estimate of the measurement function.

3 Linear Kalman Filter with a Learned Inverse Observation Model (40 points)

Now the robot is trying to catch a ball. So far, we assumed that there was some vision module that would detect the object in the image and thereby provide a noisy observation. In this part of the assignment, let us learn such a detector from annotated training data and treat the resulting detector as a sensor.

If we assume the same process as in the first task, but we have a measurement model that observes directly noisy 3D locations of the ball, we end up with a linear model whose state can be estimated with a Kalman filter. Note that since you are modifying code from previous parts and are implementing your own outlier detection for part C, there is no autograder for this problem - we will be grading based on your plots.

a. (5 pts) In the folder `data/Q3A_data` you will find a training set of 1000 images in the subfolder `training_set` and the file `Q3A_positions_train.npy` that contains the ground truth 3D position of the red ball in the image. We have provided you with the notebook `LearnedObservationModel.ipynb` that can be used to train a noisy observation model. As in PSET 3, use this notebook with Google Colab to do this – note that you'll need to upload the data directory onto a location of your choosing in Drive first. Report the training and test set mean squared error in your write-up.

The training and test set MSE are given below:

```
    Final training loss:  0.82348794
    Final testing loss:  0.6004227

```

b. (20 pts) In the folder `data/Q3B_data` you will find a set of 1000 images that show a new trajectory of the red ball. Run your linear Kalman Filter using this sequence of images as input, where your learned model provides the noisy measurements (the logic for this is provided in `LearnedObservationModel.ipynb`). Now you can work on using the model by completing `p3.py`. Tune a constant measurement noise covariance appropriately, assuming it is a zero mean Gaussian and the covariance matrix is a diagonal matrix. Plot the resulting estimated trajectory from the images, along with the detections and the ground truth trajectory (the logic for this is provided in the starter code). Please add your code to the report.

The mean error computed from trained model on Q3B data came out to be 0.2303052457992848. These predictions were used as a measurement model for the Kalman Filter. Figure 7 shows the plots for true and estimated trajectory and Listing 9 shows the code implementation.

```
@staticmethod
def system_matrix():
    """
    Output:
      A: 6x6 numpy array for the system matrix.
    """
    dt = 0.1
    A = np.array([[1., 0., 0., dt, 0., 0.],
                  [0., 1., 0., 0., dt, 0.],
                  [0., 0., 1., 0., 0., dt],
                  [0., 0., 0., 0.8, 0., 0.],
                  [0., 0., 0., 0., 0.8, 0.],
                  [0., 0., 0., 0., 0., 0.8]], dtype=np.float64)
```

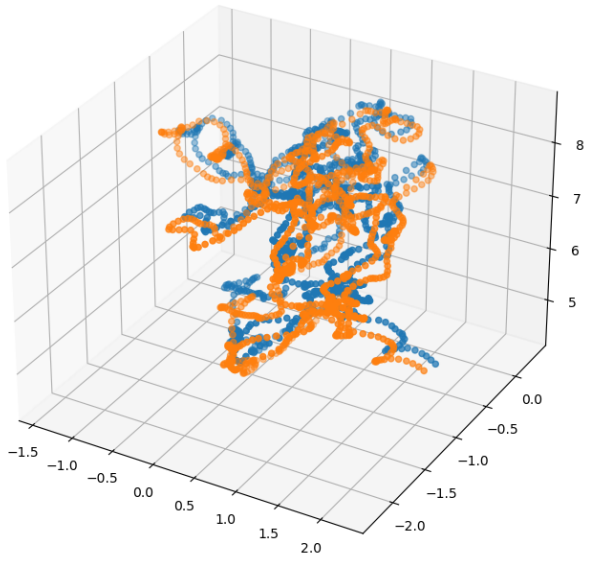

Figure 7: True (blue) and Estimated Trajectory (orange) for *Q3B_data*

```
    return A

@staticmethod
def process_noise_covariance():
    """
    Output:
      Q: 6x6 numpy array for the covariance matrix.
    """
    Q = np.zeros((6,6), dtype=np.float64)
    Q[3:,3:] = np.diag([0.05, 0.05, 0.05])
    return Q

@staticmethod
def observation_noise_covariance():
    """
    Output:
      R: 3x3 numpy array for the covariance matrix.
    """
    sigma = np.diag([0.005, 0.005, 0.01])
    return sigma

def KF(self, observations, mu_0, sigma_0, remove_outliers):
    """ Implement Kalman filtering
    Input:
      observations: (N,3) numpy array, the sequence of observations. From T=1.
      mu_0: (6,) numpy array, the mean of state belief after T=0
      sigma_0: (6,6) numpy array, the covariance matrix for state belief after
     T=0.
      remove_outliers: bool, whether to remove outliers
    Output:
      state_mean: (N,6) numpy array, the filtered mean state at each time step
    . Not including the
                  starting state mu_0.
      state_sigma: (N,6,6) numpy array, the filtered state covariance at each
    time step. Not including
                  the starting state covarance matrix sigma_0.
      predicted_observation_mean: (N,3) numpy array, the mean of predicted
    observations. Start from T=1
```

```
48        predicted_observation_sigma: (N,3,3) numpy array, the covariance matrix
    of predicted observations. Start from T=1
49      """
50      A = self.system_matrix()
51      Q = self.process_noise_covariance()
52      R = self.observation_noise_covariance()
53      state_mean = [mu_0]
54      state_sigma = [sigma_0]
55      predicted_observation_mean = []
56      predicted_observation_sigma = []
57      for ob in observations:
58          mu_bar_next = np.dot(A, state_mean[-1])
59          sigma_bar_next = np.dot(A, np.dot(state_sigma[-1], A.T)) + Q
60          # H is the measurement model which casts next state onto an
    ovservation vector
61          H = np.array([[1., 0., 0., 0., 0., 0.],
62                        [0., 1., 0., 0., 0., 0.],
63                        [0., 0., 1., 0., 0., 0.]], dtype=np.float64)
64          kalman_gain_numerator = np.dot(sigma_bar_next, H.T)
65          kalman_gain_denominator = np.dot(H, np.dot(sigma_bar_next, H.T)) + R #
     this is the innovation covariance matrix, S
66          kalman_gain = np.dot(kalman_gain_numerator, np.linalg.inv(
    kalman_gain_denominator))
67          expected_observation = np.dot(H, mu_bar_next)
68          # let's compute Mahalanobis distance
69          S = kalman_gain_denominator
70          deviation = np.sqrt(np.dot((ob - expected_observation).T, np.dot(np.
    linalg.inv(S), (ob - expected_observation))))
71          if not filter_outliers or deviation <= 10:# part D
72            mu_next = mu_bar_next + np.dot(kalman_gain, (ob -
    expected_observation).T)
73            sigma_next = np.dot((np.eye(6, dtype=np.float64) - np.dot(
    kalman_gain, H)), sigma_bar_next)
74          else:
75              mu_next = mu_bar_next
76              sigma_next = sigma_bar_next
77          state_mean.append(mu_next)
78          state_sigma.append(sigma_next)
79          predicted_observation_mean.append(expected_observation)
80          predicted_observation_sigma.append(kalman_gain_denominator)
81      return state_mean, state_sigma, predicted_observation_mean,
    predicted_observation_sigma
82
```

Listing 9: Code Snippet

c. (5 pts) Because the images are quite noisy and the red ball may be partially or completely occluded, your detector is likely to produce some false detections. In the folder `data/Q3D_data` you will find a set of 1000 images that show a trajectory of the red ball where some images are blank (as if the ball is occluded by a white object). Discuss what happens if you do not reject these outliers but instead use them to update the state estimate. Like in the previous question, run your linear Kalman Filter using the sequence of images as input that are corrupted by occlusions (this is also provided in the notebook). Plot the resulting estimated trajectory of the ball over the ground truth trajectory. Also plot the 3-D trajectory in 2-D (x vs. z) and (y vs. z) to better visualize what happens to your filter.

If we do not reject the outliers, then the estimate obtained by the Kalman Filter will start to deviate and will be way off from the true state. Slowly, as more measurements (inliers) are

received, the Kalman Filter estimates will converge back. The plots are shown in Figures 8 and 9.

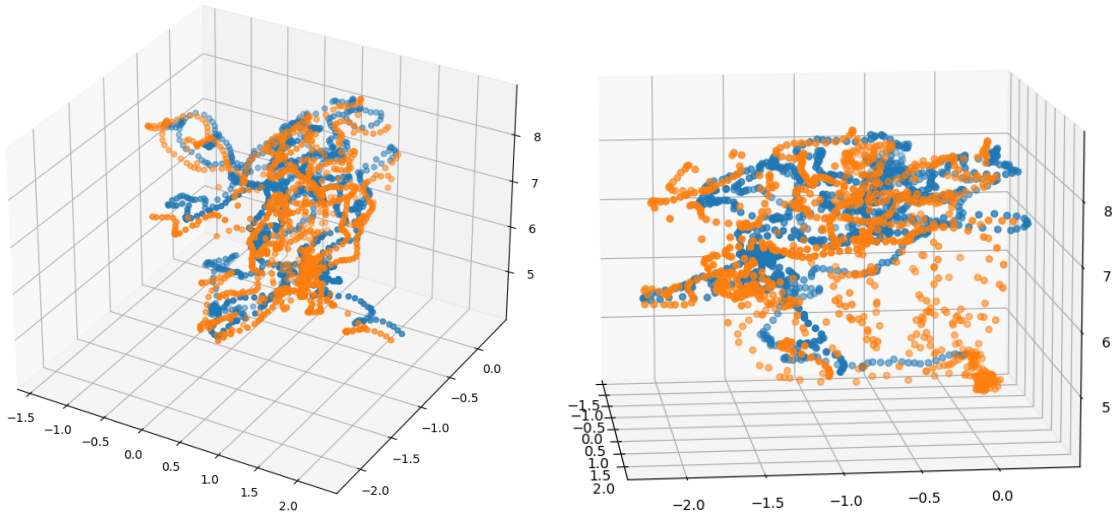

Figure 8: Estimated and ground-truth trajectory from two different viewpoints.

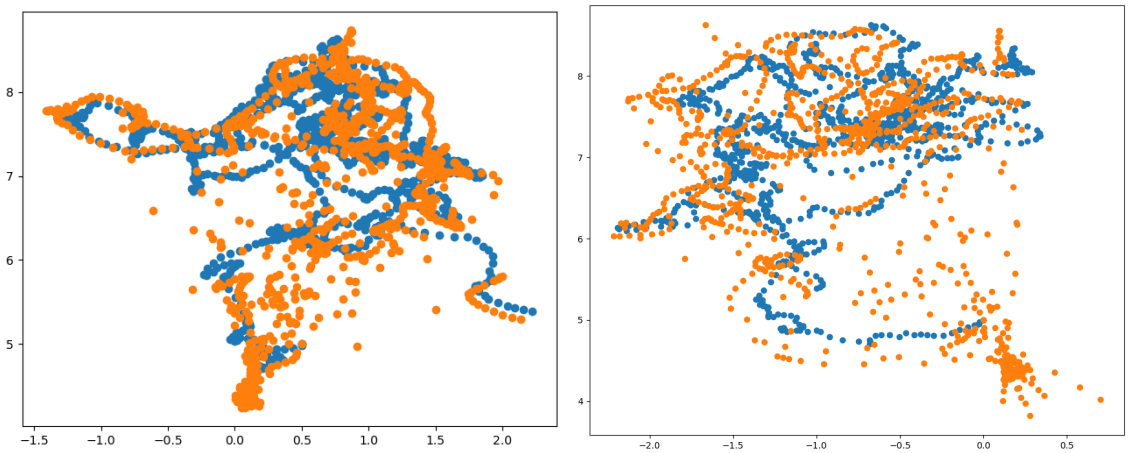

Figure 9: Estimated and ground-truth trajectory (x vs. z) and (y vs. z)

d. (10 pts) Design an outlier detector and use the data from `data/Q3D_data`. Provide the same plots as in part c with `filter_outliers=True`. Explain how you implemented your outlier detector and add your code to the report. Hint: Your observation model predicts where your measurement is expected to occur and its uncertainty.

For the outlier detection, I used Mahalanobis distance to compute the deviation between expected observation and measured observation. If this deviation is greater than certain threshold (10), then I consider it as an outlier and set the updated state to expected state. The plots are shown in Figures 10, 11, 12, and the implementation of Kalman Filter with outlier rejection is given in Listing 10.

```
def KF(self, observations, mu_0, sigma_0, remove_outliers):
    """ Implement Kalman filtering
    Input:
      observations: (N,3) numpy array, the sequence of observations. From T=1.
      mu_0: (6,) numpy array, the mean of state belief after T=0
```

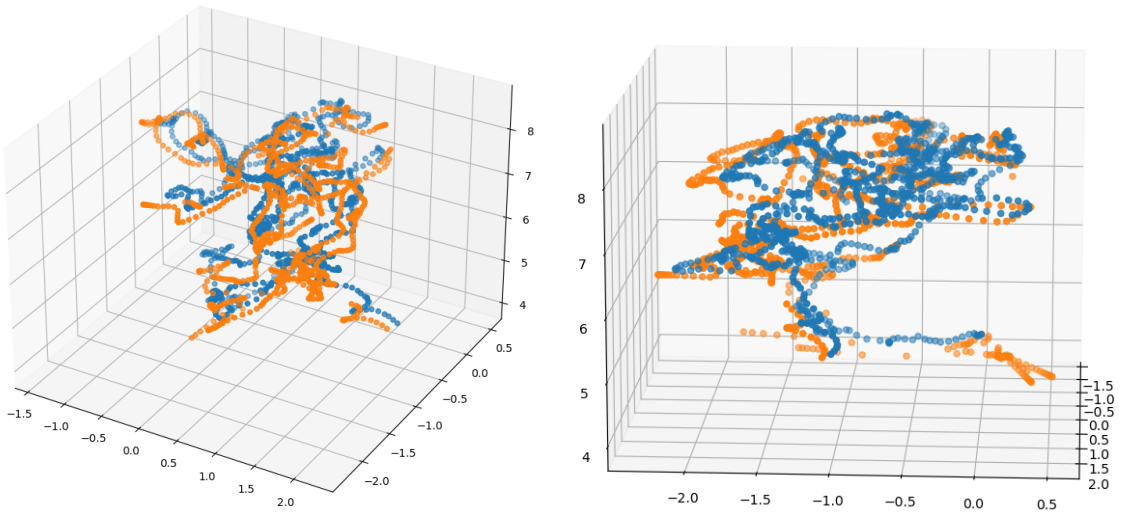

Figure 10: Estimated and ground-truth trajectory with outlier filtering from two different viewpoints.

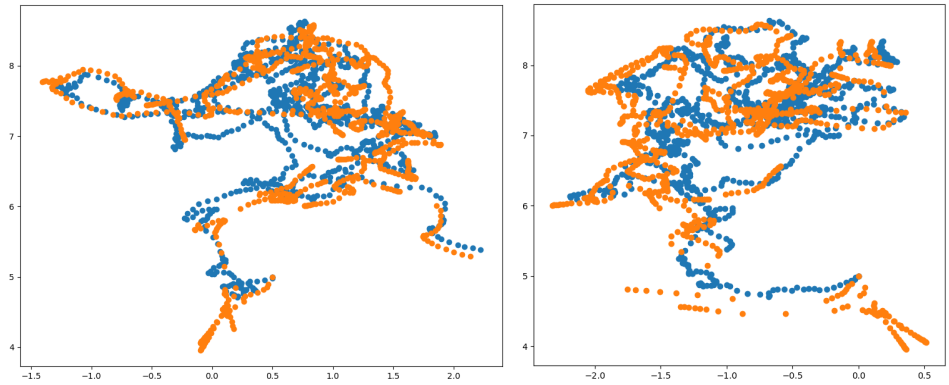

Figure 11: Estimated and ground-truth trajectory outlier filtering (x vs. z) and (y vs. z)

```
    sigma_0: (6,6) numpy array, the covariance matrix for state belief after T=0.
    remove_outliers: bool, whether to remove outliers
  Output:
    state_mean: (N,6) numpy array, the filtered mean state at each time step. Not including the
                starting state mu_0.
    state_sigma: (N,6,6) numpy array, the filtered state covariance at each time step. Not including
                the starting state covarance matrix sigma_0.
    predicted_observation_mean: (N,3) numpy array, the mean of predicted observations. Start from T=1
    predicted_observation_sigma: (N,3,3) numpy array, the covariance matrix of predicted observations. Start from T=1
  """
  A = self.system_matrix()
  Q = self.process_noise_covariance()
  R = self.observation_noise_covariance()
  state_mean = [mu_0]
  state_sigma = [sigma_0]
  predicted_observation_mean = []
  predicted_observation_sigma = []
```

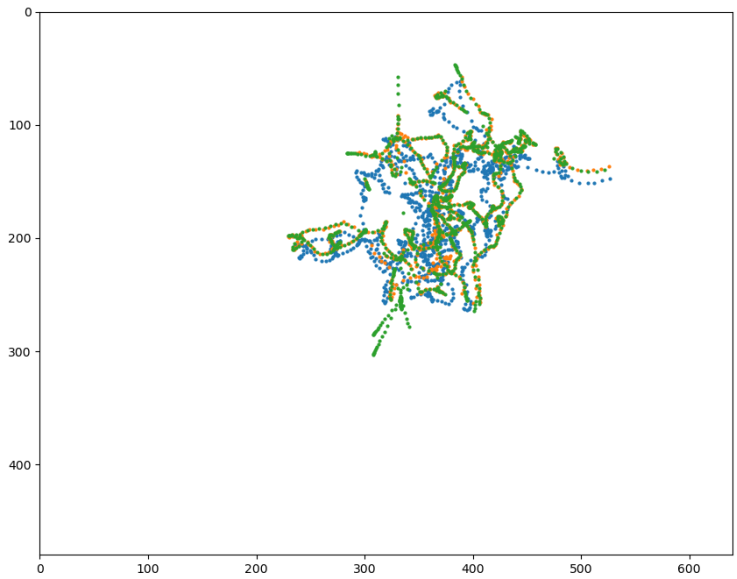

Figure 12: 2D projection of estimated and ground-truth trajectory (blue: ground-truth, orange: detection projection, green: filtered state projection

```
    for ob in observations:
        mu_bar_next = np.dot(A, state_mean[-1])
        sigma_bar_next = np.dot(A, np.dot(state_sigma[-1], A.T)) + Q
        # H is the measurement model which casts next state onto an
    ovservation vector
        H = np.array([[1., 0., 0., 0., 0., 0.],
                      [0., 1., 0., 0., 0., 0.],
                      [0., 0., 1., 0., 0., 0.]], dtype=np.float64)
        kalman_gain_numerator = np.dot(sigma_bar_next, H.T)
        kalman_gain_denominator = np.dot(H, np.dot(sigma_bar_next, H.T)) + R #
     this is the innovation covariance matrix, S
        kalman_gain = np.dot(kalman_gain_numerator, np.linalg.inv(
    kalman_gain_denominator))
        expected_observation = np.dot(H, mu_bar_next)
        # let's compute Mahalanobis distance
        S = kalman_gain_denominator
        deviation = np.sqrt(np.dot((ob - expected_observation).T, np.dot(np.
    linalg.inv(S), (ob - expected_observation))))
        if not filter_outliers or deviation <= 10:# part D
          mu_next = mu_bar_next + np.dot(kalman_gain, (ob -
    expected_observation).T)
          sigma_next = np.dot((np.eye(6, dtype=np.float64) - np.dot(
    kalman_gain, H)), sigma_bar_next)
        else:
            mu_next = mu_bar_next
            sigma_next = sigma_bar_next
        state_mean.append(mu_next)
        state_sigma.append(sigma_next)
        predicted_observation_mean.append(expected_observation)
        predicted_observation_sigma.append(kalman_gain_denominator)
    return state_mean, state_sigma, predicted_observation_mean,
    predicted_observation_sigma
```

Listing 10: Code Snippet