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

(Winter 2021) Due: **Friday, March 11**

Vasu G. Patel: vgpatel1@stanford.edu
*worked in collaboration with Raghav Khandelwal

1 Overview

This PSET will involve concepts from lectures 14 onwards. To give you more time to finish your project, it will be the shortest of the PSETs.

2 Submitting

Please put together a PDF with your answers for each problem, and submit it to the appropriate assignment on Gradescope. We recommend you to add these answers to the latex template files on our website, but you can also create a PDF in any other way you prefer. For the written report, in the case of problems that just involve implementing code you can include only the final output and in some cases a brief description if requested in the problem. There will be an additional coding assignment on Gradescope that has an autograder that is there to help you double check your code. Make sure you use the provided ".py" files to write your Python code. Submit to both the PDF and code assignment, as we will be grading the PDF submissions and using the coding assignment to check your code if needed.

3 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

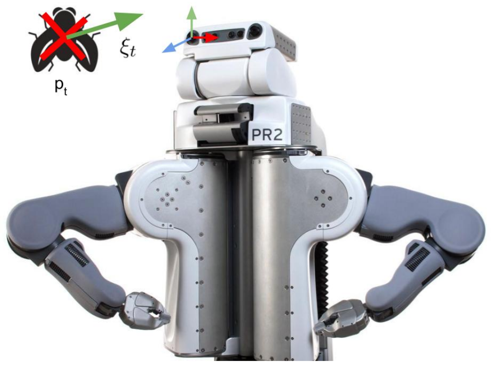


Figure 1

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).

Answer 3(a):

$SystemMatrix:$

$$\begin{bmatrix} 1. & 0. & 0. & 0.1 & 0. & 0. \\ 0. & 1. & 0. & 0. & 0.1 & 0. \\ 0. & 0. & 1. & 0. & 0. & 0.1 \\ 0. & 0. & 0. & 0.8 & 0. & 0. \\ 0. & 0. & 0. & 0. & 0.8 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0.8 \end{bmatrix}$$

Process Noise Co-variance:

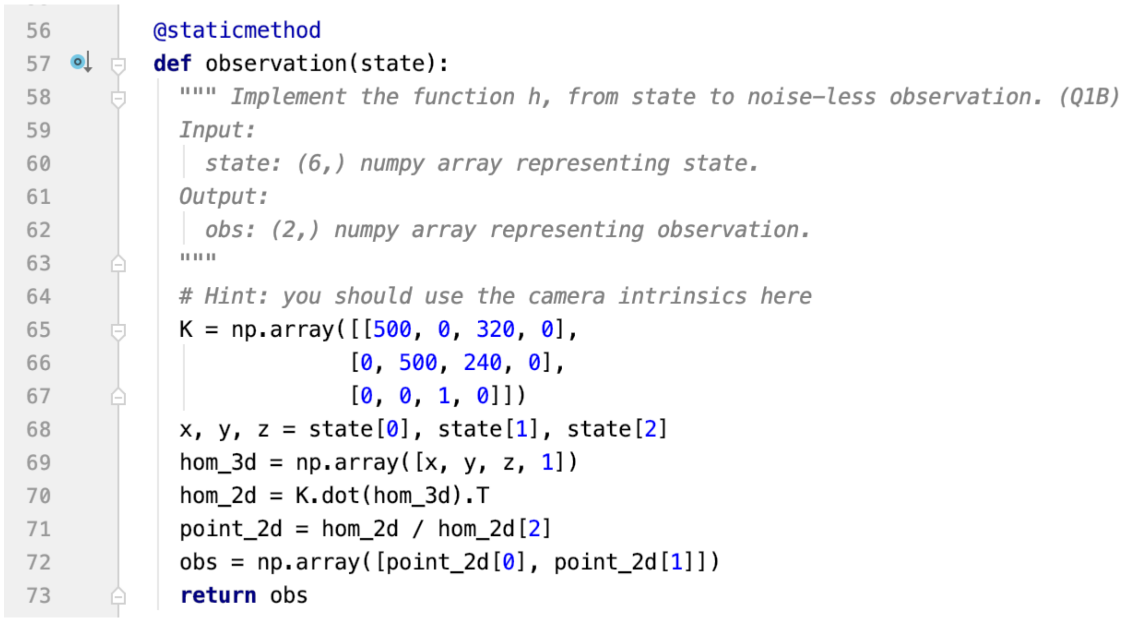
$$\begin{bmatrix} \begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0.05 & 0. & 0. \\ 0. & 0. & 0. & 0. & 0.05 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0.05 \end{bmatrix} \end{bmatrix}$$

Observation noise co-variance:

$$\begin{bmatrix} 5.0 & 0.0 \\ 0.0 & 5.0 \end{bmatrix}$$

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

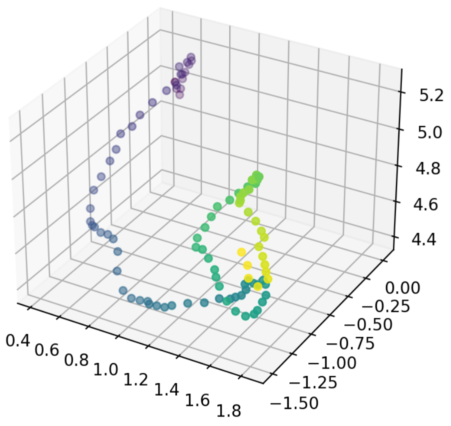
Answer 3(b): Code also attached in the gradescope.

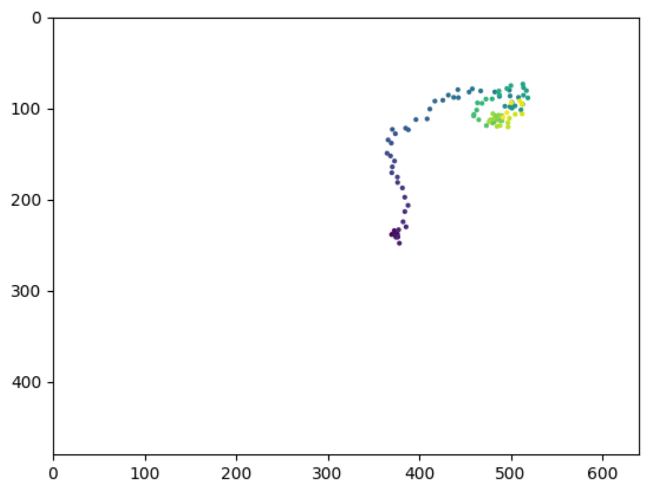


```python
@staticmethod
def observation(state):
    """ Implement the function h, from state to noise-less observation. (Q1B)
    Input:
      state: (6,) numpy array representing state.
    Output:
      obs: (2,) numpy array representing observation.
    """
    # Hint: you should use the camera intrinsics here
    K = np.array([[500, 0, 320, 0],
                  [0, 500, 240, 0],
                  [0, 0, 1, 0]])
    x, y, z = state[0], state[1], state[2]
    hom_3d = np.array([x, y, z, 1])
    hom_2d = K.dot(hom_3d).T
    point_2d = hom_2d / hom_2d[2]
    obs = np.array([point_2d[0], point_2d[1]])
    return obs
```

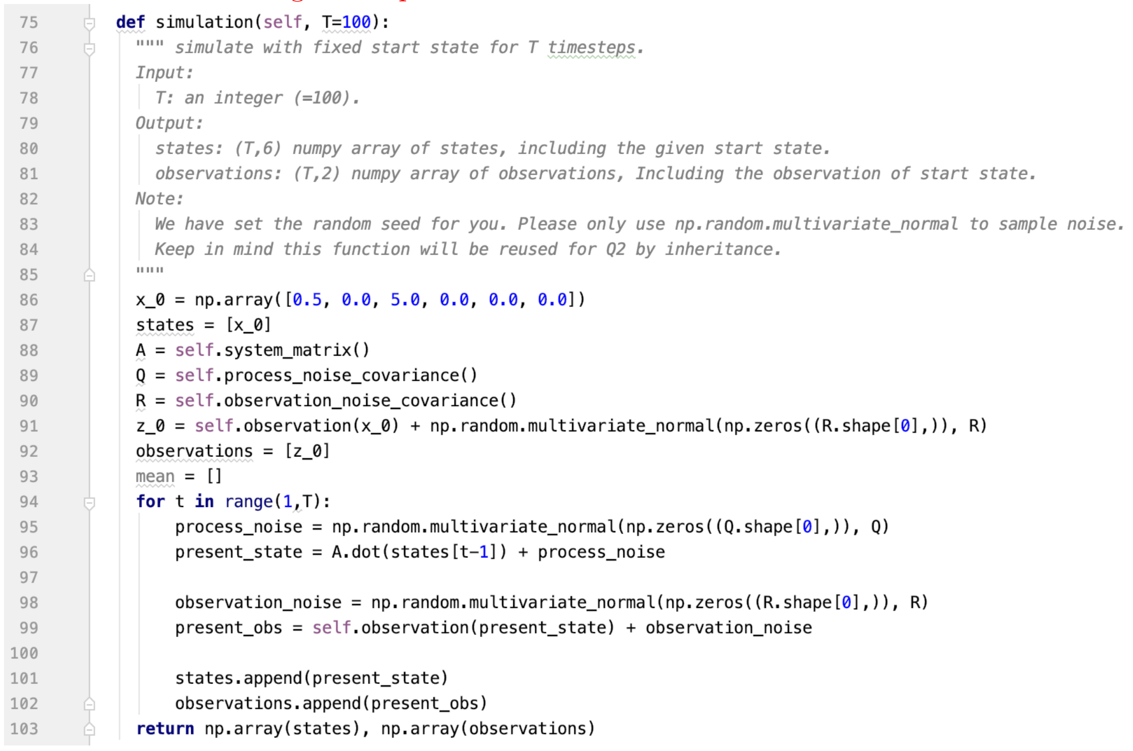
(8 pts) Initially, the fly is sitting on the fingertip of the robot when it is noticing it for the first time. Therefore, the robot knows the fly's initial position from forward kinematics to be at $\mathbf{p}_0 = (0.5, 0, 5.0)^T$ (resting velocity). Simulate in Python the 3D trajectory that the fly 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.

Answer 3(c):





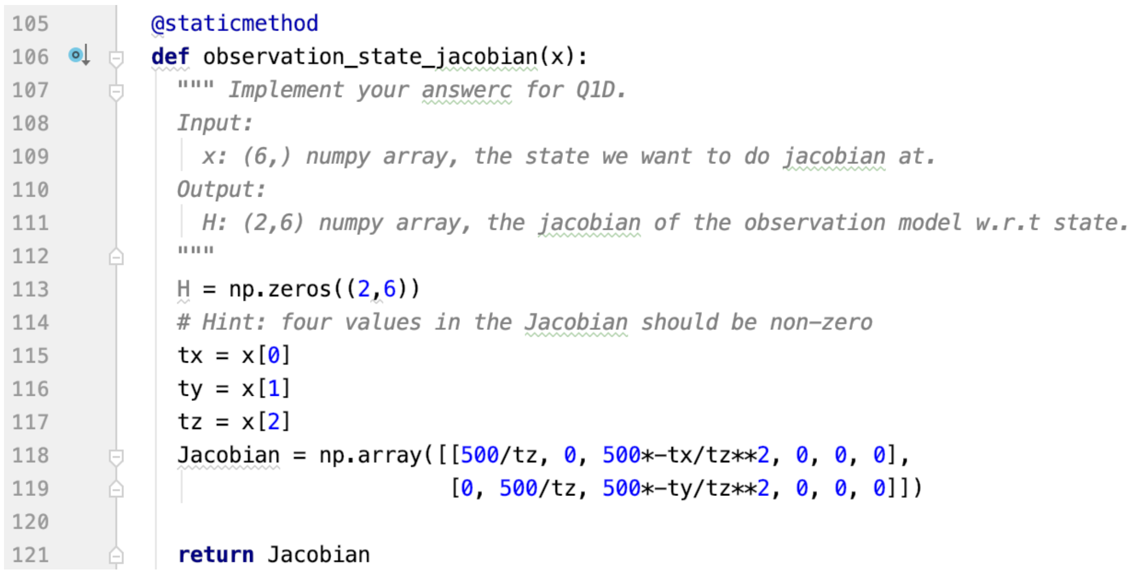
Code also attached in gradescope:



```python
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]
    mean = []
    for t in range(1,T):
        process_noise = np.random.multivariate_normal(np.zeros((Q.shape[0],)), Q)
        present_state = A.dot(states[t-1]) + process_noise

        observation_noise = np.random.multivariate_normal(np.zeros((R.shape[0],)), R)
        present_obs = self.observation(present_state) + observation_noise

        states.append(present_state)
        observations.append(present_obs)
    return np.array(states), np.array(observations)
```

(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.)

Answer 3(d):
code also attached in gradescope

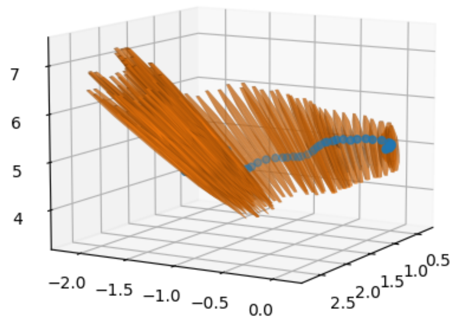


```
@staticmethod
def observation_state_jacobian(x):
    """ Implement your answerc for Q1D.
    Input:
        x: (6,) numpy array, the state we want to do jacobian at.
    Output:
        H: (2,6) numpy array, the jacobian of the observation model w.r.t state.
    """
    H = np.zeros((2,6))
    # Hint: four values in the Jacobian should be non-zero
    tx = x[0]
    ty = x[1]
    tz = x[2]
    Jacobian = np.array([[500/tz, 0, 500*-tx/tz**2, 0, 0, 0],
                         [0, 500/tz, 500*-ty/tz**2, 0, 0, 0]])

    return Jacobian
```

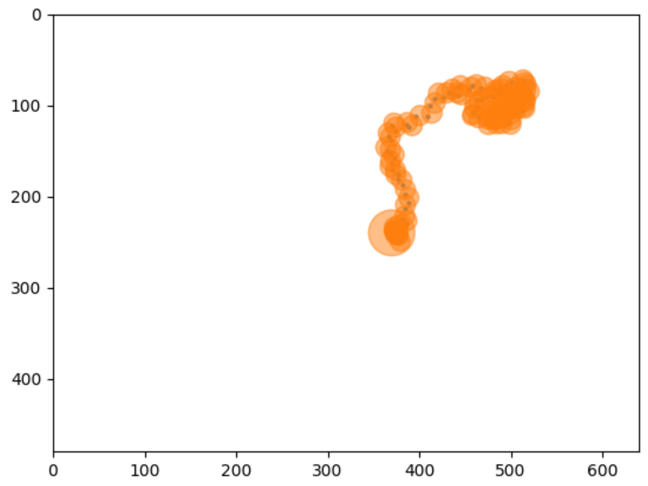
(20 pts) Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the camera. You can assume the aforementioned initial position and the following initial error covariance matrix: $P_0 = \mathrm{diag}(0.1, 0.1, 0.1)$. The measurements can be found in `data/Q1E_measurement.npy`. 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`

Answer 3(e):

3D Predictions and observed measurement



2D Predictions and observed measurement



(8 pts) Discuss the difference in magnitude of uncertainty in the different dimensions of the state.
Answer 3(f):
Highest uncertainty is found in z direction among x, y, & z, because using single camera for predicting depth will involve highest magnitude of uncertainty. Uncertainty between x & y is negligible because looking at the 3D prediction graph from 3(e) we observe that rods are nearly circular and are not stretched in either x or y direction. In addition looking at the 2D prediction graph from 3(e) we observed 2D circular rather than ellipse prediction indicating circular plots.

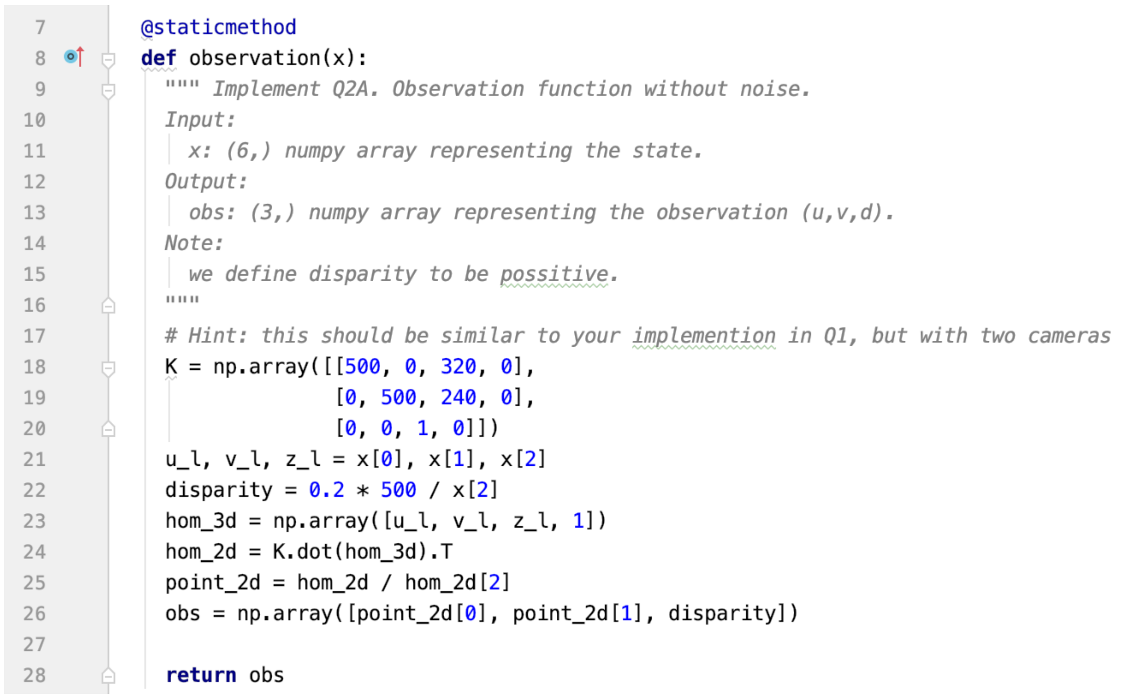
Uncertainty along v_x, v_y, v_z is even negligible as compared to x, y, & z because velocity terms are normally distributed noise.

4 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.

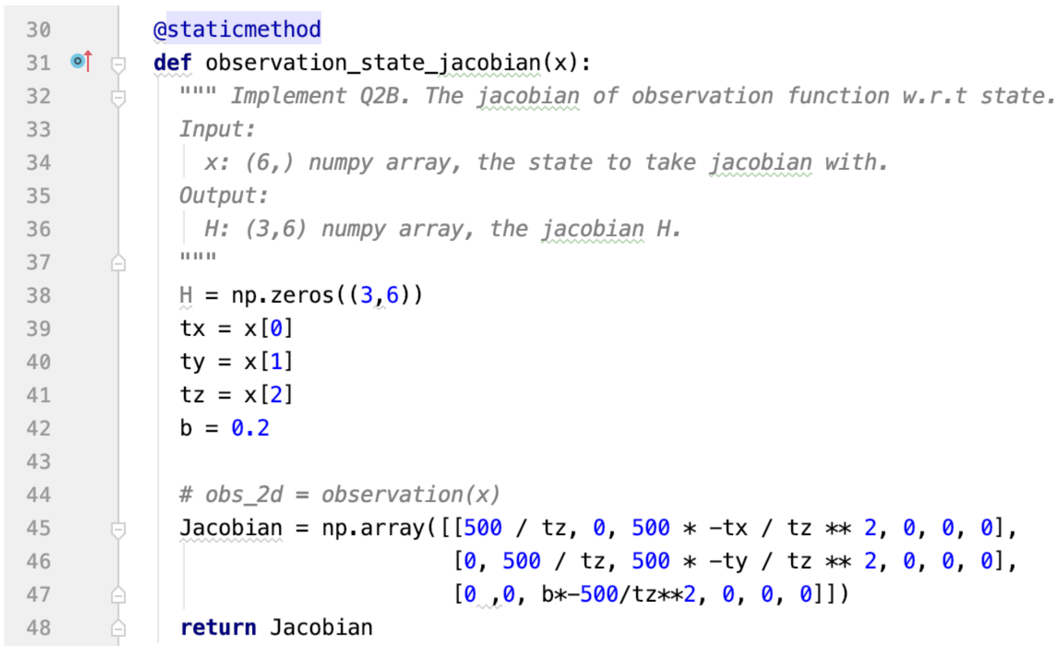
Now the robot receives as measurement $\mathbf{z}$ a pair of pixel coordinates in the left image (u^L, v^L) and right image (u^R, v^R) of the camera. Since our camera system is perfectly parallel, we will assume a measurement vector $\mathbf{z} = (u^L, v^L, d^L)$ where d^L is the disparity between the 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).
Answer 4(a): code also attached in gradescope.



```python
@staticmethod
def observation(x):
    """ Implement Q2A. 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.
    """
    # Hint: this should be similar to your implemention in Q1, but with two cameras
    K = np.array([[500, 0, 320, 0],
                  [0, 500, 240, 0],
                  [0, 0, 1, 0]])
    u_l, v_l, z_l = x[0], x[1], x[2]
    disparity = 0.2 * 500 / x[2]
    hom_3d = np.array([u_l, v_l, z_l, 1])
    hom_2d = K.dot(hom_3d).T
    point_2d = hom_2d / hom_2d[2]
    obs = np.array([point_2d[0], point_2d[1], disparity])

    return obs
```

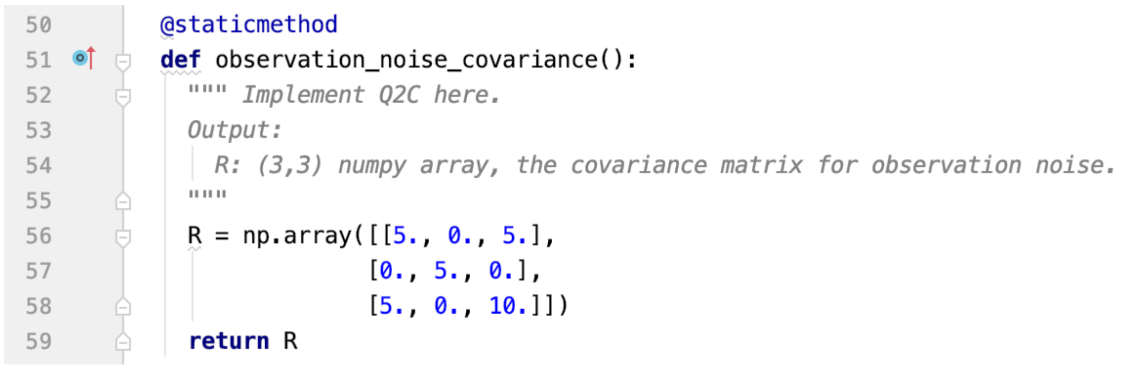
b. (6 pts) Find the Jacobian H of the observation model with respect to the fly's state x. (Implement H in function *observation_state_jacobian* in Q2.py)
Answer 4(b): code also attached in gradescope.



```python
@staticmethod
def observation_state_jacobian(x):
    """ Implement Q2B. The jacobian of observation function w.r.t state.
    Input:
        x: (6,) numpy array, the state to take jacobian with.
    Output:
        H: (3,6) numpy array, the jacobian H.
    """
    H = np.zeros((3,6))
    tx = x[0]
    ty = x[1]
    tz = x[2]
    b = 0.2

    # obs_2d = observation(x)
    Jacobian = np.array([[500 / tz, 0, 500 * -tx / tz ** 2, 0, 0, 0],
                         [0, 500 / tz, 500 * -ty / tz ** 2, 0, 0, 0],
                         [0 ,0, b*-500/tz**2, 0, 0, 0]])
    return Jacobian
```

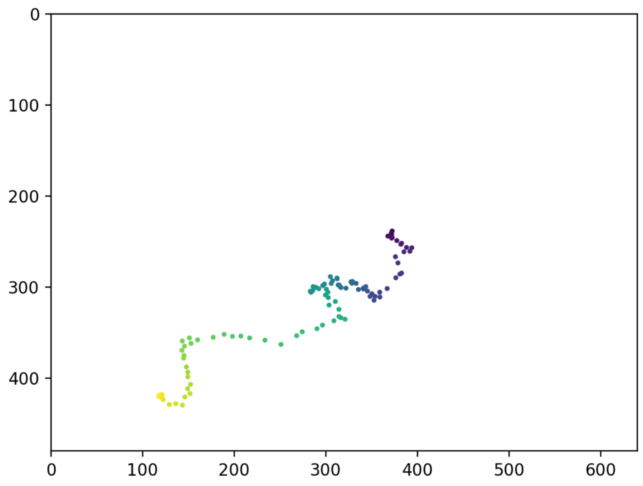
c. (6 pts) What is the new observation noise covariance matrix R? Assume the noise on (u^L, v^L), and (u^R, v^R) to be independent and to have the same distribution as the observation noise given in Question 1, respectively. (Implement R in function *observation_noise_covariance* in Q2.py).

Answer 4(c): code also attached in gradescope.

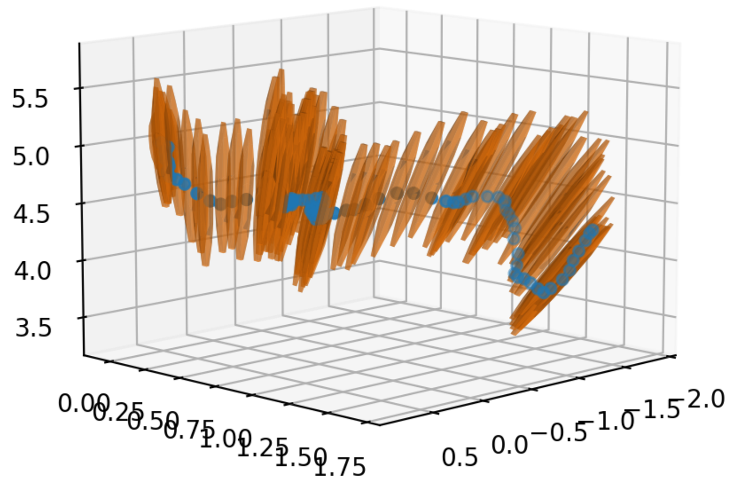


```python
@staticmethod
def observation_noise_covariance():
    """ Implement Q2C here.
    Output:
        R: (3,3) numpy array, the covariance matrix for observation noise.
    """
    R = np.array([[5., 0., 5.],
                  [0., 5., 0.],
                  [5., 0., 10.]])
    return R
```

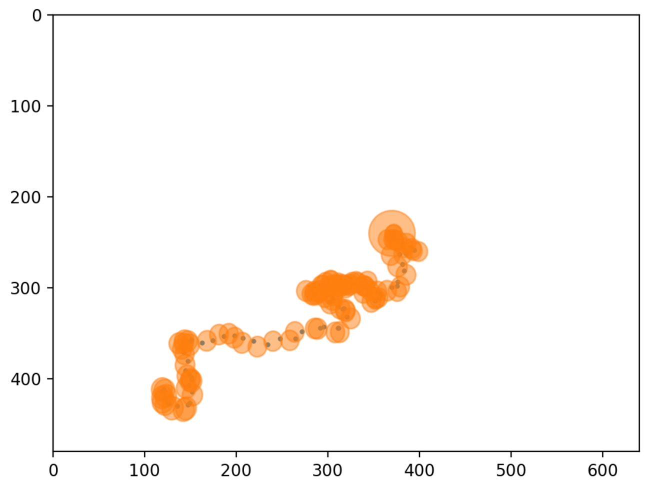
d. (6 pts) Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the left camera. You can assume the same initial position and the initial error covariance matrix as in the previous questions. Plot the means and error ellipses of the predicted measurements 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.
Answer 4(d):
2D prediction and observed values



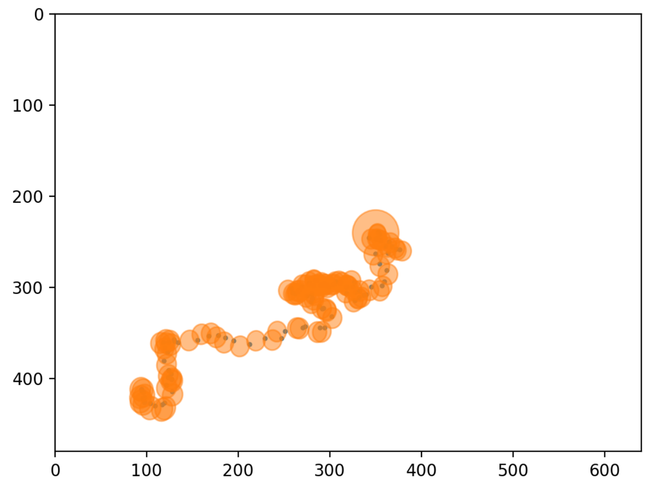
3D prediction and observed values



Left image:



Right image:



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)!
Answer 4(e):
Pros: compute depth using disparity from left and right rectified images
Cons: If the baseline of stereo cameras is big, it will result in occlusion and if the baseline is very small resulting reprojection error will be quite high.

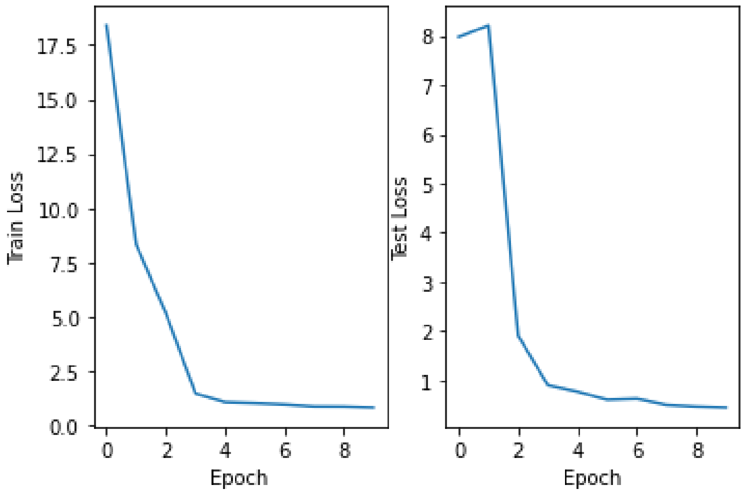
5 Linear Kalman Filter with a Learned Inverse Observation Model (40 points)

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.

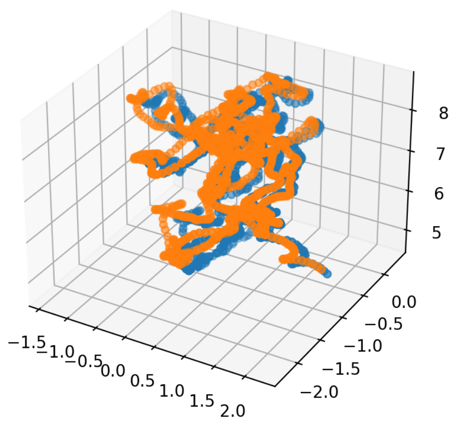
a. (10 pts) In the folder `data/Q3A_data` you will find a training set of 1000 images in the subfolder `training_set` and the file `Q3A_positions_train.npy` that contains the ground truth 3D position of the red ball in the image. We have provided you with the notebook `LearnedObservationModel.ipynb` 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.

Answer 5(a):
Final training loss: 0.81319535
Final testing loss: 0.45170546

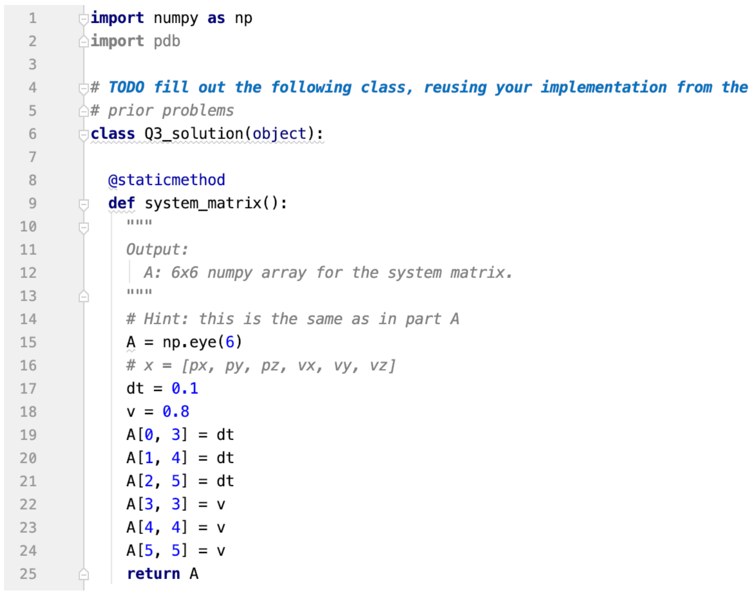


b. (30 pts) In the folder `data/Q3B_data` you will find a set of 1000 images that show a new trajectory of the red ball. Run your linear Kalman Filter using this sequence of images as input, where your learned model provides the noisy measurements (the logic for this is provided in `LearnedObservationModel.ipynb`). 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.

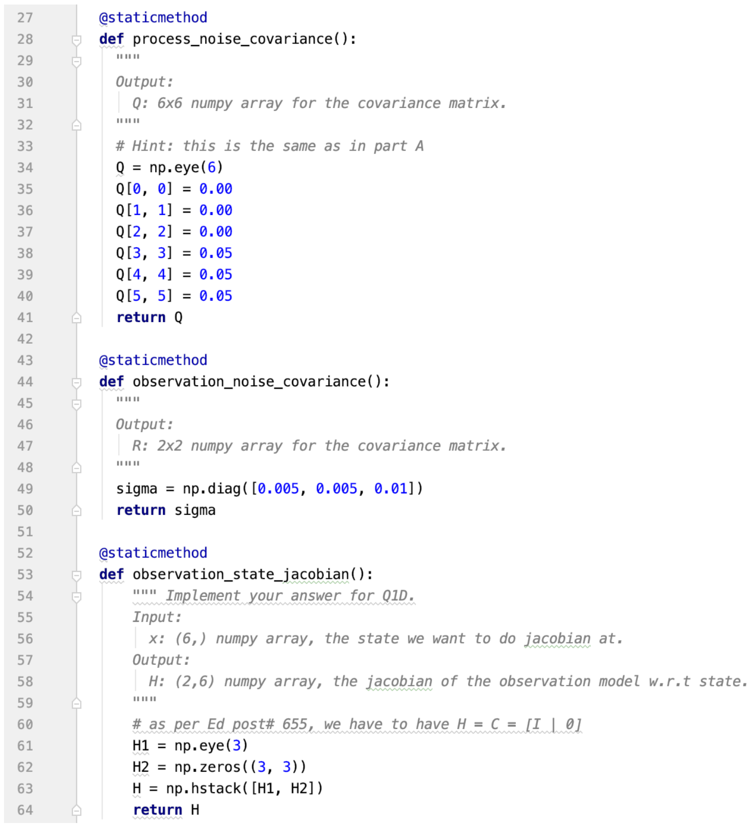
Answer 5(b):
When the part_b_data was set to True in the main function, I obtain following graph:



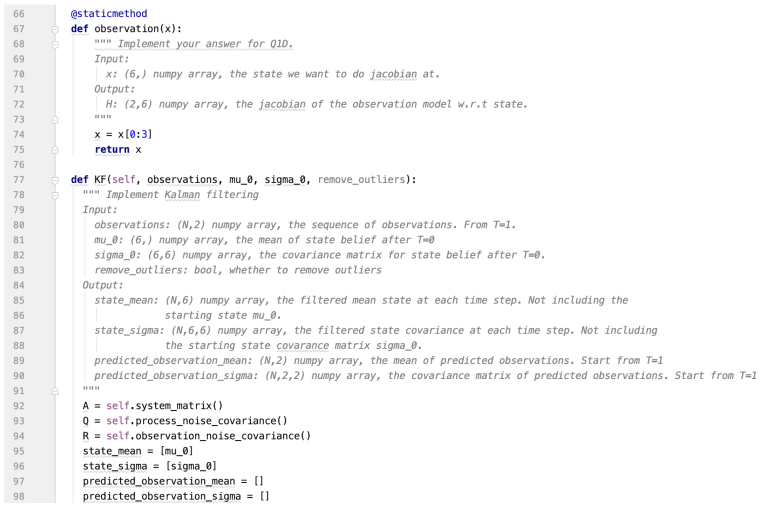
Code"



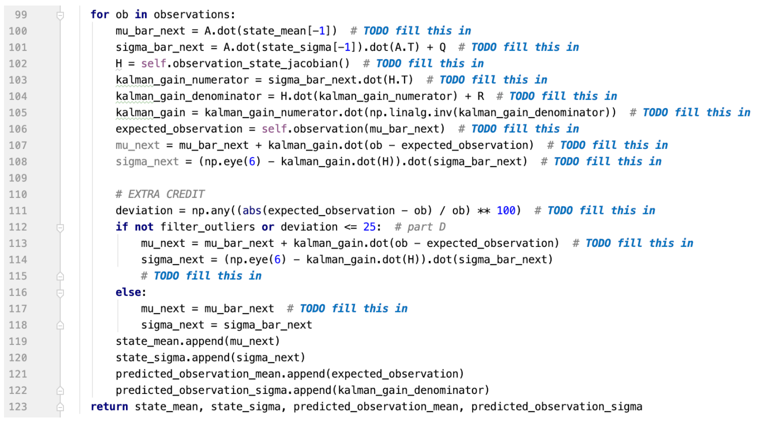
```python
import numpy as np
import pdb

# TODO fill out the following class, reusing your implementation from the
# prior problems
class Q3_solution(object):

  @staticmethod
  def system_matrix():
    """
    Output:
      A: 6x6 numpy array for the system matrix.
    """
    # Hint: this is the same as in part A
    A = np.eye(6)
    # x = [px, py, pz, vx, vy, vz]
    dt = 0.1
    v = 0.8
    A[0, 3] = dt
    A[1, 4] = dt
    A[2, 5] = dt
    A[3, 3] = v
    A[4, 4] = v
    A[5, 5] = v
    return A
```



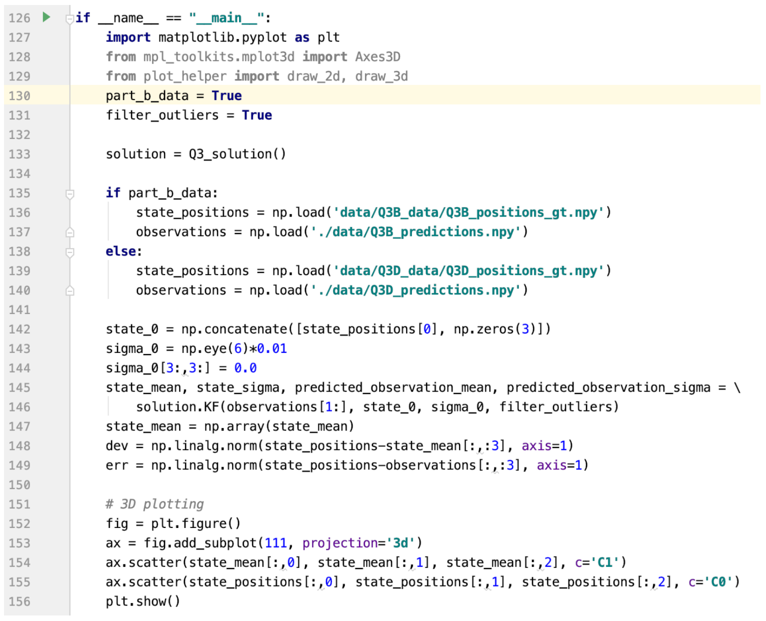
```python
  @staticmethod
  def process_noise_covariance():
    """
    Output:
      Q: 6x6 numpy array for the covariance matrix.
    """
    # Hint: this is the same as in part A
    Q = np.eye(6)
    Q[0, 0] = 0.00
    Q[1, 1] = 0.00
    Q[2, 2] = 0.00
    Q[3, 3] = 0.05
    Q[4, 4] = 0.05
    Q[5, 5] = 0.05
    return Q

  @staticmethod
  def observation_noise_covariance():
    """
    Output:
      R: 2x2 numpy array for the covariance matrix.
    """
    sigma = np.diag([0.005, 0.005, 0.01])
    return sigma

  @staticmethod
  def observation_state_jacobian():
      """ Implement your answer for Q1D.
      Input:
        x: (6,) numpy array, the state we want to do jacobian at.
      Output:
        H: (2,6) numpy array, the jacobian of the observation model w.r.t state.
      """
      # as per Ed post# 655, we have to have H = C = [I | 0]
      H1 = np.eye(3)
      H2 = np.zeros((3, 3))
      H = np.hstack([H1, H2])
      return H
```



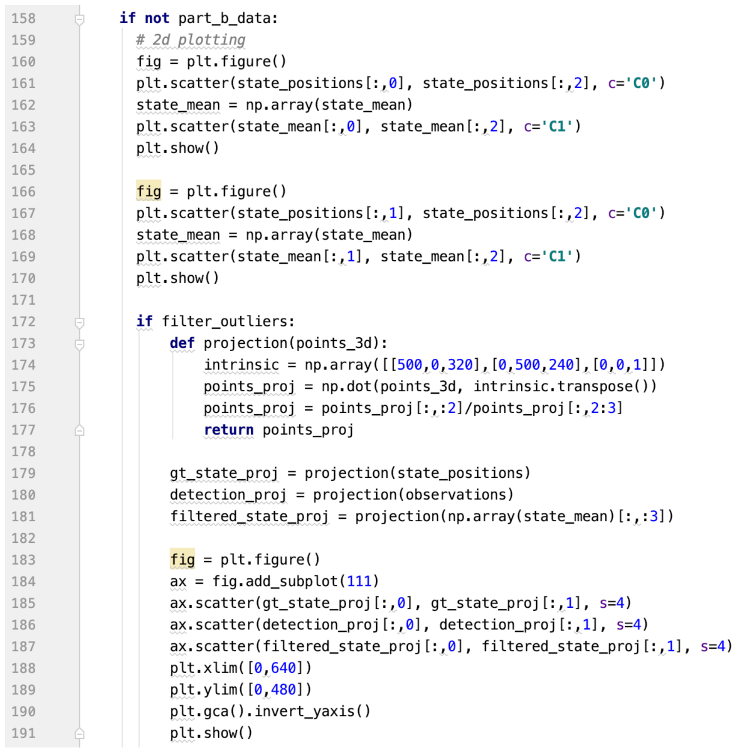
```python
@staticmethod
def observation(x):
    """ Implement your answer for Q1D.
    Input:
        x: (6,) numpy array, the state we want to do jacobian at.
    Output:
        H: (2,6) numpy array, the jacobian of the observation model w.r.t state.
    """
    x = x[0:3]
    return x

def KF(self, observations, mu_0, sigma_0, remove_outliers):
  """ Implement 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.
    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,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
  """
  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 = []
```



```python
  for ob in observations:
      mu_bar_next = A.dot(state_mean[-1])  # TODO fill this in
      sigma_bar_next = A.dot(state_sigma[-1]).dot(A.T) + Q  # TODO fill this in
      H = self.observation_state_jacobian()  # TODO fill this in
      kalman_gain_numerator = sigma_bar_next.dot(H.T)  # TODO fill this in
      kalman_gain_denominator = H.dot(kalman_gain_numerator) + R  # TODO fill this in
      kalman_gain = kalman_gain_numerator.dot(np.linalg.inv(kalman_gain_denominator))  # TODO fill this in
      expected_observation = self.observation(mu_bar_next)  # TODO fill this in
      mu_next = mu_bar_next + kalman_gain.dot(ob - expected_observation)  # TODO fill this in
      sigma_next = (np.eye(6) - kalman_gain.dot(H)).dot(sigma_bar_next)  # TODO fill this in

      # EXTRA CREDIT
      deviation = np.any((abs(expected_observation - ob) / ob) ** 100)  # TODO fill this in
      if not filter_outliers or deviation <= 25:  # part D
          mu_next = mu_bar_next + kalman_gain.dot(ob - expected_observation)  # TODO fill this in
          sigma_next = (np.eye(6) - kalman_gain.dot(H)).dot(sigma_bar_next)
          # TODO fill this in
      else:
          mu_next = mu_bar_next  # TODO fill this in
          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
```



```python
if __name__ == "__main__":
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    from plot_helper import draw_2d, draw_3d
    part_b_data = True
    filter_outliers = True

    solution = Q3_solution()

    if part_b_data:
        state_positions = np.load('data/Q3B_data/Q3B_positions_gt.npy')
        observations = np.load('./data/Q3B_predictions.npy')
    else:
        state_positions = np.load('data/Q3D_data/Q3D_positions_gt.npy')
        observations = np.load('./data/Q3D_predictions.npy')

    state_0 = np.concatenate([state_positions[0], np.zeros(3)])
    sigma_0 = np.eye(6)*0.01
    sigma_0[3:,3:] = 0.0
    state_mean, state_sigma, predicted_observation_mean, predicted_observation_sigma = \
        solution.KF(observations[1:], state_0, sigma_0, filter_outliers)
    state_mean = np.array(state_mean)
    dev = np.linalg.norm(state_positions-state_mean[:,:3], axis=1)
    err = np.linalg.norm(state_positions-observations[:,:3], axis=1)

    # 3D plotting
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(state_mean[:,0], state_mean[:,1], state_mean[:,2], c='C1')
    ax.scatter(state_positions[:,0], state_positions[:,1], state_positions[:,2], c='C0')
    plt.show()
```

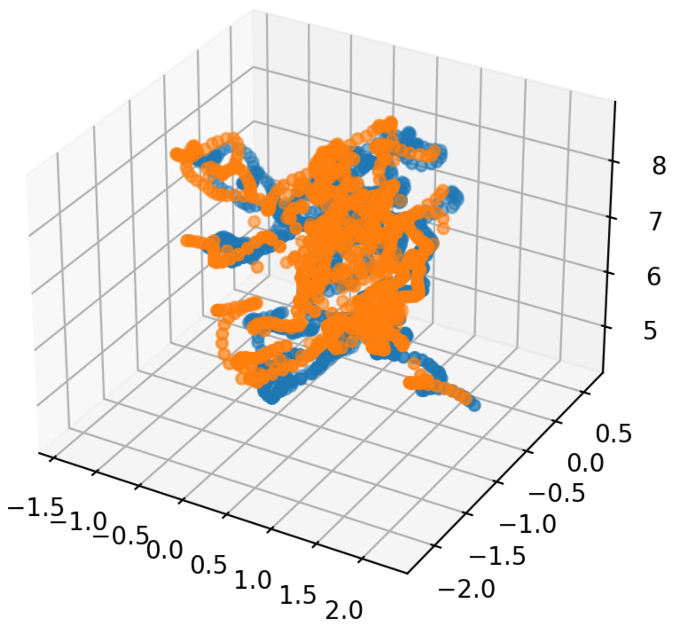


```python
if not part_b_data:
    # 2d plotting
    fig = plt.figure()
    plt.scatter(state_positions[:,0], state_positions[:,2], c='C0')
    state_mean = np.array(state_mean)
    plt.scatter(state_mean[:,0], state_mean[:,2], c='C1')
    plt.show()

    fig = plt.figure()
    plt.scatter(state_positions[:,1], state_positions[:,2], c='C0')
    state_mean = np.array(state_mean)
    plt.scatter(state_mean[:,1], state_mean[:,2], c='C1')
    plt.show()

    if filter_outliers:
        def projection(points_3d):
            intrinsic = np.array([[500,0,320],[0,500,240],[0,0,1]])
            points_proj = np.dot(points_3d, intrinsic.transpose())
            points_proj = points_proj[:,:2]/points_proj[:,2:3]
            return points_proj

        gt_state_proj = projection(state_positions)
        detection_proj = projection(observations)
        filtered_state_proj = projection(np.array(state_mean)[:,:3])

        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.scatter(gt_state_proj[:,0], gt_state_proj[:,1], s=4)
        ax.scatter(detection_proj[:,0], detection_proj[:,1], s=4)
        ax.scatter(filtered_state_proj[:,0], filtered_state_proj[:,1], s=4)
        plt.xlim([0,640])
        plt.ylim([0,480])
        plt.gca().invert_yaxis()
        plt.show()
```

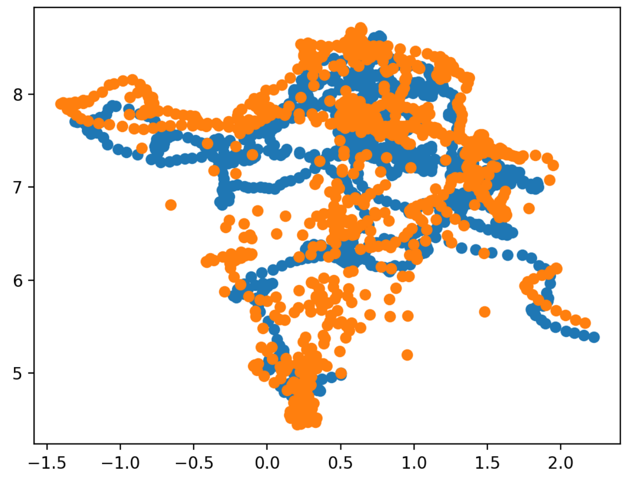
c. **Extra Credit (10 pts)** Because the images are quite noisy and the red ball may be partially or completely occluded, your detector is likely to produce some false detections. In the folder `data/Q3D_data` you will find a set of 1000 images that show a trajectory of the red ball where some images 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.

Answer 5(c):

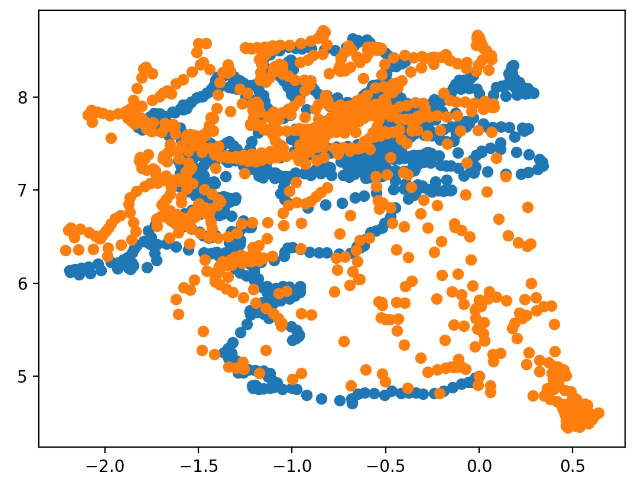
When the part_b_data was set to False in the main function, I obtain following graph:



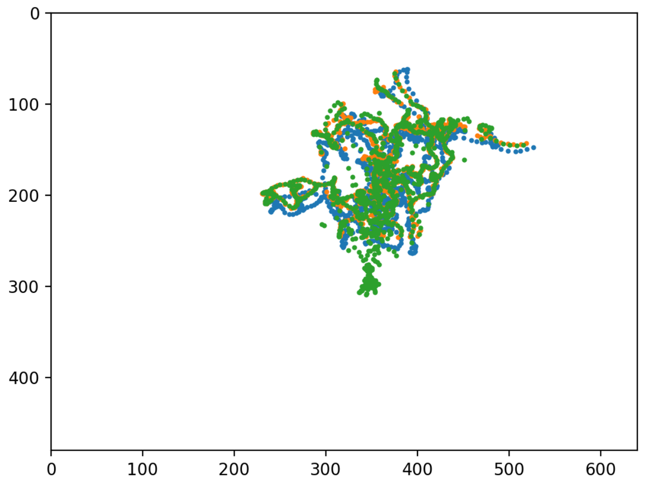
3D trajectory in 2D: x vs. z



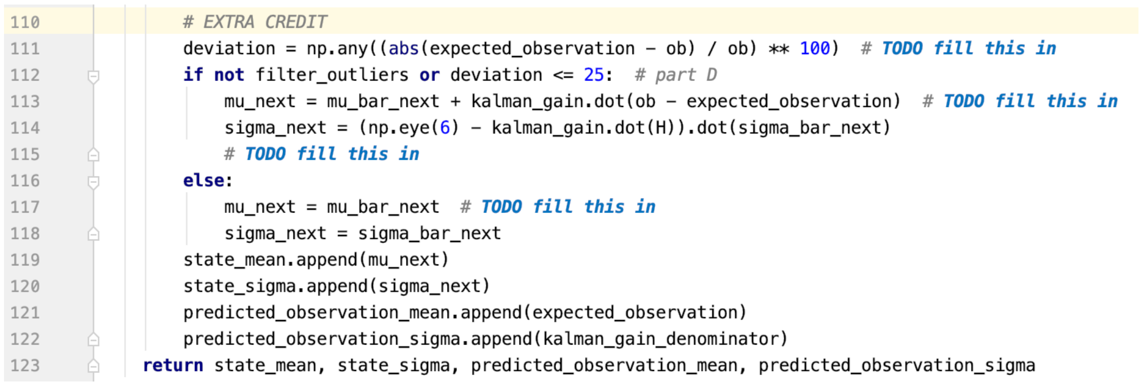
3D trajectory in 2D: y vs. z



d. **Extra Credit (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.
Answer 5(d):
When I set my outlier detector to True using the threshold of 25%. Deviation is described as innovation in the lecture and we calculate the difference between true observation and predicted observation and if deviation is greater than 25%, I set mu_next = mu_bar_next.



Code:



```python
        # EXTRA CREDIT
        deviation = np.any((abs(expected_observation - ob) / ob) ** 100)  # TODO fill this in
        if not filter_outliers or deviation <= 25:  # part D
            mu_next = mu_bar_next + kalman_gain.dot(ob - expected_observation)  # TODO fill this in
            sigma_next = (np.eye(6) - kalman_gain.dot(H)).dot(sigma_bar_next)
            # TODO fill this in
        else:
            mu_next = mu_bar_next  # TODO fill this in
            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
```