Assignment 3 – Advanced Robot Navigation (RBE595 Spring 2024)

Rohin Siddhartha Palaniappan Venkateswaran

Task 1 and 2

Pose Estimation and Visualization

The goal of this task, as described in the assignment is to estimate the pose of the drone from the camera image at a certain timestamp. Further, we need to plot the estimated trajectory and the orientations and compare it with the ground truth values given in the **'vicon'** dataset. In order to perform this task, the algorithmic flow is as follows,

1. Read the data from the .mat files provided.
2. Correct for the inconsistencies in the data by performing certain operations.
3. Write a function that calculates the coordinates of all 4 corners of all the april tags as shown in the figure in the assignment.
4. Write a function that takes in the current dataset under consideration and the tag coordinates called "estimate pose"
5. Use the **'solvePNP'** function from openCV to estimate the pose of the drone in the 3 Dimensional space.
6. Apply the correct translational and rotational offsets as provided in the assignment to perform the transformations from the world to camera to drone frame.
7. Plot the estimated trajectory in the same plot as the ground truth trajectory

```python
import numpy as np
import cv2
import scipy.io
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import os

# Function to load data from .mat file
def load_data(filename):
    return scipy.io.loadmat(filename, simplify_cells=True)

def estimate_pose(data, tag_coordinates):

    # Extract 3D coordinates of AprilTag corners from the map layout
    map_corners_3d = []
    for data_id in data['id']:
        for id, corners in tag_coordinates:
            if data_id == id:
                for corner in corners:
                    corner_3d = np.array([corner[0], corner[1], 0])
                    map_corners_3d.append(corner_3d)

    map_corners_3d = np.array(map_corners_3d)

    # Extract 2D projections of AprilTag corners from the image data
```

```
    # The format of the p1 through p4 arrays are not[ [x1, y1], [x2, y2], [x3, y3
                                                ]] like I presumed but its actually
                                                [ [x1, x2, x3], [y1, y2, y3]]

    data['p1'] = np.array(data['p1'])

    if len(data['p1']) == 1:
        data['p1'] = data['p1'].reshape(-1, 1)

    data['p1'] = data['p1'].T

    data['p2'] = np.array(data['p2'])

    if len(data['p2']) == 1:
        data['p2'] = data['p2'].reshape(-1, 1)

    data['p2'] = data['p2'].T

    data['p3'] = np.array(data['p3'])

    if len(data['p3']) == 1:
        data['p3'] = data['p3'].reshape(-1, 1)

    data['p3'] = data['p3'].T

    data['p4'] = np.array(data['p4'])

    if len(data['p4']) == 1:
        data['p4'] = data['p4'].reshape(-1, 1)

    data['p4'] = data['p4'].T

    image_corners_2d = []

    for i in range(len(data['p1'])):
        image_corners_2d.append(data['p4'][i])
        image_corners_2d.append(data['p3'][i])
        image_corners_2d.append(data['p2'][i])
        image_corners_2d.append(data['p1'][i])

    image_corners_2d = np.array(image_corners_2d)

    # Camera intrinsic parameters and distortion coefficients (from parameters.
                                                txt)
    camera_matrix = np.array([[314.1779, 0, 199.4848],
                              [0, 314.2218, 113.7838],
                              [0, 0, 1]])
    dist_coeffs = np.array([-0.438607, 0.248625, 0.00072, -0.000476, -0.0911])

    # Define the 3D coordinates of the camera with respect to the IMU
    tvec_imu_camera = np.array([-0.04, 0.0, -0.03])  # Translation vector from
                                                IMU to camera

    # Define the rotation of the camera with respect to the IMU (assuming yaw =
                                                pi/4)
    yaw = np.pi / 4

    # Write a rotation matrix to rotate about x by 180 degrees and z by 45
                                                degrees
    rot_matrix_camera_imu = np.array([[1, 0, 0], [0, np.cos(np.pi), -np.sin(np.pi
                                                )], [0, np.sin(np.pi), np.cos(np.pi)
                                                ]]) @ np.array([[np.cos(np.pi/4), -
                                                np.sin(np.pi/4), 0], [np.sin(np.pi/4
```

```
                                                ), np.cos(np.pi/4), 0], [0, 0, 1]])

    # Solve the PnP problem
    success, rvec, tvec = cv2.solvePnP(map_corners_3d, image_corners_2d,
                                                camera_matrix, dist_coeffs)

    if not success:
        raise RuntimeError("PnP solver failed to converge")

    # Convert rotation vector to rotation matrix
    rot_matrix, _ = cv2.Rodrigues(rvec)

    #Reshape tvec_imu_camera to (3, 1)
    tvec_imu_camera = tvec_imu_camera.reshape(3, 1)

    tvec = -rot_matrix.T @ tvec + -rot_matrix.T@ tvec_imu_camera

    rot_matrix = rot_matrix.T

    final_rot_matrix = rot_matrix@rot_matrix_camera_imu.T

    # Extract Euler angles from rotation matrix
    roll = np.arctan2(final_rot_matrix[2, 1], final_rot_matrix[2, 2])
    pitch = np.arctan2(-final_rot_matrix[2, 0], np.sqrt(final_rot_matrix[2, 1]**2
                                                + final_rot_matrix[2, 2]**2))
    yaw = np.arctan2(final_rot_matrix[1, 0], final_rot_matrix[0, 0])

    #return tvec_imu_camera.flatten(), np.array([roll, pitch, yaw])
    return tvec.reshape(3,), np.array([roll, pitch, yaw])

def world_corners():
    tag_size = 0.152  # meters
    tag_spacing = 0.152  # meters
    special_spacing = 0.178  # meters (for columns 3-4 and 6-7)

    # Define the grid size
    num_rows = 9
    num_cols = 12

    # Initialize list to store tag coordinates
    tag_coordinates = []

    # Iterate over each tag ID
    for row in range(num_rows):
        for col in range(num_cols):
            tag_id = row * num_cols + col

            # Determine the top-left corner of the tag
            x = col * (tag_size + tag_spacing)
            y = row * (tag_size + tag_spacing)

            # Adjust for special spacing
            if row >= 3:
                y += special_spacing
            if row >= 7:
                y += special_spacing

            # Compute coordinates of the other corners
            corner1 = (x, y)
            corner2 = (x , y+tag_size)
            corner3 = (x + tag_size, y + tag_size)
            corner4 = (x+ tag_size, y )
```

```
            # Store tag ID and its coordinates
            tag_coordinates.append((tag_id, [corner1, corner2, corner3, corner4])
                                                    )

    return tag_coordinates
```

The program above does the following functions

1. **'world_corners'** function calculates the tag coordinates of all the april tags with all the specified distances between them as specified in the assignment.
2. **'estimate pose'** takes in the data and the tag coordinates and get the position and orientation from the 'solvePNP' function.
3. Uses the Rodriguez function to convert the orientation to a rotation matrix.
4. Perform the transformation from the camera to world to drone to world frame.
5. At the end of the function, return the position and orientation after all the transformations have been performed.
6. The plotting section is also embedded within this program itself where the estimated and ground truth positions as well as velocities have been plotted and stored in a new folder called **"observation_model_plots"**.

Here are the results of position tracking by the observation model given below in Page 6,7 and 8.

Task 3

Covariance Estimation

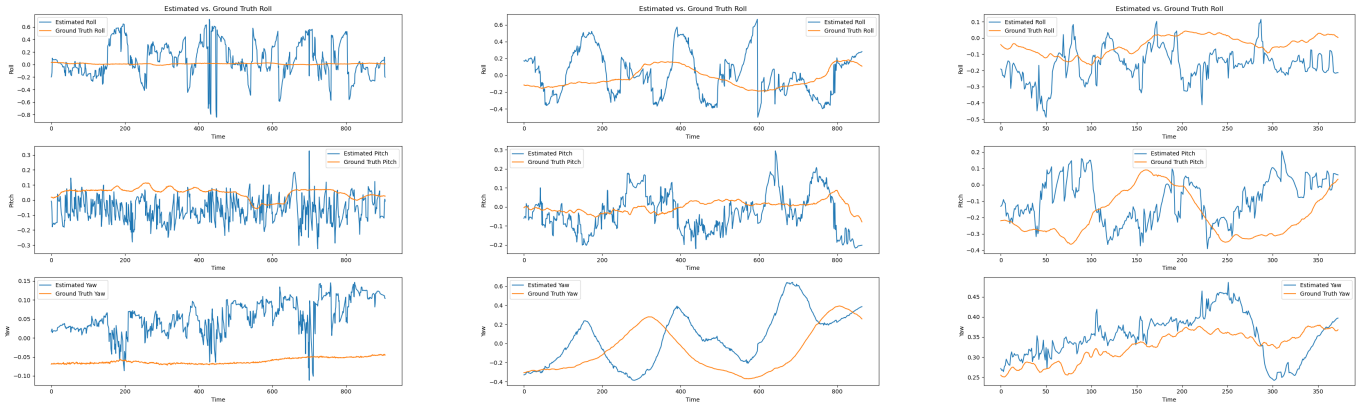
In this task, we calculate the covariance matrix for each dataset and at the end, we have to average all the covariance matrices to use as the **"R"** matrix for our filter implementation. For some datasets, the estimated positions which come bedore the first ground truth data pont. In this case, we simply skip that datapoint.

$$R = \frac{1}{N-1} \sum_{t=1}^{n} v_t v_t^T \tag{1}$$

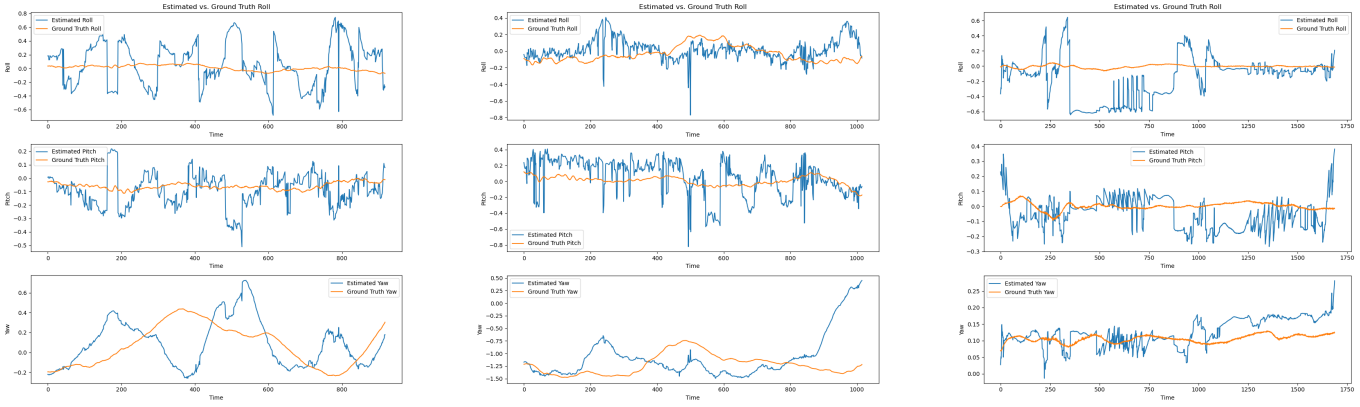
where R is our covariance matrix and N is the number of estimated samples.

The code for this task is given below

```
import numpy as np
import scipy.io
from extract_pose_final import estimate_pose, world_corners

# Function to load data from .mat file
def load_data(filename):
    return scipy.io.loadmat(filename, simplify_cells=True)

# Function to estimate observation model covariance
def estimate_covariances(data):

    # Initialize variables
    n = len(data['data'])  # Number of time steps
    R_sum = np.zeros((6, 6))  # Initialize sum of outer products

    tag_coordinates = world_corners()
```



(a) Dataset 1 (b) Dataset 2 (c) Dataset 3



(d) Dataset 4 (e) Dataset 5 (f) Dataset 6

(g) Dataset 7 (h) Dataset 8

Figure 1: Tracking orientations from observation model

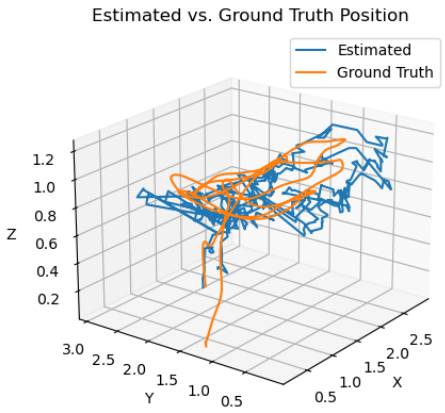


Figure 2: Dataset 0

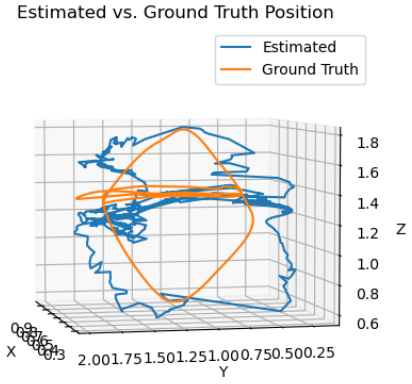


Figure 3: Dataset 1

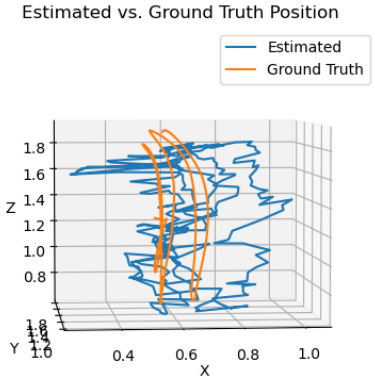


Figure 4: Dataset 2

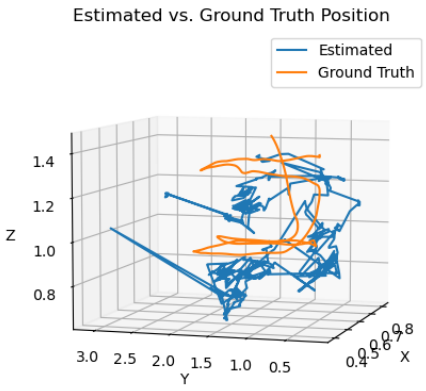


Figure 5: Dataset 3

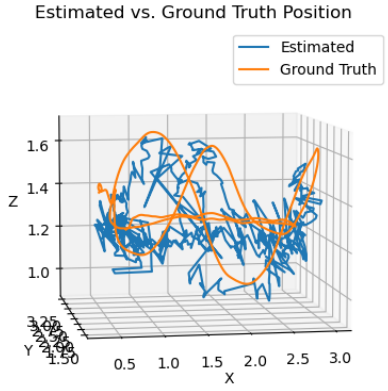


Figure 6: Dataset 4

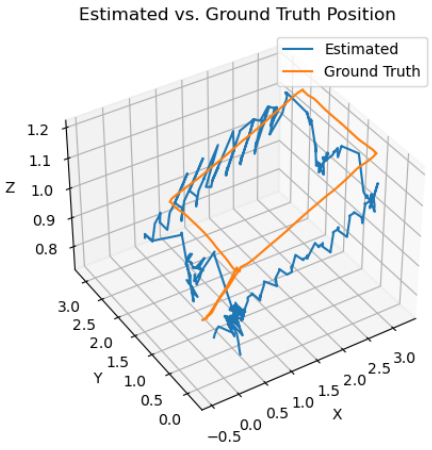


Figure 7: Dataset 5

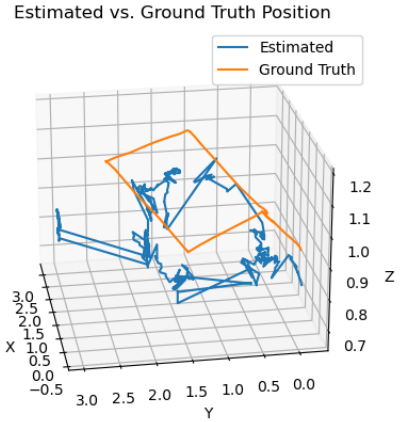


Figure 8: Dataset 6

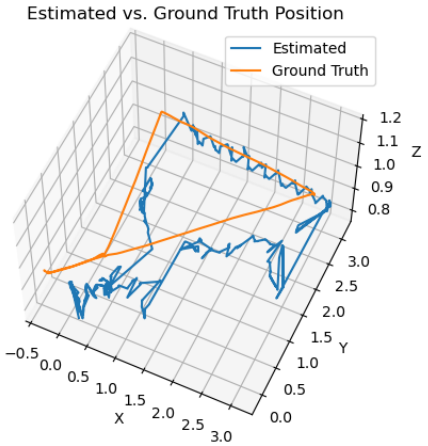


Figure 9: Dataset 7

```
    # Iterate over each time step
    for i in range(n):
        # Extract ground truth orientation data from vicon field
        position = data['vicon'][i][:3]
        orientation = data['vicon'][i][3:6]

        if len(data['data'][i]['id']) == 0:
            continue

        # Rearrange observation model equation to solve for noise term nu_t
        z = np.concatenate((position, orientation))
        z_hat = estimate_pose(data['data'][i], tag_coordinates)
        z_hat = np.array(z_hat)
        #Reshape z_hat to be (6,)
        z_hat = z_hat.reshape(6,)

        nu_t = z - z_hat

        # Compute outer product of nu_t
        R_sum += np.outer(nu_t, nu_t)

    # Compute sample covariance matrix
    R = R_sum / (n - 1)

    return R

def main():
    R_matrices = []
    #Loop through all the datasets
    for i in range(8):
        # Call the function with the filename of the .mat file containing the
                                                    data
        # Load data
        filename = 'data/studentdata{}.mat'.format(i)
        data = load_data(filename)

        #Loop through the data and print the tag IDs
        for i in range(len(data['data'])):
            # If the tag id is an integer, convert it to a list
            if isinstance(data['data'][i]['id'], int):
                data['data'][i]['id'] = [data['data'][i]['id']]

            # Check if p1, p2, p3, p4 are 1D and convert them to 2D if they are
            for point in ['p1', 'p2', 'p3', 'p4']:
                if len(data['data'][i][point].shape) == 1:
                    data['data'][i][point] = data['data'][i][point].reshape(1, -1
                                                                            )

        data['vicon'] = np.array(data['vicon'])
        #  Transpose it
        data['vicon'] = data['vicon'].T

        R = estimate_covariances(data)

        #Store all the R matrices in a list
        R_matrices.append(R)

    # Calculate the average of the R matrices
    R_avg = (1/(len(R_matrices) - 1))*np.sum(R_matrices, axis=0)

    print(R_avg)
```

```
if __name__ == "__main__":
    main()
```

The final averaged covariance matrix $\mathbf{R}$ is given as

$$\begin{bmatrix} 7.09701409e-03 & 2.66809900e-05 & 1.73906943e-03 & 4.49014777e-04 & 3.66195490e-03 & 8.76154421e-04 \\ 2.66809900e-05 & 4.70388499e-03 & -1.33432420e-03 & -3.46505064e-03 & 1.07454548e-03 & -1.69184839e-04 \\ 1.73906943e-03 & -1.33432420e-03 & 9.00885499e-03 & 1.80220246e-03 & 3.27846190e-03 & -1.11786368e-03 \\ 4.49014777e-04 & -3.46505064e-03 & 1.80220246e-03 & 5.27060654e-03 & 1.01361187e-03 & -5.86487142e-04 \\ 3.66195490e-03 & 1.07454548e-03 & 3.27846190e-03 & 1.01361187e-03 & 7.24994152e-03 & -1.36454993e-03 \\ 8.76154421e-04 & -1.69184839e-04 & -1.11786368e-03 & -5.86487142e-04 & -1.36454993e-03 & 1.21162646e-03 \end{bmatrix} \tag{2}$$

Task 4

Extended Kalman Filter

Process Model

In order to track the pose of the quadrotor, the Extended Kalman Filter has been implemented.

$$x = \begin{bmatrix} p \\ q \\ \dot{p} \\ b_g \\ b_a \end{bmatrix} \tag{3}$$

where the rotation matrix is defined as

$$R_q = \begin{bmatrix} \cos(q3)\cos(q2) - \sin(q1)\sin(q2)\sin(q3) & -\cos(q1)\sin(q3) & \cos(q3)\sin(q2) + \cos(q2)\sin(q1)\sin(q3) \\ \cos(q3)\sin(q1)\sin(q2) + \cos(q2)\sin(q3) & \cos(q1)\cos(q3) & \sin(q3)\sin(q2) - \cos(q3)\cos(q2)\sin(q1) \\ -\cos(q1)\sin(q2) & \sin(q1) & \cos(q1)\cos(q2) \end{bmatrix} \tag{4}$$

where $q = [\phi\, \theta\, \psi]$

Let us start by deriving the equations for process model first. According to the equation given in the assignment,

$$\dot{x} = f(x, u) = \begin{bmatrix} p_{\text{dot}} \\ G_q^{-1} \cdot u_w \\ g + R_q \cdot u_a \\ 0 \\ 0 \end{bmatrix} \tag{5}$$

where the control inputs to our model is given by

$$u = \begin{bmatrix} wx \\ wy \\ wz \\ vx \\ vy \\ vz \end{bmatrix} \tag{6}$$

and $G(q)$ is defined as

$$G_q = \begin{bmatrix} \cos(q_2) & 0 & -\cos(q_1)\sin(q_2) \\ 0 & 1 & \sin(q_1) \\ \sin(q_2) & 0 & \cos(q_1)\cos(q_2) \end{bmatrix} \tag{7}$$

In order to write our function $f(x, u)$ i.e only in terms of x and u, we use the **sympy** library in python. After defining the variables in symbolic notation using **sympy**, we get the F function which is added to the previous state multiplied by dt, by simple Euler integration, we propagate the current state to find the next state.

$$X_{n+1} = X_n + f(x, u) \cdot \Delta t \tag{8}$$

$$X_{n+1} = F(x, u) \cdot \Delta t \tag{9}$$

In order to implement the Extended Kalman Filter, we need to take the jacobian of this function F, which is shown in the code below.

```
    def calculate_symbolic(self):
        dt = sp.symbols('dt')

        p1, p2, p3, q1, q2, q3, p_dot1, p_dot2, p_dot3, bg1, bg2, bg3, ba1, ba2,
                                                    ba3 = sp.symbols('p1 p2 p3 q1 q2
                                                     q3 p_dot1 p_dot2 p_dot3 bg1 bg2
                                                     bg3 ba1 ba2 ba3')

        # Define the matrix elements
        G_q = sp.Matrix([
            [sp.cos(q2), 0, -sp.cos(q1)*sp.sin(q2)],
            [0, 1, sp.sin(q1)],
            [sp.sin(q2), 0, sp.cos(q1)*sp.cos(q2)]
        ])

        # Compute the inverse of G_q
        G_q_inv = G_q.inv()

        # Write R_q as a 3x3 matrix just like G_q
        R_q = sp.Matrix([
                        [ sp.cos(q3)*sp.cos(q2) - sp.sin(q1)*sp.sin(q2)*sp.sin(q3
                                                                ), -sp.cos(q1)*
                                                                sp.sin(q3), sp.
                                                                cos(q3)*sp.sin(
                                                                q2) + sp.cos(q2)
                                                                *sp.sin(q1)*sp.
                                                                sin(q3)],
                        [ sp.cos(q3)*sp.sin(q1)*sp.sin(q2) + sp.cos(q2)*sp.sin(q3
                                                                ), sp.cos(q1)*sp
                                                                .cos(q3), sp.sin
                                                                (q3)*sp.sin(q2)
                                                                - sp.cos(q3)*sp.
                                                                cos(q2)*sp.sin(
                                                                q1)],
                        [ -sp.cos(q1)*sp.sin(q2), sp.sin(q1), sp.cos(q1)*sp.cos(
                                                                q2)]
                        ])

        # Define the state vector x = [p, q, p_dot, bg, ba]
```

```
        x = sp.Matrix([p1, p2, p3, q1, q2, q3, p_dot1, p_dot2, p_dot3, bg1, bg2,
                                                bg3, ba1, ba2, ba3])

        # Create a new matrix including only p_dot1, p_dot2, p_dot3
        p_dot = sp.Matrix([p_dot1, p_dot2, p_dot3])

        # Define the input vector u = [wx, wy, wz, vx, vy, vz]
        wx, wy, wz, vx, vy, vz = sp.symbols('wx wy wz vx vy vz')
        u = sp.Matrix([wx, wy, wz, vx, vy, vz])

        uw = sp.Matrix([wx, wy, wz])
        ua = sp.Matrix([vx, vy, vz])

        # Define the gravity vector
        g = sp.Matrix([0, 0, -9.81])

        nbg = sp.Matrix([0, 0, 0])
        nba = sp.Matrix([0, 0, 0])

        # Define the x_dot equation x_dot = f(x, u) = [p_dot, G_q_inv * u, g +
                                                R_q * u, 0, 0]
        x_dot = sp.Matrix([p_dot, G_q_inv * uw, g + R_q * ua, nbg, nba])

        F = x + x_dot*dt

        # Compute the Jacobian of the process model
        Jacobian_J = F.jacobian(x)

        return F, Jacobian_J

    def process_model(self, x, delta_t, u):
        # Process model function
        dt = sp.symbols('dt')

        p1, p2, p3, q1, q2, q3, p_dot1, p_dot2, p_dot3, bg1, bg2, bg3, ba1, ba2,
                                                ba3 = sp.symbols('p1 p2 p3 q1 q2
                                                 q3 p_dot1 p_dot2 p_dot3 bg1 bg2
                                                 bg3 ba1 ba2 ba3')

        wx, wy, wz, vx, vy, vz = sp.symbols('wx wy wz vx vy vz')

        F,_ = self.calculate_symbolic()

        # Substitue the values of x and u into F
        F = F.subs({p1: x[0], p2: x[1], p3: x[2], q1: x[3], q2: x[4], q3: x[5],
                                                p_dot1: x[6], p_dot2: x[7],
                                                p_dot3: x[8], bg1: x[9], bg2: x[
                                                10], bg3: x[11], ba1: x[12], ba2
                                                : x[13], ba3: x[14], wx: u[0],
                                                wy: u[1], wz: u[2], vx: u[3], vy
                                                : u[4], vz: u[5], dt: delta_t})

        # Convert F to a numpy array of shape (15,)
        F_np = np.array(F)

        # Convert F to a numpy array of shape (15,)
        F_np = F_np.reshape(15,)

        # Return the process model
        return F_np

    def compute_process_model_jacobian(self, x, delta_t, u):
```

```
        dt = sp.symbols('dt')

        p1, p2, p3, q1, q2, q3, p_dot1, p_dot2, p_dot3, bg1, bg2, bg3, ba1, ba2,
                                                ba3 = sp.symbols('p1 p2 p3 q1 q2
                                                 q3 p_dot1 p_dot2 p_dot3 bg1 bg2
                                                 bg3 ba1 ba2 ba3')

        wx, wy, wz, vx, vy, vz = sp.symbols('wx wy wz vx vy vz')

        _, Jacobian_J = self.calculate_symbolic()

        # Substitute the values of x and u into Jacobian_J
        Jacobian_J = Jacobian_J.subs({p1: x[0], p2: x[1], p3: x[2], q1: x[3], q2:
                                                 x[4], q3: x[5], p_dot1: x[6],
                                                p_dot2: x[7], p_dot3: x[8], bg1:
                                                 x[9], bg2: x[10], bg3: x[11],
                                                ba1: x[12], ba2: x[13], ba3: x[
                                                14], wx: u[0], wy: u[1], wz: u[2
                                                ], vx: u[3], vy: u[4], vz: u[5],
                                                 dt: delta_t})

        # Convert Jacobian_J to a numpy array
        Jacobian_J_np = np.array(Jacobian_J)

        return Jacobian_J_np
```

Once, the process model is complete, we move to the observation model

Observation Model

Our observation model is relatively simple. We will be using the PnP based computer vision-based technique that measures the position and orientation: $\mathbf{z} = [\mathbf{p}\ \mathbf{q}] + \boldsymbol{\nu}$. We can write the state space equations as follows,

$$\mathbf{z} = \begin{bmatrix} \mathbf{I} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{I} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \\ \dot{\mathbf{p}} \\ \mathbf{b}_g \\ \mathbf{b}_a \end{bmatrix} + \mathcal{N}(0, \mathbf{R}) \tag{10}$$

And the code snippet is as follows

```
    def compute_observation_model_jacobian(self):
        # Compute the Jacobian of the observation model
        H = np.zeros((6, 15))
        H[:3, :3] = np.eye(3)
        H[3:, 3:6] = np.eye(3)
        return H
```

Since the observation model is linear, we need not compute the jacobians here for implementing EKF.

Putting it all together, we have the code snippet for EKF as shown below.

```
def predict(self, x, P, dt, u):
        # Predict step
        F = self.compute_process_model_jacobian(x, dt, u)
        x_pred = self.process_model(x, dt, u)
        P_pred = F @ P @ F.T + self.Q
        return x_pred, P_pred
```

```
    def update(self, x_pred, P_pred, z, estimated_pose):
        # Update step
        H = self.compute_observation_model_jacobian()
        y = estimated_pose - H @ x_pred
        S = H @ P_pred @ H.T + self.R
        S_float = S.astype(np.float64)
        K = P_pred @ H.T @ np.linalg.inv(S_float)
        x_updated = x_pred + K @ y
        P_updated = (np.eye(len(x_pred)) - K @ H) @ P_pred
        return x_updated, P_updated
```

Finally, we get the results of tracking the pose for all the datasets, which is as follows,

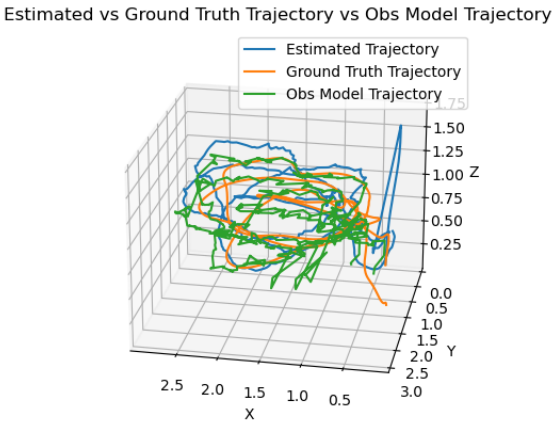


Figure 10: Dataset 0

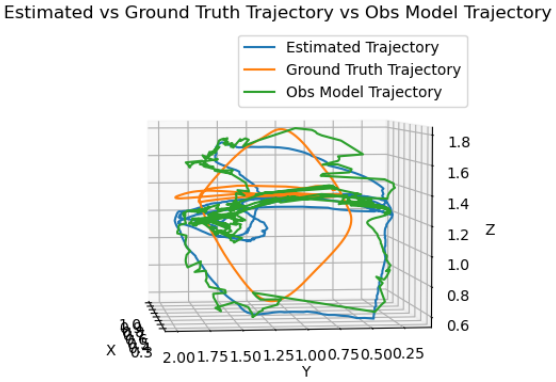


Figure 11: Dataset 1

In order to run the code, type in the following command in the terminal

```
python3 ekf.py {provide the dataset number as an argument to the script}
```

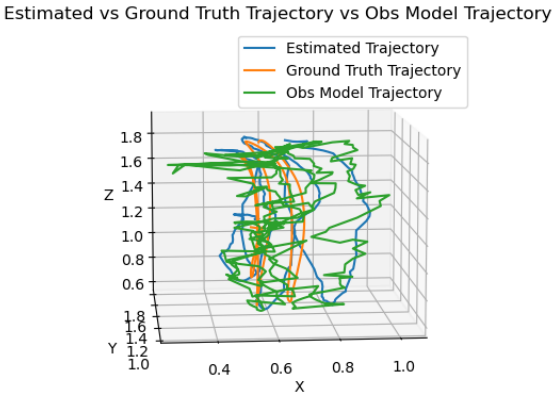


Figure 12: Dataset 2

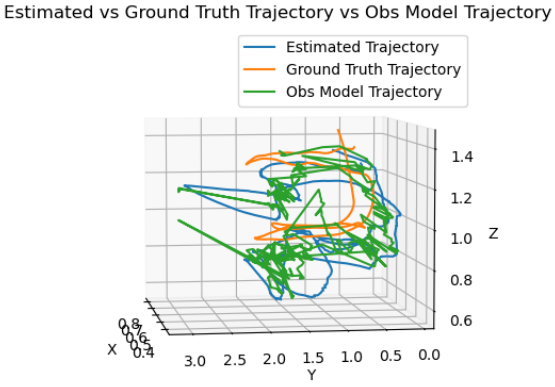


Figure 13: Dataset 3

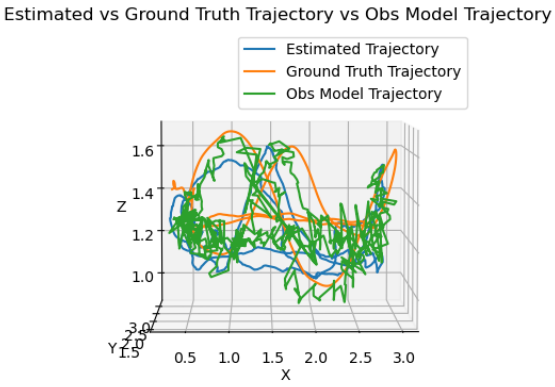


Figure 14: Dataset 4

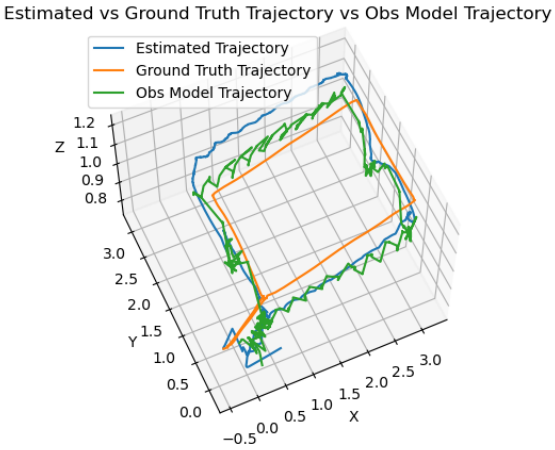


Figure 15: Dataset 5

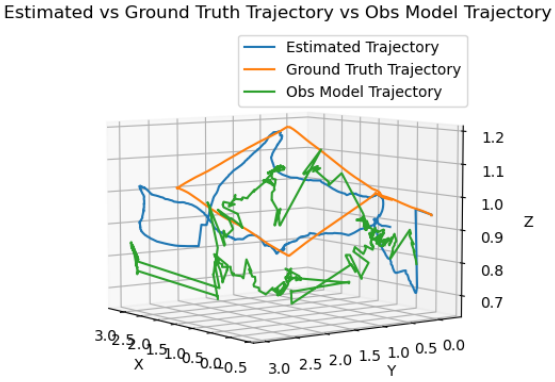


Figure 16: Dataset 6

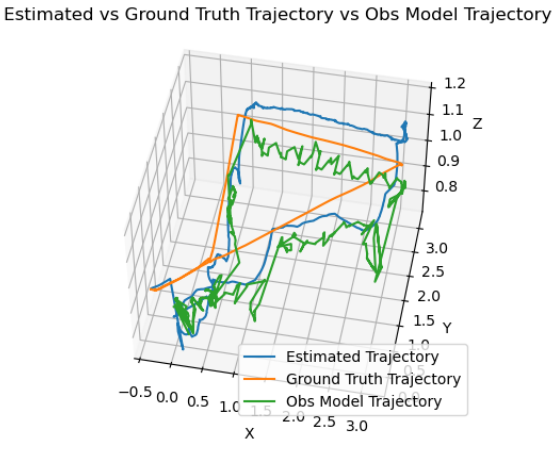


Figure 17: Dataset 7