Giorgio Mendoza

RBE595-S24-S01

Dr. Brodovsky

Nonlinear Kalman Filter Report

Task 1: Pose Estimation

Objective: Task 1 involves the use of a fifteen-state model covering the drone's linear position, orientation, linear velocity, gyroscopic bias, and accelerometer bias. A significant part of the task is estimating the pose through observed tag markers and transforming camera-based pose estimates to the drone frame for comparison against motion capture data. The pose estimation function needs to output the estimated position and orientation in the form of Euler angles (roll, pitch, yaw) from the observation data. The process involves utilizing camera calibration data to transform tag corners in the image frame to the world frame and computing the drone's measured pose from this data.

Explanation: In Task 1, the code is designed to calculate the pose of a drone based on visual data of AprilTags. These tags are unique patterns that a computer can easily recognize and use to determine its location in 3D space. The code employs the solvePnP function from the OpenCV library, which takes the 3D points (AprilTags in the environment) and their 2D projections in an image to estimate the drone's position and orientation. The LayoutMap class contains the actual positions of the tags in the world, while getPose computes the drone's pose by considering the observed tags within an image.
The process involves constructing arrays of world points and image points from observed tags, which are then fed into solvePnP. Upon solving the Perspective-n-Point problem, the rotation and translation vectors are used to generate a transformation matrix that maps the camera's view to the drone's position in the world. The code then converts this matrix into Euler angles for an interpretable description of orientation.

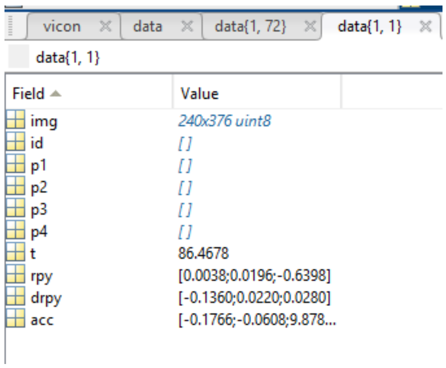
```
    def getPose(self, obsTags: List[TagLayout]) -> Tuple[np.ndarray, np.ndarray]:
        #calculates pose of drone based on observed April Tag markers
        #Parameters: obsTags (List[TagLayout]): List of April Tag objects observed in  image
        #Returns: tuple containing orientation and position of  drone
        # convert tag corner locations from observed tags to a flat array for PnP solving
        worldPts = np.array([
            getattr(self.tagPositions[tag.tag_id], corner)
            for tag in obsTags
            for corner in ['bottomLeftCorner', 'bottomRightCorner', 'topRightCorner',
'topLeftCorner']
            ], dtype=np.float32).reshape(-1, 3)

        # convert image points of tags to a flat array for PnP solving
        imgPts = np.array([
            getattr(tag, corner)
```

```
            for tag in obsTags
            for corner in ['bottomLeftCorner', 'bottomRightCorner', 'topRightCorner',
'topLeftCorner']
        ], dtype=np.float32).reshape(-1, 2)

        # ensure there are enough points to perform pose estimation
        if worldPts.shape[0] < 4 or imgPts.shape[0] < 4:
            raise ValueError("Not enough points to estimate pose.")

        # solve PnP problem to obtain rotation and translation vectors
        success, rotVector, getVector = solvePnP(worldPts, imgPts, self.camMat, self.disCoeff,
flags=cv2.SOLVEPNP_ITERATIVE)

        if not success:
            raise ValueError("solvePnP failed to find a solution")

        # convert rotation vector to a rotation matrix
        rotMat, _ = Rodrigues(rotVector)
        # reshape translation vector for consistency
        getVector = getVector.reshape(-1, 1)
        # combine rotation matrix and translation vector into a camera-to-world transform matrix
        camWorldTransf = np.hstack((rotMat, getVector))
        # add fourth row to make it a homogeneous transformation matrix
        camWorldTransf = np.vstack((camWorldTransf, [0, 0, 0, 1]))
        # define rotation matrices for adjusting coordinate system from camera to drone frame
        rotZ = 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]])
        rotX = np.array([[1, 0, 0], [0, -1, 0], [0, 0, -1]])
        getRot = np.dot(rotX, rotZ)
        # add fourth row to make it a homogeneous transformation matrix
        getRot = np.vstack((getRot, [0, 0, 0]))
        # define translation offset for camera in drone frame
        camOffset = np.array([-0.04, 0, -0.03, 1]).reshape(4, 1)
        # combine rotation and translation to create camera-to-drone frame transformation
        camDroneFrame = np.hstack((getRot, camOffset))
        # calculate world-to-drone transformation matrix
        worldDroneTransf = np.dot(np.linalg.inv(camWorldTransf), camDroneFrame)
        # extract position vector
        extractPos = worldDroneTransf[:3, -1]
        # convert rotation matrix to Euler angles
        extractOr = rotMatToEuler(worldDroneTransf[:3, :3])
        # return orientation and position of drone
        return extractOr, extractPos
```

The values for Position and Orientation have this format after estimating the pose.

```
Position: [0.3315789  1.33538624 0.44864374], Orientation: (-0.18160949255421371, -0.019475368845672636, 0.018836885894178743)
Position: [0.31754912 1.32861973 0.50385223], Orientation: (-0.1282684793048702, -0.07288413627614188, 0.02320130500798023)
Position: [0.21474194 1.20562015 0.60208907], Orientation: (0.10170705137230293, -0.17929405738675133, 0.013751108714859907)
Position: [0.21424865 1.20922841 0.6381667 ], Orientation: (0.08007160889296112, -0.16503958095395516, 0.016992757976240832)
Position: [0.21439919 1.20828188 0.64971977], Orientation: (0.07934028538008606, -0.16173009609264755, 0.017392534029229387)
```

It was also important to open the contents of the .mat files before implementing anything to see how the format of the data was stored in them. This also showed that some of the initial data points were incomplete so these points were skipped to avoid issues with the PnP function.



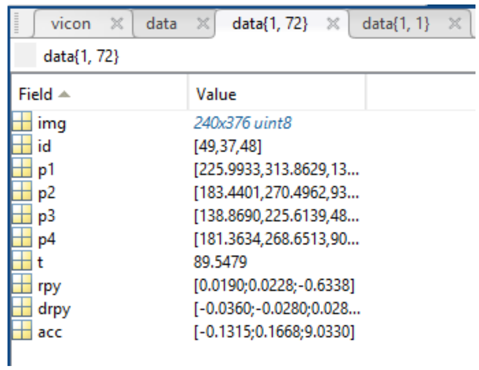


Figure 1, 2. Mat file contents

```
Not enough points for solvePnP. Only 4 object points available.
Timestamp: 88.54824829101562
Pose estimation failed for this data point.
---
Not enough points for solvePnP. Only 4 object points available.
Timestamp: 88.57861328125
Pose estimation failed for this data point.
---
```

Figure 3. PnP debugging

Task 2: Visualization

Objective: Task 2 requires visualizing the drone's flight path and orientation by comparing the estimated and actual data. It includes plotting the 3D trajectory and angles of roll, pitch, and yaw. Issues in model implementation should be noted in the report. Below are the plots for each dataset.

Explanation: For Task 2, the code visualizes the trajectory of the drone. It uses the pose estimates from Task 1, plotting them to compare the calculated positions against the ground truth over time. The get3DPlots function is responsible for generating the visual output, creating a 3D scatter plot to display the flight path. It takes in pairs of labels and corresponding data points, with colors indicating different heights (z-axis values) for clarity.

The code processes the dataset, extracting and plotting positions and orientations over time. It compares estimated trajectories and actual paths, alongside roll, pitch, and yaw comparisons. These visualizations can validate the pose estimation's accuracy and the performance of implemented models against the drone's actual movements.

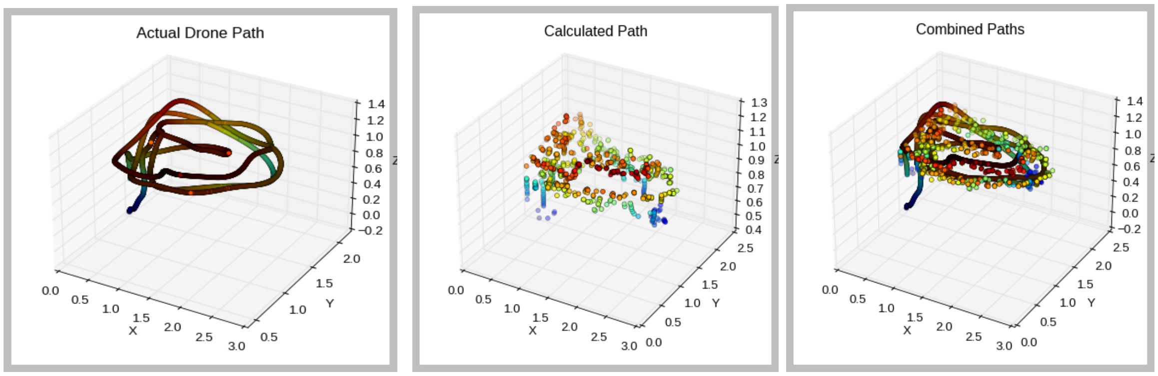


Figure 4, 5, 6: Actual and Estimated Path Plots for Dataset 0

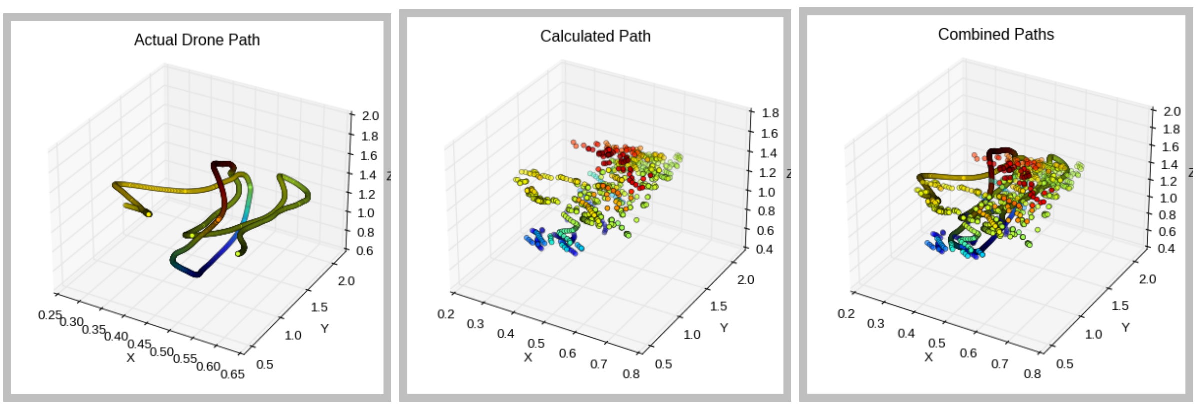


Figure 7, 8, 9: Actual and Estimated Path Plots for Dataset 1

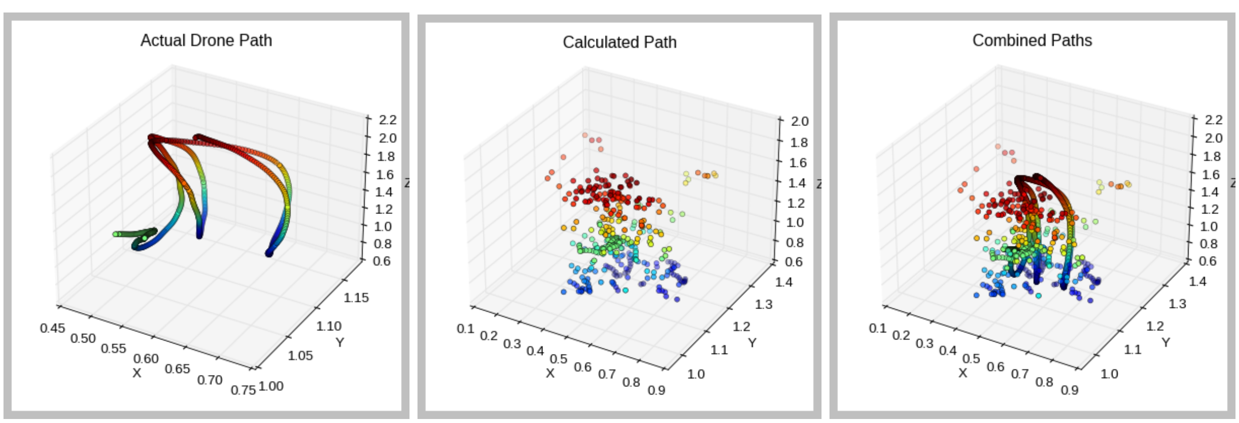


Figure 10, 11, 12: Actual and Estimated Path Plots for Dataset 2

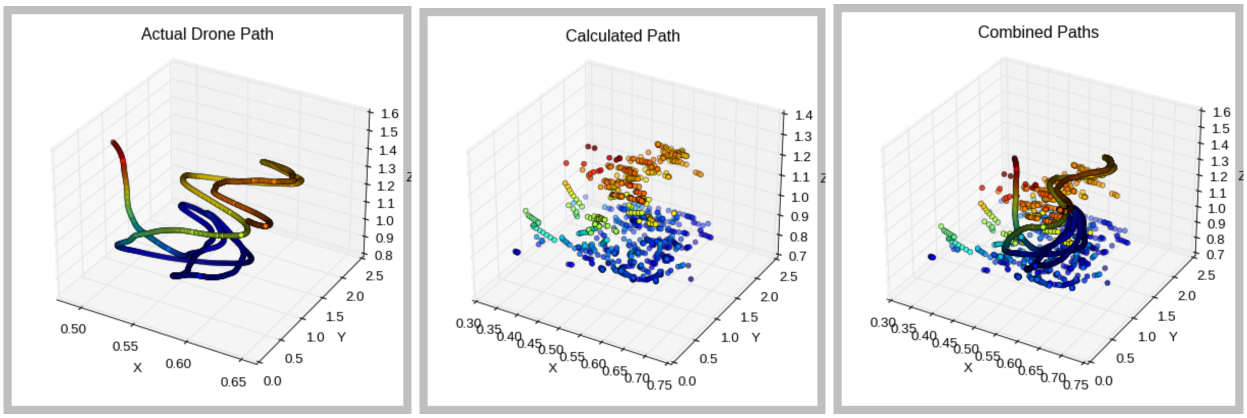


Figure 13, 14, 15: Actual and Estimated Path Plots for Dataset 3

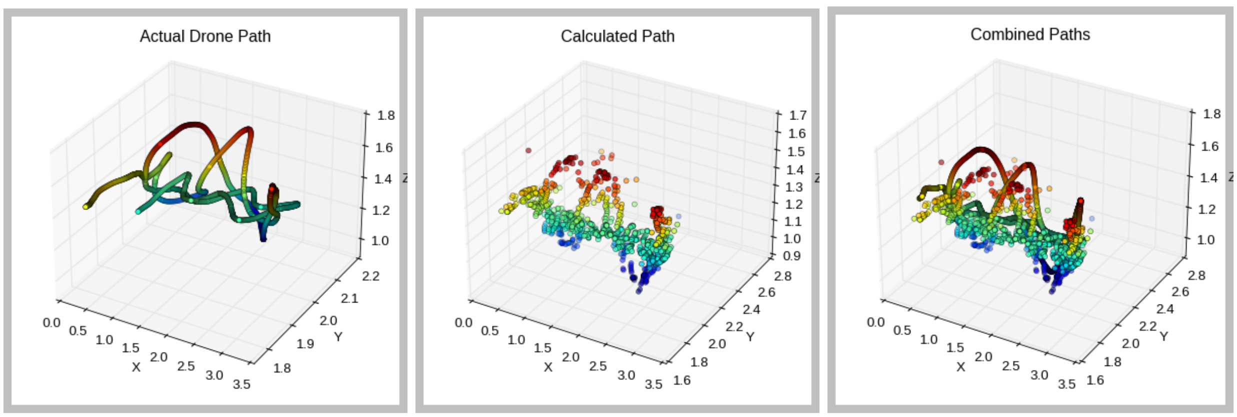


Figure 16, 17, 18: Actual and Estimated Path Plots for Dataset 4

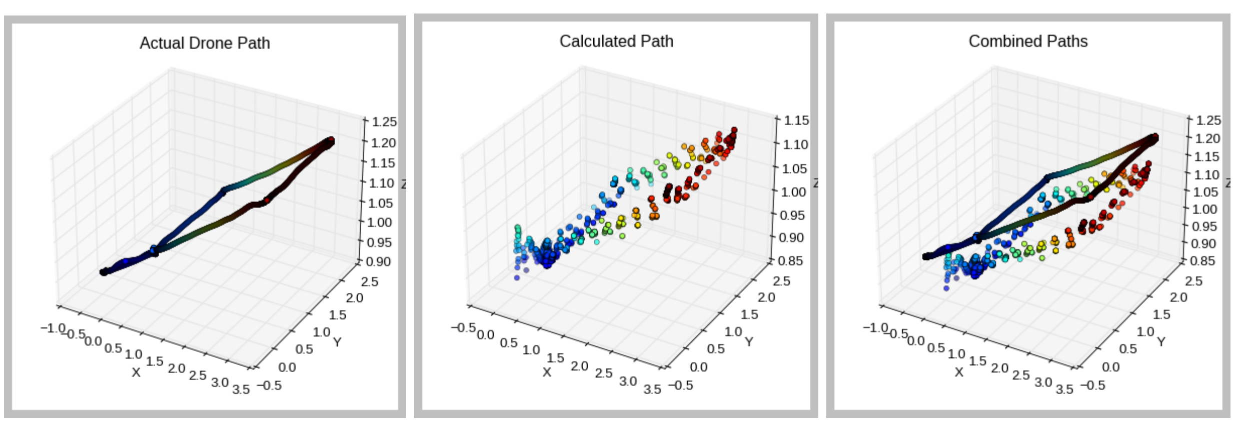


Figure 19, 20, 21: Actual and Estimated Path Plots for Dataset 5

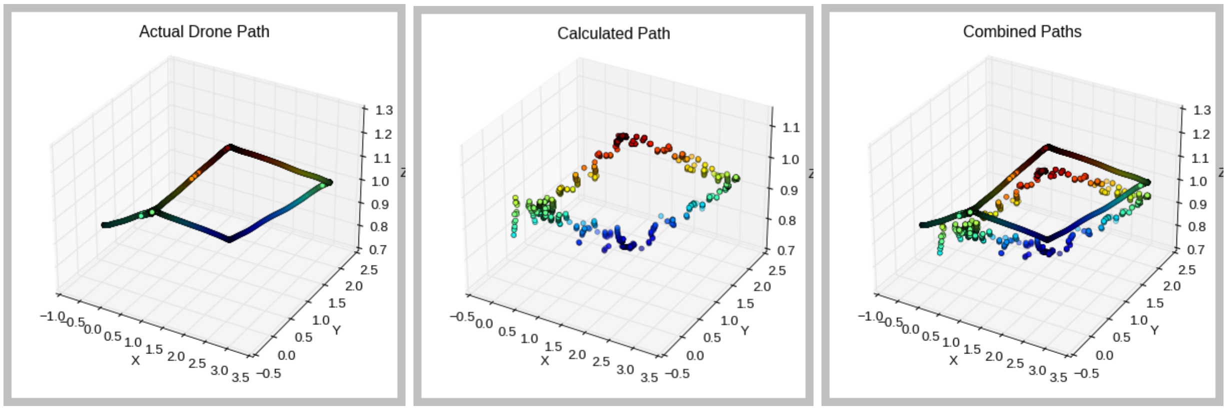


Figure 22, 23, 24: Actual and Estimated Path Plots for Dataset 6

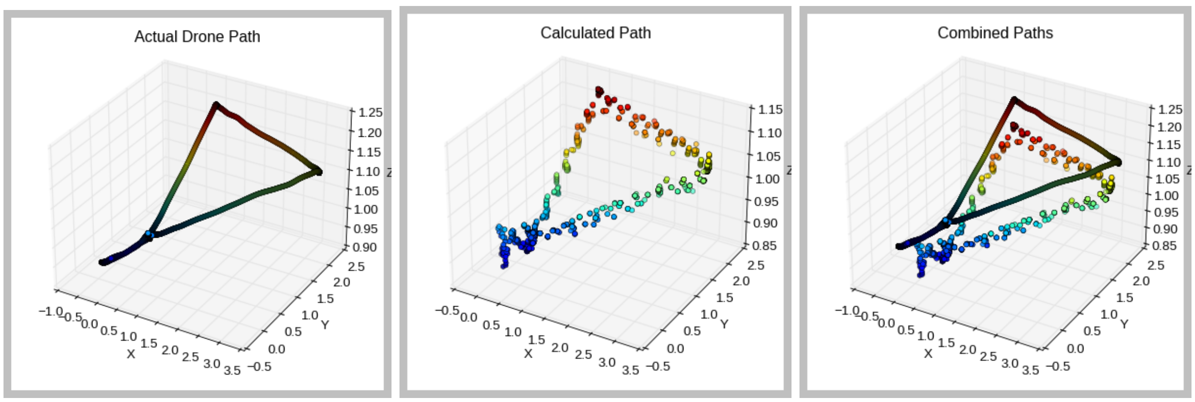


Figure 25, 26, 27: Actual and Estimated Path Plots for Dataset 7

The Yaw, Pitch and Roll can also be plotted for each .mat file, but they were omitted for brevity.

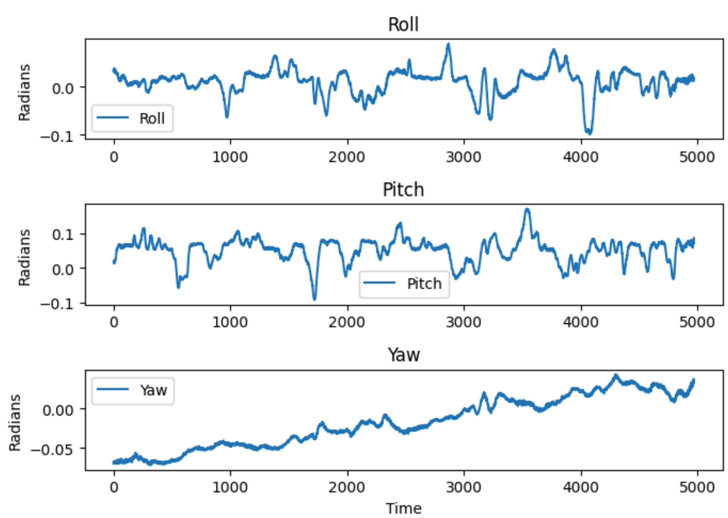


Figure 28: Roll, Pitch, Yaw plot for .mat file 0

Task 3: Covariance Estimation

Objective: Task 3 involves calculating the covariance matrix for the drone's sensor data. It uses true data to correct the observation model and find the noise over time. The output is a function that takes a data file and outputs a 6x6 matrix estimating sensor noise.

Explanation: The code implements estimateCovariance, which calculates the covariance matrix by comparing the estimated drone positions and orientations against the ground truth data. The estimation uses the genEstimatedPose function to generate an interpolated pose of the drone for moments between ground truth timestamps. This interpolation ensures that the estimated state can be directly compared to the actual state, even when the timestamps do not align perfectly.

The estimateCovariance function iterates over the dataset, skipping any data points that precede the first ground truth timestamp. For each valid data point, it extracts the corresponding estimated state (combination of position and orientation) and calculates the difference from the interpolated ground truth state, labeled as stateError. This error is then squared and summed up in errorSum to form part of the covariance computation.

To avoid skewing results with insufficient data, the code checks that there are more than one valid sample before calculating the average covariance (avgCov). If not enough valid comparisons exist, it raises an error, indicating the data is insufficient to estimate the covariance matrix.

Finally, the script prints the calculated covariance matrix to see if the estimated states match the actual states across the dataset. This matrix is important for tuning the Kalman filter in later tasks by giving insight into the accuracy and reliability of the observation model.

Here are the covariance matrices for each mat file:

For studentdata0.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata0.mat
[[ 0.01124641 -0.00167305 -0.00189018  0.00532129 -0.00396055 -0.00028349]
 [-0.00167305  0.00544109  0.0002523  -0.00102443  0.00326998 -0.00467435]
 [-0.00189018  0.0002523   0.00119224 -0.00165101  0.00271911  0.00011026]
 [ 0.00532129 -0.00102443 -0.00165101  0.00633645 -0.00593225 -0.00243686]
 [-0.00396055  0.00326998  0.00271911 -0.00593225  0.0146031  -0.00101018]
 [-0.00028349 -0.00467435  0.00011026 -0.00243686 -0.00101018  0.00723985]]
----------------------------------------------------------------------
```

For studentdata1.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata1.mat
[[ 0.00401278  0.00118289 -0.00200353  0.00403816  0.00258337 -0.00409603]
```

```
 [ 0.00118289  0.00543968 -0.00466981  0.00377619  0.00056948 -0.00704299]
 [-0.00200353 -0.00466981  0.01242909 -0.01395551 -0.00031346  0.0148666 ]
 [ 0.00403816  0.00377619 -0.01395551  0.080404    0.00471396 -0.07993726]
 [ 0.00258337  0.00056948 -0.00031346  0.00471396  0.00263729 -0.00492038]
 [-0.00409603 -0.00704299  0.0148666  -0.07993726 -0.00492038  0.08305685]]
----------------------------------------------------------------------
```

For studentdata2.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata2.mat
[[ 0.0048641  -0.00029412  0.00232463 -0.0044844   0.00336135  0.00614059]
 [-0.00029412  0.00375938  0.00030682 -0.00026312  0.00145409 -0.00189756]
 [ 0.00232463  0.00030682  0.01589493 -0.04509652  0.00344333  0.04714136]
 [-0.0044844  -0.00026312 -0.04509652  0.146573   -0.00847539 -0.15190955]
 [ 0.00336135  0.00145409  0.00344333 -0.00847539  0.0038307   0.00888352]
 [ 0.00614059 -0.00189756  0.04714136 -0.15190955  0.00888352  0.15991489]]
----------------------------------------------------------------------
```

For studentdata3.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata3.mat
[[ 0.00352548 -0.00047262  0.0030703   0.00183726  0.00360395 -0.00173159]
 [-0.00047262  0.00388425 -0.00196857 -0.00176049  0.00013449 -0.00121067]
 [ 0.0030703  -0.00196857  0.00963698 -0.01067442  0.00380695  0.01110665]
 [ 0.00183726 -0.00176049 -0.01067442  0.08239308  0.00015441 -0.08168099]
 [ 0.00360395  0.00013449  0.00380695  0.00015441  0.00445274 -0.00048898]
 [-0.00173159 -0.00121067  0.01110665 -0.08168099 -0.00048898  0.08428838]]
----------------------------------------------------------------------
```

For studentdata4.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata4.mat
[[ 5.80720538e-03  1.14210113e-03 -3.32687446e-03 -2.78683743e-02
  -2.17933268e-03  2.37843330e-02]
 [ 1.14210113e-03  3.76247753e-03 -1.65579075e-03 -9.81812218e-03
  -2.27931920e-03  8.66305962e-03]
 [-3.32687446e-03 -1.65579075e-03  1.20429799e-02  1.09017647e-01
   3.10059542e-03 -1.11745292e-01]
 [-2.78683743e-02 -9.81812218e-03  1.09017647e-01  1.29411442e+00
   3.07915878e-02 -1.32276021e+00]
 [-2.17933268e-03 -2.27931920e-03  3.10059542e-03  3.07915878e-02
   5.44454261e-03 -2.91150851e-02]
 [ 2.37843330e-02  8.66305962e-03 -1.11745292e-01 -1.32276021e+00
  -2.91150851e-02  1.36097237e+00]]
----------------------------------------------------------------------
```

For studentdata5.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata5.mat
[[ 0.01188283 -0.00015715  0.00684363 -0.01140924  0.01105843  0.01342729]
 [-0.00015715  0.00511853 -0.00070073  0.001972    0.00100476 -0.00635604]
 [ 0.00684363 -0.00070073  0.00722309 -0.01219503  0.00604826  0.01362796]
 [-0.01140924  0.001972   -0.01219503  0.02250256 -0.00969573 -0.02519427]
 [ 0.01105843  0.00100476  0.00604826 -0.00969573  0.01066001  0.01059613]
```

```
 [ 0.01342729 -0.00635604  0.01362796 -0.02519427  0.01059613  0.03233176]]
----------------------------------------------------------------------
```

For studentdata6.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata6.mat
[[ 0.00756081  0.00025849  0.00426805 -0.0132951   0.00759897  0.0155587 ]
 [ 0.00025849  0.00538572 -0.00046634  0.00128277  0.00213525 -0.00643307]
 [ 0.00426805 -0.00046634  0.00633646 -0.01918056  0.00393969  0.02061461]
 [-0.0132951   0.00128277 -0.01918056  0.06215004 -0.01260445 -0.06608203]
 [ 0.00759897  0.00213525  0.00393969 -0.01260445  0.0084574   0.01307272]
 [ 0.0155587  -0.00643307  0.02061461 -0.06608203  0.01307272  0.07623915]]
----------------------------------------------------------------------
```

For studentdata7.mat:

```
----------------------------------------------------------------------
Print results: /content/drive/MyDrive/Colab Notebooks/studentdata7.mat
[[ 0.00787653  0.00022689  0.00462656 -0.01589929  0.0069185   0.01927074]
 [ 0.00022689  0.00483996 -0.00177254  0.00660006  0.00240733 -0.01052596]
 [ 0.00462656 -0.00177254  0.00731506 -0.02696362  0.00315189  0.02975539]
 [-0.01589929  0.00660006 -0.02696362  0.10321539 -0.01038709 -0.11252002]
 [ 0.0069185   0.00240733  0.00315189 -0.01038709  0.00713616  0.01154351]
 [ 0.01927074 -0.01052596  0.02975539 -0.11252002  0.01154351  0.1271331 ]]
----------------------------------------------------------------------
```

Task 4: Nonlinear Kalman Filter

Objective: Task 4 focuses on using a Kalman filter to track a drone's movement, adjusting guesses based on previous tasks to improve precision. It involves comparing real movement data with our estimates and visualizing this in 3D plots. The aim is to fine-tune our method for higher accuracy, confirmed by calculating the average estimation error.

Explanation: In Task 4, the code undertakes the implementation of an Unscented Kalman Filter to refine the estimation of a quadrotor drone's trajectory. It initializes the UKF with a set of parameters and a previously computed covariance matrix, which is the starting point for the observation model.
The UKF employs sigma points to capture the state distribution, predicting the subsequent state based on the drone's dynamics and adjusting it with the incoming measurements. This process iteratively refines the state estimate to align closely with the actual trajectory.
The code includes functions to adjust the covariance matrix, ensuring mathematical correctness, and to map state vectors back to measurements. The UKF runs across the dataset, filtering the pose estimations to produce a smoother and more accurate trajectory.

Additionally, the code calculates the RMSE to quantify the performance of the filter, comparing estimated poses to ground truth data. This error metric, alongside the visual comparisons from the 3D plots and orientation charts can be used to evaluate the filter's effectiveness in real-time pose tracking.

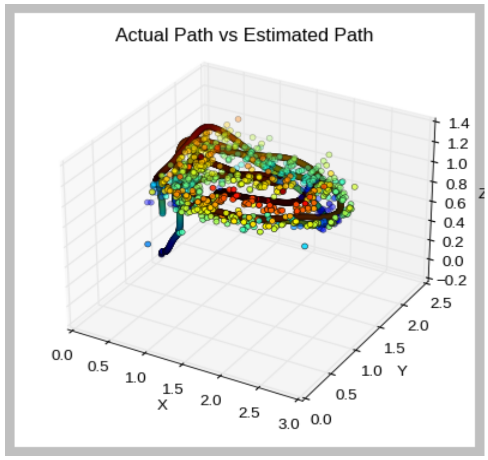


Figure 29: Actual and Filtered Path Plots for .Mat File 0

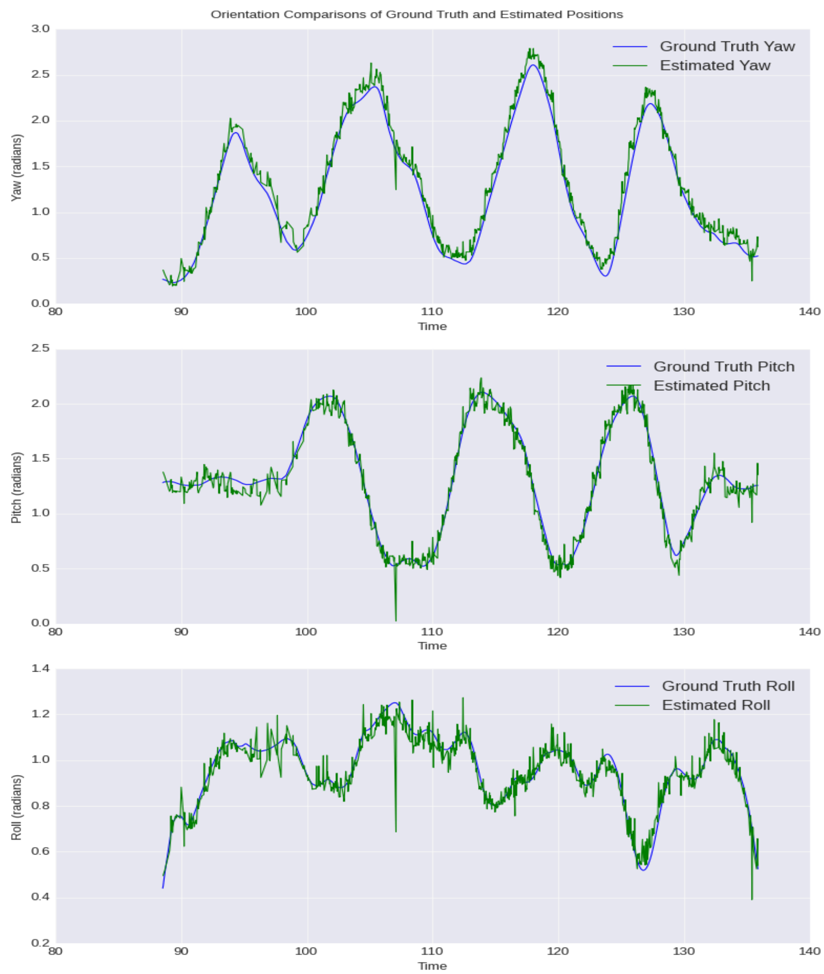


Figure 30: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 0

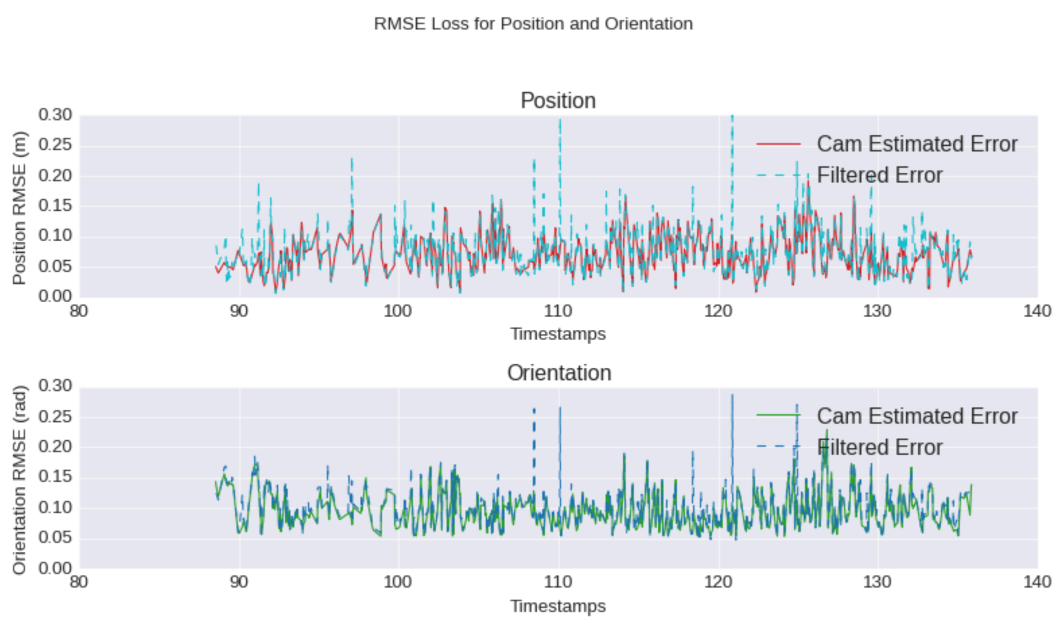


Figure 31: RMSE Plots for .Mat File 0

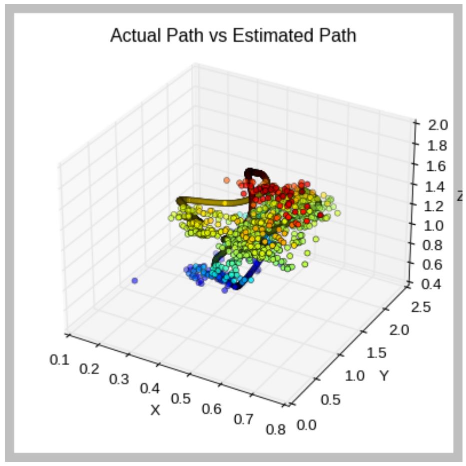


Figure 32: Actual and Filtered Path Plots for .Mat File 1

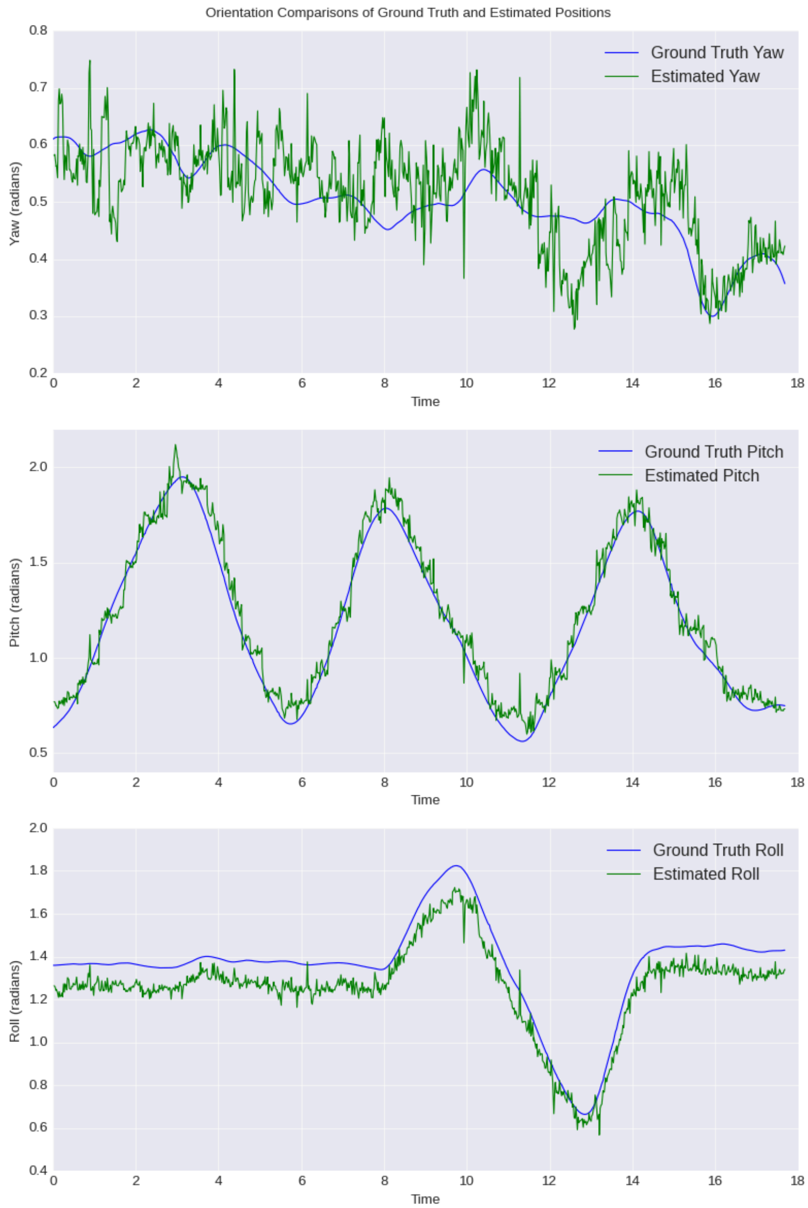


Figure 33: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 1

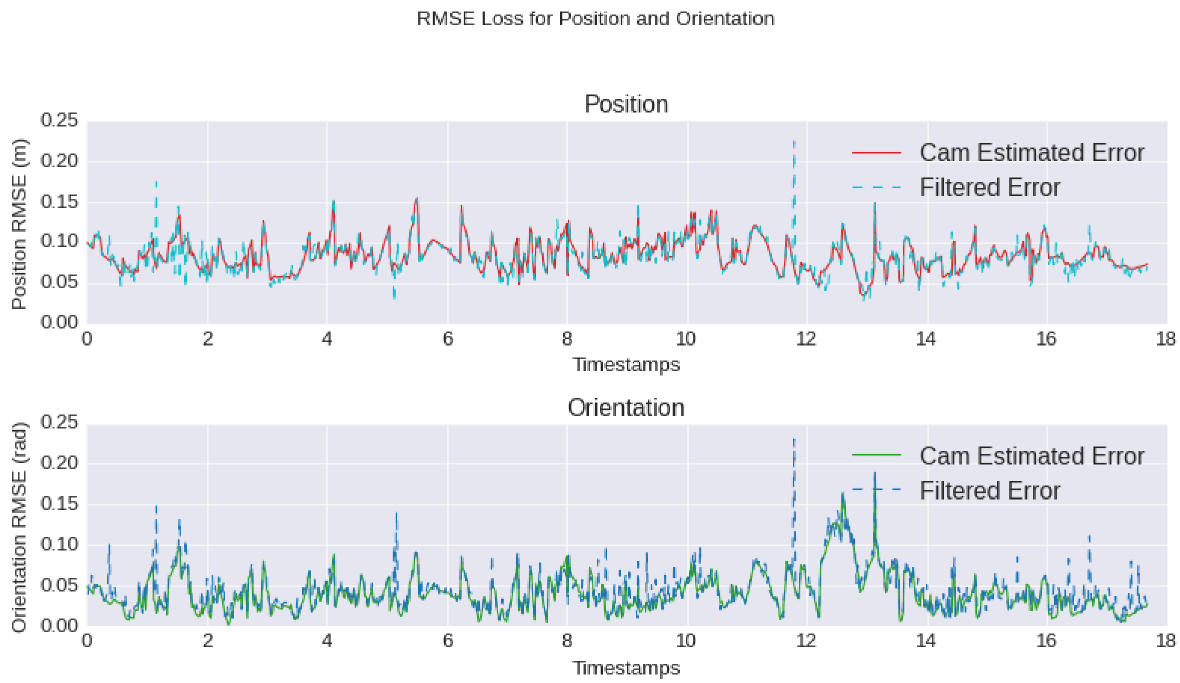


Figure 34: RMSE Plots for .Mat File 1

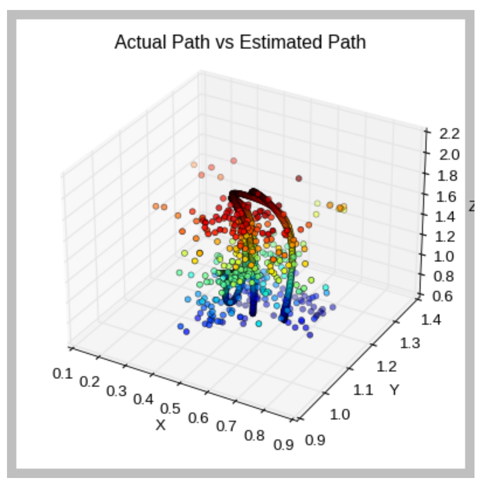


Figure 35: Actual and Filtered Path Plots for .Mat File 2

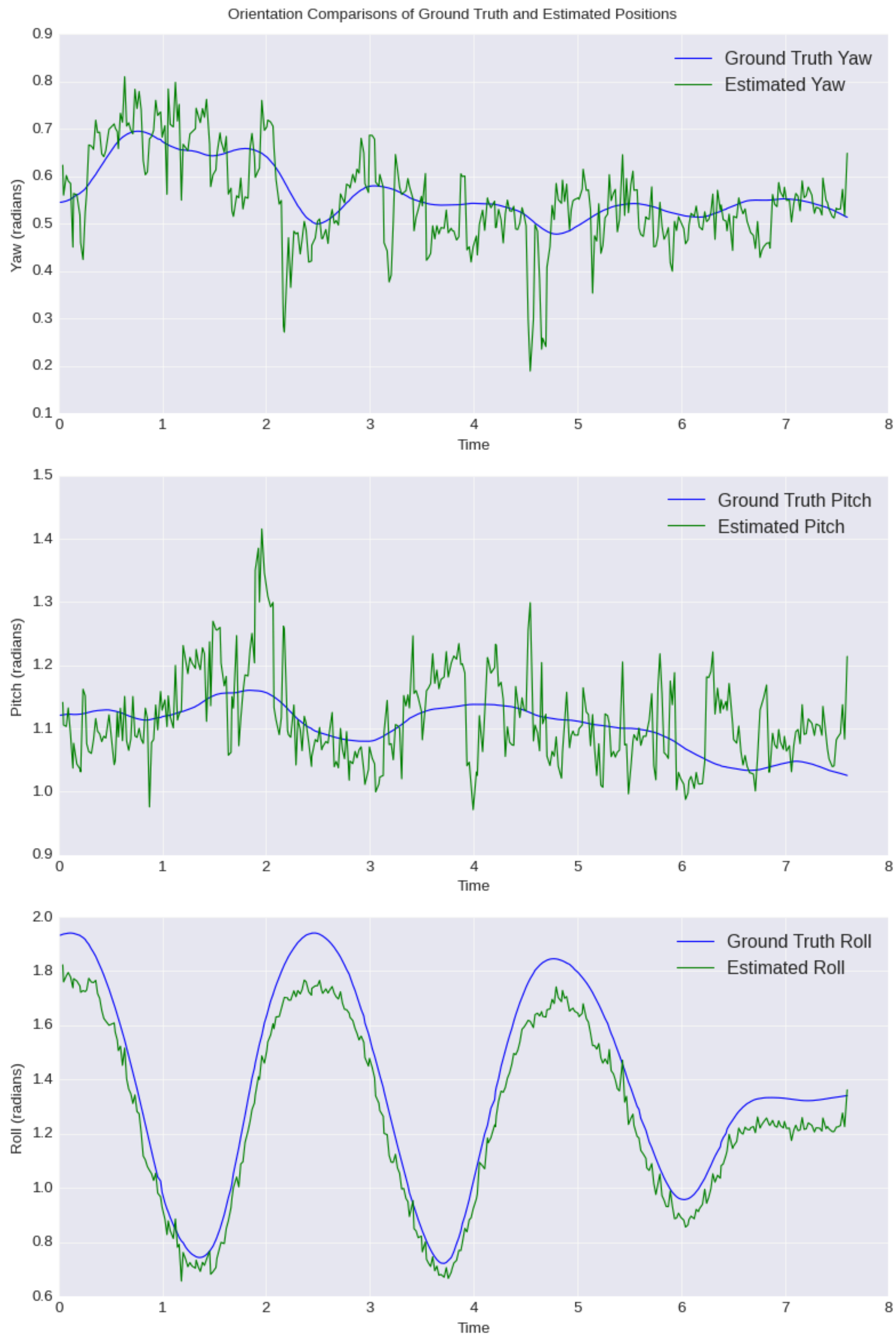


Figure 36: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 2

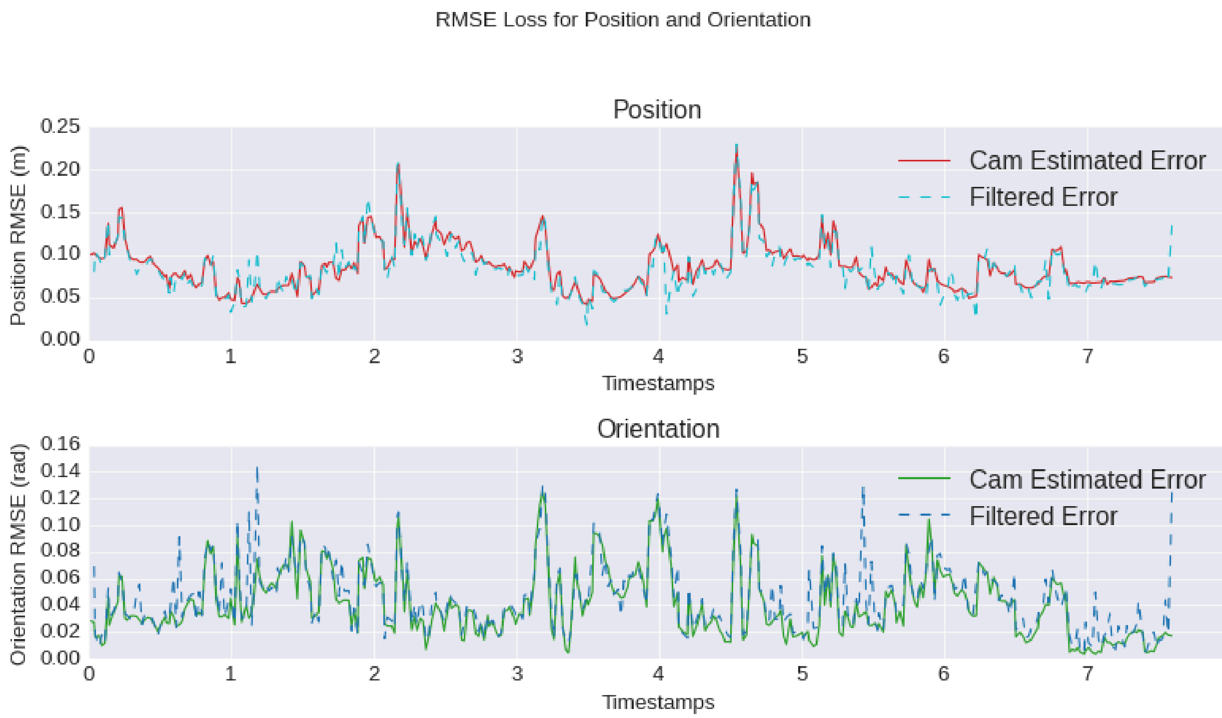


Figure 37: RMSE Plots for .Mat File 2

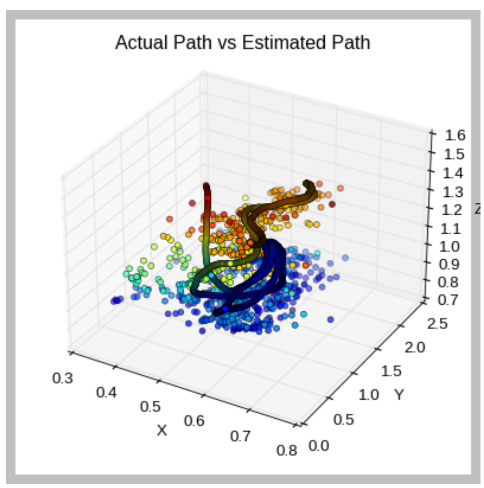


Figure 38: Actual and Filtered Path Plots for .Mat File 3

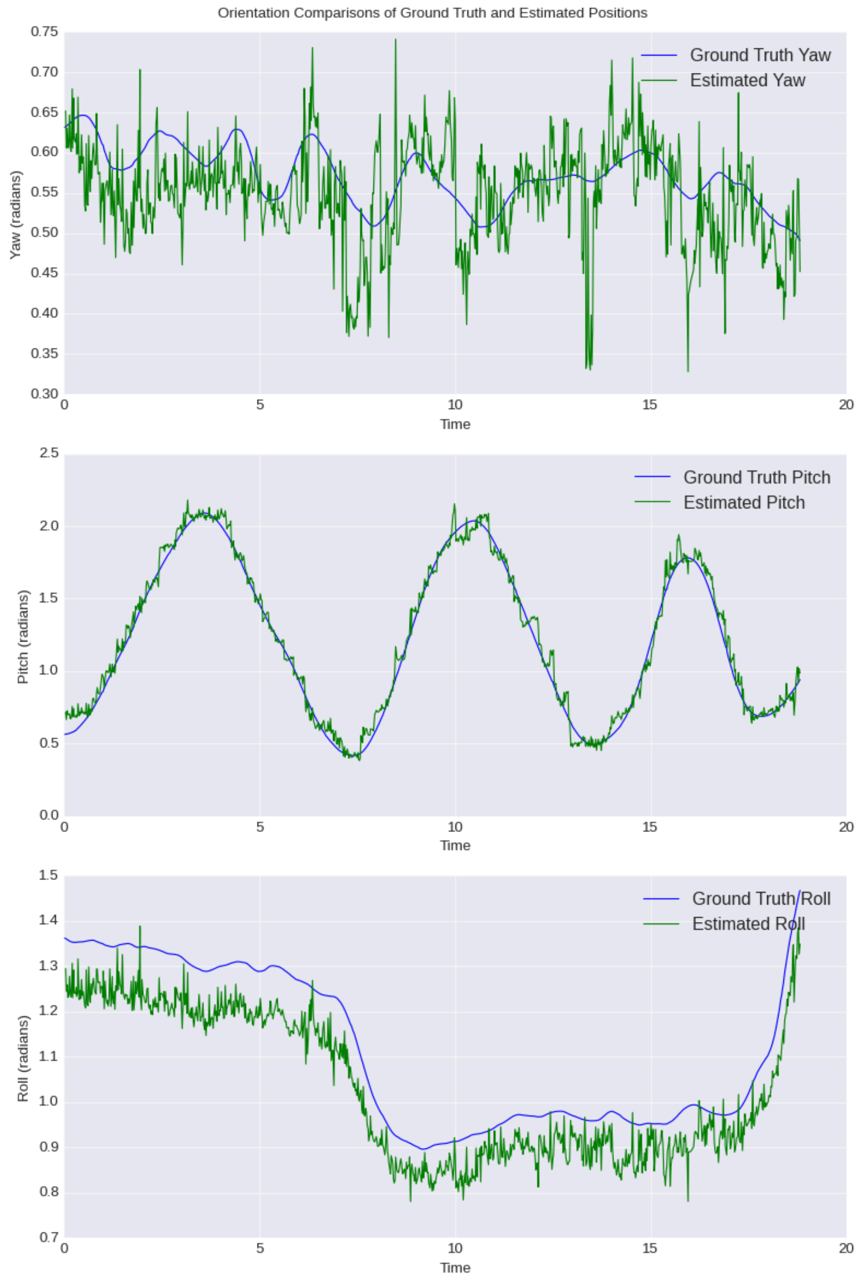


Figure 39: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 3

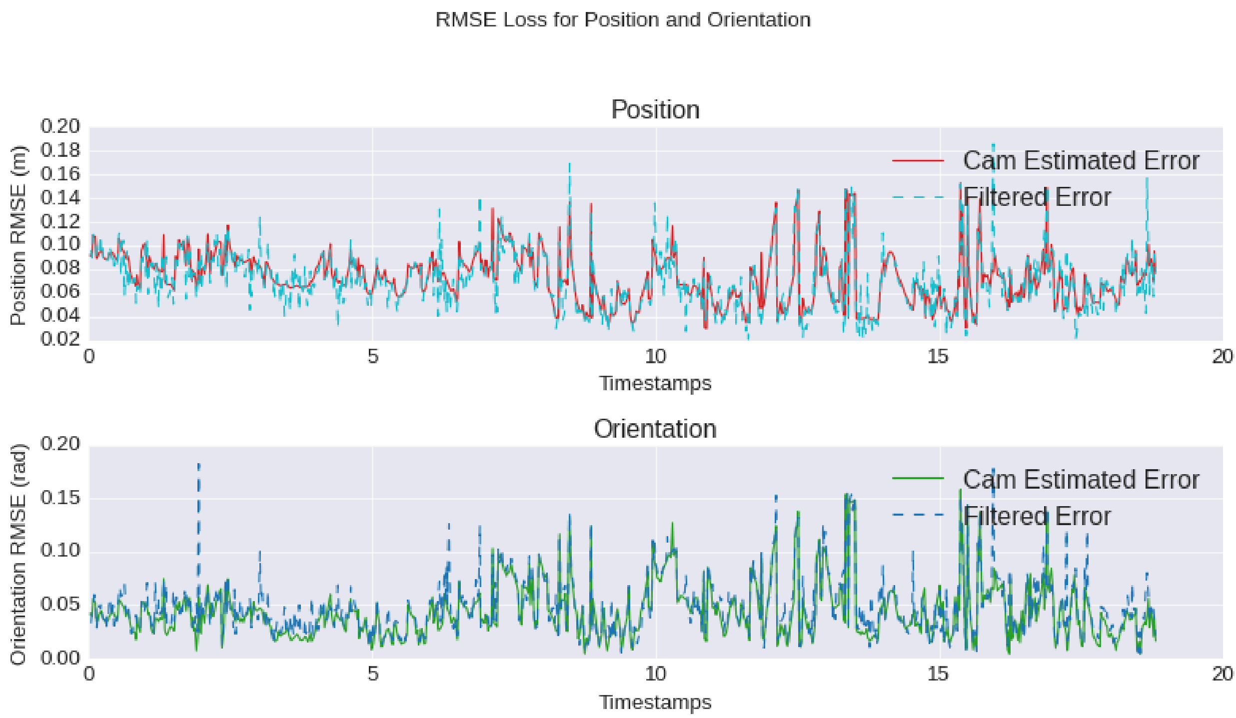


Figure 40: RMSE Plots for .Mat File 3

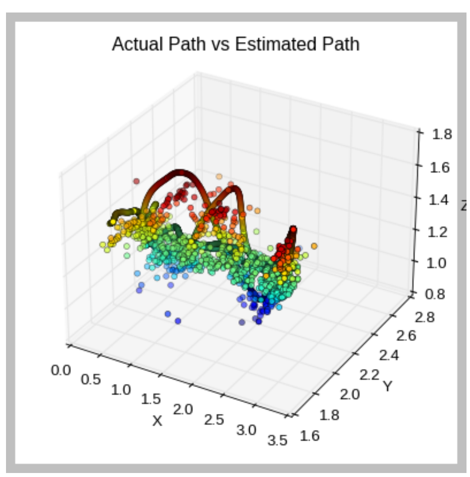


Figure 41: Actual and Filtered Path Plots for .Mat File 4

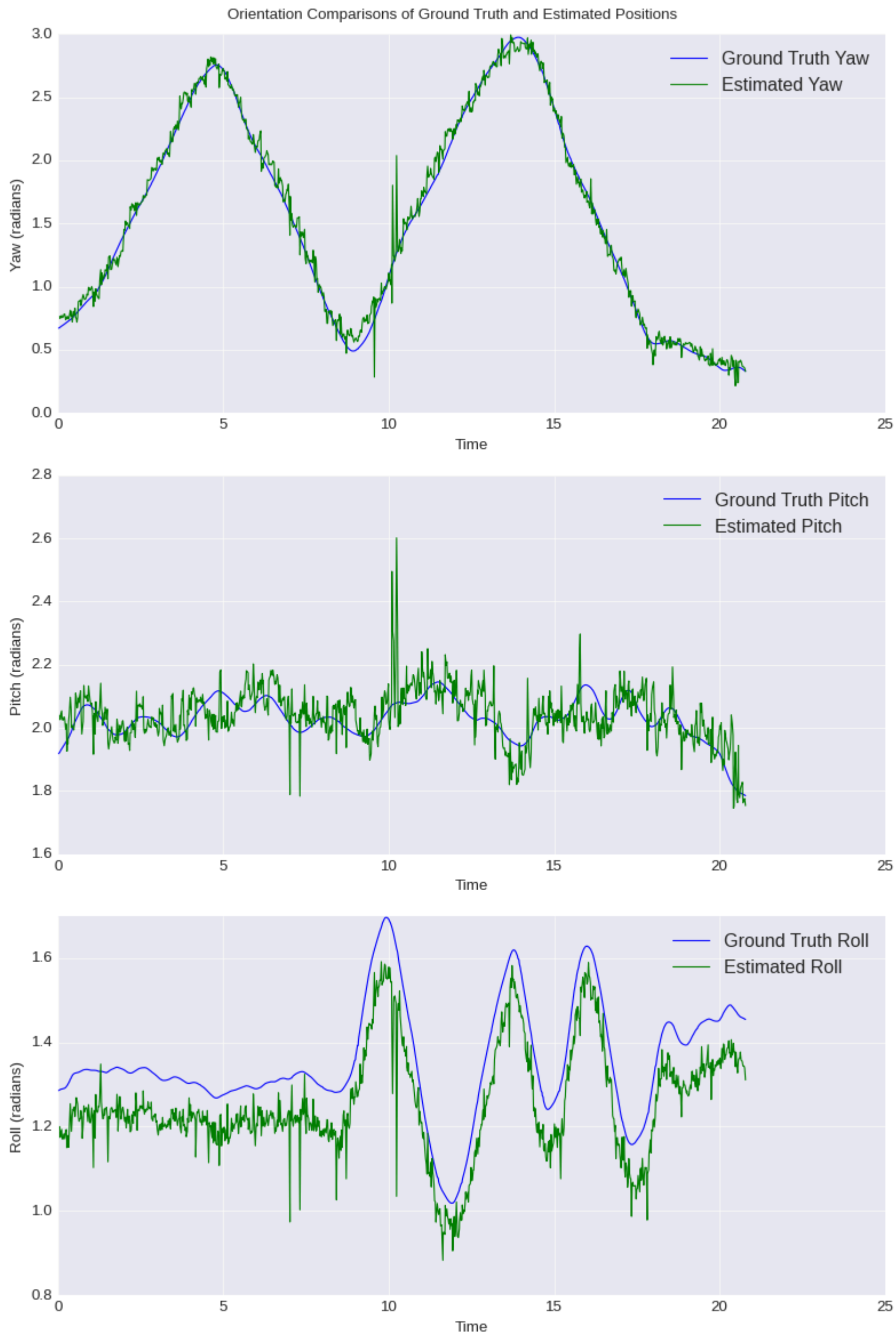


Figure 42: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 4

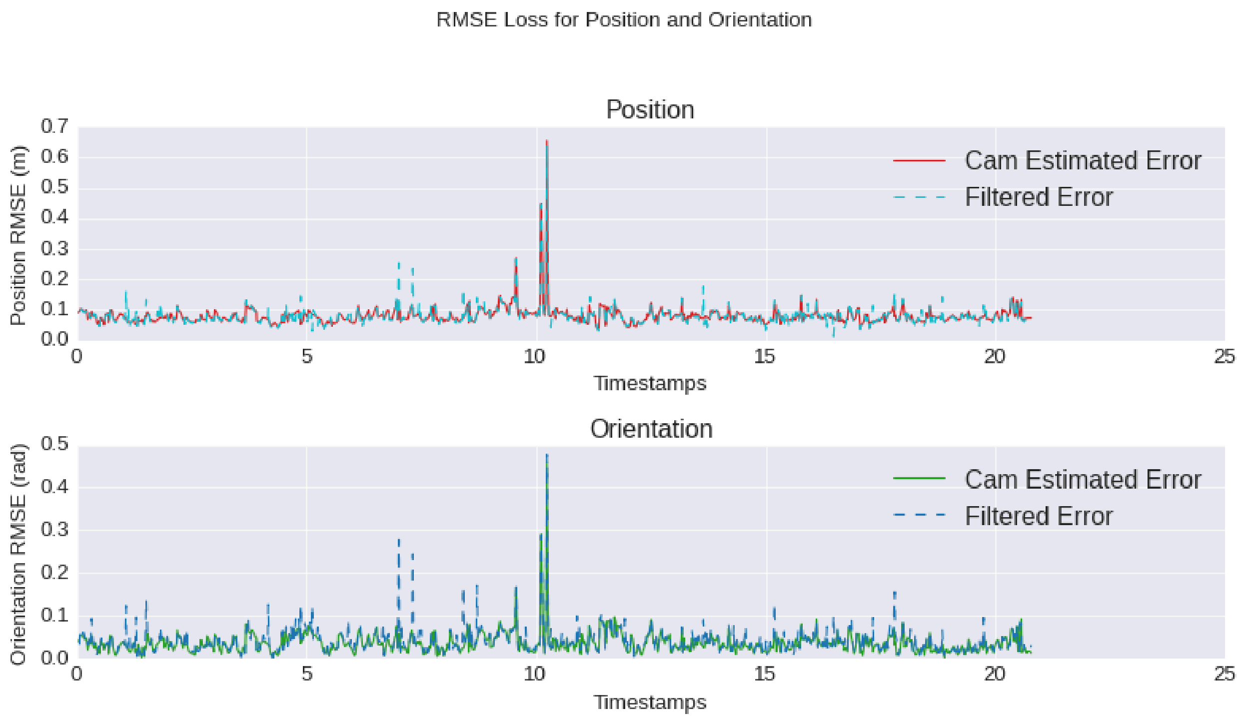


Figure 43: RMSE Plots for .Mat File 4

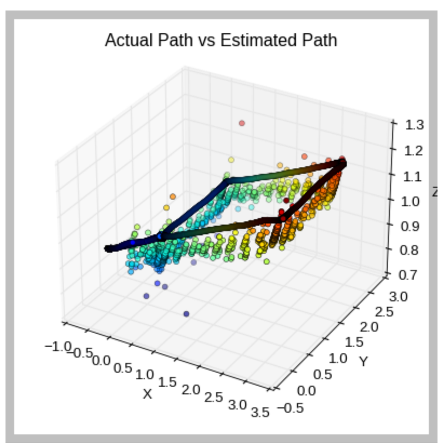


Figure 44: Actual and Filtered Path Plots for .Mat File 5

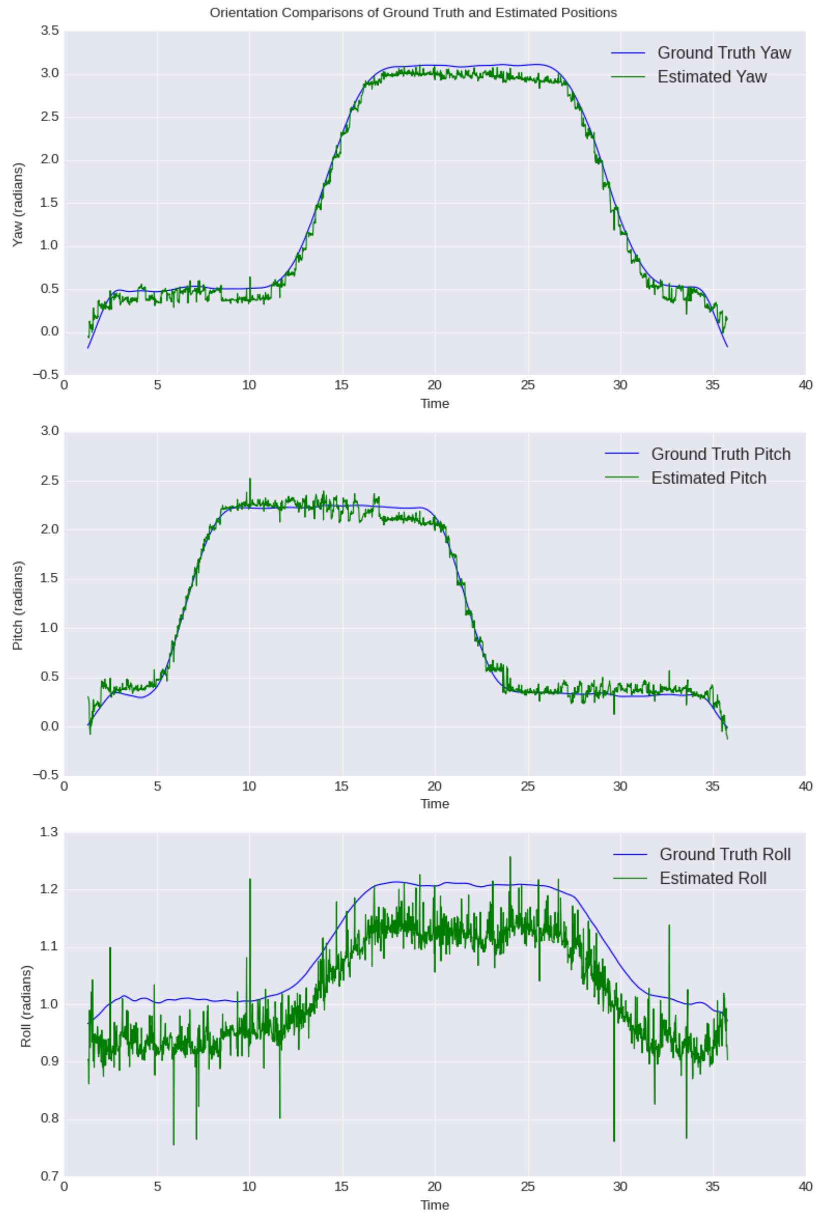


Figure 45: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 5

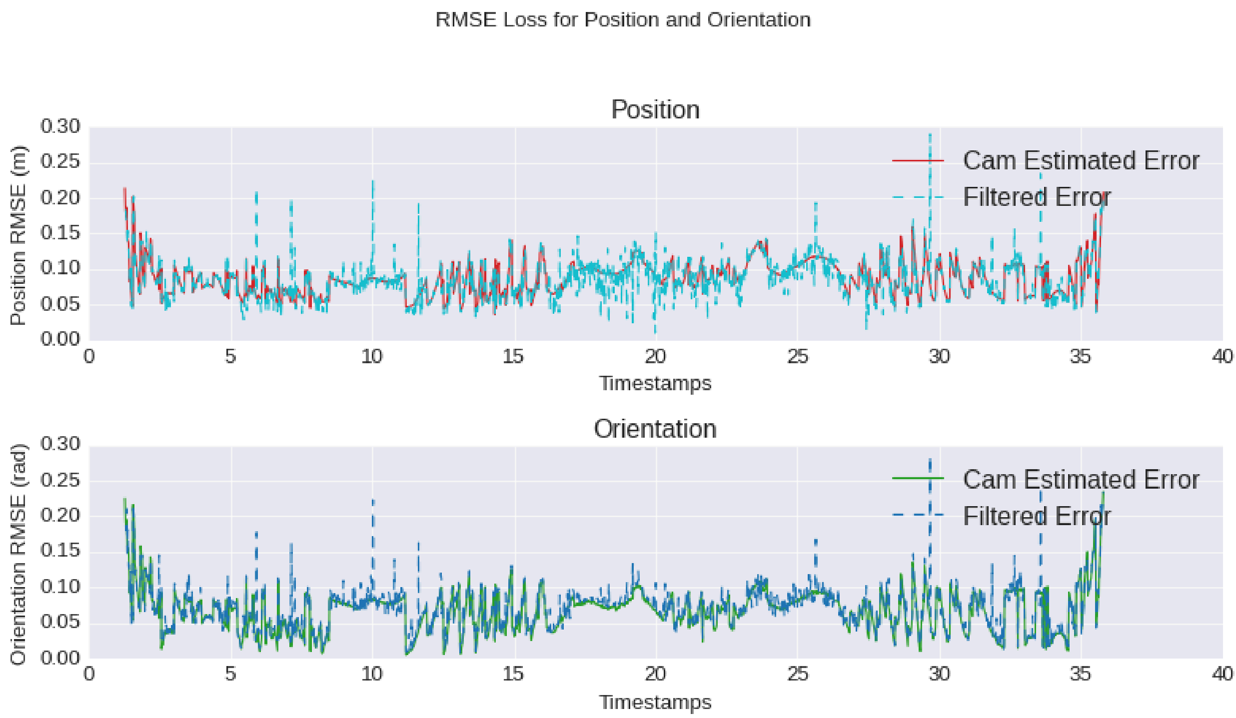


Figure 46: RMSE Plots for .Mat File 5

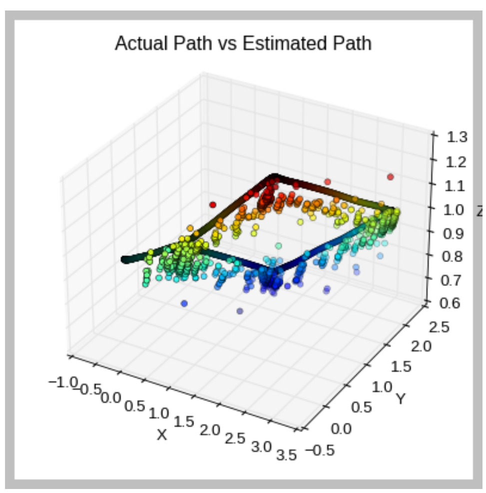


Figure 47: Actual and Filtered Path Plots for .Mat File 6

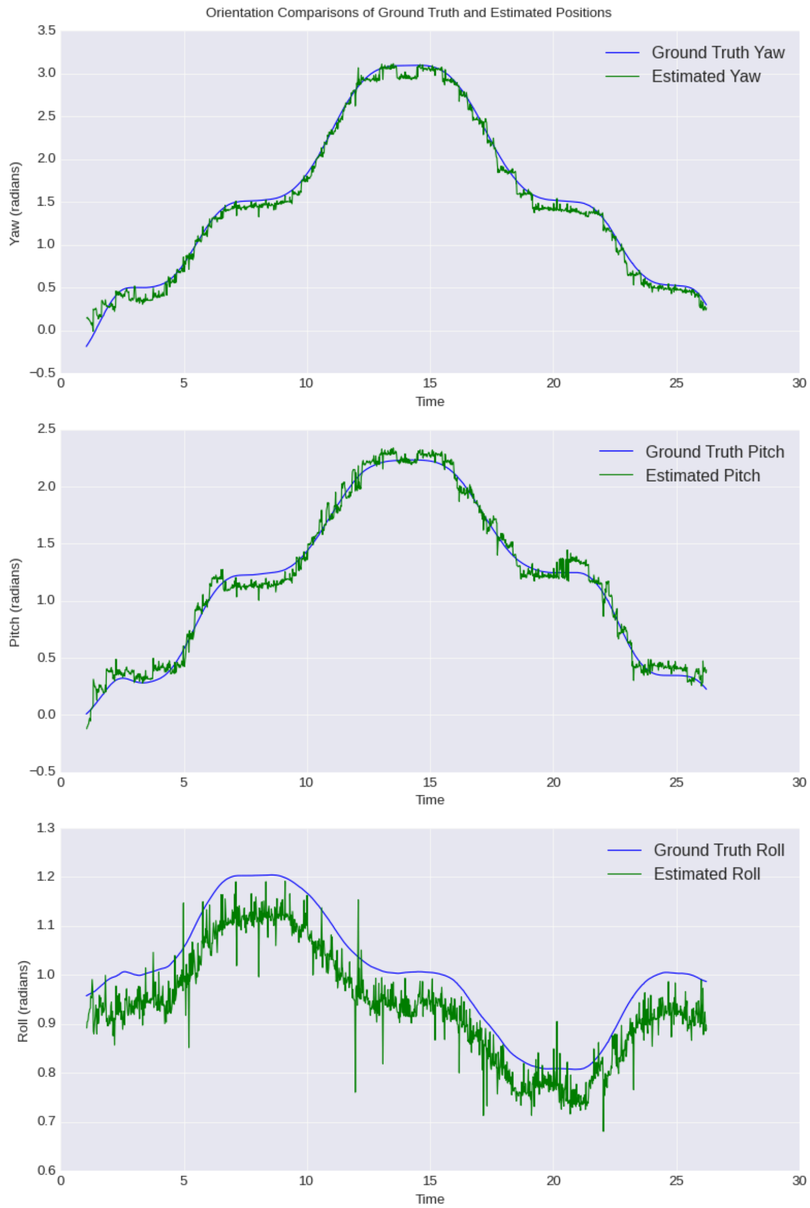


Figure 48: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 5

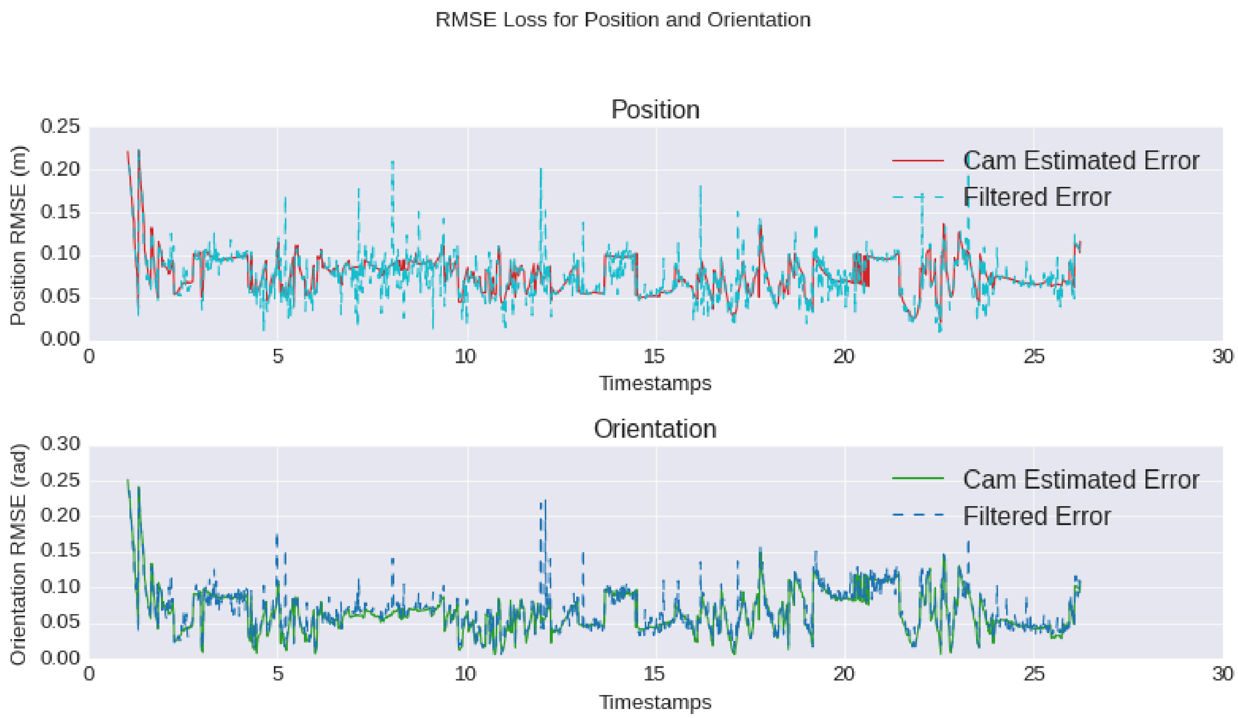


Figure 49: RMSE Plots for .Mat File 6

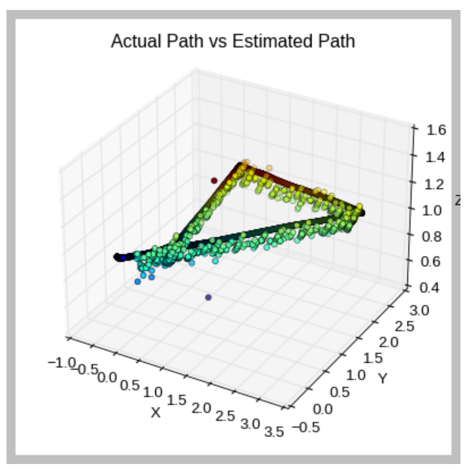


Figure 50: Actual and Filtered Path Plots for .Mat File 7

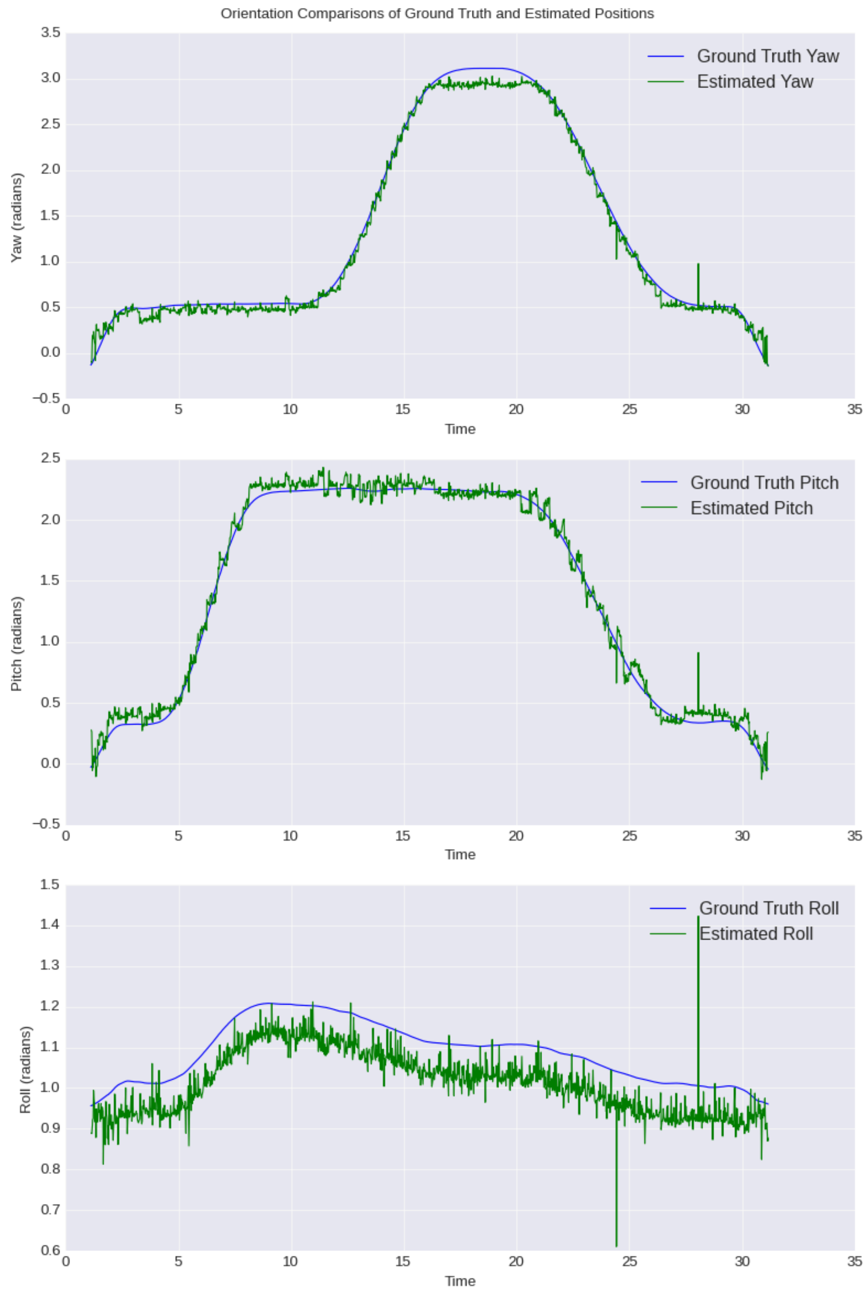


Figure 50: Actual vs Filtered Roll, Pitch, Yaw Plot for .Mat File 7

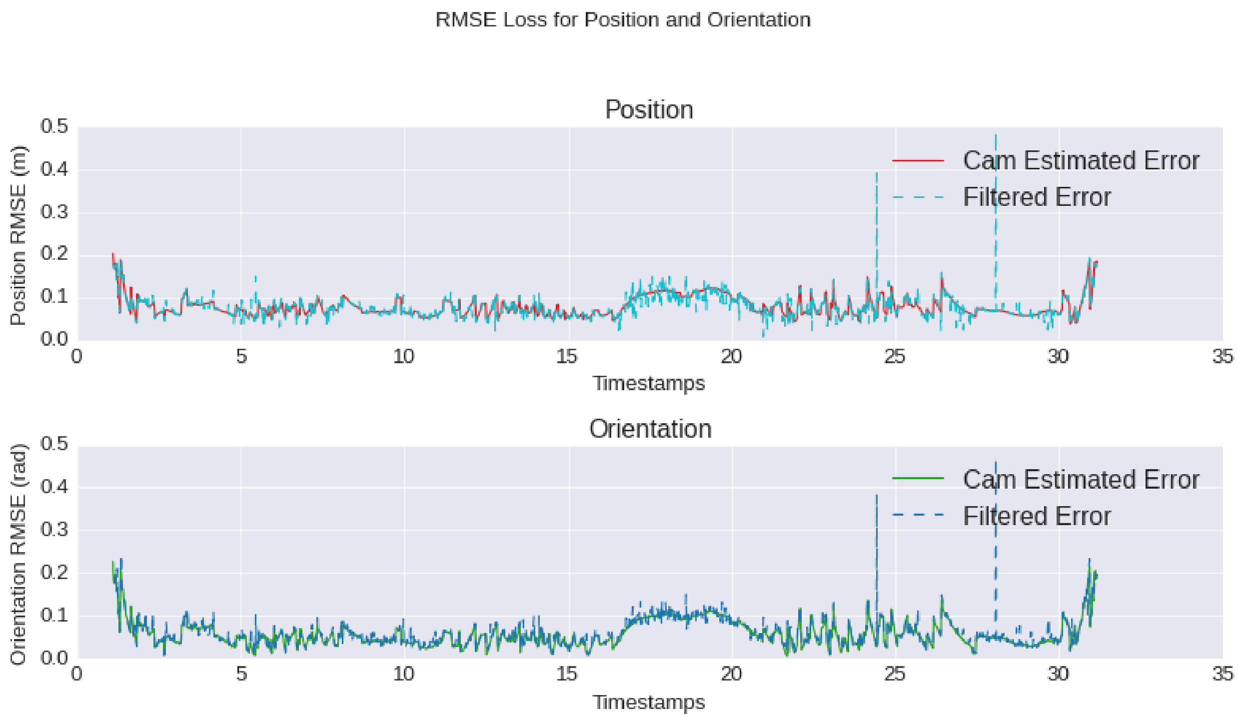


Figure 51: RMSE Plots for .Mat File 7

Conclusion:

The charts show that the filter's results are mostly in line with the actual data, which suggests the Kalman Filter is working as expected. The close match means the settings for how much we trust our measurements versus our predictions are probably set right. These positive results could be attributed due to the well-tuned covariance matrices used as recommended in the task. Also, using NumPy vectorized operations has likely helped make these results both fast and accurate.

When looking at the RMSE, or the average error in our estimates, we see a similar pattern that supports the idea that our filter is performing well.

However, there are some unexpected spikes in the filtered data which stand out. These might be due to small errors in how we're measuring or in the estimation process. Or they could show places where our filter needs a bit more fine-tuning.

Appendix

Code for Task3:

```
from typing import List
import numpy as np

# function estimates covariance matrix based on true pose, estimated positions, orientations, and
sensor data.
def estimateCovariance(groundTruth: List[TruePose], getPositions: List[np.ndarray],
getOrientations: List[np.ndarray], data: List[MeasurementData],) -> np.ndarray:
    errorSum = np.zeros((6, 6))
    validSamples = 0

    # loop through corresponding elements of data and ground truth
    for estimation, localPosition, localOrientation in zip(data, getPositions, getOrientations):
        if estimation.dataTime < groundTruth[0].dataTime:
            continue

        try:
            # generate an estimated pose to compare against measurement
            smoothedState = genEstimatedPose(groundTruth, estimation)
        except ValueError:
            continue

        # combine position and orientation into a single state vector
        estimatedState = np.concatenate([
            np.ravel(localPosition),  # convert to 1D array if not already.
            np.ravel(localOrientation)  # convert to 1D array if not already.
        ]).reshape(6, 1)

        # calculate error vector and accumulate sum
        stateError = smoothedState - estimatedState
        errorSum += stateError @ stateError.T
        validSamples += 1

    # compute average covariance matrix if there are enough valid samples
    if validSamples > 1:
        avgCov = errorSum / (validSamples - 1)
    else:
        raise ValueError("Not enough data to estimate covariances.")

    return avgCov

# function generates an estimated pose based on data from ground truth and current measurement
def genEstimatedPose(gts: List[TruePose], estimation: MeasurementData) -> np.ndarray:
    # filter ground truths to find adjacent points for interpolation
    adjacentGTs = [gt for gt in gts if gt.dataTime <= estimation.dataTime]
    if not adjacentGTs:
        raise ValueError("Estimation precedes all ground truths.")
    previousGT = adjacentGTs[-1]
    try:
        nextGT = gts[len(adjacentGTs)]
    except IndexError:
        raise ValueError("No ground truth found after the estimation timestamp.")
    # compute interpolation weights and perform interpolation
    totalDelta = nextGT.dataTime - previousGT.dataTime
    weightA = (nextGT.dataTime - estimation.dataTime) / totalDelta
    weightB = 1 - weightA
    # combine weighted vectors to get interpolated pose
```

```
    interpolatedPose = weightA * np.array([previousGT.position_x, previousGT.position_y,
previousGT.position_z, previousGT.rotationRoll, previousGT.rotationPitch,
previousGT.rotationYaw]) \
                       + weightB * np.array([nextGT.position_x, nextGT.position_y,
nextGT.position_z, nextGT.rotationRoll, nextGT.rotationPitch, nextGT.rotationYaw])

    return interpolatedPose.reshape(6, 1)


if __name__ == "__main__":
    selectedDataset = mat_file_path0
    #selectedDataset = mat_file_path1
    #selectedDataset = mat_file_path2
    #selectedDataset = mat_file_path3
    #selectedDataset = mat_file_path4
    #selectedDataset = mat_file_path5
    #selectedDataset = mat_file_path6
    #selectedDataset = mat_file_path7

    # parse data using selected dataset
    readings, groundTruth = parseData(selectedDataset)
    # initialize a LayoutMap object
    getInstance = LayoutMap()

    # initialize lists to hold processed data
    estimatedPositions: List[np.ndarray] = []
    estimatedOrientations: List[np.ndarray] = []
    validData: List[readings] = []

    # process data from dataset
    for datum in readings:
        if len(datum.tagList) == 0:
            continue
        validData.append(datum)
        # get pose estimations for each datum
        tempOrientation, tempPosition = getInstance.getPose(datum.tagList)
        estimatedPositions.append(tempPosition)
        estimatedOrientations.append(getYawPitchRoll(tempOrientation))

    # estimate covariance matrix based on gathered data
    avgCov = estimateCovariance(groundTruth, estimatedPositions, estimatedOrientations,
validData)
    print("----------------------------------------------------------------------")
    print(f"Print results: {selectedDataset}")
    print(avgCov)
    print("----------------------------------------------------------------------")
```

Code for Task4:

```
import numpy as np
from time import time
from typing import List, Optional, Tuple
from scipy.linalg import sqrtm

# function to create 3D scatter plots
def get3DPlots(plotTitle: str, *labelDataPairs, figsize: Tuple[int, int] = (10, 6)) ->
plt.Figure:
    with plt.style.context('classic'):
      if labelDataPairs is None or len(labelDataPairs) == 0:
          raise ValueError("No plotting arguments provided")
```

```
      elif len(labelDataPairs) % 2 != 0:
          raise ValueError("Arguments must be in pairs of label and data")

      # initialize figure and 3D axes for plot
      plotFigure = plt.figure(figsize=figsize)
      axes = plt.axes(projection="3d")
      axes.set_xlabel("X")
      axes.set_ylabel("Y")
      axes.set_zlabel("Z")
      axes.dist = 10
      axes.set_title(plotTitle)

      # loop through label and data pairs and add them to scatter plot
      labelDataPairs = list(labelDataPairs)
      while labelDataPairs:
          dataLabel = labelDataPairs.pop(0)
          readings = labelDataPairs.pop(0)
          axes.scatter3D([coord[0] for coord in readings], [coord[1] for coord in readings],
[coord[2] for coord in readings],
              c=[coord[2] for coord in readings], linewidths=0.5, label=dataLabel,)

      return plotFigure

# function to create RMSE charts for position and orientation
def getRMSECharts(GTs, cameraPoses, states, timestamps, figsize=(10, 6)) -> plt.Figure:
    # extract positions and orientations from inputs
    gt_pos = np.array([gt[:3] for gt in GTs]).reshape(-1, 3)
    gt_orientations = np.array([gt[3:6] for gt in GTs]).reshape(-1, 3)
    camera_pos = np.array([estimate[:3] for estimate in cameraPoses]).reshape(-1, 3)
    camera_orientations = np.array([estimate[3:6] for estimate in cameraPoses]).reshape(-1, 3)
    states_pos = np.array([state[:3] for state in states]).reshape(-1, 3)
    states_orientations = np.array([state[3:6] for state in states]).reshape(-1, 3)

    # calculate RMSE for position and orientation for camera estimates
    camera_position_rmse = np.sqrt(np.mean((gt_pos - camera_pos) ** 2, axis=1))
    camera_orientation_rmse = np.sqrt(np.mean((gt_orientations - camera_orientations) ** 2,
axis=1))

    # calculate RMSE for position and orientation for filter states
    skip_first = 0 if len(gt_pos) == len(states_pos) else 1
    state_position_rmse = np.sqrt(np.mean((gt_pos[skip_first:] - states_pos) ** 2, axis=1))
    state_orientation_rmse = np.sqrt(np.mean((gt_orientations[skip_first:] - states_orientations)
** 2, axis=1))

    rmse_figure, (ax1, ax2) = plt.subplots(2, 1, figsize=figsize)
    rmse_figure.suptitle("RMSE Loss for Position and Orientation")

    # position RMSE
    ax1.plot(timestamps, camera_position_rmse, label="Cam Estimated Error", color='tab:red')
    ax1.plot(timestamps[1:], state_position_rmse, label="Filtered Error", color='tab:cyan',
linestyle='--')
    ax1.set_title("Position")
    ax1.set_xlabel("Timestamps")
    ax1.set_ylabel("Position RMSE (m)")
    ax1.legend()

    # orientation RMSE
    ax2.plot(timestamps, camera_orientation_rmse, label="Cam Estimated Error", color='tab:green')
    ax2.plot(timestamps[1:], state_orientation_rmse, label="Filtered Error", color='tab:blue',
linestyle='--')
    ax2.set_title("Orientation")
```

```python
    ax2.set_xlabel("Timestamps")
    ax2.set_ylabel("Orientation RMSE (rad)")
    ax2.legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    return rmse_figure

def getOriPlots(ground_truth: List[np.ndarray], estimates: List[np.ndarray], timestamps: 
List[float]) -> plt.Figure:
    # extract estimated orientations
    yaw_estimated = [orientation[0] for orientation in estimates]
    pitch_estimated = [orientation[1] for orientation in estimates]
    roll_estimated = [orientation[2] for orientation in estimates]

    # extract ground truth orientations
    yaw_gt = [gt[0] for gt in ground_truth]
    pitch_gt = [gt[1] for gt in ground_truth]
    roll_gt = [gt[2] for gt in ground_truth]

    # create a figure with three subplots, one for each orientation component
    orientations_figure, axs = plt.subplots(3, 1, figsize=(10, 15))
    orientations_figure.suptitle("Orientation Comparisons of Ground Truth and Estimated 
Positions")
    estimate_timestamps = timestamps if len(timestamps) == len(estimates) else timestamps[1:]

    # yaw plot
    axs[0].plot(timestamps, yaw_gt, label="Ground Truth Yaw")
    axs[0].plot(estimate_timestamps, yaw_estimated, label="Estimated Yaw")
    axs[0].set_xlabel("Time")
    axs[0].set_ylabel("Yaw (radians)")
    axs[0].legend()

    # pitch plot
    axs[1].plot(timestamps, pitch_gt, label="Ground Truth Pitch")
    axs[1].plot(estimate_timestamps, pitch_estimated, label="Estimated Pitch")
    axs[1].set_xlabel("Time")
    axs[1].set_ylabel("Pitch (radians)")
    axs[1].legend()

    # roll plot
    axs[2].plot(timestamps, roll_gt, label="Ground Truth Roll")
    axs[2].plot(estimate_timestamps, roll_estimated, label="Estimated Roll")
    axs[2].set_xlabel("Time")
    axs[2].set_ylabel("Roll (radians)")
    axs[2].legend()

    plt.tight_layout()
    return orientations_figure

class unscentedKalmanFilter:

    def __init__(self, measCovMat: Optional[np.ndarray] = None, scalingFactor: float = 1, 
sigmaPointSpread: float = 1, distributionShape: float = 2.0,):
        # initialize filter parameters and state dimensions
        self.stateDimension = 15
        self.scalingFactor = scalingFactor
        self.sigmaPointSpread = sigmaPointSpread
        self.distributionShape = distributionShape
        # measurement covariance matrix, default or provided
        self.measCovMat = avgCov
        # calculate number of sigma points
```

```
        self.sigmaPts = 2 * self.stateDimension + 1
        # compute lambda for sigma point calculations
        self.lamb = self.sigmaPointSpread ** 2 * (self.stateDimension + self.scalingFactor) -
self.stateDimension
        tempLambda = self.lamb + self.stateDimension
        # initialize weights for mean and covariance calculations
        self.meanWeights = np.full(self.sigmaPts, 1 / (2 * tempLambda))
        self.covWeights = np.full(self.sigmaPts, 1 / (2 * tempLambda))
        # adjust first weight differently based on lambda
        self.meanWeights[0] = self.lamb / tempLambda
        self.covWeights[0] = self.lamb / tempLambda + 1 - self.sigmaPointSpread ** 2 +
self.distributionShape
        # instance of map for state vector calculations
        self.map = LayoutMap()

    def getSigmaPts(self, meanState: np.ndarray, covarianceMatrix: np.ndarray) -> np.ndarray:
        # create sigma points array
        sigmaPts = np.zeros((self.sigmaPts, self.stateDimension, 1))
        # first sigma point is the mean state
        sigmaPts[0] = meanState
        # calculate square root of scaled covariance matrix
        spreadMatrix = sqrtm((self.stateDimension + self.scalingFactor) * covarianceMatrix)
        # generate sigma points based on distribution around mean
        for i in range(self.stateDimension):
            sigmaPts[i + 1] = meanState + spreadMatrix[i].reshape((15, 1))
            sigmaPts[self.stateDimension+ i + 1] = meanState - spreadMatrix[i].reshape((15, 1))

        return sigmaPts

    def dynamicUpdate(self, state: np.ndarray, deltat: float, ua: np.ndarray, uw: np.ndarray) ->
np.ndarray:
        # extract angles/linear velocity from state
        rollAngle, pitchAngle, yawAngle = state[3:6, 0]
        velocityVector = state[6:9, 0]
        gravity = -9.8

        # compute rotation matrices for transformation
        cosTheta, sinTheta = np.cos(rollAngle), np.sin(rollAngle)
        cosPhi, sinPhi = np.cos(pitchAngle), np.sin(pitchAngle)
        cosPsi, sinPsi = np.cos(yawAngle), np.sin(yawAngle)

        # inverse transformation from body to world frame
        inverseBodytoWorldTf = np.array([[cosTheta, 0, sinTheta], [sinPhi * sinTheta / cosPhi,
1.0, -cosTheta * sinPhi / cosPhi], [-sinTheta / cosPhi, 0, cosTheta / cosPhi]])

        # rotation matrix from body to world coordinates
        bodytoWorldRotation = np.array([
            [cosPsi * cosTheta - sinPhi * sinPsi * sinTheta, -cosPhi * sinPsi, cosPsi * sinTheta
+ cosTheta * sinPhi * sinPsi],
            [cosTheta * sinPsi + cosPsi * sinPhi * sinTheta, cosPhi * cosPsi, sinPsi * sinTheta -
cosPsi * cosTheta * sinPhi],
            [-cosPhi * sinTheta, sinPhi, cosPhi * cosTheta]])

        # compute derivatives of position and orientations
        pdot = velocityVector
        omgTf = inverseBodytoWorldTf @ uw
        accTf = gravity + bodytoWorldRotation @ ua
        # calculate total derivatives and update state
        xdot = np.concatenate((pdot, omgTf, accTf, np.zeros(6)))[:, np.newaxis]
        newState = state + xdot * deltat
```

```
        return newState

    def correctState(self, state: np.ndarray, estimatedMean: np.ndarray, sigma: np.ndarray,
sigmaPts) -> np.ndarray:
        # initialize predicted measurements array for each sigma point
        predictedMeasurements = np.zeros_like(sigmaPts)
        # fill predicted measurements by transforming each sigma point
        for i in range(self.sigmaPts):
            predictedMeasurements[i] = self.stateToMeeasurement(sigmaPts[i])
        # initialize mean of predicted measurements
        predictedStateMean = np.zeros((self.stateDimension, 1))
        # compute mean of predicted measurements using weighted average
        for i in range(0, self.sigmaPts):
            predictedStateMean += self.meanWeights[i] * predictedMeasurements[i]
        # initialize measurement noise covariance matrix
        measurementNoiseCov = np.zeros((self.stateDimension, self.stateDimension))
        measurementNoiseCov[0:6, 0:6] = np.diag(self.measCovMat)
        measurementCovariance = np.zeros((self.stateDimension, self.stateDimension))
        # calculate prediction errors
        predictionErrors = predictedMeasurements - predictedStateMean
        # compute measurement covariance matrix using prediction errors and weights
        for i in range(0, self.sigmaPts):
            measurementCovariance += self.covWeights[i] * np.dot(predictionErrors[i],
predictionErrors[i].T)
        measurementCovariance += measurementNoiseCov

        # initialize cross covariance matrix
        crossCovariance = np.zeros((self.stateDimension, self.stateDimension))
        stateDifferences = sigmaPts - estimatedMean
        # compute cross covariance matrix
        for i in range(0, self.sigmaPts):
            crossCovariance += self.covWeights[i] * np.dot(stateDifferences[i],
predictionErrors[i].T)

        # compute kalman gain using pseudo-inverse of measurement covariance matrix
        Kgain = np.dot(crossCovariance, np.linalg.pinv(measurementCovariance))
        # update state using kalman gain and difference between actual state and predicted state
mean
        updatedState = estimatedMean + np.dot(Kgain, state - predictedStateMean)
        # update covariance using kalman gain and measurement covariance
        updatedCovariance = sigma - np.dot(Kgain, measurementCovariance).dot(Kgain.T)
        # adjust covariance matrix to ensure it is positive semi-definite
        updatedCovariance = self.adjustCovMat(updatedCovariance)

        return updatedState, updatedCovariance

    def adjustCovMat(self, updatedCovariance: np.ndarray, noise: float = 1e-3):
        # set number of iterations for covariance matrix adjustment
        maxIterations = 10
        # set initial regulation factor for numerical stability
        regulationFactor = noise

        # iteratively adjust covariance matrix
        for _ in range(maxIterations):
            # ensure covariance matrix is symmetric
            updatedCovariance = (updatedCovariance + updatedCovariance.T) / 2
            # compute eigenvalues and eigenvectors of covariance matrix
            eigenValues, eigenVectors = np.linalg.eig(updatedCovariance)
            # if all eigenvalues are positive, matrix is already valid

            if np.all(eigenValues > 0):
```

```
                return updatedCovariance
            # adjust eigenvalues to be positive and scale up regulation factor if needed
            eigenValues = np.where(eigenValues > 0, eigenValues, 0) + regulationFactor
            updatedCovariance = eigenVectors.dot(np.diag(eigenValues)).dot(eigenVectors.T)
            # increase regulation factor for next iteration if needed
            regulationFactor *= 10

        return updatedCovariance

    def stateToMeeasurement(self, state: np.ndarray) -> np.ndarray:
        measurementVector = np.zeros((self.stateDimension, 1))
        transformMatrix = np.zeros((6, self.stateDimension))
        transformMatrix[0:6, 0:6] = np.eye(6)

        noiseCovariance = np.diag(self.measCovMat).reshape(6, 1)
        measurementVector[0:6] = np.dot(transformMatrix, state) + noiseCovariance

        return measurementVector

    def statePropagation(self, sigmaPts: np.ndarray, ua: np.ndarray, uw: np.ndarray, deltat:
float) -> Tuple[np.ndarray]:
       # initialize an array to hold predicted states for each sigma point
        predictedStates = np.zeros_like(sigmaPts)
        # update each sigma point based on dynamic model
        for i in range(sigmaPts.shape[0]):
            predictedStates[i, :] = self.dynamicUpdate(sigmaPts[i], deltat, uw, ua)
        # calculate mean of predicted states using weights
        predictedMean = np.zeros((self.stateDimension, 1))
        for i in range(0, self.sigmaPts):
            predictedMean += self.meanWeights[i] * predictedStates[i]

        # add some noise to process covariance as a part of uncertainty
        processNoiseCovariance = np.random.normal(scale=5e-1, size=(15, 15))
        stateDifferences = predictedStates - predictedMean
        # get covariance of predicted states
        covariancePrediction = np.zeros((self.stateDimension, self.stateDimension))
        for i in range(0, self.stateDimension):
            covariancePrediction += self.covWeights[i] * np.dot(stateDifferences[i],
stateDifferences[i].T)
        covariancePrediction += processNoiseCovariance

        return predictedMean, covariancePrediction, predictedStates

    def stateVectorFromData(self, sensorData: MeasurementData, previousState: np.ndarray,
previousTimestamp: float) -> np.ndarray:
        # initialize state vector with zeros
        stateVector = np.zeros((15, 1))
        # obtain pose from sensor data
        orientation, position = self.map.getPose(sensorData.tagList)
         # asign position and orientation to state vector
        stateVector[:3, 0] = position  # Position expected as a flat array
        stateVector[3:6, 0] = orientation  # Orientation expected as a flat array

        return stateVector


    def executeFilter(self, estimatePos: List[MeasurementData]) -> List[np.ndarray]:
        # create initial state vector from first sensor data
        stateVector = self.stateVectorFromData(estimatePos[0], np.zeros((self.stateDimension,
1)), 0.0)
        # initialize velocity part of state vector
```

```python
        stateVector[6:9] = np.zeros((3, 1))
        # record initial timestamp
        prevTimeStamp = estimatePos[0].dataTime

         # initialize covariance matrix with small uncertainties
        calculateCovMat = np.eye(self.stateDimension) * 1e-3
        filteredPos = []

        # iterate over all sensor data to update state vector
        for data in estimatePos[1:]:

            # update state vector from data
            stateVector = self.stateVectorFromData(data, stateVector, prevTimeStamp)

            deltat = data.dataTime - prevTimeStamp
            prevTimeStamp = data.dataTime
            # generate sigma points for current state
            sigmaPts = self.getSigmaPts(stateVector, calculateCovMat)
            # propagate sigma points through process model
            predictedMean, predictedCovariance, predictedStates = self.statePropagation(sigmaPts,
data.accelVector, data.angularVelocity, deltat)
            # correct predicted mean and covariance based on measurements
            updatedMean, updatedCovariance = self.correctState(stateVector, predictedMean,
predictedCovariance, predictedStates)
            # update state vector and covariance matrix
            stateVector = updatedMean
            calculateCovMat = updatedCovariance
            # append updated state to list of filtered positions
            filteredPos.append(stateVector)

        return filteredPos

def extractDataFeatures(dataInput, gt):
    # process raw data to generate positions, orientations, times, and measurements
    poseEstimator = LayoutMap()
    data, smoothedGT, cameraPoses = [], [], []

   # loop through each datum in  input data
    for datum in dataInput:
        if not datum.tagList:
            continue
        try:
            # estimate and append smoothed pose to ground truth list
            smoothedGT.append(genEstimatedPose(gt, datum))
            # get pose from LayoutMap object and append to camera poses
            orientation, position = poseEstimator.getPose(datum.tagList)
            cameraPoses.append(np.concatenate([position, orientation]))
            # append processed datum to data list
            data.append(datum)
        except Exception as e:
            print(f"Skipping datum due to error: {e}")
            continue

    return data, smoothedGT, cameraPoses

def ukfWrapper(data):
    # initialize UKF and run filter over data, return results and time taken
    getInstance = unscentedKalmanFilter() # create an instance of UKF class
    init = time() # record start time.
    getFiltered = getInstance.executeFilter(data) # execute filter
    total = time() - init
```

```python
    return getFiltered, total # return filtered data and time taken

if __name__ == "__main__":
    # uncomment dataset to perform the UKF on that specific dataset
    selectedDataset = mat_file_path0
    #selectedDataset = mat_file_path1
    #selectedDataset = mat_file_path2
    #selectedDataset = mat_file_path3
    #selectedDataset = mat_file_path4
    #selectedDataset = mat_file_path5
    #selectedDataset = mat_file_path6
    #selectedDataset = mat_file_path7

    # parse selected dataset to get readings and ground truth values
    readings, groundTruth = parseData(selectedDataset)
    # extract relevant features from readings for UKF processing
    readings, smoothedGT, cameraPoses = extractDataFeatures(readings, groundTruth)
    # apply unscented kalman filter on readings and measure time taken
    ukfResults, totalTime = ukfWrapper(readings)
    # stack all ukf results into a single array for analysis
    stateArray = np.stack(ukfResults)
    # generate root mean square error charts for position and orientation
    rmsePlots = getRMSECharts(smoothedGT, cameraPoses, stateArray, [d.dataTime for d in
readings], figsize=(10, 6))
    # plot 3d trajectories of actual vs estimated paths
    show3DPlot = get3DPlots(
        "Actual Path vs Estimated Path", "GT",
        [Point3D(coord_x=groundTruth[i].position_x, coord_y=groundTruth[i].position_y,
coord_z=groundTruth[i].position_z) for i in range(len(groundTruth))], "Estimated UKF",
        [Point3D(coord_x=position[0], coord_y=position[1], coord_z=position[2]) for position in
ukfResults],  figsize=(10, 6))
    # generate orientation plots comparing ukf estimates against ground truth
    plotOrientation = getOriPlots(smoothedGT, ukfResults, [d.dataTime for d in readings])
    # display generated RMSE and 3D plots
    rmsePlots.show()
    show3DPlot.show()
    plotOrientation.show()
```