UNIVERSITY OF WATERLOO

Department of Mechanical and Mechatronics Engineering

Final Exam Report
ME 546 - Multi-Sensor Data Fusion

Prepared by:

Austin W. Milne

Course Instructor:

Prof. Arash Arami

Instructional support:

Lyndon E. Tang

Mo Shushtari

Eshan Tahvilian

Abstract

This report was prepared as the Final Exam deliverable for the Winter 2024 offering of ME 546 - Multi-Sensor Data Fusion at the University of Waterloo. The report covers the design, implementation, and results of three problems: Multi Tracking Kalman Filter, Ball Tracking with Bayesian Fusion, and Neural Network Estimation of Walking Gate Ground Force.

April 19th, 2024

Contents

List of Tables

List of Figures

List of Equations

1 Problem 1 - Multi Tracking Extended Kalman Filter

Problem 1 focuses on creating a multi-fish tracking filter for a sonar based fish detector. The sensor provides range, angle, and size measurements for each fish during a measurement sweep of the sonar. The goal is to track the position and size of each fish that appears in the sonar range. An extended Kalman Filter is used to estimate the position and size of each fish.

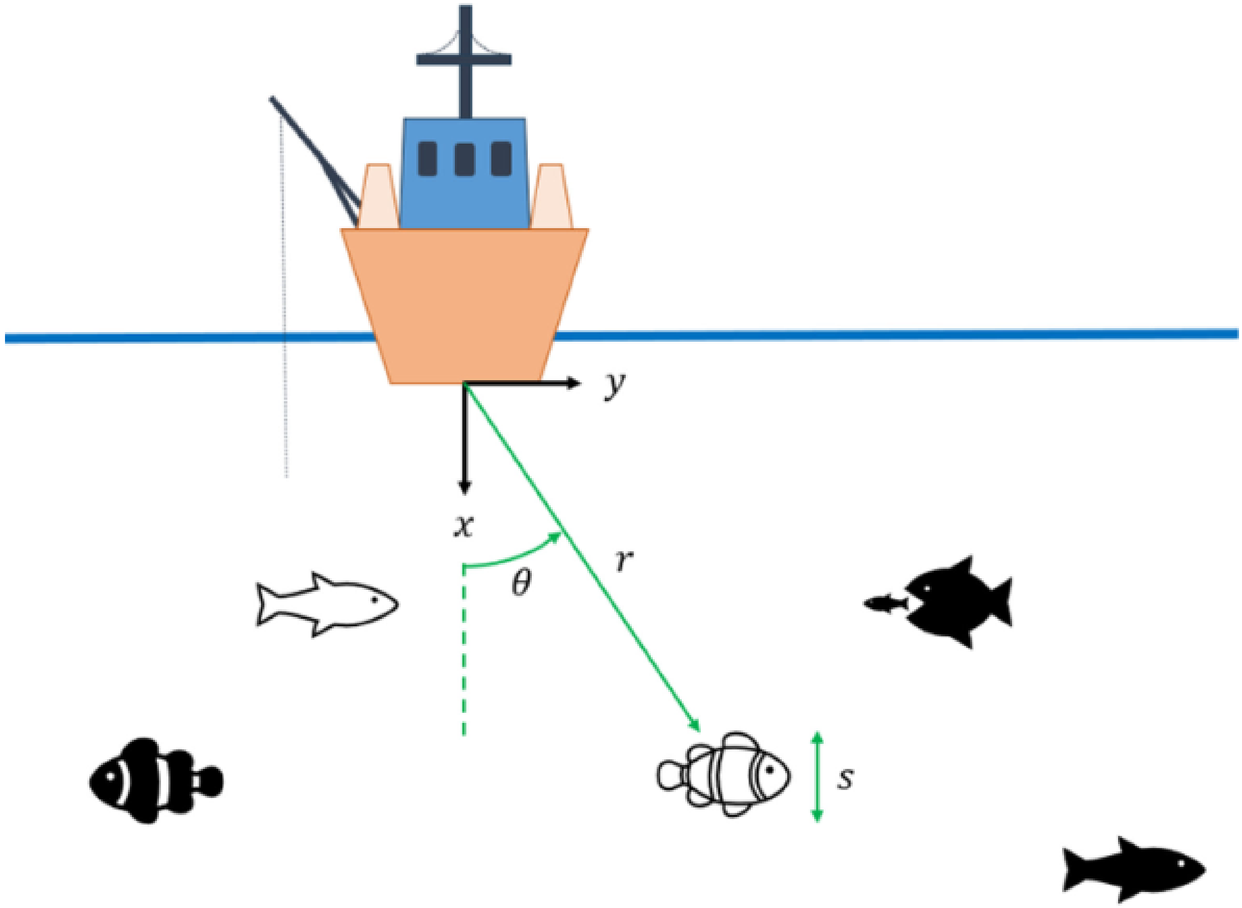


Figure 1: Sonar fish tracking system [2]

The sensor suffers from measurement noise, which is characterized by the variance of the range, angle, and size measurements. The sensor can also briefly lose "sight" of a fish and then reacquire it. The process noise is unknown, but the filter is tuned to provide the most accurate tracking results.

For this system an Extended Kalman Filter is necessary since the relation between the state and the measurements is nonlinear. Since the noise of each sensor is relative to its own axis, a coordinate transfer needs to happen within the statistical calculations. The EKF facilitates this by providing a linear approximation and a Jacobian matrix for the sensor models.

1.1 Extended Kalman Filter Design

The EKF is designed to track the position and size of each fish. In order to better predict the movement, the velocity and acceleration of each fish are also tracked. The result is a 7-state EKF Filter that tracks the position, velocity, and acceleration in the X and Y axis along with the size of the fish:

$$x = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \\ s \end{bmatrix} \tag{1}$$

With the understanding that the differential equation for the process model (F) can be described as:

$$x = Fx \tag{2}$$

$$F = \frac{\partial f(x)}{\partial x} = \begin{bmatrix} 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 0 & 1 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

1.1.1 Sensor Models

Each of the sensor conversion functions are very simple, just converting the range and angle measurements into Cartesian coordinates. The Jacobian is also computed for each sensor relative to the state vector. The models are shown below. Range sensor:

$$
h_r = r = \sqrt{x^2 + y^2}
$$

$$
H_r = \left.\frac{\partial h_r}{\partial x}\right|_x = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} \\ \frac{y}{\sqrt{x^2+y^2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4}
$$

Angle sensor:

$$
h_\theta = \theta = \tan^{-1}\left(\frac{y}{x}\right)
$$

$$
H_\theta = \left.\frac{\partial h_\theta}{\partial x}\right|_x = \begin{bmatrix} -\frac{y}{x^2+y^2} \\ \frac{x}{x^2+y^2} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5}
$$

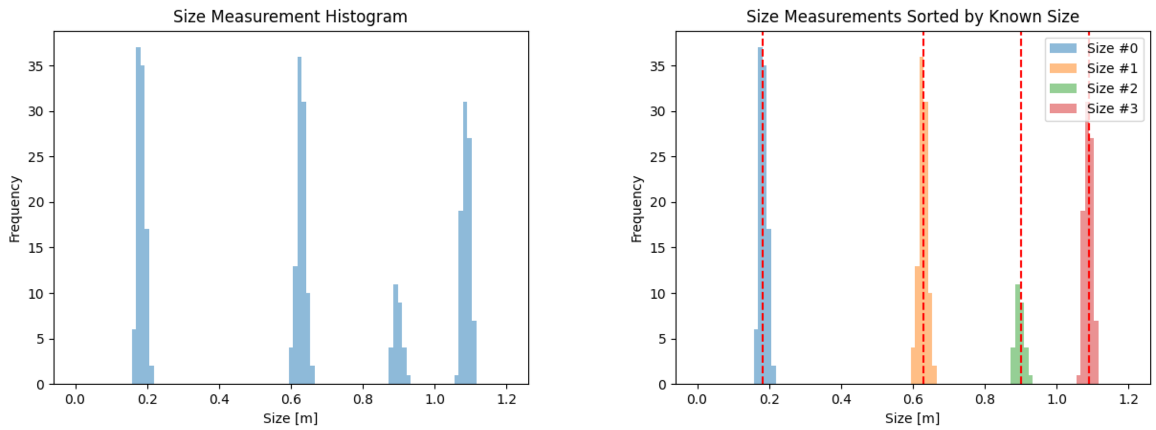
Finally, the size sensor:

$$h_s = s$$

$$H_s = \frac{\partial h_s}{\partial x}\bigg|_x = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{6}$$

1.1.2 Covariance Matrices

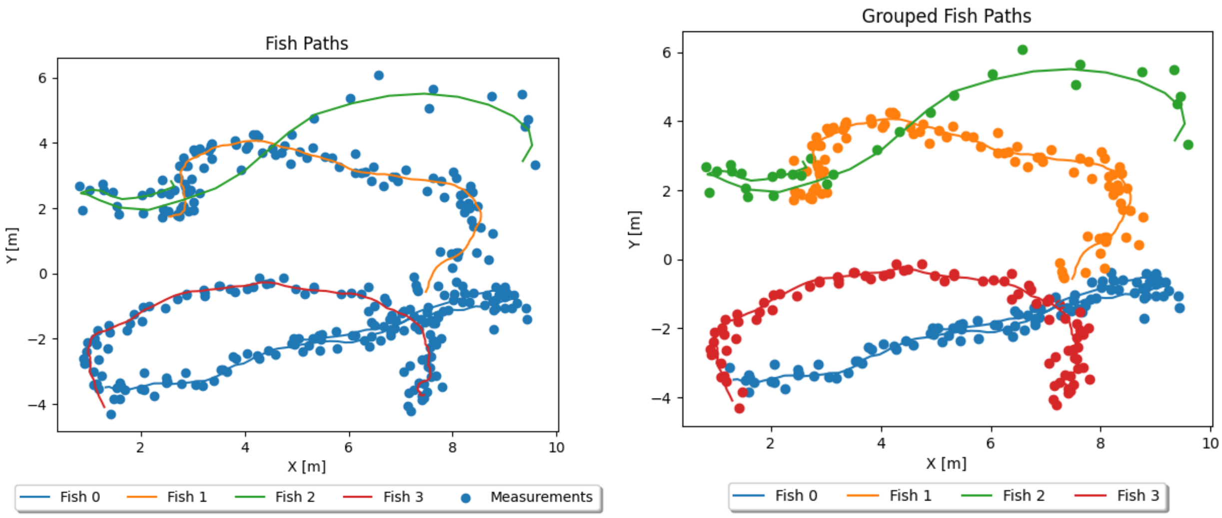
The process noise covariance matrix (Q) and the measurement noise covariance matrix (R) are crucial to creating a well-tuned Kalman Filter, allowing it to accurately determine the statistical properties of the noise in the system. The measurement noise could be directly calculated from the training data. Figures 2b and 3b show the radar sensor readings grouped with their respective fish. For readability, the position is shown in Cartesian coordinates.



(a) Reading histogram

(b) Grouped reading histogram

Figure 2: Training data size measurements



(a) Unsorted plot

(b) Sorted plot

Figure 3: Training data position measurements in Cartesian coordinates

Figures 4 and 5 then show the error distributions for the size and position measurements. The error is determined to be the distance in the relevant measurement from readout to actual. The variance of the measurements is then calculated and shown in Table 1.

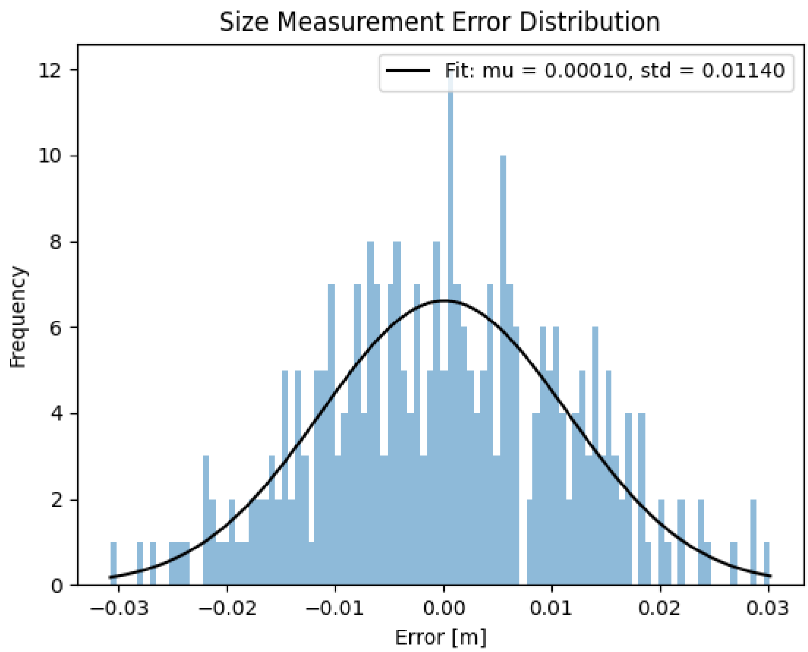
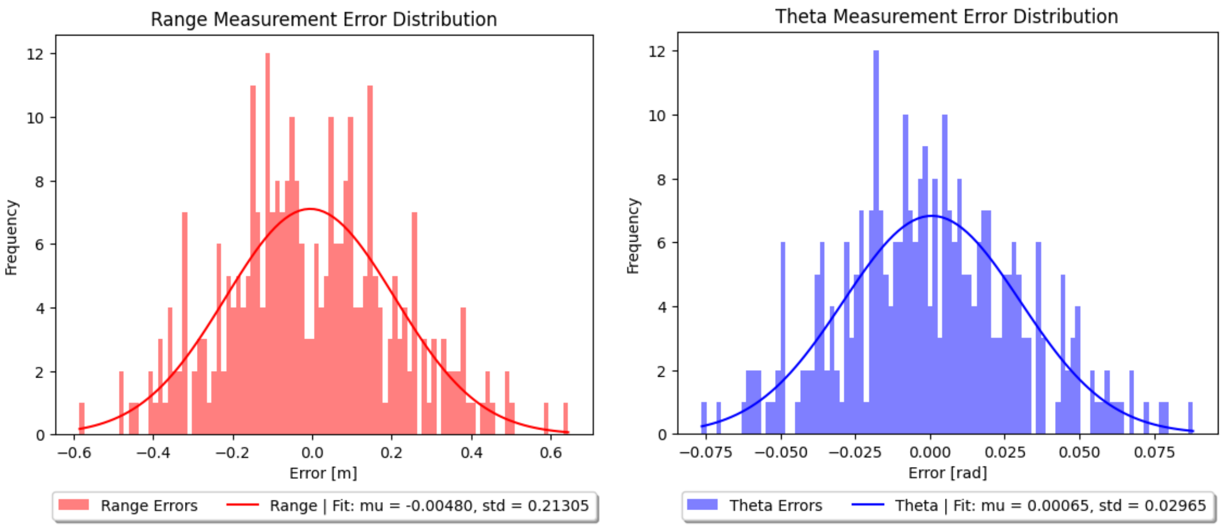


Figure 4: Training data size reading error distribution



(a) Range error distribution

(b) θ error distribution

Figure 5: Training data position measurement error distributions

Table 1: Measurement variances

Measurement	**Variance** (σ^2)
Range	0.04539078586229167
θ	0.00087889950813572
Size	0.00012988333066048

The measurement noise covariance matrix is then decided to be the diagonal matrix of the variances of the measurements:

$$R = \begin{bmatrix} 0.0453\ldots & 0 & 0 \\ 0 & 0.0008\ldots & 0 \\ 0 & 0 & 0.0001\ldots \end{bmatrix} \tag{7}$$

The process noise covariance matrix is more complicated to determine. As an alternative to the traditional approach of trying to profile the process noise from analysis of the reference truth data, the process noise is instead tuned to provide the best tracking results.

The process noise matrix is determined to be a diagonal matrix with the following values:

$$Q = \begin{bmatrix} \sigma^2_{x,y} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma^2_{x,y} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma^2_{\dot{x},\dot{y}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma^2_{\dot{x},\dot{y}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma^2_{\ddot{x},\ddot{y}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma^2_{\ddot{x},\ddot{y}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sigma^2_s \end{bmatrix} \tag{8}$$

The model with the Q matrix was run against the training data and the average root mean squared error (RMSE) was calculated. The $\sigma^2_{x,y}$, $\sigma^2_{\dot{x},\dot{y}}$, $\sigma^2_{\ddot{x},\ddot{y}}$, and σ^2_s values were then tuned using a differential evolution optimization algorithm [5] to minimize the RMSE. This was done for the entire known path of fishes 0 and 1 in the training data. The final values where averaged from fishes 0 and 1 and the resulting values are shown in Table 2.

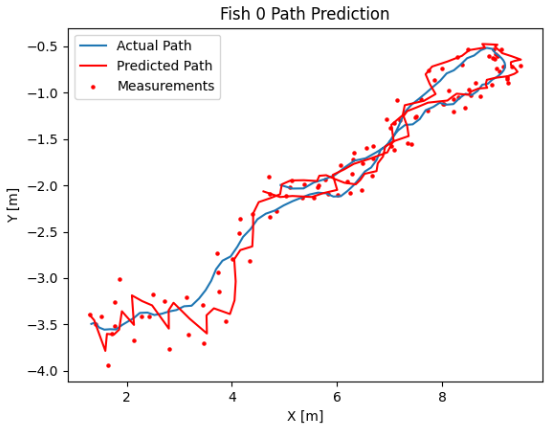
Table 2: Tuned process noise variances

Process Noise	**Variance** (σ^2)
$\sigma^2_{x,y}$	0.0011501322599521454
$\sigma^2_{\dot{x},\dot{y}}$	0.06870244206669615
$\sigma^2_{\ddot{x},\ddot{y}}$	0.0
σ^2_s	$1.5510976521855614 \times 10^{-07}$

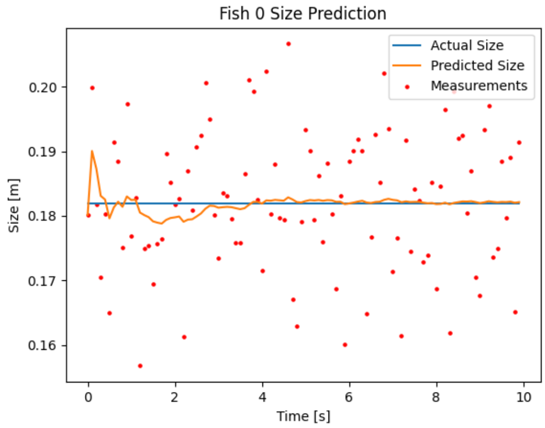
The resultant process noise covariance matrix is then:

$$Q = \begin{bmatrix} 0.00115\ldots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.00115\ldots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.06870\ldots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.06870\ldots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.55109\cdots \times 10^{-07} \end{bmatrix} \tag{9}$$

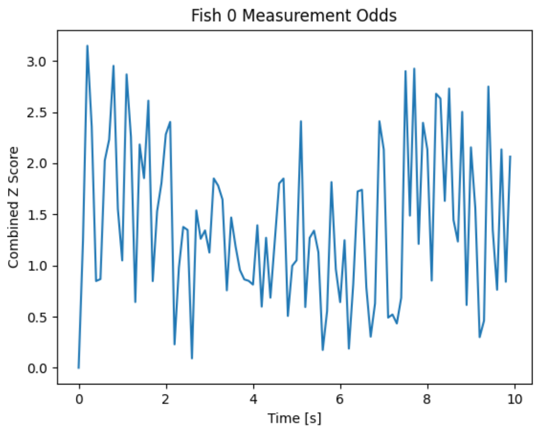
To ensure that the process noise covariance matrix is reasonable, the filter was run against the training data of fish 0. The resultant path, size, and z-score calculation (important for section 1.3) are shown in figure 6. The plot shows reasonably accurate tracking of the fish.



(a) Path Prediction



(b) Error Prediction



(c) Path Prediction

Figure 6: Test fish 0 prediction

1.2 Extended Kalman Filter Implementation

The extended kalman filter (EKF) was implemented using an existing EKF class from the FilterPy library [4]. The EKF class provides the basic update a predict functions for the filter as well as storage for the current state and covariance matrices. The EKF class was wrapped in a custom EKF class to handle other useful functions. The most useful functions implemented are the `z_score_of_measurement_match`, `score_pos_accuracy`, and `score_size_accuracy` functions. The `z_score_of_measurement_match` function calculates the z-score of a given measurement as the next value of the filter's series. This is the sum of the X and Y axis z-scores. This is used in section 1.3 for assigning measurements to tracks. The `score_pos_accuracy` and `score_size_accuracy` functions calculate the accuracy of the position and size predictions from a set of data inputs. These are used in section 1.1.2 to optimize the process noise covariance matrix. The source code for the EKF is included in Appendix B and in the additional resources of this report submission.

1.3 Multi Fish Tracking

The multi-fish tracking algorithm is implemented by running the EKF on each fish in the sonar sweep. Due to the errors and irregularities in the data as outlined above, some logical operations are needed to assign sensor readings to the correct fish's EKF. The algorithm is as follows:

1. For every combination of existing fish and new measurements:
   (a) Predict the next EKF size
   (b) Calculate the z-score of the size measurement match
   (c) if the size z-score is above a threshold: Store a position z-score of inifinity
   (d) else:
      i. Predict the next EKF position
      ii. Calculate the z-score of the position measurement match
      iii. Store the position z-score
2. Until there are no position z-score left below the threshold
   (a) Find the lowest z-score
   (b) Update the relevant fish with the relevant measurement
3. For every measurement that has not been assigned to a fish:
   (a) Create a fish tracker EKF
   (b) Initialize the EKF with the measurement
4. For every fish that has not been assigned a measurement:
   (a) Predict the next EKF position
   (b) Increment the fish's count of missed measurements
5. If a fish has missed too many measurements:
   (a) Remove the tracker

(b) Remove positions after loss of signal from position history

This combination of logic allows the filter to accurately track multiple fish in the sonar sweep while overlooking missed measurements and allowing for fish to appear and disappear independently.

Manual tuning of the parameters such as z-score limits and missed measurement limits was necessary to ensure the filter was tracking accurately. The final parameters are available in the source code in Appendix B. The final results of running the multitracking filter on the training data are shown in figure 7.

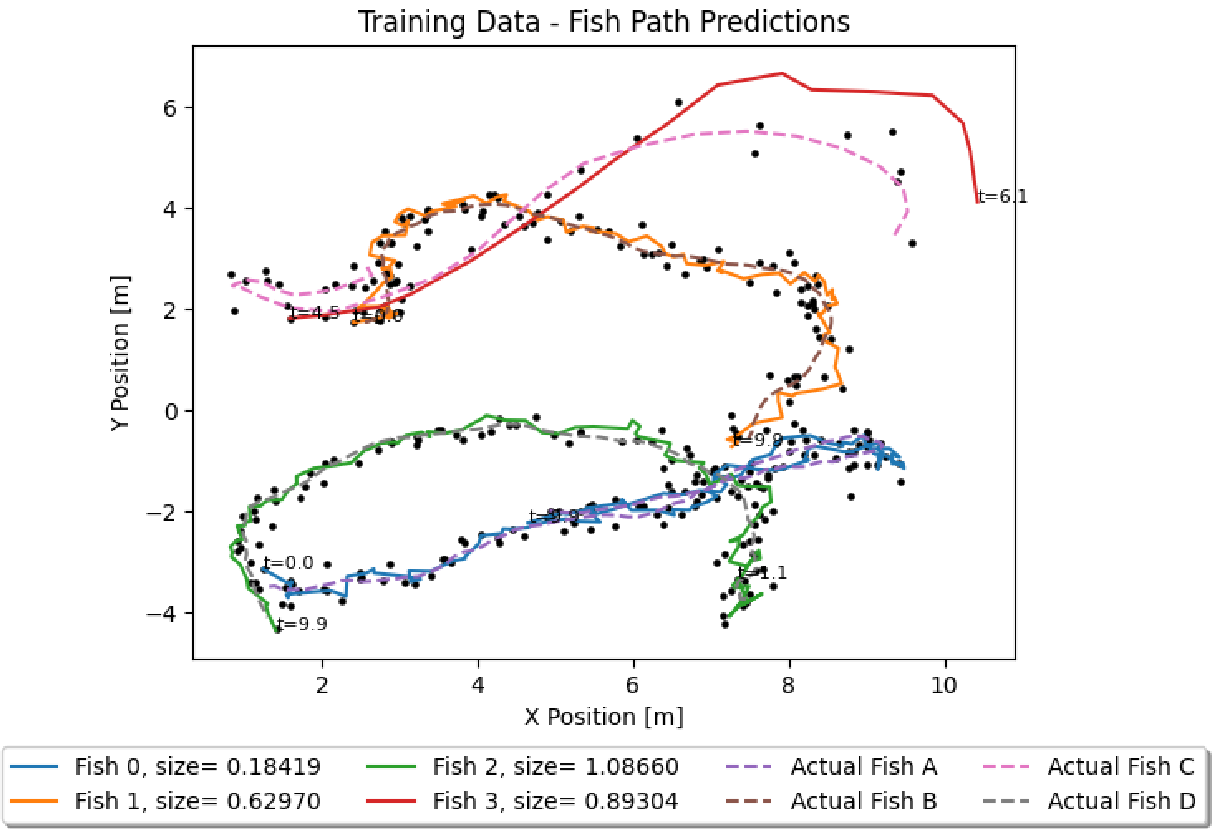


Figure 7: Training data multi fish tracking prediction compared to real paths

The tracking is satisfactorily accurate, with the fish paths closely matching the real paths. Predicted fish 3, corresponding to actual fish C, is not tracked as accurately as the other fish. This is likely due to the more rapid movement and higher acceleration compared to the other fish. Fish 3-C moves faster than fish 0 and fish 1 that were used for the process noise tuning in section 1.1.2. This would lead to the filter being more sluggish to update the acceleration component of the state vector. The result is the overshoot in the region of $(7 < x < 11, 4 < y < 7)$. Performance could likely be improved by manually tuning the process past the existing state, specifically in these types of edge cases.

1.4 Fish Tracking Results

Finally, the filter was applied to the test data. The results are shown in figure 8. The final positions and sizes of the fish are shown in table 3. The final covariance matrices for each fish are shown in table 4. While the position tracks are not smooth, they do a decent job of following the fishs' paths. The size predictions are expected to be very accurate, as they proved to be with tests in the training data.

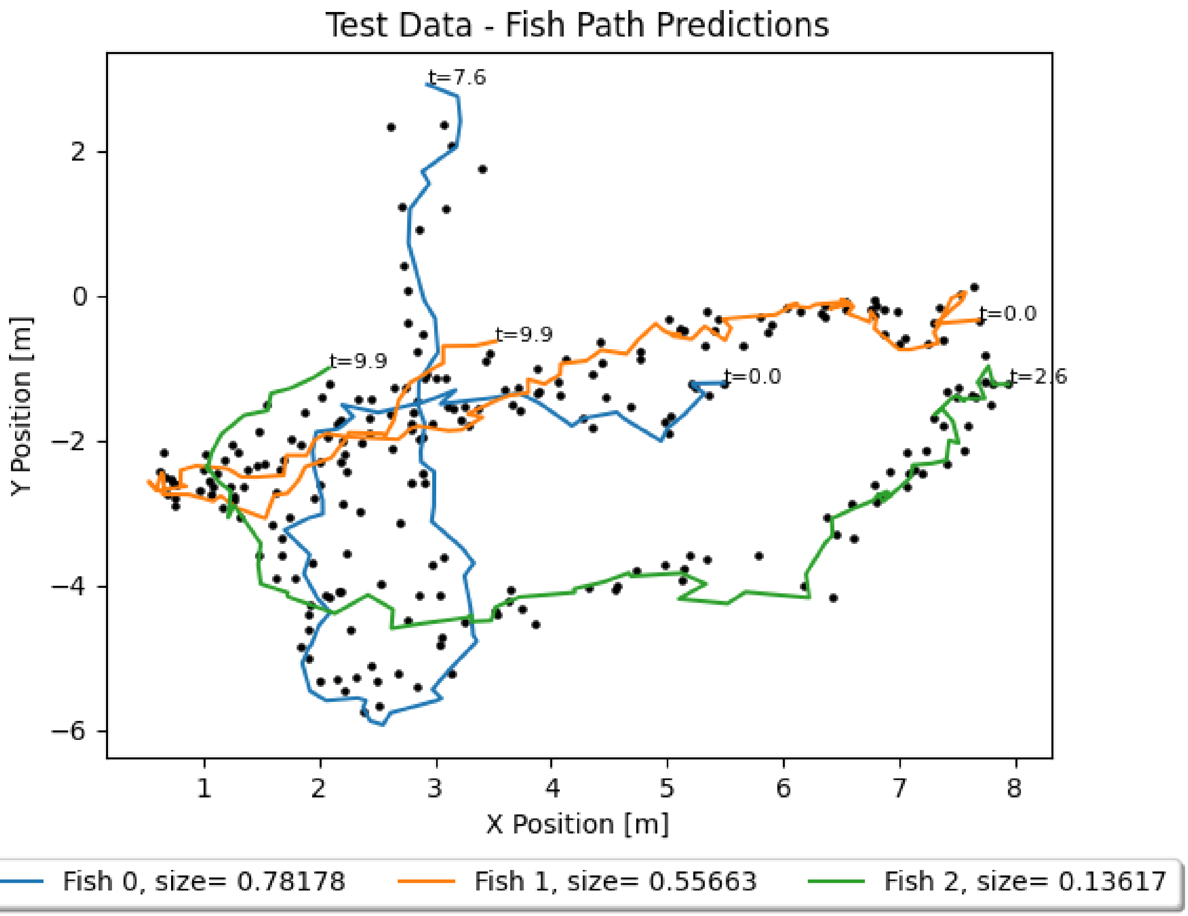


Figure 8: Test data multi fish tracking prediction

Table 3: Fish tracking results

Fish #	Size	Time Start	Time End	Final X [m]	Final Y [m]
Fish 0	+0.7813	+0.0000	+7.6000	+2.9447	+2.9158
Fish 1	+0.5564	+0.0000	+9.9000	+3.5141	-0.6288
Fish 2	+0.1361	+2.6000	+9.9000	+2.1045	-1.0043

The covariance matrices (table 4) show that the filter is confident in the position and size of the fish. The position variance being in the range of 0.015 to 0.031 and the size variance being less than 0.001 for all fish. For comparison, figure 9 shows the error distribution of the raw position measurements if converted to Cartesian coordinates. The raw measurements provide a position variance of around 0.068, meaning that the filter has about have the uncertainty of the raw measurements, in addition to multitarget tracking functionality.

Table 4: Fish tracking covariance matrices

Fish #	**Final Covariance Matrix**
Fish 0	$\begin{bmatrix} +0.026 & +0.007 & +0.049 & +0.010 & +0.007 & +0.001 & +0.000 \\ +0.007 & +0.023 & +0.009 & +0.045 & +0.001 & +0.006 & +0.000 \\ +0.049 & +0.009 & +0.259 & +0.016 & +0.034 & +0.002 & +0.000 \\ +0.010 & +0.045 & +0.016 & +0.248 & +0.002 & +0.033 & +0.000 \\ +0.007 & +0.001 & +0.034 & +0.002 & +0.052 & +0.000 & +0.000 \\ +0.001 & +0.006 & +0.002 & +0.033 & +0.000 & +0.052 & +0.000 \\ +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 \end{bmatrix}$
Fish 1	$\begin{bmatrix} +0.031 & -0.004 & +0.055 & -0.005 & +0.006 & -0.000 & +0.000 \\ -0.004 & +0.015 & -0.005 & +0.033 & -0.001 & +0.003 & +0.000 \\ +0.055 & -0.005 & +0.264 & -0.010 & +0.027 & -0.001 & +0.000 \\ -0.005 & +0.033 & -0.010 & +0.225 & -0.001 & +0.023 & +0.000 \\ +0.006 & -0.001 & +0.027 & -0.001 & +0.039 & -0.000 & +0.000 \\ -0.000 & +0.003 & -0.001 & +0.023 & -0.000 & +0.039 & +0.000 \\ +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 \end{bmatrix}$
Fish 2	$\begin{bmatrix} +0.026 & -0.009 & +0.049 & -0.013 & +0.007 & -0.002 & +0.000 \\ -0.009 & +0.016 & -0.015 & +0.033 & -0.002 & +0.005 & +0.000 \\ +0.049 & -0.015 & +0.255 & -0.026 & +0.035 & -0.004 & +0.000 \\ -0.013 & +0.033 & -0.026 & +0.231 & -0.004 & +0.032 & +0.000 \\ +0.007 & -0.002 & +0.035 & -0.004 & +0.055 & -0.000 & +0.000 \\ -0.002 & +0.005 & -0.004 & +0.032 & -0.000 & +0.054 & +0.000 \\ +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 & +0.000 \end{bmatrix}$

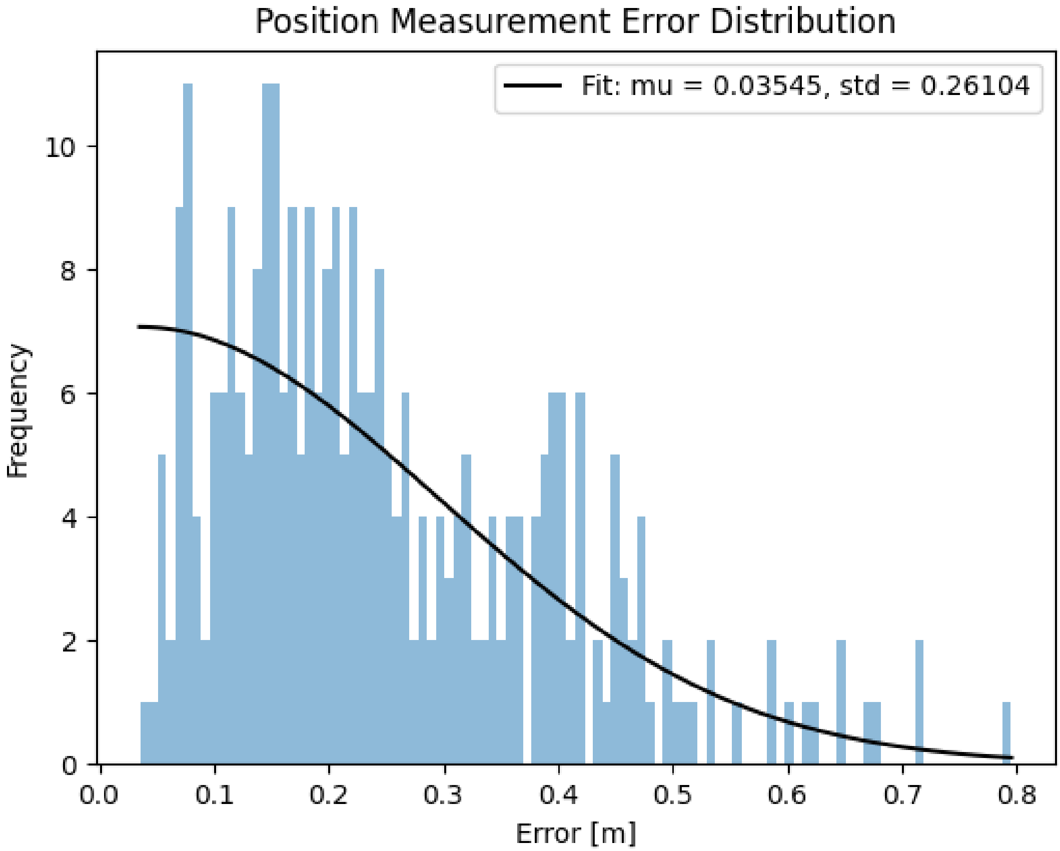


Figure 9: Training data cartesian position error distribution

As requested in final exam document [2], a CSV file is provided that contains the count of number of tracked fish at each time step. The count of fish over time is also shown in figure 10.

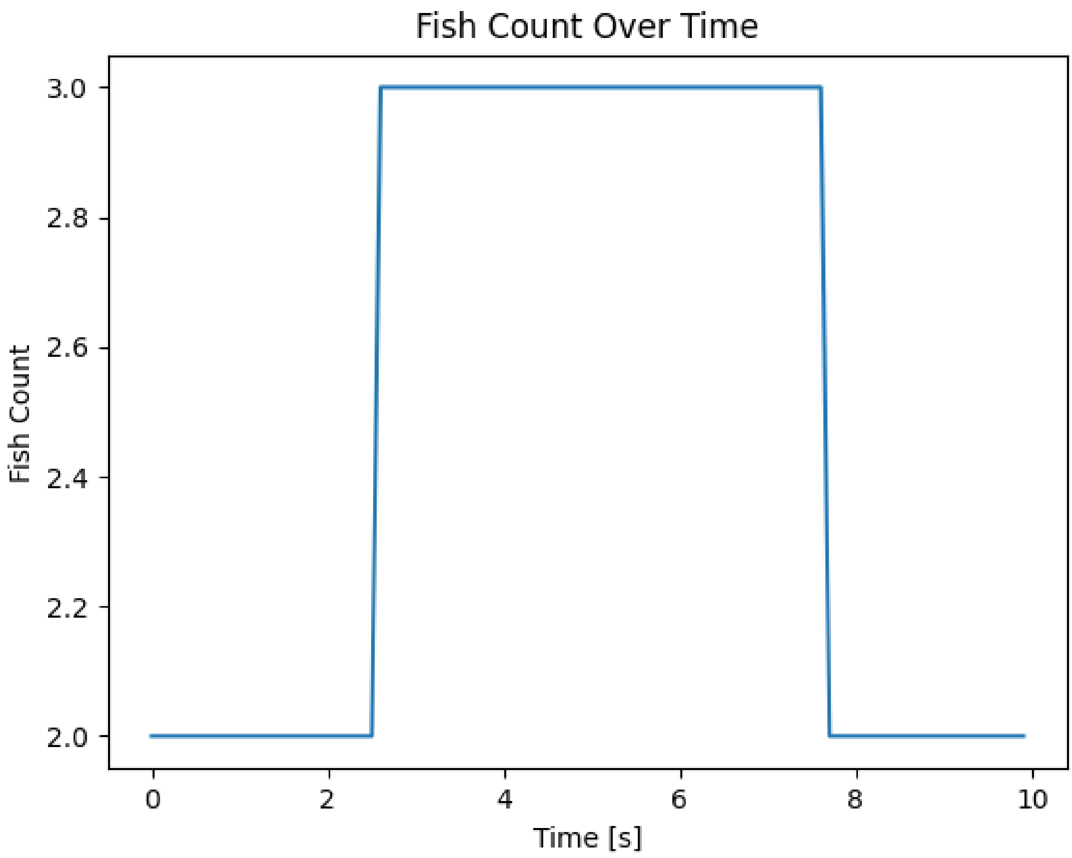


Figure 10: Fish count over time

2 Problem 2 - Ball Tracking with Bayesian Fusion

Problem 2 focuses on using a Bayesian filter to predict the shot placement of a ball after it has been kicked at specific contact patch or a mixture of contact patches. Figure 11 shows the reference frame for the ball and the relevant contact patches.

Figure 11: Reference frame for ball contact and tracking [2]

Figure 12 shows the scatter plot of kick data provided. As is to be expected, the data is varied, but shows a trend of placement being in the opposite Y and Z direction from the contact patch. The data is labelled by color for each contact patch.

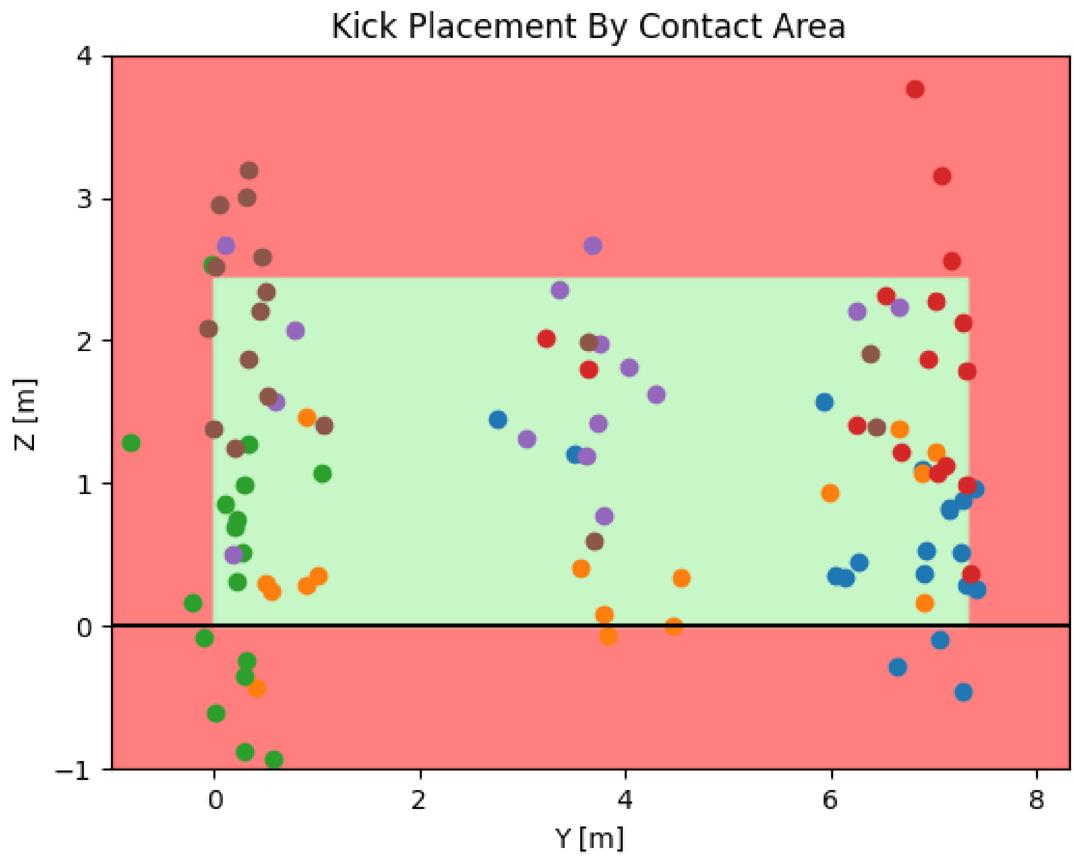
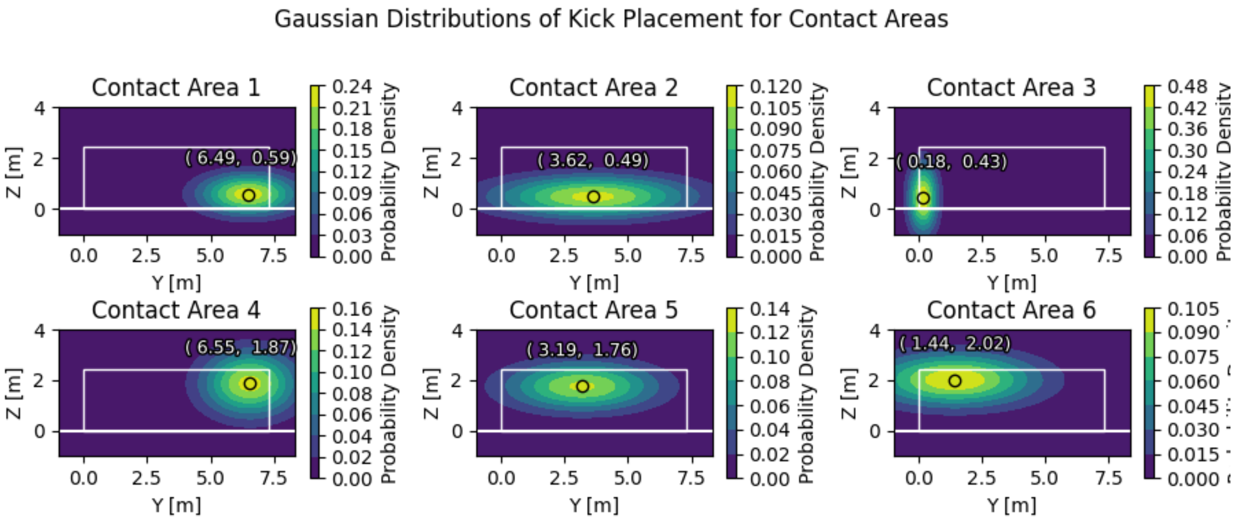


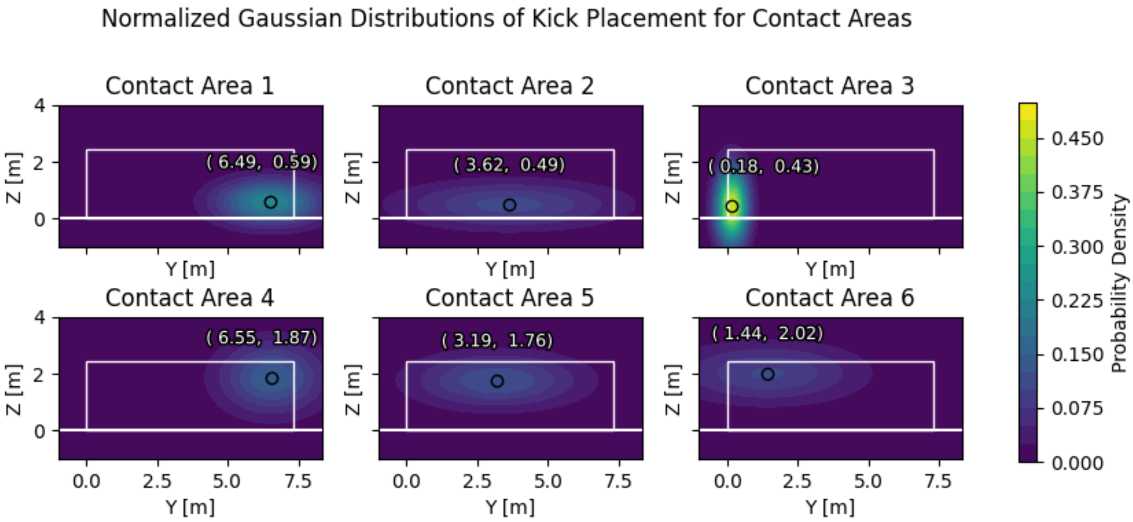
Figure 12: Kick placement scatter organized by contact area

2.1 Contact Area Probability

With the data provided, a probability field was created for a kick placed at each contact patch. Figure 13a shows the individual distributions for each contact patch, while figure 13b shows the normalized distributions. From the figured, it is evident that kicks from contact patch 3 are the most consistent, with contact patch 1 being the next most consistent and contact patch 2 having the most varied results. Table 5 shows the mean and variance of the distributions for each contact patch.



(a) Individual distributions



(b) Normalized distributions

Figure 13: Kick result probability distributions by contact area

Table 5: Contact area probability statistics

Contact Area	**Y Mean** [m]	**Y Variance**	**Z Mean** [m]	**Z Variance**
1	6.4937	1.6403	0.5852	0.2991
2	3.6229	6.7131	0.4883	0.3116
3	0.1790	0.1422	0.4347	0.8120
4	6.5535	1.5737	1.8657	0.7385
5	3.1949	3.9846	1.7620	0.4180
6	1.4352	4.7952	2.0183	0.4953

2.2 Bayesian Fusion

Since each contact patch has its own probability field, the Bayesian fusion can be used to combine the information from each contact patch with the influence of a weighting factor. Typically, the inverse of the variance of the distribution is used as the weighting factor. In this case, since the certainty of each contact patch is also provided, the weighting factor is the certainty of the contact patch multiplied by the inverse of the variance. This changes the mean and variance calculations of the Bayesian fusion to the following:

$$y_{ws} = \frac{\sum_{i=1}^{6} \frac{w_i}{\sigma_i^2} y_i}{\sum_{i=1}^{6} \frac{w_i}{\sigma_i^2}} \tag{10}$$

$$\sigma_{ws}^2 = \sum_{i=1}^{6} \left(\frac{\frac{w_i}{\sigma_i^2}}{\sum_{i=1}^{6} \frac{w_i}{\sigma_i^2}} \right)^2 \sigma_i^2 \tag{11}$$

Applying these calculations across the contact patches and on both the Y and Z axis should provided a most likely (Y, Z) position along with a variance for the prediction.

2.3 Ball Tracking Results

Applying equations 10 and 11 to the contact patch distributions and weights, the results are shown in figure 14. The results show the kick to most likely land at $(4.237, 1.700)$ with a variance of $(0.592, 0.216)$. Observing the distribution plot, it is clear that the kick will most likely fall within the bounds of the net.

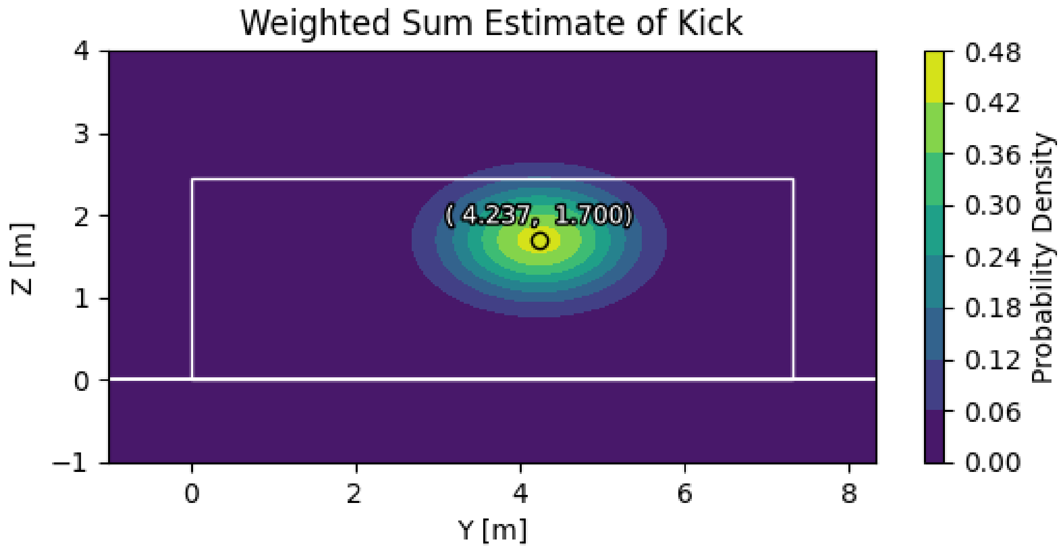


Figure 14: Kick prediction results

3 Problem 3 - Neural Network Estimation of Walking Gate Ground Force

Problem 3 focuses on using a Neural Network to estimate the ground reaction forces produced by a person walking. The input data is an array of 50 different measurements on the person's body. The goal is to predict the ground reaction force produced by the person instead of needing expensive and particular equipment to determine those values. The output is a 4 value vector representing the forces in the Y and Z directions from each of the left and right feet.

3.1 Neural Network Design

A rather simple approach was taken to the neural network design. The hope was that the neural network training would decide what values were important to the prediction and ignore whatever of the 50 values was not very relivant. The neural network was designed with 2 hidden layers, each with 100 nodes. Each hidden node used a sigmoid activation function since this is a relatively shallow network. The output layer used a linear activation function to allow for the output to be a continuous value. The rough architecture of the neural network is shown in figure 15.

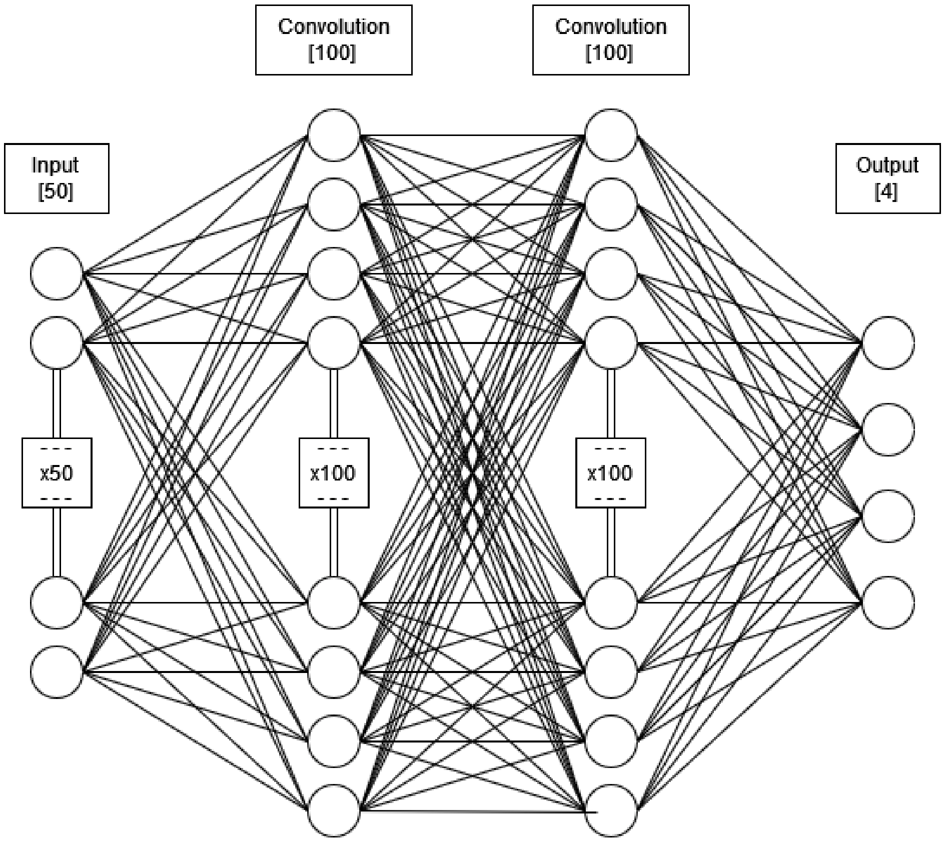


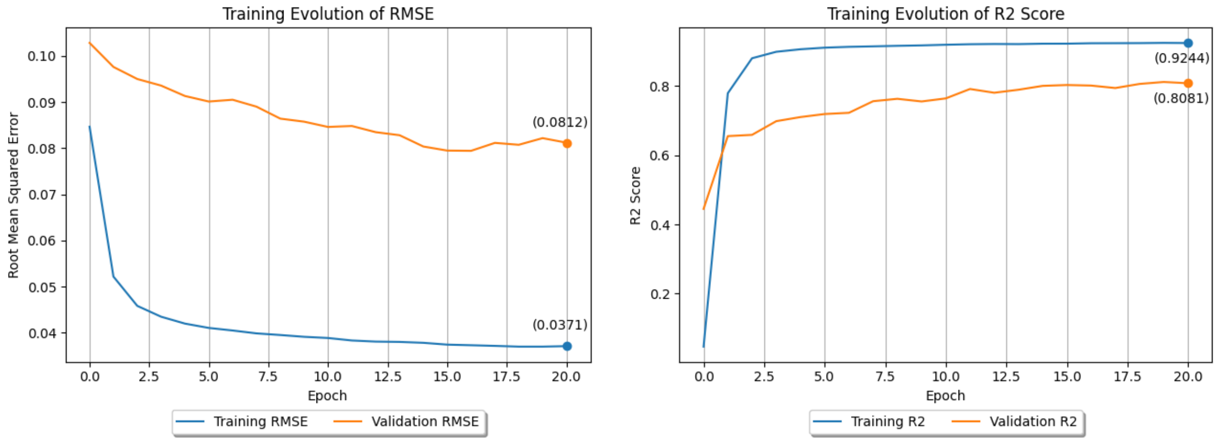
Figure 15: Neural network architecture

3.2 Neural Network Implementation

The neural network was implemented in Python using TensorFlow [1] and Keras [3], specifically the Keras models computed in TensorFlow. The code for the neural network is shown in Appendix D.

3.3 Neural Network Training

The neural network was trained on subjects 1-8 of the provided data. Subject 9 was used as a validation set to ensure the neural network was not overfitting. Training was done in batches of 50 to vastly improve the speed of training. The neural network was trained for 100 epochs, with the possibility of breaking training early if there was a stagnation in the reduction of the root mean squared error. After each epoch, the model was tested against the validation set using the metrics of root mean squared error (RMSE) and the coefficient of determination (R^2). The training history is shown in figure 16.



(a) Root mean square error (RMSE) (b) R^2

Figure 16: Training history

As to be expected, the improvement is rapid in the early epochs and slows toward the en. Due to the stagnation in the change of RMSE, the training was stopped at epoch 20. The final RMSE and R^2 values are shown in table 6.

Table 6: Training results

Metric	Training Set	Validation Set
RMSE	0.0371	0.0812
R^2	0.9244	0.8081

The training shows a significant improvement in the R^2 over the course of the 20 epochs. The RMSE is also quite low, showing that the neural network is able to predict the ground reaction forces with a high degree of accuracy.

3.4 Neural Network Results

The neural network was then used to predict the ground reaction forces of subject 10 for all 2000 measurements. As requested, a CSV file was prepared with the output of the neural network. The results are plotted in figure 17 in Appendix A. The results show a reasonable estimation of the forces produced by the subject. The gate appears consistent and force scale between the left and right feet is also consistent.

4 References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Arash Arami et al. Mte-546 multi-sensor data fusion final exam - winter 2024. Technical report, The University of Waterloo, april 2024.

[3] François Chollet et al. Keras. *https://github.com/fchollet/keras*, 2015.

[4] Roger R Labbe Jr. Filterpy v1.4.5. August 2018.

[5] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

Appendices

Appendix A Gate Reaction Force Estimation

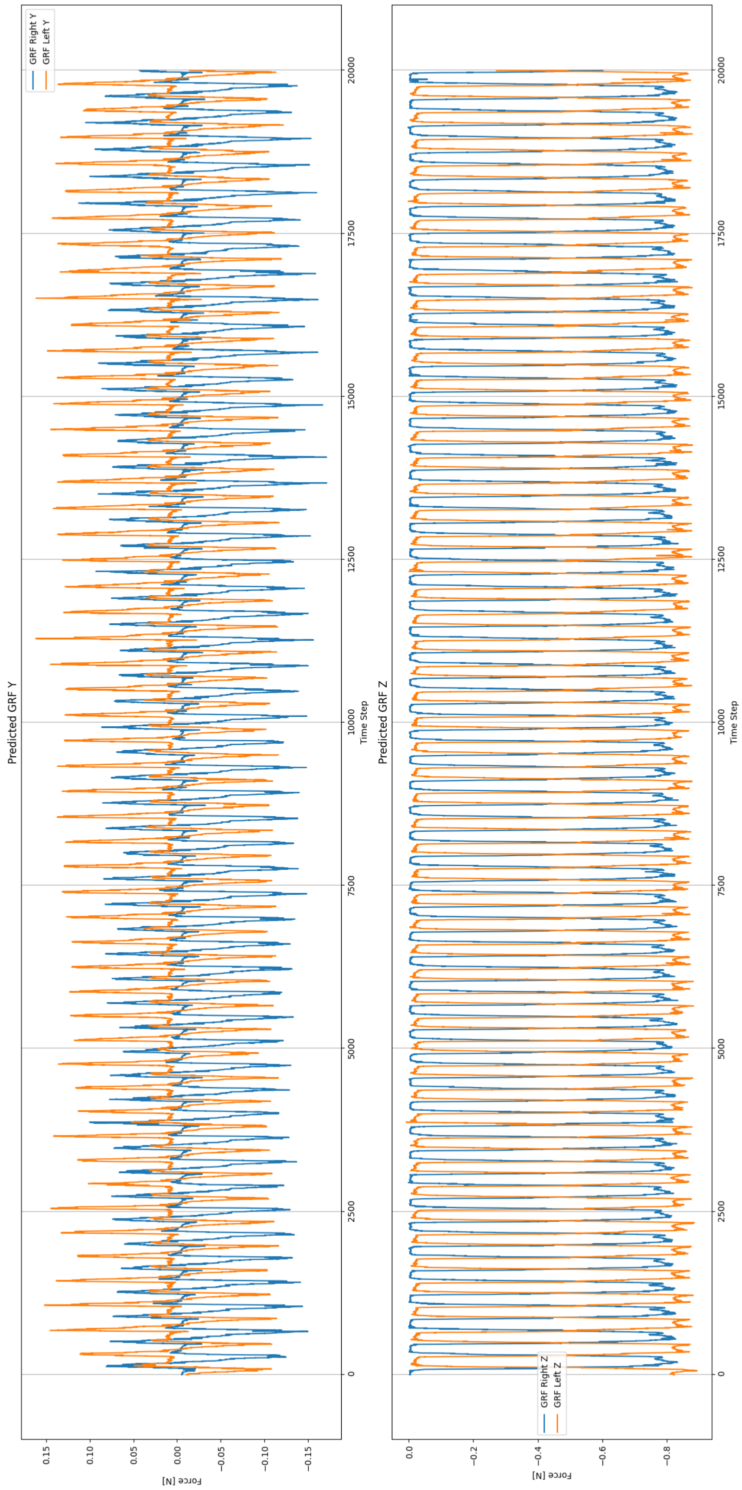


Figure 17: Gate reaction force estimation

Appendix B Problem 1 Source Code

```python
# %% [markdown]
# # Problem #1
#
# ## Data Ingest and Formatting

# %%
# Common Imports
import csv
import pathlib
import statistics
import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
from prettytable import PrettyTable
from scipy.stats import norm, halfnorm
from scipy.optimize import differential_evolution
from filterpy.kalman import ExtendedKalmanFilter

# Ensure the plot output directory exists
pathlib.Path('./out').mkdir(parents=True, exist_ok=True)

# Read in the training data, training labels, and test data
training_data = pd.read_csv('./data/ekf_training_data.csv', sep=",", skipinitialspace=True)
training_labels = pd.read_csv('./data/ekf_training_labels.csv', sep=",", skipinitialspace=True)
test_data = pd.read_csv('./data/ekf_test_data.csv', sep=",", skipinitialspace=True)

# Rename Columns for code convenience
training_data.columns = ['Time', 'dist_0', 'angle_0', 'size_0',
                                 'dist_1', 'angle_1', 'size_1',
                                 'dist_2', 'angle_2', 'size_2',
                                 'dist_3', 'angle_3', 'size_3']
training_labels.columns = ['Time', 'x_0', 'y_0', 's_0',
                                   'x_1', 'y_1', 's_1',
                                   'x_2', 'y_2', 's_2',
                                   'x_3', 'y_3', 's_3']
test_data.columns = ['Time', 'dist_0', 'angle_0', 'size_0',
                             'dist_1', 'angle_1', 'size_1',
                             'dist_2', 'angle_2', 'size_2']

# Combine the training data and labels into one dataset
training_combined = training_data.merge(training_labels, on='Time', how='outer')

# Output the list of columns for each dataset
print("Training Data Columns: ", list(training_data.columns))
print("Training Labels Columns: ", list(training_labels.columns))
print("Training Combined Columns: ", list(training_combined.columns))
print("Test Data Columns: ", list(test_data.columns))

# %% [markdown]
# ## Sensor Variance

# %% [markdown]
# ### Size Measurement Variance

# %%
# Plot a histogram of all the size measurements
measurements = pd.concat([
    training_data.iloc[:,3],
    training_data.iloc[:,6],
    training_data.iloc[:,9],
    training_data.iloc[:,12]
])
plt.figure()
bins = np.linspace(0, 1.2, 100)
plt.hist(measurements, bins=bins, alpha=0.5, label='Fish 0')
```

```python
plt.title('Size Measurement Histogram')
plt.xlabel('Size [m]')
plt.ylabel('Frequency')
plt.savefig('out/p1_size_hist.png')
plt.show()

# %%
# Group size measurement with closest known size
known_sizes = [
    training_labels.loc[pd.notnull(training_labels['s_0']), 's_0'].values[0],
    training_labels.loc[pd.notnull(training_labels['s_1']), 's_1'].values[0],
    training_labels.loc[pd.notnull(training_labels['s_2']), 's_2'].values[0],
    training_labels.loc[pd.notnull(training_labels['s_3']), 's_3'].values[0],
]
sorted_measurements = {}
for size in known_sizes:
    sorted_measurements[size] = []
for measurement in measurements:
    if np.isnan(measurement):
        continue
    closest_size = min(known_sizes, key=lambda x:abs(x-measurement))
    sorted_measurements[closest_size].append(measurement)

# Plot the grouped size measurements
plt.figure()
bins = np.linspace(0, 1.2, 100)
for i, size in enumerate(known_sizes):
    plt.hist(sorted_measurements[size], bins=bins, alpha=0.5, label=f'Size #{i}')
    plt.axvline(x=size, color='r', linestyle='--')
plt.title('Size Measurements Sorted by Known Size')
plt.xlabel('Size [m]')
plt.ylabel('Frequency')
plt.legend()
plt.savefig('out/p1_sorted_size_hist.png')
plt.show()


# %%
# Calculate the variance of the size measurement
errors = []
for size in known_sizes:
    for measurement in sorted_measurements[size]:
        errors.append(measurement - size)

# Plot the error distribution
plt.figure()
hist, bins, _ = plt.hist(errors, bins=100, alpha=0.5)
xmin, xmax = plt.xlim()
mu, std = norm.fit(errors)
p = norm.pdf(bins, mu, std)
plt.plot(bins, p/p.sum() * len(errors), 'k', label="Fit: mu = %.5f, std = %.5f" % (mu, std))
plt.title('Size Measurement Error Distribution')
plt.xlabel('Error [m]')
plt.ylabel('Frequency')
plt.legend()
plt.savefig('out/p1_size_error_hist.png')
plt.show()

# Calculate the variance of the size measurement
size_variance = np.var(errors)
print(f"Size Measurement Variance: {size_variance}")

# %% [markdown]
# ### Position Measurement Variances

# %%
# Combine all the position measurements into one dataset
positions = pd.DataFrame({
```

```
    'r': pd.concat([
        training_data.iloc[:,1],
        training_data.iloc[:,4],
        training_data.iloc[:,7],
        training_data.iloc[:,10]
    ]),
    't': pd.concat([
        training_data.iloc[:,2],
        training_data.iloc[:,5],
        training_data.iloc[:,8],
        training_data.iloc[:,11]
    ]),
    'time': pd.concat([
        training_data['Time'],
        training_data['Time'],
        training_data['Time'],
        training_data['Time']
    ])
})
positions = positions.dropna().reset_index(drop=True)
positions['x'] = positions['r'] * np.cos(np.deg2rad(positions['t']))
positions['y'] = positions['r'] * np.sin(np.deg2rad(positions['t']))

# Plot the known paths of the fish and scatter the measurements over them
plt.figure()
plt.plot(training_labels['x_0'], training_labels['y_0'], label='Fish 0')
plt.plot(training_labels['x_1'], training_labels['y_1'], label='Fish 1')
plt.plot(training_labels['x_2'], training_labels['y_2'], label='Fish 2')
plt.plot(training_labels['x_3'], training_labels['y_3'], label='Fish 3')
plt.scatter(positions['x'], positions['y'], label='Measurements')
plt.title('Fish Paths')
plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
           fancybox=True, shadow=True, ncol=5)
plt.savefig('out/p1_pos_plot.png', bbox_inches='tight')
plt.show()


# %%
# Try grouping position readings by nearest known fish position
grouped_positions = {}
for i in range(4):
    grouped_positions[i] = []
for index, row in positions.iterrows():
    time = row['time']
    distances = [
        np.linalg.norm([row['x'] - training_labels.loc[training_labels['Time']==time, f'x_{i
            }'],
                        row['y'] - training_labels.loc[training_labels['Time']==time, f'y_{i
                            }']])
        for i in range(4)
    ]
    distances = [d if not np.isnan(d) else np.inf for d in distances]
    closest_fish = np.argmin(distances, )
    grouped_positions[closest_fish].append(row)

# Plot the grouped position readings
plt.figure()
for i in range(4):
    fish_positions = pd.DataFrame(grouped_positions[i])
    plt.scatter(fish_positions['x'], fish_positions['y'], color=f'C{i}')
    plt.plot(training_labels[f'x_{i}'], training_labels[f'y_{i}'], label=f'Fish {i}', color=
        f'C{i}')
plt.title('Grouped Fish Paths')
plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
```

```
            fancybox=True, shadow=True, ncol=5)
plt.savefig('out/p1_sorted_pos_plot.png', bbox_inches='tight')
plt.show()

# %%
# Calculate the variance of the position measurements
errors = []
for i in range(4):
    for position in grouped_positions[i]:
        time = position['time']
        x = position['x']
        y = position['y']
        known_x = training_labels.loc[training_labels['Time']==time, f'x_{i}'].values[0]
        known_y = training_labels.loc[training_labels['Time']==time, f'y_{i}'].values[0]
        errors.append(np.linalg.norm([x-known_x, y-known_y]))

# Plot the error distribution
plt.figure()
hist, bins, _ = plt.hist(errors, bins=100, alpha=0.5)
xmin, xmax = plt.xlim()
mu, std = halfnorm.fit(errors)
p = halfnorm.pdf(bins, mu, std)
plt.plot(bins, p/p.sum() * len(errors), 'k', label="Fit: mu = %.5f, std = %.5f" % (mu, std))
plt.title('Position Measurement Error Distribution')
plt.xlabel('Error [m]')
plt.ylabel('Frequency')
plt.legend()
plt.savefig('out/p1_pos_error_hist.png')
plt.show()

# Plot scatter of Abs error vs distance from (0, 0)
errors = []
for i in range(4):
    for position in grouped_positions[i]:
        time = position['time']
        x = position['x']
        y = position['y']
        known_x = training_labels.loc[training_labels['Time']==time, f'x_{i}'].values[0]
        known_y = training_labels.loc[training_labels['Time']==time, f'y_{i}'].values[0]
        errors.append((np.linalg.norm([known_x, known_y]),np.linalg.norm([x-known_x, y-
            known_y])))

errors = np.array(errors)
plt.figure()
plt.scatter(errors[:,0], errors[:,1], s=4)
m, b, r, _, _ = stats.linregress(errors[:,0], errors[:,1])
x_axis = np.linspace(np.min(errors[:,0]), np.max(errors[:,0]), 100)
plt.plot(x_axis, m*x_axis + b, color='r', label=f'Fit: R^2 = {r**2:.5f}')
plt.title('Position Measurement Error vs Distance from Origin')
plt.xlabel('Distance from Origin [m]')
plt.ylabel('Error [m]')
plt.legend()
plt.savefig('out/p1_pos_error_dist_correlation.png')
plt.show()

# %%
# Calculate the variance of the r and t measurements
errors = []
for i in range(4):
    for position in grouped_positions[i]:
        time = position['time']
        r = position['r']
        t = np.deg2rad(position['t'])
        known_x = training_labels.loc[training_labels['Time']==time, f'x_{i}'].values[0]
        known_y = training_labels.loc[training_labels['Time']==time, f'y_{i}'].values[0]
        known_r = np.linalg.norm([known_x, known_y])
        known_t = np.arctan2(known_y, known_x)
        errors.append((known_r - r, known_t - t))
```

```
errors = np.array(errors)

# Plot the r error distribution
plt.figure()
hist, bins, _ = plt.hist(errors[:,0], bins=100, alpha=0.5, label='Range Errors', color='r')
xmin, xmax = plt.xlim()
mu, std = norm.fit(errors[:,0])
p = norm.pdf(bins, mu, std)
plt.plot(bins, p/p.sum() * len(errors[:,0]), label="Range | Fit: mu = %.5f, std = %.5f" % (
    mu, std), color='r')
plt.title('Range Measurement Error Distribution')
plt.xlabel('Error [m]')
plt.ylabel('Frequency')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
           fancybox=True, shadow=True, ncol=2)
plt.savefig('out/p1_r_pos_error_hist.png', bbox_inches='tight')
plt.show()

# Plot the t error distributions
plt.figure()
hist, bins, _ = plt.hist(errors[:,1], bins=100, alpha=0.5, label='Theta Errors', color='b')
xmin, xmax = plt.xlim()
mu, std = norm.fit(errors[:,1])
p = norm.pdf(bins, mu, std)
plt.plot(bins, p/p.sum() * len(errors[:,1]), label="Theta | Fit: mu = %.5f, std = %.5f" % (
    mu, std), color='b')
plt.title('Theta Measurement Error Distribution')
plt.xlabel('Error [rad]')
plt.ylabel('Frequency')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
           fancybox=True, shadow=True, ncol=2)
plt.savefig('out/p1_t_pos_error_hist.png', bbox_inches='tight')
plt.show()

# Calculate the variances
range_variance = np.var(errors[:,0])
theta_variance = np.var(errors[:,1])
print(f"Range Measurement Variance: {range_variance}")
print(f"Theta Measurement Variance: {theta_variance}")

# %% [markdown]
# ## Extended Kalman Filter for Single Fish Tracking
# #### State Matrix:
# $$
# \text{State Matrix} \, =
# \begin{bmatrix}
#     \text{position} \\
#     \text{velocity} \\
#     \text{acceleration} \\
#     \text{size} \\
# \end{bmatrix}
# =
# \begin{bmatrix}
#     x,y \\
#     \dot{x}, \dot{y} \\
#     \ddot{x}, \ddot{y} \\
#     s, 0 \\
# \end{bmatrix}
# $$
#
# #### State update equations:
# $$
# x = F x
# \\~\\
# \begin{aligned}
#     \text{Derivative Matrix} \, &=
#     \begin{bmatrix}
#         \text{velocity} \\
```

```
#           \text{acceleration} \\
#           \text{jerk} \\
#           \text{growth} \\
#       \end{bmatrix} \\
#       &=
#       \begin{bmatrix}
#           (\dot{x}, \dot{y}) + w_\text{vel}\\
#           (\ddot{x}, \ddot{y}) + w_\text{acc}\\
#           \left(\stackrel{\dotsm}{x}, \stackrel{\dotsm}{y} \right) + w_\text{jerk}\\
#           (\dot{s} + w_\text{size}, 0)
#       \end{bmatrix} \\
#       &=
#       \begin{bmatrix}
#           (\dot{x}, \dot{y}) + w_\text{vel}\\
#           (\ddot{x}, \ddot{y}) + w_\text{acc}\\
#           (0, 0) + w_\text{jerk}\\
#           (0 + w_\text{size}, 0)
#       \end{bmatrix} \\
#       &=
#       \begin{bmatrix}
#           0 & 1 & 0 & 0\\
#           0 & 0 & 1 & 0\\
#           0 & 0 & 0 & 0\\
#           0 & 0 & 0 & 0\\
#       \end{bmatrix}
#       \begin{bmatrix}
#           (x,y) \\
#           (\dot{x}, \dot{y}) \\
#           (\ddot{x}, \ddot{y}) \\
#           (s, 0) \\
#       \end{bmatrix}
#       +
#       \begin{bmatrix}
#           w_\text{vel} \\
#           w_\text{acc} \\
#           w_\text{jerk} \\
#           w_\text{size} \\
#       \end{bmatrix}
# \end{aligned}
# \\~\\
# F = \frac{\partial f(S)}{\partial S} \approx
# \begin{bmatrix}
#     0 & 1 & 0 & 0\\
#     0 & 0 & 1 & 0\\
#     0 & 0 & 0 & 0\\
#     0 & 0 & 0 & 0\\
# \end{bmatrix}
# \begin{bmatrix}
#     (x,y) \\
#     (\dot{x}, \dot{y}) \\
#     (\ddot{x}, \ddot{y}) \\
#     (s, 0) \\
# \end{bmatrix}
# +
# \begin{bmatrix}
#     w_\text{vel} \\
#     w_\text{acc} \\
#     w_\text{jerk} \\
#     w_\text{size} \\
# \end{bmatrix}
# \\~\\
# \Phi = I + F \Delta t + \frac{(F \Delta t)^2}{2!} + \frac{(F \Delta t)^3}{3!} + \dots \\
# \Phi \approx I + F \Delta t
# \\~\\
# S_k = \Phi(\Delta t) S_{k-1}
# $$
#
# #### Measurement Matrix:
```

```
# $$
# \text{Measurements} \, =
# \begin{bmatrix}
#     \text{range} \\
#     \text{angle} \\
#     \text{size} \\
# \end{bmatrix}
# $$
#
# ##### Range Measurement:
# $$
# \text{Range}\, = r = ||\text{pos}|| = \sqrt{x^2 + y^2}
# \\~\\
# R = \left.\frac{\partial r}{\partial S} \right|_{S}
# \\~\\
# R =
# \begin{bmatrix}
#     \left(
#         \frac{x}{\sqrt{x^2 + y^2}},
#         \frac{y}{\sqrt{x^2 + y^2}}
#     \right) \\
#     0, 0 \\
#     0, 0 \\
#     0, 0 \\
# \end{bmatrix}
# $$
#
# ##### Angle Measurement:
# $$
# \text{Angle}\, = \theta = \tan^{-1}\left(\frac{y}{x}\right)
# \\~\\
# \Theta = \left.\frac{\partial \theta}{\partial S} \right|_{S}
# \\~\\
# \Theta =
# \begin{bmatrix}
#     \left(
#         \frac{-y}{x^2 + y^2},
#         \frac{x}{x^2 + y^2}
#     \right) \\
#     0, 0 \\
#     0, 0 \\
#     0, 0 \\
# \end{bmatrix}
# $$
#
# ##### Size Measurement:
# $$
# \text{Size}\, = s
# \\~\\
# S_\text{ize} = \left.\frac{\partial s}{\partial S} \right|_{S}
# \\~\\
# S_\text{ize} =
# \begin{bmatrix}
#     0, 0 \\
#     0, 0 \\
#     0, 0 \\
#     0, 0 \\
# \end{bmatrix}
# $$
#

# %%

class FishExtendedKalmanFilter:
    def __init__(self, range, theta, size, pos_noise, vel_noise, acc_noise, size_noise):
        x, y = (range * np.cos(np.deg2rad(theta)),
                range * np.sin(np.deg2rad(theta)))
        self.dt = 0.1 # Seconds
```

```
        self.ekf = ExtendedKalmanFilter(dim_x=7, dim_z=3) # 4 states with 3 measurements
        self.ekf.x = np.array([x, y, 0, 0, 0, 0, size]) # Initial position and size
        self.ekf.F = np.eye(7) + np.array([[0, 0, self.dt, 0, 0.5 * self.dt**2, 0, 0], # State transition matrix
                                           [0, 0, 0, self.dt, 0, 0.5 * self.dt**2, 0],
                                           [0, 0, 0, 0, self.dt, 0, 0],
                                           [0, 0, 0, 0, 0, self.dt, 0],
                                           [0, 0, 0, 0, 0, 0, 0],
                                           [0, 0, 0, 0, 0, 0, 0],
                                           [0, 0, 0, 0, 0, 0, 0]])
        self.ekf.R = np.diag([range_variance, # Experimental measurement noise
                              theta_variance,
                              size_variance])
        self.ekf.Q = np.diag([pos_noise, # Process noise
                              pos_noise,
                              vel_noise,
                              vel_noise,
                              acc_noise,
                              acc_noise,
                              size_noise])

    # Expected range reading for State
    def _range_at(self, S):
        return np.linalg.norm(S[0:2])
    # Expected range rate for State
    def _range_J_at(self, S):
        return np.array([
            S[0] / np.linalg.norm(S[0:2]),
            S[1] / np.linalg.norm(S[0:2]),
            0,
            0,
            0,
            0,
            0,
        ])
    # Expected theta reading for State
    def _theta_at(self, S):
        return np.arctan2(S[1], S[0])
    # Expected theta rate for State
    def _theta_J_at(self, S):
        return np.array([
            -S[1] / (S[0]**2 + S[1]**2),
            S[0] / (S[0]**2 + S[1]**2),
            0,
            0,
            0,
            0,
            0,
        ])
    # Expected size reading for State
    def _size_at(self, S):
        return S[6]
    # Expected size rate for State
    def _size_J_at(self, S):
        return np.array([
            0,
            0,
            0,
            0,
            0,
            0,
            1,
        ])

        # Expected sensor readings for State
    def _H_at(self, S):
        return np.array([self._range_at(S),
                         self._theta_at(S),
```

```
                         self._size_at(S)])
    # Expected sensor rates for State
    def _H_J_at(self, S):
        return np.array([self._range_J_at(S),
                         self._theta_J_at(S),
                         self._size_J_at(S)])

    def update(self, range, theta, size):
        self.ekf.update(np.array([range, np.deg2rad(theta), size]), self._H_J_at, self._H_at)

    def predict(self):
        self.ekf.predict()

    def update_predict(self, range, theta, size):
        self.update(range, theta, size)
        self.predict()

    def covariance(self):
        return self.ekf.P

    def current_state(self):
        return self.ekf.x

    def predicted_next_state(self):
        return self.ekf.x @ self.ekf.F

    # Z-score of a measurement against the predicted measurement
    def z_score_of_measurement_match(self, range, theta, size, max_size_z_score):
        # Get expected sensor readings
        next = self.predicted_next_state()
        r_next = self._range_at(next)
        t_next = self._theta_at(next)
        s_next = self._size_at(next)

        # Calculate the likelihood of the measurement
        r_z_scr = np.abs((range - r_next) / range_variance**0.5)
        t_z_scr = np.abs((np.deg2rad(theta) - t_next) / theta_variance**0.5)
        s_z_scr = np.abs((size - s_next) / size_variance**0.5)
        # Return the position z score, or inf if the size z-score is too high
        if s_z_scr > max_size_z_score:
            return np.inf
        else:
            return r_z_scr + t_z_scr

    # RMS error of the predicted path vs the actual path
    def score_pos_accuracy(self, ranges, thetas, sizes, x, y):
        # Run an entire measurement set through the filter
        predicted = []
        for r, t, s in zip(ranges, thetas, sizes):
            self.update(r, t, s)
            self.predict()
            predicted.append(self.current_state())
        predicted = np.array(predicted)

        # Calculate the rms error from the actual path
        x = np.array(x)
        y = np.array(y)
        error = np.sqrt(np.mean((x - predicted[:,0])**2 + (y - predicted[:,1])**2))
        return error

    # RMS error of the predicted size vs the actual size
    def score_size_accuracy(self, ranges, thetas, sizes, actual_sizes):
        # Run an entire measurement set through the filter
        predicted = []
        for r, t, s in zip(ranges, thetas, sizes):
            self.update(r, t, s)
            self.predict()
```

```
            predicted.append(self.current_state())
        predicted = np.array(predicted)

        # Calculate the rms error from the actual path
        actual_sizes = np.array(actual_sizes)
        error = np.sqrt(np.mean((actual_sizes - predicted[:,6])**2))
        return error

# %%
# Optimize the Process Noise Values
print("Optimizing Process Noise Values on Fish 0...")
fish_0_test = training_combined[["Time", 'x_0', 'y_0', 's_0']]
fish_0_test = fish_0_test.assign(
    real_dist_0 = np.array([np.linalg.norm([x, y]) for x, y in zip(fish_0_test['x_0'], fish_0_test['y_0'])]),
    real_angle_0 = np.rad2deg(np.arctan2(fish_0_test['y_0'], fish_0_test['x_0'])),
    real_size_0 = fish_0_test['s_0'],
)
fish_0_test = fish_0_test.assign(
    sim_dist_0 = fish_0_test['real_dist_0'] + np.random.normal(0, range_variance**0.5, len(fish_0_test)),
    sim_angle_0 = fish_0_test['real_angle_0'] + np.random.normal(0, theta_variance**0.5, len(fish_0_test)),
    sim_size_0 = fish_0_test['real_size_0'] + np.random.normal(0, size_variance**0.5, len(fish_0_test)),
)
def pos_0_cost_function(x):
    sys = FishExtendedKalmanFilter(fish_0_test['sim_dist_0'][0],
                                   fish_0_test['sim_angle_0'][0],
                                   fish_0_test['sim_size_0'][0],
                                   *x)
    return sys.score_pos_accuracy(fish_0_test['sim_dist_0'],
                                  fish_0_test['sim_angle_0'],
                                  fish_0_test['sim_size_0'],
                                  training_labels['x_0'],
                                  training_labels['y_0'])
def size_0_cost_function(x):
    sys = FishExtendedKalmanFilter(fish_0_test['sim_dist_0'][0],
                                   fish_0_test['sim_angle_0'][0],
                                   fish_0_test['sim_size_0'][0],
                                   *x)
    return sys.score_size_accuracy(fish_0_test['sim_dist_0'],
                                   fish_0_test['sim_angle_0'],
                                   fish_0_test['sim_size_0'],
                                   training_labels['s_0'])
result_pos_0 = differential_evolution(pos_0_cost_function,
                                      [(0, 1), (0, 1), (0, 1), (0, 1)],
                                      maxiter=10000)
result_size_0 = differential_evolution(size_0_cost_function,
                                       [(0, 1), (0, 1), (0, 1), (0, 1)],
                                       maxiter=10000)
print(f"Optimized Process Noise Values on Fish 0: [{result_pos_0.x[0]},{result_pos_0.x[1]},{result_pos_0.x[2]},{result_size_0.x[3]}]")

print("Optimizing Process Noise Values on Fish 1...")
fish_1_test = training_combined[["Time", 'x_1', 'y_1', 's_1']]
fish_1_test = fish_1_test.assign(
    real_dist_1 = np.array([np.linalg.norm([x, y]) for x, y in zip(fish_1_test['x_1'], fish_1_test['y_1'])]),
    real_angle_1 = np.rad2deg(np.arctan2(fish_1_test['y_1'], fish_1_test['x_1'])),
    real_size_1 = fish_1_test['s_1'],
)
fish_1_test = fish_1_test.assign(
    sim_dist_1 = fish_1_test['real_dist_1'] + np.random.normal(0, range_variance**0.5, len(fish_1_test)),
    sim_angle_1 = fish_1_test['real_angle_1'] + np.random.normal(0, theta_variance**0.5, len(fish_1_test)),
```

```python
    sim_size_1 = fish_1_test['real_size_1'] + np.random.normal(0, size_variance**0.5, len(fish_1_test)),
)
def pos_0_cost_function(x):
    sys = FishExtendedKalmanFilter(fish_1_test['sim_dist_1'][0],
                                   fish_1_test['sim_angle_1'][0],
                                   fish_1_test['sim_size_1'][0],
                                   *x)
    return sys.score_pos_accuracy(fish_1_test['sim_dist_1'],
                                  fish_1_test['sim_angle_1'],
                                  fish_1_test['sim_size_1'],
                                  training_labels['x_1'],
                                  training_labels['y_1'])
def size_0_cost_function(x):
    sys = FishExtendedKalmanFilter(fish_1_test['sim_dist_1'][0],
                                   fish_1_test['sim_angle_1'][0],
                                   fish_1_test['sim_size_1'][0],
                                   *x)
    return sys.score_size_accuracy(fish_1_test['sim_dist_1'],
                                   fish_1_test['sim_angle_1'],
                                   fish_1_test['sim_size_1'],
                                   training_labels['s_1'])
result_pos_1 = differential_evolution(pos_0_cost_function,
                                      [(0, 1), (0, 1), (0, 1), (0, 1)],
                                      maxiter=10000)
result_size_1 = differential_evolution(size_0_cost_function,
                                       [(0, 1), (0, 1), (0, 1), (0, 1)],
                                       maxiter=10000)
print(f"Optimized Process Noise Values on Fish 1: [{result_pos_1.x[0]}, {result_pos_1.x[1]}, {result_pos_1.x[2]}, {result_size_1.x[3]}]")

# Average for Fish 0 and Fish 1
x = [(result_pos_0.x[0] + result_pos_1.x[0]) / 2,
     (result_pos_0.x[1] + result_pos_1.x[1]) / 2,
     (result_pos_0.x[2] + result_pos_1.x[2]) / 2,
     (result_size_0.x[3] + result_size_1.x[3]) / 2]
print(f"Average Optimized Process Noise Values: [{x[0]}, {x[1]}, {x[2]}, {x[3]}]")
pos_noise, vel_noise, acc_noise, size_noise = x

# %%
# Test the EKF against a sample dataset of 1 fish
fish_test = training_combined[["Time", "dist_0", 'angle_0', 'size_0', 'x_0', 'y_0', 's_0']]
fish_test = fish_test.assign(
    real_dist_0 = np.array([np.linalg.norm([x, y]) for x, y in zip(fish_test['x_0'], fish_test['y_0'])]),
    real_angle_0 = np.rad2deg(np.arctan2(fish_test['y_0'], fish_test['x_0'])),
    real_size_0 = fish_test['s_0'],
)
fish_test = fish_test.assign(
    sim_dist_0 = fish_test['real_dist_0'] + np.random.normal(0, range_variance**0.5, len(fish_test)),
    sim_angle_0 = fish_test['real_angle_0'] + np.random.normal(0, theta_variance**0.5, len(fish_test)),
    sim_size_0 = fish_test['real_size_0'] + np.random.normal(0, size_variance**0.5, len(fish_test)),
)
sys = FishExtendedKalmanFilter(fish_test["sim_dist_0"][0],
                               fish_test["sim_angle_0"][0],
                               fish_test["sim_size_0"][0],
                               pos_noise, vel_noise, acc_noise, size_noise)

predicted = []
odds = []
for index, meas in fish_test.iterrows():
    odds.append(sys.z_score_of_measurement_match(meas["sim_dist_0"], meas["sim_angle_0"], meas["sim_size_0"], 5))
    sys.update(meas["sim_dist_0"], meas["sim_angle_0"], meas["sim_size_0"])
    sys.predict()
```

```python
    state = sys.current_state()
    predicted.append(state)
predicted = np.array(predicted)
fish_test = fish_test.assign(px_0=predicted[:,0],
                             py_0=predicted[:,1],
                             ps_0=predicted[:,6])

# Calculate the measured location
fish_test['mx_0'] = fish_test['sim_dist_0'] * np.cos(np.deg2rad(fish_test['sim_angle_0']))
fish_test['my_0'] = fish_test['sim_dist_0'] * np.sin(np.deg2rad(fish_test['sim_angle_0']))

# Plot the predicted vs actual paths
plt.figure()
plt.plot(fish_test['x_0'], fish_test['y_0'], label='Actual Path')
plt.plot(fish_test['px_0'], fish_test['py_0'], label='Predicted Path', color='r')
plt.scatter(fish_test['mx_0'], fish_test['my_0'], label='Measurements', s=5, color='r')
plt.title('Fish 0 Path Prediction')
plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.legend()
plt.savefig('out/p1_test_fish_0_path.png')
plt.show()

# Plot the size prediction
plt.figure()
plt.plot(fish_test['Time'], fish_test['real_size_0'], label='Actual Size')
plt.plot(fish_test['Time'], fish_test['ps_0'], label='Predicted Size')
plt.scatter(fish_test['Time'], fish_test['sim_size_0'], label='Measurements', s=5, color='r'
    )
plt.title('Fish 0 Size Prediction')
plt.xlabel('Time [s]')
plt.ylabel('Size [m]')
plt.legend()
plt.savefig('out/p1_test_fish_0_size.png')

# Plot the odds of the measurements
plt.figure()
plt.plot(fish_test['Time'], odds)
plt.title('Fish 0 Measurement Odds')
plt.xlabel('Time [s]')
plt.ylabel('Combined Z Score')
plt.savefig('out/p1_test_fish_0_odds.png')
plt.show()


# %% [markdown]
# ## Multi-Fish Tracking
# Create an objct that can track multiple fish using the necessary number of extended Kalman
     filters.

# %%
class MultiTracker:
    def __init__(self, noises, max_pos_z_score=15):
        self.max_size_z_score = 10
        self.max_pos_z_score = max_pos_z_score
        self.max_missed = 5
        self.min_measures = 10
        self.fish_id = 0
        self.trackers = []
        self.errors = []
        self.predictions = {}
        self.pos_noise = noises[0]
        self.vel_noise = noises[1]
        self.acc_noise = noises[2]
        self.size_noise = noises[3]

    def add_tracker(self, measurement, time):
        self.trackers.append({
```

```
            "id": self.fish_id,
            "ekf": FishExtendedKalmanFilter(measurement[0],
                                            measurement[1],
                                            measurement[2],
                                            self.pos_noise,
                                            self.vel_noise,
                                            self.acc_noise,
                                            self.size_noise),
            "missed": 0,
            "measurements": 1,
            "updated": True
        })
        self.predictions[self.fish_id] = []
        self.fish_id += 1

    def update(self, time, measurements):
        # Edge-case of no existing trackers
        if len(self.trackers) == 0:
            for measurement in measurements:
                self.add_tracker(measurement, time)
        # Apply measurements to existing trackers, add trackers for unmatched measurements
        else:
            # Create an array of Z-scores for each measurement and tracker combo
            z_scores = np.zeros((len(measurements), len(self.trackers)))
            for i, measurement in enumerate(measurements):
                for j, tracker in enumerate(self.trackers):
                    z_scores[i,j] = tracker["ekf"].z_score_of_measurement_match(measurement[0],
                                                                               measurement[1],
                                                                               measurement[2],
                                                                               self.max_size_z_score)
            # print(z_scores)
            # Try to match measurements to the existing trackers
            for i in range(len(self.trackers)):
                # Ignore matches with too high of a Z-score
                if np.min(z_scores) > self.max_pos_z_score:
                    break
                # Apply the measurement for the combo with the lowest Z-score
                i_meas, i_trac = np.unravel_index(np.argmin(z_scores), z_scores.shape)
                self.trackers[i_trac]["ekf"].update_predict(measurements[i_meas][0],
                                                            measurements[i_meas][1],
                                                            measurements[i_meas][2])
                self.trackers[i_trac]["missed"] = 0
                self.trackers[i_trac]["measurements"] += 1
                self.trackers[i_trac]["updated"] = True
                # Remove the Z-score and measurement
                z_scores[i_meas,:] = np.inf
                z_scores[:,i_trac] = np.inf
                measurements[i_meas] = None
            # Add new trackers for any unused remaining measurements
            for measurement in [meas for meas in measurements if meas is not None]:
                self.add_tracker(measurement, time)

        # Check if the trackers have been updated
        for tracker in self.trackers:
            if not tracker["updated"]:
                tracker["missed"] += 1
                # next = tracker["ekf"].predicted_next_state()
                # tracker["ekf"].update_predict(tracker["ekf"]._range_at(next),
                #                               tracker["ekf"]._theta_at(next),
                #                               tracker["ekf"]._size_at(next))
                tracker["ekf"].predict()
            tracker["updated"] = False

```

```
        # Remove trackers that have been missed too many times
        for tracker in self.trackers:
            if tracker["missed"] > self.max_missed:
                # Trim the purely predicted locations
                del self.predictions[tracker["id"]][-self.max_missed:]
                # Remove the active tracker
                self.trackers.remove(tracker)

        # Record the predictions for each tracker
        for tracker in self.trackers:
            self.predictions[tracker["id"]].append({
                "Time": time,
                f"x_{tracker["id"]}": tracker["ekf"].current_state()[0],
                f"y_{tracker["id"]}": tracker["ekf"].current_state()[1],
                f"s_{tracker["id"]}": tracker["ekf"].current_state()[6],
                f"cov_{tracker["id"]}": tracker["ekf"].covariance(),
            })

    def get_predictions(self):
        # Trim outlying predictions without data
        for tracker in self.trackers:
            if tracker["missed"] > 0:
                # Trim the purely predicted locations
                del self.predictions[tracker["id"]][-tracker["missed"]:]
        # Create a dataframe of predicted Fish Locations
        ids = []
        fishes = []
        for id in self.predictions.keys():
            if len(self.predictions[id]) < self.min_measures + self.max_missed:
                continue
            ids.append(id)
            fishes.append(pd.DataFrame(self.predictions[id]))
        predictions = fishes[0]
        for i in range(len(fishes))[1:]:
           predictions = predictions.merge(fishes[i], on='Time', how='outer')
        return ids, predictions

# %%
# Try with training Data Set
tracker = MultiTracker(noises=[pos_noise, vel_noise, acc_noise, size_noise])

# Pipe Data Through the multi-tracker
for index, row in training_data.iterrows():
    time = row['Time']
    measurements = []
    for i in range(4):
        if not np.isnan(row[f'dist_{i}']):
            measurements.append((row[f'dist_{i}'], row[f'angle_{i}'], row[f'size_{i}']))
    tracker.update(time, measurements)

# Get the list of predicted tracks
ids, predictions = tracker.get_predictions()

all_measurements = pd.DataFrame({
    'Time': training_data['Time'],
    'dist_0': training_data['dist_0'],
    'angle_0': training_data['angle_0'],
    'dist_1': training_data['dist_1'],
    'angle_1': training_data['angle_1'],
    'dist_2': training_data['dist_2'],
    'angle_2': training_data['angle_2'],
    'dist_3': training_data['dist_3'],
    'angle_3': training_data['angle_3'],
})
all_measurements = all_measurements.assign(
    x_0 = all_measurements["dist_0"] * np.cos(np.deg2rad(all_measurements["angle_0"])),
    y_0 = all_measurements["dist_0"] * np.sin(np.deg2rad(all_measurements["angle_0"])),
    x_1 = all_measurements["dist_1"] * np.cos(np.deg2rad(all_measurements["angle_1"])),
```

```
    y_1 = all_measurements["dist_1"] * np.sin(np.deg2rad(all_measurements["angle_1"])),
    x_2 = all_measurements["dist_2"] * np.cos(np.deg2rad(all_measurements["angle_2"])),
    y_2 = all_measurements["dist_2"] * np.sin(np.deg2rad(all_measurements["angle_2"])),
    x_3 = all_measurements["dist_3"] * np.cos(np.deg2rad(all_measurements["angle_3"])),
    y_3 = all_measurements["dist_3"] * np.sin(np.deg2rad(all_measurements["angle_3"])),
)

# Plot the tracks together on 1 graph
plt.figure()
for i, id in enumerate(ids):
    size = np.mean([s for s in predictions[f's_{id}'] if not np.isnan(s)])
    plt.plot(predictions[f'x_{id}'], predictions[f'y_{id}'], label=f'Fish {i}, size={size:
        0.5f}')
    # Annotate the start and end of each movement with the time
    state = False
    for j, vals in predictions.iterrows():
        if ((not state and not np.isnan(vals[f'x_{id}'])) or j == len(predictions)-1):
            state = True
            plt.annotate(f't={vals['Time']}', (vals[f'x_{id}'], vals[f'y_{id}']), size=8)
        elif (state and np.isnan(vals[f'x_{id}'])):
            state = False
            vals = predictions.loc[j-1]
            plt.annotate(f't={vals['Time']}', (vals[f'x_{id}'], vals[f'y_{id}']), size=8)
for i in range(4):
    label=f'Actual Fish {['A','B','C','D'][i]}'
    plt.plot(training_labels[f'x_{i}'], training_labels[f'y_{i}'], label=label, linestyle='
        --')
for i in range(4):
    plt.scatter(all_measurements['x_'+str(i)], all_measurements['y_'+str(i)], s=5, color='
        black')
plt.title('Training Data - Fish Path Predictions')
plt.xlabel('X Position [m]')
plt.ylabel('Y Position [m]')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
           fancybox=True, shadow=True, ncol=4)
plt.savefig('out/p1_training_paths.png', bbox_inches='tight')
plt.show()

# %% [markdown]
# ## Run on Test Data and Plot Results

# %%
# Filter the Test Data
test_tracker = MultiTracker(noises=[pos_noise, vel_noise, acc_noise, size_noise])
for index, row in test_data.iterrows():
    time = row['Time']
    measurements = []
    for i in range(3):
        if not np.isnan(row[f'dist_{i}']):
            measurements.append((row[f'dist_{i}'], row[f'angle_{i}'], row[f'size_{i}']))
    test_tracker.update(time, measurements)

# Get the list of predicted tracks
ids, predictions = test_tracker.get_predictions()

# Plot the tracks together on 1 graph
plt.figure()
for i, id in enumerate(ids):
    size = np.mean([s for s in predictions[f's_{id}'] if not np.isnan(s)])
    plt.plot(predictions[f'x_{id}'], predictions[f'y_{id}'], label=f'Fish {i}, size={size:
        0.5f}')
    # Annotate the start and end of each movement with the time
    state = False
    for j, vals in predictions.iterrows():
        if ((not state and not np.isnan(vals[f'x_{id}'])) or j == len(predictions)-1):
            state = True
            plt.annotate(f't={vals['Time']}', (vals[f'x_{id}'], vals[f'y_{id}']), size=8)
        elif (state and np.isnan(vals[f'x_{id}'])):
```

```
                state = False
                vals = predictions.loc[j-1]
                plt.annotate(f't={vals['Time']}', (vals[f'x_{id}'], vals[f'y_{id}']), size=8)
# Add scatter of all measurements to the plot
for i in range(3):
    plt.scatter(test_data['dist_'+str(i)] * np.cos(np.deg2rad(test_data['angle_'+str(i)])),
                test_data['dist_'+str(i)] * np.sin(np.deg2rad(test_data['angle_'+str(i)])),
                s=5, color='black')
plt.title('Test Data - Fish Path Predictions')
plt.xlabel('X Position [m]')
plt.ylabel('Y Position [m]')
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
           fancybox=True, shadow=True, ncol=3)
plt.savefig('out/p1_test_paths.png', bbox_inches='tight')
plt.show()

# Print the final Position and covariance of each Fish
table = [['Fish #', 'Size', 'Time Start', 'Time End', 'Final X [m]', 'Final Y [m]', 'Final
    Cov.']]
for i, id in enumerate(ids):
    size = np.mean([s for s in predictions[f's_{id}'] if not np.isnan(s)])
    start_time = 0
    end_time = 0
    final_x = 0
    final_y = 0
    final_cov = 0
    state = False
    for j, vals in predictions.iterrows():
        if ((not state and not np.isnan(vals[f'x_{id}']))):
            state = True
            start_time = vals['Time']
        elif (j == len(predictions)-1):
            end_time = vals['Time']
            final_x = vals[f'x_{id}']
            final_y = vals[f'y_{id}']
            final_cov = vals[f'cov_{id}']
            break
        elif (state and np.isnan(vals[f'x_{id}'])):
            state = False
            vals = predictions.loc[j-1]
            end_time = vals['Time']
            final_x = vals[f'x_{id}']
            final_y = vals[f'y_{id}']
            final_cov = vals[f'cov_{id}']
            break
    table.append([f'Fish {i}',
                  f'{size:+0.4f}',
                  f'{start_time:+0.4f}',
                  f'{end_time:+0.4f}',
                  f'{final_x:+0.4f}',
                  f'{final_y:+0.4f}',
                  final_cov])
tab = PrettyTable(table[0])
tab.add_rows(table[1:])
with np.printoptions(formatter={'float': lambda x: "{0:+0.3f}".format(x)}):
    print(tab)

# Create an index of the number of fish at each timestamp
fish_count_list = []
for i, vals in predictions.iterrows():
    sum = 0
    for j, id in enumerate(ids):
        sum += not np.isnan(vals[f'x_{id}'])
    fish_count_list.append(sum)
# Write to a CSV file
with open('out/test_data_fish_count.csv', 'w', newline='') as csvfile:
    write = csv.writer(csvfile, delimiter=',')
    for count in fish_count_list:
```

```
        write.writerow([count])
# Plot the fish at times
plt.figure()
plt.plot(predictions['Time'], fish_count_list)
plt.title('Fish Count Over Time')
plt.xlabel('Time [s]')
plt.ylabel('Fish Count')
plt.savefig('out/p1_fish_count.png')
plt.show()
```

Appendix C Problem 2 Source Code

```python
# %% [markdown]
# # Problem #2
#
# ## Data Ingest and Formatting

# %%
# Common Imports
import csv
import pathlib
import itertools
import statistics
import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import matplotlib.patheffects as pe
from matplotlib.patches import Rectangle
from prettytable import PrettyTable


# Ensure the plot output directory exists
pathlib.Path('./out').mkdir(parents=True, exist_ok=True)

# Read in the data from excel file
kicks = pd.read_excel('./data/Penalty_kick.xlsx',sheet_name='Sheet1')
kicks.columns = ['x', 'y', 'z', 'area']
Areas = [x for x in range(1, 7)]

print(kicks.head())

# Create a dictionary to store the number of kicks in each area
area_counts = dict()
for area in Areas:
    area_counts[area] = kicks[kicks['area'] == area].shape[0]

# Helpful other config
hallow_marker = dict(marker='o', markersize=3,
                           color='black',
                           markerfillstyle='none')

# %% [markdown]
# ## Probability Profile for each Area
# Note: `Y` is viewed as horizontal Axis and `Z` is viewed as vertical Axis
#
# ![image.png](attachment:image.png)

# %%
# Plot all the kicks on a 2d plane
fig, ax = plt.subplots()
ax.add_patch(Rectangle((-1, -1), 9.32, 5, color="red", alpha=0.5))
ax.add_patch(Rectangle((0, 0), 7.32, 2.44, color="white", alpha=1))
ax.add_patch(Rectangle((0, 0), 7.32, 2.44, color="lightgreen", alpha=0.5))
ax.axhline(y=0, color='black', linestyle='-')
for area in Areas:
    ax.scatter(kicks[kicks['area'] == area]['y'],
               kicks[kicks['area'] == area]['z'],
               label=f'Contact Area {area}')
ax.set_xlim((-1, 8.32))
ax.set_ylim((-1, 4.00))
ax.set_xlabel('Y [m]')
ax.set_ylabel('Z [m]')
handles, labels = ax.get_legend_handles_labels()
order = [0, 3, 1, 4, 2, 5]
fig.legend([handles[i] for i in order], [labels[i] for i in order],
           loc='upper center', bbox_to_anchor=(0.5, 0),
           fancybox=True, shadow=True, ncol=3)
ax.set_title('Kick Placement By Contact Area')
```

```python
plt.savefig('./out/kick_plot_by_contact_area.png')
plt.show()

# %%
# Create a probability density function for the area of the kick
mean = np.array([kicks[kicks['area'] == area][['y', 'z']].mean() for area in Areas])
var = np.array([kicks[kicks['area'] == area][['y', 'z']].var() for area in Areas])

# Plot the probability density function for the area of the kick
rows, columns = (2, 3)
fig, axs = plt.subplots(rows, columns, sharex=False, sharey=False, figsize=(3*columns, 2*
    rows))
fig.suptitle(f'Gaussian Distributions of Kick Placement for Contact Areas')
fig.tight_layout(h_pad=-1, w_pad=3)
for i, area in enumerate(Areas):
    column = i % 3
    row = int((i - column) / 3)
    ax = axs[row, column]
    y = np.linspace(-1, 8.32, 100)
    z = np.linspace(-1, 4.00, 100)
    Y, Z = np.meshgrid(y, z)
    pos = np.empty(Y.shape + (2,))
    pos[:, :, 0] = Y
    pos[:, :, 1] = Z
    rv = stats.multivariate_normal(mean[i,:], np.diag(var[i,:]))
    ax.set_aspect('equal', adjustable='box')
    con = ax.contourf(Y, Z, rv.pdf(pos))
    ax.add_patch(Rectangle((0, 0), 7.32, 2.44, fill=False, edgecolor="white"))
    ax.axhline(y=0, color='white', linestyle='-')
    ax.plot(mean[i,0], mean[i,1], marker='o', fillstyle='none', color="black")
    ax.text(np.clip(mean[i,0],1.3,6.2), np.clip(mean[i,1]+1.2, 0.4, 3.3),
            f'({mean[i,0]: 0.2f}, {mean[i,1]: 0.2f})',
            fontsize=9, ha='center', color="white",
            path_effects=[pe.withStroke(linewidth=2, foreground="black")])
    ax.set_xlim((-1, 8.32))
    ax.set_ylim((-1, 4.00))
    ax.set_xlabel('Y [m]')
    ax.set_ylabel('Z [m]')
    fig.colorbar(con, ax=ax, shrink=0.85, label='Probability Density')
    ax.set_title(f'Contact Area {area}')
plt.savefig('./out/contact_area_probability_dist.png')
plt.show()

# Plot the normalized probability density function for the area of the kick
rows, columns = (2, 3)
fig, axs = plt.subplots(rows, columns, sharex=True, sharey=True, figsize=(3*columns, 2*rows)
    )
fig.suptitle(f'Normalized Gaussian Distributions of Kick Placement for Contact Areas', x
    =0.415)
fig.tight_layout(h_pad=-1, w_pad=1.5)
levels = np.linspace(0, 0.5, 21)
con = None
for i, area in enumerate(Areas):
    column = i % 3
    row = int((i - column) / 3)
    ax = axs[row, column]
    y = np.linspace(-1, 8.32, 100)
    z = np.linspace(-1, 4.00, 100)
    Y, Z = np.meshgrid(y, z)
    pos = np.empty(Y.shape + (2,))
    pos[:, :, 0] = Y
    pos[:, :, 1] = Z
    rv = stats.multivariate_normal(mean[i,:], np.diag(var[i,:]))
    ax.set_aspect('equal', adjustable='box')
    con = ax.contourf(Y, Z, rv.pdf(pos), levels)
    ax.add_patch(Rectangle((0, 0), 7.32, 2.44, fill=False, edgecolor="white"))
    ax.axhline(y=0, color='white', linestyle='-')
    ax.plot(mean[i,0], mean[i,1], marker='o', fillstyle='none', color="black")
```

```python
    ax.text(np.clip(mean[i,0],1.3,6.2), np.clip(mean[i,1]+1.2, 0.4, 3.3),
            f'({mean[i,0]: 0.2f}, {mean[i,1]: 0.2f})',
            fontsize=9, ha='center', color="white",
            path_effects=[pe.withStroke(linewidth=2, foreground="black")])
    ax.set_xlim((-1, 8.32))
    ax.set_ylim((-1, 4.00))
    ax.set_xlabel('Y [m]')
    ax.set_ylabel('Z [m]')
    ax.set_title(f'Contact Area {area}')
fig.colorbar(con, ax=axs, shrink=0.85, label='Probability Density')
plt.savefig('./out/normalized_contact_area_probability_dist.png')
plt.show()

# Print the properties of the probability density for each contact area
headers = ['Contact Area', 'Y Mean', 'Y Deviation', 'Z Mean', 'Z Deviation']
rows = []
for i, area in enumerate(Areas):
    rows.append([area,
                 f'{mean[i,0]:0.4f}',
                 f'{var[i,0]:0.4f}',
                 f'{mean[i,1]:0.4f}',
                 f'{var[i,1]:0.4f}'])
table = PrettyTable(headers)
table.add_rows(rows)
with np.printoptions(formatter={'float': lambda x: "{0:+0.3f}".format(x)}):
    print(table)

# %% [markdown]
# ## Weighted Sum Fusion
#
# Weighted sum calcs:
# $$
# y_{ws} = \frac{\sum_{i=1}^{6} \frac{w_i}{\sigma_i^2} y_i}{\sum_{i=1}^{6} \frac{w_i}{\sigma_i^2}}
# \\~\\
# \sigma_{ws}^2 = \sum_{i=1}^{6} \left( \frac{\frac{w_i}{\sigma_i^2}}{\sum_{i=1}^6 \frac{w_i}{\sigma_i^2}} \right)^2 \sigma_i^2
# $$

# %%
weight = np.array([0.02, 0.02, 0.02, 0.46, 0.46, 0.02])

# Calculate the expected value of the kick
expected_value = np.sum(mean * weight[:, None] / var , axis=0) / np.sum(weight[:, None] / var, axis=0)
expected_var = np.sum(((weight[:, None] / var) / (np.sum(weight[:, None] / var, axis=0)))**2 * var, axis=0)

# Plot the probability field
fig, ax = plt.subplots()
ax.set_title("Weighted Sum Estimate of Kick")
y = np.linspace(-1, 8.32, 100)
z = np.linspace(-1, 4.00, 100)
Y, Z = np.meshgrid(y, z)
pos = np.empty(Y.shape + (2,))
pos[:, :, 0] = Y
pos[:, :, 1] = Z
rv = stats.multivariate_normal(expected_value[:], np.diag(expected_var[:]))
con = ax.contourf(Y, Z, rv.pdf(pos))
ax.set_aspect('equal', adjustable='box')
ax.add_patch(Rectangle((0, 0), 7.32, 2.44, fill=False, edgecolor="white"))
ax.axhline(y=0, color='white', linestyle='-')
ax.plot(expected_value[0], expected_value[1],
        marker='o', fillstyle='none', color="black")
ax.text(expected_value[0], expected_value[1]+0.2,
        f'({expected_value[0]: 0.3f}, {expected_value[1]: 0.3f})',
        fontsize=9, ha='center', color="white",
        path_effects=[pe.withStroke(linewidth=2, foreground="black")])
```

```python
ax.set_xlim((-1, 8.32))
ax.set_ylim((-1, 4.00))
ax.set_xlabel('Y [m]')
ax.set_ylabel('Z [m]')
fig.colorbar(con, ax=ax, shrink=0.575, label='Probability Density')
plt.savefig('./out/weighted_sum_probability_dist.png')
plt.show()

# Output the values
print(f'Expected Value of Kick: {expected_value}')
print(f'Expected Var of Kick: {expected_var}')
```

Appendix D Problem 3 Source Code

```python
# %% [markdown]
# # Problem #3
#
# ## Data Ingest and Formatting

# %%
# Common Imports
import csv
import pathlib
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from prettytable import PrettyTable

import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Activation
from tensorflow.keras.metrics import R2Score
from tensorflow.keras.metrics import Accuracy
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.callbacks import History
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras import ops
from tensorflow.keras import layers
import tensorflow.keras.backend as K

# Ensure the plot output directory exists
pathlib.Path('./out').mkdir(parents=True, exist_ok=True)

# Import the subject data as dataframes
training_data = []
for i in range(1, 9):
    training_data.append({
        "in": pd.read_csv(f'./data/sub_{i}_input.csv'),
        "out": pd.read_csv(f'./data/sub_{i}_output.csv')
    })
test_data = {
    "in": pd.read_csv('./data/sub_9_input.csv'),
    "out": pd.read_csv('./data/sub_9_output.csv')
}
exam_data = pd.read_csv('./data/sub_10_input.csv')

# Array of field names
input_fields = [
    'q_hip_right',
    'q_knee_right',
    'q_hip_left',
    'q_knee_left',
    'dq_hip_right',
    'dq_knee_right',
    'dq_hip_left',
    'dq_knee_left',
    'u_hip_right',
    'u_knee_right',
    'u_hip_left',
    'u_knee_left',
    'gyro_right_thigh_x',
    'gyro_right_thigh_y',
    'gyro_right_thigh_z',
    'gyro_left_thigh_x',
    'gyro_left_thigh_y',
    'gyro_left_thigh_z',
```

```
    'acc_right_thigh_x',
    'acc_right_thigh_y',
    'acc_right_thigh_z',
    'acc_left_thigh_x',
    'acc_left_thigh_y',
    'acc_left_thigh_z',
    'acc_gu_right_foot_x',
    'acc_gu_right_foot_y',
    'acc_gu_right_foot_z',
    'acc_gu_left_foot_x',
    'acc_gu_left_foot_y',
    'acc_gu_left_foot_z',
    'acc_gu_right_shank_x',
    'acc_gu_right_shank_y',
    'acc_gu_right_shank_z',
    'acc_gu_left_shank_x',
    'acc_gu_left_shank_y',
    'acc_gu_left_shank_z',
    'gyro_gu_right_foot_x',
    'gyro_gu_right_foot_y',
    'gyro_gu_right_foot_z',
    'gyro_gu_left_foot_x',
    'gyro_gu_left_foot_y',
    'gyro_gu_left_foot_z',
    'gyro_gu_right_shank_x',
    'gyro_gu_right_shank_y',
    'gyro_gu_right_shank_z',
    'gyro_gu_left_shank_x',
    'gyro_gu_left_shank_y',
    'gyro_gu_left_shank_z',
    'sf_right',
    'sf_left'
]
output_fields = [
    'grf_right_y',
    'grf_right_z',
    'grf_left_y',
    'grf_left_z',
]

# %% [markdown]
# ## Machine Learning Model
# ### Training

# %%
# Create the Model
inputs = Input(shape=(len(input_fields),))
x = layers.Dense(len(input_fields)*2, activation='sigmoid')(inputs)
x = layers.Dense(len(input_fields)*2, activation='sigmoid')(x)
# x = layers.Dense(len(input_fields)*2, activation='sigmoid')(x)
outputs = layers.Dense(len(output_fields), activation='linear')(x)
model = Model(inputs=inputs, outputs=outputs, name='WalkieBoi')
model.summary()
model.build(input_shape=(None, len(input_fields)))

def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))

model.compile(optimizer='rmsprop',
              loss = root_mean_squared_error,
              metrics=[RootMeanSquaredError(), R2Score()])

# Train the Model
X = np.empty((0,len(input_fields)))
Y = np.empty((0,len(output_fields)))
for i in range(1, 9):
    X = np.concatenate((X, training_data[i-1]['in'].to_numpy()), axis=0)
    Y = np.concatenate((Y, training_data[i-1]['out'].to_numpy()), axis=0)
```

```
history = model.fit(x=X, y=Y, batch_size=50, shuffle='batch', epochs=100,
                    validation_data=(test_data['in'], test_data['out']),
                    callbacks=[History(),
                               EarlyStopping(monitor='val_loss',
                                             patience=5,
                                             restore_best_weights=True)])
history = pd.DataFrame(history.history)

# %% [markdown]
# ### Training Results

# %%
# Plot the RSME Training History
fig, ax = plt.subplots()
ax.plot(history['root_mean_squared_error'], label='Training RMSE')
ax.plot(history['val_root_mean_squared_error'], label='Validation RMSE')
ax.plot(len(history['root_mean_squared_error'])-1,
        history['root_mean_squared_error'].iloc[-1],
        'o', label='', color='tab:blue')
ax.text(len(history['root_mean_squared_error'])-1.25,
        history['root_mean_squared_error'].iloc[-1]+0.003,
        f'({history["root_mean_squared_error"].iloc[-1]:.4f})',
        fontsize=10, ha='center', va='bottom')
ax.plot(len(history['val_root_mean_squared_error'])-1,
        history['val_root_mean_squared_error'].iloc[-1],
        'o', label='', color='tab:orange')
ax.text(len(history['val_root_mean_squared_error'])-1.25,
        history['val_root_mean_squared_error'].iloc[-1]+0.003,
        f'({history["val_root_mean_squared_error"].iloc[-1]:.4f})',
        fontsize=10, ha='center', va='bottom')
ax.set_title('Training Evolution of RMSE')
ax.set_xlabel('Epoch')
ax.set_ylabel('Root Mean Squared Error')
ax.xaxis.grid()
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
          fancybox=True, shadow=True, ncol=2)
plt.tight_layout()
plt.savefig('./out/training_history_rmse.png')
plt.show()

# Plot the R2 Score Training History
fig, ax = plt.subplots()
ax.plot(history['r2_score'], label='Training R2')
ax.plot(history['val_r2_score'], label='Validation R2')
ax.plot(len(history['r2_score'])-1,
        history['r2_score'].iloc[-1],
        'o', label='', color='tab:blue')
ax.text(len(history['r2_score'])-1.25,
        history['r2_score'].iloc[-1]-0.065,
        f'({history["r2_score"].iloc[-1]:.4f})',
        fontsize=10, ha='center', va='bottom')
ax.plot(len(history['val_r2_score'])-1,
        history['val_r2_score'].iloc[-1],
        'o', label='', color='tab:orange')
ax.text(len(history['val_r2_score'])-1.25,
        history['val_r2_score'].iloc[-1]-0.065,
        f'({history["val_r2_score"].iloc[-1]:.4f})',
        fontsize=10, ha='center', va='bottom')
ax.set_title('Training Evolution of R2 Score')
ax.set_xlabel('Epoch')
ax.set_ylabel('R2 Score')
ax.xaxis.grid()
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.125),
          fancybox=True, shadow=True, ncol=2)
plt.tight_layout()
plt.savefig('./out/training_history_r2.png')
plt.show()
```

```python

# %% [markdown]
# ### Prediction

# %%
# Predict the output for subject 10
predictions = model.predict(test_data['in'], batch_size=1)
predictions = pd.DataFrame(predictions, columns=output_fields)

# Save the predictions to a CSV file
predictions.to_csv('./out/sub_10_output.csv', index=False, float_format='%.16f')

# %%
# Plot the Predictions
fig, axs = plt.subplots(2, 1, figsize=(24, 12))
axs[0].plot(predictions['grf_right_y'], label='GRF Right Y')
axs[0].plot(predictions['grf_left_y'], label='GRF Left Y')
axs[0].set_title('Predicted GRF Y')
axs[0].set_xlabel('Time Step')
axs[0].set_ylabel('Force [N]')
axs[0].legend()
axs[0].xaxis.grid()
axs[1].plot(predictions['grf_right_z'], label='GRF Right Z')
axs[1].plot(predictions['grf_left_z'], label='GRF Left Z')
axs[1].set_title('Predicted GRF Z')
axs[1].set_xlabel('Time Step')
axs[1].set_ylabel('Force [N]')
axs[1].legend()
axs[1].xaxis.grid()
plt.tight_layout()
plt.savefig('./out/predictions.png')
plt.show()
```