# MTE546 - Final Exam

**Multi-Sensor Data Fusion**

Logan Hartford

20739675

University of Waterloo

April 12, 2025

# 1 Troubleshooting an Existing Sensor Model

## 1.1 Initial Analysis

The estimated and desired outputs for the inhouse sensor is plotted using the provided data, sensor model and ideal model, the result can be seen in Figure 1 below.

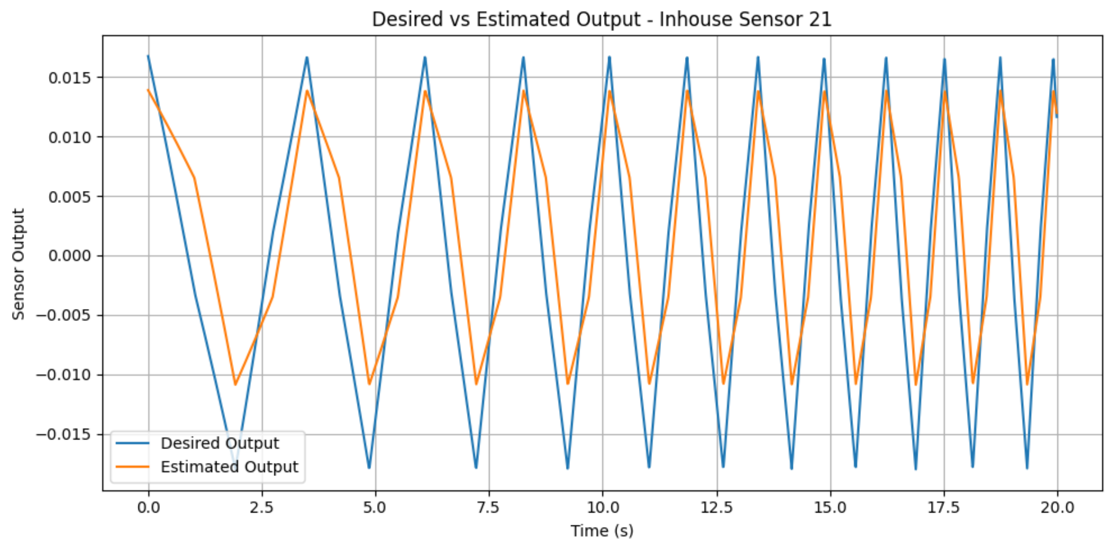

Figure 1: Desired vs estimate output for the inhouse sensor.

The estimated output in the plot above follows the same general waveform as the desired output, that being a periodic signal with increasing frequency. However, the estimated output's amplitude is consistently less that that of the desired output, and there seems to be some phase delay between the two outputs. Next, a scatter plot is generated with the estimated output on the y-axis and desired output on the x-axis for the inhouse sensor and client sensor 21, the results can be seen in Figure 2 below.

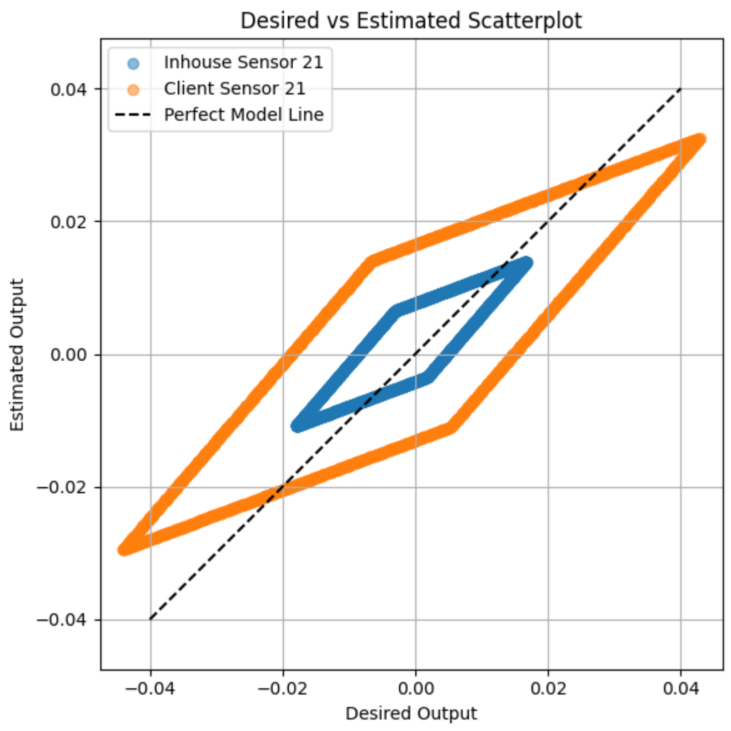

Figure 2: Scatter plot of desired and estimated outputs for the inhouse and client sensors.

The perfect model line is added to the plot, which is just y=x since the estimated output of a perfect model would always match the desired output. Referring to the plot above, the client sensor shows greater deviation and variability compared to the inhouse sensor, suggesting it is either less accurate or experiencing drift, noise, or miscalibration. This validates the client's claim of inconsistent performance.

## 1.2 Independent Model Development

### 1.2.1 Pre-Processing Steps

The data for the client and inhouse sensors were loaded using pandas and then the desired output for each dataset was computed using the $q1$ and $q2$ columns and the provided analytical model. Lastly the $v1$ and $v2$ columns as well as the desire output for the two datasets were concatenated into an $X$ and a $Y$ dataset.

### 1.2.2 Method

The method used to generate the sensor model was Ordinary Least Squared (OLS) using the `LinearRegression()` model from the `scikit-learn` library. The fit method was called on the model with the $X$ data set as the input and $Y$ dataset as the target. The fit method uses OLS to find linear combination of inputs that minimizes the error between the estimated and desired output.

### 1.2.3 Justification

Linear regression using OLS was determined to be a suitable method for developing a model given that the original sensor model was linear, and the provided analytical model was also linear. OLS provides a simple and interpretable model that clearly shows how each input contributes to the output. Additionally, this method is computationally efficient, easy to implement and provides a simple estimation of model uncertainty through residual analysis. Implementing and fitting this model was done with `scikit-learn` for simplicity and easy of implementation.

#### 1.2.4 Model Equations and Parameters

The computed sensor model,

$$\hat{y} = 0.010957\nu_1 + 0.029573\nu_2 - 0.000235 + \eta, \quad \eta \sim \mathcal{N}(0, 0.0007175)$$

- $\nu_i$ – Sensor voltages
- $\hat{y}$ – Predicted output
- $\eta$ – Modeling noise

The residuals of the model were plotted and can be seen in Figure 3 below.

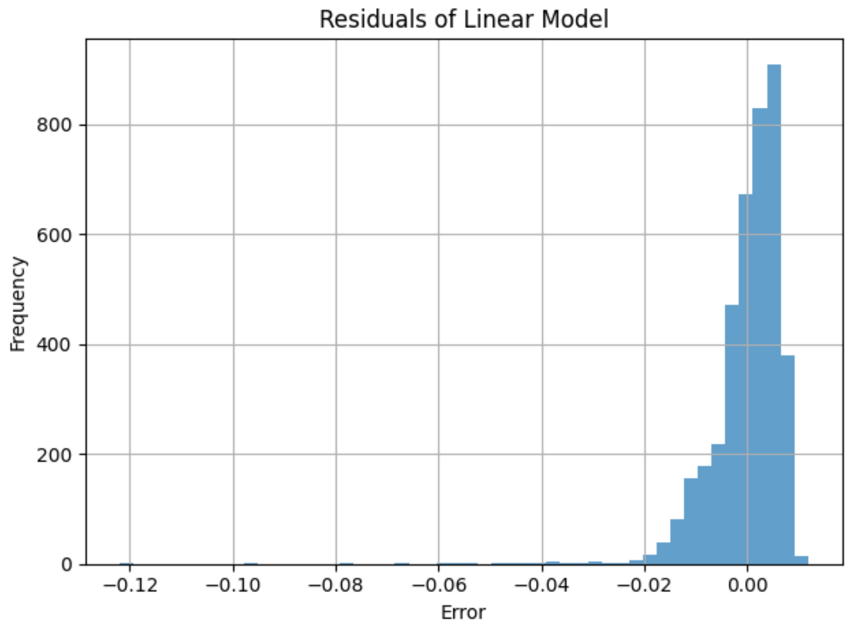

Figure 3: Residuals between linear model estimation and desired output.

### 1.3 Model Generalizability and Model 2

The bootstrapping method was used to find the distribution of model coefficients. From a random sample size of 1000, 100 models were computed, the distribution of their coefficients can be seen in Figure 4 below.

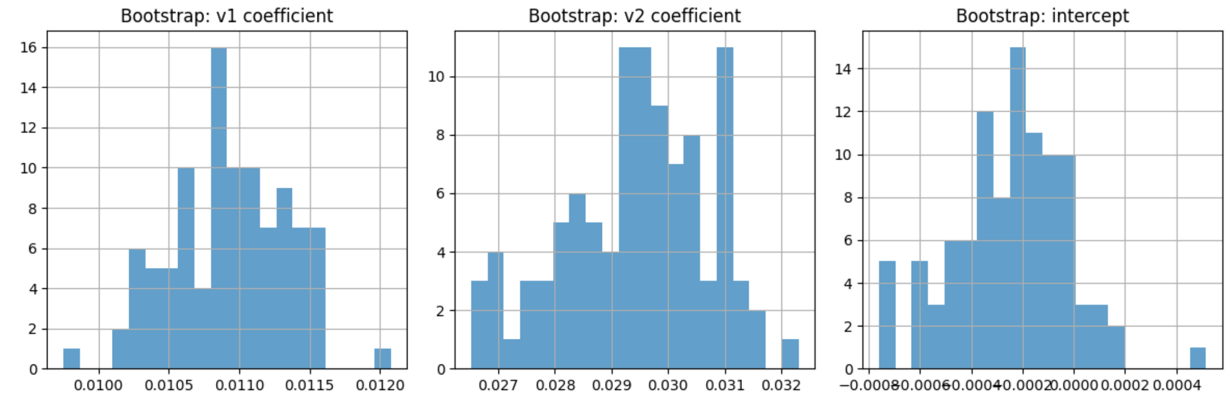

Figure 4: Coefficient histogram with frequency on the y-axis and coefficient magnitude on the x-axis.

From this data, the mean coefficient vector and covariance matrix of coefficients was computed. Next a model was fit using the same method as in 2A using the client 22 sensor data and then the Mahalanobis distance between the client 22 model and the mean coefficient vector was computed. The computed Mahalanobis distance was 24.8205, which provides strong evidence against the generalizability of the model. The Mahalanobis distance quantifies how far a new point is from the distribution of the trained model's coefficients, in units of standard deviation. In this case, the squared distance $d^2 \approx 616.2$ greatly exceeds the 99.9% chi-squared threshold of 16.27 for 3 degrees of freedom. This indicates that the coefficients obtained from client sensor 22 are extremely unlikely to have come from the same distribution as those derived from client and inhouse sensors 21. Therefore, client sensor 22's behavior differs substantially from that of the previously seen sensors, and the model trained on sensor 21 data may not generalize well to sensor 22.

## 1.4 Model Comparison

The performance of the original sensor model and the sensor model obtain in 2A were compared using average error, error variance, Root Mean Squared Error (RMSE) and coefficient determination ($R^2$). The metrics for both models are summarized in Table 1 below.

Table 1: Performance metrics for 2A and original sensor models.

| **Metric** | **2A Model** | **Original Model** |
|---|---|---|
| Average Error | -0.005281 | -0.009263 |
| Variance of Error | 0.000308 | 0.000308 |
| RMSE | 0.010323 | 0.019832 |
| $R^2$ | 0.892561 | 0.603458 |

From the information in the table above, it was determined that the independent derived model from 2A should be used in the next batch of sensors. Compared to the original model, the 2A model had a smaller average error, lower RSME and higher coefficient determination, while the variance of the error was the same for both models. This indicates that on the client 22 data, the new model performs significantly better than the original model, and on this basis it should be used in favor of the original model.

## 2 EKF Based State Estimation for the Improbability Drive

### 2.1 EKF Derivation

The state vector is defined as:

$$X_i = \begin{bmatrix} x_i \\ y_i \\ z_i \\ v_i \\ \phi_i \\ \theta_i \\ \psi_i \end{bmatrix}, \quad u_i = \begin{bmatrix} a_i \\ \omega_{x,i} \\ \omega_{y,i} \\ \omega_{z,i} \end{bmatrix}$$

The nonlinear motion model is given by:

$$X_{i+1} = g(X_i, u_i) + q_i$$

The update is broken down into three parts in the sections below.

#### 2.1.1 Position Update

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \\ z_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + R(\phi_i, \theta_i, \psi_i) \begin{bmatrix} v_i \Delta T + \frac{1}{2} a_i \Delta T^2 \\ 0 \\ 0 \end{bmatrix}$$

where,

$$R(\phi_i, \theta_i, \psi_i) = \begin{bmatrix} C_{\phi_i} C_{\theta_i} & C_{\phi_i} S_{\theta_i} S_{\psi_i} - S_{\phi_i} C_{\psi_i} & C_{\phi_i} S_{\theta_i} C_{\psi_i} + S_{\phi_i} S_{\psi_i} \\ S_{\phi_i} C_{\theta_i} & S_{\phi_i} S_{\theta_i} S_{\psi_i} + C_{\phi_i} C_{\psi_i} & S_{\phi_i} S_{\theta_i} C_{\psi_i} - C_{\phi_i} S_{\psi_i} \\ -S_{\theta_i} & C_{\theta_i} S_{\psi_i} & C_{\theta_i} C_{\psi_i} \end{bmatrix}$$

and $S$ and $C$ are shorthand for cos and sin.

#### 2.1.2 Velocity Update

Using a relativistic correction:

$$v_{i+1} = v_i + a_i \sqrt{1 - \frac{v_i^2}{V_c^2}}$$

where $V_c = 299,792,458\text{m/s}$.

#### 2.1.3 Orientation Update

$$\begin{bmatrix} \phi_{i+1} \\ \theta_{i+1} \\ \psi_{i+1} \end{bmatrix} = \begin{bmatrix} \phi_i \\ \theta_i \\ \psi_i \end{bmatrix} + \begin{bmatrix} 1 & \sin\phi_i \tan\theta_i & \cos\phi_i \tan\theta_i \\ 0 & \cos\phi_i & -\sin\phi_i \\ 0 & \frac{\sin\phi_i}{\cos\theta_i} & \frac{\cos\phi_i}{\cos\theta_i} \end{bmatrix} \begin{bmatrix} \omega_{x,i} \\ \omega_{y,i} \\ \omega_{z,i} \end{bmatrix}$$

Where $J(\phi, \theta)$ is:

$$J(\phi, \theta) = \begin{bmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \frac{\sin\phi}{\cos\theta} & \frac{\cos\phi}{\cos\theta} \end{bmatrix}$$

### 2.1.4 Jacobian of the Motion Model

We define the Jacobian matrix:

$$F_i = \frac{\partial g}{\partial X_i} \in \mathbb{R}^{7\times 7}$$

We now compute each partial derivative.

#### Position Rows (1–3)

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \\ z_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + \begin{bmatrix} \cos\phi_i \cos\theta_i \left(v_i \Delta t + \frac{1}{2} a \Delta T^2\right) \\ \sin\phi_i \cos\theta_i \left(v_i \Delta t + \frac{1}{2} a \Delta T^2\right) \\ -\sin\theta_i \left(v \Delta T + \frac{1}{2} a \Delta T^2\right) \end{bmatrix}$$

Taking partial derivatives, rows 1-3 of the Jacobian are computed,

$$\begin{bmatrix} 1 & 0 & 0 & \cos\phi_i \cos\theta_i \Delta T & -\sin\phi_i \cos\theta_i \left(v_i \Delta T + \frac{1}{2} a_i \Delta T^2\right) & -\cos\phi_i \sin\theta_i \left(v_i \Delta T + \frac{1}{2} a_i \Delta T^2\right) & 0 \\ 0 & 1 & 0 & \sin\phi_i \cos\theta_i \Delta T & \cos\phi_i \cos\theta_i \left(v_i \Delta T + \frac{1}{2} a_i \Delta T^2\right) & -\sin\phi_i \sin\theta_i \left(v_i \Delta T + \frac{1}{2} a_i \Delta T^2\right) & 0 \\ 0 & 0 & 1 & -\sin\theta_i \Delta T & 0 & -\cos\theta_i \left(v_i \Delta T + \frac{1}{2} a_i \Delta T^2\right) & 0 \end{bmatrix}$$

#### Velocity Row (4)

$$v_{i+1} = v_i + a_i \sqrt{1 - \frac{v_i^2}{V_c^2}}$$

$$\frac{\partial v_{i+1}}{\partial v_i} = 1 + a_i \cdot \frac{1}{2} \cdot \left(1 - \frac{v_i^2}{V_c^2}\right)^{-\frac{1}{2}} \cdot \left(-\frac{2v_i}{V_c^2}\right) = 1 - a_i v_i \left(1 - \frac{v_i^2}{V_c^2}\right)^{-\frac{1}{2}} \cdot \frac{1}{V_c^2}$$

$$\frac{\partial v_{i+1}}{\partial v_i} = 1 - \frac{a_i v_i}{{V_c}^2 \sqrt{1 - \frac{v_i^2}{V_c^2}}}$$

Taking partial derivatives, row 4 of the Jacobian is computed,

$$\begin{bmatrix} 0 & 0 & 0 & 1 - \dfrac{a_i v_i}{{V_c}^2 \sqrt{1 - \dfrac{v_i^2}{V_c^2}}} & 0 & 0 & 0 \end{bmatrix}$$

#### Orientation Rows (5–7)

The component-wise update is,

$$\begin{aligned} \phi_{i+1} &= \phi_i + \omega_{x,i} + \omega_{y,i} \sin\phi_i \tan\theta_i + \omega_{z,i} \cos\phi_i \tan\theta_i \\ \theta_{i+1} &= \theta_i + \cos\phi_i \cdot \omega_{y,i} - \sin\phi_i \cdot \omega_{z,i} \\ \psi_{i+1} &= \psi_i + \frac{\sin\phi_i}{\cos\theta_i} \cdot \omega_{y,i} + \frac{\cos\phi_i}{\cos\theta_i} \cdot \omega_{z,i} \end{aligned}$$

Taking partial derivatives, rows 5-7 of the Jacobian are computed,

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 + C\phi_i \tan\theta_i \cdot \omega_{y,i} - S\phi_i \tan\theta_i \cdot \omega_{z,i} & S\phi_i \sec^2\theta_i \cdot \omega_{y,i} + C\phi_i \sec^2\theta_i \cdot \omega_{z,i} & 0 \\ 0 & 0 & 0 & 0 & -S\phi_i \cdot \omega_{y,i} - C\phi_i \cdot \omega_{z,i} & 1 & 0 \\ 0 & 0 & 0 & 0 & \frac{C\phi_i}{C\theta_i} \cdot \omega_{y,i} - \frac{S\phi_i}{C\theta_i} \cdot \omega_{z,i} & S\phi_i \sec\theta_i \tan\theta_i \cdot \omega_{y,i} + C\phi_i \sec\theta_i \tan\theta_i \cdot \omega_{z,i} & 1 \end{bmatrix}$$

#### Final Motion Jacobian

Combining the rows of the motion Jacobian derived above, the full motion Jacobian can be constructed. The full representation is omitted here due to it not fitting within the page margins.

### 2.1.5 Jacobian of the Measurement Model

The Jacobian matrix of the measurement model is:

$$H_i = \frac{\partial h}{\partial X_i}$$

Since the measurement is a direct observation of $x_i, y_i, z_i, \phi_i$, and does not depend on $v_i, \theta_i, \phi_i$, the Jacobian matrix is simply:

$$H_i = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{4\times 7}$$

## 2.2 EKF Implementation

The EKF was implemented as a class in Python. This is an adaptation of the implementation used by group 5 in the course project, which is an adaptation of the implementation used in the MTE544 labs.

```python
import numpy as np

class EKF:
    def __init__(self, P, Q, R, x, dt, Vc=299792458):
        self.P = P
        self.Q = Q
        self.R = R
        self.x = x
        self.dt = dt
        self.Vc = Vc
        self.K = None

    def predict(self, u):
        self.A = self.jacobian_A(u)  # Compute state transition Jacobian
        self.motion_model(u)        # Predict state using motion model
        self.H = self.jacobian_H()  # Measurement Jacobian
        self.P = self.A @ self.P @ self.A.T + self.Q  # Prediction covariance

    def update(self, z):
        self.K = self.P @ self.H.T @ np.linalg.inv(
            self.H @ self.P @ self.H.T + self.R
        )  # Kalman gain
        self.x = self.x + self.K @ (z - self.measurement_model())  # State update
        I = np.eye(self.A.shape[0])
        self.P = (I - self.K @ self.H) @ self.P @ (I - self.K @ self.H).T + \
                 self.K @ self.R @ self.K.T  # Covariance update
```

```
def motion_model(self, u):
    dt = self.dt
    a, wx, wy, wz = u
    x, y, z, v, phi, theta, psi = self.x

    R_mat = self.rotation_matrix(phi, theta, psi)
    D = v * dt + 0.5 * a * dt**2

    # Position
    pos_update = R_mat[:, 0] * D
    x_new = x + pos_update[0]
    y_new = y + pos_update[1]
    z_new = z + pos_update[2]

    # Velocity
    v_new = v + a / np.sqrt(1 - (v / self.Vc)**2)

    # Orientation update using the provided Jacobian J:
    phi_new = phi + wx + np.sin(phi) * np.tan(theta) * wy + \
              np.cos(phi) * np.tan(theta) * wz
    theta_new = theta + np.cos(phi) * wy - np.sin(phi) * wz
    psi_new = psi + (np.sin(phi) / np.cos(theta)) * wy + \
              (np.cos(phi) / np.cos(theta)) * wz

    self.x = np.array([x_new, y_new, z_new, v_new, phi_new, theta_new, psi_new])

def rotation_matrix(self, phi, theta, psi):
    cphi = np.cos(phi)
    sphi = np.sin(phi)
    ctheta = np.cos(theta)
    stheta = np.sin(theta)
    cpsi = np.cos(psi)
    spsi = np.sin(psi)

    R = np.array([
        [cphi * ctheta, cphi * stheta * spsi - sphi * cpsi,
         cphi * stheta * cpsi + sphi * spsi],
        [sphi * ctheta, sphi * stheta * spsi + cphi * cpsi,
         sphi * stheta * cpsi - cphi * spsi],
        [-stheta, ctheta * spsi, ctheta * cpsi]
    ])
    return R

def jacobian_A(self, u):
    dt = self.dt
    a, wx, wy, wz = u
    x, y, z, v, phi, theta, psi = self.x
    D = v * dt + 0.5 * a * dt**2
    Vc = self.Vc

    A = np.zeros((7, 7))

    # Position
```

```
        # x
        A[0, 0] = 1
        A[0, 3] = dt * np.cos(phi) * np.cos(theta)
        A[0, 4] = -np.sin(phi) * np.cos(theta) * D
        A[0, 5] = -np.cos(phi) * np.sin(theta) * D

        # y
        A[1, 1] = 1
        A[1, 3] = dt * np.sin(phi) * np.cos(theta)
        A[1, 4] = np.cos(phi) * np.cos(theta) * D
        A[1, 5] = -np.sin(phi) * np.sin(theta) * D

        # z
        A[2, 2] = 1
        A[2, 3] = -dt * np.sin(theta)
        A[2, 4] = 0
        A[2, 5] = -np.cos(theta) * D

        # Velocity
        A[3, 3] = 1 + a * v / (Vc**2 * np.sqrt(1 - (v / Vc)**2))

        # Orientation
        # phi
        A[4, 4] = 1 + np.cos(phi) * np.tan(theta) * wy - \
                  np.sin(phi) * np.tan(theta) * wz
        A[4, 5] = (np.sin(phi) / (np.cos(theta)**2)) * wy + \
                  (np.cos(phi) / (np.cos(theta)**2)) * wz

        # theta
        A[5, 4] = -np.sin(phi) * wy - np.cos(phi) * wz
        A[5, 5] = 1

        # psi
        A[6, 4] = (np.cos(phi) / np.cos(theta)) * wy - \
                  (np.sin(phi) / np.cos(theta)) * wz
        A[6, 5] = (np.sin(phi) * np.sin(theta) / (np.cos(theta)**2)) * wy + \
                  (np.cos(phi) * np.sin(theta) / (np.cos(theta)**2)) * wz
        A[6, 6] = 1

        return A

    def jacobian_H(self):
        H = np.zeros((4, 7))
        H[0, 0] = 1
        H[1, 1] = 1
        H[2, 2] = 1
        H[3, 4] = 1
        return H

    def measurement_model(self):
        x, y, z, v, phi, theta, psi = self.x
        return np.array([x, y, z, phi])
    def get_state(self):
        return self.x
```

An example of how to run the EKF:

```
for i in range(1, N):
    u_i = u.iloc[i].values.astype(float)
    ekf.predict(u_i)
    z_i = z.iloc[i].values.astype(float)
    ekf.update(z_i)
    x_est_store[i, :] = ekf.get_state()
```

## 2.3 Tuning Q and R

The Kalman filter is run on the provided data with,

$$Q_0 = \text{diag}([1, 1, 1, 2.5, 0.5, 0.5, 0.5])$$

$$R_0 = \text{diag}([500, 500, 500, 1])$$

the state estimation and actual state are plotted in the figures below.

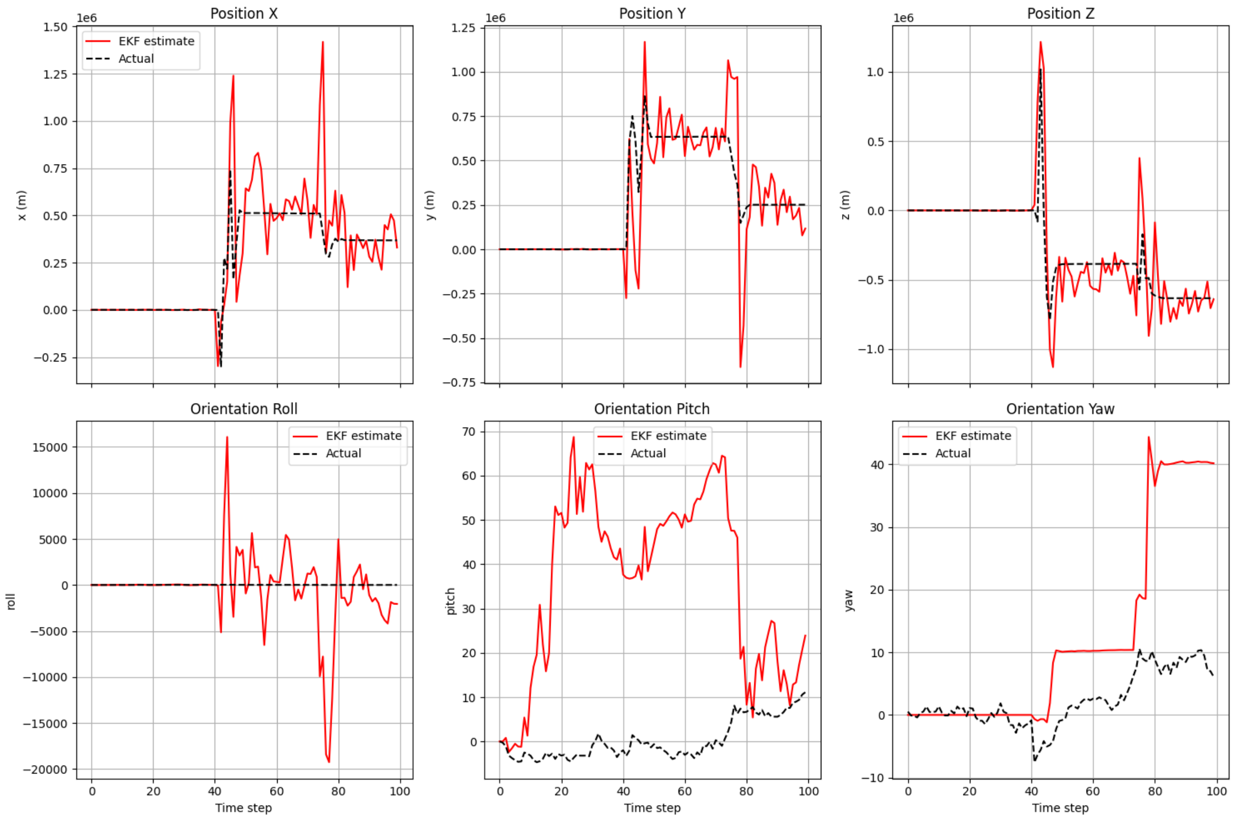

Figure 5: Estimated and actual state for untuned EKF.

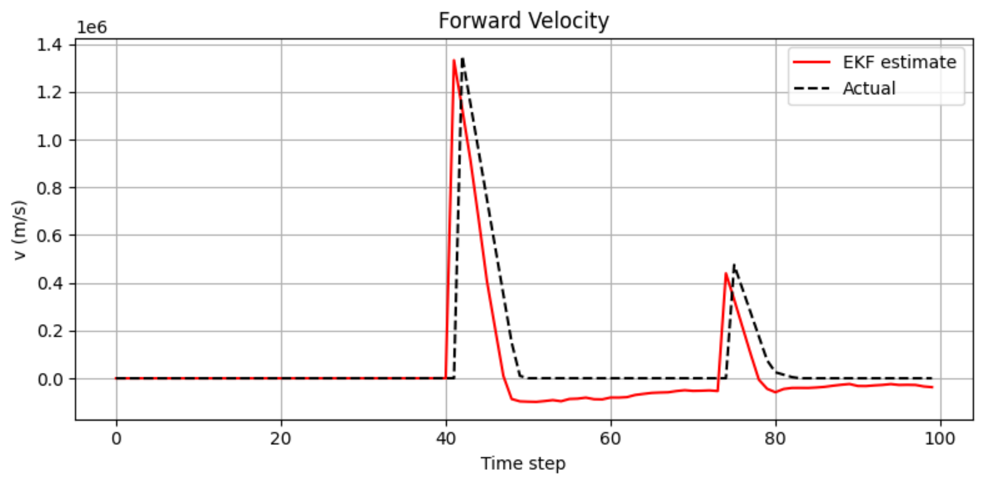

Figure 6: Estimated and actual velocity for untuned EKF.

The tracking for the positions and velocity seem to roughly follow the actual state, but the orientation, notably roll, deviates excessively from the actual state. The RMSE of the state estimation was,

Table 2: RMSE for each state component

| **State** | **RMSE** |
|---|---|
| x | $2.27 \times 10^5$ |
| y | $2.00 \times 10^5$ |
| z | $2.42 \times 10^5$ |
| v | $1.58 \times 10^5$ |
| roll | $1.60 \times 10^2$ |
| pitch | $2.09 \times 10^1$ |
| yaw | $2.83 \times 10^1$ |

An attempt was made to tune these parameters using a optimization algorithm which minimizes the error between the predicted and actual state, however these attempts failed. This is likely due to the massive non-nonlinearities introduced by the probability drive as well as the massive differences in scale between the filter components at various time slices. Instead the filter was tuned by tweaking each diagonal of the Q and R matrices until performance improved. The final values were,

$$Q_f = \text{diag}([1, 1, 1, 2.5, 0.5, 5.0, 5.0])$$
$$R_f = \text{diag}([5000, 5000, 5000, 100])$$

The performance of the filter with these update parameters is show in the figures below.

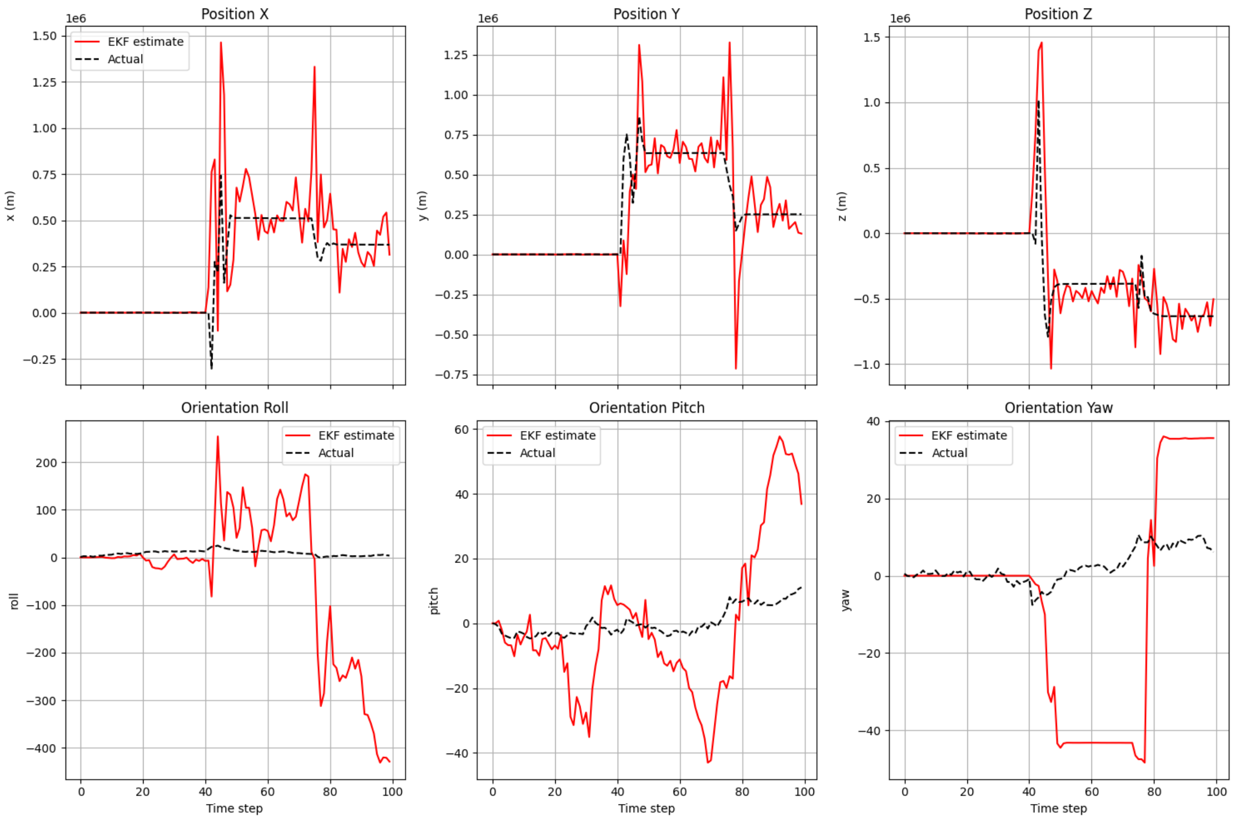

Figure 7: Estimated and actual state of tuned EKF.

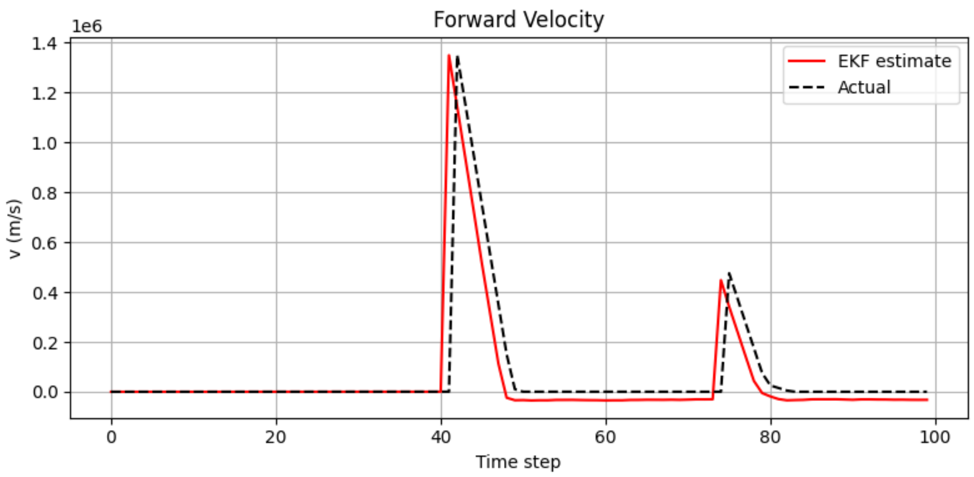

Figure 8: Estimated and actual velocity of tuned EKF.

There was a massive improvement in the estimate of $\phi$ by placing much less trust in the measurement of $\phi$. Similarly, the position estimations track much more closely due to the lower emphasis placed on the measurements for these components. The velocity state estimation qualitatively improved moderately, likely as a down stream side-effect of the other improvements. $\theta$ and $\psi$ saw negligible improvements, as the measurement has no direct impact on these components so they tend to accumulate error in the process model. With this new initialization, decreasing the trust in the process model for $\theta$ and $\psi$ stabilized their performances. The rest of the Q parameters did not show any particular improvements with tuning.

The RSME of the state estimations for the untuned and tuned cases are shown in the table below.

| **State** | **Untuned RMSE** | **Tuned RMSE** |
|---|---|---|
| $x$ | $2.27 \times 10^5$ | $1.97 \times 10^5$ |
| $y$ | $2.00 \times 10^5$ | $1.97 \times 10^5$ |
| $z$ | $2.42 \times 10^5$ | $2.15 \times 10^5$ |
| $v$ | $1.58 \times 10^5$ | $1.69 \times 10^5$ |
| $roll$ | $1.60 \times 10^2$ | $4.13 \times 10^3$ |
| $pitch$ | $2.09 \times 10^1$ | $4.18 \times 10^1$ |
| $yaw$ | $2.83 \times 10^1$ | $1.58 \times 10^1$ |

The qualitative performance observations are mostly reflected quantitatively.

## 2.4 Adaptive Q and R

The closed form adaptive Q and R matrices are,

$$Q_{\text{adaptive}} = Q_0 \cdot \text{prob}$$

$$R_{\text{adaptive}} = R_0 \cdot \frac{\|z\|}{\text{prob}}$$

*prob* is set to 1 always except for when an improbable event occurs, in which case *prob* takes the value of the probability of the improbable event. This is because it the amount of energy released by such an event is proportional to it's improbability, but in the normal case we don't care about the magnitude of the probability. When an improbable even occurs, the ships dynamics become extremely aggressive, which makes the sensor measurements unreliable, so we divide the baseline R matrix by *prob* which is a number close to zero and multiply Q by *prob* to place more trust in the process model which is driven by the control action. After an improbable event, the ships speed greatly increases even long after the event. This increases the magnitude of the variance of the measurement noise, which is accounted for my multiplying R by the norm of the measurement vector. $\|z\|$ is the norm of the measurement vector.

## 2.5 EKF1 and EKF2 Estimation Comparison

The $X(1,:)$ component of the state estimation for both filters is plotted along side the true state and the measurement of $X(1,:)$ in Figure 9 below.

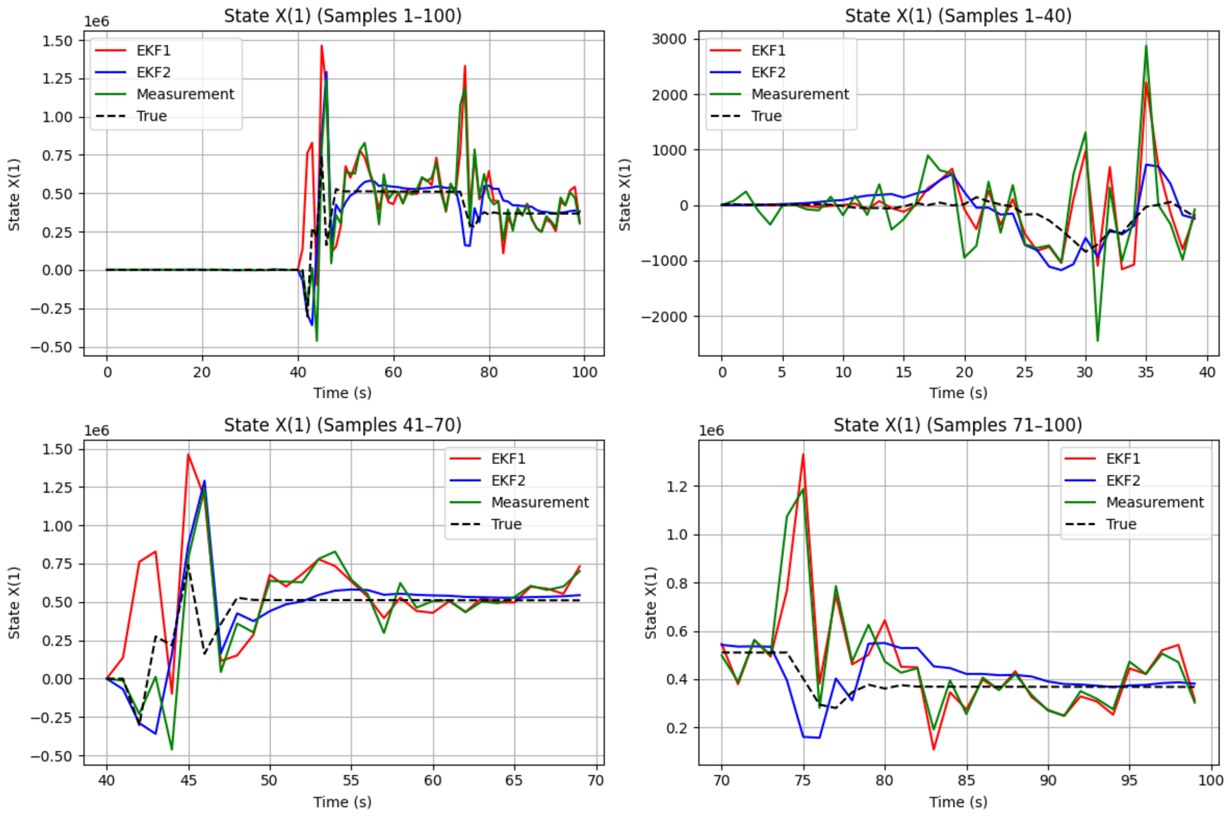

Figure 9: Comparing x position estimation for EKF1 and EKF2

Looking at the plots above it is noted that,

- In the begining of the 1-40 plot, the two filters perform similarly, with EKF1 occasionally out-performing EKF2 when the variance of the noise is low. However, more often EKF1 tracks the noise more closely and deviates further from the true state than EKF2, as is evident as the magnitude of the noise increases. We see that as the noise increases, both filters get pulled away from the true state, but EKF2 does a better job or tracking the true state.
- At the beginning of the 41-70 plot, we can see that EKF2 is right on to of the measurement signal, whereas EKF1 deviates from the true state even more aggressively than the measurement. Towards the end of this plot, EKF2 begins to closely track the true state again, whereas EKF1 continues to follow the measurement noise.
- In the 71-100 plot, There is a large spike in them measurement which EKF2 mostly ignores and tracks the true state, where EKF1 is stays right on top of the measurement for the duration of the plot.

The $X(4,:)$ component of the state estimation for both filters is plotted along side the true state and the measurement of $X(4,:)$ in Figure 11 below.

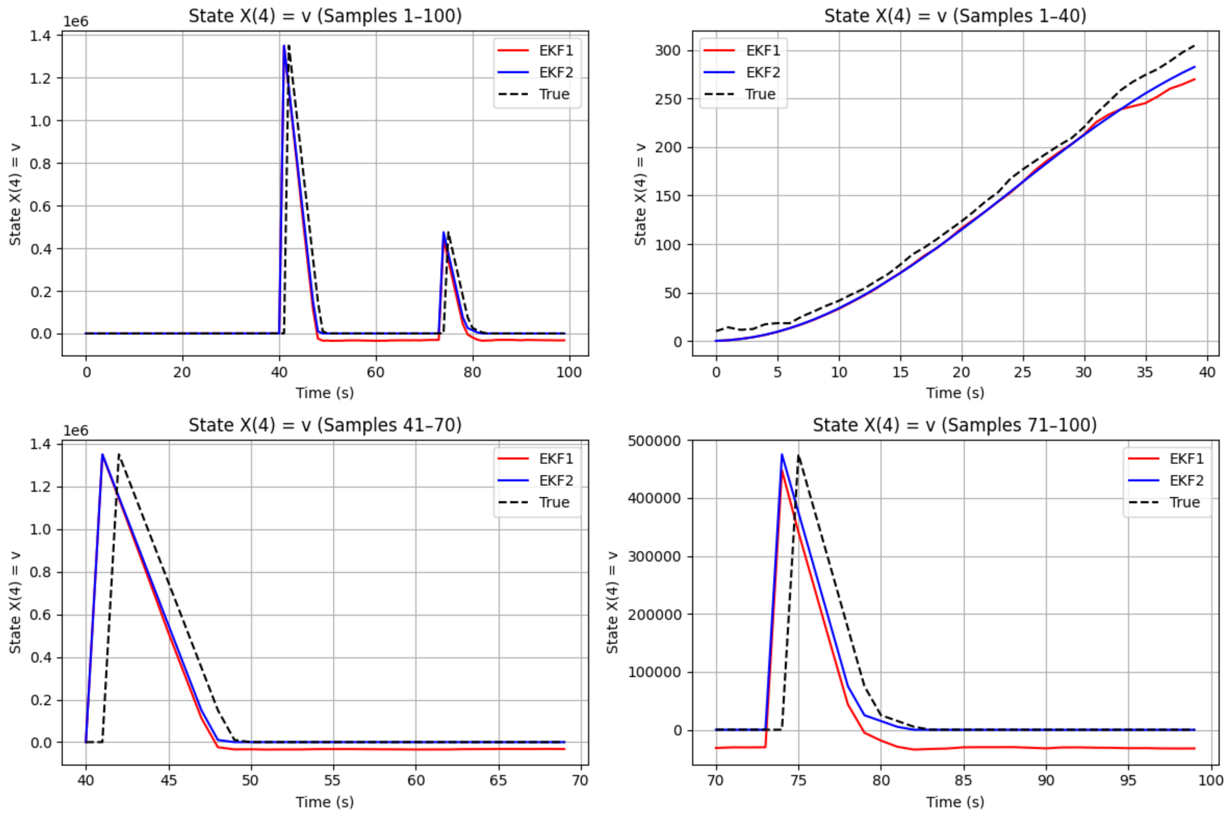

Figure 10: Comparing velocity estimation for EKF1 and EKF2

Looking at the plots above it is noted that,

- In the 1-40 plot there was negligible difference between the two state estimators.
- In the 41-70 plot, both EKF1 and EKF2 have the same phase delay tracking the true velocity, but once the velocity reaches a steady state, EKF2 does a better job of correcting the steady state error than EKF1.
- In the 71-100 plot, both filters have some delay tracking the transient but EKF2 does a better job tracking at steady state.

Based on the observations of the state estimator's performances for $X(1,:)$ and $X(4,:)$, EKF2 out performed EKF1. Both filters had some trouble tracking the true state during highly transient periods, however, in almost all cases EKF2 was able to more closely track the true state and reject measurement noise.

Looking at the state estimation errors more quantitatively, some error metrics for $X(1,:)$ and $X(4,:)$ are shown in the table below.

| **Metric** | **State** | **EKF1** | **EKF2** |
|---|---|---|---|
| RMSE | $X(1,:)$ | $2.28 \times 10^5$ | $1.44 \times 10^5$ |
| Mean Error | $X(1,:)$ | $-5.52 \times 10^4$ | $-1.46 \times 10^4$ |
| Std Dev | $X(1,:)$ | $2.21 \times 10^5$ | $1.43 \times 10^5$ |
| RMSE | $X(4,:)$ | $1.58 \times 10^5$ | $1.54 \times 10^5$ |
| Mean Error | $X(4,:)$ | $1.86 \times 10^4$ | $4.29 \times 10^1$ |
| Std Dev | $X(4,:)$ | $1.57 \times 10^5$ | $1.54 \times 10^5$ |

In terms of quantitative performance, EKF2 had a lower state estimation error compared to EKF1 in every case.

## 2.6 EKF1 and EKF2 Kalman Gains Comparison

The Kalman gain norms for both estimators are plotted in Figure 11 below.

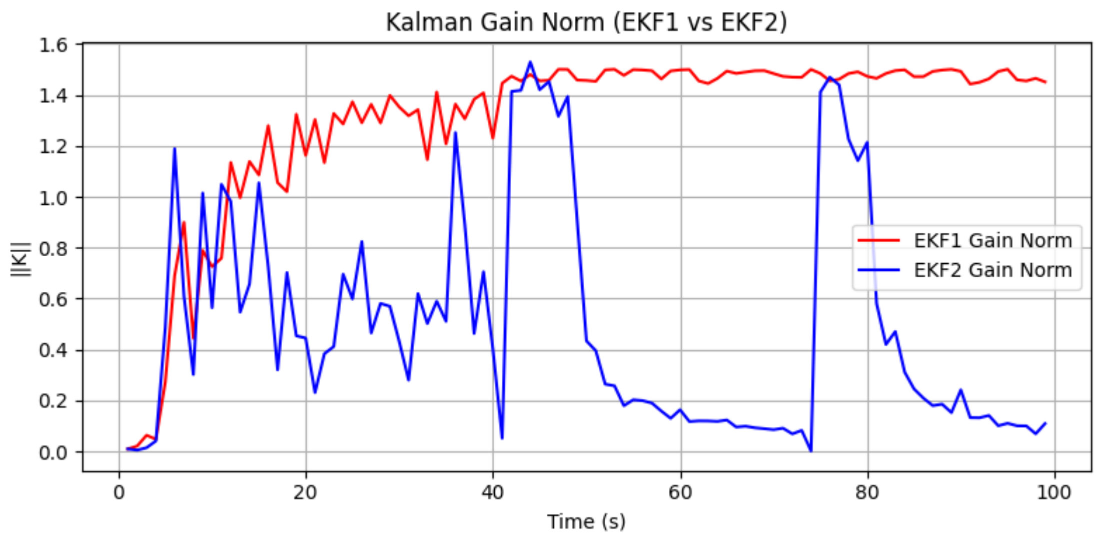

Figure 11: Kalamn gain norm for EKF1 and EKF2.

EKF1 starts with a low gain and then appears to asymptotically converge to about a gain of 1.5. Looking at EKF2, the Kalman gain is much more dynamic, jumping between values as high as 1.5 and as low as 0.1 throughout the dataset. The mean and standard deviation of the Kalman gain for both filters, across various segments of the data is summarized in the table below.

| **Range** | **EKF1** | **EKF2** |
|---|---|---|
| 1–40 | $1.0288 \pm 0.4222$ | $0.5646 \pm 0.3026$ |
| 41–70 | $1.4787 \pm 0.0194$ | $0.4849 \pm 0.5399$ |
| 71–99 | $1.4761 \pm 0.0173$ | $0.4188 \pm 0.4785$ |

EKF1 consistently has a mean gain that is 2-3X larger than EKF2 across all segments, while in all cases accept the 1-40 range, EKF2 has a higher std by about a factor of 10. EKF1 only has a slightly higher std in the 1-4 range as it is still converging, but in the later segments it remains stable. Meanwhile, as mentioned before, EKF2 remains dynamic. This is reflected in what was observed in previous sections where as the filters progress, EKF1 closely followed the measurement noise whereas EKF2 was able to reject more of the noise and more closely track the true state. EKF1's consistently higher gain implies that the filter places more trust in the measurements throughout trial, whereas EKF2 trusts the sensors more adaptively.

## 2.7 Strengths ans Weaknesses

A weakness of the design of EKF1 is that in order to be able to reject enough of the noise during and after the improbable event periods, the measurement covariances need to be large but this leads to the filter over trusting the measurements in the more stable tracking cases which leads to worse performance overall compared toe EKF2. A strength of EKF2 is that it is able to avoid this by adaptively scaling Q and R in response to higher magnitude sensor readings and improbable events. A weakness of EKF2 is that it requires an additional input to the filter, which in this case is accessible but may not always be the case. At the very beginning of the the 1-40 plot for the estimation of $X(1,:)$, it appears that EKF2's lower baseline R values make it too trusting of the sensor measurements, which leads to worse performance. In a case where the ship was moving slowly, EKF2 may perform worse overall.

## 2.8 Improving the State Estimator

The current state estimator based on the Extended Kalman Filter struggles to handle the nonlinearities and scale differences in the Improbability Drive system. In particular, the EKF relies on first-order linearization of the dynamics and measurement models, which can be inadequate when the system undergoes abrupt changes or when sensor measurements become unreliable. To address these challenges, several alternative approaches and enhancements are suggested below.

### 2.8.1 UKF and CKF

One alternative is to adopt a filtering method that does not rely on explicit linearization. The Unscented Kalman Filter (UKF) and the Cubature Kalman Filter (CKF) propagate a set of sigma (or cubature) points through the full nonlinear system dynamics. This approach provides a more accurate approximation of the true posterior distribution, particularly in scenarios where the nonlinearity is severe.

### 2.8.2 Particle Filtering

In cases where the state distribution is highly non-Gaussian or even multimodal, particle filters offer a viable solution. Particle filters approximate the posterior distribution using a set of weighted samples, which allows for flexible modeling of complex distributions that cannot be well represented by a Gaussian.

### 2.8.3 Incorporating Data-Driven Methods

An alternative strategy is to develop hybrid state estimators that combine classical model-based approaches with data-driven techniques. For example, one may integrate an EKF or UKF with a neural network trained on historical data to learn the residual error dynamics or unmodeled behaviors.

## 2.9 EKF1 and EKF2 Further Results

Estimates for the 101-150 time steps were generated and saved to `ImprobDrive_20736975.csv`, however and issue was encountered since $u$ only contains 149 values. An assumption was made that the data should be appended with a row of zeros.

# 3 Feedforward Neural Network for Sensor Fusion in Wind Turbine Condition Monitoring

## 3.1 Seperate Predictions

### 3.1.1 Model Architecture

Two separate feedforward neural networks (ANN1 and ANN2) were designed for classification using nacelle and bearing accelerometer data, respectively. Each network consisted of the following architecture:

- Input Layer: 3 neurons corresponding to the three-axis accelerometer readings.
- Hidden Layer 1: 16 neurons with ReLU activation, followed by a Dropout layer with a rate of 0.3.
- Hidden Layer 2: 8 neurons with ReLU activation, followed by a Dropout layer with a rate of 0.3.
- Output Layer: 2 neurons with Softmax activation for binary classification.

This compact architecture was selected to ensure sufficient model capacity while avoiding overfitting, given the relatively small input dimensionality and size of the dataset.

### 3.1.2 Preprocessing

Input features from both accelerometers were standardized using z-score normalization:

$$X' = \frac{X - \mu}{\sigma}$$

where $\mu$ and $\sigma$ denote the mean and standard deviation computed from the training data. Standardization ensures that each input feature contributes equally during gradient-based optimization, improving convergence stability.

### 3.1.3 Loss Function and Output Encoding

The output labels were one-hot encoded as,

$$\text{Label } 0 \to [1, 0], \quad \text{Label } 1 \to [0, 1]$$

and the categorical cross-entropy loss function was used to evaluate prediction performance,

$$\mathcal{L} = -\sum_{i=1}^{N} \sum_{j=1}^{2} y_{ij} \log(\hat{y}_{ij})$$

This loss function is standard for multi-class classification problems with softmax outputs, providing well-calibrated probability estimates.

### 3.1.4 Optimizer and Training Procedure

Training was performed using the Adam optimizer with a learning rate of 0.001. The Adam optimizer was selected due to its adaptive learning rate capabilities and strong empirical performance across a wide range of neural network training tasks. Higher learning rates tended to make ANN2 perform worse, while lower learning rates had negligible performance impact. The batch size was set to 128, and training was allowed for up to 200 epochs. Batch sizes smaller than 128 tended to perform worse while batch sizes larger than 128 yielded no improvements. However, early stopping was employed based on validation accuracy with a patience of 20 epochs to prevent overfitting and to automatically restore the best model weights. The training and validation accuracies at each epoch are plotted in Figure 12 below. The test accuracy vs epochs was requested in the assignment, however there are no labels for the test set.

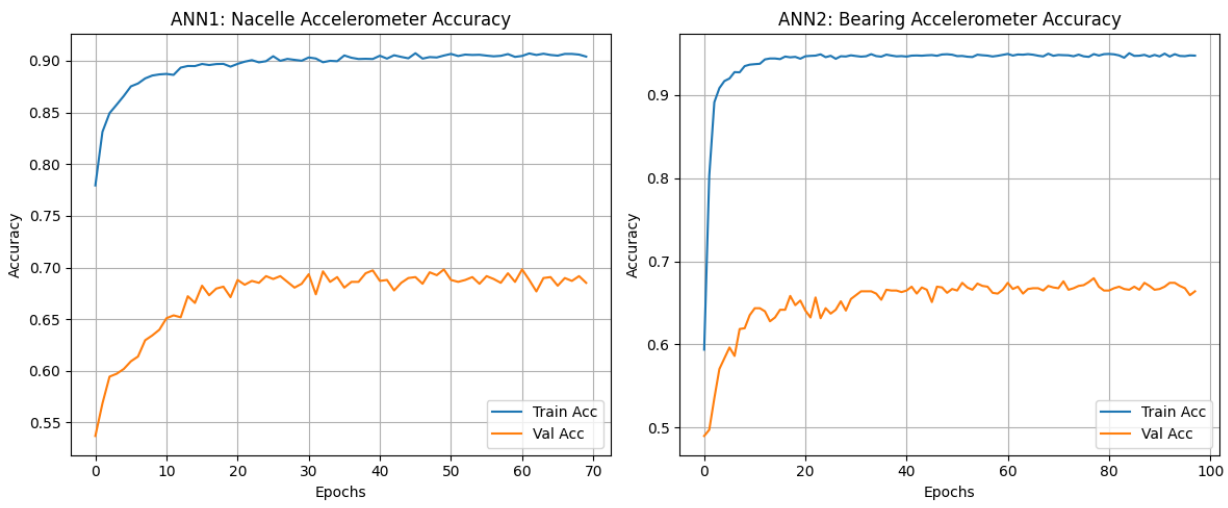

Figure 12: Training and validation accuracy for both NNs.

This configuration provided consistent convergence and validation performance exceeding the target threshold of 67% accuracy. The confusion matrices for each NN were also plotted in Figure 13

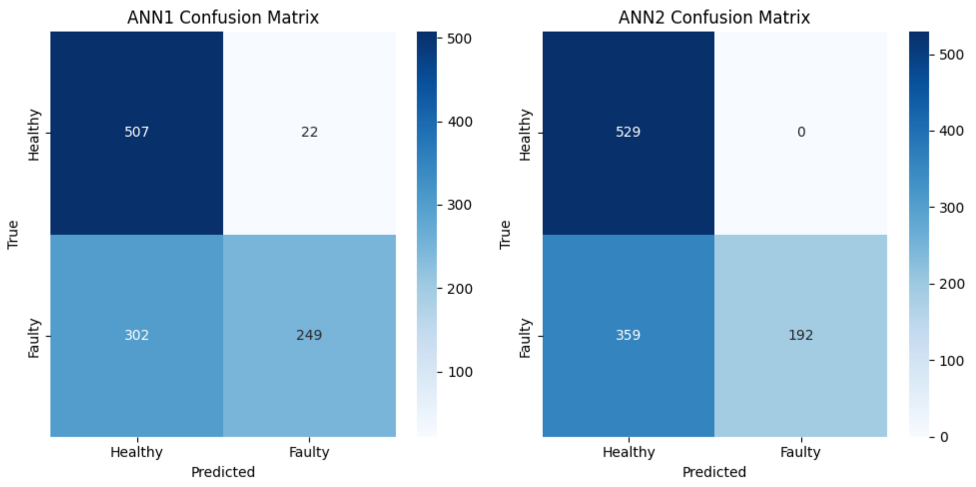

Figure 13: Training and validation accuracy for both NNs.

ANN1 and ANN2 suffer from a high number of false negatives, indicating a tendency to under-detect faults. Both networks exhibits good specificity, with very few false positives. This suggests the networks are highly conservative and only predict the healthy state when very confident, however this results in a substantial number of false negatives.

## 3.2 Fused Predictions

The early fusion model (ANN3) was implemented with the exact same architecture as ANN1 and ANN2, but the size of the input layer was expanded to 6 to accommodate both sets of accelerometer data. The late fusion model was implemented by using the same training procedures for ANN1 and ANN2, but callbacks were added which saved the training and validation predictions at each epoch for each network. The results were then fused to generate the the accuracy vs epoch plots. The training and validation accuracies for both fusion methods are show in Figure 14.

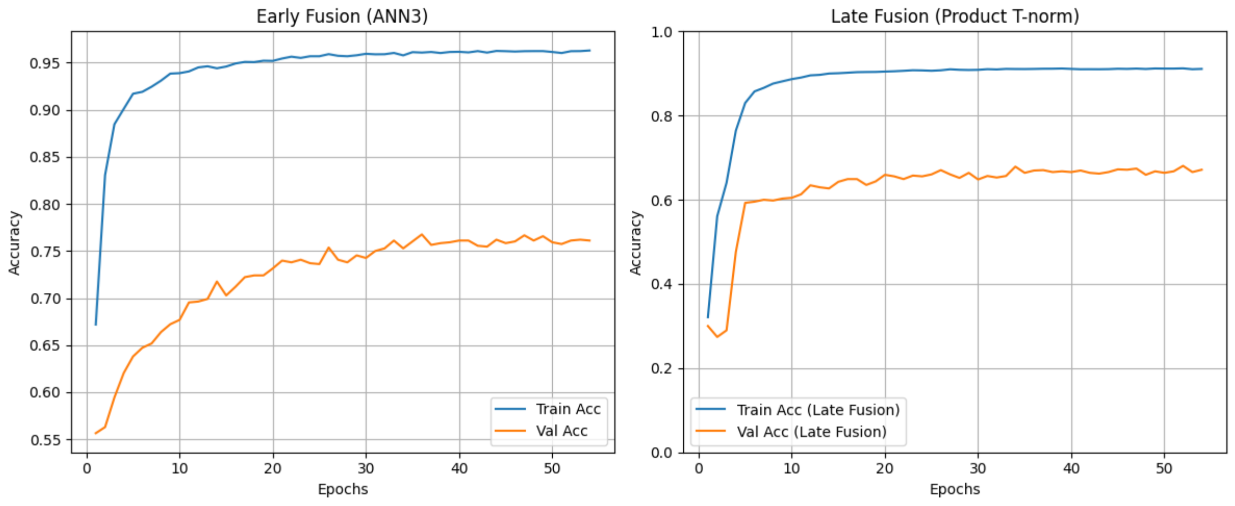

Figure 14: Training and validation accuracy for fusion architectures.

The confusion matrices were also plotted in Figure 15

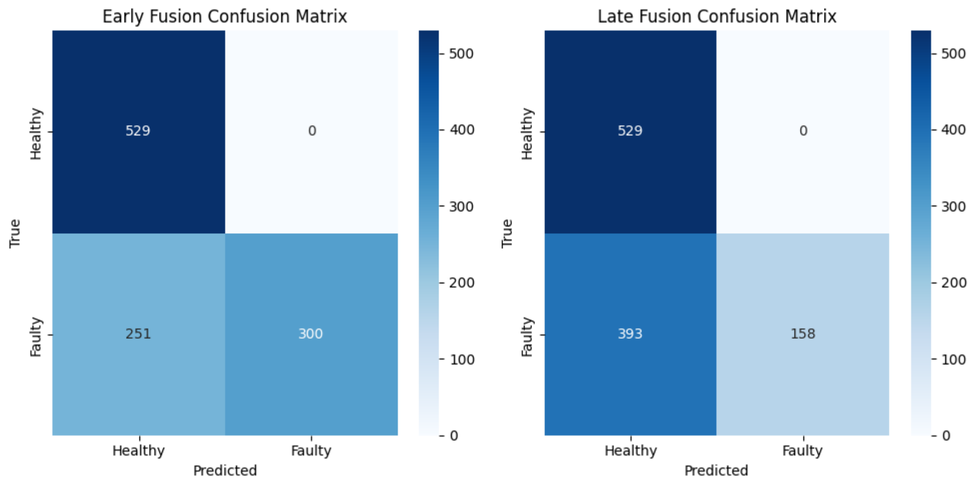

Figure 15: Confusion matrices for fusion architectures.

The learning and confusion plots above both indicate that the early fusion architecture outperformed the late fusion architecture. The late fusion architecture appears to have performed slightly worse than than ANN1 or ANN2. Instead of removing uncertainty, fusing the outputs through product T-norm seems to have added the uncertainty of both models. This is reflected in the lower overall validation accuracy as well as in the confusion matrix. In the confusion matrix, the late fusion architecture has more false negatives than either ANN1 or ANN2, suggesting it has captured the error of both models. ANN3 on the other hand, outperformed both ANN1 and ANN2, achieving a higher validation accuracy and a cleaner confusion matrix. This suggests that exposing the network to more axes of data during training allowed the model to learn the relationships between separate data streams and make better predictions. From this it is clear that ANN3 is the superior fusion method.