NTNU

Kunnskap for en bedre verden

MSc Mechatronics and Automation

AIS4004 Digital Twins for Predictive Maintenance

Individual Project

Author:
Elisha Adimalara Adiburo

March 25, 2025

Adaptive Extended Kalman Filter (AEKF) Implementation for Otter Actuator Fault Diagnosis

March 25, 2025

1. Introduction

This report details the implementation of an **Adaptive Extended Kalman Filter (AEKF)** for diagnosing actuator faults on the Otter vessel. The Otter's dynamics are described by a six-state model comprising the vessel's position, heading, and body-fixed velocities. We incorporate an actuator fault model and demonstrate how the AEKF is used to estimate both the vessel's states and the fault parameters in real time.

The code that implements this solution is appended as part of the **Appendix**, along with the generated plots that illustrate the simulation results. The main objectives, as per the problem statement, include:

1. Formulating the **discrete-time model** of the Otter.
2. Formulating the **actuator fault diagnosis problem**.
3. Implementing and testing the **Adaptive Kalman Filter** (in this case, an Adaptive Extended Kalman Filter) to diagnose the faults, and plotting/discussing the results.

2. Otter Vessel Model

2.1 Continuous-Time Dynamics

The Otter's motion is governed by the standard 3-DOF (Degrees of Freedom) equations for surge (u), sway (v), and yaw (r). The state vector is defined as:

$$\eta = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix}, \quad \nu = \begin{bmatrix} u \\ v \\ r \end{bmatrix},$$

where:

- η represents the vessel's position (x, y) in a North-East-Down (NED) frame and the heading ψ.
- ν represents the body-fixed velocities in surge (u) and sway (v) and the yaw rate (r).

The continuous-time dynamic model can be written as:

1. **Kinematic equation**:
$$\dot{\eta}(t) = R(\psi)\,\nu(t),$$
where $R(\psi)$ is the standard 2D rotation matrix (extended to 3×3 by including a 1 in the bottom-right corner).

2. **Kinetic equation**:
$$M\,\dot{\nu}(t) = -\big(C_M(\nu) + D_M(\nu)\big)\,\nu(t) \;+\; B_\tau\,\tau_c(t),$$
with:

   - M as the mass-inertia matrix (including added mass terms).
   - $C_M(\nu)$ as the Coriolis and centripetal matrix.
   - $D_M(\nu)$ as the damping matrix, which includes linear and nonlinear damping terms.
   - B_τ as the input mapping matrix (often the identity or a selection matrix).
   - τ_c as the control input vector, $\tau_c = [\tau_u, \tau_v, \tau_r]^T$.

2.2 Discrete-Time Approximation

To implement a Kalman filter in a simulation, we need a **discrete-time** version of these equations. We use a simple **Euler method** to obtain:

$$X_{k+1} = \big(I + \Delta t\,A_c(X_k)\big)\,X_k \;+\; \Delta t\,B_c\,\tau_c \;+\; (\text{fault and noise terms}),$$

where:

- $X = \begin{bmatrix}\eta \\ \nu\end{bmatrix}$ is the 6-dimensional state vector.
- $A_c(X_k)$ is the linearization of the continuous-time dynamics at step k.
- Δt is the simulation time step.

In the final code, the discrete-time system matrices A_d and B_d are computed at each time step, capturing the local linearization around the current state estimate.

3. Actuator Fault Diagnosis Problem

3.1 Fault Modeling

We assume that the control inputs τ_c (surge, sway, yaw commands) can be corrupted by additive faults. A simple additive fault model can be expressed as:

$$\tau_{\text{actual}} = \tau_c + \Psi(\theta),$$

where θ is a vector of fault parameters, and $\Psi(\theta)$ represents how these fault parameters enter the dynamics. In the code, this is often implemented as a diagonal or direct injection mapping:

$$X_{k+1} = A_d X_k + B_d\,\tau_c + \Psi_d\,\theta + \ldots$$

By estimating θ, we can detect and diagnose actuator faults (e.g., partial actuator failure or drift).

3.2 Adaptive Extended Kalman Filter

Because the system is nonlinear (due to the rotation matrix, Coriolis terms, and damping terms), we use an **Extended Kalman Filter** (EKF). Additionally, we make the EKF *adaptive* by updating the process and measurement noise covariance matrices online, and by estimating the fault parameters θ in tandem with the system states.

Key steps include:

1. **Prediction** using the linearized state matrix F_X.
2. **Update** using the measurement matrix C (which is often the identity in a full-state measurement scenario).
3. **Adaptive noise update**, which recalculates or adapts Q and R based on the filter residuals.
4. **Fault parameter estimation** via an augmented approach, adding the fault parameters into the EKF state or using an additional adaptation law.

4. Control Inputs and Scheduling

An important part of this project is how we **schedule** or **select** control inputs. Rather than abruptly switching the inputs, we employ **cosine blending** to create smooth transitions. The piecewise control inputs for surge, sway, and yaw are defined at specific time segments:

- **Time breakpoints**: $t_{\text{control}} = [0, 3, 6, 9, 12, 15]$
- **Profile values**: For each segment, we specify a starting and ending value (e.g., from -30 N to 200 N in surge).

Within each segment, the code blends the inputs via:

$$\tau_u(t) = \tau_u^{\text{start}} + \frac{\tau_u^{\text{end}} - \tau_u^{\text{start}}}{2} \left[1 - \cos\Big(\pi \frac{t - t_{\text{start}}}{t_{\text{end}} - t_{\text{start}}}\Big)\right].$$

This ensures that the control signals change smoothly, reducing transients that might otherwise complicate the fault detection.

5. Implementation Overview

1. **Initialization**
   - Define simulation parameters (Δt, final time T).
   - Initialize states (η, ν), fault parameters (θ), and noise covariance matrices (QF, RF).
   - Generate **smooth control profiles** $\tau_u(t)$, $\tau_v(t)$, $\tau_r(t)$ using cosine blending.
2. **True State Simulation**

- We run a "true" simulation loop (without the filter) to record ground-truth states.
- This loop also injects known fault changes at specified times (e.g., at 50% and 75% of the simulation) to simulate real actuator degradation or failure.

3. **AEKF Estimation**
   - For each time step:
     (a) *Compute* the rotation matrix and linearized dynamics around the current estimate.
     (b) *Predict* the state and covariance.
     (c) *Incorporate* measurement updates (assuming full-state measurement with additive noise).
     (d) *Adapt* the noise matrices QF and RF based on filter residuals.
     (e) *Update* the fault parameter estimates θ.
   - Save the estimated states and faults for plotting.

4. **Result Visualization**
   - **Position and Trajectory**: Compare the estimated (x, y) to the true (x, y).
   - **Body-Fixed Velocities**: Compare the estimated (u, v, r) to the true velocities.
   - **Fault Parameters**: Compare the estimated fault parameters $\hat{\theta}$ to the true values.
   - **Control Inputs**: Plot the final control profiles that were applied.

6. Discussion of Results

Although the figures are provided in the Appendix, we highlight the key observations from each:

1. **Vessel Trajectory (Figure 1)**
   - The black line ("True Trajectory") represents the actual path of the Otter.
   - The red dashed line ("Estimated Trajectory") follows closely, indicating good performance of the AEKF in estimating position and heading.
   - The green pentagrams mark the control change points. Notice how the trajectory changes direction or speed around these instants.

2. **Body-Fixed Velocities (Figure 2)**
   - The plots for surge (u), sway (v), and yaw rate (r) show the true values (in black) versus the estimated values (in red).
   - The estimates track the true velocities fairly well, with minor deviations during abrupt changes or fault injection times.

3. **Fault Parameter Estimation (Figure 3)**

- The black lines ("True Value") represent the actual fault magnitudes scheduled at 50% and 75% of the simulation.
- The red dashed lines ("Estimated Value") show how quickly the AEKF adapts and converges to the correct fault parameters.
- Small overshoots or transients might occur, but the filter stabilizes around the true fault levels.

4. **Control Inputs (Figure 4)**
   - Each subplot shows the smoothed control commands over time.
   - The cosine blending ensures a gentle, sinusoidal-like transition between command segments, reducing discontinuities.
   - Surge input (τ_u) exhibits the largest amplitude changes, while sway (τ_v) and yaw (τ_r) have smaller ranges.

From these results, we can conclude:

- The **AEKF** successfully detects and tracks changes in the actuator fault parameters.
- The **smooth control profile** helps keep the system stable and avoids sharp transients that might confuse or destabilize the filter.
- The **state estimates** (position, heading, velocities) remain accurate, demonstrating robust performance of the AEKF in the presence of noise and faults.

7. Answers to Key Questions

Referring to the attached PDF, we address the primary tasks:

1. **Formulate the Discrete-Time Model**
   We derived the discrete-time model by applying a first-order (Euler) discretization to the continuous-time equations of motion. Specifically,

$$X_{k+1} = X_k + \Delta t \left[f(X_k, \tau_c(k)) \right],$$

   plus the linearization that leads to A_d and B_d in the code. This satisfies the requirement to capture the system in a discrete-time form suitable for Kalman filtering.

2. **Formulate the Actuator Fault Diagnosis Problem**
   The fault is introduced as an additive term $\Psi_d \theta$ in the input channel. By treating θ as an unknown parameter vector, we augment the state or use an adaptation law to estimate it in real time. This approach directly addresses actuator faults by comparing the commanded control to the actual effect on the system.

3. **Test and Implement the Adaptive Kalman Filter**
   We employed an **Extended Kalman Filter** to handle the nonlinear dynamics, augmented with an **adaptive** scheme that updates QF and RF based on the filter residuals. The simulation plots (in the Appendix) illustrate the filter's performance in tracking the vessel's states and the fault parameters. The discussion above confirms the filter converges and accurately diagnoses the introduced actuator faults.

8. Conclusion

This report presents a comprehensive approach to **actuator fault diagnosis** for the Otter vessel using an **Adaptive Extended Kalman Filter (AEKF)**. By carefully selecting smooth control inputs via cosine blending, we ensure a stable and interpretable simulation scenario. The results confirm that:

- The AEKF can reliably estimate both the vessel states and the fault magnitudes.
- Smooth transitions in the control inputs minimize abrupt dynamics, enhancing fault detectability.
- The filter's adaptive noise updates help it remain robust against measurement and process uncertainties.

The appended **MATLAB code** and **plots** provide a full demonstration of the methodology and confirm its effectiveness.

Appendix

1 Simulation Results

Below presents the simulation results through four figures showing different aspects of the system's behavior.

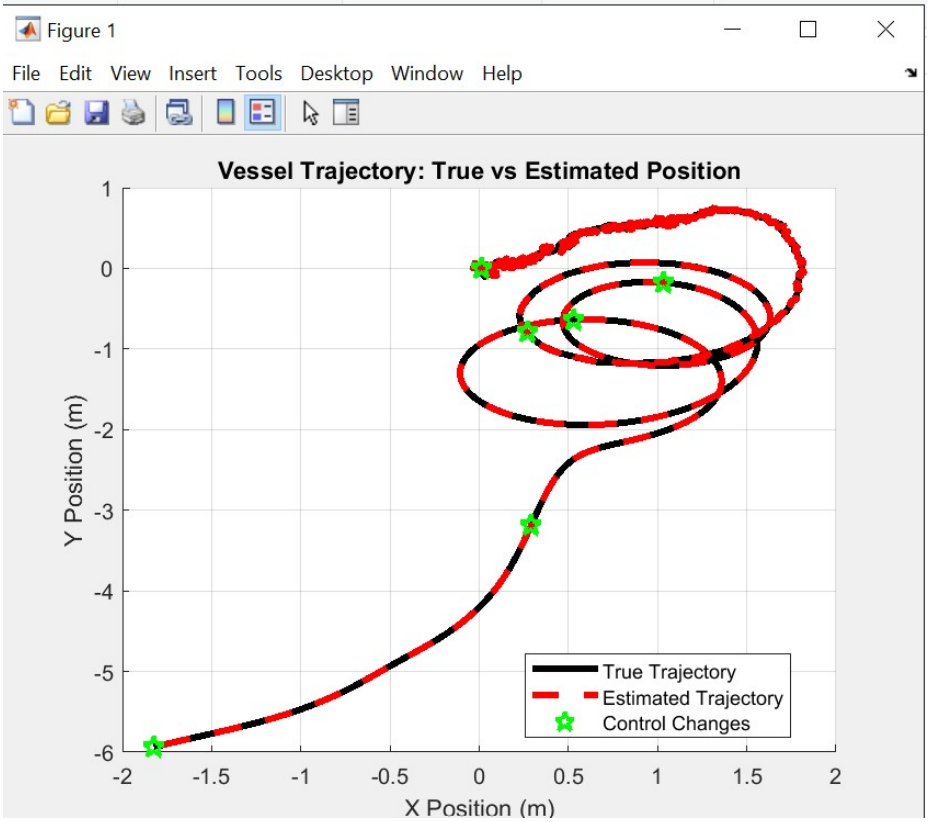

Figure 1: Comparison of true vessel trajectory and AEKF estimated trajectory. Markers indicate control change events.

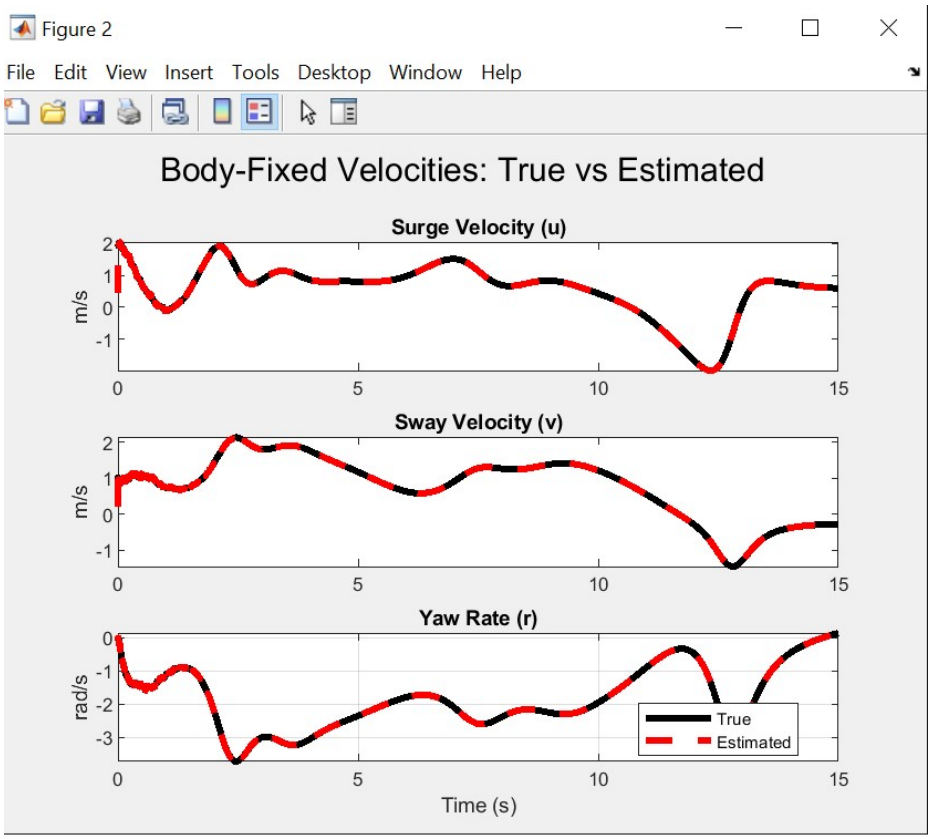

Figure 2: Time evolution of body-fixed velocities: surge (u), sway (v), and yaw rate (r). Solid lines show true values, dashed lines show estimates.

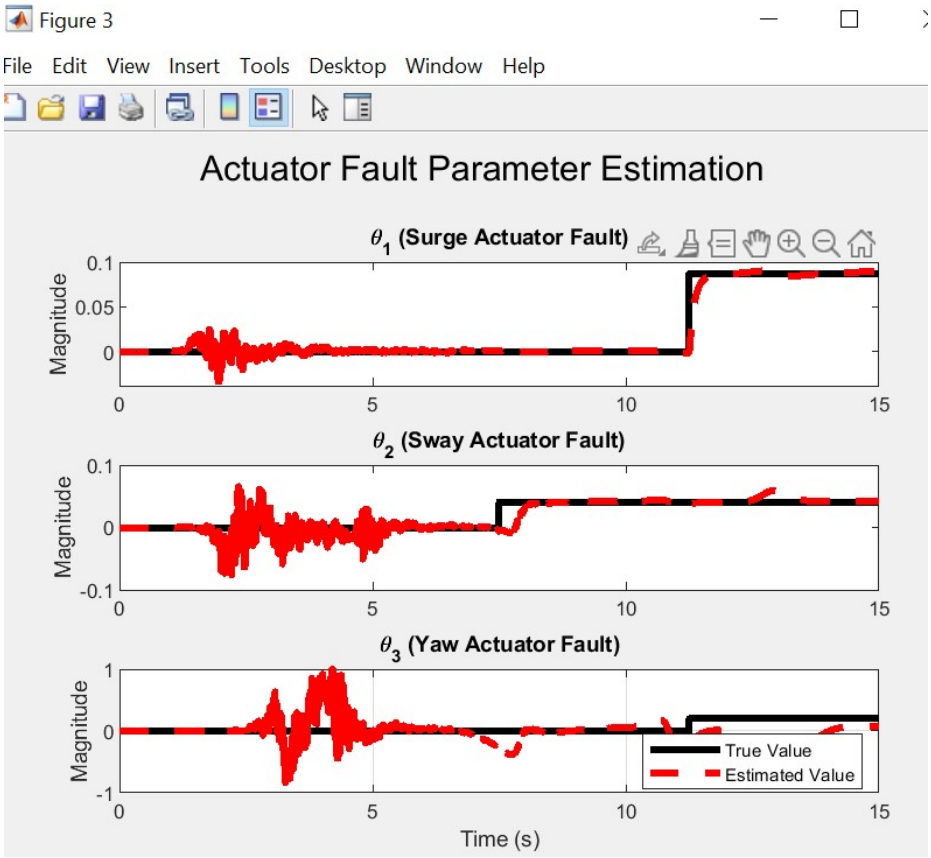

Figure 3: Estimation of actuator fault parameters using AEKF. True values (solid) vs estimated values (dashed).

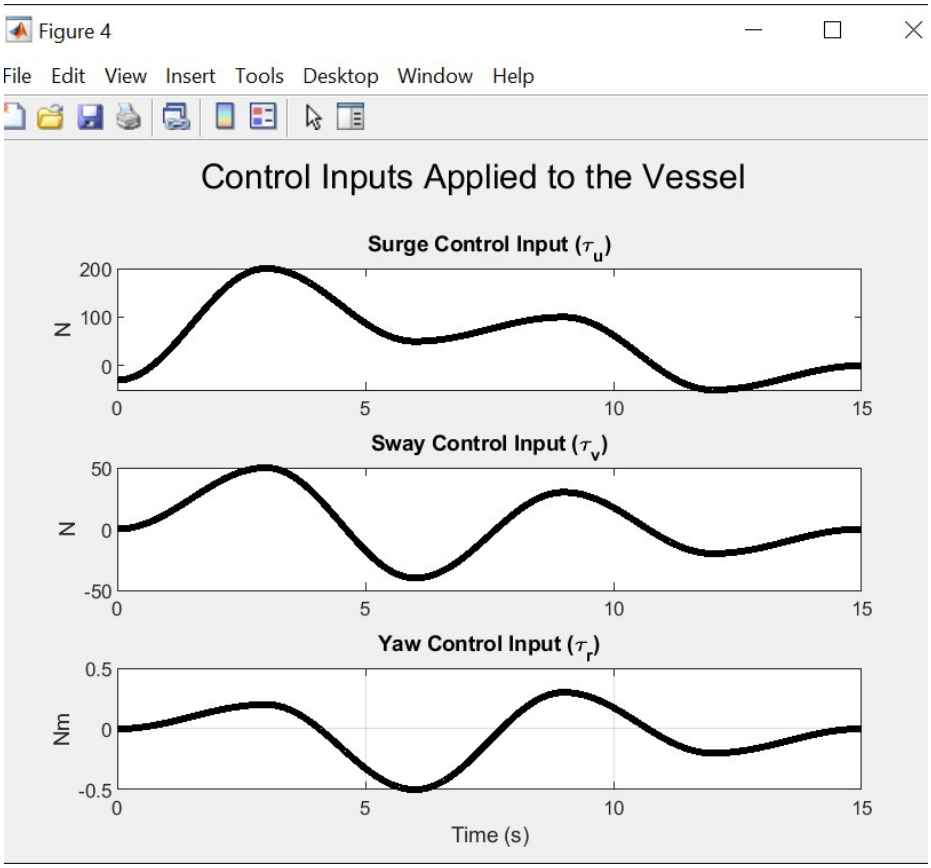

Figure 4: Time history of control inputs applied to the vessel: surge (τ_u), sway (τ_v), and yaw (τ_r) commands.

A MATLAB Code Listing

Below is the complete MATLAB code used for the simulation and AEKF implementation:

```
%% Otter Actuator Fault Diagnosis using AEKF (Modified Solution)
% This script implements an adaptive extended Kalman filter (AEKF) for diagnosing
% actuator faults on a vessel with enhanced control scheduling and improved visualization.

clear; clc;

%% Parameters
Xu      = -0.7225;    Xuu     = -1.3274;    m       = 23.8;      X_udot = -2.0;
Yv      = -0.8612;    Yvv     = -36.2823;   Yv_dot  = -10.0;     xg     = 0.046;
Yr      = 0.1079;     Nrr     = -1;         Yr_dot  = 0.0;       Nv_dot = 0.0;
Nv      = 0.1052;     Nr      = -0.5;       Izz     = 1.76;      Nr_dot = -1.0;

%% Time Variables
dt = 0.001;
T  = 15;
t  = 0:dt:T;    % Start at 0 to allow valid indexing

%% Noise Matrices
QF = 10 * eye(6);
RF = 40 * eye(6);

%% Initial Values
x   = 0; y   = 0; yaw = 0;
u   = 2; v   = 1; r   = 0;
tau_u = -30; tau_v = 0; tau_r = 0;
eta   = [x; y; yaw];              % vessel position and heading
nu    = [u; v; r];                % body-fixed velocities
tau_c = [tau_u; tau_v; tau_r];    % initial control input
tau_true = tau_c;
X     = [eta; nu];                % full state vector
X_true = X;
xhat    = zeros(6,1);             % initial state estimate
theta   = zeros(3,1);             % true fault parameter
thetahat = zeros(3,1);            % fault parameter estimate

%% Modified Control Unit with Smooth Transitions
% Define piecewise control inputs with smooth transitions using cosine blending
t_control = [0, 3, 6, 9, 12, 15];             % Control change timepoints (s)
tau_u_profile = [ -30, 200, 50, 100, -50, 0 ];  % Surge control profile
tau_v_profile = [ 0, 50, -40, 30, -20, 0 ];     % Sway control profile
tau_r_profile = [ 0, 0.2, -0.5, 0.3, -0.2, 0 ];  % Yaw control profile

% Preallocate control vectors
tau_u = zeros(size(t));
tau_v = zeros(size(t));
tau_r = zeros(size(t));
```

```
47
48 % Generate smooth control signals using cosine interpolation
49 for i = 1:length(t_control)-1
50     idx = t >= t_control(i) & t < t_control(i+1);
51     tau_u(idx) = (tau_u_profile(i+1) - tau_u_profile(i))/2 * ...
52         (1 - cos(pi*(t(idx)-t_control(i))/(t_control(i+1)-t_control(i))))
               + tau_u_profile(i);
53     tau_v(idx) = (tau_v_profile(i+1) - tau_v_profile(i))/2 * ...
54         (1 - cos(pi*(t(idx)-t_control(i))/(t_control(i+1)-t_control(i))))
               + tau_v_profile(i);
55     tau_r(idx) = (tau_r_profile(i+1) - tau_r_profile(i))/2 * ...
56         (1 - cos(pi*(t(idx)-t_control(i))/(t_control(i+1)-t_control(i))))
               + tau_r_profile(i);
57 end
58
59 tau_c = [tau_u; tau_v; tau_r]; % Full control input vector based on
       smooth transitions
60
61 %% Tuning Parameters
62 lambda = 0.995;
63 a      = 0.999;
64
65 %% System Matrices
66 A = eye(6);
67 M = [ m - X_udot,          0,           0;
68       0,           m - Yv_dot,  m*xg - Yr_dot;
69       0,           m*xg - Nv_dot, Izz - Nr_dot ];
70 m11 = M(1,1); m22 = M(2,2); m23 = M(2,3); m32 = M(3,2);
71 B_c = [ zeros(3); inv(M)*eye(3) ];  % control input matrix
72 C   = eye(6);
73 Psi_d = -dt * B_c * diag(tau_c);     % fault injection mapping
74
75 %% Estimation Parameters
76 Pplus = eye(rank(A));
77 S = 0.1 * eye(3);
78 UpsilonPlus = 0 * B_c;  % zero matrix (same size as B_c)
79
80 %% Preallocate Arrays for Plotting
81 x_true_vec   = zeros(6, length(t));
82 eta_vec      = zeros(3, length(t));
83 nu_vec       = zeros(3, length(t));
84 tau_c_vec    = zeros(3, length(t));
85 theta_vec    = zeros(3, length(t));
86 thetahat_vec = zeros(3, length(t));
87 xhat_vec     = zeros(6, length(t));
88
89 %% True State Simulation
90 for i = 1:length(t)
91     % Schedule a change in true actuation at 25% and 50% of the
          simulation:
92     if i == round(0.25 * length(t))
93         tau_true = [200; 50; 0.2];
94     end
95     if i == round(0.5 * length(t))
```

```
        tau_true = [50; -40; -0.5];
    end

    % Compute rotation matrix based on current true yaw:
    R = [ cos(X_true(3)), -sin(X_true(3)), 0;
          sin(X_true(3)),  cos(X_true(3)), 0;
          0, 0, 1 ];

    % Compute hydrodynamic matrices (based on true state)
    C_M = [ 0, 0, -M(2,2)*X_true(5) - 0.5*(M(2,3)+M(3,2))*X_true(6);
            0, 0, M(1,1)*X_true(4);
            M(2,2)*X_true(5) + 0.5*(M(2,3)+M(3,2))*X_true(6),
               -M(1,1)*X_true(4), 0 ];
    D_M = -[ Xu + Xuu*abs(X_true(4)), 0, 0;
             0, Yv + Yvv*abs(X_true(5)), Yr;
             0, Nv, Nr + Nrr*abs(X_true(6)) ];
    A_c = [ zeros(3), R;
            zeros(3), -inv(M)*(C_M+D_M) ];

    % Discretize continuous dynamics:
    A_d = eye(size(A_c)) + dt * A_c;
    B_d = dt * B_c;

    % Update the true state with process noise:
    X_true = A_d * X_true + B_d * tau_true + QF * dt *
       randn(size(X_true));
    x_true_vec(:, i) = X_true;
end

%% Simulation and AEKF Estimation
for i = 1:length(t)
    % Get current control input from smooth profile
    current_tau_c = [tau_u(i); tau_v(i); tau_r(i)];

    % Schedule fault parameter changes
    if i == round(0.5 * length(t))
        theta = [0; 0.041; 0];
    end
    if i == round(0.75 * length(t))
        theta = [0.087; 0.041; 0.2];
    end

    % Update system matrices based on current simulation state (eta, nu):
    R = [ cos(eta(3)), -sin(eta(3)), 0;
          sin(eta(3)),  cos(eta(3)), 0;
          0, 0, 1 ];
    C_M = [ 0, 0, -M(2,2)*nu(2) - 0.5*(M(2,3)+M(3,2))*nu(3);
            0, 0, M(1,1)*nu(1);
            M(2,2)*nu(2) + 0.5*(M(2,3)+M(3,2))*nu(3), -M(1,1)*nu(1), 0 ];
    D_M = -[ Xu + Xuu*abs(nu(1)), 0, 0;
             0, Yv + Yvv*abs(nu(2)), Yr;
             0, Nv, Nr + Nrr*abs(nu(3)) ];
    A_c = [ zeros(3), R;
            zeros(3), -inv(M)*(C_M+D_M) ];
```

```
148
149     % Discretize the system:
150     A_d = eye(size(A_c)) + dt * A_c;
151     B_d = dt * B_c;
152
153     % Fault injection mapping based on current control:
154     Psi_d = -B_d * diag(current_tau_c);
155
156     % Update the simulation state with fault injection and noise:
157     X = A_d * X + B_d * current_tau_c + Psi_d * theta + QF * dt *
            randn(size(X));
158     eta = X(1:3);
159     nu  = X(4:6);
160
161     % Measurement (full state with noise):
162     y = C * X + RF * dt * randn(size(X));
163
164     %% Adaptive Extended Kalman Filter (AEKF) Update
165     % Shortcut variables for dynamics linearization:
166     sig2 = 0.5 * (m23 + m32);
167     sig1 = xhat(5) * m22 + xhat(6) * sig2;
168
169     % Compute nudot (3x3 matrix for dynamic states)
170     nudot = -inv(M) * [ Xu + 2*Xuu*abs(xhat(4)),          xhat(6)*m22,
                    sig1 + xhat(6)*sig2;
171                            -xhat(6)*m11,                    Yv +
                           2*Yvv*abs(xhat(5)), Yr - xhat(4)*m11;
172                          xhat(5)*m11 - sig1,  Nv + xhat(4)*m11 -
                             xhat(4)*m22,  Nr + 2*Nrr*abs(xhat(6)) -
                             xhat(4)*sig2 ];
173
174     % Construct the Jacobian FX (6x6)
175     % Top block (3x6) based on kinematics:
176     top_block = [ 0, 0, -xhat(5)*cos(xhat(3)) - xhat(4)*sin(xhat(3)),
            cos(xhat(3)), -sin(xhat(3)), 0;
177                   0, 0,  xhat(4)*cos(xhat(3)) - xhat(5)*sin(xhat(3)),
                       sin(xhat(3)),  cos(xhat(3)), 0;
178                   0, 0, 0, 0, 0, 1 ];
179     % Bottom block (3x6) with zeros in the first three columns:
180     bottom_block = [ zeros(3,3), nudot ];
181     FX = A + dt * [ top_block; bottom_block ];
182
183     % Kalman Filter prediction update:
184     Pmin = FX * Pplus * FX' + QF;
185     Sigma = C * Pmin * C' + RF;
186     KF = Pmin * C' / Sigma;
187     Pplus = (eye(rank(A)) - KF * C) * Pmin;
188
189     % Innovation:
190     ytilde = y - C * xhat;
191
192     % Adaptive updates for noise covariances:
193     QF = a * QF + (1 - a) * (KF * (ytilde * ytilde') * KF');
194     RF = a * RF + (1 - a) * (ytilde * ytilde' + C * Pmin * C');
```

```
195
196     % Fault parameter estimation gain update:
197     Upsilon = (eye(rank(A)) - KF * C) * FX * UpsilonPlus + (eye(rank(A))
            - KF * C) * Psi_d;
198     Omega   = C * FX * UpsilonPlus + C * Psi_d;
199     Lambda_mat = inv(lambda * Sigma + Omega * S * Omega');
200     Gamma = S * Omega' * Lambda_mat;
201     S = (1 / lambda) * S - (1 / lambda) * S * Omega' * Lambda_mat * Omega
            * S;
202     UpsilonPlus = Upsilon;
203
204     % Update fault parameter estimate:
205     thetahat = thetahat + Gamma * (y - C * xhat);
206
207     % Update state estimate:
208     xhat = A_d * xhat + B_d * current_tau_c + Psi_d * thetahat + QF * dt
            * randn(size(X)) ...
209             + KF * (y - C * xhat) + Upsilon * Gamma * (y - C * xhat);
210
211     % Store data for plotting:
212     eta_vec(:, i)      = X(1:3);
213     nu_vec(:, i)       = X(4:6);
214     tau_c_vec(:, i)    = current_tau_c;
215     xhat_vec(:, i)     = xhat;
216     theta_vec(:, i)    = theta;
217     thetahat_vec(:, i) = clamp(thetahat);
218 end
219
220 %% Enhanced Visualization with Descriptive Titles
221 % Figure 1: Vessel Trajectory
222 figure(1)
223 clf;
224 hold on
225 plot(eta_vec(1,:), eta_vec(2,:), 'k', 'LineWidth', 3)
226 plot(xhat_vec(1,:), xhat_vec(2,:), 'r--', 'LineWidth', 3)
227 plot(eta_vec(1, round(t_control/dt)+1), eta_vec(2,
        round(t_control/dt)+1), ...
228     'gpentagram', 'MarkerSize', 10, 'LineWidth', 2)
229 title('Vessel Trajectory: True vs Estimated Position')
230 xlabel('X Position (m)')
231 ylabel('Y Position (m)')
232 legend('True Trajectory', 'Estimated Trajectory', 'Control Changes',
        'Location', 'best')
233 grid on
234
235 % Figure 2: Body-Fixed Velocities
236 figure(2)
237 clf;
238 subplot(3,1,1)
239 plot(t, nu_vec(1,:), 'k', t, xhat_vec(4,:), 'r--', 'lineWidth',3)
240 title('Surge Velocity (u)')
241 ylabel('m/s')
242
243 subplot(3,1,2)
```

```
plot(t, nu_vec(2,:), 'k', t, xhat_vec(5,:), 'r--', 'lineWidth',3)
title('Sway Velocity (v)')
ylabel('m/s')

subplot(3,1,3)
plot(t, nu_vec(3,:), 'k', t, xhat_vec(6,:), 'r--', 'lineWidth',3)
title('Yaw Rate (r)')
ylabel('rad/s')
xlabel('Time (s)')
sgtitle('Body-Fixed Velocities: True vs Estimated')
legend('True', 'Estimated', 'Location', 'best')
grid on

% Figure 3: Fault Parameter Estimation
figure(3)
clf;
subplot(3,1,1)
plot(t, theta_vec(1,:), 'k', t, thetahat_vec(1,:), 'r--', 'lineWidth',3)
title('\theta_1 (Surge Actuator Fault)')
ylabel('Magnitude')

subplot(3,1,2)
plot(t, theta_vec(2,:), 'k', t, thetahat_vec(2,:), 'r--', 'lineWidth',3)
title('\theta_2 (Sway Actuator Fault)')
ylabel('Magnitude')

subplot(3,1,3)
plot(t, theta_vec(3,:), 'k', t, thetahat_vec(3,:), 'r--', 'lineWidth',3)
title('\theta_3 (Yaw Actuator Fault)')
ylabel('Magnitude')
xlabel('Time (s)')
sgtitle('Actuator Fault Parameter Estimation')
legend('True Value', 'Estimated Value', 'Location', 'best')
grid on

% Figure 4: Control Inputs
figure(4)
clf;
subplot(3,1,1)
plot(t, tau_c(1,:), 'k', 'LineWidth', 3)
title('Surge Control Input (\tau_u)')
ylabel('N')

subplot(3,1,2)
plot(t, tau_c(2,:), 'k', 'LineWidth', 3)
title('Sway Control Input (\tau_v)')
ylabel('N')

subplot(3,1,3)
plot(t, tau_c(3,:), 'k', 'LineWidth', 3)
title('Yaw Control Input (\tau_r)')
ylabel('Nm')
xlabel('Time (s)')
sgtitle('Control Inputs Applied to the Vessel')
```

```
grid on

%% Clamping Function
function clamped_arr = clamp(arr)
    clamped_arr = arr;
    for i = 1:length(arr)
        x = arr(i);
        if x > 1
            x = 1;
        elseif x < -1
            x = -1;
        end
        clamped_arr(i) = x;
    end
end
```

Listing 1: MATLAB script for Otter Actuator Fault Diagnosis using AEKF