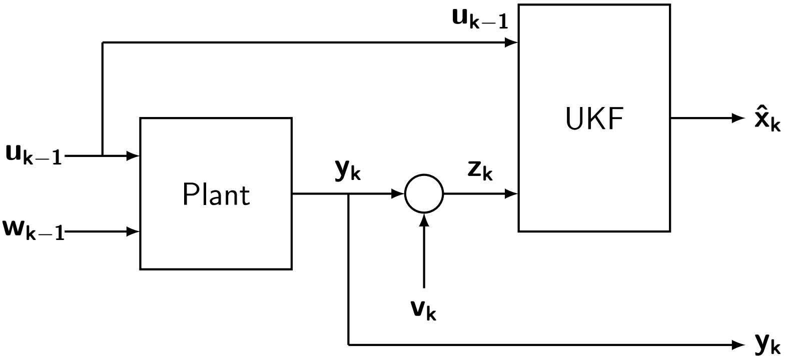
UKF and Plant Interaction Diagram



Note: In this scenario, since there is no input u, we assume that there is also no process noise w.

Sigma Point Transformation

- **Step 1: Unscented Transformation (UT)**
  Transform the corrected state and covariance to sigma points:

  $$\mathbf{x}_{\mathrm{cor}_{k-1}} \longrightarrow \mathbf{X}_{\mathrm{cor}_{k-1}}(\sigma \text{ points})$$

  We will spread $\mathbf{2N}+\mathbf{1}$ sigma points around the state estimate as follows:

  $$\mathbf{X}_{\mathrm{cor}_{k-1}} \begin{cases} = \mathbf{x}_{\mathrm{cor}_{k-1}} & , \mathbf{i} = 1 \\ = \mathbf{x}_{\mathrm{cor}_{k-1}} + \sqrt{(\mathbf{N}+\lambda)\mathbf{P}_{\mathbf{cor_{k-1}}}}, & \mathbf{i} = 2, ..., \mathbf{N}+1 \\ = \mathbf{x}_{\mathrm{cor}_{k-1}} - \sqrt{(\mathbf{N}+\lambda)\mathbf{P}_{\mathbf{cor_{k-1}}}}, & \mathbf{i} = \mathbf{N}+2, ..., \mathbf{2N}+1 \end{cases}$$

State Prediction

- **Step 2: Propagation through Process Model**
  Each sigma point is propagated using the nonlinear function:

$$\mathbf{X}_k = \mathbf{f_d}(\mathbf{X}_{\mathrm{cor}_{k-1}})$$

Predicted State Mean

- **Step 3: Predicted State Mean**
  The predicted state is calculated as the weighted sum of the propagated sigma points:

$$\mathbf{x}_{\mathrm{pre}_k} = \sum_{\mathbf{i}=1}^{2\mathbf{n}+1} \mathbf{w_m}^{(\mathbf{i})} \mathbf{X}_k^{(\mathbf{i})}$$

Covariance Prediction

- **Step 4: State Deviation**
  The deviation of each sigma point is computed as:

$$\Delta \mathbf{x}_k = \mathbf{X}_k - \mathbf{x}_{\text{pre}_k}$$

- **Step 5: Predicted Covariance**
  Covariance is calculated as:

$$\mathbf{P}_{\text{pre}_k} = \Delta \mathbf{x}_k \text{diag}(\mathbf{w_c}) \Delta \mathbf{x}_k^T + \mathbf{Q_k}$$

Measurement Update (Part 1)

- **Step 6: UT Transfer for Measurement**
  Transform the predicted state and covariance to sigma points:

$$\mathbf{x}_{\mathrm{pre}_k} \longrightarrow \mathbf{X}_{\mathrm{pre}_k} (\sigma \text{ points})$$

We will spread $\mathbf{2N} + \mathbf{1}$ sigma points around the state estimate as follows:

$$\mathbf{X}_{\mathrm{pre}_k} \begin{cases} = \mathbf{x}_{\mathrm{pre}_k}, \mathbf{i} = 1 \\ = \mathbf{x}_{\mathrm{pre}_k} + \sqrt{(\mathbf{N} + \lambda)\mathbf{P}_{\mathbf{pre_k}}}, \mathbf{i} = \mathbf{i} + 1 \\ = \mathbf{x}_{\mathrm{pre}_k} - \sqrt{(\mathbf{N} + \lambda)\mathbf{P}_{\mathbf{pre_k}}}, \mathbf{i} = \mathbf{i} + \mathbf{N} + 1 \end{cases}$$

Measurement Update (Part 2)

- **Step 7: Measurement Prediction**
  Nonlinear measurement model is applied to each sigma point:

$$\mathbf{Z}_k = \mathbf{h}(\mathbf{X}_{\text{pre}_k})$$

- **Step 8: Predicted Measurement**
  The predicted measurement is computed as:

$$\mathbf{z}_{\text{pre}_k} = \sum_{\mathbf{i}=1}^{2\mathbf{n}+1} \mathbf{w_m}^{(\mathbf{i})} \mathbf{Z}_k^{(\mathbf{i})}$$

Measurement Innovation and Covariance Updates I

- **Step 9: Measurement Innovation**
  The measurement innovation (or residual) is calculated as the difference between the actual measurement and the predicted measurement:

$$\Delta \mathbf{z}_k = \mathbf{Z}_k - \mathbf{z}_{\text{pre}_k}$$

- **Step 10: Predicted Measurement Covariance**
  The predicted measurement covariance is calculated using the sigma points:

$$\mathbf{P}_{z,\text{pre}_k} = \Delta \mathbf{z}_k \text{diag}(\mathbf{w_c}) \Delta \mathbf{z}_k^T + \mathbf{R}_k$$

Measurement Innovation and Covariance Updates II

- **Step 11: Cross-Covariance**
  The cross-covariance between the state and the measurement is computed as:

$$\mathbf{P}_{xz,\text{pre}_k} = \Delta\mathbf{x}_k \text{diag}(\mathbf{w_c}) \Delta\mathbf{z}_k^T$$

Kalman Gain and State Correction

- **Step 12: Kalman Gain**
  Kalman gain is calculated as:

$$\mathbf{K}_k = \frac{\mathbf{P}_{xz_{\text{pre}}k}}{\mathbf{P}_{z_{\text{pre}}k}}$$

- **Step 13: State Correction**
  The state is corrected based on the measurement residual:

$$\mathbf{x}_{\text{cor}_k} = \mathbf{x}_{\text{pre}_k} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{z}_{\text{pre}_k})$$

Covariance Correction

- **Step 14: Correct the Covariance**
  The covariance is updated as:

$$\mathbf{P}_{x\,\mathrm{cor}_k} = \mathbf{P}_{x\,\mathrm{pre}_k} - \mathbf{K}_k \mathbf{P}_{z_{\mathrm{pre}\,k}} \mathbf{K}_k^T$$

MATLAB Code: matrix_sqrt.m I

```
function sqrt_A = matrix_sqrt(A)
    % Step 1: Compute eigenvalues and eigenvectors
    [Q, D] = eig(A);

    % Step 2: Compute the square root of the eigenvalues
    D_sqrt = sqrt(D);

    % Step 3: Reconstruct the square root of the matrix
    sqrt_A = Q * D_sqrt * inv(Q);
end
```

MATLAB Code: plotUKFResults.m I

```matlab
function plotUKFResults(x_true, x_cor, z, P_cor, num_steps, delta_t)
    % Plot results of UKF
    % Convert time steps to actual time based on delta_t
    time = (0:num_steps-1) * delta_t;
    labels = {'X Position', 'Z Position', 'Velocity', 'Pitch Angle', 'Drag Coefficient'};

    % Plot state variables (true, estimated, observed)
    figure;
    for i = 1:5
        subplot(3, 2, i);
        plot(0:num_steps-1, x_true(i,:), '-', 'LineWidth', 2, 'Color', 'b', 'DisplayName', 'True State');
        hold on;
        plot(0:num_steps-1, x_cor(i,:), '--', 'LineWidth', 2, 'Color', 'r', 'DisplayName', 'Estimated State');
        hold on;
        if i <= 2
            plot(0:num_steps-1, z(i,:), '-', 'LineWidth', 2, 'Color', 'magenta', 'DisplayName', 'Observations');
        end
        xlabel('Time Step');
        ylabel(labels{i});
        legend;
        title([labels{i} ': True State vs Estimated State and Observations']);
    end
    % Plot trajectory
    subplot(3, 2, 6);
```

MATLAB Code: plotUKFResults.m II

```
    plot(x_true(1,:), x_true(2,:), '-', 'LineWidth', 2, 'Color', 'b', 'DisplayName', 'True State');
    hold on;
    plot(x_cor(1,:), x_cor(2,:), '--', 'LineWidth', 2, 'Color', 'r', 'DisplayName', 'Estimated State');
    hold on;
    plot(z(1,:), z(2,:), '-', 'LineWidth', 2, 'Color', 'magenta', 'DisplayName', 'Observations');
    xlabel('X Position');
    ylabel('Z Position');
    legend;
    title('Trajectory: True State vs Estimated State and Observations');
    % Plot covariance matrices for all state variables
    figure;
    for i = 1:5
        subplot(5, 1, i);
        plot(time(2:end), squeeze(P_cor(i, i, 2:end)), '-d', 'LineWidth', 2, 'Color', 'k', 'DisplayName', ['P_{cor}(' num2str(i) ',' num2str(i) ')']);
        xlabel('Time (s)');
        ylabel(['P_{est}(' num2str(i) ',' num2str(i) ')']);
        legend;
        title(['Estimated State Covariance for ' labels{i}]);
    end
end
```

MATLAB Code: QuadraticDragmodel.m I

```matlab
function f = Quadraticdragmodel(x,h)
    % Define constants
    g = 9.81;  % Gravitational acceleration
    % Define the continuous function
    f_continous = @(x) [
        x(3) * cos(x(4));                 % dx/dt = v * cos(theta)
        x(3) * sin(x(4));                 % dz/dt = v * sin(theta)
        -g * sin(x(4)) - g * x(5) * x(3)^2;   % dv/dt = -g * sin(theta) - k * v^2
        -g * cos(x(4)) / x(3);            % dtheta/dt = -g * cos(theta) / v
        0                                 % dk/dt = 0 (assumed constant for simplicity)
    ];

    % Solve using Runge-Kutta method
    k1 = f_continous(x);
    k2 = f_continous(x + 0.5 * h * k1);
    k3 = f_continous(x + 0.5 * h * k2);
    k4 = f_continous(x + h * k3);
    k = (k1 + 2 * k2 + 2 * k3 + k4) / 6;
    x_next = x + k * h;
    % Store the result in the output variable f
    f = x_next;
end
```

MATLAB Code: QuadraticDragProjectileMotion.m I

```
function [t,y] = QuadraticDragProjectileMotion(x0, z0, v0, theta0, k)
    % Constants
    global g h

    % Initial state vector
    initial_conditions = [x0; z0; v0; theta0];

    % Time span for the simulation (use a large end time)
    t_start = 0; % start time (s)
    t_end = 100; % end time (s)


    % Define the system of ODEs
    % y(1) = x position
    % y(2) = z position
    % y(3) = velocity
    % y(4) = pitch angle
    f = @(t, y) [
         y(3) * cos(y(4));            % dx/dt = v * cos(theta)
         y(3) * sin(y(4));            % dz/dt = v * sin(theta)
        -g * sin(y(4)) - g * k * y(3)^2;  % dv/dt = -g * sin(theta) - k * v^2
        -g * cos(y(4)) / y(3)         % dtheta/dt = -g * cos(theta) / v
    ];

    % Initialize time and state vectors
    t = t_start:h:t_end;
```

MATLAB Code: QuadraticDragProjectileMotion.m II

```matlab
    n = length(t);
    y = zeros(4, n);
    y(:, 1) = initial_conditions;

    % RK4 Method
    for i = 1:n-1
        y(:, i+1) = RK4(f, t(i), y(:,i), h);

        % Stop if the projectile hits the ground
        if y(2,i+1) < 0
            % Truncate the arrays to the point where the projectile hits the ground
            t = t(1:i+1);
            y = y(:, 1:i+1);
            break;
        end
    end
end
```

MATLAB Code: RK4.m I

```
function y_next = RK4(f, t, y, h)
    k1 = f(t, y);
    k2 = f(t + 0.5 * h, y + 0.5 * h * k1);
    k3 = f(t + 0.5 * h, y + 0.5 * h * k2);
    k4 = f(t + h, y + h * k3);
    k = (k1 + 2 * k2 + 2 * k3 + k4) /6;
    y_next = y + k*h;
end
```

MATLAB Code: UKF.m I

```
function [x_cor_new, P_x_cor_new, x_pre, P_x_pre] = UKF(x_cor, P_x_cor, Q, R,
    N, kappa, alpha, beta, f, h, z)
    [w_m, w_c, lambda] = weight(N, kappa, alpha, beta);
    % Generate sigma points
    X_cor = UT(x_cor, P_x_cor, N, lambda);
    % Prediction step
    for i = 1:2 * N + 1
        X(:, i) = f(X_cor(:, i));
    end
    x_pre = X * w_m';
    diff_x = X - x_pre;
    P_x_pre = diff_x * diag(w_c) * diff_x' + Q;
    % Generate new sigma points
    X_pre = UT(x_pre, P_x_pre, N, lambda);
    for i = 1:2 * N + 1
        Z(:, i) = h(X_pre(:, i));
    end
    % Update step
    z_pre = Z * w_m';
    diff_z = Z - z_pre;
    P_z_pre = diff_z * diag(w_c) * diff_z' + R;
    P_xz_pre = diff_x * diag(w_c) * diff_z';
    K = P_xz_pre / P_z_pre;
    x_cor_new = x_pre + K * (z - z_pre);
    P_x_cor_new = P_x_pre - K * P_z_pre * K';
end
```

MATLAB Code: UT.m I

```
function X = UT(x, P, N, lambda)
    X = zeros(N, 2 * N + 1);
    X(:, 1) = x;
    %sqrt_P = sqrtm((N + lambda) * P);
    sqrt_P = matrix_sqrt((N + lambda) * P);
    for i = 1:N
        X(:, i + 1) = x + sqrt_P(:, i);
        X(:, i + N + 1) = x - sqrt_P(:, i);
    end
end
```

MATLAB Code: weight.m I

```
function [w_m, w_c, lambda] = weight(N, kappa, alpha, beta)
    lambda = alpha^2 * (N + kappa) - N;
    w_m = zeros(1, 2 * N + 1);
    w_c = zeros(1, 2 * N + 1);
    w_m(1) = lambda / (N + lambda);
    w_c(1) = lambda / (N + lambda) + (1 - alpha^2 + beta);
    w_i = 1 / (2 * (N + lambda));
    w_m(2:end) = w_i;
    w_c(2:end) = w_i;
end
```