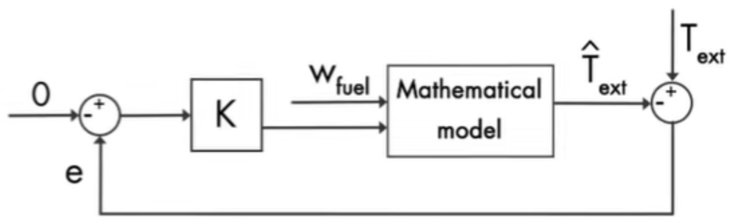
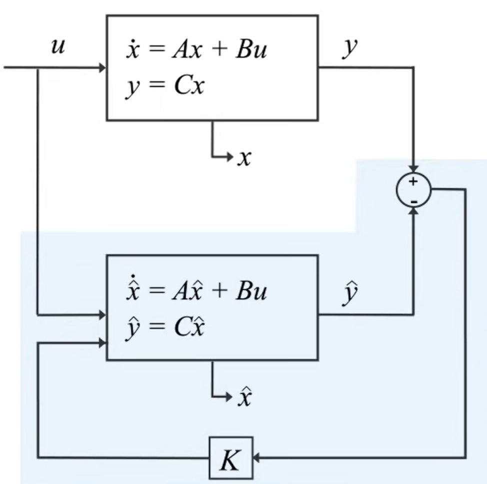
Kalman Filters

- Generally used to obtain the optimal estimate measurement after fusion of sensors when some sensors give uncertain or no value
- Algorithm generally used in applications like determining location speed etc
- State estimator- used to find quantities that are difficult or unable to find using mathematical model
- The methodology used in kalman filter is similar to feedback control mechanism where the difference between estimated and measured is minimised by using a controller K



$e = \text{error} = T_{ext} - \hat{T}_{ext}$



STATE OBSERVER

$e_{obs} = x - \hat{x}$

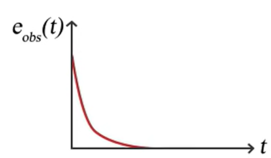
$\dot{x} = Ax + Bu \qquad y = Cx$

$\dot{\hat{x}} = A\hat{x} + Bu + K(y - \hat{y}) \qquad \hat{y} = C\hat{x}$

$\dot{e}_{obs} = (A - KC)e_{obs} \qquad y - \hat{y} = Ce_{obs}$

$\hookrightarrow e_{obs}(t) = e^{(A-KC)t}e_{obs}(0)$

If $(A\text{-}KC) < 0$, then $e_{obs} \to 0$ as $t \to \infty$. So, $\hat{x} \to x$.

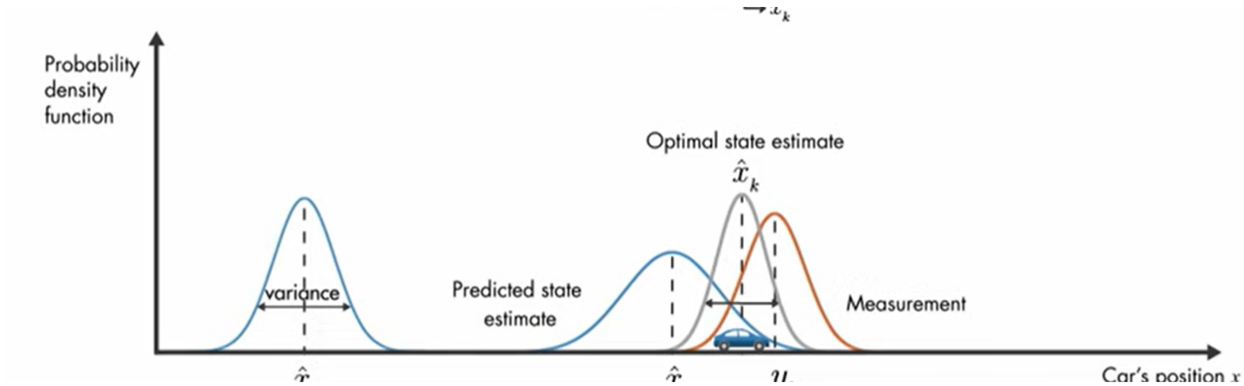


State observer $\quad \hat{x}_{k+1} = A\hat{x}_k + Bu_k + K(y_k - C\hat{x}_k)$

Kalman filter $\quad \hat{x}_k = A\hat{x}_{k-1} + Bu_k + K_k(y_k - C(A\hat{x}_{k-1} + Bu_k))$

Kalman algorithm generally follows steps as follows

- Prediction- expected measurement



Optimal state function(gaussian function) is obtained by multiplying the predicted and measured PDF and the mean of this gives the optimal estimate

- Variance and standard deviation

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2$$

Kalman filter

$$\hat{x}_k = \underbrace{\hat{x}_k^-}_{\text{Predict}} + \underbrace{K_k(y_k - C\hat{x}_k^-)}_{\text{Update}}$$

Posterori estimate

Prediction

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$$

$$P_k^- = AP_{k-1}A^T + Q$$

Initial estimates for $\hat{x}_{k-1}$ and P_{k-1}

Update

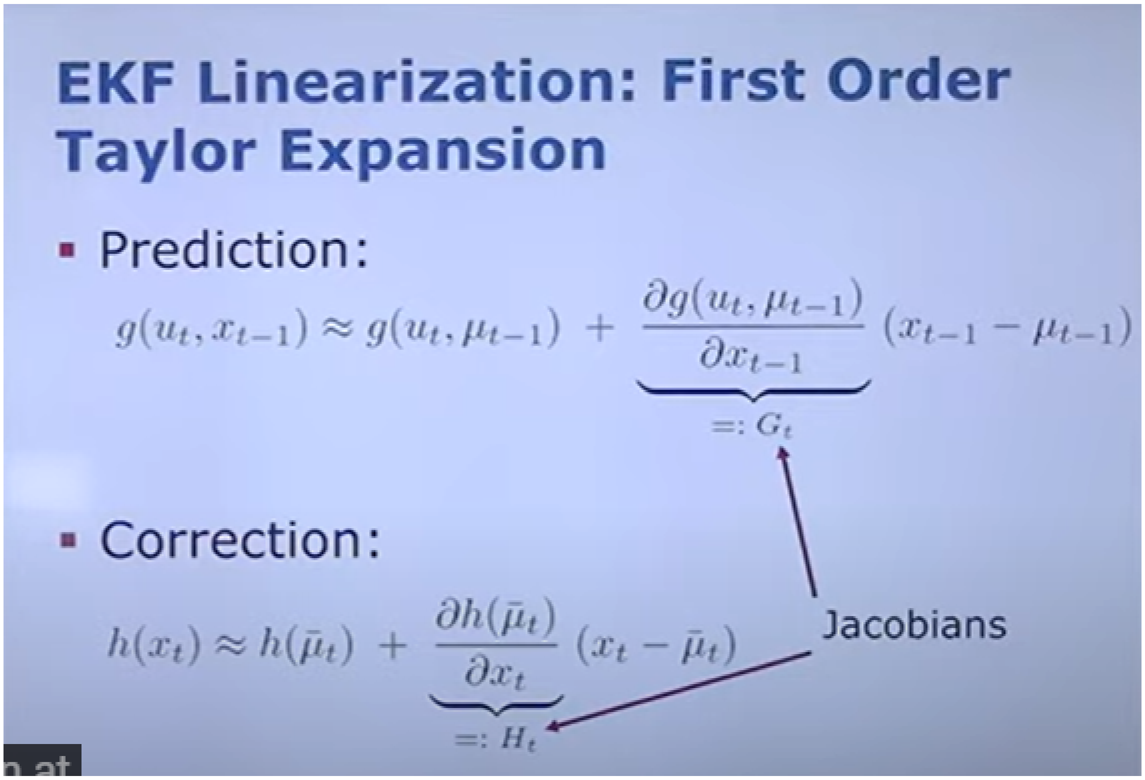
$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R}$$

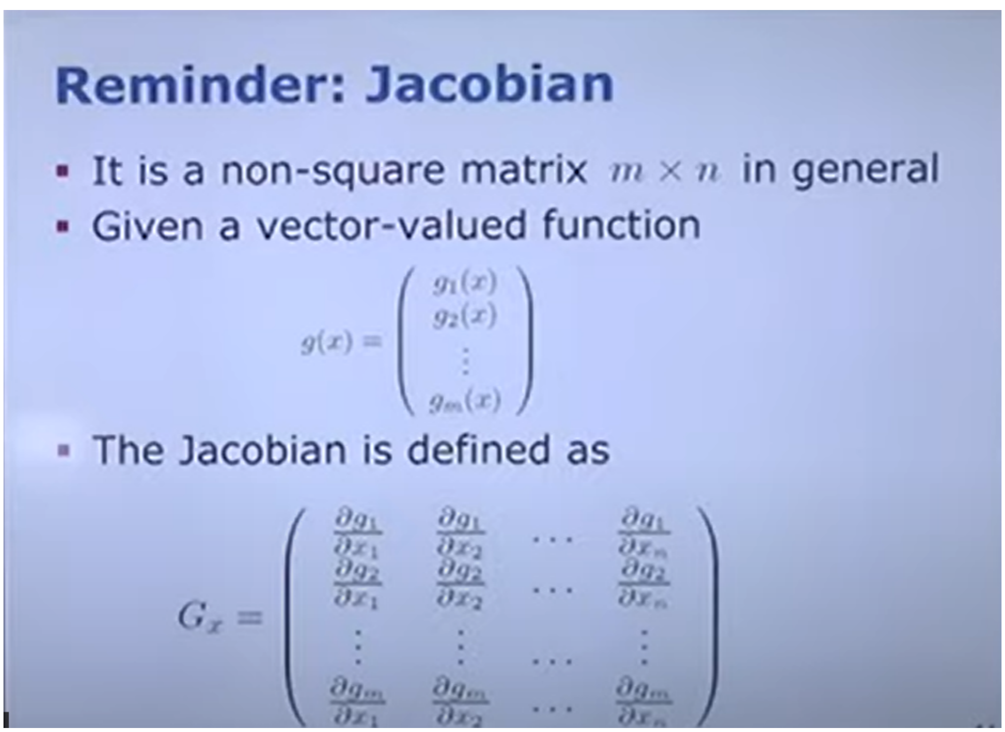
$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-)$$

$$P_k = (I - K_k C)P_k^-$$

- Distribution in graph

- Precision


EKF Linearization: First Order Taylor Expansion
Prediction:
$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + \underbrace{\frac{\partial g(u_t, \mu_{t-1})}{\partial x_{t-1}}}_{=: G_t} (x_{t-1} - \mu_{t-1})$
Correction:
$h(x_t) \approx h(\bar{\mu}_t) + \underbrace{\frac{\partial h(\bar{\mu}_t)}{\partial x_t}}_{=: H_t} (x_t - \bar{\mu}_t)$
Jacobians


Reminder: Jacobian
It is a non-square matrix $m \times n$ in general
Given a vector-valued function
$g(x) = \begin{pmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_m(x) \end{pmatrix}$
The Jacobian is defined as
$G_x = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial g_m}{\partial x_1} & \frac{\partial g_m}{\partial x_2} & \cdots & \frac{\partial g_m}{\partial x_n} \end{pmatrix}$

TEST CODE FOR MULTIPLE OBJECT INTERACTION

```
#!/usr/bin/env python

import rospy
import numpy as np
from sensor_msgs.msg import LaserScan
from std_msgs.msg import Float32
from filterpy.kalman import ExtendedKalmanFilter
from scipy.optimize import linear_sum_assignment

class MultiObjectTracker:
    def __init__(self):
        self.ekfs = []
        self.track_ids = []

    def create_new_ekf(self, measurement):
        ekf = ExtendedKalmanFilter(dim_x=4, dim_z=2)
        ekf.x = np.array([measurement[0], 0, measurement[1], 0])  # Initial state [x, x_velocity, y,
y_velocity]
        ekf.F = np.array([[1, 1, 0, 0], [0, 1, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1]])  # State transition matrix
        ekf.H = np.array([[1, 0, 0, 0], [0, 0, 1, 0]])  # Measurement function
        ekf.P *= 1.0  # Initial covariance matrix
        ekf.R = np.array([[0.1, 0], [0, 0.1]])  # Measurement noise
        ekf.Q = np.array([[0.01, 0, 0, 0], [0, 0.01, 0, 0], [0, 0, 0.01, 0], [0, 0, 0, 0.01]])  # Process
noise

        return ekf

    def update_ekf(self, ekf, measurement):
        ekf.predict()
        ekf.update(measurement)

    def associate_measurements(self, measurements, tracked_objects):
        cost_matrix = np.zeros((len(tracked_objects), len(measurements)))
```

```
        for i, tracked_obj in enumerate(tracked_objects):
            for j, measurement in enumerate(measurements):
                diff = tracked_obj[0:2] - measurement
                cost_matrix[i, j] = np.linalg.norm(diff)

        row_indices, col_indices = linear_sum_assignment(cost_matrix)
        associations = zip(row_indices, col_indices)

        return associations

    def update_tracks(self, measurements):
        new_tracks = []

        if not self.ekfs:
            for measurement in measurements:
                ekf = self.create_new_ekf(measurement)
                self.ekfs.append(ekf)
                self.track_ids.append(rospy.Time.now().to_sec())
        else:
            tracked_objects = [ekf.x for ekf in self.ekfs]
            associations = self.associate_measurements(measurements, tracked_objects)

            for tracked_index, measurement_index in associations:
                ekf = self.ekfs[tracked_index]
                measurement = measurements[measurement_index]
                self.update_ekf(ekf, measurement)
                new_tracks.append(ekf.x)
                self.track_ids[tracked_index] = rospy.Time.now().to_sec()

        return new_tracks

class SensorFusionNode:
    def __init__(self):
        rospy.init_node('multi_object_fusion_node')

        self.lidar_sub = rospy.Subscriber('/lidar_topic', LaserScan, self.lidar_callback)
        self.ultrasonic_sub = rospy.Subscriber('/ultrasonic_topic', Float32, self.ultrasonic_callback)

        self.fused_pub = rospy.Publisher('/fused_objects', Float32, queue_size=10)

        self.tracker = MultiObjectTracker()

    def lidar_callback(self, data):
```

```
        # Process LIDAR data and obtain measurements for each detected object
        measurements = []
        # Append measurements in the format (x_position, y_position)
        self.update_and_publish_fusion(measurements)

    def ultrasonic_callback(self, data):
        # Process ultrasonic data and obtain measurements for each detected object
        measurements = []
        # Append measurements in the format (x_position, y_position)
        self.update_and_publish_fusion(measurements)

    def update_and_publish_fusion(self, measurements):
        new_tracks = self.tracker.update_tracks(measurements)

        for track in new_tracks:
            rospy.loginfo("Object Position: (%.2f, %.2f)" % (track[0], track[2]))

if __name__ == '__main__':
    try:
        node = SensorFusionNode()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```