Project : Extended Kalman Filters

Introduction:

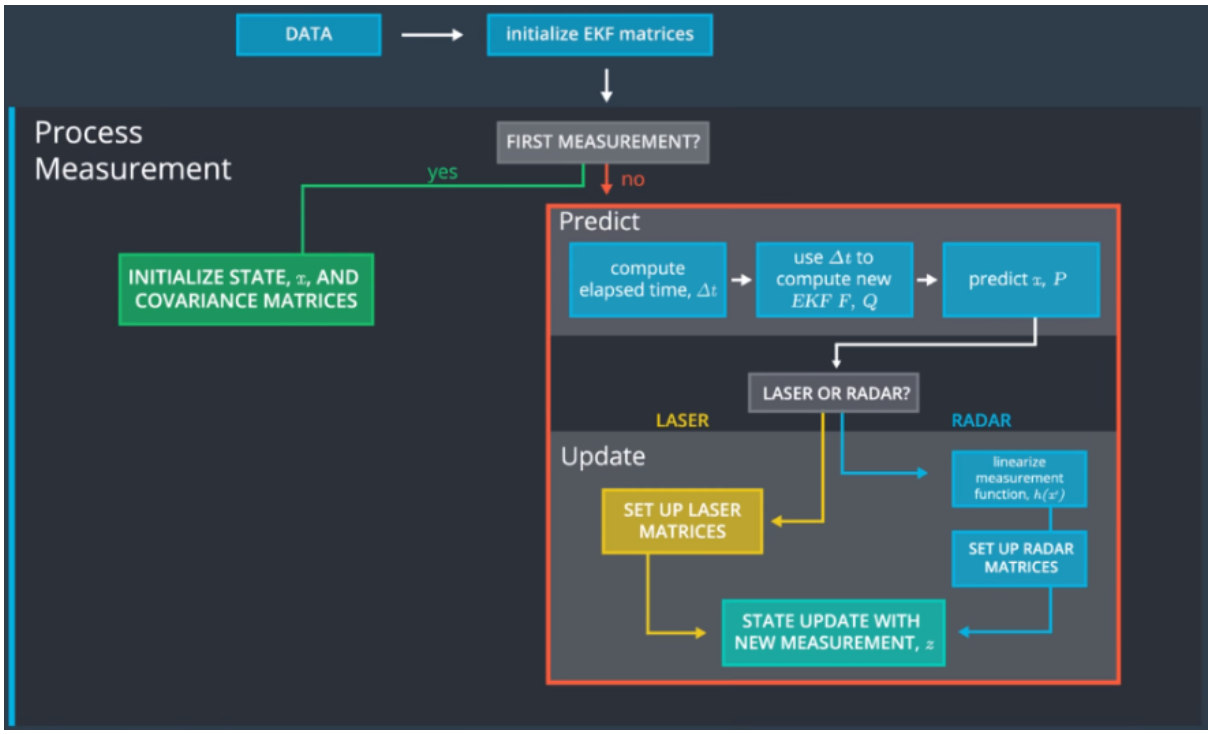
In estimation theory, the **extended Kalman filter (EKF)** is the nonlinear version of the Kalman filter which linearizes about an estimate of the current mean and covariance. In the case of well defined transition models, the EKF has been considered[1] the *de facto* standard in the theory of nonlinear state estimation, navigation systems and GPS.

In statistics and control theory, **Kalman filtering**, also known as **linear quadratic estimation** (**LQE**), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. The filter is named after Rudolf E. Kálmán, one of the primary developers of its theory.

The Kalman filter has numerous applications in technology. A common application is for guidance, navigation, and control of vehicles, particularly aircraft and spacecraft. Furthermore, the Kalman filter is a widely applied concept in time series analysis used in fields such as signal processing and econometrics. Kalman filters also are one of the main topics in the field of robotic motion planning and control, and they are sometimes included in trajectory optimization.

Working:

The algorithm consists of the steps **First Measurement, Initialize state and covariance matrices, Predict** and **Update(LiDAR) / UpdateEKF(Radar).**



The calculations made for Extended Kalman Filters are shown below:

- **first measurement** - the filter will receive initial measurements of the bicycle's position (i.e. targeted object) relative to the car. These measurements will come from a radar or lidar sensor.

- **initialize state and covariance matrices** - the filter will initialize the bicycle's position based on the first measurement.

  Then the car will receive another sensor measurement after a time period \Delta{t}Δ*t*.

- **predict** - the algorithm will predict where the bicycle will be after time \Delta{t}Δ*t*. One basic way to predict the bicycle location after \Delta{t}Δ*t* is to assume the bicycle's velocity is constant; thus the bicycle will have moved velocity * \Delta{t}Δ*t*. In the extended Kalman filter lesson, we will assume the velocity is constant.

- **update** - the filter compares the "predicted" location with what the sensor measurement says. The predicted location and the measured location are combined to give an updated location. The Kalman filter will put more weight on either the predicted location or the measured location depending on the uncertainty of each value.
- then the car will receive another sensor measurement after a time period \Delta{t}Δ*t*. The algorithm then does another **predict** and **update** step.

LIDAR:

The measurement values of LIDAR are in the form of (x-co-ordinate, y-co-ordinate, velocity in x direction, velocity in y directon)

Prediction:

$$x' = Fx + u$$
$$P' = FPF^T + Q$$

Measurement Update:

$$y = z - Hx'$$
$$S = HP'H^T + R$$
$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$
$$P = (I - KH)P'$$

Radar:

The measurement values of RADAR are in the form of (range, angle made by the range vector i.e. phi, velocity vector.)

Prediction:

$$x' = f(x, u)$$
$$P' = F_j P F_j^T + Q$$

Measurement Update:

$$y = z - h(x')$$

After y is computed the same steps are followed to find S, K and update of x, P.

The significance for every matrix of the above steps can be found at Wikipedia.

Code:

Files Used

1) KalmanFilters.cpp :
   - Void Predict(measurements)
   - Void Update(measurements) //LIDAR
   - Void UpdateEKF(measurements) //RADAR
   - Void UpdateWithY(Eigen::VectorXd y) //Calculation of S, K and update x, P

2)FusionEKF.cpp :

   - First Measurement
   - Void ProcessMeasurement(measurements)
     Initialization of state and covariance matrices
     Call predict and either UpdateEKF() or Update() as per sensor measurement received.

3)Tools.cpp :

   - Void RMSE(ground truth, estimations) //To calculate RootMeanSquareError
   - Void CalculateJacobian(CurrentStateX) //To calculate Jacobian Matrix for a given stateX used for UpdateEKF()

Output:

The results obtained by this algorithm are displayed below

The algorithm is successful in keeping the RMSE values in the desired error range.