EXTENDED KALMAN FILTER

PROJECT 5

TERM 1

SELF DRIVING CAR ENGINEER NANODEGREE

The Extended Kalman Filter (EKF) project is the 5th and final project of term 1 of the Nanodegree. To pass the project students are required to implement EKF in C++ for sensor fusion. The sensors used are Radar and Lidar. Applying the EKF algorithm we are supposed to predict the position of the car in the provided simulator. Sensor data and ground truth data are provided which is read by main.cpp.
The files given in the workspace are:

- `main.cpp` - communicates with the Term 2 Simulator receiving data measurements, calls a function to run the Kalman filter, calls a function to calculate RMSE
- `FusionEKF.cpp` - initializes the filter, calls the predict function, calls the update function
- `kalman_filter.cpp`- defines the predict function, the update function for LIDAR, and the update function for radar
- `tools.cpp`- function to calculate RMSE and the Jacobian matrix

The workflow of EKF is given in figure 1.

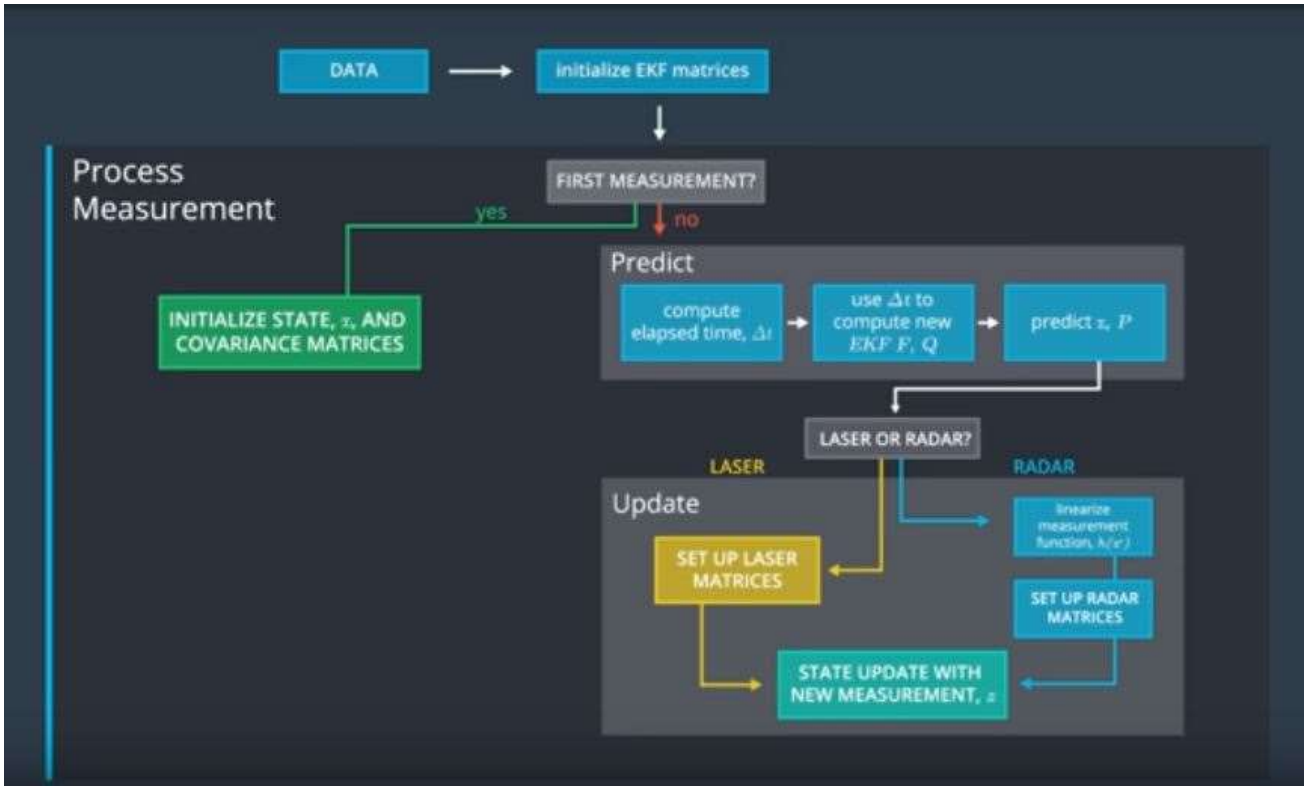

Figure1: Extended Kalman filter block diagram

Reading Data

Main.cpp reads the sensor data into a variable.
For laser inputs are x and y positions.

meas_package.raw_measurements_ << px, py;

In case of Radar inputs are rho, theta and rho_dot.

meas_package.raw_measurements_ << ro, theta, ro_dot;

Main.cpp also calls functions to run Kalman filter and to calculate RMSE value.

Files to Edit:

To complete the project, we need to modify only 3 files:

- **FusionEKF.cpp,**
- **kalman_filter.cpp, and**
- **tools.cpp.**

Tools.cpp

Tools.cpp is the easiest and direct to modify, hence I modified this file first. It has 2 functionalities, which are to calculate the RMSE and to calculate the Jacobian.
The RMSE function takes in the estimates and ground truth to find the RMSE using the following code (figure 2):

```
VectorXd rmse(4);
rmse << 0,0,0,0;

if (estimations.size() != ground_truth.size() || estimations.size() == 0) {
  std::cout << "Invalid estimation or ground_truth data" << endl;
  return rmse;
}

for (unsigned int i=0; i < estimations.size(); ++i) {

  VectorXd residual = estimations[i] - ground_truth[i];
  residual = residual.array()*residual.array();
  rmse += residual;
}
rmse = rmse/estimations.size();
rmse = rmse.array().sqrt();
return rmse:
```

Figure 2

The Jacobian function is used to linearize the input function of radar. In the case of Radar, the measurement projections function h'(x) that projects the predicted state to measurement state is non-linear. Hence we linearize it using Tylor series expansion, which in turn gives the Jacobian Matrix.

```
double c1 = px*px+py*py;
double c2 = sqrt(c1);
double c3 = (c1*c2);

// check division by zero
if (fabs(c1) < 0.0001) {
  cout << "CalculateJacobian () - Error - Division by Zero" << endl;
  return Hj;
}
else{
  // compute the Jacobian matrix
  Hj << (px/c2), (py/c2), 0, 0,
        -(py/c1), (px/c1), 0, 0,
        py*(vx*py - vy*px)/c3, px*(px*vy - py*vx)/c3, px/c2, py/c2;

return Hj;
```

Figure 3

The calculation of Jacobian Matrix is in shown in the figure 3. px, py, vx,vy are position and velocity information derived from Radar measurements.

kalman_filter.cpp

The **Kalman_Filter** class is defined in **kalman_filter.h** and **kalman_filter.cpp**. We only require editing the kalman_filter.cpp file, which contains functions for prediction and updating.

KalmanFilter::Predict()

This function executes the prediction step of the Kalman filter. The steps are the same for both Laser and Radar inputs which are as shown in figure 4. The values in x variable are different for Laser and Radar. The variables are initialized in **FusionEKF.cpp**

```
x_ = F_ * x_;
MatrixXd Ft = F_.transpose();
P_ = F_ * P_ * Ft + Q_;
```

Figure 4

KalmanFilter::Update()

This function is called when the sensor input is from Laser. It is the execution of Kalman filter equations as shown in figure 5. Except for the error equation which finds the value of y, all the other equations are the same for EKF update. But the values of variables vary according to the sensor.

```
VectorXd z_pred = H_ * x_;
VectorXd y = z - z_pred;
MatrixXd Ht = H_.transpose();
MatrixXd S = H_ * P_ * Ht + R_;
MatrixXd Si = S.inverse();
MatrixXd PHt = P_ * Ht;
MatrixXd K = PHt * Si;

//new estimate
x_ = x_ + (K * y);
long x_size = x_.size();
MatrixXd I = MatrixXd::Identity(x_size, x_size);
P_ = (I - K * H_) * P_;
```

Figure 5

KalmanFilter::UpdateEKF()

This function is called when the input is from the Radar. Here the error equation is given by **VectorXd y = z – h,** where the h is given by the equations in figure 6. Variable px, py, vx and vy are positions and velocity information derived from Radar inputs. When we calculate y, we need to ensure that the value of phi is between **–pi** and **+pi.**

```
double rho = sqrt(px*px + py*py);
double theta = atan2(py, px);
double rho_dot = (px*vx + py*vy) / rho;
VectorXd h = VectorXd(3);
h << rho, theta, rho_dot;
```

Figure 6

The rest of the equations are same as the equations of Kalman filter, except for the fact that the value of variables H and R are different. H matrix is the Jacobian matrix which is discussed in the following section

FusionEKF.cpp

FusionEKF.cpp has mainly 3 functionalities, which are to initialize the filter, calls predict and update functions.

Initialize the filter

We need to initialize the variables for KalmanFilter class defined in kalman_filter.h and kalman_filter.cpp. The KalmanFilter class requires initialization of the following:

Vector x: state
Matrix P: state covariance matrix
Matrix F: State transition matrix
Matrix H: Measurement matrix
Matrix R: measurement covariance matrix
Matrix Q: Process covariance matrix

The variable called **ekf_,** which is an instance of KalmanFilter class, will hold the values of the above variables (eg: ekf_.P for P matrix).

First, the variables R_laser, R_radar, H_laser, and P (as ekf_.P, since it is common for both and can be passed directly) are initialized. Before the **Predict () function** we check for the type of sensor. If the sensor type is RADAR, from measurement_pack.raw_measurements_ (0, 1, 2 positions) we receive rho, phi and rhodot values, which are in turn converted to Cartesian coordinates using equations shown in figure 6. The position and velocity values are passed to kalman_filter function as ekf_.x.

```
double px = rho * cos(phi);
double py = rho * sin(phi);
double vx = rhodot * cos(phi);
double vy = rhodot * sin(phi);
// TODO: Convert radar from pola
//         and initialize state.
ekf_.x_ << px, py, vx , vy;
```

Figure 7

If the sensor is LASER, from measurement_pack.raw_measurements_ (0, 1 positions) we receive px, py values and vx, vy are initialized as 0. These values are passed to x variable in this case. The F Matrix and Q matrix are common for both the sensors and are initialised after assigning the value of **dt** variable. These variables are used in the Prediction step of Kalman filter.

Now, we need to initialize the variables for updating step of Kalman filter**.** If the sensor is RADAR, we update the value of H matrix as Jacobian (calculated using **CalculateJacobian()** function in tools.cpp) of the input vector (h vector) for linearizing the non-linear functions in the vector. Then we pass H and R matrices to the **UpdateEKF() function** of kalman_filter class. If the sensor is LASER, the specified H and R matrices are passed to **Update () function** of kalman_filter class.

Result and Discussion

The workspace has all the dependencies pre-installed. Navigate to the src folder in the workspace using cd command in terminal and execute the following commands to make and run the program.

mkdir build && cd build

cmake .. && make

./ExtendedKF

The built in simulator shows a real time graphical display of values sensor outputs and output from Kalman filter. RMSE values are also displayed in the window as shown in figure 8.

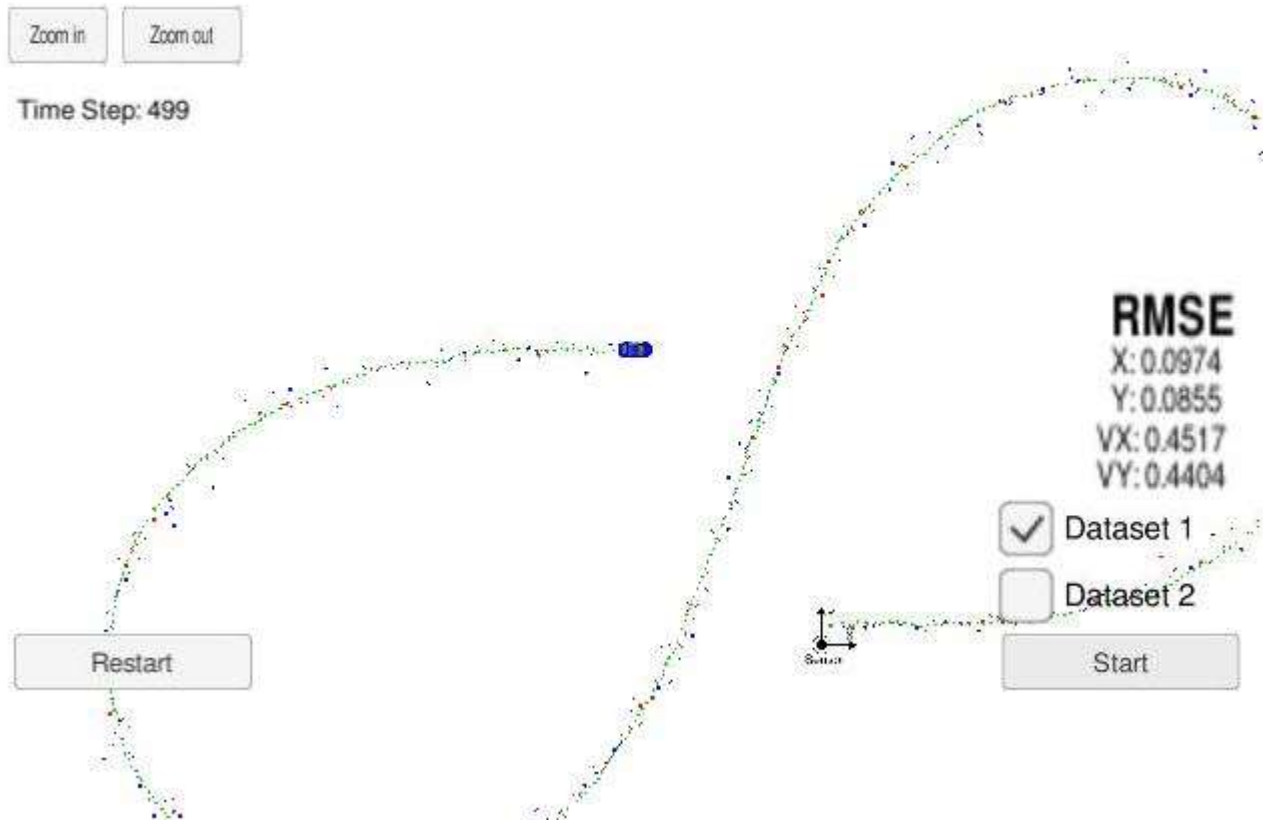

Figure 8: Simulator Output

The RMSE values are 0.0974, 0.0855 for position predictions in x and y directions and 0.4517 and 0.4404 for velocity predictions in x and y directions.

THANK YOU