Extended-Kalman-Filter

Write up

1st submit: August 22th, Kenta Kumazaki

1. Purpose

In this project you utilized a Kalman Filter to estimate the state of a moving object of interest with noisy lidar and radar measurements. Passing the project requires obtaining RMSE values that are lower than the tolerance outlined in the project rubric as below.

RMSE of px, py, vx, and vy should be less than or equal to the values [0.11, 0.11, 0.52, 0.52]

2. Goals/Steps

The goals / steps of this project are the following:

- Complete the codes in FusionEKF.cpp, kalman_filer.cpp, tools.cpp.
- Run Term2 Simulator and validate RMSE achieves target performance.
- Summarize the results with a written report

3. Submission

(1) GitHub

https://github.com/kkumazaki/Self-Drivig-Car_Project5_Extended-Kalman-Filter

(2) Directory

- **Writeup_of_Lesson24.pdf**: This file
- **src**
  - **main.cpp**: (didn't modify this file)
    - Communicates with Term2 Simulator receiving data measurements.
    - Calls the function to run Kalman Filter.
    - Calls the function to calculate RMSE.
  - **FusionEKF.cpp**:
    - Initialize the filter.
    - Calls the prediction function.
    - Calls the update function.
  - **kalman_filter.cpp**:
    - Define the prediction function.
    - Define the update function for LiDAR and Radar.
  - **tools.cpp**:
    - Calculate RMSE and Jacobian Matrix.

4. Coding

(1) FusionEKF.cpp

In Constructor, I initialized variables (previous_timestamp_) and Matrices (R, H, F, P, Q, etc).

```
#include "FusionEKF.h"
#include <iostream>
#include "Eigen/Dense"

using Eigen::MatrixXd;
using Eigen::VectorXd;
using std::cout;
using std::endl;
using std::vector;

/**
 * Constructor.
 */
FusionEKF::FusionEKF() {
//1. Initialize variables and matrices.
  is_initialized_ = false;

  previous_timestamp_ = 0;

  // initializing matrices
  R_laser_ = MatrixXd(2, 2);
  R_radar_ = MatrixXd(3, 3);
  H_laser_ = MatrixXd(2, 4);
  Hj_ = MatrixXd(3, 4);
  ekf_.F_ = MatrixXd(4, 4);
  ekf_.P_ = MatrixXd(4, 4);
  ekf_.Q_ = MatrixXd(4, 4);

  //measurement covariance matrix - laser
  R_laser_ << 0.0225, 0,
              0, 0.0225;

  //measurement covariance matrix - radar
  R_radar_ << 0.09, 0, 0,
              0, 0.0009, 0,
              0, 0, 0.09;

  //measurement matrix - laser
  H_laser_ << 1.0, 0.0, 0.0, 0.0,
              0.0, 1.0, 0.0, 0.0;

  //Jacobian matrix - radar (set 0 here)
  Hj_ << 0.0, 0.0, 0.0, 0.0,
         0.0, 0.0, 0.0, 0.0,
         0.0, 0.0, 0.0, 0.0;

```

```
  //state transition matrix (set delta_t = 0.5sec here)
  ekf_.F_ << 1.0, 0.0, 0.5, 0.0,
             0.0, 1.0, 0.0, 0.5,
             0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 1.0;

  //process noises (set large numbers for initialization)
  ekf_.P_ << 1000.0, 0.0, 0.0, 0.0,
             0.0, 1000.0, 0.0, 0.0,
             0.0, 0.0, 1000.0, 0.0,
             0.0, 0.0, 0.0, 1000.0;

  //process covariance matrix (set 0 here)
  ekf_.Q_ << 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0;

  //measurement noises
  // Any more??
}

/**
 * Destructor.
 */
FusionEKF::~FusionEKF() {}
```

In the function ProcessMeasurement(), at first I initialize Kalman Filter position vector with 1st sensor measurements.

```
void FusionEKF::ProcessMeasurement(const MeasurementPackage &measurement_pack) {
//2. Initialize Kalman Filter position vector with 1st sensor measurements.
  /**
   * Initialization
   */
  if (!is_initialized_) {
    /**
     * TODO: Initialize the state ekf_.x_ with the first measurement.
     * TODO: Create the covariance matrix.
     * You'll need to convert radar from polar to cartesian coordinates.
     */

    cout << "EKF: " << endl;
    ekf_.x_ = VectorXd(4);
    ekf_.x_ << 1, 1, 1, 1;

    if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
      // Initialize the state ekf_.x_ with the first measurement.
      // Convert radar from polar to cartesian coordinates

      float rho = measurement_pack.raw_measurements_(0);
      float phi = measurement_pack.raw_measurements_(1);
      float rhodot = measurement_pack.raw_measurements_(2);

      float px = rho * cos(phi);
      float py = rho * sin(phi);
      float vx = rhodot * cos(phi);
      float vy = rhodot * sin(phi);

      ekf_.x_ << px, py, vx, vy;

    }
    else if (measurement_pack.sensor_type_ == MeasurementPackage::LASER) {
      // Initialize the state ekf_.x_ with the first measurement.
      float px = measurement_pack.raw_measurements_(0);
      float py = measurement_pack.raw_measurements_(1);

      ekf_.x_ << px, py, 0.0, 0.0;

    }

    // update timestamp
    previous_timestamp_ = measurement_pack.timestamp_;

    // done initializing, no need to predict or update
    is_initialized_ = true;
    return;
  }
```

Initialization (Radar)

Initialization (Laser)

Next, I modify F and Q prior to prediction step based on the elapsed time delta_t.

```cpp
//3. Modify F and Q prior to prediction step based on the elapsed time delta_t.
  /**
   * TODO: Update the state transition matrix F according to the new elapsed time.
   * Time is measured in seconds.
   * TODO: Update the process noise covariance matrix.
   * Use noise_ax = 9 and noise_ay = 9 for your Q matrix.
   */

  float delta_t = (measurement_pack.timestamp_ - previous_timestamp_) / 1000000.0; // micro second --> second

  float noise_ax = 9.;
  float noise_ay = 9.;

  float delta_t2 = delta_t * delta_t;
  float delta_t3 = delta_t2 * delta_t;
  float delta_t4 = delta_t3 * delta_t;

  ekf_.F_ << 1.0, 0.0, delta_t, 0.0,
             0.0, 1.0, 0.0, delta_t,
             0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 1.0;

  ekf_.Q_ << delta_t4/4.0*noise_ax, 0.0, delta_t3/2.0*noise_ax, 0.0,
        0.0, delta_t4/4.0*noise_ay, 0.0, delta_t3/2.0*noise_ay,
        delta_t3/2.0*noise_ax, 0.0, delta_t2*noise_ax, 0.0,
        0.0, delta_t3/2.0*noise_ay, 0.0, delta_t2*noise_ay;

  // update timestamp
  previous_timestamp_ = measurement_pack.timestamp_;
```

Finally, I call update step for either Radar or Laser sensor measurements.

To avoid division by zero when I calculate Jacobian, **I added assertion (A)**.

```cpp
  //4. Call update step for either Radar and Laser sensor measurements.
  //   There are different functions for updating Radar and Laser.

    ekf_.Predict();

    /**
     * Update
     */

    /**
     * TODO:
     * - Use the sensor type to perform the update step.
     * - Update the state and covariance matrices.
     */

    if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
      // TODO: Radar updates
      float px = ekf_.x_[0];
      float py = ekf_.x_[1];

      float pxpy = px*px + py*py;

      // check the validity that px, py should not be zero
      //assert(pxpy > 0.0001);
      if (pxpy <= 0.0001){
        cout << "check the validity that px, py should not be zero ";
        assert(pxpy > 0.0001);
      }

      Hj_ = tools.CalculateJacobian(ekf_.x_);
      ekf_.H_ = Hj_;
      ekf_.R_ = R_radar_;
      ekf_.UpdateEKF(measurement_pack.raw_measurements_);

    } else {
      // TODO: Laser updates
      ekf_.H_ = H_laser_;
      ekf_.R_ = R_laser_;
      ekf_.Update(measurement_pack.raw_measurements_);
    }

    // print the output
    cout << "x_ = " << ekf_.x_ << endl;
    cout << "P_ = " << ekf_.P_ << endl;
  }
```

Prediction
(Same for Radar and Laser)

A

Measurement
(Radar)

Measurement
(Laser)

Output
(Same for Radar and Laser)

(2) kalman_filter.cpp

This file starts with Initialization of x vector and Matrices (P, F, H, R, Q).

Then, prediction function is described for both Radar and Laser.

```cpp
#include "kalman_filter.h"
#include "tools.h"
#include <bits/stdc++.h>

using Eigen::MatrixXd;
using Eigen::VectorXd;

using std::cout;

/*
 * Please note that the Eigen library does not initialize
 *   VectorXd or MatrixXd objects with zeros upon creation.
 */

KalmanFilter::KalmanFilter() {}

KalmanFilter::~KalmanFilter() {}

void KalmanFilter::Init(VectorXd &x_in, MatrixXd &P_in, MatrixXd &F_in,
                        MatrixXd &H_in, MatrixXd &R_in, MatrixXd &Q_in) {
  x_ = x_in;
  P_ = P_in;
  F_ = F_in;
  H_ = H_in;
  R_ = R_in;
  Q_ = Q_in;
}

void KalmanFilter::Predict() {
  x_ = F_ * x_;
  MatrixXd Ft = F_.transpose();
  P_ = F_ * P_ * Ft + Q_;
}
```

Prediction

(Same for Radar and Laser)

Then I create update function for Laser.

```cpp
void KalmanFilter::Update(const VectorXd &z) {
  // Preparation
  MatrixXd I = MatrixXd::Identity(4, 4);

  // Measurement Update
  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;
  P_ = (I - K * H_) * P_;
}
```

Measurement

(Laser)

Finally, I create update function for Radar.

Radar measurement model is not linear, so **I calculated z_pred not using H matrix (A)**.

Angle phi should be between -PI and +PI, so **I added calculation (B)**.

```
void KalmanFilter::UpdateEKF(const VectorXd &z) {
  // Preparation
  float px = x_(0);
  float py = x_(1);
  float vx = x_(2);
  float vy = x_(3);
  float c1 = px*px + py*py;
  float c2 = sqrt(c1);
  float c4 = atan2(py,px);
  float c5 = (px*vx + py*vy)/c2;

  //MatrixXd Hj = tools.CalculateJacobian(x_); //don't use here...
  MatrixXd I = MatrixXd::Identity(4, 4);

  // Measurement Update
  VectorXd z_pred(3); // Different from Lasar, nonlinear Vector.
  z_pred << c2, c4, c5;
  VectorXd y = z - z_pred;

  if(y(1) < -M_PI){
    y(1) += 2*M_PI;
  }
  else if(y(1) > M_PI){
    y(1) -= 2*M_PI;
  }

  //MatrixXd Ht = Hj.transpose(); // Different from Lasar, use Jacobian. //don't use here...
  //MatrixXd S = Hj * P_ * Ht + R_; // Different from Lasar, use Jacobian. //don't use here...
  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;
  //P_ = (I - K * Hj) * P_;  // Different from Lasar, use Jacobian.//don't use here...
  P_ = (I - K * H_) * P_;
}
```

A

B

Measurement

(Radar)

(3) tools.cpp

The following function calculates RMSE, which is used in main.cpp.

To check the validity of the inputs, **I added assertion (A)**.

```cpp
#include "tools.h"
#include <iostream>
#include <cassert>

using Eigen::VectorXd;
using Eigen::MatrixXd;
using std::vector;

using std::cout;

Tools::Tools() {}

Tools::~Tools() {}

VectorXd Tools::CalculateRMSE(const vector<VectorXd> &estimations,
                              const vector<VectorXd> &ground_truth) {
  // Initialize RMSE value
  VectorXd rmse(4);
  rmse << 0,0,0,0;

  // check the validity of the following inputs:
  //  * the estimation vector size should not be zero
  //  * the estimation vector size should equal ground truth vector size
  //assert(estimations.size() != 0);
  //assert(estimations.size() == ground_truth.size());

  if (estimations.size() == 0){
    cout << "the estimation vector size should not be zero";
    assert(estimations.size() != 0);
  }
  if (estimations.size() != ground_truth.size()){
    cout << "the estimation vector size should equal ground truth vector size";
    assert(estimations.size() == ground_truth.size());
  }

  // Accumulate squared residuals
  for (int i=0; i<estimations.size(); i++){
     VectorXd tmp = estimations[i] - ground_truth[i];
     tmp = tmp.array()*tmp.array();
     rmse += tmp;
  }

  // Calculate the mean
  rmse /= estimations.size();

  // Calculate the squared root
  rmse = rmse.array().sqrt();

  return rmse;
```

The following function calculates Jacobian, which is used in FunctionEKF.cpp.

```cpp
MatrixXd Tools::CalculateJacobian(const VectorXd& x_state) {
  MatrixXd Hj(3,4);

  // Recover state parameters
  float px = x_state(0);
  float py = x_state(1);
  float vx = x_state(2);
  float vy = x_state(3);

  // Pre-compute a set of terms
  float c1 = px*px + py*py;
  float c2 = sqrt(c1);
  float c3 = c1*c2;

  // Check division by zero // It's done in FunctionEKF.cpp
  //assert(fabs(c1) < 0.0001);
  //if (fabs(c1) <= 0.0001){
  //  cout << "jacobian should not be divided by zero";
  //  assert(fabs(c1) > 0.0001);
  //}

   // 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;
}
```

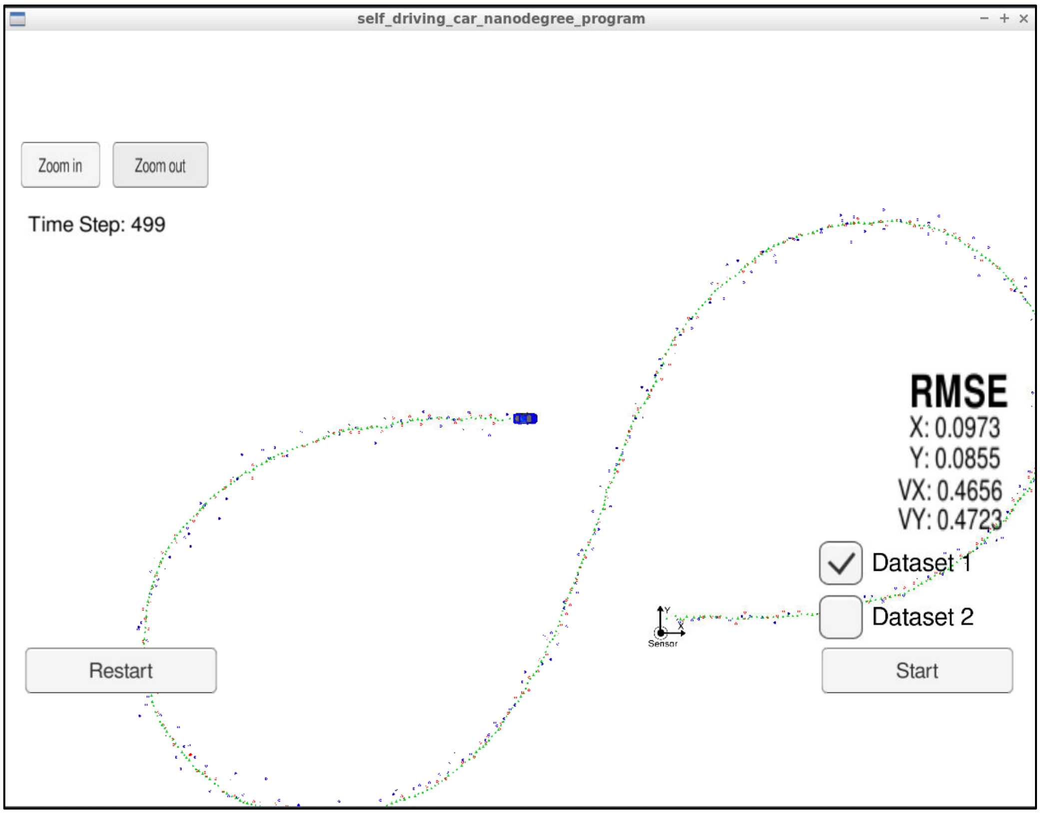
2. Result

I show the result below.

It achieved the target performance of RMSE in Project Rubric.

RMSE_x = 0.0973 < 0.11, RMSE_y = 0.0855 < 0.11

RMSE_vx = 0.4656 < 0.52, RMSE_vy = 0.4723 < 0.52



However, the **measurement positions of Radar** are less accurate than **Laser**, so sometimes **estimated positions become unstable**. I assume that the accuracy will be improved if I neglect the measurement positions that deviate a lot from the estimation positions.