

carnd_t2_p2_unscented_kalman_filter

Carnd - term 2 - project 2 - unscented kalman filter

Overview

The goal of this project is to build an unscented Kalman Filter (UKF) model to process a series of sensor data provided by radar and lidar. Similar to Project 1, Extended Kalman Filter (EKF) model, same source of radar and lidar data is used. The UKF model is connected to the same simulator via uWebSocketIO.

Project Repository

All resource are located in Udacity's project repository [CarND-Unscented-Kalman-Filter-Project](https://github.com/udacity/CarND-Unscented-Kalman-Filter-Project) (<https://github.com/udacity/CarND-Unscented-Kalman-Filter-Project>)

Project Submission

All modified code including results are committed to my personal github page [carnd-t2-p2-unscented-kalman-filter](https://github.com/chriskcheung/carnd_t2_p2_unscented_kalman_filter) (https://github.com/chriskcheung/carnd_t2_p2_unscented_kalman_filter)

Key Files

main.cpp

establishes communication between simulator and UKF model using uWebSocketIO, and reads in data during set time interval and send sensor measurements to UKF::ProcessMeasurement() in ukf.cpp for processing

ukf.cpp

contains 4 main functions: UKF::ProcessMeasurement(), UKF::Prediction(), UKF::UpdateLidarUKF(), UKF::UpdateRadar().

UKF::ProcessMeasurement() initializes state x and process covariance matrix P based on sensor data stypes: radar vs lidar. After initialization, delta time and meas_package will be fed to UKF::Prediction() for corresponding sigma point generation.

UKF::Prediction() generates sigma points X base on x and P_- . Then follow by generating augmented sigma points X_{sig_aug} withing adding acceleration noise stda and radial acceleration noise stdyawdd to existing process covariance matrix, becoming $P_{_aug}$. Then, using function $h(s, noise)$ to transform X_{sig_aug} predicted sigma points, X_{sig_pred} . Form $X_{sigpred}$, we applied weighted summation to its sigma points to calculate the predicted mean x , $(x_{k+1|k})$, and predicted covariance matrix P_- , $(P_{k+1|k})$.

UKF::UpdateRadar() uses state transaction function $h(x)$ to transform X_{sig_pred} into predicted measurement points Z_{sig} . From Z_{sig} , applies weighted summation to extract predicted measurement mean, z_pred . With the weighted summation of different between Z_{sig_pred} and z_pred , and the noise of radar equipment, we calculates measurement covariance matrix, S . Then using S , we determines

cross-correlation matrix, T , and calculated Kalman gain matrix, K_g . With T and K_g , state x and process covariance P is updated with the new state and covariance. At the end, Normalized Innovation Square, NIS, is calculated to verify consistence.

UKF::UpdateLidarUKF() uses the same method as UpdateRadar, but instead of using the state transition function $h(x)$ that composes of a 3-dimensional function for ρ , ϕ , and $\dot{\rho}$, the $h(x)$ is replaced with a 2-dimensional function for p_x and p_y .

tools.cpp

includes CalculateRMSE() for calculating the root means square of the predicted result versus real groundtruth measurement

Implementation Challenge

Initialization

If radar measurement input is received, use the ρ , ϕ , and $\dot{\rho}$ measurement to update state vector x . Otherwise, lidar measurement is straightforward to use p_x and p_y directly.

```
float px      = cos(phi)*rho;    // calculate position of x from radar
    metric phi and rho
float py      = sin(phi)*rho;    // calculate position of y from radar
    metric phi and rho
```

The last, update the `timeus` variable with the timestamp from measurement input as all measured sensor data are processed at this time.

Prediction

Prediction applies to both Radar and Lidar the same way as so each other. No different thread is needed to decide. One thing to look for is DO NOT attempt to normalize angles at sigma point creation. Will discuss more in later parallel.

```

// Prepare x augmented by adding noise to x vector, then calculate X augmented sigma points
VectorXd x_aug(n_aug_); // 7
x_aug.head(n_x_) = x_; // fill the first 5 element with x
x_aug(5) = 0;
x_aug(6) = 0;
MatrixXd P_aug(n_aug_, n_aug_); // P_aug size is 7x7
P_aug.fill(0.0);
P_aug.topLeftCorner(n_x_, n_x_) << P_;
P_aug(5,5) = std_a_ * std_a_;
P_aug(6,6) = std_yawdd_*std_yawdd_;
MatrixXd Xsig_aug(n_aug_, 2*n_aug_+1); //Xsig_aug size is 7x15
MatrixXd A_aug = P_aug.llt().matrixL(); // P_aug transpose
Xsig_aug.col(0) = x_aug;
for (int i=0; i<n_aug_; i++){
    Xsig_aug.col(i+1) = x_aug + sqrt(lambda_*n_aug_)*A_aug.col(i);
    // a mistake of normalizing angle at sigma point generations. notice that the first angle gets
    // normalized started at 1 instead of 0. So ends up some angles get normalized while some doesn't.
    // Leaving this here as commented code for example of bad normalization practice.
    // normalized angle
    while(Xsig_aug(3,i+1) > M_PI) { Xsig_aug(3,i+1) -= 2*M_PI; }
    while(Xsig_aug(3,i+1) < -M_PI) { Xsig_aug(3,i+1) += 2*M_PI; }*/

    Xsig_aug.col(i+n_aug_+1) = x_aug - sqrt(lambda_*n_aug_)*A_aug.col(i);
    // normalized angle
    // while(Xsig_aug(3,i+n_aug_+1) > M_PI) { Xsig_aug(3,i+n_aug_+1) -= 2
    *M_PI; }
    // while(Xsig_aug(3,i+n_aug_+1) < -M_PI) { Xsig_aug(3,i+n_aug_+1) += 2
    *M_PI; }*/
}

```

Update

Update for radar and lidar are different. Kalman filter update() for Lidar is quite straight forward by using state x directly.

```

        for(int i=0; i<2*n_aug+1; i++){
            MatrixXd Xx  = Xsig_pred_.col(i) - x_;

            // normalize angle
            while(Xx(3) >  M_PI) { Xx(3) -= 2*M_PI; }
            while(Xx(3) < -M_PI) { Xx(3) += 2*M_PI; }

            P_ += weights_(i) * Xx * Xx.transpose();
        }
        cout << "inside prediction(223)\n";
        cout << "predicted x_=\n" << x_ << endl;
        cout << "predicted P_=\n" << P_ << endl;

```

Normalizing Phi Angle

It is crucial to normalize phi angle in the range between π and $-\pi$. The explanation had been discussed previously in Project 1. Here for unscented Kalman Filter, phi angle is normalized every time there is a subtraction on the angle, normalization should take place to ensure the range of turning can only goes from 90 degree from the center (straight) to the left(-) and to the right(+). [Here](https://discussions.udacity.com/t/ekf-gets-off-track/276122/19?u=drivewell) (<https://discussions.udacity.com/t/ekf-gets-off-track/276122/19?u=drivewell>)

Overly done at prediction and sigma point generation can cause incorrect results on state x and covariance matrix P .

```

// normalize phi, y(1) in this case to be within -pi to pi
while(y(1) >  M_PI){ y(1) -= 2.*M_PI; }
while(y(1) < -M_PI){ y(1) += 2.*M_PI; }

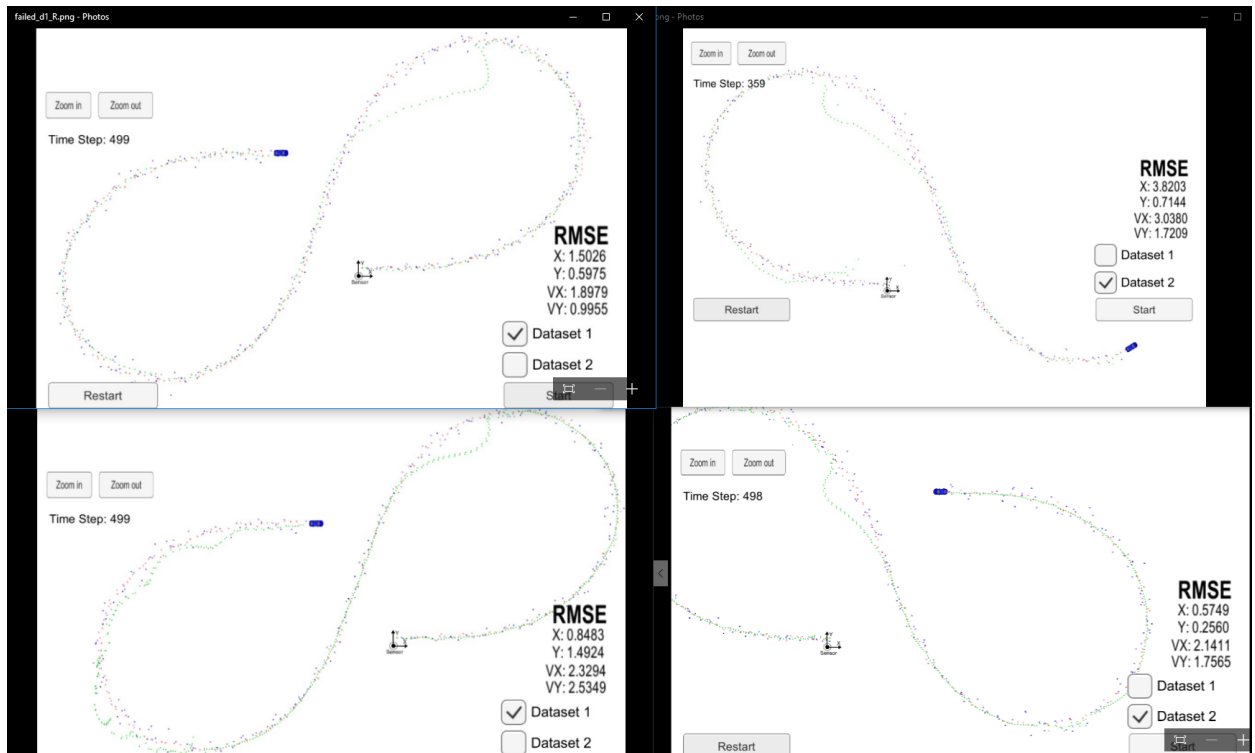
```

Result

Initial bring up of the code is short as I am using the old code structure from project 1. At the beginning, I got green tracker faulty mapped the estimation and prediction with bumps (off tracked) around 160 and 350 time stamps. As I observed, the calculated state x and covariance state matrix were increasing exponentially as raw measurement data was being processed. The problem was normalizing phi angles at sigma points generations and predicting X_{aug_pred} . It was a coding bug that normalization was not applied to the first sigma point phi but the rest of the sigma points, and causing an inconsistency in phi angles.

Failed attempts

The diagram below shows Radar measurement only on dataset 1 and dataset 2 on top, while Radar and Lidar measurement on dataset1 and dataset2 at the bottom



After removing the unnecessary phi normalization, the estimation (green tracker) was able to follow the tail of the car properly. Another bug discovered was caused by `std_a` and `std_yawdd` being too larger for bicycle database. After multiple attempts to adjust the noise, I settled to the following. I also chose to initialize `P_` matrix to zero except `P_(0,0)=1` and `P_(1,1)=1`. This gave me a better result on RMSE. But sure there is room to improve.

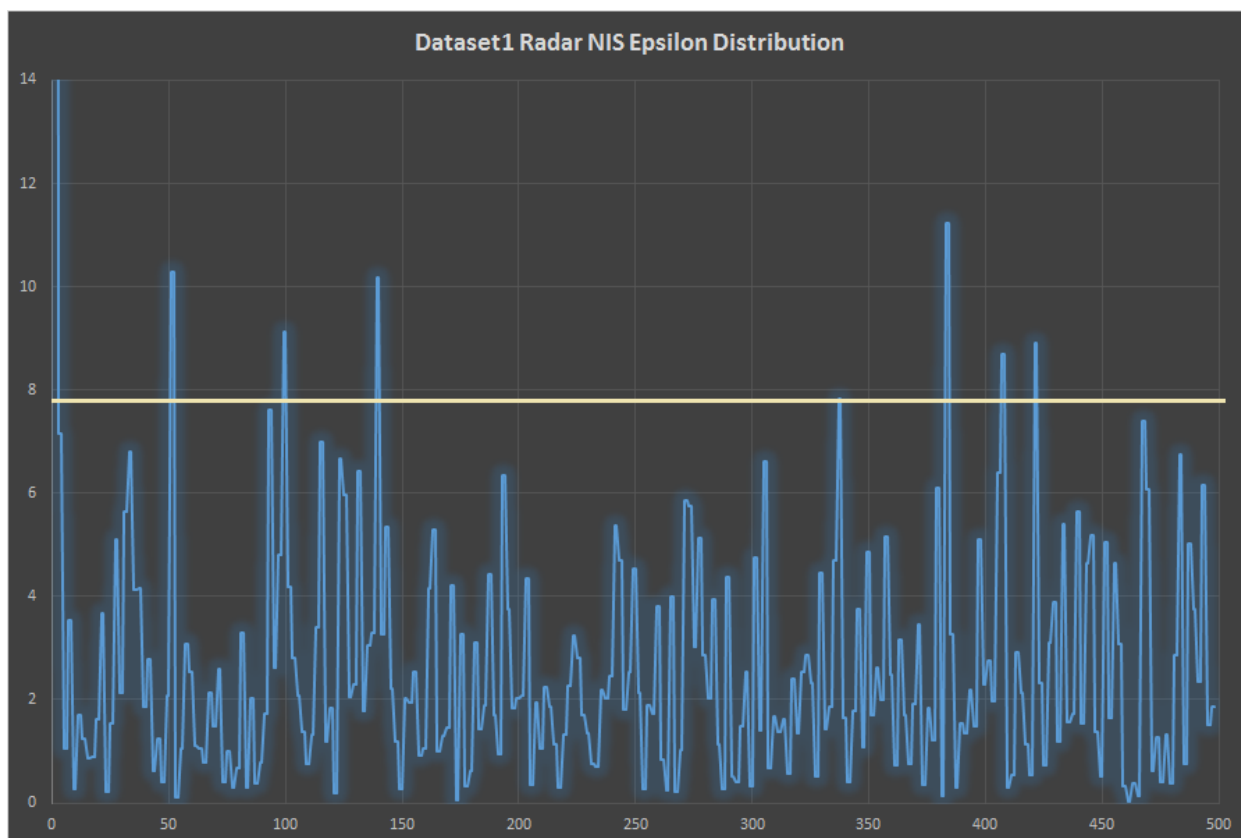
```
std_a_ = 2.5;
std_yawdd_ = 0.5;
```

The best RMSE is close to the required rubric guideline:

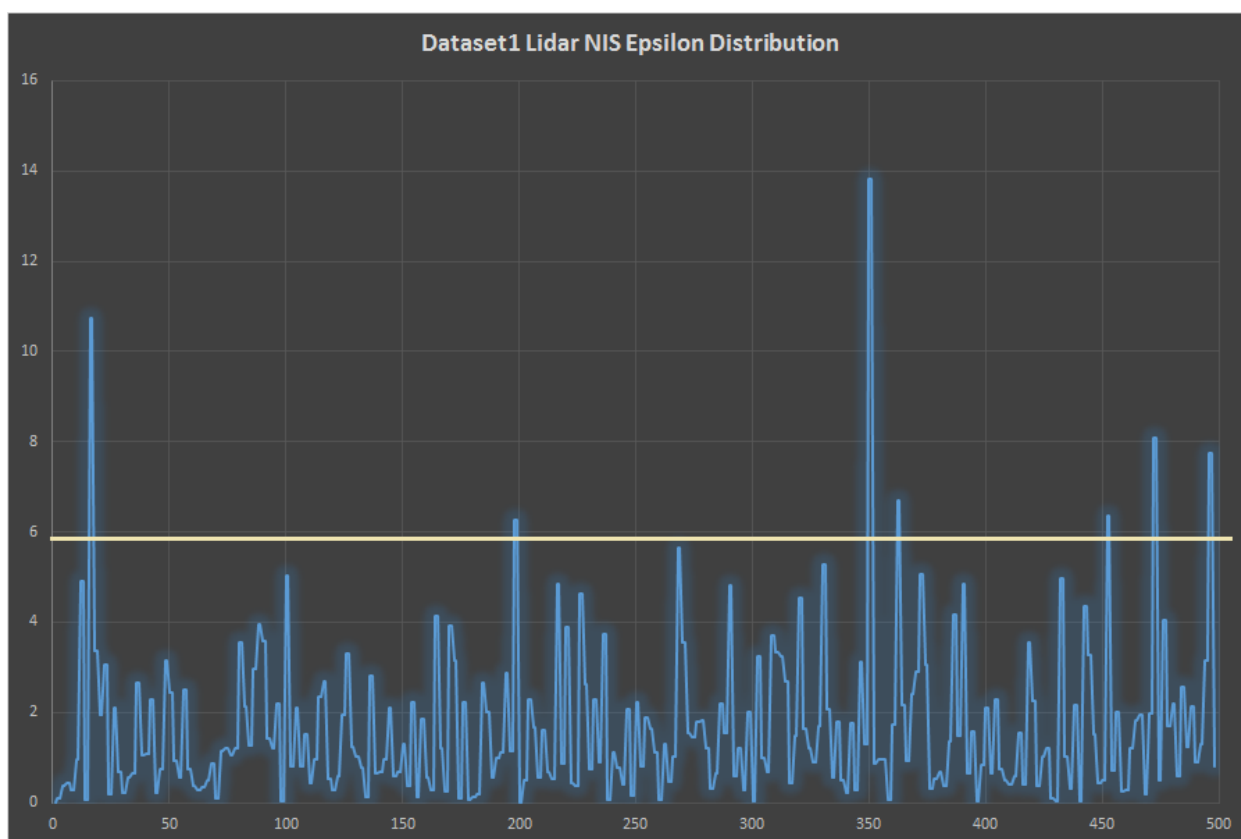
```
for dataset1, (px, py, vx, vy)=(0.07, 0.08, 0.37, 0.18)
for dataset2, (px, py, vx, vy)=(0.08, 0.07, 0.52, 0.19)
```

The radar epsilon value seems within the expectation as suggested from the lection. 95% of epsilon value are below 7.8 according to a 3 degree of freedom, while lidar epsilon value stays within consistency as well as 95% of epsilon value are below 5.9 according to 2 degree of freedom as shown in charts below.

Radar Epsilon

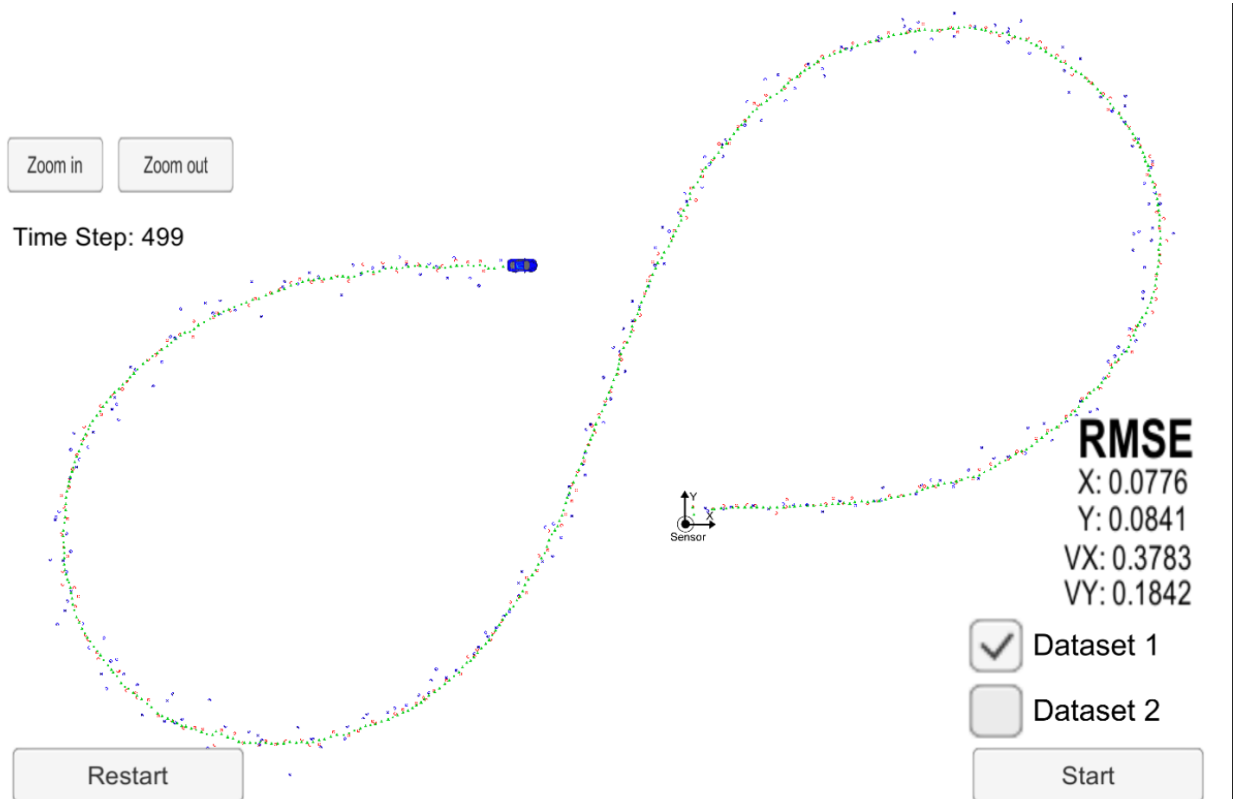
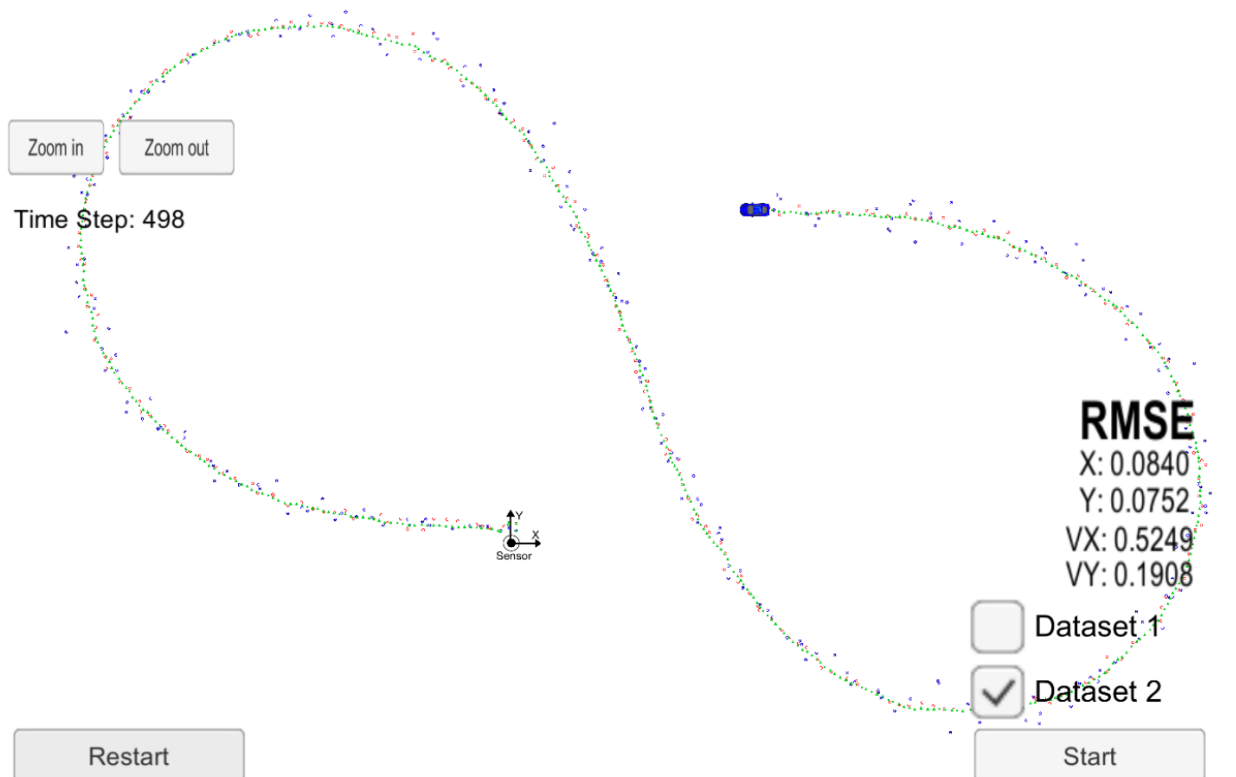


Lidar Epsilon



Passing cases after fixing bug

Radar and Lidar measurement

dataset 1**dataset 2**

Overall, initial bring up of this project is not difficult as all instruction had been provided. However, it did require a lot time to debug issue and tuning the acceleration and radial acceleration noises to minimize the RMSE.

In []: