Extended Kalman Filters

This document is the writeup of the Extended Kalman Filters. Here are the rubric points.

Compiling

Code compiles on my Mac OS 10.12.16 with XCode Version 9.0 (9A235). I used the provided XCode profile and not changed the CMakeLists.txt nor added nor removed files.

The programs implement the Extended Kalman Filter for Radar measurements and the Kalman Filter for Laser measurements.

Also the program compiles correctly with cmake/make but gives a warning :

```
ld: warning: directory not found for option '-L/usr/local/Cellar/libuv/1.11.0/lib'
```

that doesn't seem to affect its behavior.

Accuracy

I measured accuracy over DataSet 1 and DataSet 2. Also I added a different initialization that is compared in the New Init columns.

I run the code with Laser and Radar, only Laser and only Radar. Results are in the following table:

	Dataset 1			
RMSE	**Laser & Radar**	**Laser & Radar new Init**	Only Laser	Only Radar
x	**0.0974**	**0.0967**	0.1353	0.8335
y	**0.0856**	**0.0851**	0.1137	0.617
vx	**0.4676**	**0.4499**	1.3272	0.8193
vy	**0.4346**	**0.4226**	1.1668	1.1314

	Dataset 2			
RMSE	**Laser & Radar**	**Laser & Radar new Init**	Only Laser	Only Radar
x	**0.0728**	**0.0728**	0.1123	0.8857
y	**0.0967**	**0.0965**	0.1133	0.5875
vx	**0.468**	**0.4433**	1.3329	0.8915
vy	**0.5237**	**0.513**	1.2272	1.1518

Values are below the reference of 1.1, 1.1, 0.52, 0.52 in the Dataset 1 for Laser and Radar and also in Dataset 2 except for vy in the Laser and Radar.

When using the new init values are a little better and the vy in Dataset 2 also is below the desired value.

Follows the correct algorithm

Your Sensor Fusion algorithm follows the general processing flow as taught in the preceding lessons :

- The program, for each measurement does the Predict and the Update steps.

```
/*****************************************************************************
 *  Prediction
 ****************************************************************************/

double delta = (measurement_pack.timestamp_ - previous_timestamp_) / 1000000.0;

double sigma2x = 9;     // Logically should came from measurement?
double sigma2y = 9;

ekf_.F_(0, 2) = delta;
ekf_.F_(1, 3) = delta;

ekf_.Q_ = MatrixXd(4, 4);

ekf_.Q_ << pow(delta,4)*sigma2x/4, 0, pow(delta,3)*sigma2x/2, 0,
    0, pow(delta,4)*sigma2y/4, 0, pow(delta,3)*sigma2y/2,
    pow(delta,3)*sigma2x/2, 0, pow(delta,2)*sigma2x, 0,
    0, pow(delta,3)*sigma2y/2, 0, pow(delta,2)*sigma2y;

ekf_.Predict();

/*****************************************************************************
 *  Update
 ****************************************************************************/

if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
    if (true){
        ekf_.R_ = R_radar_;
        ekf_.UpdateEKF(measurement_pack.raw_measurements_);
        Hj_ = ekf_.H_;
        previous_timestamp_ = measurement_pack.timestamp_;
    }

} else {
    if (true){
        ekf_.H_ = H_laser_;
        ekf_.R_ = R_laser_;
        ekf_.Update(measurement_pack.raw_measurements_);
        H_laser_ = ekf_.H_;
        // Laser updates
        previous_timestamp_ = measurement_pack.timestamp_;
    }
}
```

Your Kalman Filter algorithm handles the first measurements appropriately:

- The program uses the first measurement to initialize the state. A change from the proposed initialization is to use the R values to set the Covariance Matrix for x and y.
- In the Laser case they are written directly.

- In the Radar case there is a little more involved. See the **Initializing Covariance Matrix with Radar**.
- Result with this more sophisticated initialization is better in the two datasets.

```
if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
    /**
     Convert radar from polar to cartesian coordinates and initialize state.
     */

    double r = measurement_pack.raw_measurements_[0];
    double theta = measurement_pack.raw_measurements_[1];

    ekf_.x_[0]  = r * cos(theta);
    ekf_.x_[1]  = r * sin(theta);
    ekf_.P_ = MatrixXd(4,4);

    ekf_.P_ << R_radar_(0,0)*cos(theta)+r*sin(theta)*R_radar_(1,1), 0, 0, 0,
    0, R_radar_(0,0)*sin(theta)+r*cos(theta)*R_radar_(1,1), 0, 0,
    0, 0, 10000, 0,
    0, 0, 0, 10000;

}
else if (measurement_pack.sensor_type_ == MeasurementPackage::LASER) {
    /**
     Initialize state.
     */

    ekf_.x_[0] = measurement_pack.raw_measurements_[0];
    ekf_.x_[1] = measurement_pack.raw_measurements_[1];
    ekf_.P_ = MatrixXd(4,4);

    ekf_.P_ << R_laser_(0,0), 0, 0, 0,
    0, R_laser_(1,1), 0, 0,
    0, 0, 10000, 0,
    0, 0, 0, 10000;

}
```

Your Kalman Filter algorithm first predicts then updates:

- As seen in the code posted in the first point, first comes Predict() and then Update().
- Prediction doesn't depend on the type of measurement
- Updating depends on it.
-

Your Kalman Filter can handle radar and lidar measurements

- Radar and Lidar measurements are treated different in :
    - Initialization in first measure
    - Calling the Update functions
- But essentially differences are in the setting of the matrices (in FusionEKF.cpp) and in computing the value to subtract to the measurement from the state.
- Rest is done in the same functions.

Code Efficiency

Your algorithm should avoid unnecessary calculations.

- Matrix values that doesn't change are taken out of the loops either in object creation or in the initialization.
- When multiplying matrices, Inverse and Transpose are computed just once, stored in a variable and then used in the computations.

- Angle normalization is done only in the error angle, not in the measurement of the state and removes all 2Pi multiples. That is very important for the correct function of the program but normalizing the measurement or state angles is useless as they are subtracted and we only use the difference value sot when normalizing this one all 2PI multiples will be removed.
- In the Jacobian computation some intermediate values are also computed in advance so the expression is much clearer. (tools.cpp file)
- Instead of using **atan** function and having to test different cases of y/x when x is near 0, we use **atan2** function that already does all the work for us. (kalman_filter.cpp).
- No new structures are created. Just added a new function **UpdateKalman()** in kalman_filter.h and kalman_filter.cpp.
- This modification allow us to have a function that applies the Kalman update equations for normal and Extended Kalman Filters. The corresponding functions just use H or compute Hj and then compute the error value differently in each case. Then UpdateKalman is called with the error :

```
void KalmanFilter::Update(const VectorXd &z) {

    VectorXd y  = z - H_ * x_; // H_ is fixed in this case. No need to
recompute it every timestep
    UpdateKalman(y);
}

void KalmanFilter::UpdateEKF(const VectorXd &z) {

    Tools  t = Tools();
    H_ = t.CalculateJacobian(x_);   // In Extended Kalman H is the Jacobian

    // Compute the converted state. Function is applied directly
    //

    VectorXd xc(3);
        xc << sqrt(x_[0]*x_[0]+x_[1]*x_[1]),
        atan2(x_[1],x_[0]),
        (x_[0]*x_[2]+x_[1]*x_[3])/sqrt(x_[0]*x_[0]+x_[1]*x_[1]);

    VectorXd y = z - xc;  // Compute Error

    // Normalize angle error. That is important if not error may be offseted
by 2PI

    while (y[1] > M_PI){
        y[1] -= 2*M_PI;
    }

    while (y[1] < -M_PI){
        y[1] += 2*M_PI;
    }
    UpdateKalman(y);
 }
```

and

```
void KalmanFilter::UpdateKalman(VectorXd &error){
    long x_size = x_.size();
    MatrixXd I = MatrixXd::Identity(x_size, x_size);

    MatrixXd Ht = H_.transpose();
    MatrixXd S = (H_ * P_ * Ht) + R_;
    MatrixXd K = P_ * Ht * S.inverse();
```

```
            x_ = x_ + (K * error);
            P_ = (I - K * H_) * P_;

        }
```

Initializing Covariance Matrix with Radar.

I have modified the standard covariance matrix initialization because we have some information about the error of the measurement.
So in the Lidar case it is simple. Instead of using 1 or a constant I use the value of R.

But in the Radar case we must translate the error values in distance and angle to error values in position so we have :

$$x = r\,Cos(\varphi)$$
$$y = r\,Sin(\varphi)$$

So taking partial derivatives against r and φ we get, adding the errors without sign:

$$\Delta x = \Delta r\,Cos(\varphi) + r\,Sin(\varphi)\Delta\varphi$$
$$\Delta y = \Delta r\,Sin(\varphi) + r\,Cos(\varphi)\Delta\varphi$$

So we may initialize the Covariance matrix with these values as R_Radar gives us Δr and $\Delta\varphi$.

Discussion

As stated before I run the algorithm with Radar and Lidar, only Radar and only Lidar.

Comparing Radar and Lidar with only Lidar we see that RMSE increases a little for x and y but near triplicates for vx and vy. That's quite logical as Radar gives us a speed measurement.

When using only Radar error in position x and y increases about 8 times and although speed errors are not as big as using only lidar they are not very good.

All this compares correctly with the characteristics of the sensors:

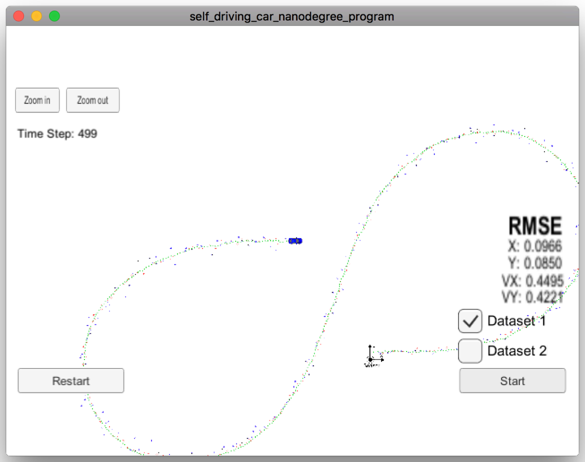
LIDAR : Precision but only gives us position
RADAR : Less precision but gives us radial speed

Fusion of the two sensors give us the best of the two worlds.

When adding the new initialization, values get better:

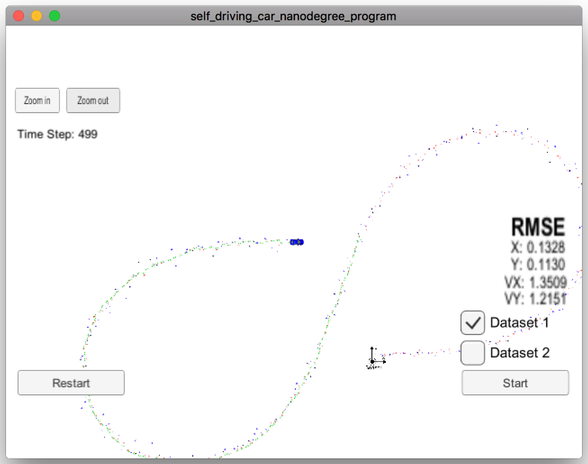
RMSE	Laser & Radar	Laser & Radar new Init
x	0.0974	0.0967
y	0.0856	0.0851
vx	0.4676	0.4499

vy	0.4346	0.4226



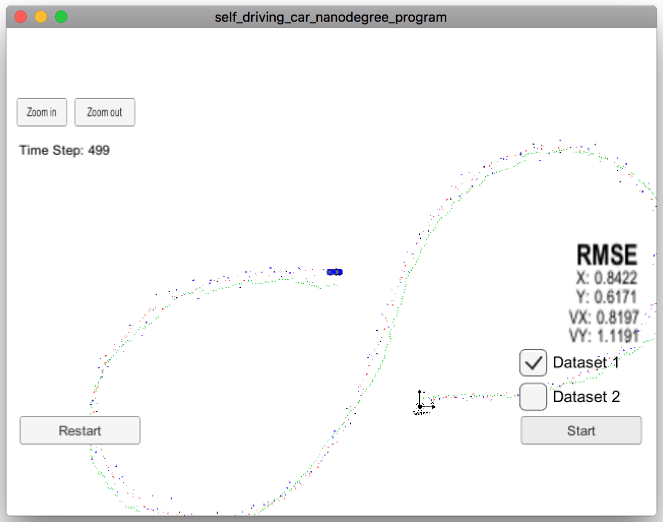
RMSE
X: 0.0966
Y: 0.0850
VX: 0.4495
VY: 0.4221

Figure 1 Lidar and Radar



RMSE
X: 0.1328
Y: 0.1130
VX: 1.3509
VY: 1.2151

Figure 2 Only Lidar



RMSE
X: 0.8422
Y: 0.6171
VX: 0.8197
VY: 1.1191

Figure 3 Only Radar