Extended Kalman Filter Project

Please find below the steps done for EKF.

Initialization

In FusionEKF.cpp file, the following has been done:

- In FusionEKF constructor:
  - The variables have been initialized:
    - R_laser, R_Radar, H_laser_, F_ (before adding delta t)
- In ProcessMeasurement function:
  - State covariance matrix P is initialized and Q
  - The state initial value is set to the sensor's initial value

```
ekf_.x_ << measurement_pack.raw_measurements_[0],
           measurement_pack.raw_measurements_[1],
           0,
           0;
```

  - If Radar:
    - The state is converted from Cartesian to polar, by multiplying ro* cos(phi) and ro * sin(phi).

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

  // Convert position from polar to cartesian and initialize state
  float ro = measurement_pack.raw_measurements_[0];
  float phi = measurement_pack.raw_measurements_[1];
  ekf_.x_ << ro * cos(phi),
             ro * sin(phi),
             0,
             0;

  ekf_.R_ = R_radar_;

}
```

    - The filters R is set to R_radar
  - If Laser:
    - State is kept as is with the initial value
    - The filter's R is set to R_Laser
    - The state is already initialized from the beginning.
  - Dt is calculated to compute the Q matrix and update F matrix.

```
// dt - expressed in seconds
float dt = (measurement_pack.timestamp_ - previous_timestamp_) / 1000000.0;
previous_timestamp_ = measurement_pack.timestamp_;
float dt_2 = dt * dt;
float dt_3 = dt_2 * dt;
float dt_4 = dt_3 * dt;

// update F
ekf_.F_(0,2) = dt;
ekf_.F_(1,3) = dt;

// update Q
ekf_.Q_ <<  dt_4/4*noise_ax, 0, dt_3/2*noise_ax, 0,
      0, dt_4/4*noise_ay, 0, dt_3/2*noise_ay,
      dt_3/2*noise_ax, 0, dt_2*noise_ax, 0,
      0, dt_3/2*noise_ay, 0, dt_2*noise_ay;
```

Prediction step

Prediction step is common between laser and radar sensors. The below equation is implemented.

```
void KalmanFilter::Predict() {
  /**
   * TODO: predict the state
   */

  cout << " ************************************** PREDICT ************************************** " << std::endl;
  x_ = F_ * x_;
  MatrixXd Ft = F_.transpose();
  P_ = F_ * P_ * Ft + Q_;

  cout << "(Debug) - x_= " << x_ << std::endl;
}
```

Before Each step, I added "****** Step Name *****" for better visualization in terminal.

Update steps (function calls)

- If sensor is laser, then R is assigned the (2x2) R_laser.
- H is assigned H_laser
- Update function is called

```
} else {
  // TODO: Laser updates

  // Update R
  ekf_.R_ = R_laser_;

  // Update H
  ekf_.H_ = H_laser_;

  // Call laser update function
  ekf_.Update(measurement_pack.raw_measurements_);

}
```

- If sensor is Radar, R is assigned (3x3) R_radar.
- H is assigned Hj (jacobian matrix)
- UpdateEKF function is called

```
if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
  // TODO: Radar updates

  // Update R
  ekf_.R_ = R_radar_;

  // Update H with jacobian
  ekf_.H_ = tools.CalculateJacobian(ekf_.x_);

  // Call Radar updated function
  ekf_.UpdateEKF(measurement_pack.raw_measurements_);
```

Update Function (Laser)

- Z_pred is obtained by multiplying the state directly with H, and Kalman equations are implemented as below

```
void KalmanFilter::Update(const VectorXd &z) {
  cout << " ************************************** UPDATE LASER ****************
  /**
   * TODO: update the state by using Kalman Filter equations
   */
  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_;

  cout << "(Debug) - x_ = " << x_ << std::endl;
}
```

UpdateEKF function (Radar):

- Z_pred is initialized and obtained by first converting the state into polar coordinates and then subtract it from the sensor measurement (z)
- Instead of doing atan(py/px) , atan2(py,px) is used which provides output from -180 to 180 exactly as the sensor measurement z

```
void KalmanFilter::UpdateEKF(const VectorXd &z) {
  /**
   * TODO: update the state by using Extended Kalman Filter equations
   */

  cout << " ************************************** UPDATE RADAR ****************

  // Create the h matrix
  VectorXd z_pred(3);
  float px = x_[0];
  float py = x_[1];
  float vx = x_[2];
  float vy = x_[3];

  // z_pred << sqrt(pow(px,2) + pow(py,2)),
  //              atan(py / px),
  //              (px * vx + py * vy) / (sqrt(pow(px,2) + pow(py,2)));

  z_pred << sqrt(pow(px,2) + pow(py,2)),
            atan2(py , px),
            (px * vx + py * vy) / (sqrt(pow(px,2) + pow(py,2)));
```

- Hj has been already computed and assigned to H before calling UpdateEKF function
- Kalman equations are implemented using the Hj instead of H.

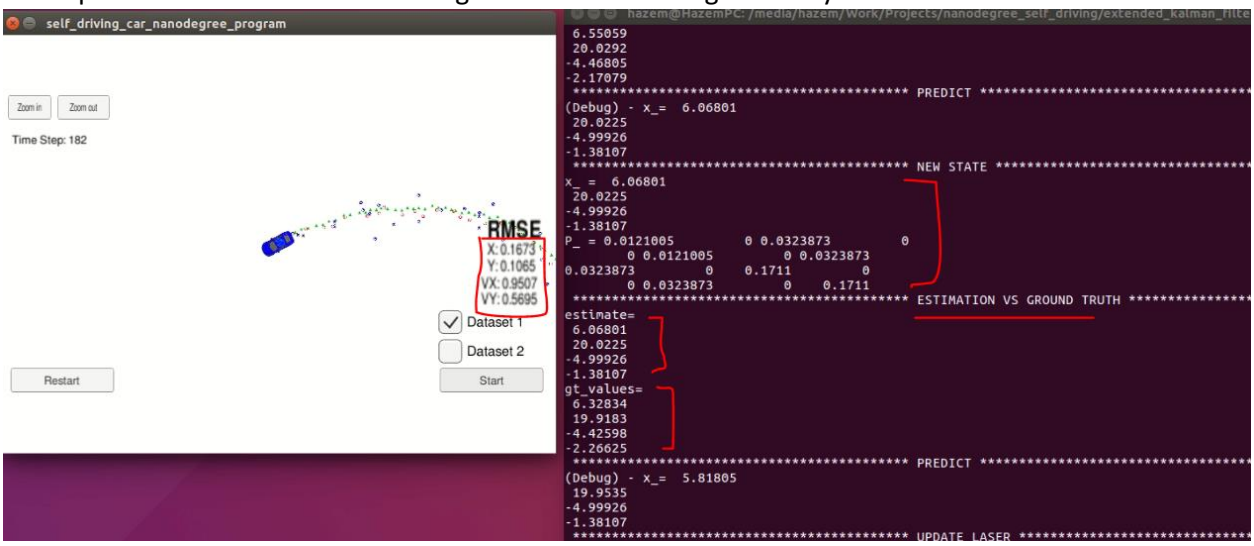
Tools file

- CalculateRMSE and CalculateJacobian are implemented as the instructions.

Results

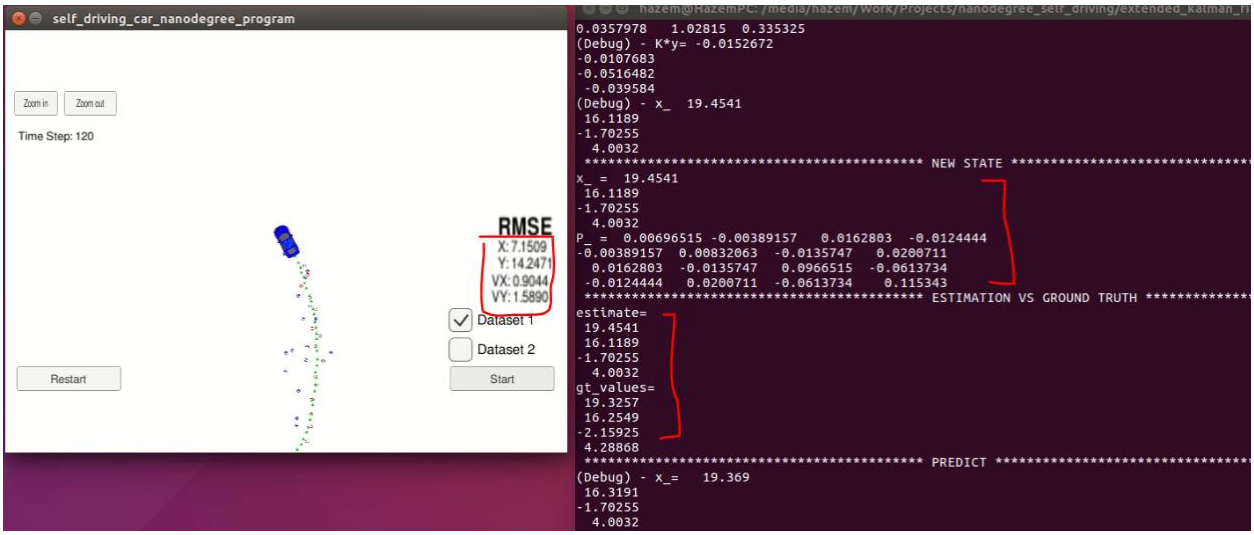
Laser updates only:

- When the radar update function UpdateEKF is commented out, laser updates only are considered.
- RMSE provides low values and tracking seems to be working correctly



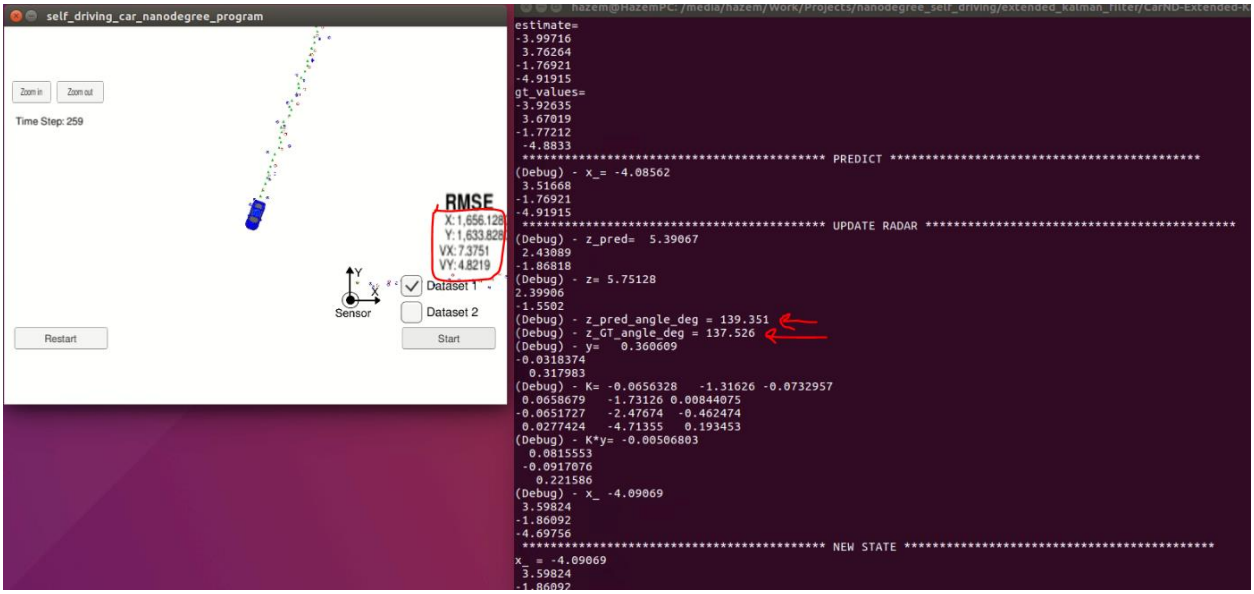
Sensor Fusion:

- When UpdateEKF function is uncommented, accuracy should increase, however RMSE increases as shown below

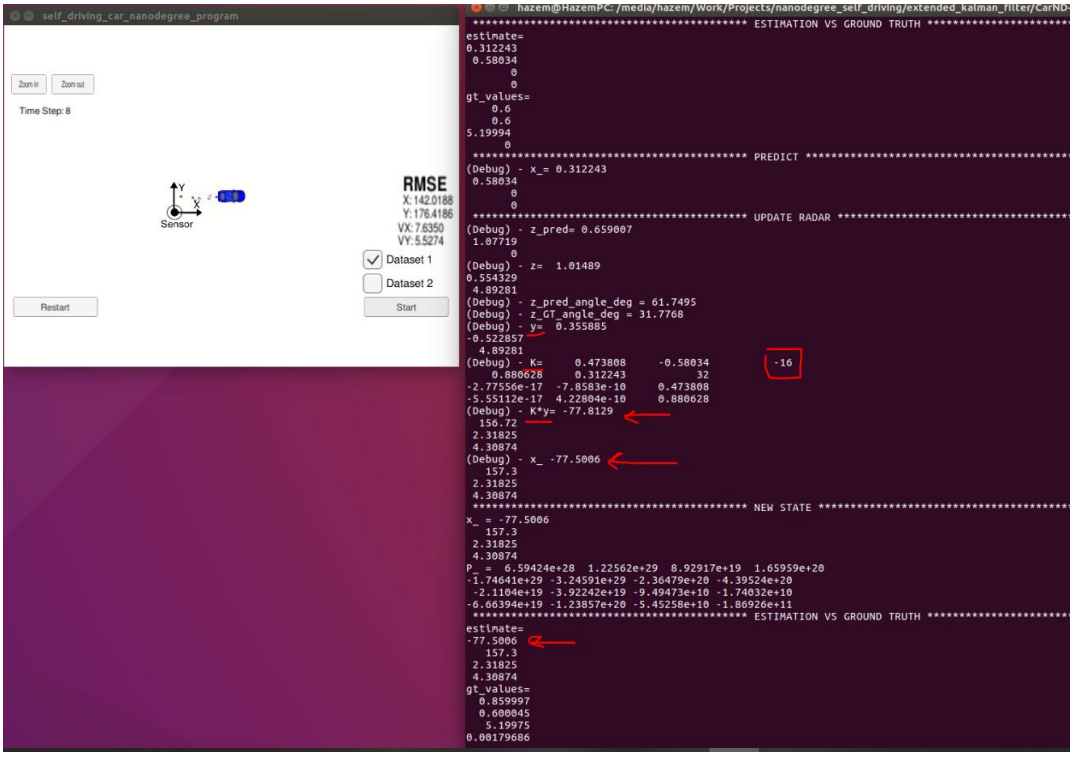


Radar updates only:

- RMSE increases and obviously there should be an error. I tried to debug that, but until this moment it is not found.
- It is noticed that atan2(py,px) produces angle values which are very similar to the sensor measurement, which confirms the implementation should be correct.



- In the beginning, Kalman gain is very large which results in incorrect prediction for the state, and error accumulates.



- I tried debugging the jacobian matrix, however, it is exactly implemented as done in instructions. Y value seems correct, so I believe there is a problem in polar to Cartesian conversion.
- I tried debugging that part, but still, it seems for me the conversion is done correctly.

Summary

- Kalman filter is implemented, and sensor fusion is implemented.
- Prediction function performs quite well.
- Laser update function perform quite well.
- Radar update function seem to have an error which is not visible for me. I believe it is something related to conversion from polar to Cartesian, because this is the only difference between laser and radar updates. The other differences which are jacobian matrix is implemented as instructions and tested, so there should be no error in that part, and the resulting Hj is used in the rest of equations.
- It would be great if you have a look on the code, and advise if there is a problem in the UpdateEKF function.