Extended Kalman Filter Project

Jean-François Vanreusel

Introduction

In this project, we use Extended Kalman Filter to track a cyclist that travels around our car. The state of the cyclist is provided through his/her 2-dimensional position (Px, Py) and 2-dimensional velocity (Vx, Vy).

Data is provided through 2 types of sensors: LIDAR and RADAR. LIDARs only provide information on the position. RADARs provide information on both position and velocity.

Our software returns predicted positions of the cyclist from two different data sets.

It also returns the accuracy of our software by comparing our predictions with provided ground truth data.

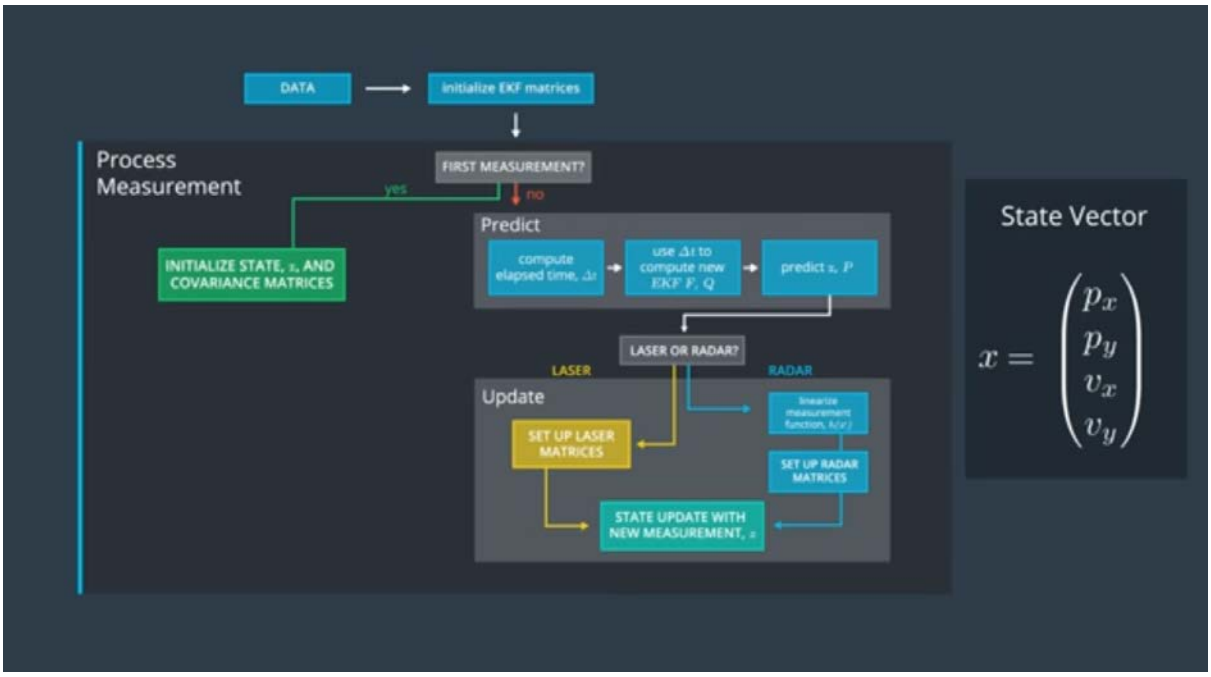
Deliverables

For this project, we provided the following files:

- **Source files**
  - **Main.cpp :** main file for this software. Takes in the arguments, reads in data files and outputs results.
  - **Fusion EFKF.cpp (&.h)**: operates the fusion of different signals. It initializes the Extended Kalman Filter matrices and calls the different EK Filter methods based on their source (LIDAR, RADAR).
  - **Kalman_filter.cpp (&.h):** performs Kalman filter algorithm for both LIDAR and RADAR data. It basically predicts the new state and updates the measurements.
  - **Tools.cpp (&.h):** includes the code for the Jacobian matrix and the Root-Mean-Squared Error that we use to validate our results.
  - **Ground_truth_package.h:** defines the Ground Truth Package class.
  - **Measurement_package.h:** defines the measurement class.
- **Build file**
  - **CMakeLists.txt:** Make file to compile and link source code.
- **Data files**
  - **Data/sample-laser-radar-measurement-data-1.txt:** first set of LIDAR and RADAR data as measured by devices.
  - **Data/sample-laser-radar-measurement-data-2.txt:** second set of LIDAR and RADAR data as measured by devices.
  - **Data/output1.txt:** results from running software on first dataset
  - **Data/output2.txt:** results from running software on second dataset

- **Extended Kalman Filter Project.docx.pdf:** You're reading it. This contains our project overview.

Overall EKF Algorithm



The cyclist position and velocity is defined through a state vector. Px, Py represent the position of the cyclist. Vx, Vy represent the cyclist's velocity.

As we receive data, we first initialize our state x and covariance matrices. Covariance matrices provide an idea on the accuracy of the state. Then, with subsequent data, we continuously go through a "predict" and "measurement update" step – until no more data is provided.

The Predict step starts by computing the time lapse between the time we received the previous data and when we receive our current one. For example, if we know that the cyclist's velocity was 2 steps per second in a certain direction, we can predict where he would be after 2 seconds (i.e. 4 steps away from the previous position).

Once we predict the cyclist's position, we also update our state and covariance matrices. We have two types of data source: LIDAR and RADAR. Each data type needs to be handled in a slightly different way because LIDARs provide coordinates in a Cartesian system while RADARs provide polar coordinates. Also, because RADAR provide coordinates in a polar system, our prediction function is non-linear which doesn't work with Kalman filters. So, we first need to transform a non-linear measurement function into a linear approximation of our measurement function. We do this through the First order Taylor expansion, using a Jacobian matrix.

With RADAR, we also need to convert our predicted state from a Cartesian system into a Polar system used by RADAR. This allows us to compare our current state with the measurement provided by the RADAR.

FusionEKF.cpp (and .h)

The FusionEKF .cpp and .h files focus on the fusion algorithm to (1) initialize the state and the covariance matrices and (2) dispatch the data to the right Kalman Filter algorithm.

In the FusionEKF.cpp file, we initialize the state with the first data record. When this data record is provided by a radar, we need to convert its coordinates from a polar system into a Cartesian system as follows:

```
/**
      Convert radar from polar to cartesian coordinates and initialize state.
      */

      //(rho, phi, rho_dot) -> (rho*cos(phi),rho*sin(phi))
      double rho = measurement_pack.raw_measurements_[0];
      double phi = measurement_pack.raw_measurements_[1];

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

      ekf_.x_ << px, py, 0, 0;
```

Fusion EKF also initializes the covariance matrices and the initial state using the provided data. It then calls the right Kalman Filter function based on the data source (LIDAR, RADAR).

Kalman_filter.cpp (and .h)

The Kalman filter file includes the methods to perform the Prediction and Measurement update steps of the Kalman Filter approach. The same Prediction method can be used for both LIDAR and RADAR. The measurement update method needs to be adapted for each data source because of the non-linear aspect of the RADAR function.

Tools.cpp (and .h)

The Tools files include methods to

(1) Compute Root-Mean-Squared-Error (RMSE) between the predicted states and ground_truth states
(2) Compute the Jacobian matrix. The Jacobian matrix is necessary to “linearize” the measurement update function.

Results

The software was run on two different data sets:

- Data/sample-laser-radar-measurement-data-1.txt
- Data/sample-laser-radar-measurement-data-2-txt

Results are included in the following files:

- Data/output-1.txt

```
x_:
[[ 11.37];
 [-1.876];
 [0.7339];
 [ 2.689]]
P_:
[[ 0.004021, 0.0007907,  0.007534,  0.003089];
 [0.0007907,  0.008773,  0.002058,   0.02408];
 [ 0.007534,  0.002058,   0.07314,   0.01334];
 [ 0.003089,   0.02408,   0.01334,    0.1526]]

Accuracy - RMSE:
0.0651649
0.0605378
  0.54319
 0.544191
```

RMSE is within the expected range.

- Data/output-2-txt

RMSE is within the expected range:

```
x_:
[[   204];
 [  36.2];
 [ 1.203];
 [0.2307]]
P_:
[[  0.01423, -0.001438,   0.01088, -0.004644];
 [-0.001438,   0.02207, -0.004765,   0.03692];
 [  0.01088, -0.004765,    0.0873,    -0.131];
 [-0.004644,   0.03692,    -0.131,    0.8009]]

Accuracy - RMSE:
0.185496
0.190302
0.476754
0.804469
```

Validation with Tracker tool

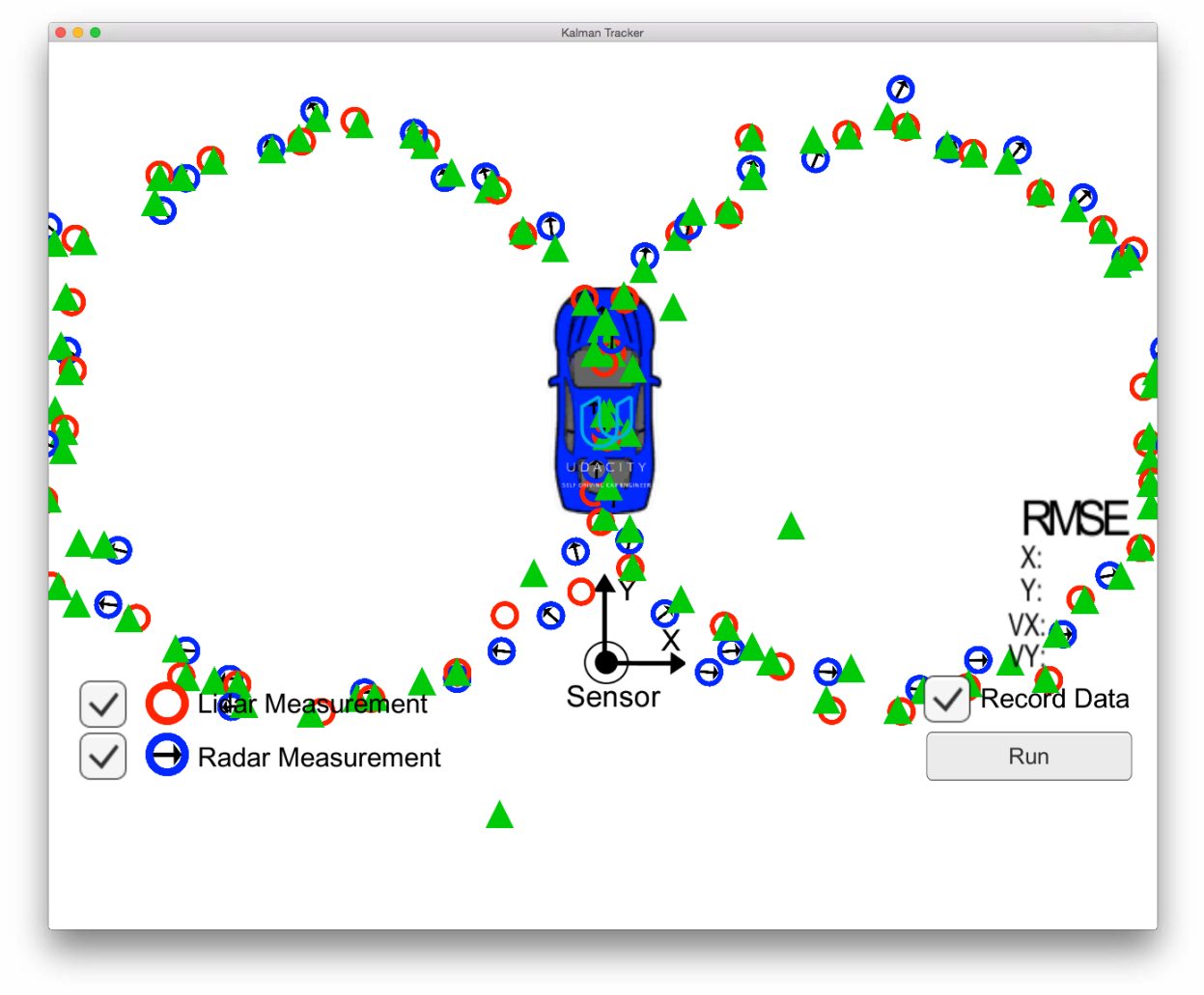


Figure 1: Tracking both Lidar and Radar measurements

Using both Lidar and Radar measurement, we observe fair results on the left loop and stronger results with the right loop.

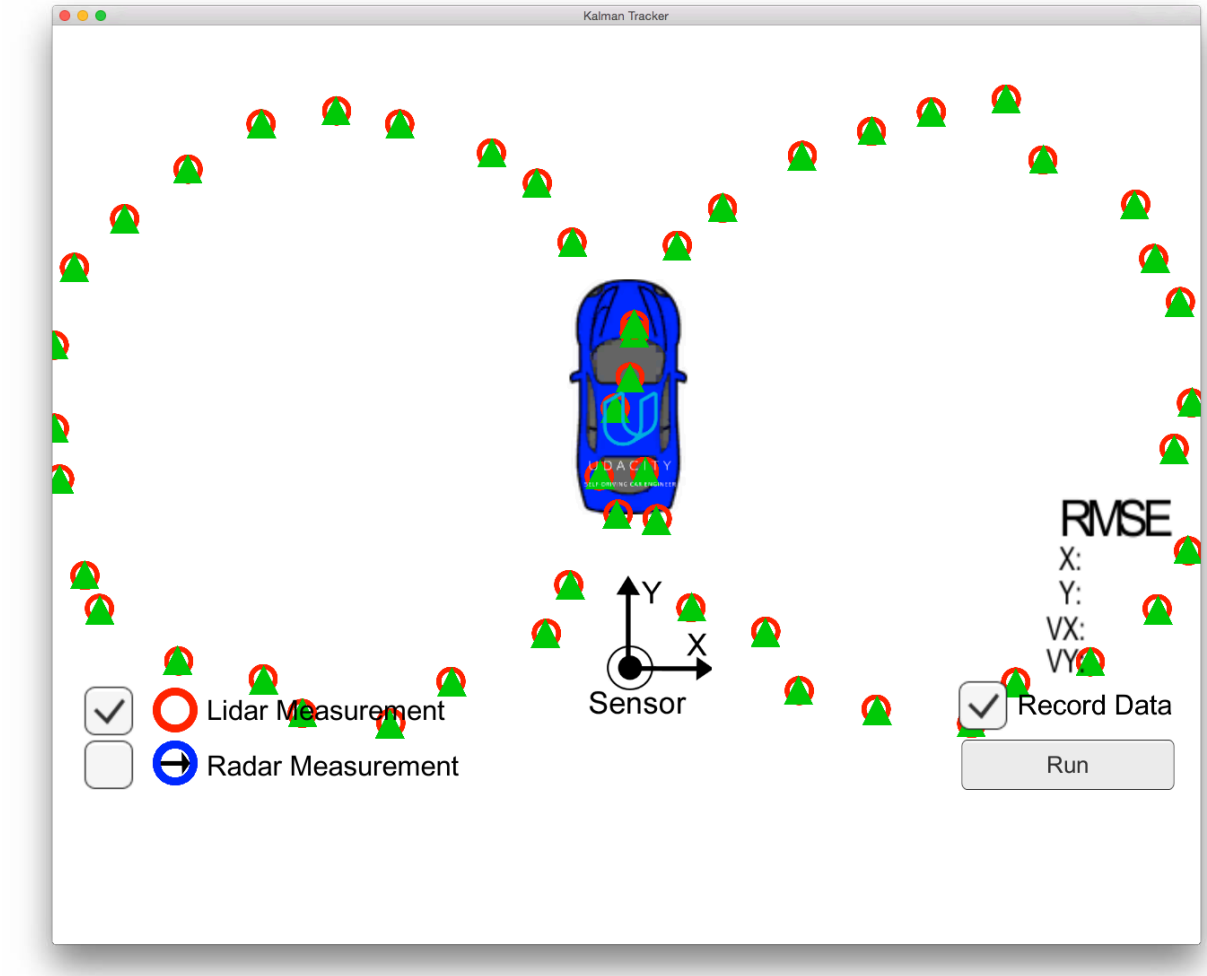


Figure 2: Lidar only measurements

Our software predicts shows some strong results with the Lidar measurements. This is as anticipated since Lidar usually provide stronger accuracy for positions.

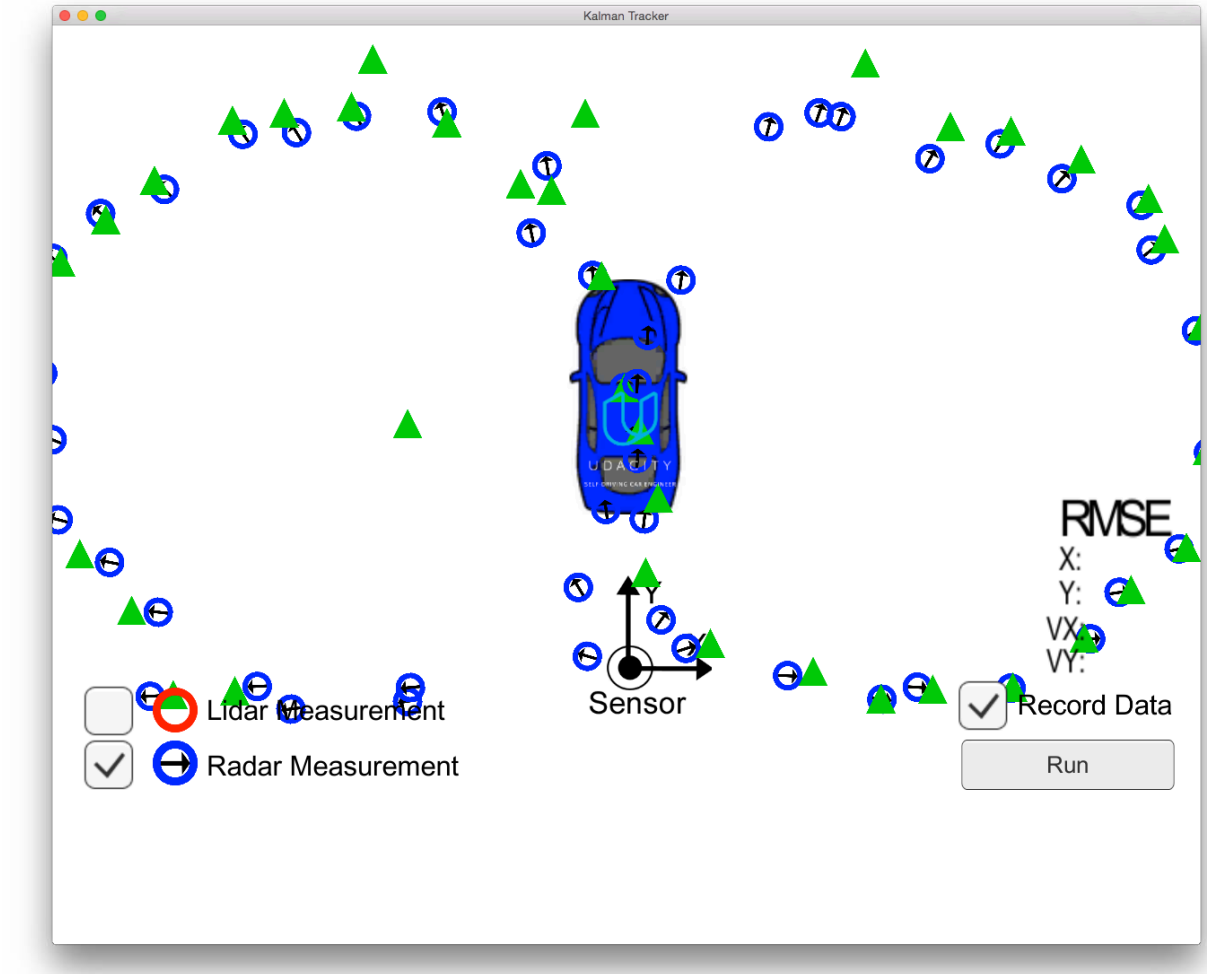


Figure 3: Radar only measurements

Results achieved with our radar measurements are less accurate than with Lidar data. In the Cartesian->polar conversion, I initially used atan C++ function. In the tracker, it only performed well in the last quadrant of the 8-loop. When switching to atan2 (as suggested) to keep the angle with –pi/2 and +pi/2, it provided stronger results. Though it may still have an issue with the last quadrant of the first loop.