Extended Kalman Filter Project Starter Code

Overview

This project implements Extended Kalman Filter Algorithm in C++ and tested on Simulator provided by Udacity. The simulator generates RADAR and LIDAR measurements with white noise for the position and velocity of the object. The project is to fusion both these measurements and predict the position using Extended Kalman filter algorithm. The communication between the simulator and the EKF is done using WebSocket using the uWebSockets implementation on the EKF side.

This project utilizes the starter code given by Udacity. Following are the outputs available as outcome of this project:

1. Vide0.mp4 – Contains recorded video of the output in the simulator on data set1

2. Video2.mp4 - Contains recorded video of the output in the simulator on data set2

2. log files – Log files contain output of cmake and make. Also various values and status are reported through cout in C++ and collected in a separate log file run – output data set1

3. rmse.csv – Contains the list of all the rmse values collected from the Simulator

Environment:

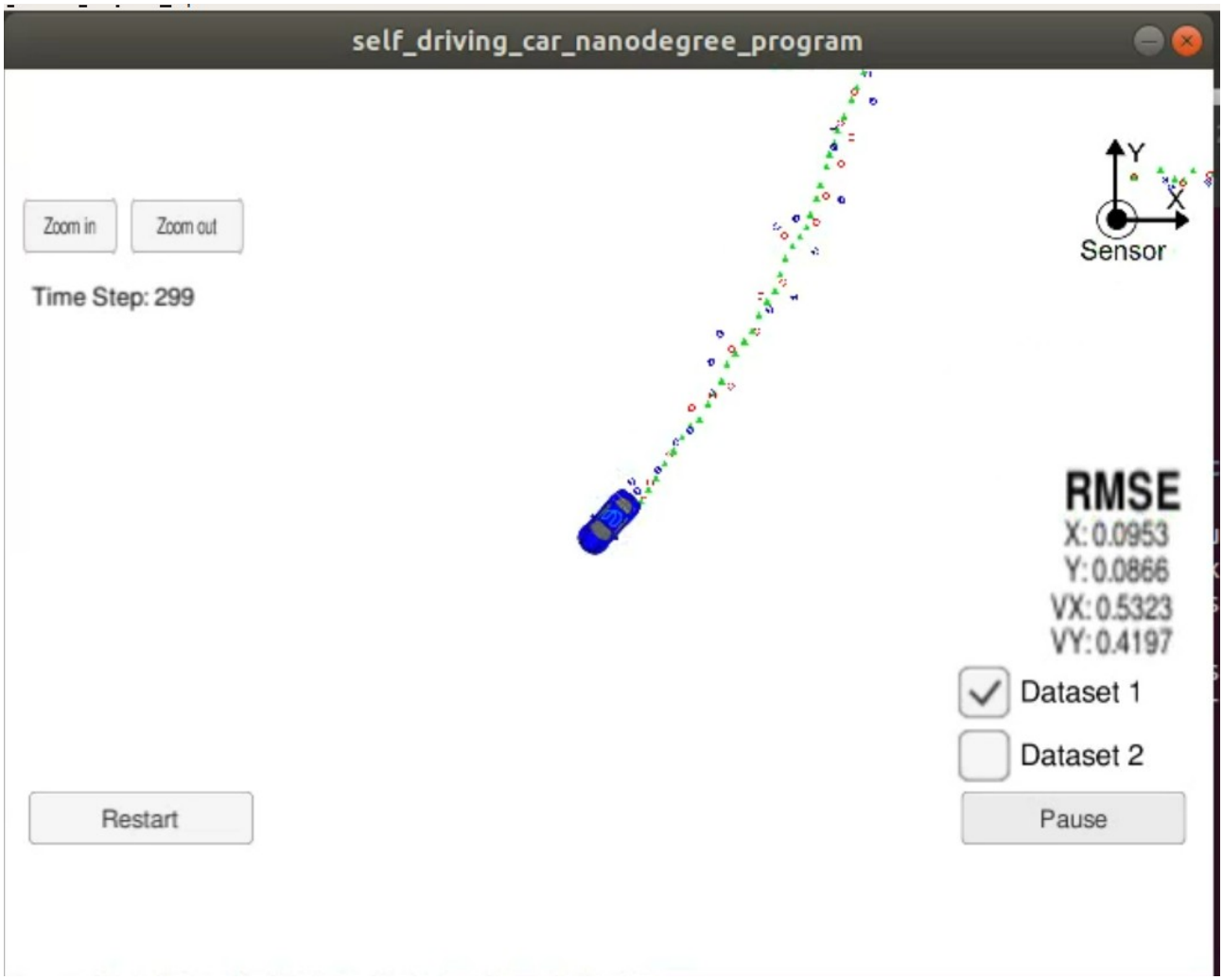
This project was done in Ubuntu 18 LTS and following are the components

- cmake >= 3.5
- make >= 4.1
- gcc/g++ >= 5.4

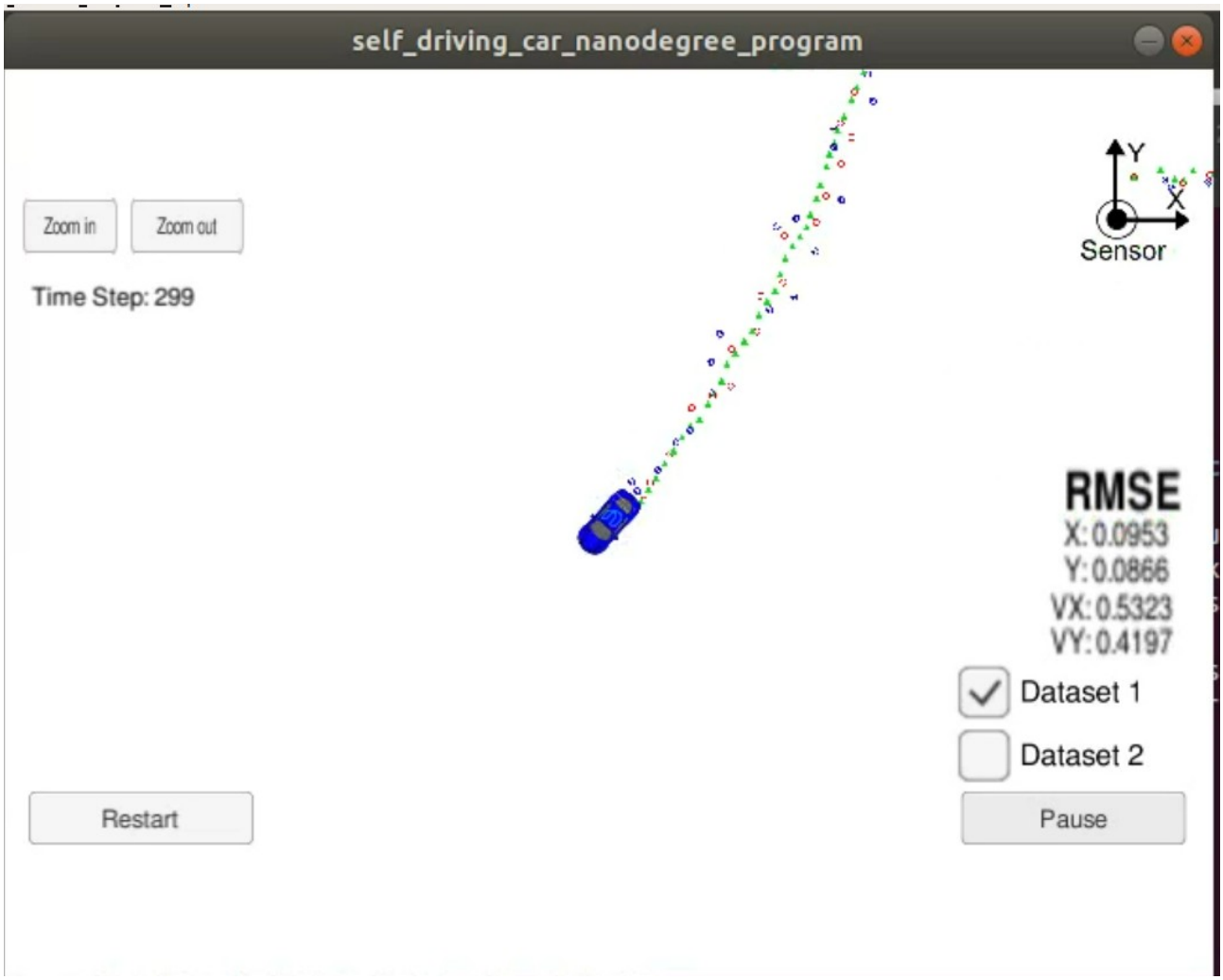
Ouput:

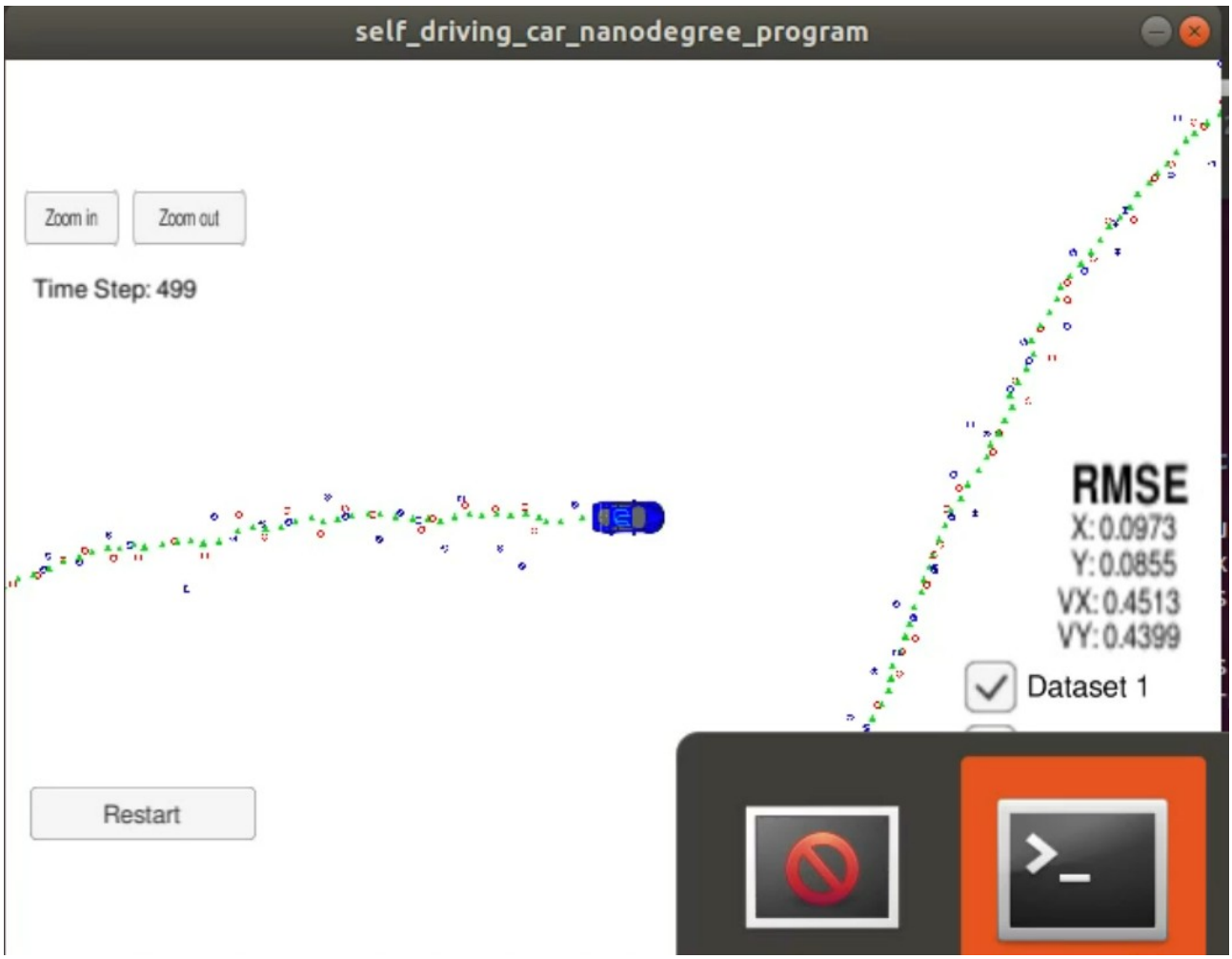
This project is the output from the Simulator on data set 1


self_driving_car_nanodegree_program
Zoom in
Zoom out
Time Step: 0
Y
X
Sensor
RMSE
X:
Y:
VX:
VY:
Dataset 1
Dataset 2
Restart
Start


self_driving_car_nanodegree_program
Zoom in
Zoom out
Time Step: 299
Y
X
Sensor
RMSE
X: 0.0953
Y: 0.0866
VX: 0.5323
VY: 0.4197
Dataset 1
Dataset 2
Restart
Pause

Overview


self_driving_car_nanodegree_program
Zoom in
Zoom out
Time Step: 299
Y
X
Sensor
RMSE
X: 0.0953
Y: 0.0866
VX: 0.5323
VY: 0.4197
Dataset 1
Dataset 2
Restart
Pause



RUBRICS:

This project satisfies all the rubric points

Compiling

Criteria	Meets Specifications
Your code should compile.	Code must compile without errors with `cmake` and `make`.

The code compiles properly without any error. Please refer to CMAKE-OUTPUT.LOG and make-output.log Below is the output:

CMAKE:

-- Configuring done

-- Generating done

-- Build files have been written to: /home/rameshbaboov/udacity/CarND-Extended-Kalman-Filter-project/build

MAKE:

[100%] Built target ExtendedKF

Accuracy

Criteria	Meets Specifications
px, py, vx, vy output coordinates must have an RMSE <= [.11, .11, 0.52, 0.52] when using the file: "obj_pose-laser-radar-synthetic-input.txt" which is the same data file the simulator uses for Dataset 1.	Your algorithm will be run against Dataset 1 in the simulator which is the same as "data/obj_pose-laser-radar-synthetic-input.txt" in the repository. We'll collect the positions that your algorithm outputs and compare them to ground truth data. Your px, py, vx, and vy RMSE should be less than or equal to the values [.11, .11, 0.52, 0.52].

The RMSE is within the limits. Please refer to the video or rmse.csv. RMSE.CSV was extracted from the output log run-output-dataset1.log. Though first few set of rmse are above limit, quickly the model converges the error to smaller no as shown below. Below is the last few rmse

{rmse_x:0.0978198 rmse_Y:0.0858336 rmse_vx:0.459799 rmse_vY:0.446953 },1

{rmse_x:0.0979508 rmse_Y:0.0860281 rmse_vx:0.459411 rmse_vY:0.4471 },1

{rmse_x:0.0979405 rmse_Y:0.0860024 rmse_vx:0.458931 rmse_vY:0.44682 },1

{rmse_x:0.0979423 rmse_Y:0.0860124 rmse_vx:0.458453 rmse_vY:0.44661 },1

{rmse_x:0.0979081 rmse_Y:0.0861413 rmse_vx:0.457978 rmse_vY:0.446556 },1

{rmse_x:0.0979434 rmse_Y:0.0863129 rmse_vx:0.457527 rmse_vY:0.446434 },1

{rmse_x:0.0978843 rmse_Y:0.0864319 rmse_vx:0.457057 rmse_vY:0.446202 },1

{rmse_x:0.0978572 rmse_Y:0.0863965 rmse_vx:0.45677 rmse_vY:0.445805 },1

{rmse_x:0.0979112 rmse_Y:0.0863112 rmse_vx:0.45652 rmse_vY:0.445461 },1

{rmse_x:0.0979758 rmse_Y:0.0862234 rmse_vx:0.456204 rmse_vY:0.44507 },1

{rmse_x:0.0979479 rmse_Y:0.0861395 rmse_vx:0.455769 rmse_vY:0.444718 },1

{rmse_x:0.0979074 rmse_Y:0.0860582 rmse_vx:0.455311 rmse_vY:0.444302 },1

{rmse_x:0.0978639 rmse_Y:0.0860325 rmse_vx:0.45485 rmse_vY:0.443986 },1

{rmse_x:0.0977787 rmse_Y:0.0859622 rmse_vx:0.454403 rmse_vY:0.443546 },1

{rmse_x:0.0976941 rmse_Y:0.0859251 rmse_vx:0.45395 rmse_vY:0.443142 },1

{rmse_x:0.0975994 rmse_Y:0.0858527 rmse_vx:0.45351 rmse_vY:0.442696 },1

{rmse_x:0.097509 rmse_Y:0.0858237 rmse_vx:0.453055 rmse_vY:0.442287 },1

{rmse_x:0.0974103 rmse_Y:0.0857403 rmse_vx:0.45262 rmse_vY:0.441844 },1

{rmse_x:0.0974975 rmse_Y:0.0857352 rmse_vx:0.452654 rmse_vY:0.441589 },1

{rmse_x:0.0976057 rmse_Y:0.0857164 rmse_vx:0.452619 rmse_vY:0.441242 },1

{rmse_x:0.0975115 rmse_Y:0.0856301 rmse_vx:0.452168 rmse_vY:0.440799 },1

{rmse_x:0.0974149 rmse_Y:0.0855442 rmse_vx:0.45172 rmse_vY:0.440368 },1

{rmse_x:0.0973178 rmse_Y:0.0854597 rmse_vx:0.451267 rmse_vY:0.439935 },1

Follows the Correct Algorithm

Rubric#1

Criteria	Meets Specifications
Your Sensor Fusion algorithm follows the general processing flow as taught in the preceding lessons.	While you may be creative with your implementation, there is a well-defined set of steps that must take place in order to successfully build a Kalman Filter. As such, your project should follow the algorithm as described in the preceding lesson.

The code implements the correct Kalman filter algorithm and in two files:

***Kalman_filter.cpp** -> Contains the key methods as given below:*

1. KalmanFilter::Predict() - to predict the state using previous estimate

2. KalmanFilter::Update – to update the state using the current measurement with the estimate (for Laser)

3. KalmanFilter::UpdateEKF – same as previous but used for Radar

Rubric#2

Your Kalman Filter algorithm handles the first measurements appropriately.	Your algorithm should use the first measurements to initialize the state vectors and covariance matrices.

*The first measurement is handled to initialize the state vector and covariance matrices. There are separate lines of codes to handle both Radar and Laser. Please refer to line no 59 to 119 in **FusionEKF.cpp***

Rubric#3

Your Kalman Filter algorithm first predicts then updates.	Upon receiving a measurement after the first, the algorithm should predict object position to the current timestep and then update the prediction using the new measurement.

The Kalman algorithm first predict the object position based on the timestep and then update the prediction using new measurements. Prediction is from line no 120 to 159 and update is from 160 to 201 in ***FusionEKF.cpp***

Rubric#4

Your Kalman Filter can handle radar and lidar measurements.	Your algorithm sets up the appropriate matrices given the type of measurement and calls the correct measurement function for a given sensor type.

There are separate codes to handle Radar and lidar and the codes works properly when Radar or Lidar is switched off by toggling comments on line no 132 to 135 and 103 to 105 in ***FusionEKF.cpp***

Rubric#5 - Code Efficiency

Criteria	Meets Specifications
	This is mostly a "code smell" test. Your algorithm does not need to sacrifice comprehension, stability, robustness or security for speed, however it should maintain good practice with respect to calculations.
Your algorithm should avoid unnecessary calculations.	Here are some things to avoid. This is not a complete list, but rather a few examples of inefficiencies. • Running the exact same calculation repeatedly when you can run it once, store the value and then reuse the value later. • Loops that run too many times. • Creating unnecessarily complex data structures when simpler structures work equivalently. • Unnecessary control flow checks.

The code adheres to all teh above criteria