UDACITY

PROJECT

Extended Kalman Filters

A part of the Self-Driving Car Engineer Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

Congratulations, your project meets all specifications!

I've enjoyed very much reviewing your project, you can be proud of the work you've done!

Keep learning and stay ***Udacious***! :D

Compiling

Code must compile without errors with `cmake` and `make`.

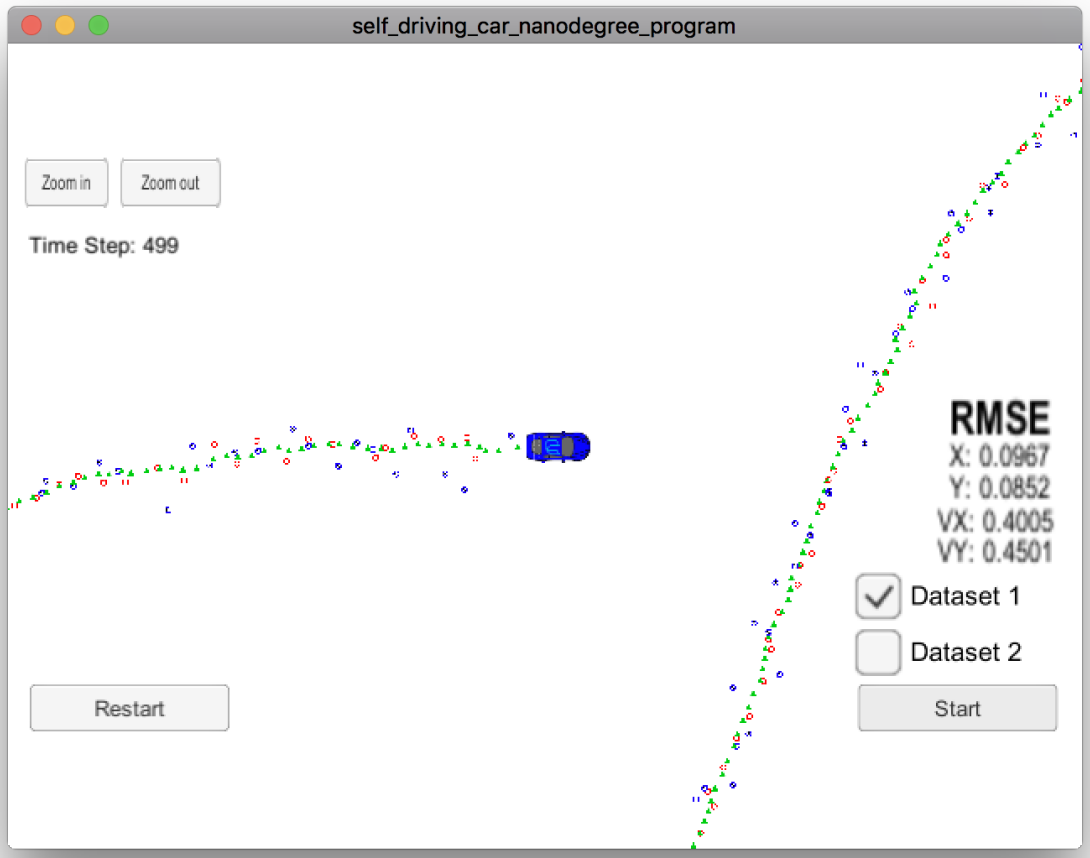
Given that we've made CMakeLists.txt as general as possible, it's recommended that you do not change it unless you can guarantee that your changes will still compile on any platform.

The code compiles without errors, great job here!

Accuracy

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].

After compiling the code, the accuracy test was passed successfully, great job!



Follows the Correct Algorithm

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 Kalman Filter core equations were properly implemented for both prediction and update procedures.

Your algorithm should use the first measurements to initialize the state vectors and covariance matrices.

The filter is properly initialized using the first measurements, from either radar or lidar. Well done!

Improvement Suggestion

I've noticed you've only initialized the position for the radar measurement:

```
91
92      if (measurement_pack.sensor_type_ == MeasurementPackage::RADAR) {
93        /**
94        Convert radar from polar to cartesian coordinates and initialize state.
95        */
96        float ro = measurement_pack.raw_measurements_[0];
97        float theta = measurement_pack.raw_measurements_[1];
98  //      float ro_dot = measurement_pack.raw_measurements_[2];
99
100       ekf_.x_[0] = ro * cos(theta);
101       ekf_.x_[1] = ro * sin(theta);
102     }
```

Please consider also initializing the velocities when a radar measurement is initiated, e.g:

```
double rho = measurement_pack.raw_measurements_[0];
double phi = measurement_pack.raw_measurements_[1];
double rho_dot = measurement_pack.raw_measurements_[2];
double x= rho * cos(phi);
double y= rho * sin(phi);
double vx = rho_dot * cos(phi);
double vy = rho_dot * sin(phi);
ekf_.x_ << x, y, vx, vy;
```

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.

Prediction is performed at the actual time step and then updated with the new measurement. Great job!

Improvement Suggestion

Please note that when delta time gets a near zero value could cause some arithmetic inconsistency in the calculations, to solve that possible issues it's recommended including a test to prevent those situations, e.g:

```
if ( dt > 0.001 )
{
```

```
    <normal filter operation>
}
else
{
    <do nothing>
}
```

Your algorithm sets up the appropriate matrices given the type of measurement and calls the correct measurement function for a given sensor type.

The algorithm sets up the appropriate matrices and measurement functions depending on the type of sensor. Great job here as well!

Code Efficiency

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.

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.**

In a general overview, the code is good and repetitive calculations are implemented in an efficient way, avoiding unnecessary repetitions. Good job!

DOWNLOAD PROJECT

RETURN TO PATH

Student FAQ