UDACITY

Back to Self-Driving Car Engineer

Extended Kalman Filters

REVIEW

CODE REVIEW 2

HISTORY

Meets Specifications

Nice work in applying what you've learned, building an Extended Kalman Filter in C++, and testing it against the sample data!

Just in case you haven't read it yet, this blog does a very good job of explaining the math behind kalman filter, particularly concerning the derivation of Kalman gain K.

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.

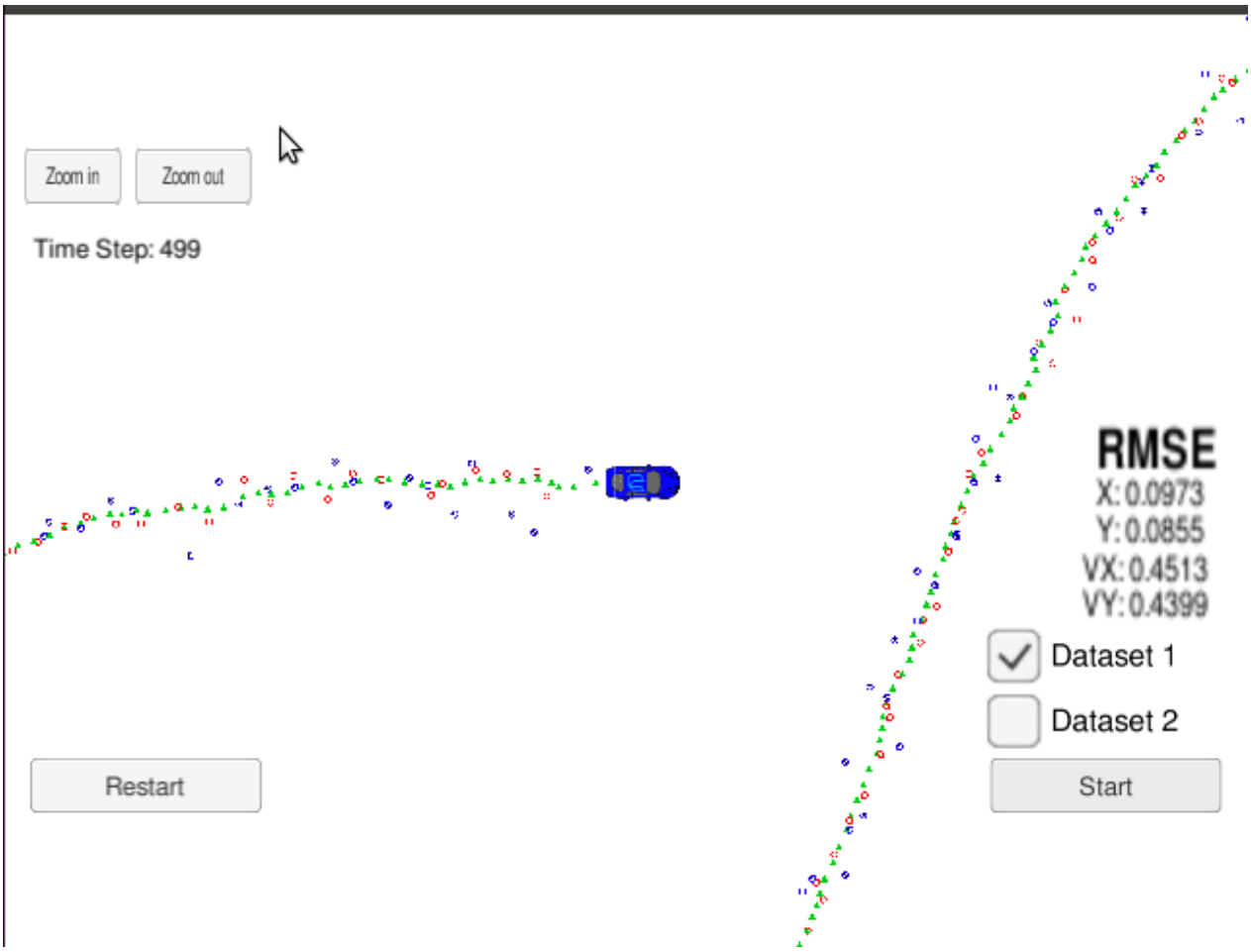
Nice job. Your project compiles successfully with cmake and make.

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].

When running your algorithm against "obj_pose-laser-radar-synthetic-input.txt", the RMSE obtained is as below, which is less than the required values [.11, .11, 0.52, .52]



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.

In your implementation, all the steps involved in building a Kalman Filter is properly followed: state vector and covariance matrices initialization, predict and update step.

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

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.

Well done. predict and update step are correctly applied after new receiving measurements.

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

During update step, laser and radar measurements are handled differently since they have different measurement function. Well done.

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.**

DOWNLOAD PROJECT

2 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review