UDACITY

‹ Back to Self-Driving Car Engineer

Unscented Kalman Filters

REVIEW

CODE REVIEW 11

HISTORY

Meets Specifications

Future SDC engineer,

You have shown a great understanding of various complex concepts of UKF in this project. The work is well documented and articulated one. 👌 I enjoyed reviewing your work!

For further reading and to dive deeper into the concept of UKF and basic filtering I suggest these materials which I enjoyed reading and watching:

If you need math behind the Kalman Filter then this youtube series is awesome: THE KALMAN FILTER

This interactive tutorial is very helpful to visualize and have a better grasp of workings in this area: The Extended Kalman Filter: An Interactive Tutorial for Non-Experts

If you are looking for what other filters might be apart from EKF and UKF then you can check this. It's little math heavy and based on Matlab: Optimal Filtering with Kalman Filters and Smoothers

A nice explanation of choosing between EKF and UKF and then it's implementation guide. It's in two parts: Understanding Nonlinear Kalman Filters Part I

And Part II

A Python library for Kalman filtering which you can check out: pykalman

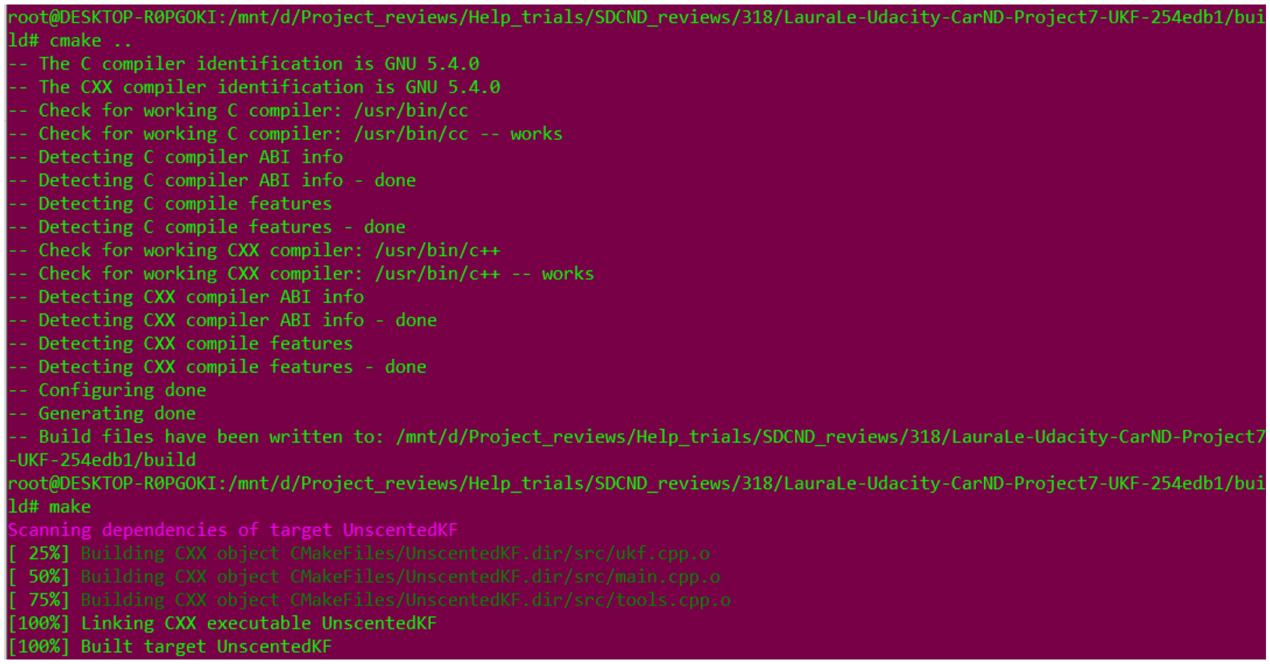
The UKF paper: The Unscented Kalman Filter for Nonlinear Estimation

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.

Awesome Job in this section!! The code compiled without errors with cmake and make when running from the build directory. Great keep it up! 👍

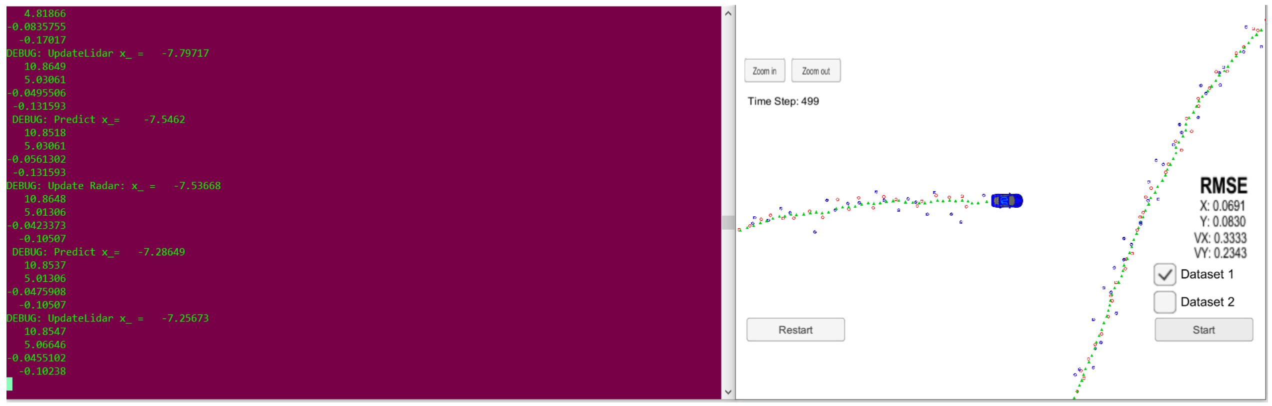


```
root@DESKTOP-R0PGOKI:/mnt/d/Project_reviews/Help_trials/SDCND_reviews/318/LauraLe-Udacity-CarND-Project7-UKF-254edb1/bui
ld# cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/Project_reviews/Help_trials/SDCND_reviews/318/LauraLe-Udacity-CarND-Project7
-UKF-254edb1/build
root@DESKTOP-R0PGOKI:/mnt/d/Project_reviews/Help_trials/SDCND_reviews/318/LauraLe-Udacity-CarND-Project7-UKF-254edb1/bui
ld# make
Scanning dependencies of target UnscentedKF
[ 25%] Building CXX object CMakeFiles/UnscentedKF.dir/src/ukf.cpp.o
[ 50%] Building CXX object CMakeFiles/UnscentedKF.dir/src/main.cpp.o
[ 75%] Building CXX object CMakeFiles/UnscentedKF.dir/src/tools.cpp.o
[100%] Linking CXX executable UnscentedKF
[100%] Built target UnscentedKF
```

Accuracy

Your algorithm will be run against "obj_pose-laser-radar-synthetic-input.txt". 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 [.09, .10, .40, .30].

Great Performance in this section. As expected your algorithm gives reasonably low RMSE values. In fact, your RMSE values are lower than the requirements!! 👏 Impressive!



```
   4.81866
-0.0835755
  -0.17017
DEBUG: UpdateLidar x_ =   -7.79717
   10.8649
   5.03061
-0.0495506
 -0.131593
 DEBUG: Predict x_=    -7.5462
   10.8518
   5.03061
-0.0561302
 -0.131593
DEBUG: Update Radar: x_ =   -7.53668
   10.8648
   5.01306
-0.0423373
  -0.10507
 DEBUG: Predict x_=   -7.28649
   10.8537
   5.01306
-0.0475908
  -0.10507
DEBUG: UpdateLidar x_ =   -7.25673
   10.8547
   5.06646
-0.0455102
  -0.10238
```

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.

Right steps were taken in this section following the algorithm as described in the preceding lesson. Nice work in the `ukf.cpp' . You have followed all the well-defined set of steps of UKF. Excellent job!! 🎉

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

A great implementation of first measurements to initialize the state vectors and covariance matrices. An awesome job here ⭐ ⭐

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.

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

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.

An outstanding implementation of the project. The code was well optimized and used simple structures and no unnecessary control flow checks.

DOWNLOAD PROJECT

11 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review