**Meets Specifications**

Dear Excellent Student,

Well done!  
This implementation possesses efforts that are commendable indeed. I enjoyed the way the algorithm was implemented as well as the precautions that were taken to ensure that this submission runs with success. I would love to endorse the determination and enthusiasm employed in this project urging that the spirit invested in this work be carried throughout learning with us here at udacity. Stay focused as always and Happy learning. :udacious:

**More helpful resource in this domain**

* [Applying the unscented Kalman filter for nonlinear state estimation](http://www.sciencedirect.com/science/article/pii/S0959152407001655)
* [Uncented Kalman Filter for Dummies](https://robotics.stackexchange.com/questions/9233/uncented-kalman-filter-for-dummies)
* [Learning the Unscented Kalman Filter](https://www.mathworks.com/matlabcentral/fileexchange/18217-learning-the-unscented-kalman-filter?requestedDomain=www.mathworks.com)
* [Robot Mapping Extended Kalman Filter](http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam03-ekf.pdf)
* [Introduction to Robotics #10: Extended Kalman Filter](http://correll.cs.colorado.edu/?p=1464)
* [Marrying Kalman Filtering & Machine Learning](https://www.datasciencecentral.com/profiles/blogs/marrying-kalman-filtering-machine-learning)

**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 codes in this submission could be compiled successfully with just cmake and make. Nice kickoff!

**Extra Material**

To know more about cmake and make, visit these links:

* [Cmake FAQS](https://cmake.org/Wiki/CMake_FAQ).
* [Using make and writing Makefiles](https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html).
* [Youtube set of tutorials on using make and writing Makefile](https://www.youtube.com/watch?v=aw9wHbFTnAQ).
* [MakeFiles](https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html)

**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].**

The results are indeed outstanding. The RMSE values are well within the required range. This only goes to show that the algorithm used was carefully implemented and all necessary caution was administered. This is really commendable.

[Graphical user interface

Description automatically generated](https://udacity-reviews-uploads.s3.us-west-2.amazonaws.com/_attachments/58227/1607063703/Screenshot_from_2020-12-04_07-34-49.png)

**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.**

This submission adequately follows the predefined steps set in the lesson. The FusionEKF.cpp does a good job to implement all the computational steps required to achieve a successful Extended Kalman Filter with clearly written code. This I must say was nicely done!

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

This implemented algorithm with success uses the first measurements to initialize the state and covariance matrices as found in the FusionEKF.cpp. Well done!

**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.**

Good job in this section, the algorithm adequately predicts the object position to the correct time step and then updates the measurements using the prediction and Update functions of the code in FusionEKF.cpp file.

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

A wonderful job is done in this implementation as the algorithm sets up the correct matrices based on the type of measurement (radar or lidar) and accurately calls the correct function for that type. Nicely done udacian!

**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.**

This Implementation completely stays clear of unnecessary complex data structures, avoids loops that run repeatedly, avoids running exact calculations severally and also unnecessary control flow checks are avoided.

**Extra Material**

Here are a few tips for improving on code efficiency and optimization:

* The most efficient types:
  + When defining an object to store an integer number, use the int or the unsigned int type, except when a longer type is needed
  + When defining an object to store a character, use the char type, except when the wchar\_t type is needed
  + When defining an object to store a floating point number, use the double type, except when the long double type is needed.
  + If the resulting aggregate object is of medium or large size, replace each integer type with the smallest integer type that is long enough to contain it (but without using [bit-fields](https://en.cppreference.com/w/cpp/language/bit_field)) and replace the [floating point types](https://www.learncpp.com/cpp-tutorial/floating-point-numbers/) with the float type, except when greater [precision](https://stackoverflow.com/questions/55767385/precision-of-floating-point-data-types-in-c) is needed.
* This [article](https://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/) will give some high-level ideas on how to improve the speed of your program. This includes the printf and scanf Vs cout and cin, Using Operators, if Condition Optimization, Problems with Functions, Optimizing Loops, Data Structure Optimization and a lot more.
* [Optimizing C++/Writing efficient code/Performance improving features.](https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Writing_efficient_code/Performance_improving_features)
* [Efficient C++ Performance Programming Techniques](http://www.whigg.ac.cn/resource/program/CPP/201010/P020101023562491092566.pdf)
* [10 Tips for C and C++ Performance Improvement Code Optimization](http://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/)