Extended Kalman Filters

1. **Purpose :**

   The purpose of project is determine a bicycle's position using Kalman filter by referring to the inputs provided by RADAR and LiDAR. The RADAR and LiDAR measurements are generated by a simulator.

2. **Project Inputs :**

   a. LiDAR and RADAR measurements generated by simulator.

   b. Data file used by simulator to generate LiDAR and RADAR measurements - *obj_pose-laser-radar-synthetic-input.txt*

   c. Files in the Source (src) folder includes –

      i. Main.cpp
      ii. FusionEKF.cpp
      iii. Kalman_filter.cpp
      iv. Tools.cpp

3. **Project Outputs :**

   a. To satisfy project rubric following artifacts are expected –

      i. Build Folder
      ii. ExtendedKF run file

   b. To satisfy project rubric following specifications to be satisfied –

      i. Code must compile without errors with cmake and make
      ii. The output px, py, vx, and vy RMSE should be less than or equal to the values [.11, .11, 0.52, 0.52].
      iii. Sensor Fusion algorithm follows the general processing flow as taught in the preceding lessons.
      iv. Kalman Filter algorithm handles the first measurements appropriately.
      v. Kalman Filter algorithm first predicts then updates.

vi. Kalman Filter can handle RADAR and LiDAR measurements.
vii. Algorithm should avoid unnecessary calculations.

4. Introduction to Kalman Filters :

In statistics and control theory, Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. The filter is named after Rudolf E. Kálmán, one of the primary developers of its theory.

The algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

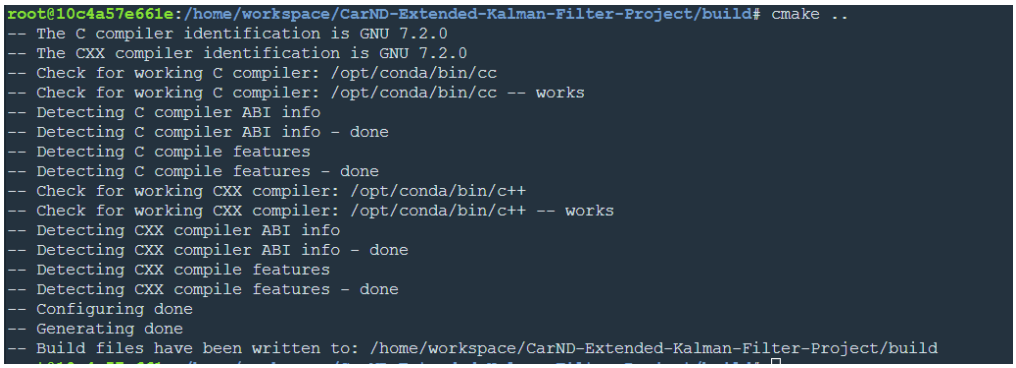
(Reference: https://en.wikipedia.org/wiki/Kalman_filter)

5. Project Execution :

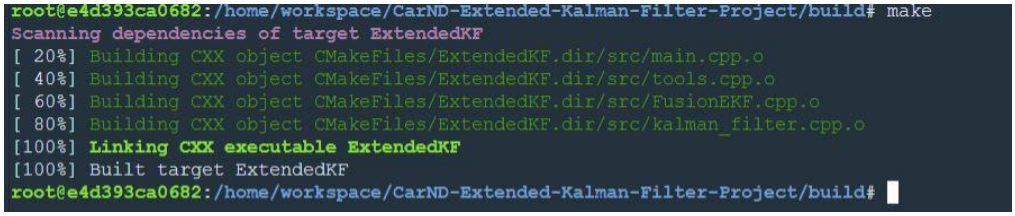
a. *Code Compilation* :

Following are the snapshots from code compilation –

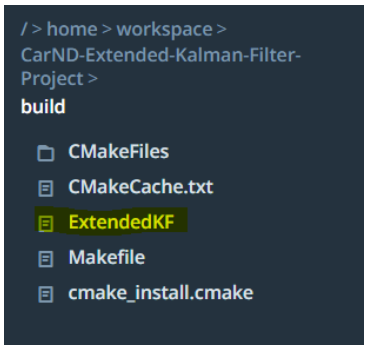
Execution of *cmake* command -



Execution of *make* command -



ExtendedKF file in build folder -



For me the execution of cmake and make commands was not as simple as it looks. I got success after multiple attempts, faced few failures as shown below before getting the final compiled files –

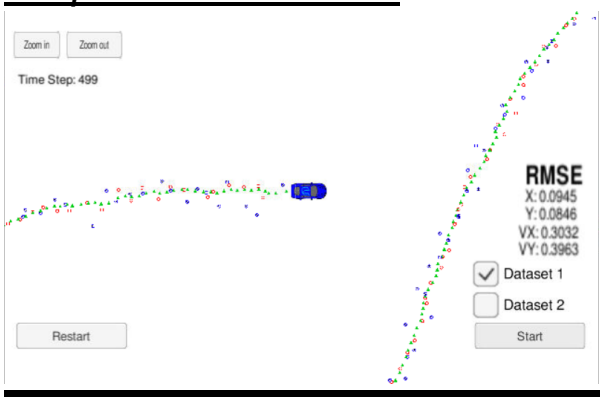
```
[ 20%] Linking CXX executable ExtendedKF
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_use_certificate_chain_file@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_want@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_write@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_set_fd@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_set_connect_state@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_ctrl@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_free@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_library_init@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_ctrl@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_set_default_passwd_cb_userdata@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSLv23_server_method@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_pending@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSLv23_client_method@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SHA1@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_new@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_use_PrivateKey_file@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_set_accept_state@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_free@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_get_error@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_CTX_set_default_passwd_cb@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_new@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_shutdown@OPENSSL_1.0.0'
/usr/lib/libuWS.so: undefined reference to `SSL_read@OPENSSL_1.0.0'
collect2: error: ld returned 1 exit status
CMakeFiles/ExtendedKF.dir/build.make:172: recipe for target 'ExtendedKF' failed
make[2]: *** [ExtendedKF] Error 1
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/ExtendedKF.dir/all' failed
make[1]: *** [CMakeFiles/ExtendedKF.dir/all] Error 2
```

```
root@fee9adf73a16:/home/workspace/CarND-Extended-Kalman-Filter-Project/build# make
[ 20%] Linking CXX executable ExtendedKF
Error running link command: No such file or directory
CMakeFiles/ExtendedKF.dir/build.make:172: recipe for target 'ExtendedKF' failed
make[2]: *** [ExtendedKF] Error 2
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/ExtendedKF.dir/all' failed
make[1]: *** [CMakeFiles/ExtendedKF.dir/all] Error 2
Makefile:83: recipe for target 'all' failed
make: *** [all] Error 2
root@fee9adf73a16:/home/workspace/CarND-Extended-Kalman-Filter-Project/build#
```

b. *Output Accuracy* *:*

Following are the snapshots of the Simulator output –

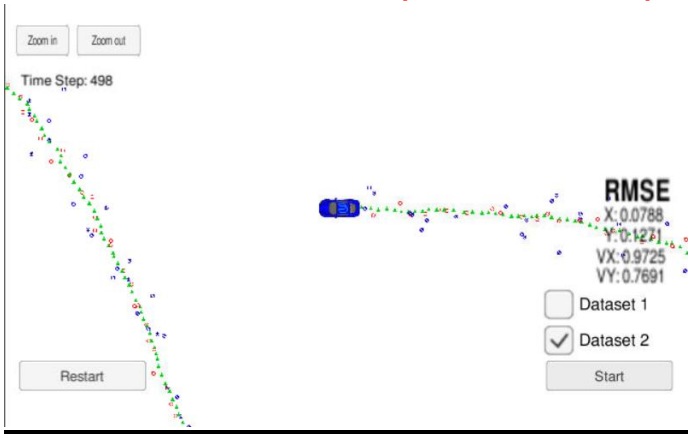
Output for Dataset 1:



State Vector	RMSE
px	0.0945
py	0.0846
vx	0.3032
vy	0.3963

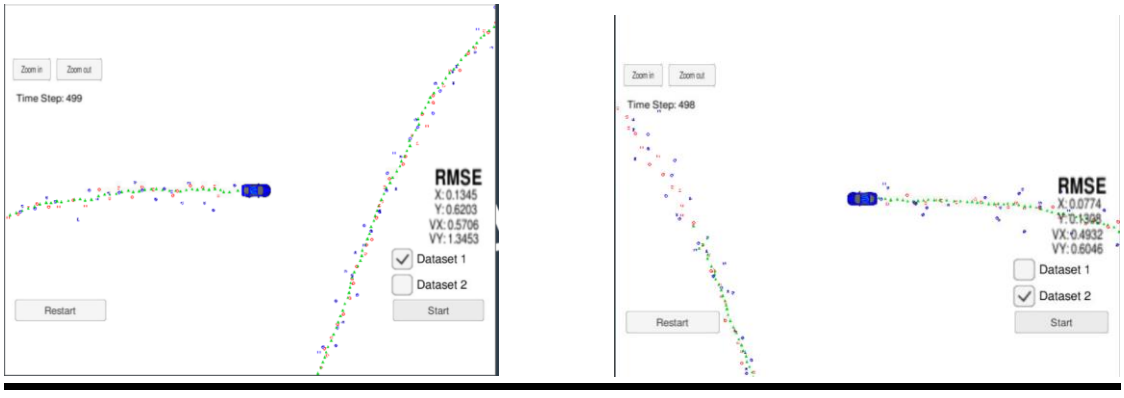
Output for Dataset 2:

For Dataset2, the output is not as per expectation -



State Vector	RMSE
px	0.0788
py	0.1271
vx	0.9725
vy	0.7691

Also tried with Dataset-1 and Dataset-2 with modifying the process noise value, but without any success –



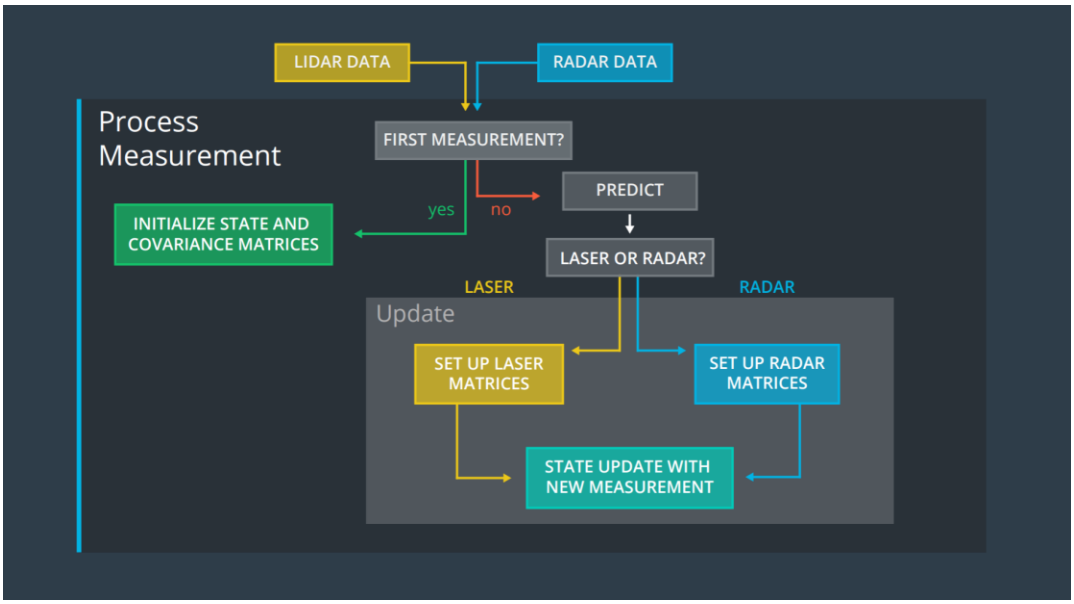
c. *Algorithm Implementation* :

i. The Sensor Fusion algorithm is followed as per the provided lessons in Udacity Self Driving Car Engineer course.

ii. The first measurements are used to initialize the state vectors and covariance matrices.

iii. The algorithm predicts the object position to the current time step and then updates the prediction using the new measurements.

iv. Algorithm is implemented to setup the appropriate matrices given the type of measurement and calls the correct measurement function for a given sensor type.



The Kalman Filter algorithm goes through the following steps:

- **First measurement -** the filter receives initial measurements of the bicycle's position relative to the car. These measurements come from a RADAR or LiDAR sensor simulator.
- **Initialize state and covariance matrices -** the filter initializes the bicycle's position based on the first measurement. Then the car receives another sensor measurement after a time period Δt.

- **Predict -** the algorithm predicts where the bicycle will be after time Δt. One basic way to predict the bicycle location after Δt is to assume the bicycle's velocity is constant; thus the bicycle will have moved velocity * Δt. In the extended Kalman filter lesson, we it is assumed that the velocity is constant.
- **Update -** the filter compares the "predicted" location with what the sensor measurement says. The predicted location and the measured location are combined to give an updated location. The Kalman filter puts more weight on either the predicted location or the measured location depending on the uncertainty of each value. Then the car receives another sensor measurement after a time period Δt. The algorithm then does another predict and update step.

  Following files were modified for implementation in the Source (src) folder includes –
  - FusionEKF.cpp
  - Kalman_filter.cpp
  - Tools.cpp

d. *<u>Code Efficiency</u> :*

To ensure the code efficiency, following points avoided as far as possible –

i. Running the exact same calculation repeatedly when you can run it once, store the value and then reuse the value later.
ii. Loops that run too many times.
iii. Creating unnecessarily complex data structures when simpler structures work equivalently.
iv. Unnecessary control flow checks.