The Tasks

Once again, you will be building up your estimator in pieces. At each step, there will be a set of success criteria that will be displayed both in the plots and in the terminal output to help you along the way.

Project outline:

Step 1: Sensor Noise

For the controls project, the simulator was working with a perfect set of sensors, meaning none of the sensors had any noise. The first step to adding additional realism to the problem, and developing an estimator, is adding noise to the quad's sensors. For the first step, you will collect some simulated noisy sensor data and estimate the standard deviation of the quad's sensor.

1. Run the simulator in the same way as you have before

2. Choose scenario `06_NoisySensors`. In this simulation, the interest is to record some sensor data on a static quad, so you will not see the quad move. You will see two plots at the bottom, one for GPS X position and one for The accelerometer's x measurement. The dashed lines are a visualization of a single standard deviation from 0 for each signal. The standard deviations are initially set to arbitrary values (after processing the data in the next step, you will be adjusting these values). If they were set correctly, we should see ~68% of the measurement points fall into the +/- 1 sigma bound. When you run this scenario, the graphs you see will be recorded to the following csv files with headers: `config/log/Graph1.txt` (GPS X data) and `config/log/Graph2.txt` (Accelerometer X data).

3. Process the logged files to figure out the standard deviation of the the GPS X signal and the IMU Accelerometer X signal.

4. Plug in your result into the top of `config/6_Sensornoise.txt`. Specially, set the values for `MeasuredStdDev_GPSPosXY` and `MeasuredStdDev_AccelXY` to be the values you have calculated.

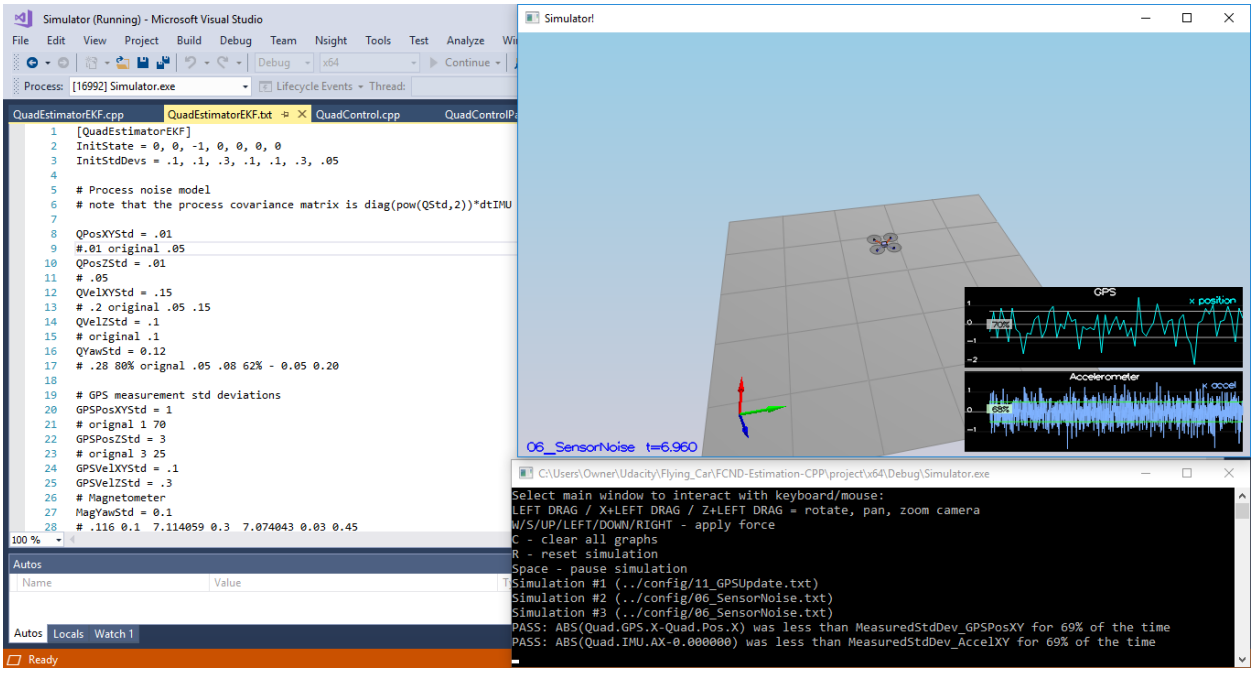
5. Run the simulator. If your values are correct, the dashed lines in the simulation will eventually turn green, indicating you're capturing approx 68% of the respective measurements (which is what we expect within +/- 1 sigma bound for a Gaussian noise model)

Success criteria: *Your standard deviations should accurately capture the value of approximately 68% of the respective measurements.*

NOTE: Your answer should match the settings in `SimulatedSensors.txt`, where you can also grab the simulated noise parameters for all the other sensors.

Calculated by finding the standard deviation of calculus using a calculator:

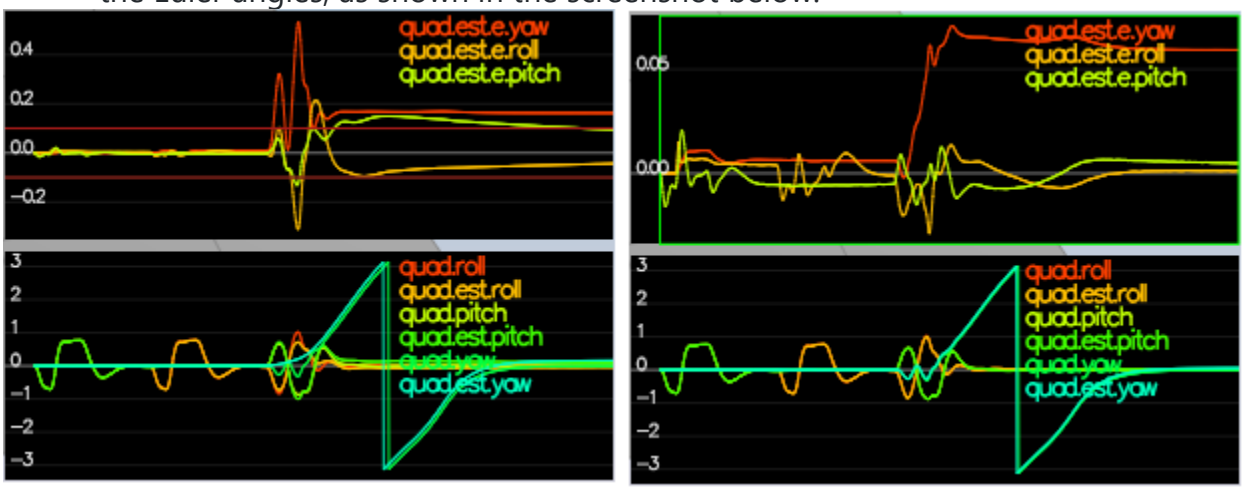
```
### STUDENT SECTION

MeasuredStdDev_GPSPosXY = 0.7 rounded
# 0.6569747495144732 0.7
MeasuredStdDev_AccelXY =  0.5 rounded
# 0.4909264198233001 0.5

### END STUDENT SECTION
```



Step 2: Attitude Estimation

Now let's look at the first step to our state estimation: including information from our IMU. In this step, you will be improving the complementary filter-type attitude filter with a better rate gyro attitude integration scheme.

1. Run scenario `07_AttitudeEstimation`. For this simulation, the only sensor used is the IMU and noise levels are set to 0 (see `config/07_AttitudeEstimation.txt` for all the settings for this simulation). There are two plots visible in this simulation.
    - The top graph is showing errors in each of the estimated Euler angles.
    - The bottom shows the true Euler angles and the estimates. Observe that there's quite a bit of error in attitude estimation.
2. In `QuadEstimatorEKF.cpp`, you will see the function `UpdateFromIMU()` contains a complementary filter-type attitude filter. To reduce the errors in the estimated attitude (Euler Angles), implement a better rate gyro attitude integration scheme. You should be able to reduce the attitude errors to get within 0.1 rad for each of the Euler angles, as shown in the screenshot below.



In the screenshot above the attitude estimation using linear scheme (left) and using the improved nonlinear scheme (right). Note that Y axis on error is much greater on left.

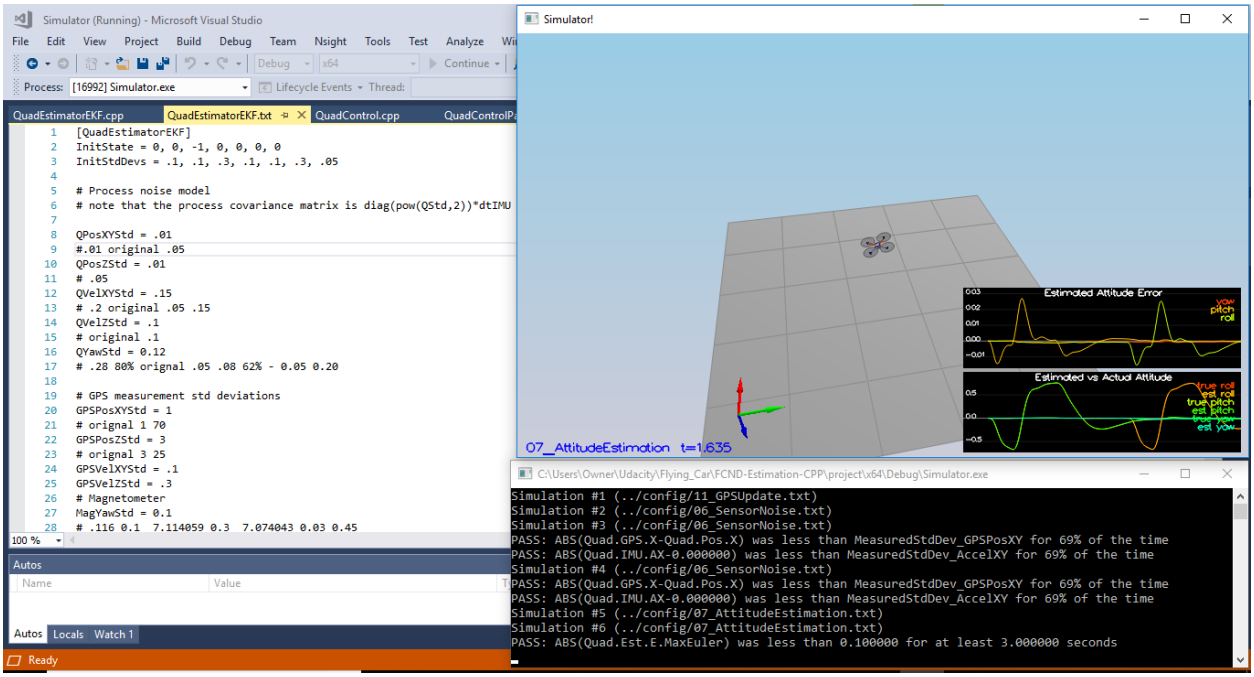
Success criteria: *Your attitude estimator needs to get within 0.1 rad for each of the Euler angles for at least 3 seconds.*

Hint: see section 7.1.2 of Estimation for Quadrotors for a refresher on a good non-linear complimentary filter for attitude using quaternions.

See the video of 07_AttitudeEstimation

Here is the code used to solve this part of the program:

```
Quaternion<float> qt = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, ekfState(6));
Quaternion<float> dq;
dq = dq.IntegrateBodyRate(gyro, dtIMU); // non-static function = durn

Quaternion<float> qt_bar = dq * qt;

float predictedPitch = qt_bar.Pitch();
float predictedRoll = qt_bar.Roll();
ekfState(6) = qt_bar.Yaw();

// normalize yaw to -pi .. pi
if (ekfState(6) > F_PI) ekfState(6) -= 2.f*F_PI;
if (ekfState(6) < -F_PI) ekfState(6) += 2.f*F_PI;
```



Step 3: Prediction Step

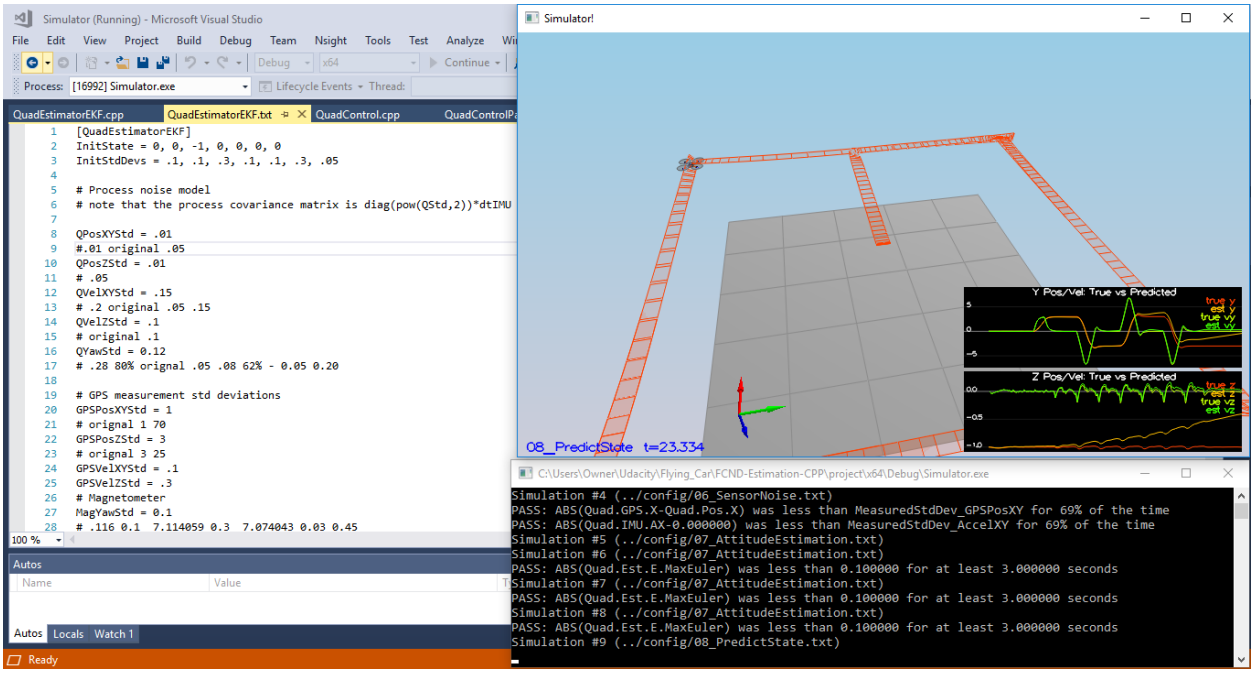
In this next step you will be implementing the prediction step of your filter.

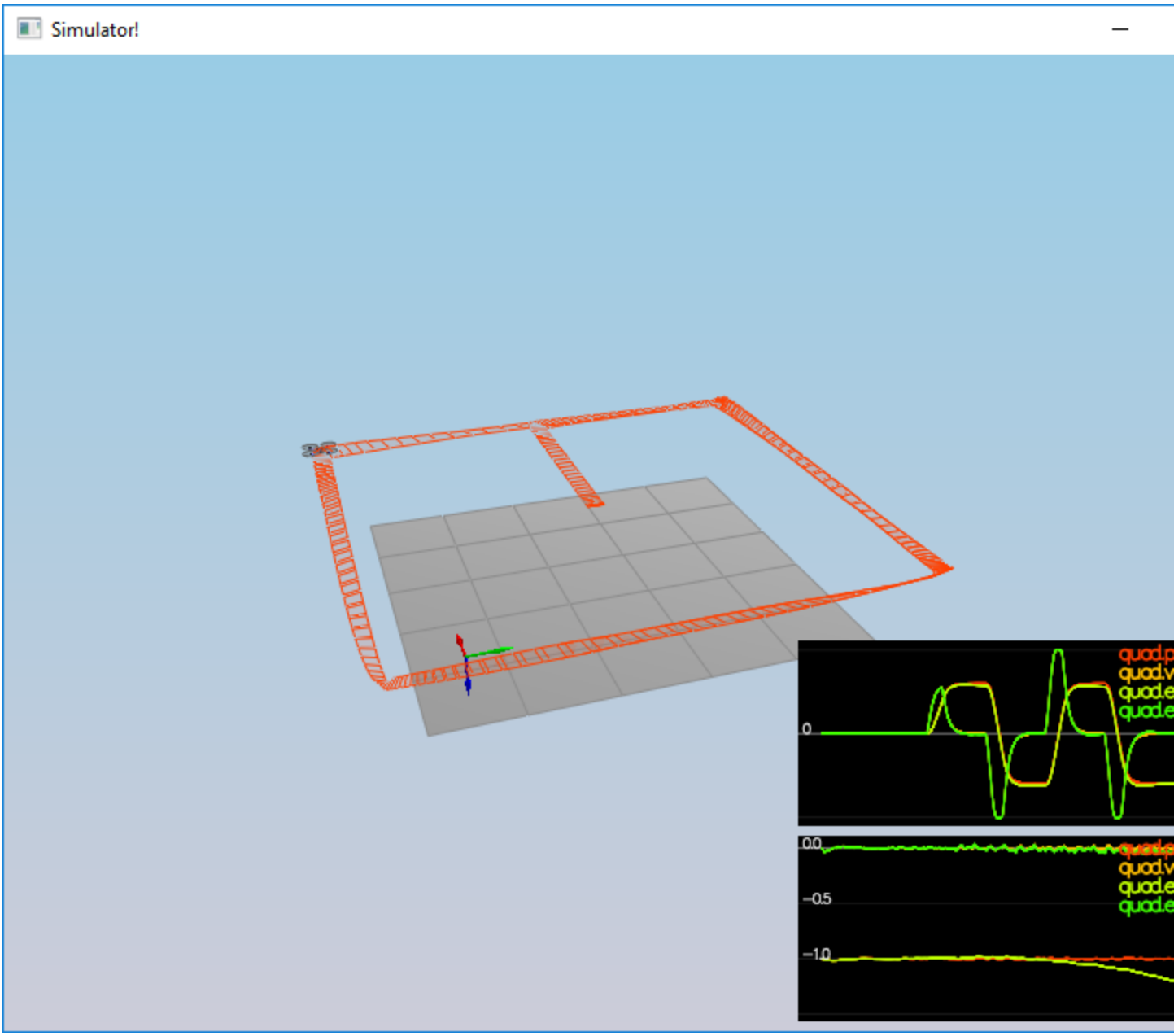
1. Run scenario `08_PredictState`. This scenario is configured to use a perfect IMU (only an IMU). Due to the sensitivity of double-integration to attitude errors, we've made the accelerometer update very insignificant (`QuadEstimatorEKF.attitudeTau = 100`). The plots on this simulation show element of your estimated state and that of the true state. At the moment you should see that your estimated state does not follow the true state.
2. In `QuadEstimatorEKF.cpp`, implement the state prediction step in the `PredictState()` functon. If you do it correctly, when you run

scenario `08_PredictState` you should see the estimator state track the actual state, with only reasonably slow drift, as shown in the figure below:

See video 08_PredictState

```
///////////////////////////////// BEGIN STUDENT CODE ////////////////////////////

//From Lesson 17 - Dead Reckoning Exercise

predictedState(0) += curState(3) * dt; // x = dx * dt
predictedState(1) += curState(4) * dt; // y = dy * dt
predictedState(2) += curState(5) * dt; // z = dz * dt

V3F rot = attitude.Rotate_BtoI(accel);
predictedState(3) += rot.x * dt;
predictedState(4) += rot.y * dt;
predictedState(5) += (rot.z - CONST_GRAVITY) * dt;


///////////////////////////////// END STUDENT CODE ////////////////////////////
```



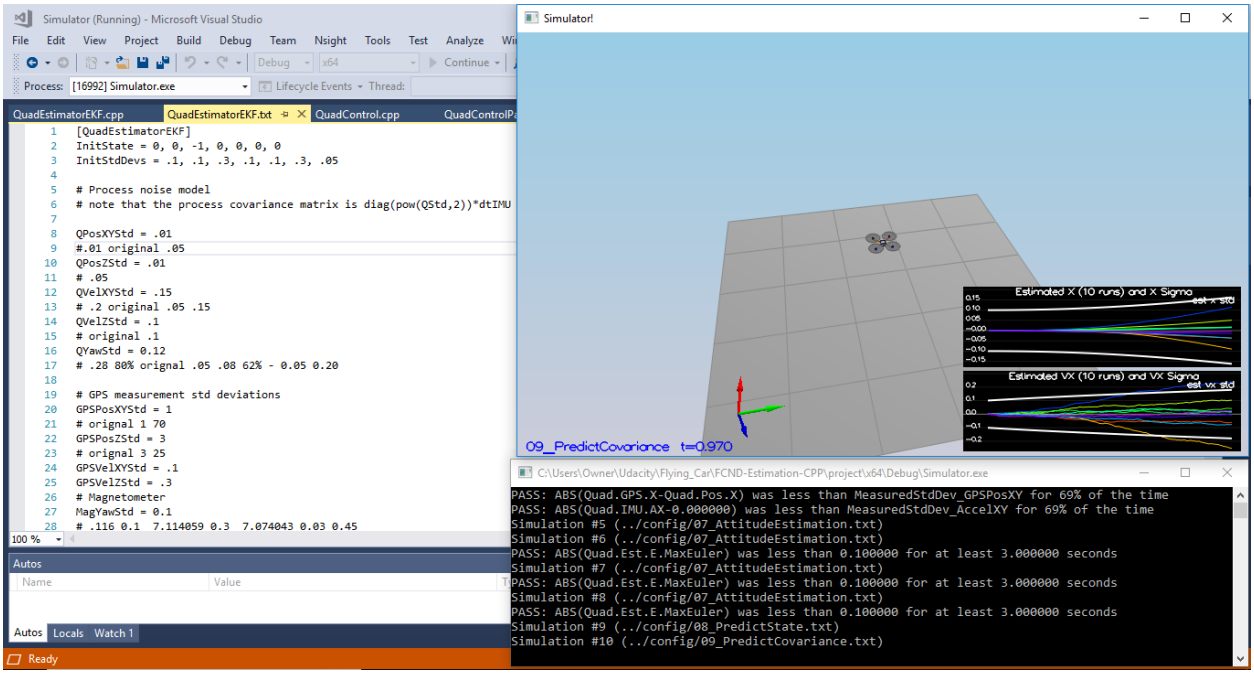


3. Now let's introduce a realistic IMU, one with noise. Run scenario `09_PredictionCov`. You will see a small fleet of quadcopter all using your prediction code to integrate forward. You will see two plots:
    - The top graph shows 10 (prediction-only) position X estimates
    - The bottom graph shows 10 (prediction-only) velocity estimates You will notice however that the estimated covariance (white bounds) currently do not capture the growing errors.
4. In `QuadEstimatorEKF.cpp`, calculate the partial derivative of the body-to-global rotation matrix in the function `GetRbgPrime()`. Once you have that function

implement, implement the rest of the prediction step (predict the state covariance forward) in `Predict()`.

See video 09_PredictConvarance

```
/////////////////////////////// BEGIN STUDENT CODE ////////////////////////////

// From "Estimation for Quadrotors" paper ( Eq. 52 )

RbgPrime(0, 0) = -cos(pitch)*sin(yaw);
RbgPrime(0, 1) = -sin(roll)*sin(pitch)*sin(yaw) - cos(roll)*cos(yaw);
RbgPrime(0, 2) = -cos(roll)*sin(pitch)*sin(yaw) + sin(roll)*cos(yaw);

RbgPrime(1, 0) = cos(pitch)*cos(yaw);
RbgPrime(1, 1) = sin(roll)*sin(pitch)*cos(yaw) - cos(roll)*sin(yaw);
RbgPrime(1, 2) = cos(roll)*sin(pitch)*cos(yaw) + sin(roll)*sin(yaw);

RbgPrime(2, 0) = RbgPrime(2, 1) = RbgPrime(2, 2) = 0.0;

/////////////////////////////// END STUDENT CODE ////////////////////////////
```



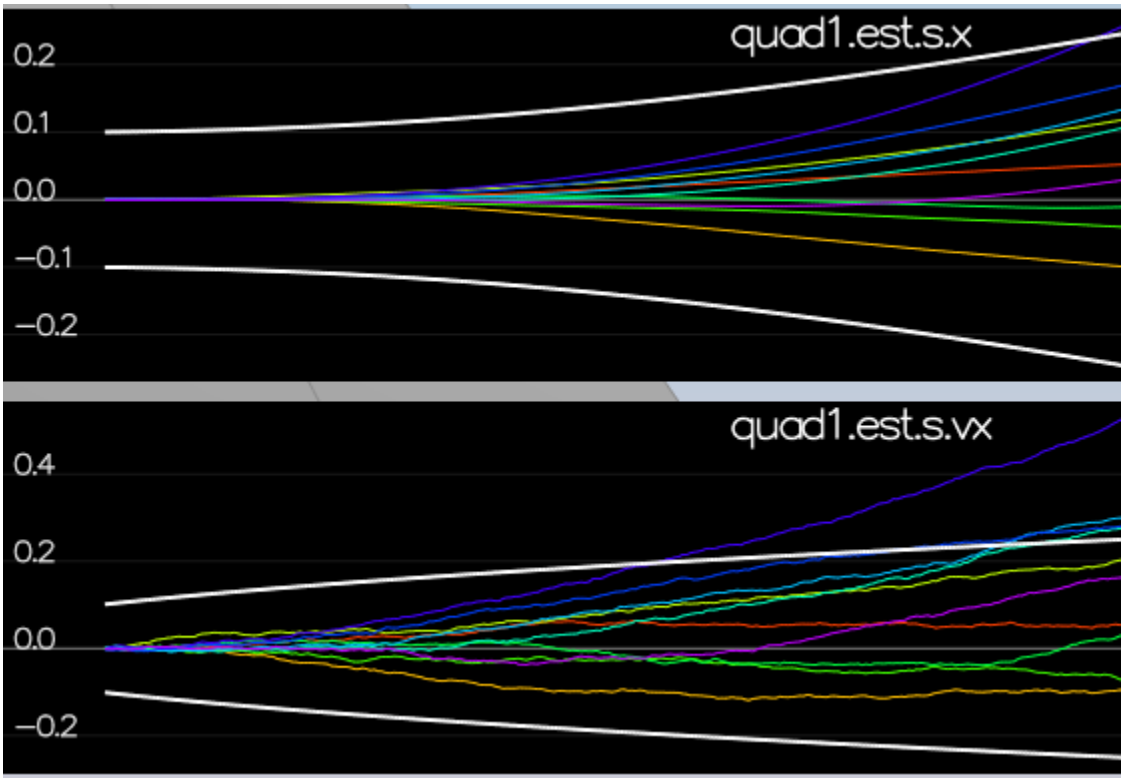
Hint: see section 7.2 of Estimation for Quadrotors for a refresher on the the transition model and the partial derivatives you may need

Hint: When it comes to writing the function for GetRbgPrime, make sure to triple check you've set all the correct parts of the matrix.

Hint: recall that the control input is the acceleration!

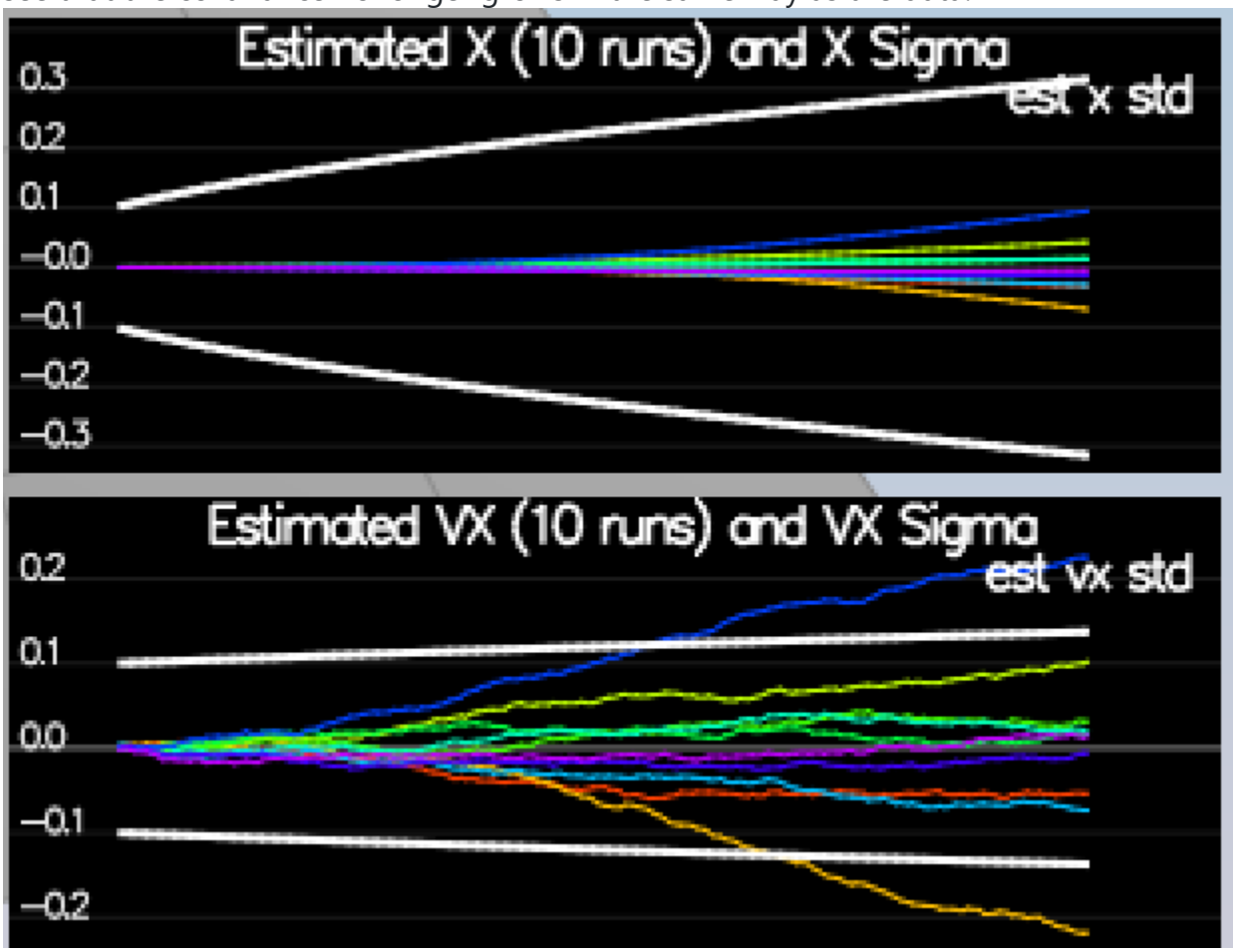
5. Run your covariance prediction and tune the `QPosXYStd` and the `QVelXYStd` process parameters in `QuadEstimatorEKF.txt`to try to capture the magnitude of the error you see. Note that as error grows our simplified model will not capture the real error dynamics (for example, specifically, coming from attitude errors), therefore try to make it look reasonable only for a relatively short prediction period (the scenario is set for one second). A good solution looks as follows:

```
///////////////////////////// BEGIN STUDENT CODE ////////////////////////////

// From "Estimation for Quadrotors" paper ( Eq. 51 )

gPrime(0, 3) = gPrime(1, 4) = gPrime(2, 5) = dt;

gPrime(3, 6) = (RbgPrime(0) * accel).sum() * dt;
gPrime(4, 6) = (RbgPrime(1) * accel).sum() * dt;
gPrime(5, 6) = (RbgPrime(2) * accel).sum() * dt;

ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;

///////////////////////////// END STUDENT CODE ////////////////////////////
```

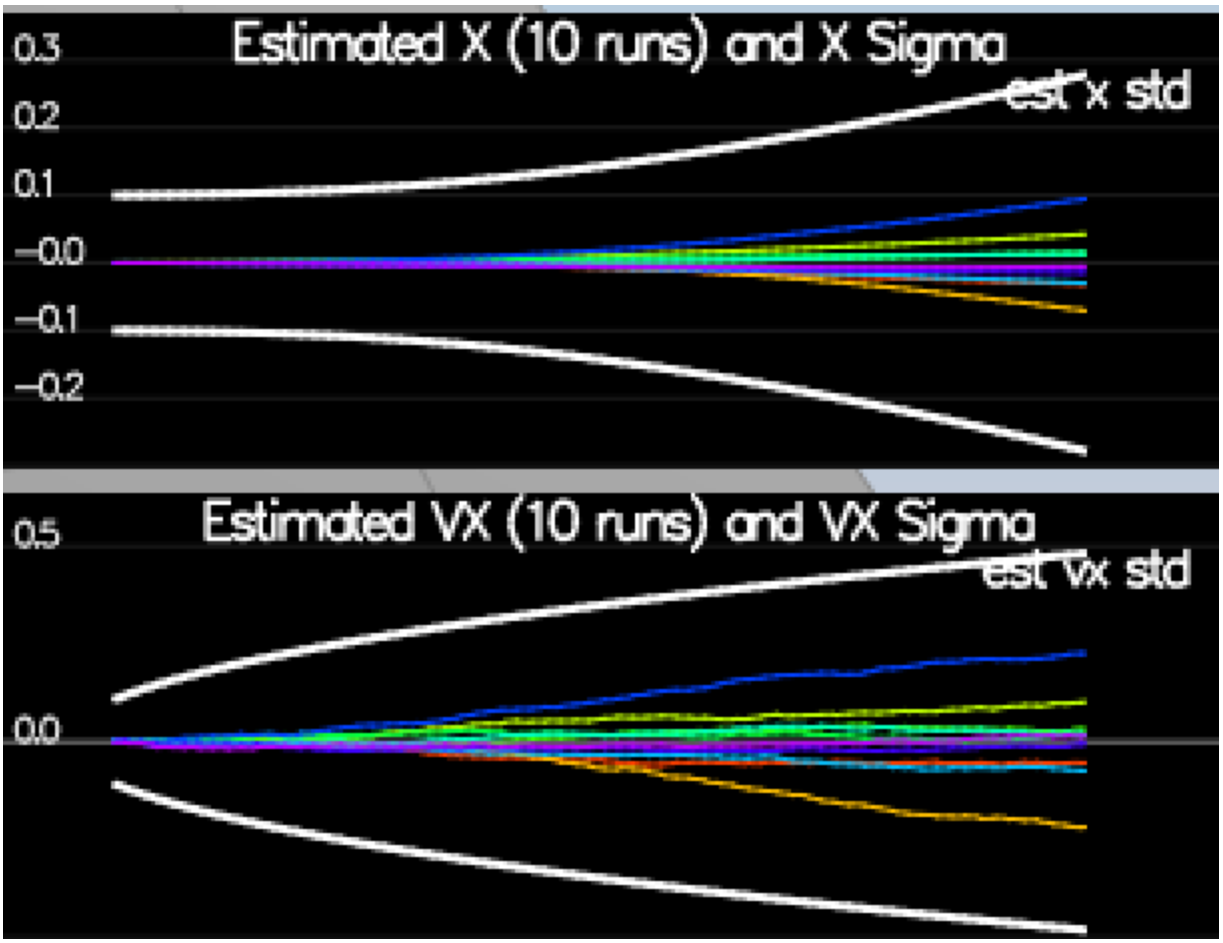


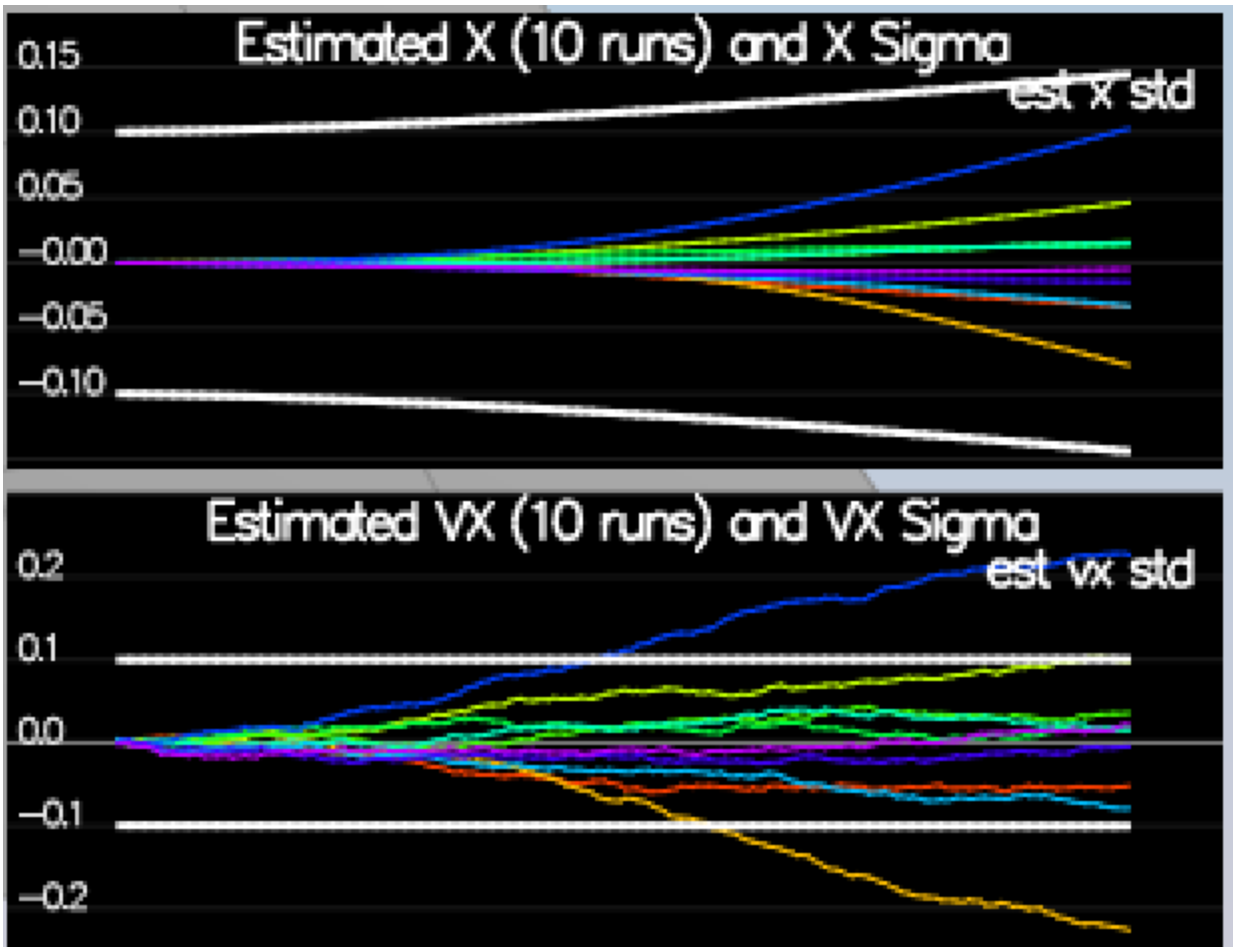
Looking at this result, you can see that in the first part of the plot, our covariance (the white line) grows very much like the data.

If we look at an example with a `QPosXYStd` that is much too high (shown below), we can see that the covariance no longer grows in the same way as the data.



Another set of bad examples is shown below for having a `QVelXYStd` too large (first) and too small (second). As you can see, once again, our covariances in these cases no longer model the data well.


Estimated X (10 runs) and X Sigma
est x std
0.3
0.2
0.1
−0.0
−0.1
−0.2
Estimated VX (10 runs) and VX Sigma
est Vx std
0.5
0.0

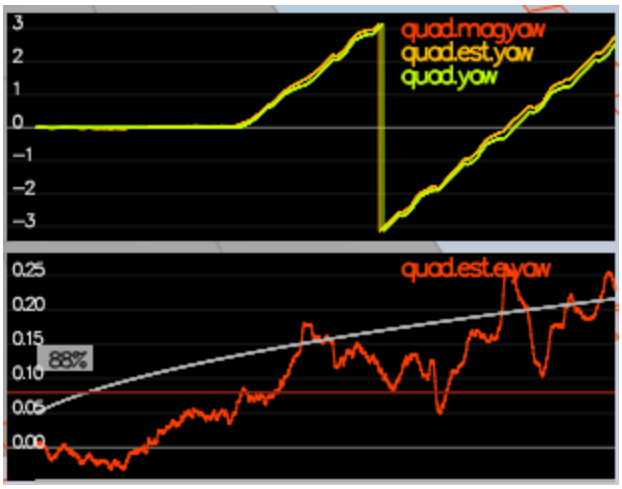


Success criteria: *This step doesn't have any specific measurable criteria being checked.*

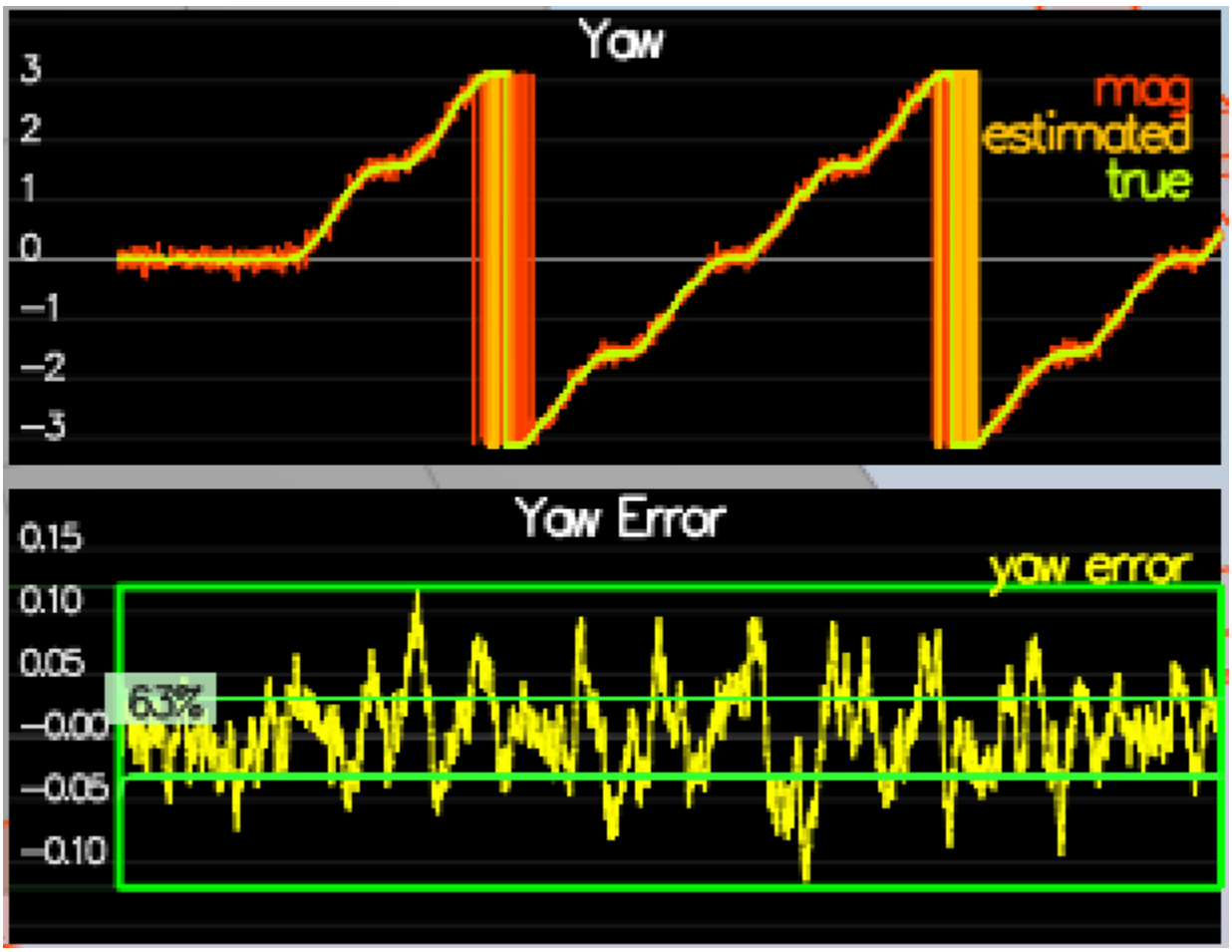
Step 4: Magnetometer Update

Up until now we've only used the accelerometer and gyro for our state estimation. In this step, you will be adding the information from the magnetometer to improve your filter's performance in estimating the vehicle's heading.

1. Run scenario `10_MagUpdate`. This scenario uses a realistic IMU, but the magnetometer update hasn't been implemented yet. As a result, you will notice that the estimate yaw is drifting away from the real value (and the estimated standard deviation is also increasing). Note that in this case the plot is showing you the estimated yaw error (`quad.est.e.yaw`), which is drifting away from zero as the simulation runs. You should also see the estimated standard deviation of that state (white boundary) is also increasing.
2. Tune the parameter `QYawStd` (`QuadEstimatorEKF.txt`) for the QuadEstimatorEKF so that it approximately captures the magnitude of the drift, as demonstrated here:



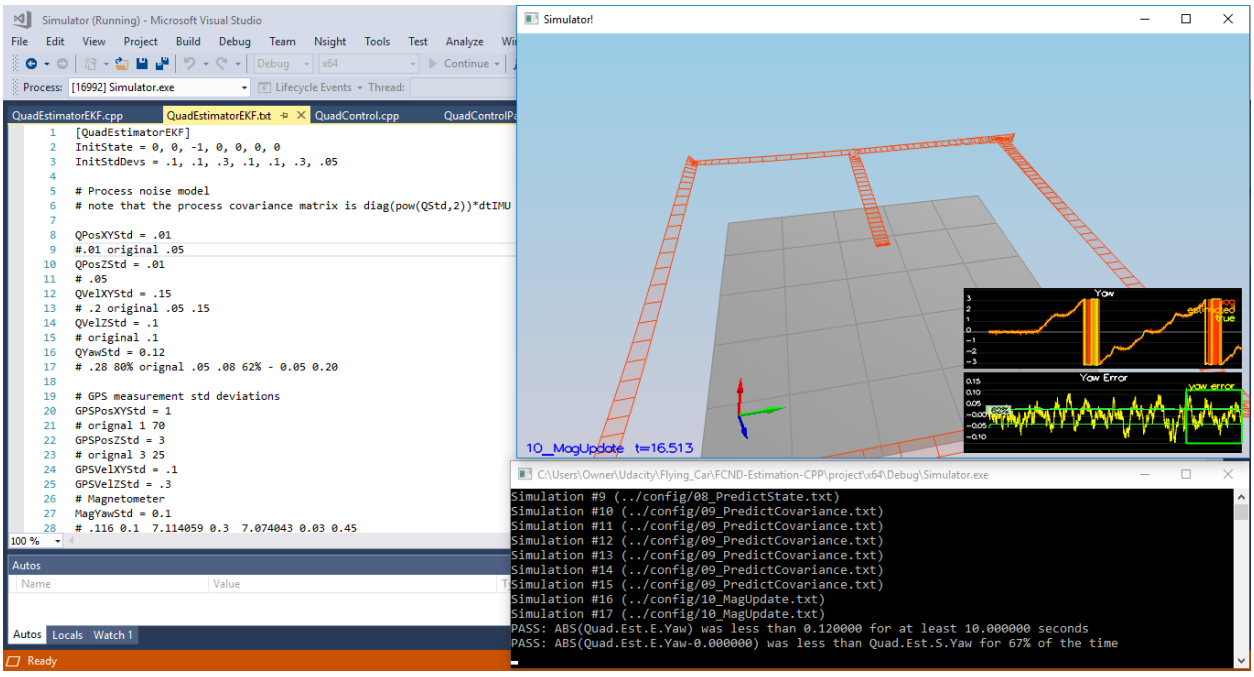
3. Implement magnetometer update in the function `UpdateFromMag()`. Once completed, you should see a resulting plot similar to this one:



Success criteria: *Your goal is to both have an estimated standard deviation that accurately captures the error and maintain an error of less than 0.1rad in heading for at least 10 seconds of the simulation.*

See Video 10_MagUpdate

```
// MAGNETOMETER UPDATE
// Hints:
//  - Your current estimated yaw can be found in the state vector: ekfState(6)
//  - Make sure to normalize the difference between your measured and estimated yaw
//    (you don't want to update your yaw the long way around the circle)
//  - The magnetomer measurement covariance is available in member variable R_Mag
///////////////////////////// BEGIN STUDENT CODE ////////////////////////////

hPrime(6) = 1.0; // set yaw; [0 0 0 0 0 0 1]

zFromX = hPrime * ekfState;
if (magYaw - zFromX[0] > F_PI) {
    zFromX[0] += 2.0 * F_PI;
}

else if (magYaw - zFromX[0] < -F_PI) {
    zFromX[0] += -2.0 * F_PI;
}

///////////////////////////// END STUDENT CODE ////////////////////////////
```



Hint: after implementing the magnetometer update, you may have to once again tune the parameter `QYawStd` to better balance between the long term drift and short-time noise from the magnetometer.
Hint: see section 7.3.2 of Estimation for Quadrotors for a refresher on the magnetometer update.

Step 5: Closed Loop + GPS Update

1. Run scenario `11_GPSUpdate`. At the moment this scenario is using both an ideal estimator and and ideal IMU. Even with these ideal elements, watch the position and velocity errors (bottom right). As you see they are drifting away, since GPS update is not yet implemented.
2. Let's change to using your estimator by setting `Quad.UseIdealEstimator` to 0 in `config/11_GPSUpdate.txt`. Rerun the scenario to get an idea of how well your estimator work with an ideal IMU.
3. Now repeat with realistic IMU by commenting out these lines in `config/11_GPSUpdate.txt`:

```
#SimIMU.AccelStd = 0,0,0
#SimIMU.GyroStd = 0,0,0
```

4. Tune the process noise model in `QuadEstimatorEKF.txt` to try to approximately capture the error you see with the estimated uncertainty (standard deviation) of the filter.
5. Implement the EKF GPS Update in the function `UpdateFromGPS()`.

6. Now once again re-run the simulation. Your objective is to complete the entire simulation cycle with estimated position error of < 1m (you'll see a green box over the bottom graph if you succeed). You may want to try experimenting with the GPS update parameters to try and get better performance.

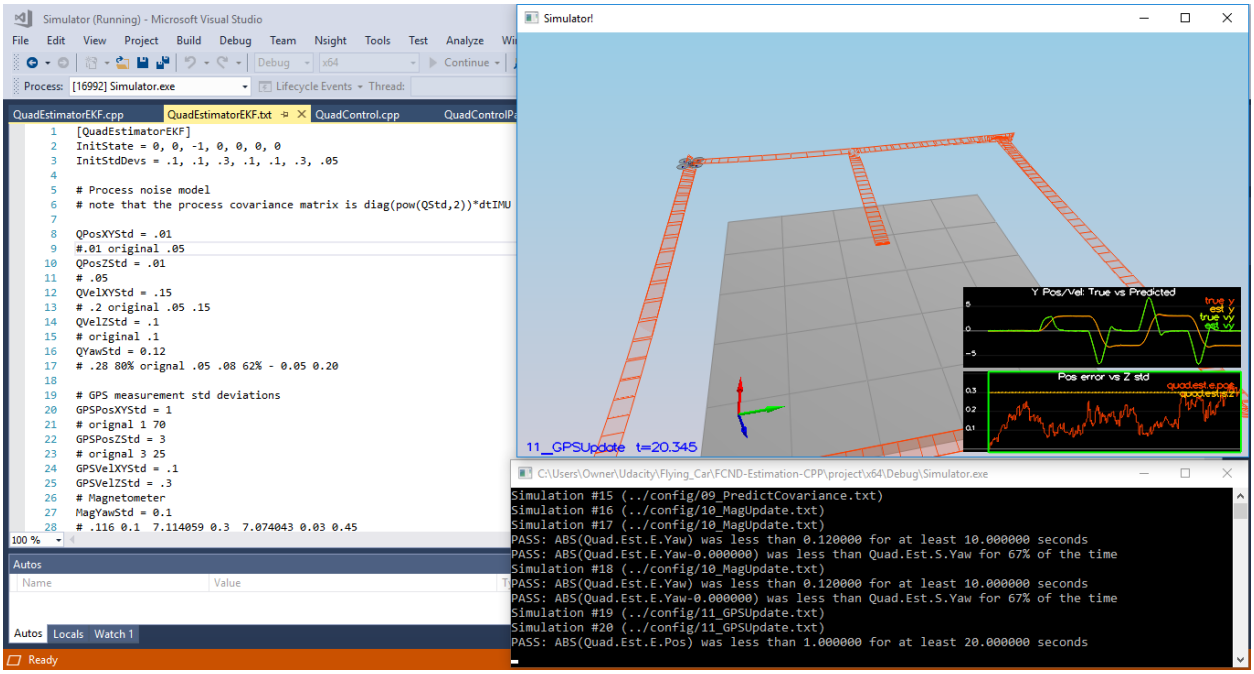
Success criteria: *Your objective is to complete the entire simulation cycle with estimated position error of < 1m.*

Hint: see section 7.3.1 of Estimation for Quadrotors for a refresher on the GPS update.

At this point, congratulations on having a working estimator!

See video 11_GPSUpdate

```
// GPS UPDATE
// Hints:
//  - The GPS measurement covariance is available in member variable R_GPS
//  - this is a very simple update
//////////////////////////////// BEGIN STUDENT CODE ////////////////////////////

for (char i = 0; i < 6; i++) {
    // From "Estimation for Quadrotors" paper ( Eq. 55 )
    hPrime(i, i) = 1.0;
    // From "Estimation for Quadrotors" paper ( Eq. 53 & Eq. 54 )
    zFromX(i) = ekfState(i);
}

//////////////////////////////// END STUDENT CODE //////////////////////////////
```



Step 6: Adding Your Controller

Up to this point, we have been working with a controller that has been relaxed to work with an estimated state instead of a real state. So now, you will see how well your controller performs and de-tune your controller accordingly.

1. Replace `QuadController.cpp` with the controller you wrote in the last project.
2. Replace `QuadControlParams.txt` with the control parameters you came up with in the last project.

3. Run scenario `11_GPSUpdate`. If your controller crashes immediately do not panic. Flying from an estimated state (even with ideal sensors) is very different from flying with ideal pose. You may need to de-tune your controller. Decrease the position and velocity gains (we've seen about 30% detuning being effective) to stabilize it. Your goal is to once again complete the entire simulation cycle with an estimated position error of < 1m.

Hint: you may find it easiest to do your de-tuning as a 2 step process by reverting to ideal sensors and de-tuning under those conditions first.

Success criteria: *Your objective is to complete the entire simulation cycle with estimated position error of < 1m.*

Tips and Tricks

- When it comes to transposing matrices, `.transposeInPlace()` is the function you want to use to transpose a matrix
- The Estimation for Quadrotors document contains a helpful mathematical breakdown of the core elements on your estimator

Submission

For this project, you will need to submit:

- a completed estimator that meets the performance criteria for each of the steps by submitting:
    - `QuadEstimatorEKF.cpp`
    - `config/QuadEstimatorEKF.txt`
- a re-tuned controller that, in conjunction with your tuned estimator, is capable of meeting the criteria laid out in Step 6 by submitting:
    - `QuadController.cpp`
    - `config/QuadControlParams.txt`
- a write up addressing all the points of the rubric