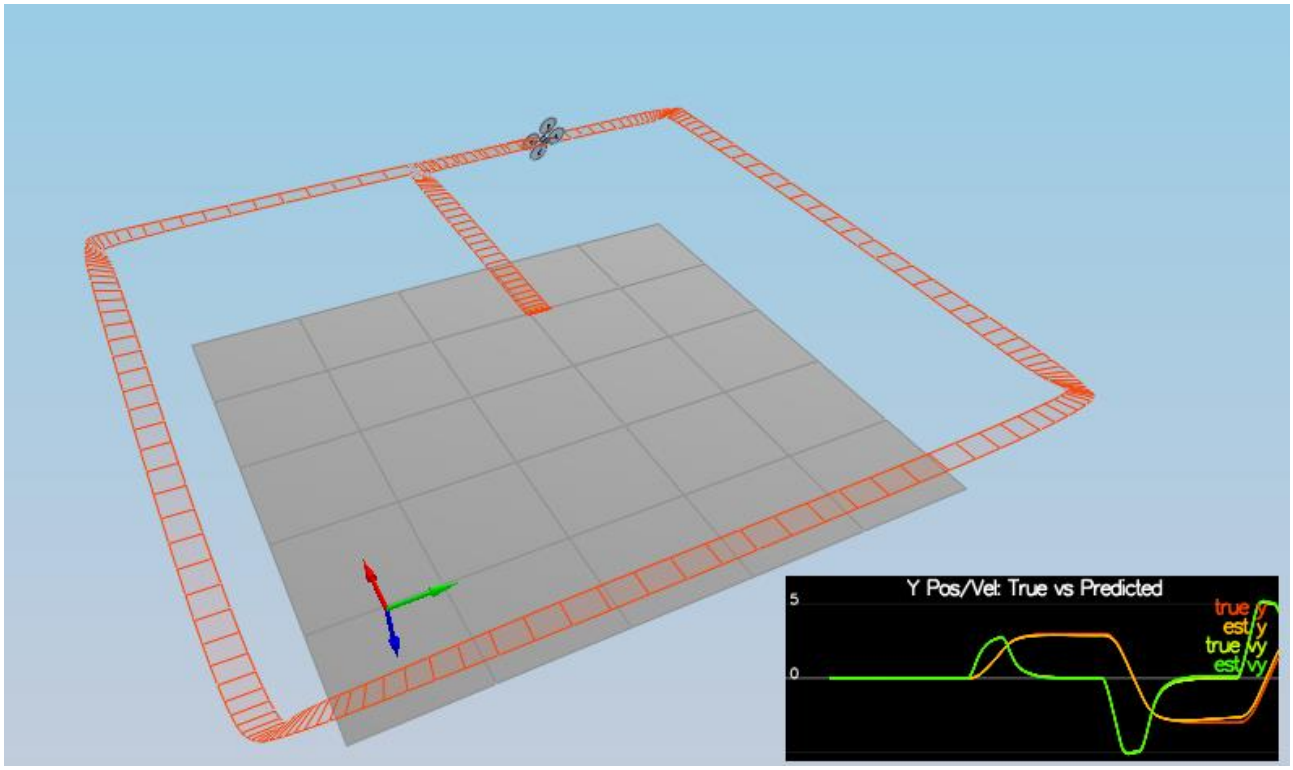
06/03/2019

# Project#4: 3D Estimation

# Student: Enrico Baracaglia




This is the last project of the Flying Car Nanodegree and it involves the development of an Extended Kalman Filter (EKF) with the objective of building the estimation portion of the controller used in the C++ simulator.

The files to be submitted for revision are:

*/config/QuadEstimatorEKF.txt*: This file contains the tuning parameters for the EKF, which can be modified while the simulation is running.

*/src/QuadEstimatorEKF.cpp*: This is where the EKF will be developed and implemented.

*/src/QuadControl.cpp*: This file comes from the previous project (*Project 3: Building a Controller*) and it contains the implementation of the PID controller used to control the drone. It will be used at the end of the project together with the estimation part.

*/config/QuadControlParams.txt*: This file also comes from the previous project and it contains the tuning parameters for the PID controller, which can be tuned while the simulation is running.

**The tasks**

The project is structured in different tasks, each of them having its success criteria used to validate the task and proceed to the next one. As before, the success criteria are displayed both in the plots and in the terminal output to help the student along the way.

**Project outline**

- Step 1: Sensor Noise
- Step 2: Attitude Estimation
- Step 3: Prediction Step
- Step 4: Magnetometer Update
- Step 5: Closed Loop + GPS Update
- Step 6: Adding Your Controller

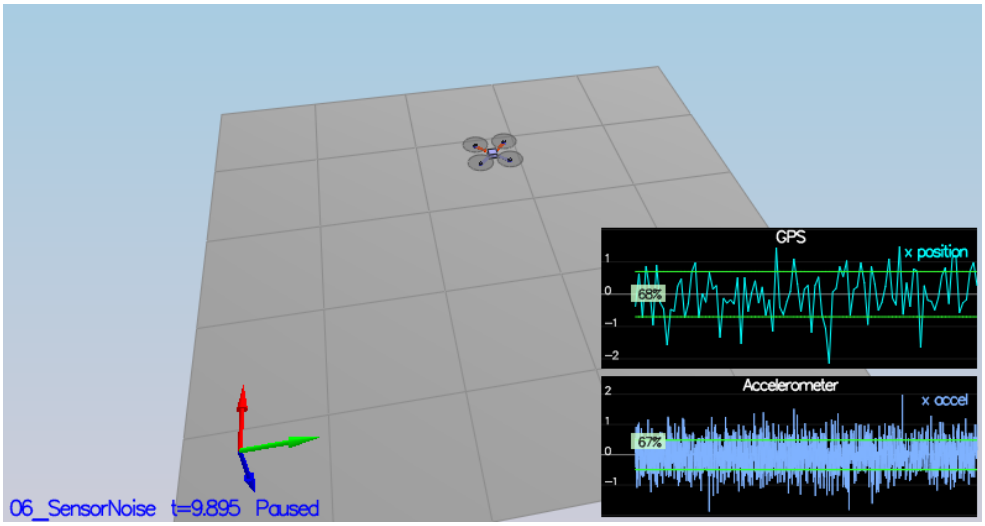
**Step 1: Sensor Noise**

This step is used to add realism to the project since here we are adding noise to the quad's sensors. The first task was to run the `06_NoisySensors` scenario to record some sensor data. In particular, the GPS X position and the accelerometer's x measurements were recorded and analysed in order to calculate the standard deviation of both samples. After writing a short code in Python (shown below), the following standard deviations were computed:

- `MeasuredStdDev_GPSPosXY = 0.70`
- `MeasuredStdDev_AccelXY = 0.49`

```python
import statistics
import matplotlib.pyplot as plt

def main():

    timeSteps = []
    values = []
    file = open('Graph2.txt', 'r')
    file.readline()
    for line in file:
        mySplit = line.split(',')
        timeSteps.append(float(mySplit[0]))
        values.append(float(mySplit[1]))
    file.close()
    standDeviation = statistics.stdev(values)

    plt.plot(values)
    plt.show()

    return(standDeviation)

if __name__ == "__main__":
    print(main())
```

The success criteria required the standard deviations to accurately capture the value of approximately 68% of the respective measurements. In the figure below, we can see the scenario while passing the test.



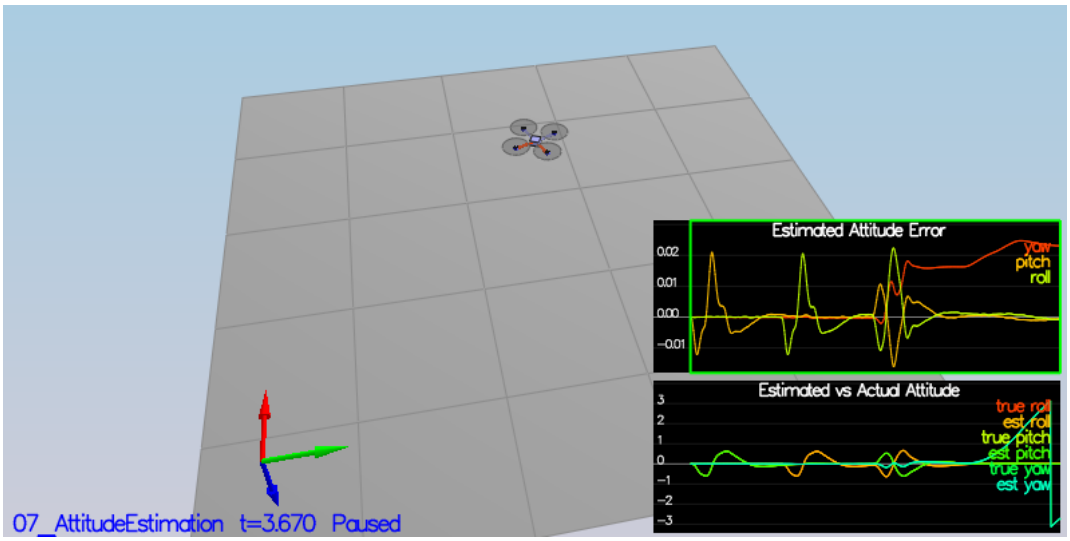

**Step 2: Attitude Estimation**

This step involved the use of the information coming from the IMU in order to improve the complementary filter with a better rate gyro attitude integration scheme. Here, it was required to run scenario `07_AttitudeEstimation` and modify the code in the `UpdateFromIMU()` function. In particular, we needed to modify the linear implementation of the filter and implement a non-linear one in order to get better results.

We proceeded by creating a rotation matrix based on the current Euler angles in order to find the roll, pitch and yaw derivatives $(\dot{\varphi}, \dot{\theta}, \dot{\psi})$ from the angular rates of change of the quad in body frame $(p, q, r)$ coming from the gyro. The equation used was the one explained in the lecture on controls:

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

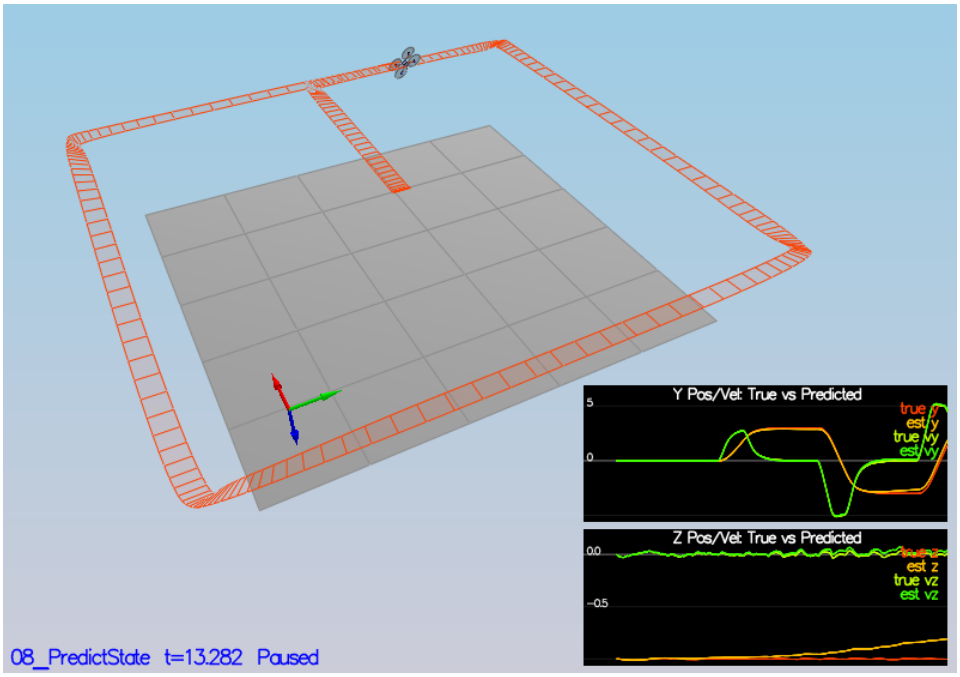
This was done with the objective of implementing a better integration method that used the current attitude estimates. The success criteria required the attitude estimator to get within 0.1 rad for each of the Euler angles for at least 3 seconds. The scenario passing the test is shown in the figure below.




**Step 3: Prediction Step**

This step did not have any specific measurable criteria being checked; however, the task was to implement the prediction step of the filter in the `PredictState()` function by using scenario `08_PredictState`.

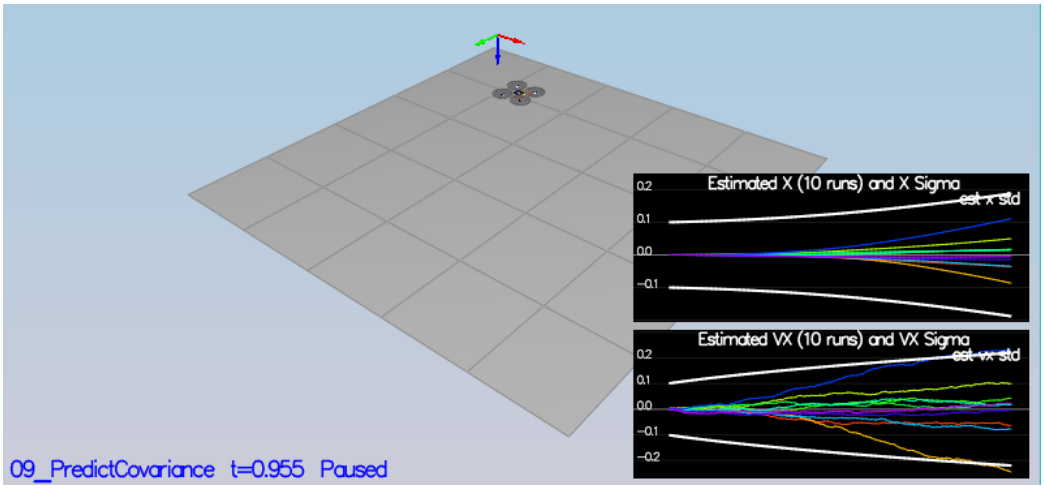
In order to predict the current state forward by time dt, we used the current state and the measurements from the accelerometer and the gyro. Since the dt was very short (on the order of 1ms), simplistic integration methods were considered to be fine. After the implementation of the code, we run the scenario and we observed the following result.




As we can see, the estimator state tracks the actual state, with only reasonably slow drift.

After that, we introduced an IMU with noise and in scenario `09_PredictionCov` we observed the results for a small fleet of quadrotors. We noticed that the estimated covariance (white bounds) did not capture the growing errors, requiring therefore the update of the covariance matrix. Following section 7.2 of *Estimation for Quadrotors*, the partial derivative of the body-to-global rotation matrix was implemented in the `GetRbgPrime()` function. After that, the rest of the prediction step was implemented in the `Predict()` function.

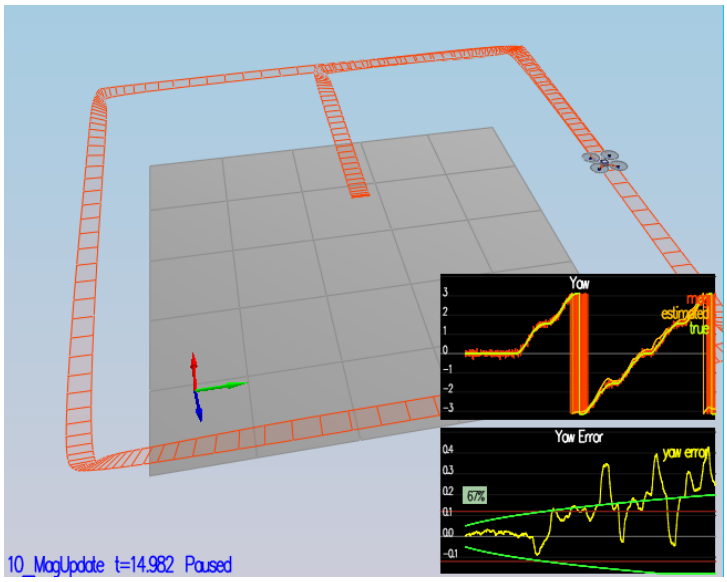
Then, we run the covariance prediction and tuned the `QPosXYStd` and the `QVelXYStd` parameters in `QuadEstimatorEKF.txt` in order to try to capture the magnitude of the error shown in the graphs.

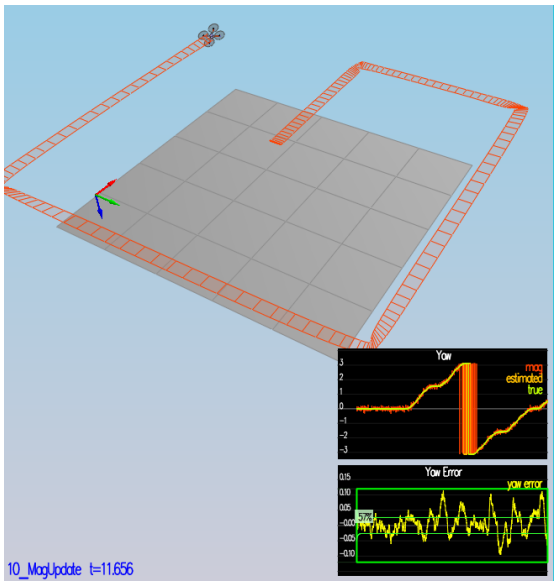



**Step 4: Magnetometer Update**

Until now, we only used the gyro and accelerometer for our state estimation. In this step, we update the state also taking into account the magnetometer measurements. The implementation of the magnetometer update was done in the `UpdateFromMag()` function, following the steps outlined in section 7.3.2 of the *Estimation for Quadrotors* paper.

After that, we run the scenario `10_MagUpdate` and we tuned the parameter `QYawStd` in `QuadEstimatorEKF.txt` in order to approximately capture the magnitude of the drift. The success criteria required to both have an estimated standard deviation that accurately captured the error and maintained an error of less than 0.1rad in heading for at least 10 seconds of the simulation. The successful result is shown in the figures below.

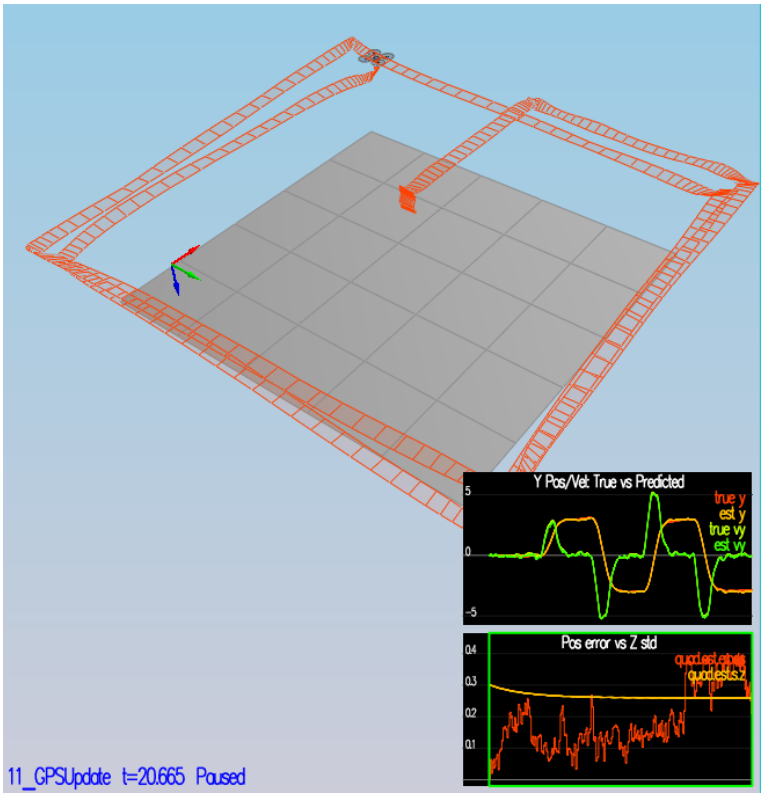



**Step 5: Closed Loop + GPS Update**

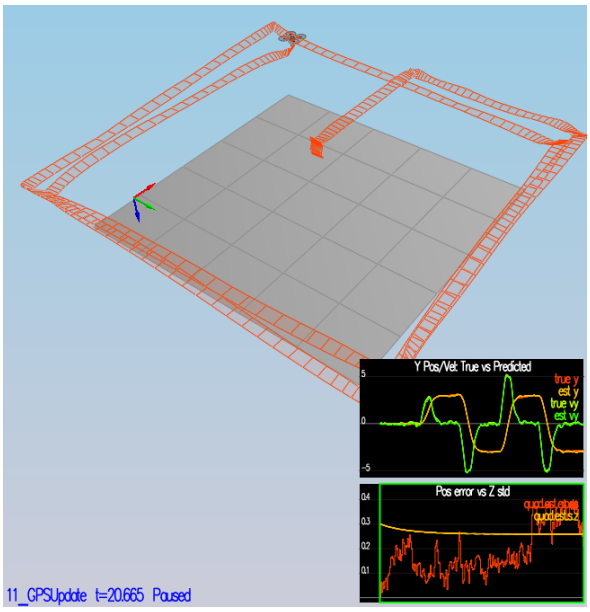
In this step, the objective was to finish the EKF implementation by also using the GPS update. Following the steps outlined in section 7.3.1 of the *Estimation for Quadrotors* paper we implemented the GPS update and we run the `11_GPSUpdate` scenario.

The success criteria required to complete the entire simulation cycle with an estimated position error of less than 1m. The successful result is shown below.

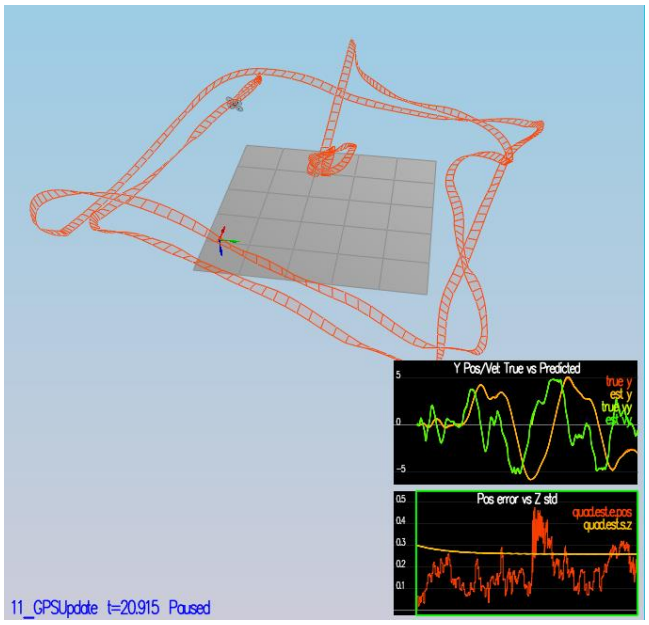


**Step 6: Adding Your Controller**

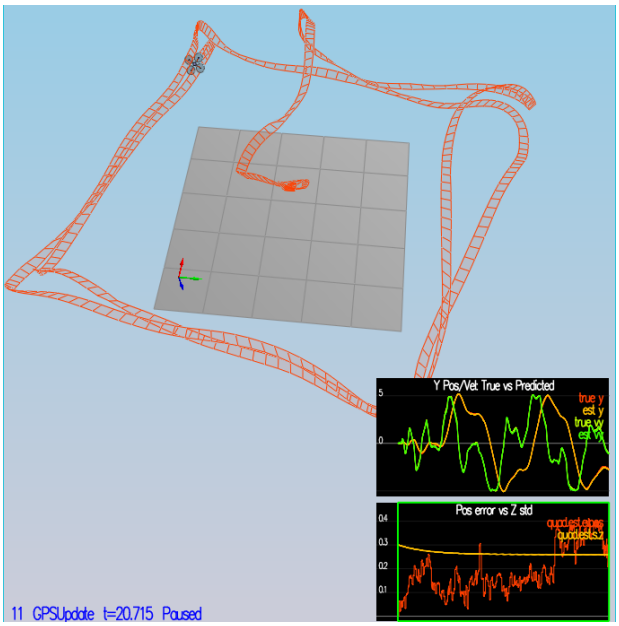
The last step was to add the controller written in the previous project to see how it behaved in a noisy scenario. The first result, keeping the same controller parameters used in the last project, was meeting the success criteria; however, the followed trajectory was quite noisy (figure on the left). Therefore, we de-tuned a bit the controller parameters in order to stabilize the drone. In particular, the Udacity suggestion was followed and the position and velocity gains were reduced by around 30%. After some re-tuning, the quad was more stable, following a less noisy trajectory (on the right).




Controller coupled with perfect sensors




Before retuning




After retuning