Control of a 3D Quadrotor

Naruhiko Nakanishi

Abstract—The controller is implemented in C++. The code here is eventually transferred to a real drone. The repository is cloned and gotten familiar with the C++ environment. Each of the scenarios (Body rate and roll/pitch control (scenario 2), Position/velocity and yaw angle control (scenario 3), Non-idealities and robustness (scenario 4)) is completed. This involves implementing and tuning controllers incrementally. The controller is tuned and it works to successfully meet each of the evaluations in each scenario.

Index Terms—Flying Car, Quadrotor, Drone, Control, C++.

1 Introduction

In this project, some of logic is ported over to a controller that's written in C++. This code controls a drone in an entirely new simulator. The simulator in this project is more bare-bones than the Python / Unity simulator, but it's more realistic in the physics that it models.

Once the controller meets the required specs with the C++ simulator, the project has been completed. And for more hardware minded students, it is also ready for running the controller on a real drone.

The goal of this project is as follows. In the real world the flight controller is usually implemented in C or C++. So in this project the controller is implemented in C++. The code here is eventually transferred to a real drone. Fig.1 shows the 3D Control Architecture Diagram. This shows a cascade PID control system.

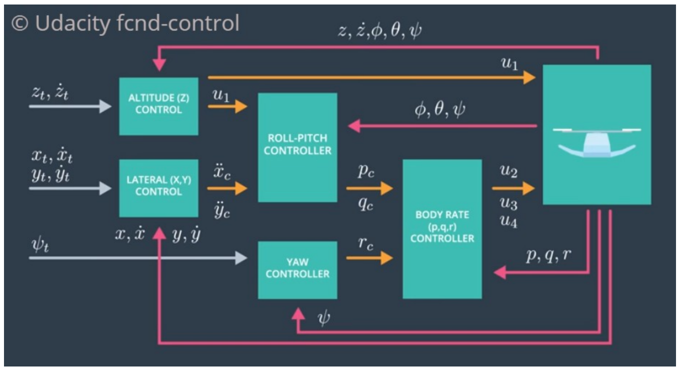

Fig. 1. 3D Control Architecture

2 Development Environment Setup

Regardless of the development platform, the first step is to download or clone the C++ simulator repository. [1]

git clone https://github.com/udacity/FCND-Controls-CPP.git

Once the code for the simulator is obtained, the necessary compiler and IDE necessary for running the simulator are installed.

3 Simulator Walkthrough

3.1 The Code

For the project, the majority of the code is written in src/QuadControl.cpp. This file contains all of the code for the controller that is developed.

All the configuration files for the controller and the vehicle are in the config directory. For example, for all the control gains and other desired tuning parameters, there is a config file called QuadControlParams.txt set up.

3.2 The Simulator

The simulator is also used to fly some different trajectories to test out the performance of the C++ implementation of the controller. These trajectories, along with supporting code, are found in the traj directory of the repo.

In the simulator window itself, the window is right clicked to select between a set of different scenarios that are designed to test the different parts of your controller.

4 The Tasks

4.1 Testing it Out (scenario 1 " Intro")

When the simulator is run, the quad is falling straight down. This is due to the fact that the thrusts are simply being set to: QuadControlParams.Mass * 9.81 / 4.

Therefore, if the mass doesn't match the actual mass of the quad, it'll fall down. Take a moment to tune the Mass parameter in QuadControlParams.txt to make the vehicle more or less stay in the same spot.

4.2 Body rate and roll/pitch control (scenario 2)

First, the body rate and roll / pitch control is implemented. For the simulation, Scenario 2 is used. In this scenario, a quad is seen above the origin. It is created with a small initial rotation speed about its roll axis. The controller needs to stabilize the rotational motion and bring the vehicle back to level attitude.

4.3 Position/velocity and yaw angle control (scenario 3)

Next, the position, altitude and yaw control for the quad is implemented. For the simulation, Scenario 3 is used. This creates 2 identical quads, one offset from its target point (but initialized with yaw = 0) and second offset from target point but yaw = 45 degrees.

4.4 Non-idealities and robustness (scenario 4)

In this part, some of the non-idealities and robustness of a controller are explored. For this simulation, Scenario 4 is used. This is a configuration with 3 quads that are all trying to move one meter forward.

This time, these quads are all a bit different: The green quad has its center of mass shifted back. The orange vehicle is an ideal quad. The red vehicle is heavier than usual.

4.5 Tracking trajectories (scenario 5)

Now that all the working parts of a controller are obtained, put all together and test the performance once again on a trajectory. For this simulation, Scenario 5 is used. This scenario has two quadcopters: The orange one is following traj/FigureEight.txt. The other one is following traj/FigureEightFF.txt. For now this is the same trajectory.

5 EVALUATION

To assist with tuning of the controller, the simulator contains real time performance evaluation. A set of performance metrics is defined for each of the scenarios that the controllers must meet for a successful submission.

5.1 Performance Metrics

The specific performance metrics are as follows.

Scenario 2: Roll should less than 0.025 radian of nominal for 0.75 seconds (3/4 of the duration of the loop). Roll rate should less than 2.5 radian/sec for 0.75 seconds.

Scenario 3: X position of both drones should be within 0.1 meters of the target for at least 1.25 seconds. Quad2 yaw should be within 0.1 of the target for at least 1 second.

Scenario 4: Position error for all 3 quads should be less than 0.1 meters for at least 1.5 seconds.

Scenario 5: Position error of the quad should be less than 0.25 meters for at least 3 seconds.

6 RESULTS

QuadController.cpp and QuadControlParams.txt are edited. [2] They contain the completed C++ controller and associated gains.

6.1 Testing it Out (scenario 1 ” Intro”)

Fig.2 shows the result of scenario 1 ” Intro” with the proper mass. The Mass parameter is tuned in QuadControlParams.txt to make the vehicle more or less stay in the same spot.

6.2 Body rate and roll/pitch control (scenario 2)

Fig.3 shows the result of scenario 2.

6.2.1 Implement body rate control

The codes in the function GenerateMotorCommands() and in the function BodyRateControl() are implemented, and kpPQR in QuadControlParams.txt is tuned to get the vehicle to stop spinning quickly but not overshoot.

The rotation of the vehicle about roll (omega.x) gets controlled to 0 while other rates remain zero.

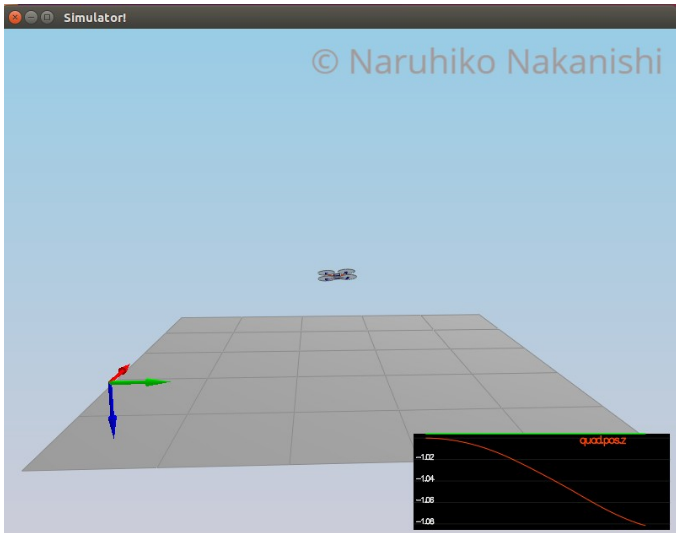

Fig. 2. scenario 1: Testing it Out

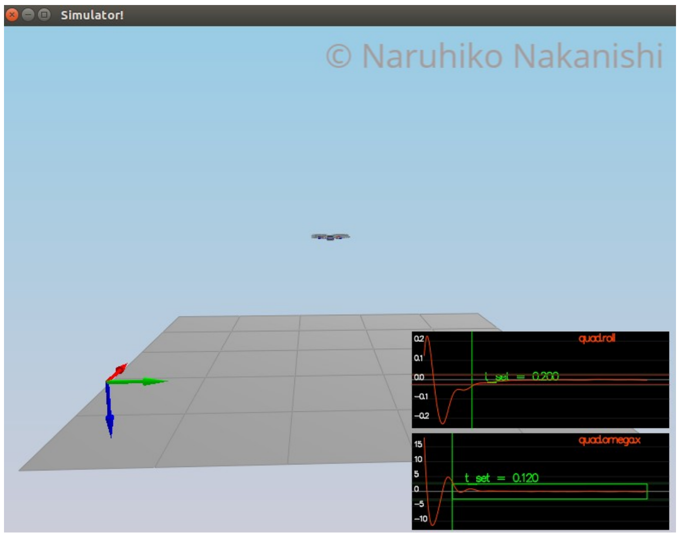

Fig. 3. scenario 2; Body rate and roll/pitch control

6.2.2 Implement roll / pitch control

The code in the function RollPitchControl() is implemented, and kpBank in QuadControlParams.txt is tuned to minimize settling time but avoid too much overshoot.

The quad levels itself, though it is still flying away slowly, and the vehicle angle (Roll) gets controlled to 0.

6.3 Position/velocity and yaw angle control (scenario 3)

The codes in the function LateralPositionControl() and in the function AltitudeControl() are implemented, and parameters kpPosXY, kpPosZ, kpVelXY and kpVelZ are tuned.

The quads is going to their destination points and tracking error is going down. However, one quad remains rotated in yaw. So the code in the function YawControl() is implemented, and parameters kpYaw and the 3rd (z) component of kpPQR are tuned.

For a second order system, such as the one for this quadcopter, the velocity gain (kpVelXY and kpVelZ) is 4 times greater than the respective position gain (kpPosXY and kpPosZ).

Fig.4 shows the result scenario 3.

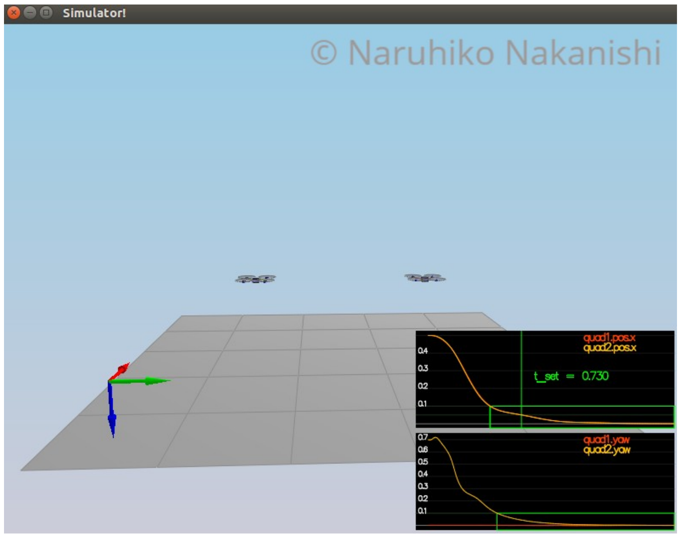

Fig. 4. scenario 3: Position/velocity and yaw angle control

6.4 Non-idealities and robustness (scenario 4)

The controller and parameter set are run. AltitudeControl() is edited to add basic integral control to help with the different-mass vehicle. The integral control, and other control parameters are tuned until all the quads successfully move properly.

Fig.5 shows the result scenario 4.

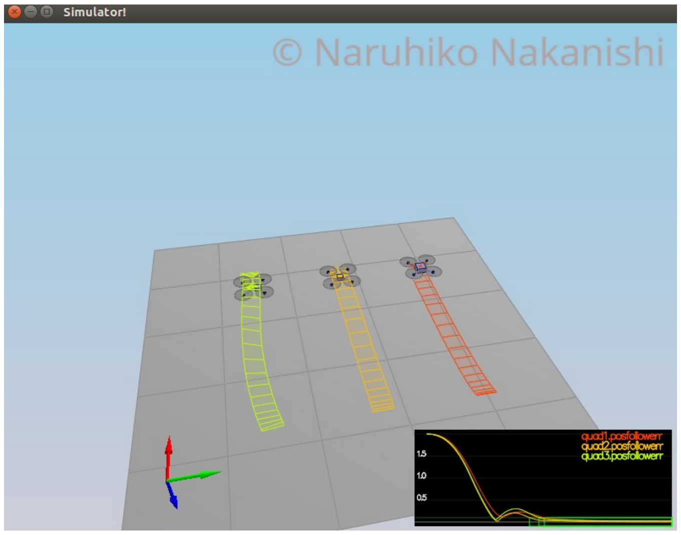

Fig. 5. scenario 4: Non-idealities and robustness

6.5 Tracking trajectories (scenario 5)

The performance of the drone is improved by adjusting how the trajectory is defined. The drone follows the trajectory well. It is able to hold to the path fairly well.

Fig.6 shows the result scenario 5.

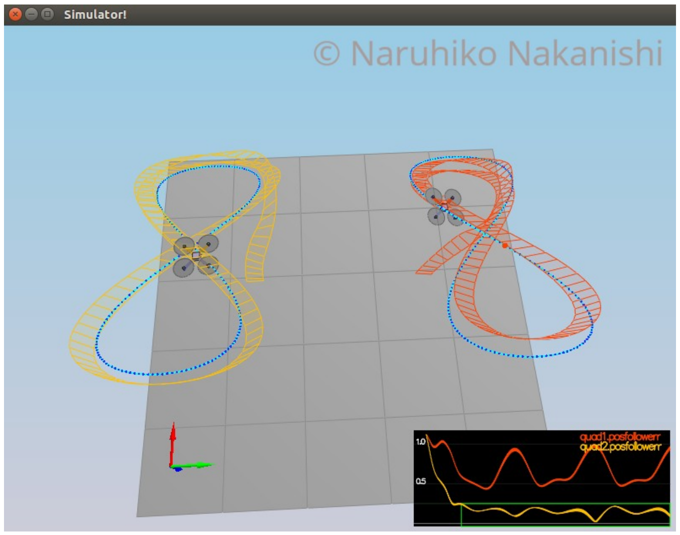

Fig. 6. scenario 5: Tracking trajectories

7 Conclusion

The controller is implemented in C++. The code here is eventually transferred to a real drone. The repository is cloned and gotten familiar with the C++ environment. Each of the scenarios (Body rate and roll/pitch control (scenario 2), Position/velocity and yaw angle control (scenario 3), Non-idealities and robustness (scenario 4)) is completed. This involves implementing and tuning controllers incrementally. The controller is tuned and it works to successfully meet each of the evaluations in each scenario.

7.1 Implemented Controller

Body rate control in C++ is Implemented. The controller is a proportional controller on body rates to commanded moments. The controller takes into account the moments of inertia of the drone when calculating the commanded moments.

Roll pitch control in C++ is implemented. The controller uses the acceleration and thrust commands, in addition to the vehicle attitude to output a body rate command. The controller accounts for the non-linear transformation from local accelerations to body rates.

Altitude controller in C++ is implemented. The controller uses both the down position and the down velocity to command thrust. The output value is indeed thrust (the drone's mass needs to be accounted for) and the thrust includes the non-linear effects from non-zero roll/pitch angles. Additionally, the C++ altitude controller contains an integrator to handle the weight non-idealities presented in scenario 4.

Lateral position control in C++ is implemented. The controller uses the local NE position and velocity to generate a commanded local acceleration.

Yaw control in C++ is implemented. The controller is a linear/proportional heading controller to yaw rate commands (non-linear transformation not required).

Calculating the motor commands given commanded thrust and moments in C++ is implemented. The thrust and moments are converted to the appropriate 4 different desired thrust forces for the moments. The dimensions of the drone are properly accounted for when calculating thrust from moments.

7.2 Flight Evaluation

The C++ controller is successfully able to fly the provided test trajectory and visually passes inspection of the scenarios leading up to the test trajectory. In each scenario the drone looks stable and performs the required task. Specifically the controller is able to handle the non-linearities of scenario 4.

REFERENCES

[1] Udacity, "https://github.com/udacity/fcnd-controls-cpp,"
[2] N. Nakanishi, "https://github.com/grapestone5321/fcnd-control of a 3d-quadrotor cpp,"