FCND-Term1-P3-3D-Quadrotor-Controller

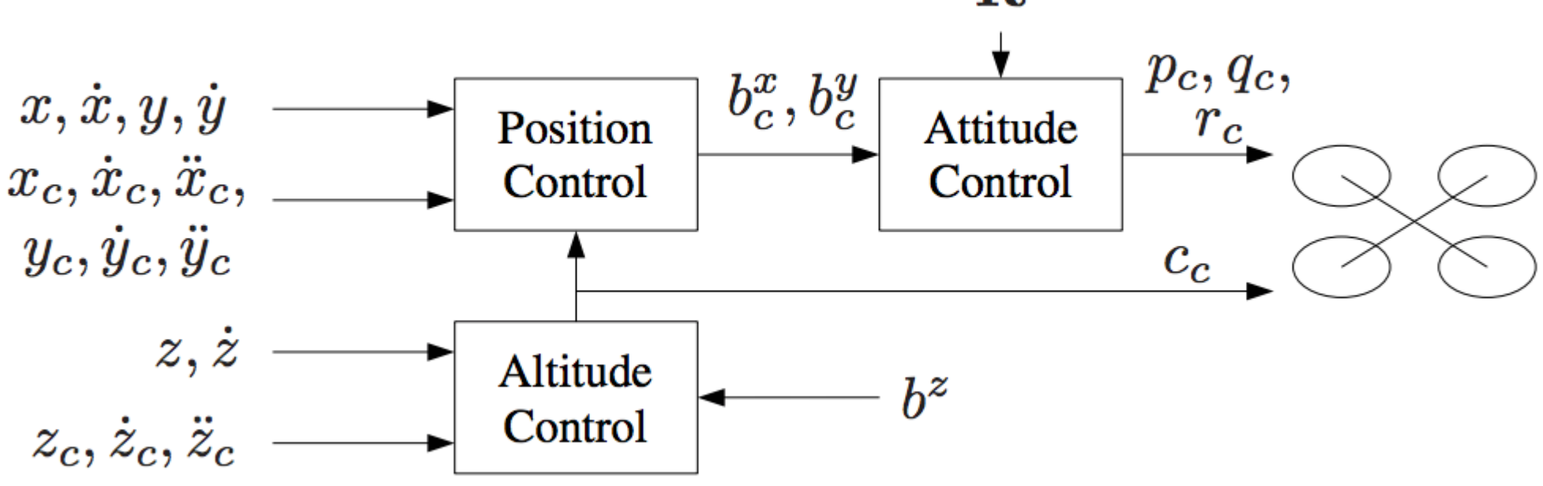
Udacity Flying Car Nanodegree - Term 1 - Project 3 - 3D Quadrotor Controller

View on GitHub

FCND-Term1-P3-3D-Quadrotor-Controller

Udacity Flying Car Nanodegree - Term 1 - Project 3 - 3D Quadrotor Controller

In this project, you get to implement and tune a cascade PID controller for drone trajectory tracking. The theory behind the controller design using feed-forward strategy is explained in details on our instructor, Angela P. Schoellig, on her paper Feed-Forward Parameter Identification for Precise Periodic Quadrocopter Motions. The following diagram could be found on that paper describing the cascaded control loops of the trajectory-following controller:



Project description

There two parts for this project where the controller needs to be implemented with python on the first one, and with c++ in the second one.

Python implementation

Based on the first project on the FCND. We need to control a simulated drone using python to fly in a square trajectory in a backyard. The controller needs to be implemented on the controller.py class. Udacity provides a seed project all the code you need to be able to focus only on the controller, but changes on that code is welcome as well. Udacity's FCND Simulator could be downloaded here. The python code use Udacidrone API to communicate with the simulator. This API use MAVLink protocol.

Prerequisites

To run this project, you need to have the following software installed:

- Miniconda with Python 3.6. I had some problems while installing this on my Mac after having an older version install and some other packages install with Homebrew. I have to manually delete all the `~/*conda*` directory from my home and then install it with `bash Miniconda3-latest-MacOSX-x86_64.sh -b`.
- Udacity FCND Simulator the latest the better.

Run the code

Change directory to where you clone this repo and move to the /python directory. Let's call that directory **REPO_PYTHON_PATH**. Create the conda environment for this project:

```
conda env create -f environment.yml
```

Note: This environment configuration is provided by Udacity at the FCND Term 1 Starter Kit repo.

Activate your environment with the following command:

```
source activate fcnd
```

Start the drone simulator. You will see something similar to the following image:

UDACITY

SELF-DRIVING CAR ENGINEER

FLYING CAR SIMULATOR

BACKYARD FLYER

MOTION PLANNING

CONTROLS

QUIT

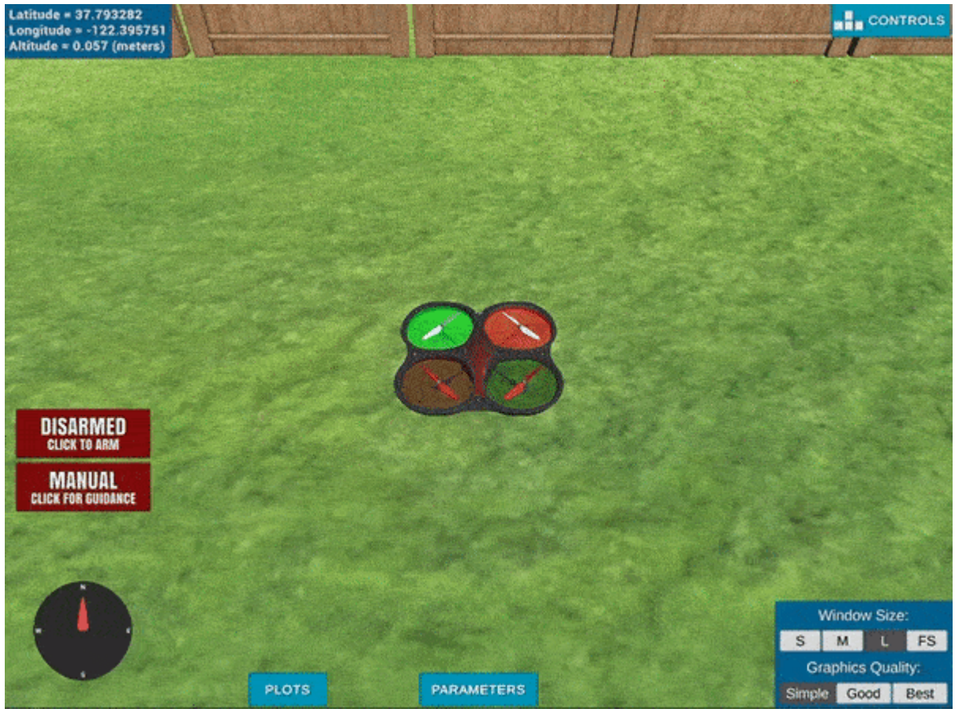
Click on **CONTROLS** item, and you will see the drone on the ground:


Latitude = 37.793282
Longitude = -122.395754
Altitude = 0.243 (meters)
CONTROLS
DISARMED
CLICK TO ARM
MANUAL
CLICK FOR GUIDANCE
N
W
E
S
PLOTS
PARAMETERS
Window Size:
S
M
L
FS
Graphics Quality:
Simple
Good
Best

At this point, the simulator is waiting for connections ready to fly the drone. Time to run the controller.

```
python controls_flyer.py
```

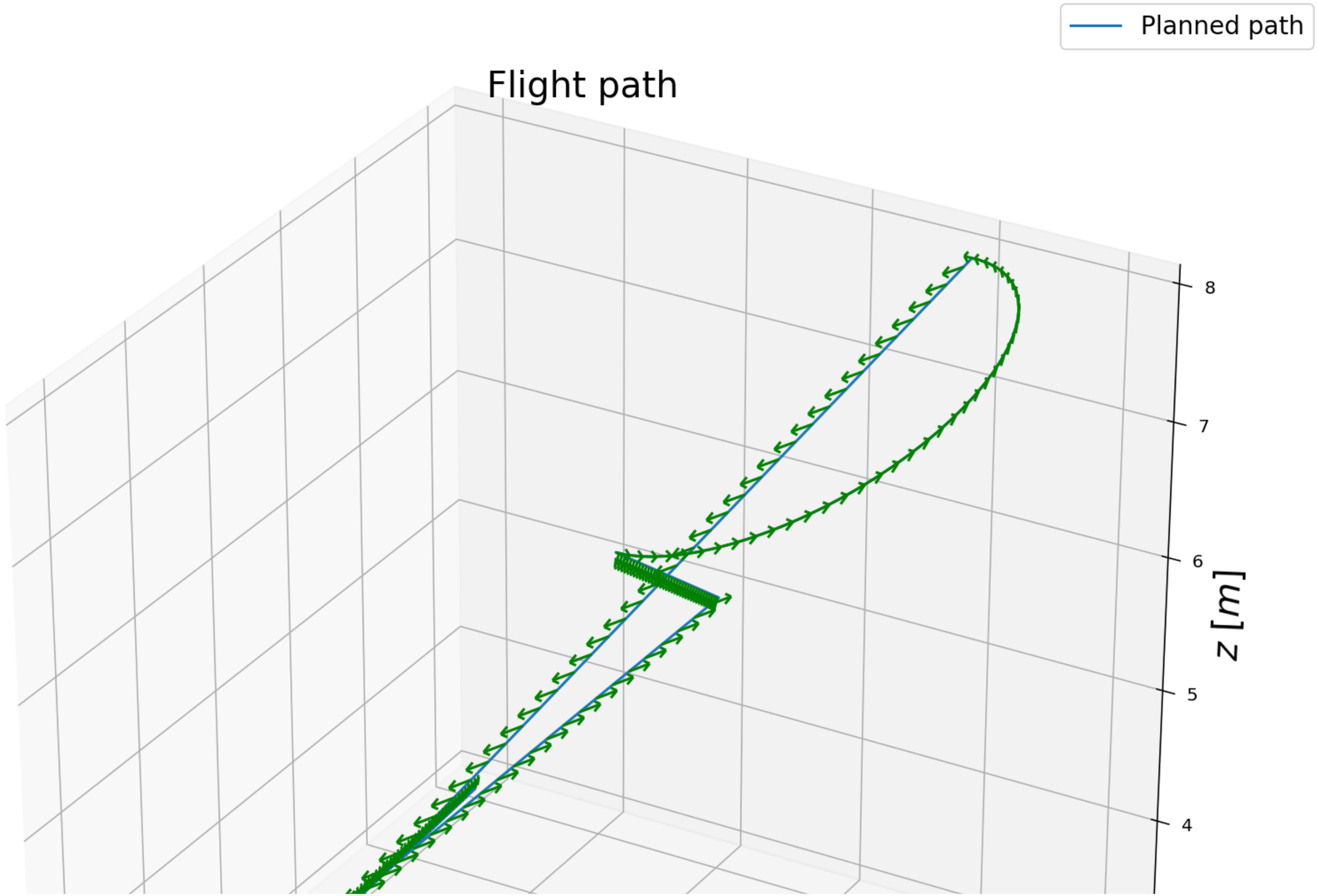
The drone will follow the trajectory specified by the file test_trajectory.txt. The video of the drone could be found here:

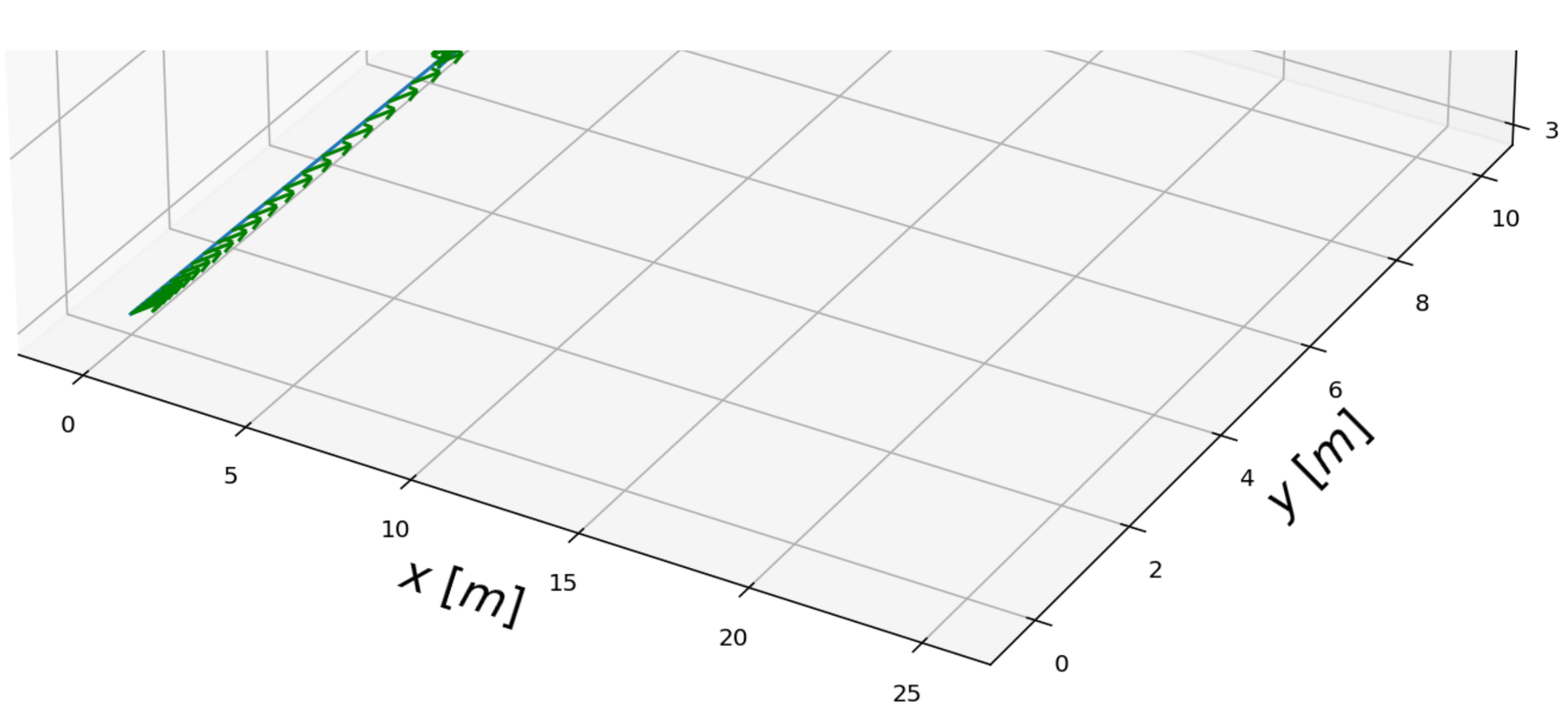


At the end of the execution, the code evaluates the performance based on the project acceptance criteria. You should see something similar to this:

```
python controls_flyer.py
Logs/TLog.txt
Logs/NavLog.txt
starting connection
arming transition
takeoff transition
landing transition
disarm transition
manual transition
Closing connection ...
Maximum Horizontal Error:  1.896498169249138
Maximum Vertical Error:  0.636822964449316
Mission Time:  2.121786
Mission Success:  True
```

The telemetry file for this particular execution is this file. To tune the controller is very hard on this trajectory. In the beginning, you don't know what to expect. The trajectory looks something like this:





In order to check the implementation and do some tuning of the parameters before testing them on the real trajectory, I generated the following trajectories:

- go_north_east
- go_north
- stay_there

To load one of this trajectories instead of test_trajectory.txt, uncomment line 210 on controls_flyer.py and set the desired trajectory file there. The generation of the trajectories was done by Test Trajectory Jupyter Notebook.

C++ implementation

This is the more complicated part of the project. If the parameter tuning on the python part was hard, this part is ten times harder. The C++ part is just a detail on this onerous task. In this case, the simulator is enforced more real limits to the implementation, and things can go wrong when some of those limits are not implemented correctly. More interesting than that is when things are not entirely wrong, just a bit. Udacity also provides a seed project with the simulator implementation and placeholders for the controller code. The seed project README.md give guides to run the project and information of the task we need to execute for implementing the controller. There are five scenarios we need to cover. The simulator runs in a loop on the current scenario and show on the standard output an indication the scenario pass or not.

All the C++ code is in the /cpp directory. The more interesting files are:

- /cpp/config/QuadControlParams.txt: This file contains the configuration for the controller. While the simulator is running, you can modify this file, and the simulator will "refresh" those parameters on the next loop execution.
- /cpp/src/QuadControl.cpp: This is where all the fun is, but I should not say this because this file contains the implementation of the controller only. Most of the time needed to pass the scenarios is spend on the parameter tuning.

Prerequisites

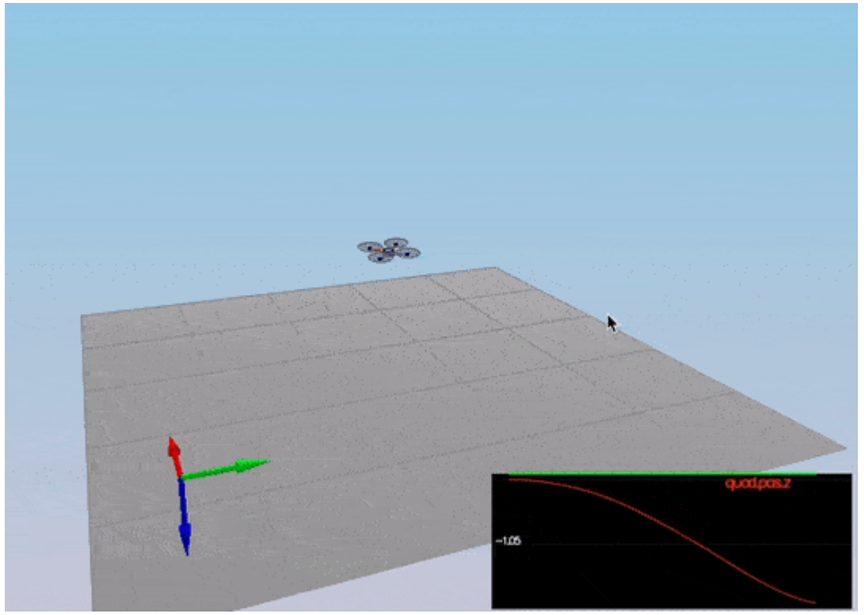
Nothing extra needs to install but the IDE is necessary to compile the code. In my case XCode because I am using a Macbook. Please, follow the instructions on the seed project README.md.

Run the code

Following the instruction on the seed project, load the project on the IDE. Remember the code is on /cpp.

Scenario 1: Intro

In this scenario, we adjust the mass of the drone in /cpp/config/QuadControlParams.txt until it hovers for a bit:



This video is cpp-scenario-1.mov

When the scenario is passing the test, you should see this line on the standard output:

```
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
```

Scenario 2: Body rate and roll/pitch control

Now is time to start coding. The GenerateMotorCommands method needs to be coded resolving this equations:

$$F_1 + F_4 - F_2 - F_3 = \tau_x / l \quad = t_1 \tag{1}$$

$$F_1 + F_2 - F_3 - F_4 = \tau_y / l \quad = t_2 \tag{2}$$

$$F_1 - F_2 + F_3 - F_4 = -\tau_z / \kappa \quad = t_3 \tag{3}$$

$$F_1 + F_2 + F_3 + F_4 = F_t \quad = t_4 \tag{4}$$

Where all the `F_1` to `F_4` are the motor's thrust, `tao(x,y,z)` are the moments on each direction, `F_t` is the total thrust, kappa is the drag/thrust ratio and `l` is the drone arm length over square root of two. These equations come from the classroom lectures. There are a couple of things to consider. For example, on NED coordinates the `z` axis is inverted that is why the moment on `z` was inverted here. Another observation while implementing this is that `F_3` and `F_4` are switched, e.g. `F_3` in the lectures is `F_4` on the simulator and the same for `F_4`.

The second step is to implement the BodyRateControl method applying a P controller and the moments of inertia. At this point, the `kpPQR` parameter has to be tuned to stop the drone from flipping, but first, some thrust needs to be commanded in the altitude control because we don't have thrust commanded on the `GenerateMotorCommands` anymore. A good value is `thurst = mass * CONST_GRAVITY`.

Once this is done, we move on to the RollPitchControl method. For this implementation, you need to apply a few equations. You need to apply a P controller to the elements `R13` and `R23` of the rotation matrix from body-frame accelerations and world frame accelerations:

$$\dot{b_c^x} = k_p(b_c^x - b_a^x) \tag{1}$$

$$\dot{b_c^y} = k_p(b_c^y - b_a^y) \tag{2}$$

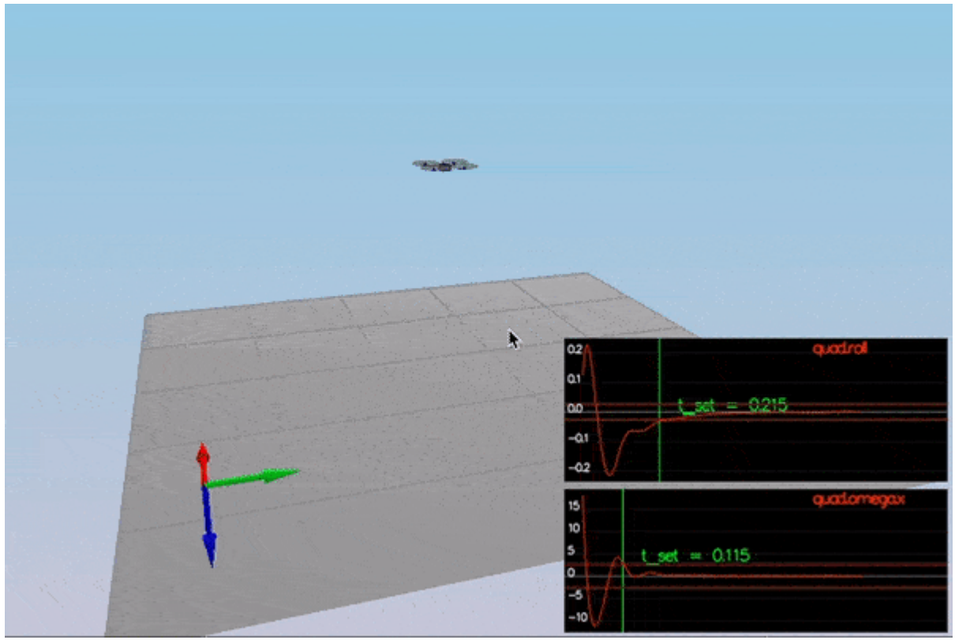
$$b_a^x = R_{13} \tag{3}$$

$$b_a^y = R_{23} \tag{4}$$

But the problem is you need to output roll and pitch rates; so, there is another equation to apply:

$$\begin{pmatrix} p_c \\ q_c \end{pmatrix} = \frac{1}{R_{33}} \begin{pmatrix} R_{21} & -R_{11} \\ R_{22} & -R_{12} \end{pmatrix} \times \begin{pmatrix} \dot{b_c^x} \\ \dot{b_c^y} \end{pmatrix} \tag{1}$$

It is important to notice you received thrust and thrust it need to be inverted and converted to acceleration before applying the equations. After the implementation is done, start tuning `kpBank` and `kpPQR` (again? yes, and it is not the last time) until the drone flies more or less stable upward:



This video is cpp-scenario-2.mov

When the scenario is passing the test, you should see this line on the standard output:

```
PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds
```

Scenario 3: Position/velocity and yaw angle control

There are three methods to implement here:

- AltitudeControl: This is a PD controller to control the acceleration meaning the thrust needed to control the altitude.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + R \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix} \tag{1}$$

$$b^x = R_{13} \tag{2}$$

$$b^y = R_{23} \tag{3}$$

$$b^z = R_{33} \tag{4}$$

$$\bar{u}_1 = \ddot{z} = cb^z + g \tag{5}$$

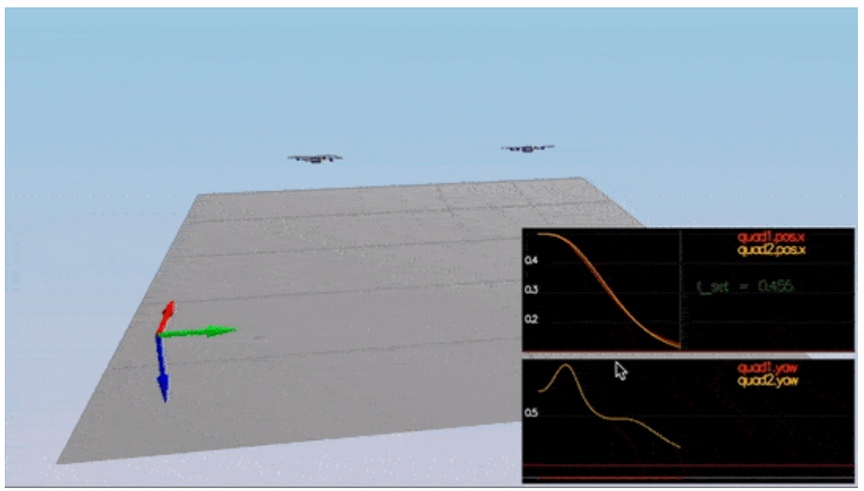
$$c = (\bar{u}_1 - g)/b^z \tag{6}$$

$$\bar{u}_1 = k_{p-z}(z_t - z_a) + k_{d-z}(\dot{z}_t - \dot{z}_a) + \ddot{z}_t \tag{7}$$

To test this, go back to scenario 2 and make sure the drone doesn't fall. In that scenario, the PID is configured not to act, and the thrust should be `mass * CONST_GRAVITY`.

- LateralPositionControl This is another PID controller to control acceleration on `x` and `y`.
- YawControl: This is a simpler case because it is P controller. It is better to optimize the yaw to be between `[-pi, pi]`.

Once all the code is implemented, put all the `kpYaw`, `kpPosXY`, `kpVelXY`, `kpPosZ` and `kpVelZ` to zero. Take a deep breath, and start tuning from the altitude controller to the yaw controller. It takes time. Here is a video of the scenario when it passes:



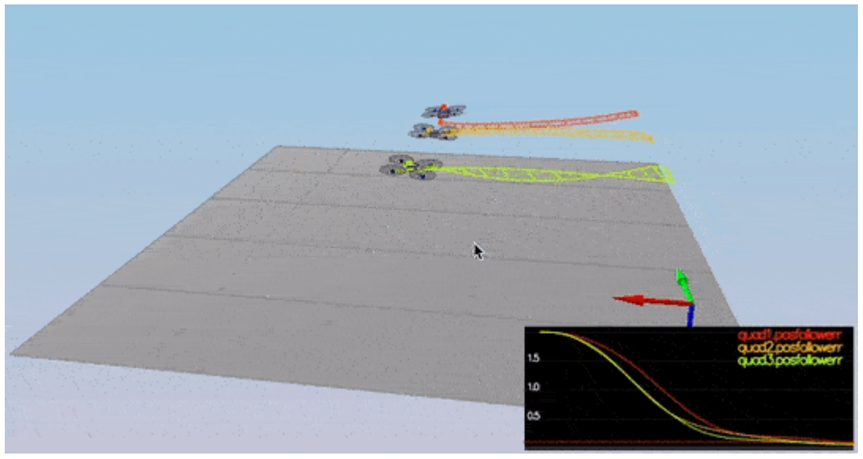
This video is cpp-scenario-3.mov

When the scenario is passing the test, you should see this line on the standard output:

```
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
```

Scenario 4: Non-idealities and robustness

This is a fun scenario. Everything is coded and tuned already, right? Ok, we need to add an integral part to the altitude controller to move it from PD to PID controller. What happens to me here is that everything starts not working correctly, and I have to tune everything again, starting from scenario -1. Remember patience is a “virtue”, and to it again. If you cannot and get frustrated talk to your peers, they will be able to give you hints. It is hard but doable:



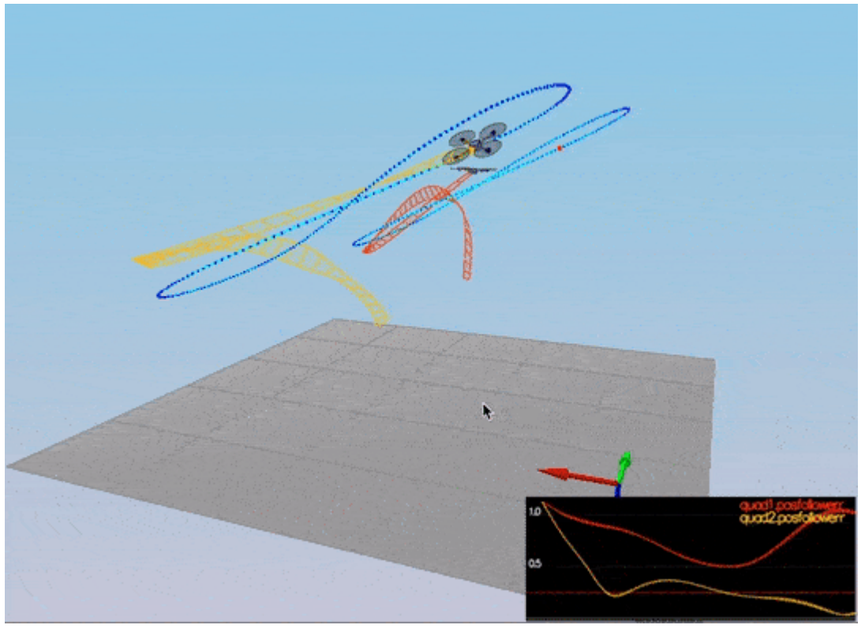
This video is cpp-scenario-4.mov

When the scenario is passing the test, you should see this line on the standard output:

```
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
```

Scenario 5: Tracking trajectories

This is the final non-optional scenario. The drone needs to follow a trajectory. It will show all the errors in your code and also force you to tune some parameters again. Remember there are comments on the controller methods regarding limits that need to be imposed on the system. Here those limits are required in order to pass.

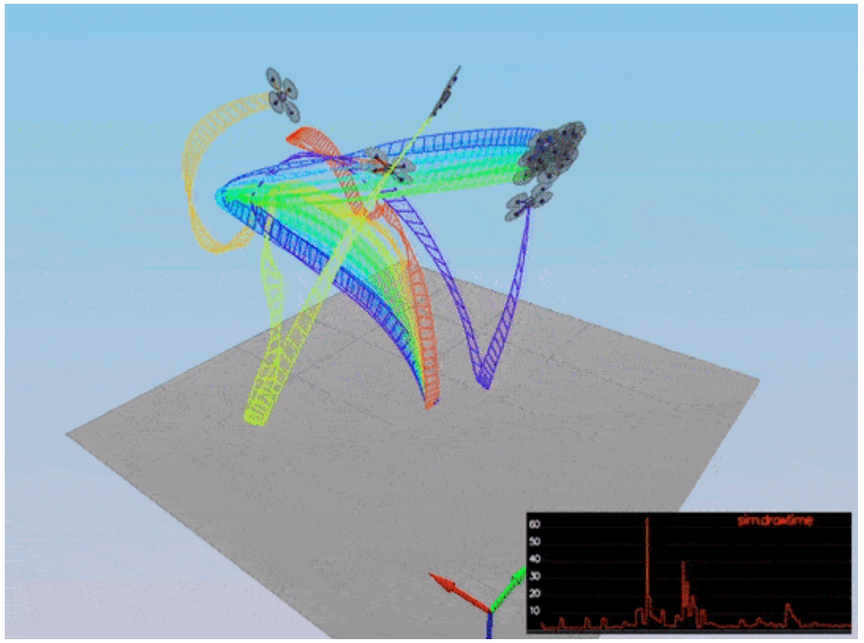


This video is cpp-scenario-5.mov

When the scenario is passing the test, you should see this line on the standard output:

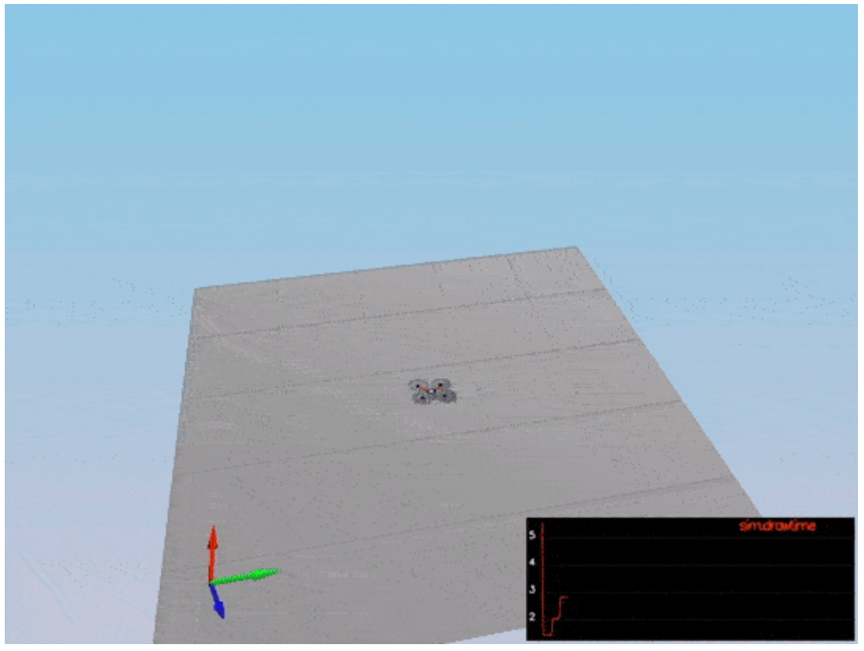
```
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
```

There are a few optional scenarios on this project, but I was exhausted. Too many long hours were tuning parameters and finding bugs. There should be a lot of room for improvement. Here is the video of a multi-drone scenario:



No idea why some of them go nuts!!!!! (and then come back to the “formation”.)

Post submit note The tilt angle limit enforcing was missing on the `RollPitchControl`. Here is a video with no-crazy drones:



Here is the video.

Project Rubric

Writeup

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your write-up as markdown or pdf.

This markdown is the write-up.

Implemented Controller

Implemented body rate control in python and C++.

The body rate control is implemented as proportional control in /python/controller.py body_rate_control method from line 179 to 195 using Python and in /cpp/src/QuadControl::BodyRateControl method from line 95 to 121 using C++.

Implement roll pitch control in python and C++.

The roll pitch control is implemented in /python/controller.py roll_pitch_controller method from line 142 to 177 using Python and in /cpp/src/QuadControl::RollPitchControl method from line 124 to 167 using C++.

Implement altitude control in python.

The altitude control is implemented in /python/controller.py altitude_control method from line 112 to 139 using Python.

Implement altitude controller in C++.

The altitude control is implemented in /cpp/src/QuadControl::AltitudeControl method from line 169 to 212 using C++.

Implement lateral position control in python and C++.

The lateral position control is implemented in /python/controller.py lateral_position_control method from line 93 to 124 using Python and in /cpp/src/QuadControl::LateralPositionControl method from line 215 to 267 using C++.

Implement yaw control in python and C++.

The yaw control is implemented in /python/controller.py yaw_control method from line 197 to 214 using Python and in /cpp/src/QuadControl::YawControl method from line 270 to 302 using C++.

Implement calculating the motor commands given commanded thrust and moments in C++.

The calculation implementation for the motor commands is in /cpp/src/QuadControl::GenerateMotorCommands method from line 58 to 93.

Flight Evaluation

Your python controller is successfully able to fly the provided test trajectory, meeting the minimum flight performance metrics.

> For this, your drone must pass the provided evaluation script with the default parameters. These metrics being, your drone flies the test trajectory faster than 20 seconds, the maximum horizontal error is less than 2 meters, and the maximum vertical error is less than 1 meter.

The Python implementation meets the minimum flight performance metrics:

```
Maximum Horizontal Error:  1.896498169249138
Maximum Vertical Error:  0.636822964449316
Mission Time:  2.121786
Mission Success:  True
```

Telemetry files are provided on the /python/telemetry directory.

Your C++ controller is successfully able to fly the provided test trajectory and visually passes the inspection of the scenarios leading up to the test trajectory.

> Ensure that in each scenario the drone looks stable and performs the required task. Specifically check that the student's controller is able to handle the non-linearities of scenario 4 (all three drones in the scenario should be able to perform the required task with the same control gains used).

The implementation pass scenarios 1 - 5 on the C++ simulator:

```
# Scenario 1
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
# Scenario 2
PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds
# Scenario 3
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
# Scenario 4
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
# Scenario 5
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
```

FCND-Term1-P3-3D-Quadrotor-Controller is maintained by darienmt.
This page was generated by GitHub Pages.