Control Project: Control a 3D Quadrotor

[1. Sebastian Introduction](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/89f2e38d-ae2d-4ebf-a8a5-39d53674ef56)

<https://www.youtube.com/watch?v=d1kpZ_LPMsM>

[2. Changes to this Project](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/5a57f117-d265-4f3a-9bea-d5221e587306)

# **Changes to this Project**

**As of May 4 2018, the Python portion of this project is no longer required. You will only be submitting the C++ portion for evaluation.**

If you began the project prior to this date, you are welcome to work on and submit the Python portion as well, but that section will not be evaluated by the project reviewer.

#### **Why are we no longer requiring this portion?**

We constantly listen to student feedback and do our best to improve the learning experience based on that feedback.

After analyzing the feedback from the first round of students who went through this project we learned that it was taking much longer to complete than we had anticipated.

There were a few reasons why this project was taking so long, but a lot of it had to do with debugging and tuning. Simply put: we didn't include enough support for incremental development and tuning of the individual controllers in the Python portion of the project. This meant that some students were spending hours and hours trying to tune their controllers (all without knowing whether their controller implementations were even correct). This is not a productive use of your valuable learning time!

The C++ portion of the project is broken into several smaller steps. Each of these steps provides some feedback on the correctness of your controller. You'll still have to deal with some of the pain of tuning, but it wouldn't be a controls project if that weren't the case.

[3. Project Introduction](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/20fede4d-0c7b-4f35-8d59-de12e40d55c8)

# **Project Introduction**

Welcome to the Controls Project!

At this point, you've implemented individual control loops in Python. In this project, you will be porting some of that logic over to a controller that's written in **C++**. This code will control a drone in an entirely new simulator. The simulator you'll be using in this project is more bare-bones than the Python / Unity simulator you've been working with so far, but it's more realistic in the physics that it models.

Once your controller meets the required specs with the C++ simulator, you'll have completed the project! And for you more hardware minded students, you'll also be ready for running your controller on a real drone!

Before you can dive into writing your controller, we'll first set up the environment you'll be needing for your development in a couple steps:

1. Set up the C++ simulator and the development environment you'll need for it.
2. Get familiar with some of the tools you'll be able to use to evaluate your controller's performance.

[4. [OPTIONAL] Python Setup](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/3c3d9484-cb07-42ca-b465-50f984e9bffe)

## **Note: this portion of the project is no longer required.**

We are leaving the Python setup instructions in place in case you're interested. All the instructions on this page are entirely optional.

### **Setup Instructions**

For the python part of the project, you will be modifying your *Backyard Flyer* project solution to handle the low level control loops of the drone, in addition to the higher level path and trajectory following control logic.

There are some modifications that are required to your backyard\_flyer.py solution to prepare it for the project, which we will walk through here. In addition to these changes, there is also a new simulator with some tools to help visualize your controller as you build it!

## **Setup Your Environment**

### **Step 1: download the simulator**

Even if you've already downloaded the simulator, download the [most recent version](https://github.com/udacity/FCND-Simulator-Releases/releases) that is appropriate for your OS.

### **Step 2: set up your python environment**

If you haven't already, set up your Python environment and get all the relevant packages installed using Anaconda following instructions in [this repository](https://github.com/udacity/FCND-Term1-Starter-Kit)

#### **Make sure Udacidrone is up to date**

Let's quickly make sure you have the most up to date version of udacidrone, which will allow you to use the full functionality of the controls simulator environment.

First make sure you have activated your environment:

source activate fcnd

Then run the update:

pip install -U git+https://github.com/udacity/udacidrone.git

### **Step 3: clone this repository**

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

### **Step 4: test setup**

Your starting point here will be the [solution code](https://github.com/udacity/FCND-Backyard-Flyer/blob/solution/backyard_flyer.py) for the Backyard Flyer project. Before you start modifying the code, make sure that your Backyard Flyer solution code works as expected and your drone can perform the square flight path in the new simulator. To do this, start the simulator and run the [backyard\_flyer.py](https://github.com/udacity/FCND-Backyard-Flyer/blob/solution/backyard_flyer.py) script.

source activate fcnd *# if you haven't already sourced your Python environment, do so now.*  
python backyard\_flyer.py

The quad should take off, fly a square pattern and land, just as in the previous project. If everything works then you are ready to move to the next step and modify backyard\_flyer.py to get it ready to use your custom controller.

## **Update Backyard Flyer Solution**

The following modifications need to be made to the solution backyard\_flyer.py. Feel free to use a copy of your own solution to the Backyard Flyer Project or the one in the link provided.

### **Step 1**

Import the UnityDrone and NonlinearController classes and modify the BackyardFlyer class to be a subclass of UnityDrone instead of Drone. UnityDrone is a subclass of Drone, so it provides all the functionality of Drone along with additional Unity specific commands/functionality (see below).

**from** unity\_drone **import** UnityDrone  
**from** controller **import** NonlinearController  
...  
**class** **BackyardFlyer**(UnityDrone):

### **Step 2**

Add a controller object in the \_\_init\_\_ method:

**def** **\_\_init\_\_**(self, connection):  
 ...  
 self.controller = NonlinearController()

### **Step 3**

Add the following three methods to your class to incorporate the controller into the backyard flyer.

**def** **position\_controller**(self):  
 """Sets the local acceleration target using the local position and local velocity"""  
  
 (self.local\_position\_target, self.local\_velocity\_target, yaw\_cmd) = self.controller.trajectory\_control(self.position\_trajectory, self.yaw\_trajectory, self.time\_trajectory, time.time())  
 self.attitude\_target = np.array((0.0, 0.0, yaw\_cmd))  
  
 acceleration\_cmd = self.controller.lateral\_position\_control(self.local\_position\_target[0:2], self.local\_velocity\_target[0:2], self.local\_position[0:2], self.local\_velocity[0:2])  
 self.local\_acceleration\_target = np.array([acceleration\_cmd[0], acceleration\_cmd[1], 0.0])  
  
**def** **attitude\_controller**(self):  
 """Sets the body rate target using the acceleration target and attitude"""  
 self.thrust\_cmd = self.controller.altitude\_control(-self.local\_position\_target[2], -self.local\_velocity\_target[2], -self.local\_position[2], -self.local\_velocity[2], self.attitude, 9.81)  
 roll\_pitch\_rate\_cmd = self.controller.roll\_pitch\_controller(self.local\_acceleration\_target[0:2], self.attitude, self.thrust\_cmd)  
 yawrate\_cmd = self.controller.yaw\_control(self.attitude\_target[2], self.attitude[2])  
 self.body\_rate\_target = np.array([roll\_pitch\_rate\_cmd[0], roll\_pitch\_rate\_cmd[1], yawrate\_cmd])  
  
**def** **bodyrate\_controller**(self):   
 """Commands a moment to the vehicle using the body rate target and body rates"""  
 moment\_cmd = self.controller.body\_rate\_control(self.body\_rate\_target, self.gyro\_raw)  
 self.cmd\_moment(moment\_cmd[0], moment\_cmd[1], moment\_cmd[2], self.thrust\_cmd)

### **Step 4**

Register and add callbacks for the RAW\_GYROSCOPE, ATTITUDE, and LOCAL\_VELOCITY messages. Note that you may already have the velocity\_callback() function implemented; if so, replace velocity\_callback() with the callback below. Call the appropriate level of control in each callback (i.e. bodyrate\_controller() is called in gyro\_callback()):

**def** **\_\_init\_\_\_**(self,connection):  
 ...  
 self.register\_callback(MsgID.ATTITUDE, self.attitude\_callback)  
 self.register\_callback(MsgID.RAW\_GYROSCOPE, self.gyro\_callback)  
 self.register\_callback(MsgID.LOCAL\_VELOCITY, self.velocity\_callback)  
  
**def** **attitude\_callback**(self):  
 ...  
 **if** self.flight\_state == States.WAYPOINT:  
 self.attitude\_controller()  
  
**def** **gyro\_callback**(self):  
 ...  
 **if** self.flight\_state == States.WAYPOINT:  
 self.bodyrate\_controller()  
  
**def** **velocity\_callback**(self):  
 ...  
 **if** self.flight\_state == States.WAYPOINT:  
 self.position\_controller()

### **Step 5**

In the waypoint transition method, replace the self.cmd\_position method (which is disabled by UnityDrone) with setting the target local position. Note: local\_position\_target should be in NED coordinates, the backyard\_flyer solution may calculate the box in NE altitude coordinates

*# replace this*  
self.cmd\_position(self.target\_position[0], self.target\_position[1], self.target\_position[2], 0.0)  
  
*# with this*  
self.local\_position\_target = np.array((self.target\_position[0], self.target\_position[1], self.target\_position[2]))

### **Step 6**

For this project we will no longer be flying the waypoint box, but rather a full flight trajectory, so remove calculate box and load the test trajectory:

*# replace this*  
self.all\_waypoints = self.calculate\_box()  
  
*# with this*  
(self.position\_trajectory, self.time\_trajectory, self.yaw\_trajectory) = self.load\_test\_trajectory(time\_mult=0.5)  
self.all\_waypoints = self.position\_trajectory.copy()  
self.waypoint\_number = -1

### **Step 7**

As our trajectory defines a waypoint with both time and location, change the transition criterion from proximity based to time based:

*# Replace this*  
**if** np.linalg.norm(self.target\_position[0:2] - self.local\_position[0:2]) < 1.0:  
 ...  
  
*# with this*  
**if** time.time() > self.time\_trajectory[self.waypoint\_number]:  
 ...  
...  
  
**def** **waypoint\_transition**(self):  
 ...  
 self.waypoint\_number = self.waypoint\_number+1

### **See what happens with no control**

Now your backyard\_flyer.py solution is ready to use your custom controller. Since you have yet to write any of the control functions, your quad will be incapable of flying, but just to make sure your script is working, start up the simulator and run your script:

python backyard\_flyer.py

If you've got everything set up properly, you should see your quad quite unceremoniously fall down to the ground!

Now you have everything you need ready to go for the Python portion of the controls project. Next, let's get everything set up for the C++ portion of the project.

### **Alternative to setting up your backyard\_flyer**

We have provided start code that takes the backyard\_flyer solution and adds the above modifications in the [controls\_flyer.py](https://github.com/udacity/FCND-Controls/blob/master/controls_flyer.py) script. Feel free to use that as the starting point, or your own script.

[5. C++ Setup](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/6671a450-8301-4c86-8a06-9a801cd22d45)

Regardless of your development platform, the first step is to download or clone the [C++ simulator repository](https://github.com/udacity/FCND-Controls-CPP.git).

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

Once you have the code for the simulator, you will need to install the necessary compiler and IDE necessary for running the simulator.

Below you can find the setup and install instructions for each of the recommended IDEs for each different OS options.

Once you have successfully installed the recommended IDE for your OS and have the simulator running, we can move on to taking a look around at the simulator and the files provided.

### **Windows**

For Windows, the recommended IDE is Visual Studio. Here are the steps required for getting the project up and running using Visual Studio.

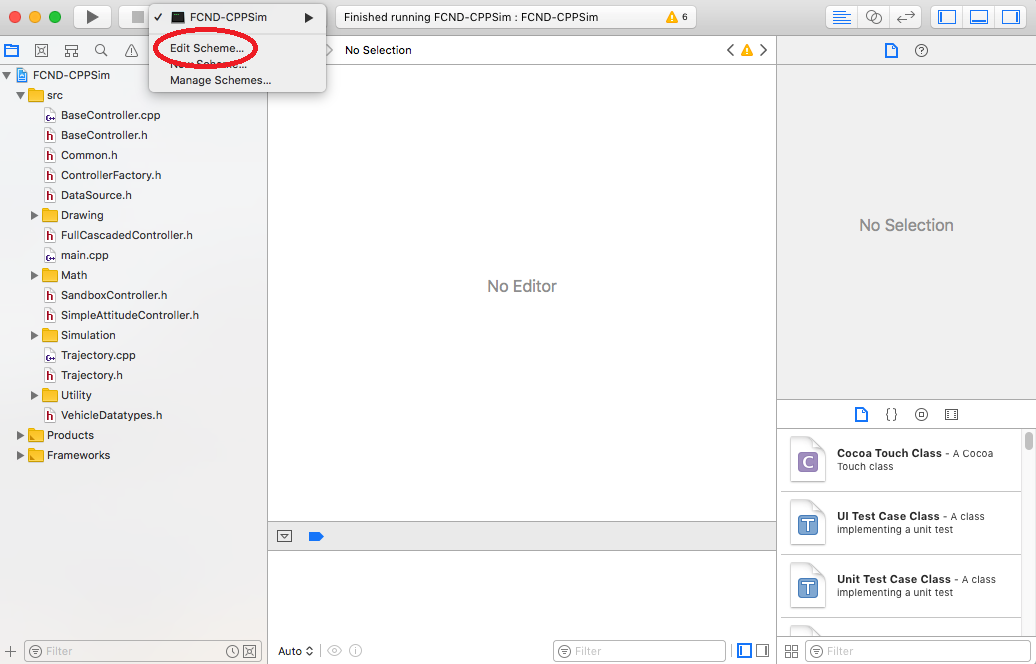
1. Download and install [Visual Studio](https://www.visualstudio.com/vs/community/)
2. Select *Open Project / Solution* and open <simulator>/Simulator.sln
3. From the *Project* menu, select the *Retarget solution* option and select the Windows SDK that is installed on your computer (this should have been installed when installing Visual Studio or upon opening of the project).
4. To compile and run the project / simulator, simply click on the green play button at the top of the screen. When you run the simulator, you should see a single quadcopter, falling down.

### **Mac OS X**

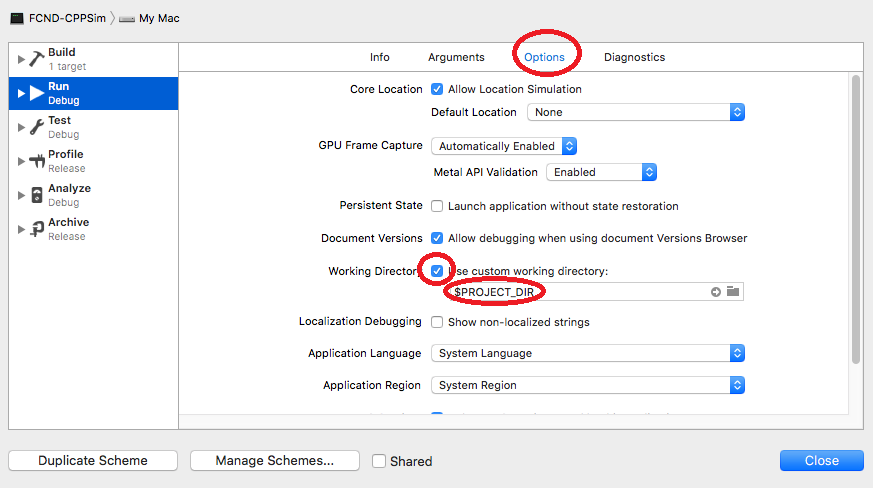
For Mac OS X, the recommended IDE is XCode, which you can get via the App Store.

1. Download and install XCode from the App Store if you don't already have it installed.
2. Open the project from the <simulator>/project directory.
3. After opening project, you need to set the working directory:

Go to *(Project Name)* | *Edit Scheme*

**

In new window, under *Run/Debug* on left side, under the *Options* tab, set Working Directory to $PROJECT\_DIR and check ‘use custom working directory’.



Finally, compile and run the project. You should see a single quadcopter, falling down.

### **Linux**

For Linux, the recommended IDE is QtCreator.

1. Download and install QtCreator.
2. Open the .pro file from the <simulator>/project directory.
3. Compile and run the project (using the tab Build select the qmake option. You should see a single quadcopter, falling down.

**NOTE:** You may need to install the GLUT libs using sudo apt-get install freeglut3-dev

[6. C++ Simulator](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/9579db79-f805-4f81-8ff6-8789709976cd)

Now that you have all the code on your computer and the simulator running, let's walk through some of the elements of the code and the simulator itself.

### **The Code**

For the project, the majority of your code will be written in src/QuadControl.cpp. This file contains all of the code for the controller that you will be developing.

All the configuration files for your controller and the vehicle are in the config directory. For example, for all your control gains and other desired tuning parameters, there is a config file called QuadControlParams.txt set up for you. An import note is that while the simulator is running, you can edit this file in real time and see the affects your changes have on the quad!

The syntax of the config files is as follows:

* [Quad] begins a parameter namespace. Any variable written afterwards becomes Quad.<variablename> in the source code.
* If not in a namespace, you can also write Quad.<variablename> directly.
* [Quad1 : Quad] means that the Quad1 namespace is created with a copy of all the variables of Quad. You can then overwrite those variables by specifying new values (e.g. Quad1.Mass to override the copied Quad.Mass). This is convenient for having default values.

You will also be using the simulator to fly some different trajectories to test out the performance of your C++ implementation of your controller. These trajectories, along with supporting code, are found in the traj directory of the repo.

### **The Simulator**

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

The simulation (including visualization) is implemented in a single thread. This is so that you can safely breakpoint code at any point and debug, without affecting any part of the simulation.

Due to deterministic timing and careful control over how the pseudo-random number generators are initialized and used, the simulation should be exactly repeatable. This means that any simulation with the same configuration should be exactly identical when run repeatedly or on different machines.

Vehicles are created and graphs are reset whenever a scenario is loaded. When a scenario is reset (due to an end condition such as time or user pressing the ‘R’ key), the config files are all re-read and state of the simulation/vehicles/graphs is reset -- however the number/name of vehicles and displayed graphs are left untouched.

When the simulation is running, you can use the arrow keys on your keyboard to impact forces on your drone to see how your controller reacts to outside forces being applied.

#### **Keyboard / Mouse Controls**

There are a handful of keyboard / mouse commands to help with the simulator itself, including applying external forces on your drone to see how your controllers reacts!

* Left drag - rotate
* X + left drag - pan
* Z + left drag - zoom
* arrow keys - apply external force
* C - clear all graphs
* R - reset simulation
* Space - pause simulation

### **Testing it Out**

When you run the simulator, you'll notice your 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.

Note: if you want to come back to this later, this scenario is "1\_Intro".

### **Almost There**

Now that you have the simulator and skeleton code up and running, you're almost ready to start writing your controller! One last thing to see is some of the tools we have provided you for evaluating the performance of your controller and some general tips and tricks for the project.

[7. Controller Performance Evaluation](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/b325e8a0-c153-42c6-9384-01088cdd34ff)

## **Evaluation Metrics**

This project is broken into a series of "scenarios".

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

There are two ways to view the output of the evaluation:

* in the command line, at the end of each simulation loop, a **PASS** or a **FAIL** for each metric being evaluated in that simulation
* on the plots, once your quad meets the metrics, you will see a green box appear on the plot notifying you of a **PASS**

### **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

[8. Tips and Tricks](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/b78ec22c-5afe-444b-8719-b390bd2b2988/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/concepts/74671657-bda5-4aff-a635-1fde48b15b33)

## **C++ Tips and Tricks**

Here are some tips and tricks you may find useful while writing your C++ code. Keeping in mind that the target platform for this code is a microcontroller, there are some additional factors you should consider:

* Use floats, don’t use doubles (in practice, floats will run much faster on many microcontrollers than doubles)
* Please don’t add additional library dependencies or allocate memory or use fancy STL features (you shouldn’t need to), if you keep your dependencies and code complexity down, the code you implement should easily port to real hardware. A real embedded environment is often stripped down to bare essentials, and even basic C++ STL features like <string> or <map> are often missing.

Functions you may find useful:

* Lots of the code uses a data type called V3F, this is a convenience class for working with vectors of 3 floats. For example, position/velocity/acceleration in 3D, body rates, etc.
* CONSTRAIN() is a function that allows you to constrain your value within a set of bounds. This is very helpful to have an upper and lower limit on a command (e.g. limiting a drone's bank angle).

# **Parameter Tuning Tips and Tricks**

Tuning the controller can be tricky. The parameters for your controller are defined in a file called QuadControlParams.txt. In that file you'll see the following initial values for the parameters

# Position control gains  
kpPosXY = 1  
kpPosZ = 1  
KiPosZ = 20  
  
# Velocity control gains  
kpVelXY = 4  
kpVelZ = 4  
  
# Angle control gains  
kpBank = 5  
kpYaw = 1  
  
# Angle rate gains  
kpPQR = 23, 23, 5

**Approximate parameter values**

These initial values are all **too low**. To get our "solution" controller properly tuned we needed values that were between 2 and 4 times larger than the values shown here.

**Approximate parameter ratios**

Sergei Lupashin put together a [one-page document](https://www.overleaf.com/read/bgrkghpggnyc#/61023787/) deriving the ratio of velocity gains to position gains for a "critically damped" system (where the damping ratio is 1). In that document he shows that the ratio K\_v / K\_p = 4*Kv*​/*Kp*​=4 for such a system.

**Sequencing**

The C++ project is organized as a sequence of scenarios. Each scenario will have you implement one or two controllers and tune the associated parameters. If you can, try to keep the number of parameters that you're tuning at any given time small! For example, once you've tuned kpPQR and then moved on to the next controller, try to avoid further changes to kpPQR.

Dronecode provides some good [guidelines on PID tuning](https://docs.px4.io/en/advanced_config/pid_tuning_guide_multicopter.html#pid-controller-overview).

### **Acknowledgement**

The C++ simulator and project scenarios were largely designed and built by [Fotokite](https://fotokite.com/). Big thanks to the Fotokite team (and Sergei Lupashin in particular) for their great work!

[9. Project: Control of a 3D Quadrotor](https://classroom.udacity.com/nanodegrees/nd787/parts/3619d672-ce5b-4a29-8a15-172cb667b5bb/modules/undefined/lessons/81acb1ca-af00-4553-b8bb-b1467fbcfc51/project)

## Project Submission

## **The Goal of this Project**

In the real world the flight controller is usually implemented in C or C++. So in this project you will implement your controller in C++. The code you write here can eventually be transferred to a real drone!

## **Project Steps**

This project originally had two parts. In the first part we asked students to implement a controller in Python. We've since removed that portion of the project, but you may find the [solution implementation](https://github.com/udacity/FCND-Controls/blob/solution/controller.py)helpful to consult as a reference.

1. Make sure you have cloned the repository and gotten familiar with the C++ environment as outlined in **C++ Setup**.
2. Complete each of the scenarios outlined in the [C++ project readme](https://github.com/udacity/FCND-Controls-CPP). This will involve implementing and tuning controllers incrementally:
   * Body rate and roll/pitch control (scenario 2)
   * Position/velocity and yaw angle control (scenario 3)
   * Non-idealities and robustness (scenario 4)
3. Tune your controller and make sure it works to successfully meet each of the evaluations in each scenario.

For more detailed instructions on the individual steps, make sure to read through the [C++ project readme](https://github.com/udacity/FCND-Controls-CPP), specifically the [section outlining the tasks for the project](https://github.com/udacity/FCND-Controls-CPP#the-tasks).

## **Evaluation**

For the submission, you will be evaluated on your completion of the five control functions, the motor command functions, and on your controller's performance in each scenario. Performance metrics are provided for each of the different scenarios, and your controller will need to meet these minimum performance metrics for each scenario. For the specific metrics for each scenario look at the [evaluation portion of the C++ readme](https://github.com/udacity/FCND-Controls-CPP#evaluation).

## **Submission**

For this project you will need to submit:

1. QuadController.cpp and QuadControlParams.txt, containing your completed C++ controller and associated gains.
2. A writeup addressing all the points of the [project rubric](https://review.udacity.com/#!/rubrics/1643/view)

## **Feedback**

Please fill out the [Control of a 3D Quadrotor Feedback Form](https://goo.gl/forms/Zy6HjhlTvIOVgkjJ3) after you have completed the project.

## **Share Your Project**

Have something fun to share? Controller not listening and teetering away? Or maybe its going above and beyond with synchronized dancing? We love to see any and all of your creations!

What you’ve accomplished is no small feat. Give yourself a pat on the back and some well-deserved recognition by sharing your amazing project with your network.

Create a gif or youtube video of your project implementation in the simulator and share with us in the [Flying Car Slack Community](https://udacity-flying-car.slack.com/messages/C77GCKZ8U) and on social media using the links below!

Make sure to use **@Udacity**, **#flyingcarnd**, **#quadcontrol**, and **#madewithudacity** in your posts!

Enter the full URL of your Gif or Youtube video:

**Share on Facebook**

**Share on LinkedIn**

You have not submitted the project yetSUBMIT PROJECT