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 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

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 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 script.

```
source activate fcnd # if you haven't already sourced your Python environment, do so i
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):
```

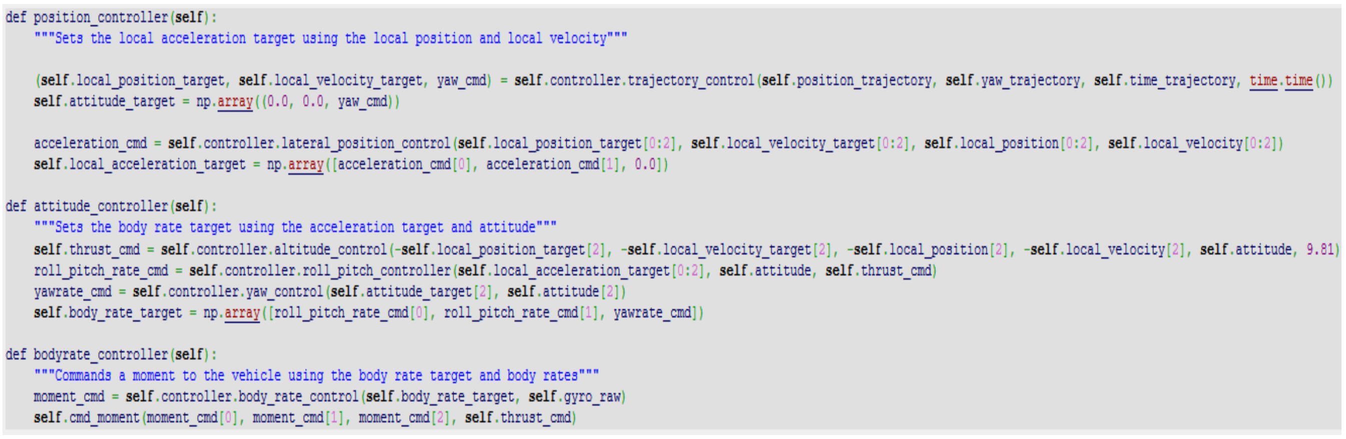
Step 2

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)
```

```
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` script. Feel free to use that as the starting point, or your own script.