CS 7638: Artificial Intelligence for Robotics

Drone Control (PID) Project

Spring 2023 - Deadline: Mon Mar 13th, 11:59pm AOE

Quick Instructions

To run all test cases from command line, use:

python DualRotor_TestSuite.py

To run a single test case, use:

python -m unittest DualRotor_TestSuite.Part_1_a_TestCase.test_case01

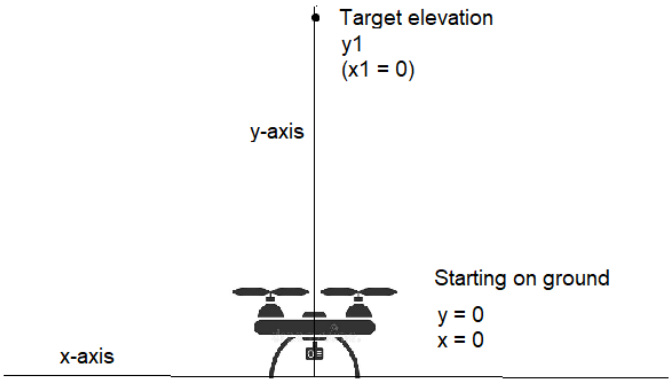
(replace "test_case01" with the test case number you want to run)

Project Description

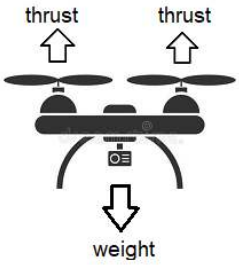
Autonomous drones are used to maintain critical infrastructure, e.g., inspect gas pipelines for leaks. In this project you will implement a PID controller for an autonomous drone to fly to a target elevation and horizontal position and hover at some target location for a specified time.

A PID controller is a component of a feedback control system. The system (usually called "plant") is supposed to be maintained at a target "steady" state. For example, a car that needs to stay in the middle of a lane, or a thermostat that needs to maintain a particular temperature, etc. The Drone starts on the ground with its rotors off. For simplicity we only consider a two dimensional world, and a dual rotor drone. We also don't consider any ground effects (https://www.rchelicopterfun.com/ground-effect.html) while simulating the drone.

Consider the simple case first, where the Drone only has to lift off vertically and achieve a target elevation and hover there.

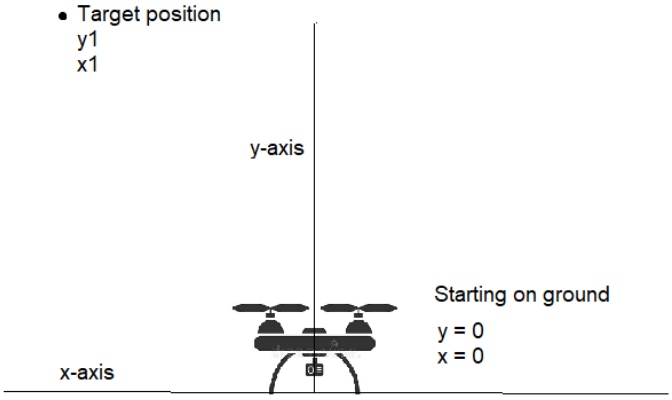


We have to induce a certain rpm (rotations per minute) equally in both the rotors. As the rpm of its rotors increase, they push the air down and this exerts a net upward force (thrust) on the Drone.

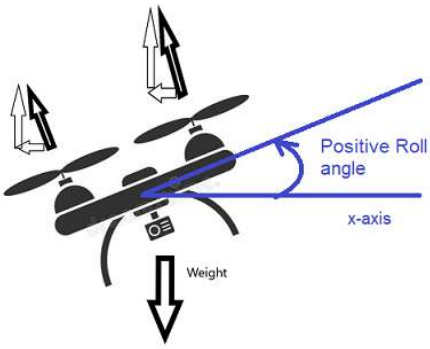


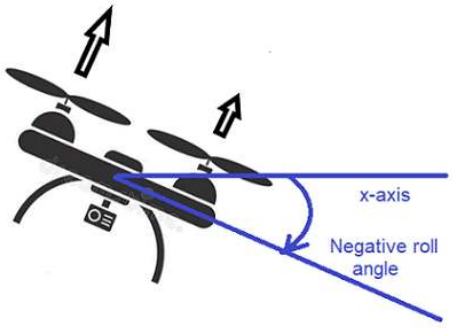
When this force equals the weight of the Drone, it starts to hover. Increasing the rpm beyond this point will lift the Drone up. To reach a certain elevation within some limited time, the thrust has to be sufficiently high. But as the drone nears the target, we have to start reducing the thrust until it is back equal to its weight ideally right at the target elevation. There, we have to maintain this thrust in order for it to hover for the desired period.

Now let's consider the situation where we also want to move the drone horizontally, besides vertically.



For this we need to tilt the drone sideways, which is called Roll (measured in angle). This is done by varying the thrust (or rpm) between the rotors of the Drone. In a Dual Rotor Drone, if thrust on the right side is higher, the Drone will roll left. If thrust on the left side is higher, then the Drone will roll right.





When the drone is at rest, it has a roll angle of zero. A positive roll angle corresponds to a leftward tilt, and a negative roll angle corresponds to a rightward tilt.

Once the drone rolls in the appropriate direction, the horizontal component of thrust moves the drone in the horizontal direction. The Drone needs to have enough roll to reach the target x1 location in a specific timeframe (specified by the test case). Note that the vertical component of thrust in each rotor keeps the drone levitating. Combined, they should equal or exceed the weight of the drone to keep it hovering, or lift it up to the desired height.

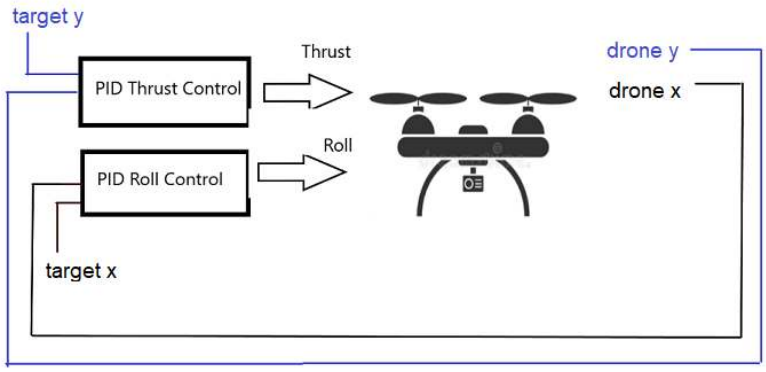
As it reaches close to the target x1, we need to reduce the roll until it reaches 0 degree, ideally exactly at the target. Then it has to hover there for a specified time. Note that in order to roll and move side-ways, the Drone needs to be at some elevation, so roll is applied along with some thrust.

We only need to return a single thrust and roll angle from the PID controller (actually, a change in thrust and roll angle), and the simulator will take care of inducing the appropriate rpms in the rotors. If the roll angle is zero, the thrust is going to be equally distributed. Otherwise, it will be distributed according to the roll angle.

You will implement the PID controllers for thrust and roll in the file drone_pid.py. You will be provided a target elevation for the PID Thrust controller, and a target horizontal position for the PID Roll controller. You will also implement the function “find_parameters” to find and tune the gain values for your controllers. Here you can implement the twiddle algorithm (or any other method if you like). The find_parameters method will be passed a “run_callback” function, which you will call
passing in your PID gain parameters to simulate the drone. The run_callback function is the run() function in DroneSimulator. It will take in your PID parameters and invoke your PID control functions to move the drone.

As a reminder, the PID control value (alpha) is calculated by the following formula:

alpha = tau_p * error + tau_i * error_integral + tau_d * error_derivative



Files

These are some key files of the project that you would need to know about:

a. DualRotor_TestSuite.py: This file contains the test cases, and is your starting point to run the simulation. To run a single test case:

python -m unittest DualRotor_TestSuite.Part_1_a_TestCase.test_case01

To run all test cases:

python DualRotor_TestSuite.py

You can also change various parameters in the file (e.g. VISUALIZE, DEBUG, TWIDDLE). For example, you might want to turn off TWIDDLE to run a test with manually supplied PID gain values to test your code. Each test case in this file do the following logical sequence of work:

1. Initialize DroneSimulator
2. Call find_parameters() in drone_pid.py (this will be your implementation of twiddle)
3. Call DroneSimulator.run(), with the PID values found from the step above.
4. Calculate your score.

b. DroneSimulator.py: This file contains the code to run the simulation of the Drone. The initialize() method is called by the test suite. The run() method executes the following high-level logic:

1. Do the following in a loop for the specified timesteps:

i. Call your pid_thrust() and pid_roll() implementations in drone_pid.py file, passing in the target coordinates and the Drone's current coordinates.
ii. Call DualRotor.move(), passing in the thrust and roll values retrieved from the above step.

2. Calculate error values used by DualRotor_TestSuite to calculate a score.

Each iteration of the loop represents one timestep and is assumed to be 1/10th of a second (the actual frequency of a real hardware-in-a-loop system could vary depending on the actuators involved).The run() method will also be passed to your implementation of twiddle in find_parameters_XXX() function of drone_pid.py. Your twiddle code will call this run() method to simulate the drone flight and tune the PID gain values.

c. drone_pid.py: This is the only file that you need to put your code in. The descriptions of the functions you need to implement are given below:

1. pid_thrust - In this function, you have to implement the code for a PID controller for Thrust. It is important to note that what you return from this function is the change in thrust, not a target thrust. The function is called from DroneSimulator.run() in a loop. At every step, there is a max change in rpm that can be applied by the Drone. This can be a positive or negative change. If the value returned by your implementation is going to cause the rpm to change outside of the max possible change in rpm per timestep, it is clipped at the max (or min if negative).
2. pid_roll - In this function, you have to implement the code for a PID controller for Roll. The return value is the change in roll angle. The function is called in a loop from DroneSimulator.run(). The change in roll at each step is also constrained by the max and min change in rpm at each rotor. For the purpose of this assignment, there is a max roll angle that the Drone can't go beyond, and any change beyond this value will be ignored. As roll can't be applied without some elevation, this function is always called along with the pid_thrust function by the simulator.
3. find_parameters_XXX - In this function, you will implement the twiddle algorithm to find gain values for your controllers. You will be passed a reference to the DroneSimulator.run() method, which you will call with the thrust and roll PID gain values that you want to test from your twiddle algorithm. For simplicity, this assignment provides 3 find_parameter() functions, one only for thrust, another one for thrust and roll, and a third one for Integral gain (with thrust).

Submitting your Assignment

Your submission will consist of the drone_pid.py file (only) which will be uploaded to Canvas->GradeScope. Do not archive (zip,tar,etc) it. Your code must be valid python version 3 code.

We ask that you keep any testing code in a separate file that you do not submit. Your code should also NOT display a GUI or Visualization when we import or call your function under test.

Calculating your score

The max score for the entire project is 100. There are 8 test cases in total - these are defined in the file test_cases.txt. Each test case will have a continuous score upto the maximum points for that test case. Your score is determined by how well the drone meets the requirements of hovering, velocity and oscillations. The test case descriptions and point distributions are as follows:

Test cases 1, 2, 3 (30 points each, total 90 points): These test cases only require that your drone reaches a target elevation, i.e., no horizontal movement is required. So it will only test your Thrust PID controller, and your twiddle algorithm's ability to find good gain values for that.

Test case 4 (2 points): This test case tests for elevation (Thrust PID) only, and specifically tests for the Integral part of your controller. It simulates an error in the drone's rotors so that they lose a certain RPM at every time step. Therefore, an Integral gain will most likely be needed to reach the target. In addition, during the final run of the simulator (for calculating the score), the program simulates an additional force pulling down on the drone for part of the time window. You may also need to handle Integral Wind-up and Control Saturation in this case.

Test cases 5, 6, 7, 8 (2 points each, total 8 points): These test cases require both an elevation as well as a horizontal movement. So they will test both your Thrust PID controller, as well as your Roll PID controller. Consequently, your twiddle algorithm will need to find gain values for both Thurst and Roll PID controllers.

To understand what is contained in test_cases.txt, here is the breakdown of testcase_1 as an example:

"testcase_1": { "path": [[0,0], [0,5]], "target_time": [0, 15], "hover_time": [0, 5], "max_velocity": 1.5, "max_oscillations": 1, "score_weight": 0.30 },

- testcase_1: The name of the test case. This is loaded by the corresponding test_caseXX() method in DualRotor_TestSuite.py.
- path: Defines the target path that the drone needs to follow in (x,y) coordinates. In the above example, the path starts from coordinates (0,0) and go to (0,5) (i.e., straight upward). Note that the drone's

starting position is always (0,0). The path can consist of multiple segments. For simplicity, each segment is either straight vertical or straight horizontal. Vertical can be up or down. Horizontal can be right or left.

- target_time: Maximum number of seconds in which the drone should reach the corresponding target coordinates. In the above example, the Drone should reach (0,5) in 15 seconds. (The Drone always starts at (0,0) so the target time to the first target of (0,0) is 0 seconds). Recall from above that each iteration of the loop in DroneSimulator.run() is 1/10th of a second. So 15 seconds translates to 150 iterations (time steps) of the loop, and that's also what you will see in the visualizations.
- hover_time: Number of seconds after the target_time that the Drone should hover at the target coordinates. In the above example, it doesn't need to hover at (0,0) (since that's the starting position), and should hover for 5 seconds at (0,5) from the 16th to the 20th second. If the drone reaches (0,5) before the 16th second, it still should hover at (0,5) till the 20th second (so hover time can be more than 5 seconds, but not less). (Note that like the target_time, 5 seconds of hover time also appear as 50 timesteps in the visualizations).
- max_velocity: The max velocity in meters/sec that the Drone should have at any point along its path. In the above case, it is set at 1.5 meters/sec. This is another way to control how fast the drone should reach a target location.
- max_oscillations: The maximum number of oscillations that your controller is allowed around the target. In the above example, it is 1, which means in flying from (0,0) to (0,5) (which is straight up), it is acceptable if the drone goes slightly above (0,5) before coming back down to it, as long as it happens only once. If after this, it goes down below (0,5), and then comes back up to hover at (0,5), that is counted as a second oscillation which will incur a penalty.
- score_weight: This is the weight of the test case out of a total of 100 (this should match the above scoring description).

You will also see a few other attributes for testcase_4:

- rpm_error: Number of RPMs that the drone loses at every time step. This is used for simulating a systemic error in the drone.
- extra_load: Mass of the extra load that the program simulates the drone is carrying while running the scoring run of simulator.run(). This is not used during twiddle.
- extra_load_carry_time: Length of time in seconds that the drone carries the above mentioned extra load.

Control Saturation and Integral Wind-up

In a feedback control system, control saturation occurs when an actuator (e.g. motor or steering angle) has reached its limit, but the steady state (i.e. the desired target, like target x, y position for drone) has not been reached. So the controller keeps increasing the output of the actuator. This in itself may not be a problem because many real systems will simply ignore the additional commands. However the problem occurs if there is an integral term in the controller. It will continue to accumulate more and more error (called Integral windup), as it is not able to release any of this error because the actuator is operating at its limit. When the system reaches the steady state, the error for P term is zero which will try to lower the actuator output. However, now the Integral term begins to unwind and it will continue to move the actuator in the same direction and take the system beyond the steady state. To handle this situation, you need to check if the control output is saturated and handle the integral term appropriately. In this assignment, we pass you a flag to indicate if control is saturated, but you will need to check for it in your code and handle the integral term appropriately.

Testing Your Code

NOTE: The test cases in this project are subject to change.

We have provided a testing suite similar to the one we'll be using for grading the project, which you can use to ensureyourcodeisworkingcorrectly.

We are using the Canvas->GradeScope autograder system which allows you to upload and grade your assignment with a remote / online autograder. You must submit your `drone_pid.py` file to Gradescope to receivecredit.

We **are likely, but not required, to use the last grade you receive, (or your selected grade) via the Canvas->GradeScope autograder as your final grade** at our discretion. (See the "Online Grading" sectionoftheSyllabus.)