Write up

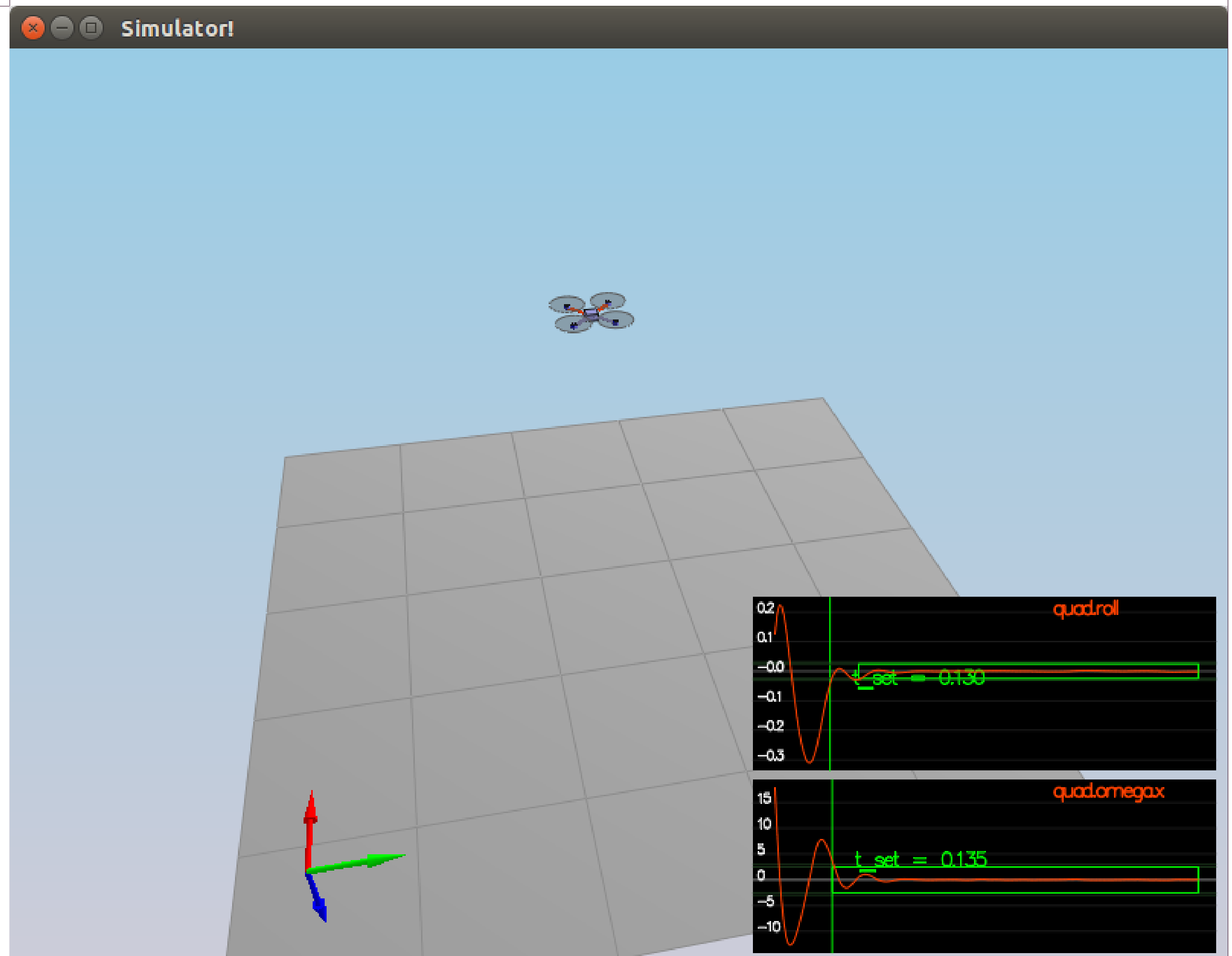
CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

Implemented body rate control in C++.	The controller should be a proportional controller on body rates to commanded moments. The controller should take into account the moments of inertia of the drone when calculating the commanded moments.	The BodyRateControl function is implemented using the following equations
		`p_error = p_target - p_actual`
		`u_bar_p = k_p_p * p_error`
		`and tau_x = Ix * u_bar_p`
		The code implemented in lines QuadControl.cpp:112..114
		These equations are carefully either derived from the exiting formulas or the translated from python implementations during exercises.
		Tuned the kpPQR parameter to make sure that the spinning is stopped quickly.

Implement roll pitch control in C++.	The controller should use the acceleration and thrust commands, in addition to the vehicle attitude to output a body rate command. The controller should	The following formulas are used to implement the roll-pitch control.
		`We have the formulas;`
		`b_x_c_dot = k_p * ( b_x_c - b_x_a), where b_x_a = R13`
		`b_y_c_dot = k_p * ( b_y_c - b_y_a), where b_y_a = R23`
		`Also, the angular velocities (p_c, q_c) = (1/R33) * ((R21, -R11), (R22, -R12)) * (b_x_c_dot, b_y_c_dot)`

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

	account for the non-linear transformation from local accelerations to body rates. Note that the drone's mass should be accounted for when calculating the target angles.	`This gives us;`
		
		`p_c = (1/R33) * (R21 * b_x_c_dot - R11 * b_y_c_dot)`
		
		`=>`
		
		`p_c = (1/R33) * (R21 * k_p * (b_x_c - R13) - R11 * k_p * (b_y_c - R23))`
		
		`q_c = (1/R33) * (R22 * b_x_c_dot - R12 * b_y_c_dot)`
		
		`=>`
		
		`q_c = (1/R33) * (R22 * k_p * (b_x_c - R13) - R12 * k_p * (b_y_c - R23))`
		
		`We also know;`
		
		`c = F/m`
		
		`p_c and q_c are the roll and putch rates`
		
		`So, pqrCmd.x = p_c; pqrCmd.y = q_c; and we will set pqrCmd.z = 0;`
		
		The code is implemented in lines under QuadControl.cpp:145..155.
		
		Tuned the kpBank parameter to make sure that both the roll and omega are passing the unit tests.
		
		Simulation #245 (../config/2_AttitudeControl.txt)
		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
		Simulation #246 (../config/2_AttitudeControl.txt)
		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
		The graphs end with green boxes as shown below in the screen captures.

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS
		

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

Implement altitude controller in C++.	The controller should use both the down position and the down velocity to command thrust. Ensure that the output value is indeed thrust (the drone's mass needs to be accounted for) and that the thrust includes the non-linear effects from non-zero roll/pitch angles.
	
	Additionally, the C++ altitude controller should contain an integrator to handle the weight non-idealities presented in scenario 4.
		We use the following formulas to implement the altitude controller
		
		`We have formulas;`
		
		`c = (u_1_bar - g)/b_z, where b_z = R33;`
		`u_1_bar = k_p_z * (z_target - z_atual) * k_d_z(z_dot_target - z_dot_zctual) + z_dot_dot_target`
		`and F = c * m`
		
		`using these formulas and integrating the altitude error we can code as below`
		
		The code in implemented in lines QuadControl.cpp:186..190

Implement lateral position control in C++.	The controller should use the local NE position and velocity	We use the following formulas to code the lateral position control function `From the python implementation in the exercises we have`

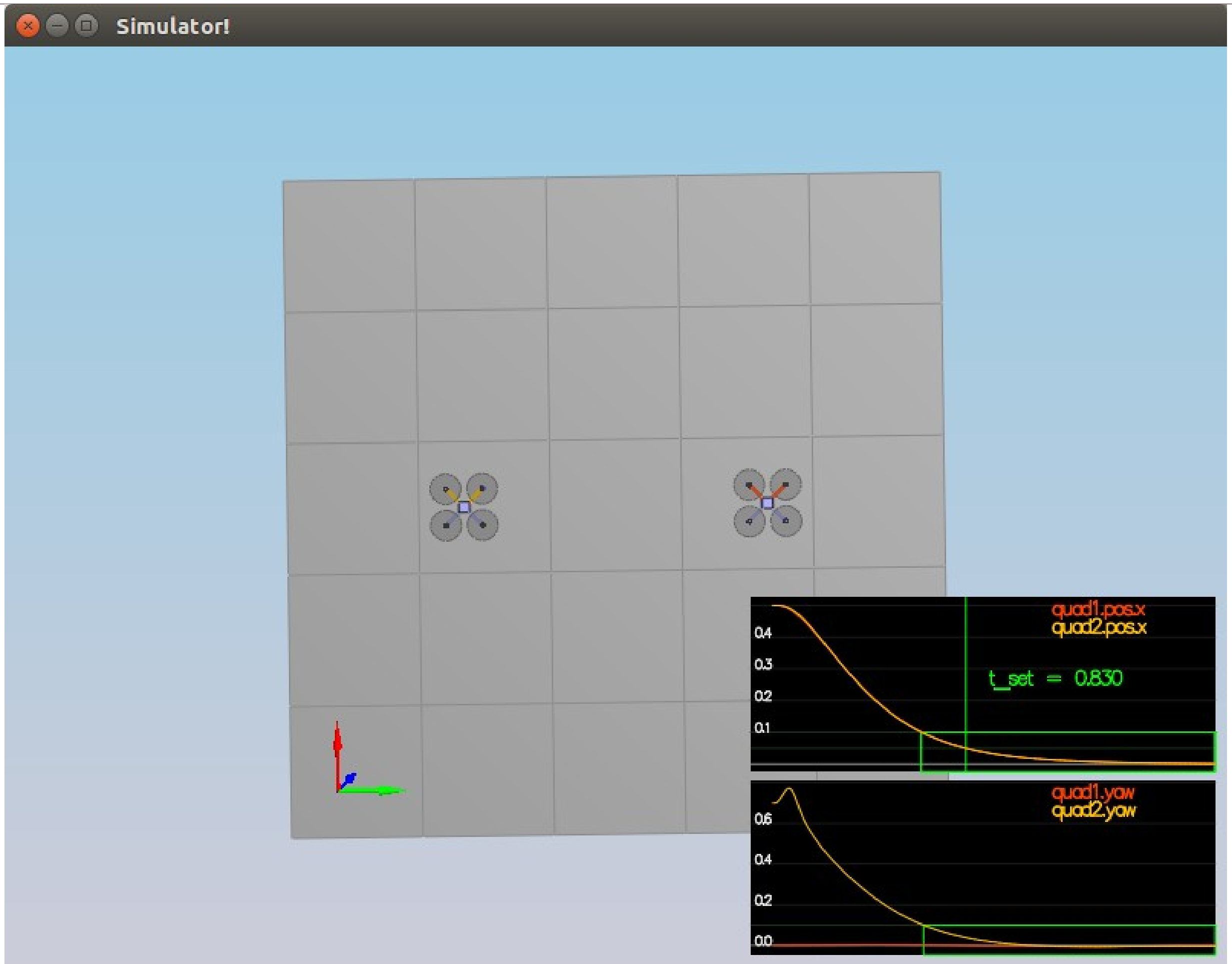
CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

	to generate a commanded local acceleration.	`x_dot_dot_command = x_k_p * (x_target - x_actual) + x_k_d * (x_dot_target - x_dot_actual) + x_dot_dot_target`
		
		`realigning the inputs here as:`
		
		`x_dot_dot_target = accelCmdFF;`
		`x_target = posCmd`
		`x_actual = pos`
		`x_dot_target = velCmd`
		`x_dot_actual = vel`
		
		`and limiting the maximum horizontal velocity and acceleration to maxSpeedXY and maxAccelXY as per requirements, we can code.`
		
		The code in implemented in lines QuadControl.cpp:228..239

Implement yaw control in C++.	The controller can be a linear/proportional heading controller to yaw rate commands (non-linear transformation not required).	`Using the equation;`
		
		`r_c = k_p_yaw * ( psi_target – psi_actual`
		
		`we  code YAW CONTROL function.`
		
		The code in implemented in lines QuadControl.cpp:228..239
		
		Tuned the following parameters after implementing the body rate control, lateral position control and yaw control
		
		`# Position control gains`
		`kpPosXY = 2`
		`kpPosZ = 2`
		`KiPosZ = 40`
		`# Velocity control gains`
		`kpVelXY = 6`

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

		`kpVelZ = 8`
		
		`# Angle control gains`
		`kpBank = 16`
		`kpYaw = 2`
		
		The two drones are stable and converge well as expected.
		
		Please observe the unit tests pass as below.
		
		Simulation #4031 (../config/3_PositionControl.txt)
		
		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
		
		The following is the screen capture of the drone and the graphs.

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS
		
Implement calculating the	The thrust and moments should be	Used the following explanation to implement the GenerateMotorCommands

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

motor commands given commanded thrust and moments in C++.	converted to the appropriate 4 different desired thrust forces for the moments. Ensure that the dimensions of the drone are properly accounted for when calculating thrust from moments.	<pre>Equations for generating the desiredThrusts
		
		We know;
		
		tau_x = (F1 - F2 - F3 + F4) * l
		tau_y = (F1 + F2 - F3 - F4) * l
		tau_z = tau_1 + tau_2 + tau_3 + tau_4
		F = F1 + F2 + F3 + F4
		
		 We need to solve these for F1, F2, F3, F4 (desired thrusts)
		
		We also know;
		
		tau_1 = k_m * omega1 ** 2; F1 = k_f * omega1 ** 2 => tau_1 = k_m * (F1 / k_f)
		tau_2 = - k_m * omega2 ** 2; F2 = k_f * omega2 ** 2 => tau_2 = -k_m * (F2 / k_f)
		tau_3 = k_m * omega3 ** 2; F3 = k_f * omega3 ** 2 => tau_3 = k_m * (F3 / k_f)
		tau_4 = -k_m * omega ** 4; F4 = k_f * omega4 ** 2 => tau_4 = -k_m * (F4 / k_f)
		
		This leads to;
		
		tau_z = kappa * (F1 - F2 + F3 - F4), where kappa is k_m/k_f (drag/thrust)
		l = L / (2 * sqrt(2))
		
		This gives us
		
		F1 - F2 - F3 + F4 = tau_x/l - (1)
		F1 + F2 - F3 - F4 = tau_y/l - (2)
		F1 - F2 + F3 - F4 = tau_z / kappa - (3)
		F1 + F2 + F3 + F4 = F - (4)
		
		 (1) + (4) gives us;
		
		 2 * F1 + 2 * F4 = (tau_x/l) + F - (5)</pre>

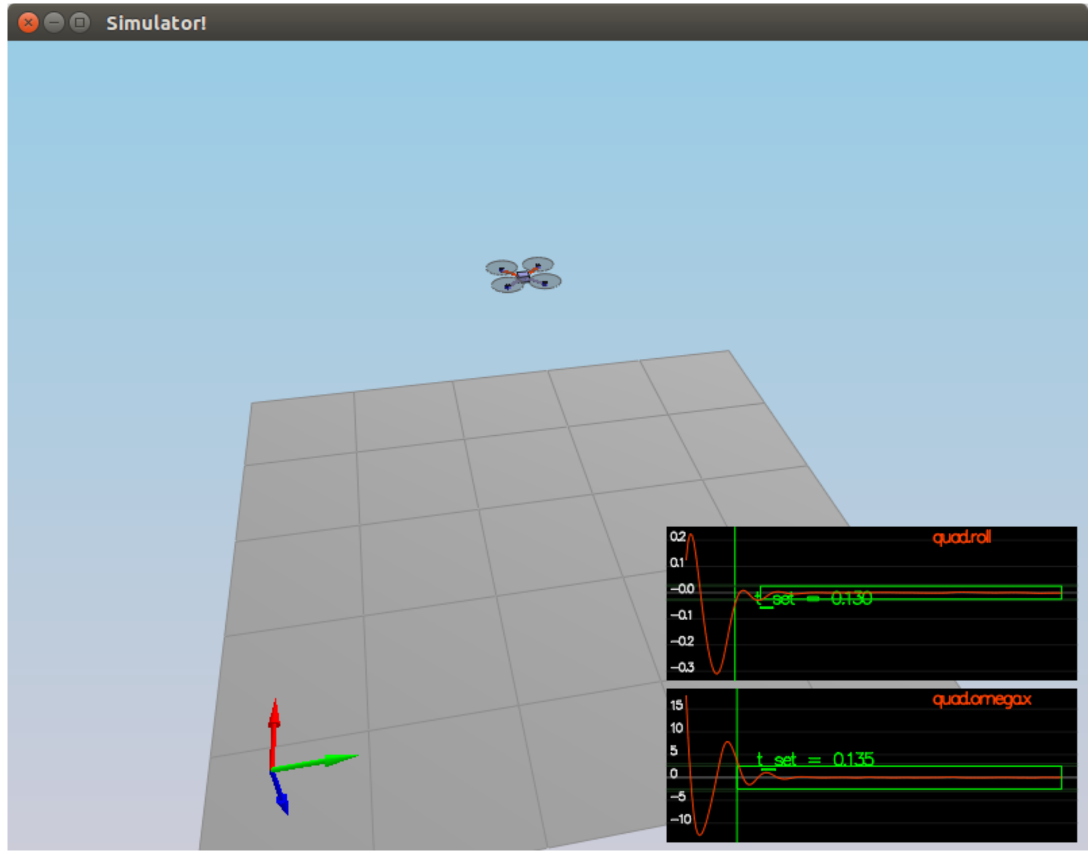
CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS
		(2) + (3) gives us; 2 * F1 - 2 * F4 = (tau_y/l) + (tau_z / kappa) - (6) (4) - (1) gives us; 2 * F2 + 2 * F3 = F - tau_x / l; - (7) (2) - (3) gives us; 2 * F2 - 2 * F3 = tau_y/l - tau_z / kappa - (8) (5) + (6) gives us; 4 * F1 = (tau_x/l) + F + (tau_y/l) + (tau_z / kappa) => F1 = F/4 + (tau_x)/l/4 + tau_y/l/4 + tau_z/kappa/4 (7) + (8) gives us; 4 * F2 = F - tau_x / l + tau_y / l - tau_z / kappa => F2 = F/4 - (tau_x)/l/4 + tau_y/l/4 - tau_z/kappa/4 (7) - (8) gives us; 4 * F3 = F - tau_x / l - tau_y/l + tau_z / kappa => F3 = F/4 - tau_x/l/4 - tau_y/l/4 + tau_z/kappa/4 (5) - (6) gives us; 4 * F4 = F + tau_x/l - tau_y/l - tau_z / kappa => F4 = F/4 + tau_x/l/4 - tau_y/l/4 - tau_z/kappa/4 Since tau's are moments and F is the total force, let us relate them to inputs tau_x = momentCmd.x tau_y = momentCmd.y

CRITERIA	MEETS SPECIFICATIONS	IMPLEMENTATION DETAILS

		`tau_z = -momentCmd.z`
		`F = collThrustCmd`
		
		`Let us code with these values and the above equations and translating l to L.`
		
		The code implemented in lines QuadControl.cpp:81..89

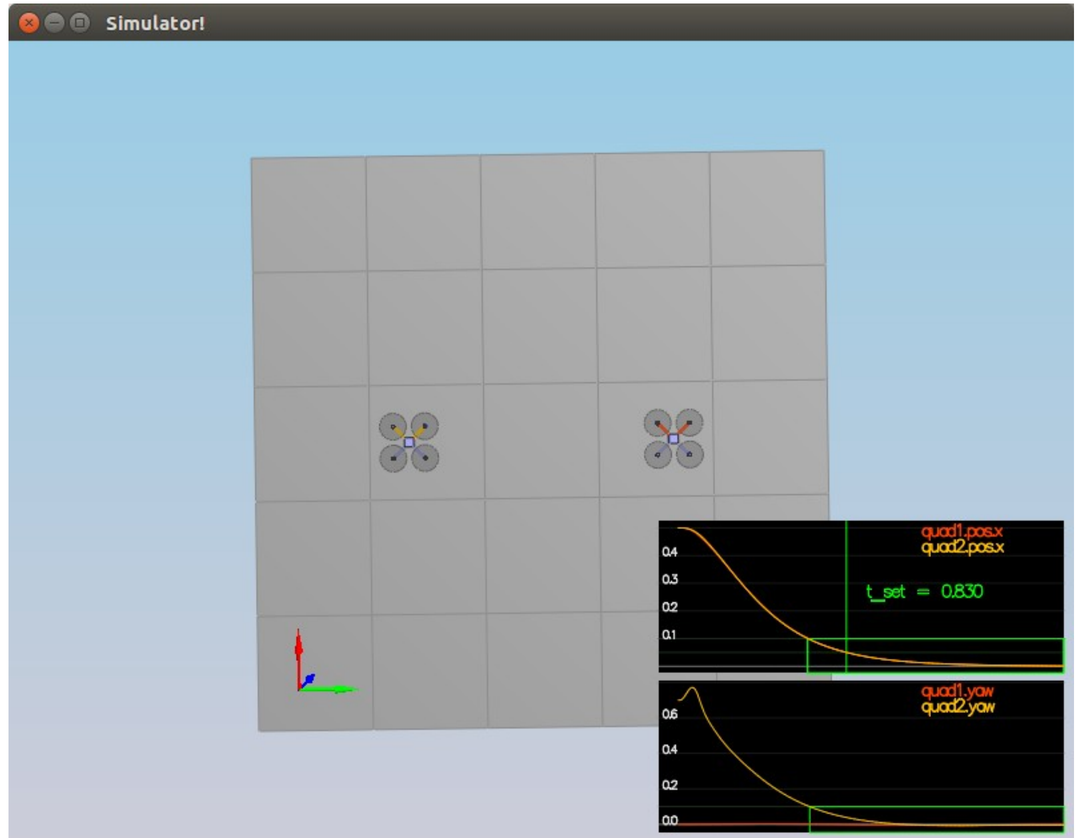
Flight Evaluation

CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
Your C++ controller is successfully able to fly the provided test trajectory and visually passes 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).	Scenario-2 After building the GenerateMotorCommands(), BodyRateControl() and RollPitchControl() and tuning the parameter kpBank and kpPQR we have stable flight with tests passing. Simulation #736 (../config/2_AttitudeControl.txt) 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

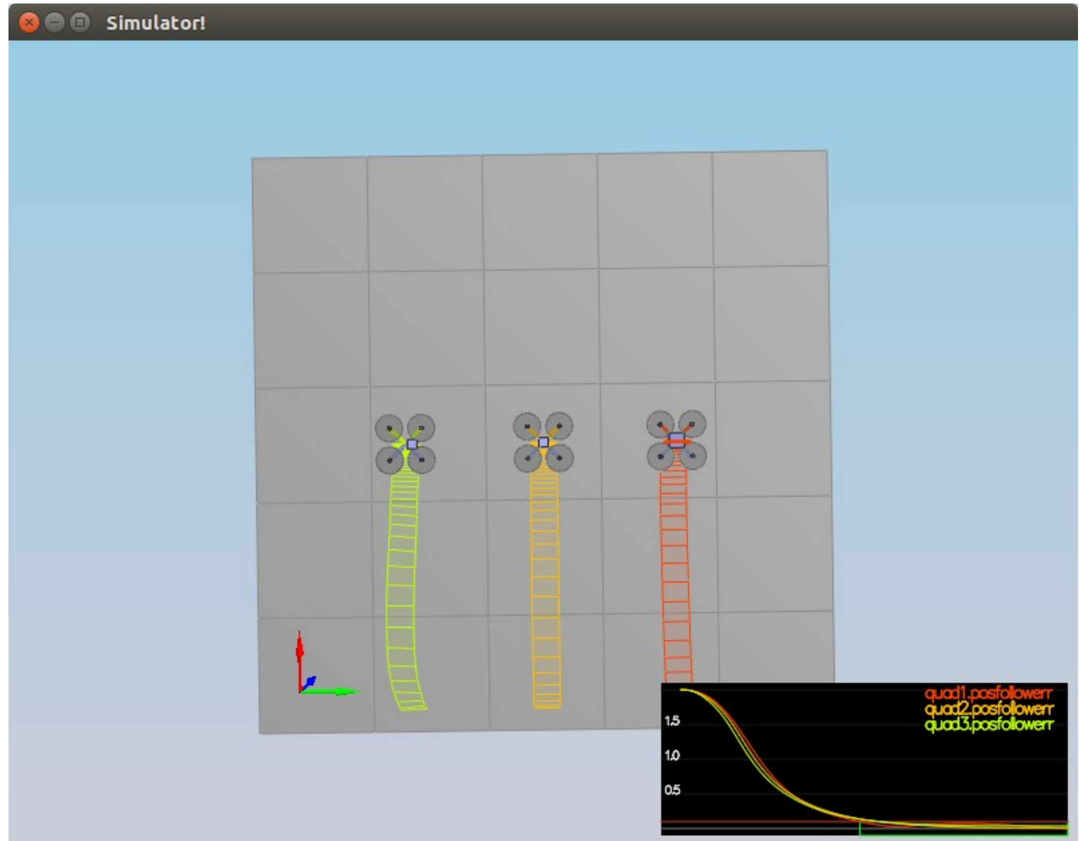
CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
		 Scenario-3: After implementing the functions LateralPositionControl(), AltitudeControl(), YawControl() and tuning parameters kpPosZ, kpPosZ, kpVelXY, kpVelZ, kpYaw and kpPQR we have the following results.

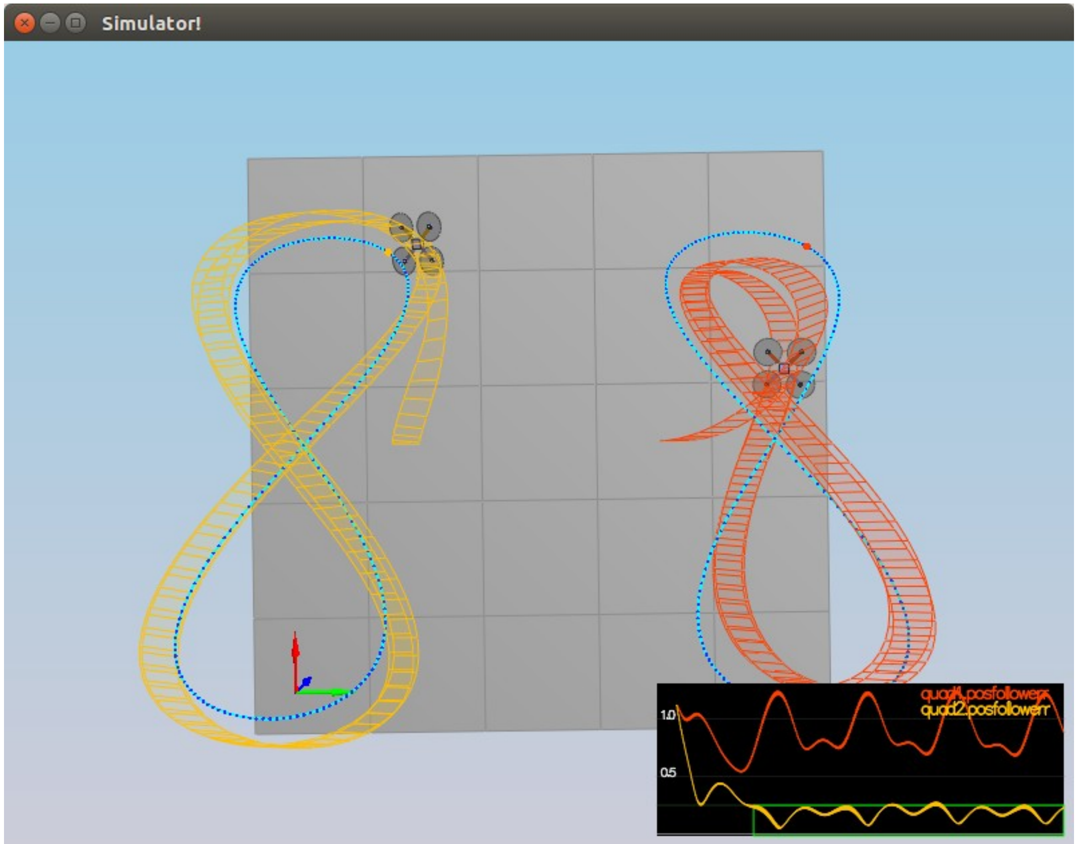
CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS

		The parameters are tuned as:
		
		# Position control gains
		
		`kpPosXY = 2`
		`kpPosZ = 2`
		`KiPosZ = 50`
		`# Velocity control gains`
		`kpVelXY = 10`
		`kpVelZ = 10`
		
		`# Angle control gains`
		`kpBank = 15`
		`kpYaw = 2`
		
		`# Angle rate gains`
		`# kpPQR = 23, 23, 5`
		
		`kpPQR = 96, 96, 6`
		
		The results from the log are:
		
		Simulation #903 (../config/3_PositionControl.txt)
		
		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

CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
		 Scenario-4: With the tuning of parameters the tests pass in scenario-4 and the details are as below. Simulation #1015 (../config/4_Nonidealities.txt)

CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
		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

CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
		

CRITERIA	MEETS SPECIFICATIONS	SCENARIO TESTS
		Scenario-5:  Based on the parameters tuned we get the following trajectory path and it is stable. Simulation #1043 (../config/5_TrajectoryFollow.txt) PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds