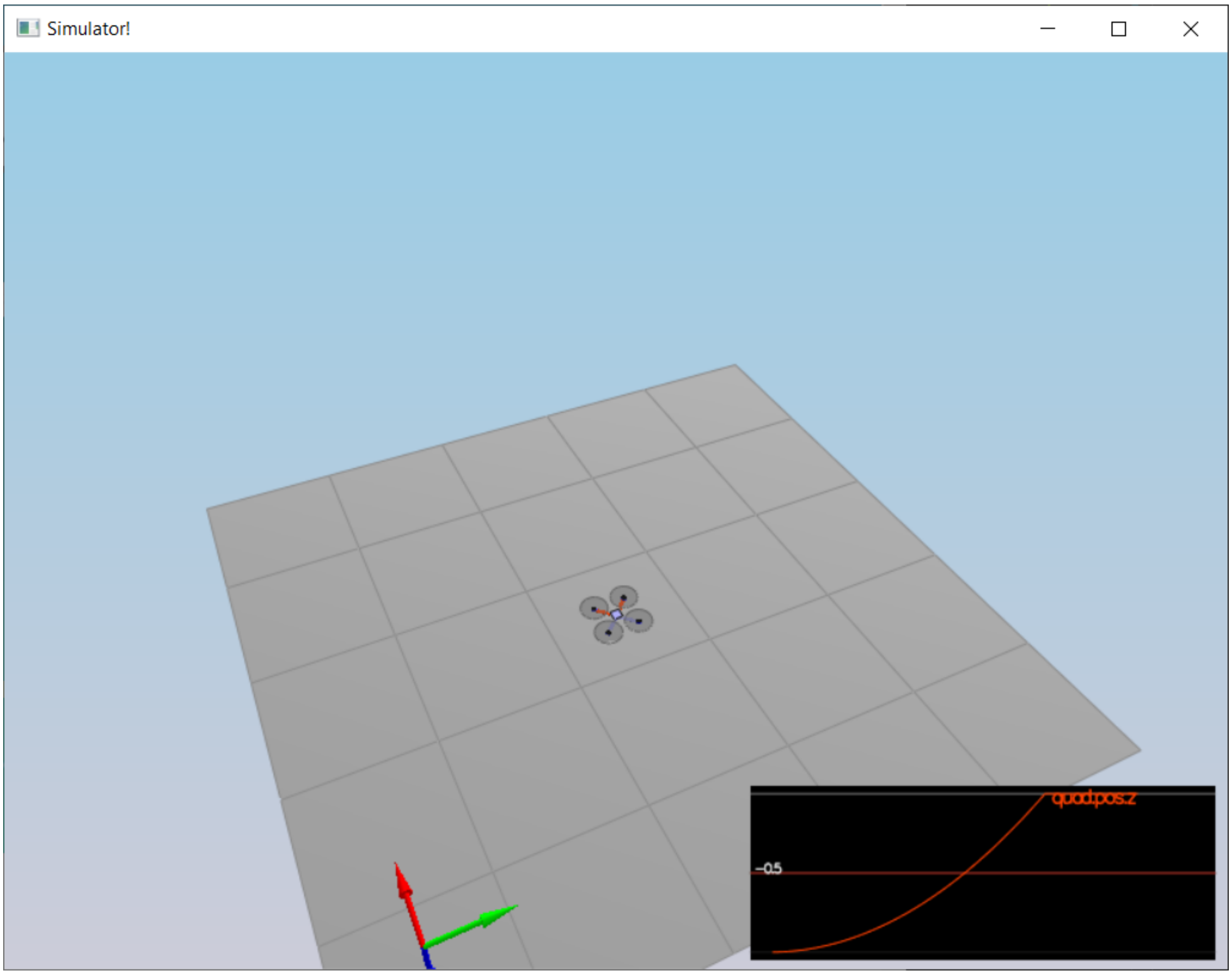
Project 3: Control of a 3D Quadrotor

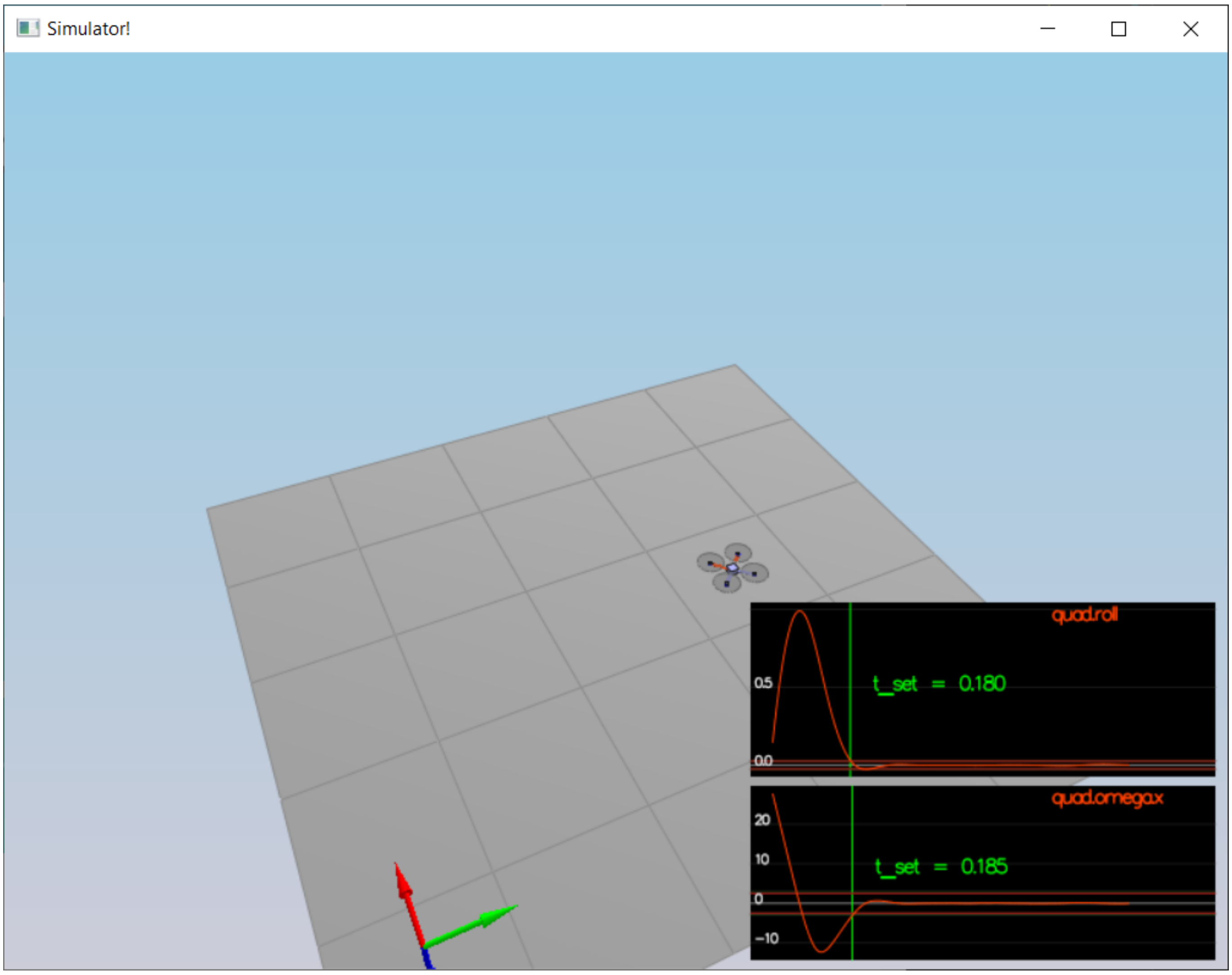
Chris Dalke

Scenario 1: Intro



This first scenario is a test of the simulation, and did not require any implementation. I tweaked the value of the mass of the drone to `0.5`, which made the drone (roughly) hover.

Scenario 2: Attitude Control



The second scenario tests the attitude control system for the drone, which requires the body rate controller and the roll / pitch controller.

2.1: Converting collective commands to individual motor thrusts

The first step is to implement `GenerateMotorCommands`. This function takes in a collective thrust and moment, and computes the individual motor thrust commands.

Conceptually, this is the "innermost" math - The output of the controllers is a collective control signal, and this function converts it into motor commands.

To implement this, I used the equations for computing angular velocity of the individual propellers. The system of equations, after simplifying all terms to compute force instead of angular velocity:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} = \begin{bmatrix} \bar{c} \\ \frac{\tau_x}{l} \\ \frac{\tau_y}{l} \\ \frac{\tau_z}{k} \end{bmatrix}$$

The constants can all be computed based on the inputs `collThrustCmd` and `momentCmd`:

```
float c_bar = collThrustCmd;
float p_bar = momentCmd.x / l;
float q_bar = momentCmd.y / l;
float r_bar = momentCmd.z / kappa;
```

I solved the system of linear equations to provide the following formulas for the forces:

```
float f_1 = (c_bar + p_bar + q_bar + r_bar) / 4.0f;
float f_2 = (c_bar - p_bar + q_bar - r_bar) / 4.0f;
float f_4 = (c_bar + p_bar - q_bar - r_bar) / 4.0f;
float f_3 = (c_bar - p_bar - q_bar + r_bar) / 4.0f;
```

A few other steps used in my code:

- Constrain the values to min/max motor thrusts
- Reverse the direction of the r_bar terms. Since the Z-axis is pointed down, the rotation due to the moment must be reversed.
- Map the thrusts from the clockwise numbering pattern used in the formulas, to the pattern used in the project.

2.2: Implementing the Body Rate Controller

The body-rate controller is a simple P controller, since it is a first-order system. The P controller computes a rate command:

```
V3F prqError = pqrCmd - pqr;
V3F rateCmd = kpPQR * prqError;
```

The rate command is converted to a 3-axis moment using the moment of inertia:

```
V3F I(Ixx, Iyy, Izz);
momentCmd = I * rateCmd;
```

2.3: Implementing the Roll-Pitch Controller

The last part of implementing the attitude controller is the roll-pitch controller, which commands roll & pitch rates in order to hit a desired lateral acceleration and collective thrust.

This controller drives two terms, B_x and B_y, "control knobs" for x and y. The control loop for these terms is a P controller, since the roll-pitch controller is also a first-order system.

$$\dot{b}_c^x = k_p(b_c^x - b_a^x)$$

$$\dot{b}_c^y = k_p(b_c^y - b_a^y)$$

```
// Compute target with P controllers
float b_x_c_target = -CONSTRAIN(accelCmd.x / collThrustAccel, -maxTiltAngle,
maxTiltAngle);
float b_y_c_target = -CONSTRAIN(accelCmd.y / collThrustAccel, -maxTiltAngle,
maxTiltAngle);
float b_x_c_dot = kpBank * (-b_x_c_target + R(0, 2));
float b_y_c_dot = kpBank * (-b_y_c_target + R(1, 2));
```

The output of the P controller (b_x_c_dot and b_y_c_dot) is fed through a system of equations to get p_c and q_c, the pitch and roll rates:

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

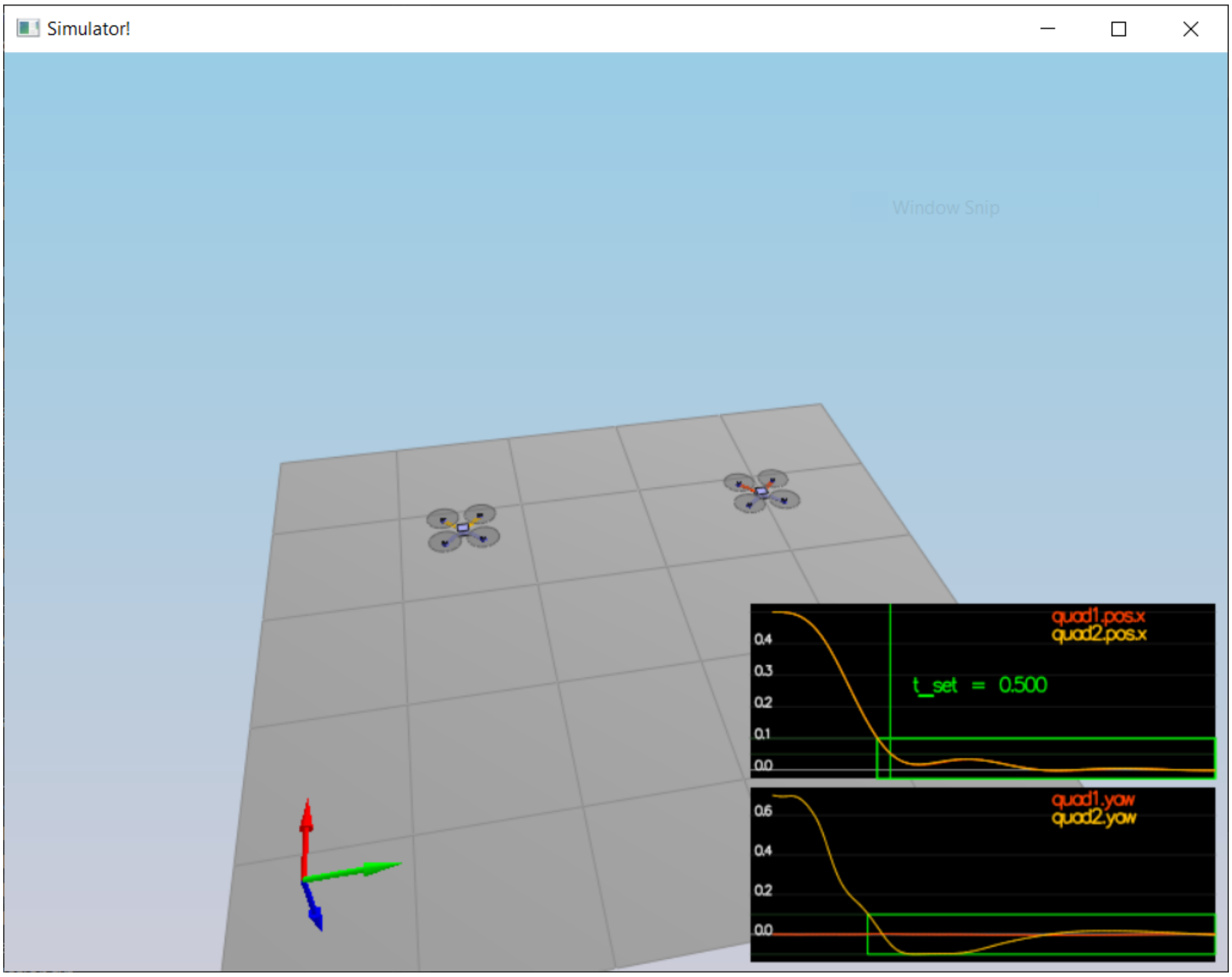
```
// Calculate roll/pitch rate commands with some linear algebra
pqrCmd.x = (-(R(1, 0) * b_x_c_dot) + (R(0, 0) * b_y_c_dot)) / R(2, 2);
pqrCmd.y = (-(R(1, 1) * b_x_c_dot) + (R(0, 1) * b_y_c_dot)) / R(2, 2);
```

I had to fiddle with the direction of the roll/pitch commands to get the vehicle to respond in the correct direction.

2.4: Tuning

I tuned the `kpPQR` and `kpBank` gains experimentally to get the vehicle to stop spinning as quickly as possible with minimal overshoot. For `kpPQR`, I chose P and Q gains of `40`. For `kpBank`, I chose a gain of `10`.

Scenario 3: Position Control



This scenario tests the position and yaw control of the quadrotor.

3.1: Lateral Position control

Lateral (translation about X/Y) position is controlled in the `LateralPositionControl` method, where a PD controller is used to control the lateral position through a lateral acceleration command. This is a second-order system:

```
// PD controller; controls the lateral position through acceleration (Second-order
system)
accelCmd += (kpPosXY * (posCmd - pos)) + (kpVelXY * (velCmd - vel));
```

In this method, I also constrain the velocity and acceleration to their maximum amplitudes. To constrain the vectors proportionally, I use the following logic which computes the magnitude and scales the vector to fit under the limit.

```
// Constrain the velocity to the maximum proportionally
float velMagnitude = velCmd.magXY();
```

```
if (velMagnitude > maxSpeedXY) {
    velCmd *= maxSpeedXY / velMagnitude;
}


// Constrain the acceleration to the maximum proportionally
float accelMagnitude = accelCmd.magXY();
if (accelMagnitude > maxAccelXY) {
    accelCmd *= maxAccelXY / accelMagnitude;
}
```

3.2: Altitude Control

Altitude (translation about Z) is controlled in the `AltitudeControl` method. The altitude controller computes a target collective thrust. I do this in several steps, separated for simplicity.

First, I run a P controller which computes the command velocity, which also includes a feed-forward term. The velocity is limited to the vehicle constraints.

```
float z_dot_command = (kpPosZ * (posZCmd - posZ)) + velZCmd;
z_dot_command = CONSTRAIN(z_dot_command, -maxDescentRate, maxAscentRate);
```

Next, I plug the commanded velocity into another controller which computes a commanded acceleration based on the target velocity.

```
// P controller to control vertical velocity via acceleration.
float z_dot_dot_command = (kpVelZ * (z_dot_command - velZ)) + accelZCmd;
```

Note that this is really a PD controller, just broken up into separate steps.

Lastly, the acceleration is converted to a thrust value.

```
// Convert the vertical acceleration to a thrust in the body frame
// Thrust is inverted; negative thrust in NED frame actually means UP
thrust = -((z_dot_dot_command - g) * mass) / R(2, 2);
```

3.3: Yaw Control

Yaw (rotation about the Z axis) is controlled in the `YawControl` method. This is a very simple P controller with a few minor tweaks since the angle is on a circle.

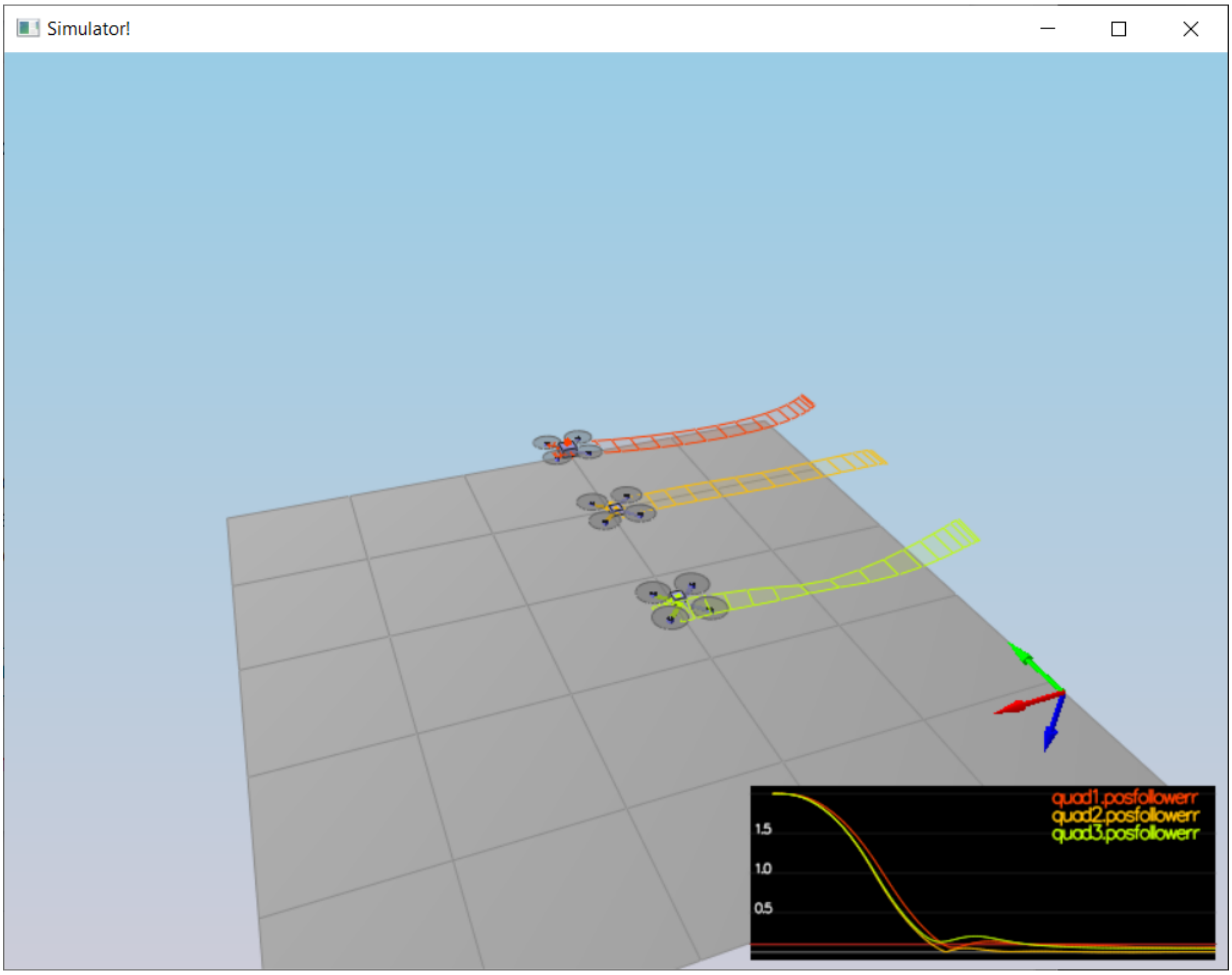
We want the vehicle to always take the shortest direction, so I added some logic to clamp the angle and ensure that the quad always chooses the direction that will be the shortest.

```
// Compute yaw, wrapped around in a circle, and the error
float yawCmdWrapped = fmodf(yawCmd, 2.0f * F_PI);
float yawError = yawCmd - yaw;

// If the error is greater than PI, flip it so we travel the other direction
// This ensures we're always taking the closest path to the target
if (yawError > F_PI) {
yawError -= 2.0f * F_PI;
}
else if (yawError < -F_PI) {
yawError += 2.0f * F_PI;
}

// Run P controller
yawRateCmd = (kpYaw * (yawCmd - yaw));
```

Scenario 4: Nonidealities



This scenario tests conditions where the drone has some configuration that interferes with its mechanics, such as a shifted weight or increased mass. In order to account for these cases, I added the I term to the altitude controller.

At each iteration, I add the z error to the integrator:

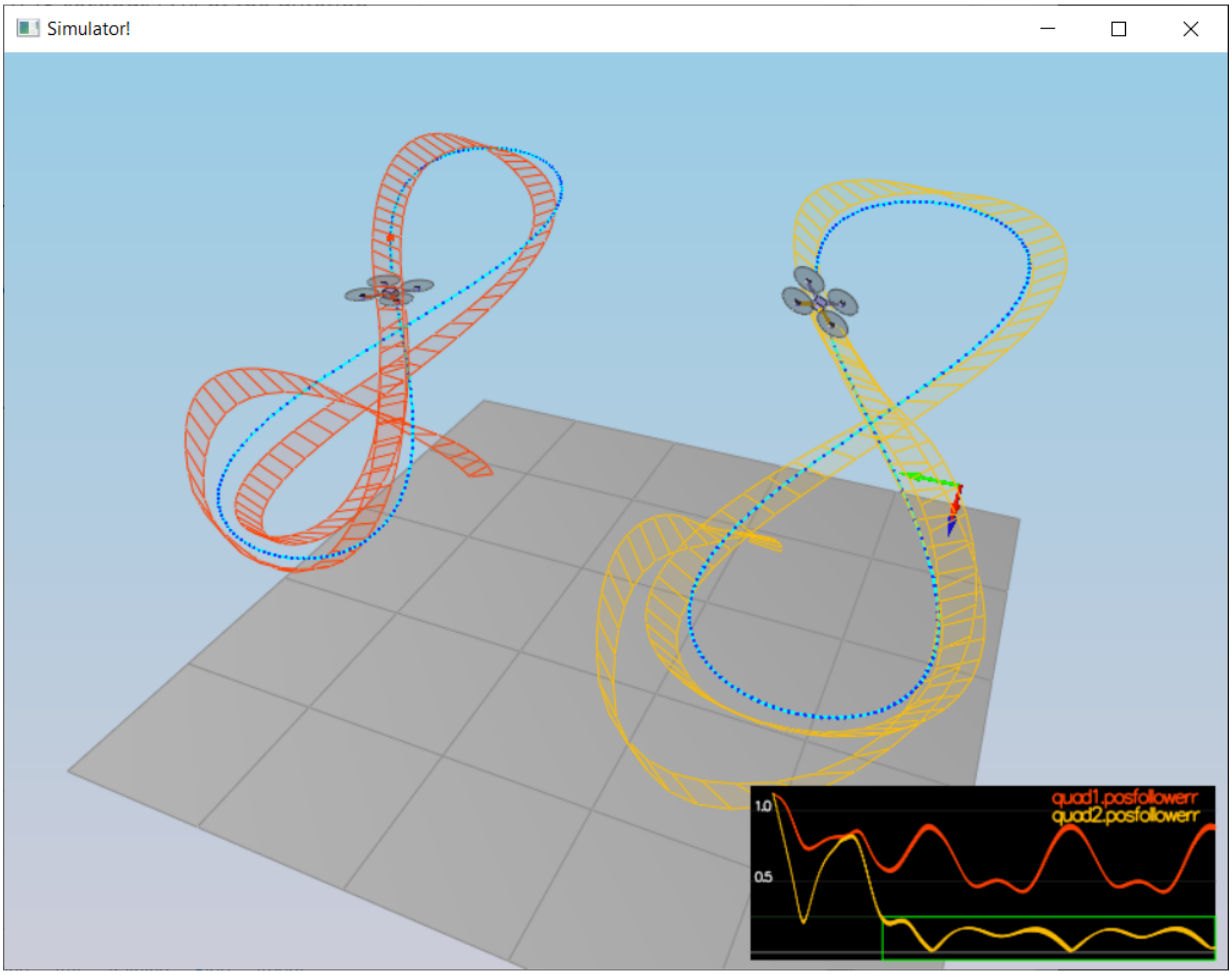
```
integratedAltitudeError += z_error * dt;
```

This term is added to the controller:

```
float z_dot_dot_command = (kpVelZ * (z_dot_command - velZ)) + accelZCmd + (KiPosZ
* integratedAltitudeError);
```

The I term helps accommodate for the persistent altitude error on the red drone, which had increased mass and hovered below its target position.

Scenario 5: Trajectory Following



This scenario tests the ability of the drones to follow a trajectory. I was able to tune the yellow drone (The feed-forward version) to follow the trajectory closely.