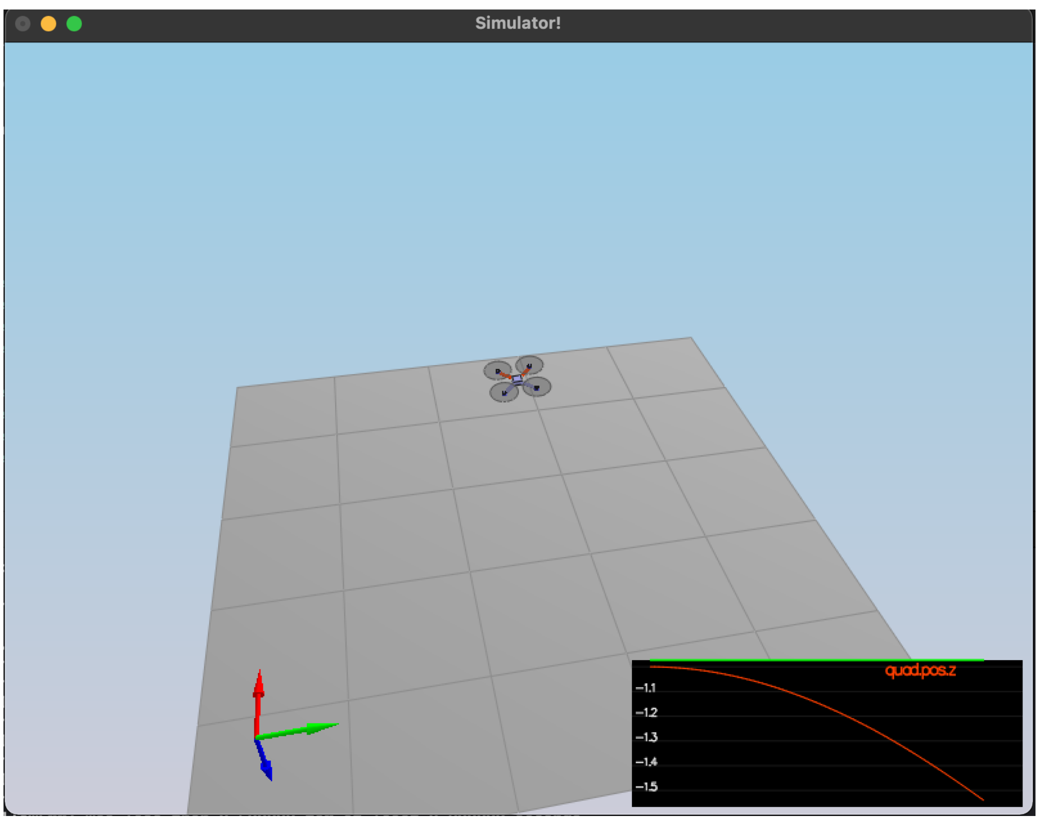
1_Intro:

The original problem was the *Mass* parameter was not the actual mass of the quadrotor therefore by running simulation the drone falls down. To fix this the *Mass* parameter is update to represent the actual mass of the drone and the thrust generated is enough to keep the drone to hover (note this will be the "*else"* scenario where controller is not sending additional thrust command):

```
//hover
  if (collThrustCmd > 0){
      float d = L * sqrt(2.f) / 2.f;
      float p_bar = momentCmd.x / d;
      float q_bar = momentCmd.y / d;
      float r_bar = - momentCmd.z / kappa;
      float c_bar =  collThrustCmd ;

      cmd.desiredThrustsN[0] = (c_bar + p_bar + q_bar + r_bar) / 4.f;
      cmd.desiredThrustsN[1] = (c_bar - p_bar + q_bar - r_bar) / 4.f;
      cmd.desiredThrustsN[2] = (c_bar + p_bar - q_bar - r_bar) / 4.f;
      cmd.desiredThrustsN[3] = (c_bar - p_bar - q_bar + r_bar) / 4.f;
  }else{
      cmd.desiredThrustsN[0] = mass * 9.81f / 4.f ; // front left
      cmd.desiredThrustsN[1] = mass * 9.81f / 4.f ; // front right
      cmd.desiredThrustsN[2] = mass * 9.81f / 4.f ; // rear left
      cmd.desiredThrustsN[3] = mass * 9.81f / 4.f ; // rear right
  }
```



GeneratedMotorCommands Modification

The function is updated to include the scenario where thrust command is given by the controller (greater than zero). Using force and moment equations of motion, the desire thrust in calculated by considering drag trust ration ($\frac{K_f}{K_m}$) and rotors being in 45° with respect to the body axis(x, y). This implementation is on the "*if*" statement under the GeneratedMotorCommands function.

Body-Rate Controller Implementation

Proportional controller is implemented using equations below to calculate commanded roll, pitch and yaw moments:

$$\overline{U_P} = \frac{\tau_x}{I_x}$$

$$\overline{U_q} = \frac{\tau_y}{I_x}$$

$$\overline{U_r} = \frac{\tau_r}{I_x}$$

Additionally:

$$\overline{U_P} = K_p^p (p_c - p_a)$$

$$\overline{U_q} = K_p^q (q - q_a)$$

$$\overline{U_r} = K_p^r (r_c - r_a)$$

Using vector notation, the 3-axis moment command are returned from BodyRateControl function:

```
V3F QuadControl::BodyRateControl(V3F pqrCmd, V3F pqr)
{
  // Calculate a desired 3-axis moment given a desired and current body rate
  // INPUTS:
  //   pqrCmd: desired body rates [rad/s]
  //   pqr: current or estimated body rates [rad/s]
  // OUTPUT:
  //   return a V3F containing the desired moments for each of the 3 axes

  // HINTS:
  //  - you can use V3Fs just like scalars: V3F a(1,1,1), b(2,3,4), c; c=a-b;
  //  - you'll need parameters for moments of inertia Ixx, Iyy, Izz
  //  - you'll also need the gain parameter kpPQR (it's a V3F)

  V3F momentCmd;

  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  V3F MOI;
  MOI.x = Ixx;
  MOI.y = Iyy;
  MOI.z = Izz;
  momentCmd  = MOI * kpPQR * (pqrCmd - pqr);

  ////////////////////////////// END STUDENT CODE ////////////////////////////

  return momentCmd;
}
```

Roll-Pitch Controller Implementation

Proportional controller is designed to calculate roll and pitch rate in body-frame. The rate of change can be calculated using the following equation:

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

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

Using rotational metric, angular velocities can be calculated:

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

The implementation is done in the RollPitchController function. Note the given collective thrust is converted to the acceleration first. Additionally maxTiltAngle is used to apply the constraint to ensure the action is feasible for the drone.

```
  V3F pqrCmd;
  Mat3x3F R = attitude.RotationMatrix_IwrtB();

  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  float R11 = R(0,0);
  float R12 = R(0,1);
  float R13 = R(0,2);
  float R21 = R(1,0);
  float R22 = R(1,1);
  float R23 = R(1,2);
  float R33 = R(2,2);

  if (collThrustCmd > 0) {
      float c_d = - collThrustCmd / mass;
      float bxCmd  = CONSTRAIN(accelCmd.x / c_d, -sin(maxTiltAngle), sin(maxTiltAngle));
      float byCmd = CONSTRAIN(accelCmd.y / c_d , -sin(maxTiltAngle), sin(maxTiltAngle));

      float bxdotCmd = kpBank * (bxCmd - R13);
      float bydotCmd = kpBank * (byCmd - R23);

      pqrCmd.x = ( R21 * bxdotCmd - R11 * bydotCmd ) / R33;
      pqrCmd.y = ( R22 * bxdotCmd - R12 * bydotCmd) / R33;
  }else{
      pqrCmd.x = 0;
      pqrCmd.y = 0;
  }
  pqrCmd.z = 0;
  ////////////////////////////// END STUDENT CODE ///////////////////////////

  return pqrCmd;
}
```

Altitude Controller Implementation

For the Altitude controller, I used a cascaded controller in order to better tune scenario 3 and 4. Additionally the calculated angular acceleration is converted to force. MaxDescentRate is used to apply constraint to commanded velocity to make sure the command is feasible for the drone. Same constraint is applied for acceleration by translating MaxDescentRate to acceleration (divided by delta time).

```
  Mat3x3F R = attitude.RotationMatrix_IwrtB();
  float thrust = 0;

  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  float R33 = R(2,2);

  float posZDelta = posZCmd - posZ;
  integratedAltitudeError += posZDelta * dt;

  velZCmd += kpPosZ * posZDelta + KiPosZ * integratedAltitudeError;
  velZCmd = CONSTRAIN(velZCmd, - maxDescentRate, maxDescentRate);
  float velZDelta = velZCmd - velZ;

  float ubar = kpPosZ * posZDelta + kpVelZ * velZDelta + accelZCmd + KiPosZ * integratedAltitudeError;
  float a = - CONSTRAIN((ubar - CONST_GRAVITY) / R33, - maxDescentRate / dt, maxAscentRate /dt);
  thrust = mass * a;

  ////////////////////////////// END STUDENT CODE ///////////////////////////
  return thrust;
}
```

Lateral Controller Implementation

To calculate horizontal acceleration, based on the desired lateral pos, velocity and acceleration, a cascaded controller is implemented. The control equations for acceleration is as follow (same equation for y direction):

$$
\begin{aligned}
\ddot{x}_{\text{command}} &= c b^x_c \\
\ddot{x}_{\text{command}} &= k^x_p(x_t - x_a) + k^x_d(\dot{x}_t - \dot{x}_a) + \ddot{x}_t \\
b^x_c &= \ddot{x}_{\text{command}}/c
\end{aligned}
$$

```
  V3F accelCmd = accelCmdFF;

  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  V3F kpPos;
  kpPos.x = kpPosXY;
  kpPos.y = kpPosXY;
  kpPos.z = 0.f;

  V3F kpVel;
  kpVel.x = kpVelXY;
  kpVel.y = kpVelXY;
  kpVel.z = 0.f;

  velCmd += kpPos * (posCmd - pos);

  if (velCmd.mag() > maxSpeedXY){
      velCmd = velCmd.norm() * maxSpeedXY;
  }
  V3F posDelta = posCmd - pos;
  V3F velDelta = velCmd - vel;

  accelCmd += kpPos * posDelta + kpVel * velDelta;

  if (accelCmd.mag() > maxAccelXY){
      accelCmd = accelCmd.norm() * maxAccelXY;
  }
  ////////////////////////////// END STUDENT CODE /////////////////////////////

  return accelCmd;
}
```

Yaw Controller Implementation

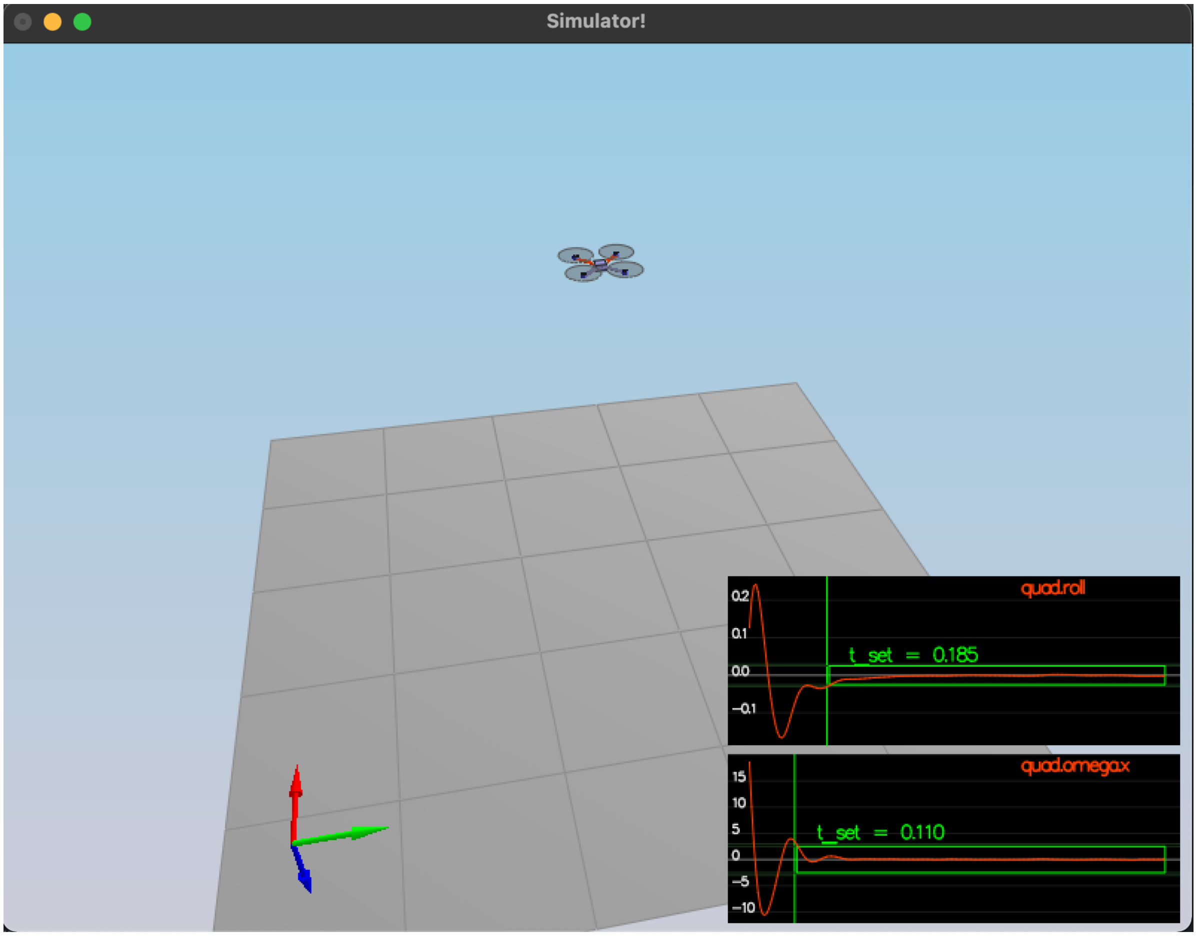
Proportional controller is implemented with the control equation below to calculate yaw command to achieve the desired yaw rate.

$$r_c = k_p(\psi_t - \psi_a)$$

```
  float yawRateCmd=0;
  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
    yawRateCmd = kpYaw * (yawCmd - yaw);//yaw_error;

  ////////////////////////////// END STUDENT CODE /////////////////////////////

  return yawRateCmd;

}
```

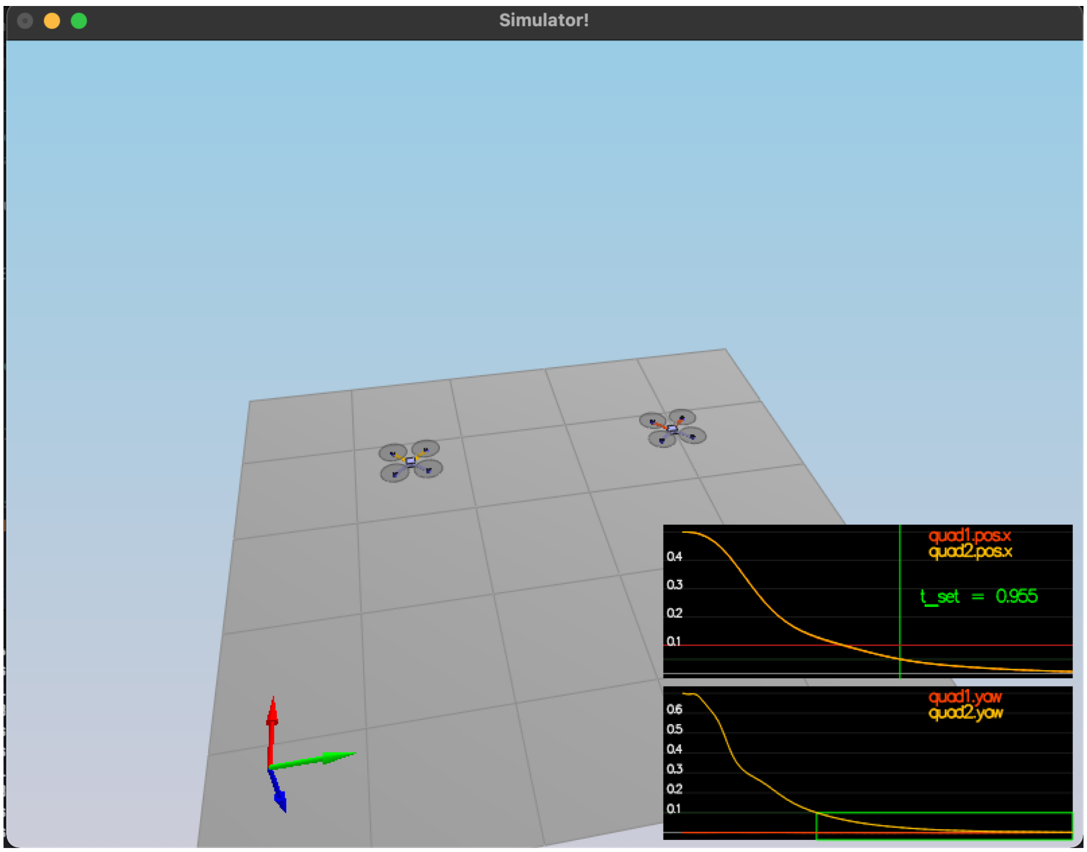
2 AltitudeControl Scenario

In order to stabilize drone for this scenario, where rotational motion is disturbed, KpPQR is tuned (to set Omega for rotation around x axis to zero) under BodyRateControlle function and KpBank under RollPitchController (to control the amount of overshoot).



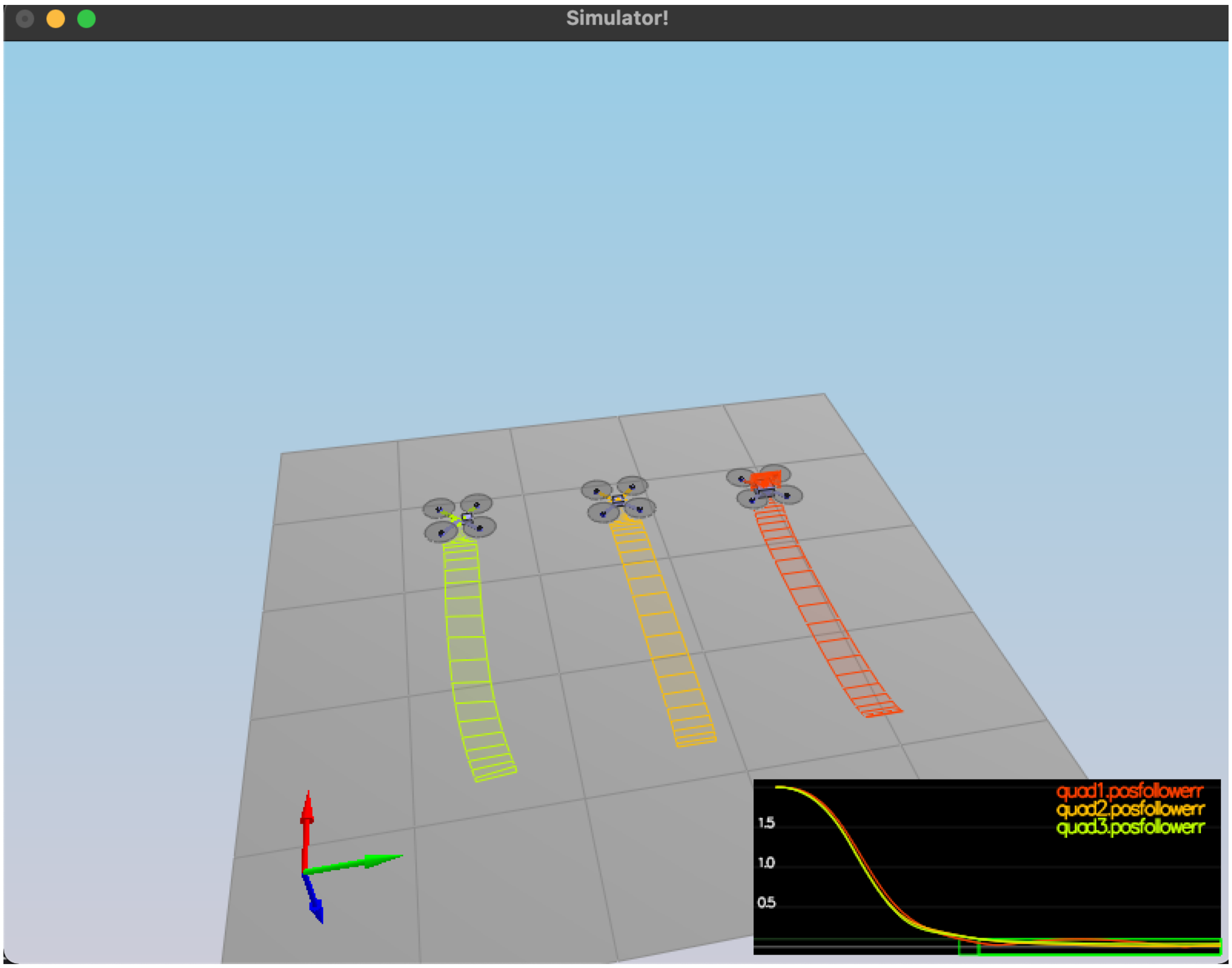
3_PositionControl Scenario

For this scenario parameters KpPosZ, kpVelZ (used under AltitudeControll function) and kpPosXY and kpVelXY (used under LateralController function) are tuned. Additionally to stabilize the yaw motion, kpYaw is tuned to saturate the error to zero.



4_NonIdealities Scenario

For this scenario KiPosZ is also tuned to correct the mass disturbance. I also had to re-tune the parameters from scenario 3 to get the pos errors saturated.



5_TrajectoryFollow Scenario

For this scenario an adjustment was done on parameters to make the yellow drone follow the desired trajectory.