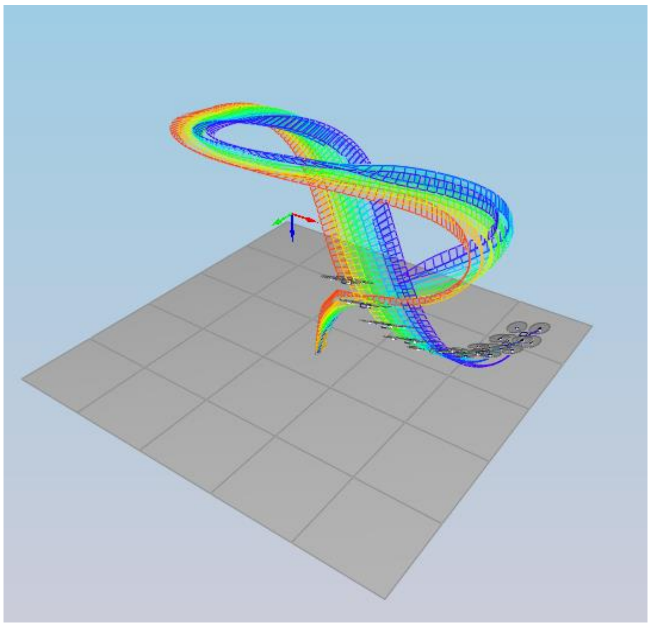
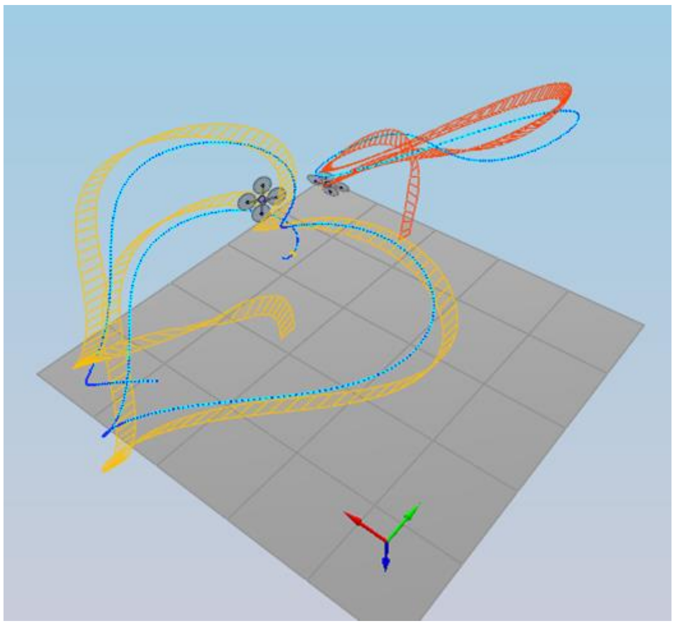
31/01/2019

Project#3: Building a controller

Student: Enrico Baracaglia





Objective of the project

The objective of this project was to implement a C++ controller in order to make a drone fly in different scenarios with given requirements to meet. The evaluation of the project is based on the following performance metrics:

Scenario 2

- roll should be less than 0.025 radian of nominal for 0.75 seconds (3/4 of the duration of the loop)
- roll rate should be less than 2.5 radian/sec for 0.75 seconds

Scenario 3

- X position of both drones should be within 0.1 meters of the target for at least 1.25 seconds
- Quad2 yaw should be within 0.1 of the target for at least 1 second

Scenario 4

- position error for all 3 quads should be less than 0.1 meters for at least 1.5 seconds

Scenario 5

- position error of the quad should be less than 0.25 meters for at least 3 seconds

The first step was setting the C++ environment as outlined in outlined in the **C++ project readme**. I personally worked on Windows with Visual Studio 2017. The majority of the code was written in `src/QuadControl.cpp`, in which the algorithms for the controllers have been implemented and in `QuadControlParams.txt`, where it was possible to tune the different gains for the controllers.

Body rate and roll/pitch control (scenario 2)

Before implementing the controller, it was asked to complete the section *GenerateMotorCommands* in order to convert the 3-axis moment and collective thrust in individual commands for the motors. The main idea was to use the following equations

$$F1 + F2 + F3 + F4 = F_{tot} \qquad (1)$$

$$F1 - F2 - F3 + F4 = \frac{M_x}{l} \qquad (2)$$

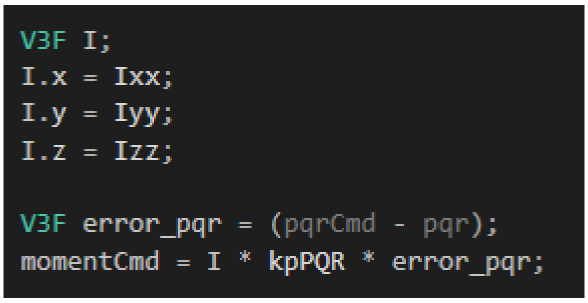
$$F1 + F2 - F3 - F4 = \frac{M_y}{l} \qquad (3)$$

$$-F1 + F2 - F3 + F4 = \frac{M_z}{kappa} \qquad (4)$$

Where $F1$, $F2$, $F3$ and $F4$ (the motors' thrust) are linked to the moments Mx, My, Mz produced by the motors on each axis. The value $kappa$ represents, in this case, the drag/thrust ratio, which is km/kf, while the l is the arm length parameter.

After this implementation, a proportional controller has been coded in order to control the body rate variables p, q and r. Here the method used was:

1) calculate the error between the desired body rates and the actual ones
2) Multiply this error by the proportional gain $kpPQR$ and the moments of inertia on the three axes in order to get a Vector3 with the desired moments command.



```
V3F I;
I.x = Ixx;
I.y = Iyy;
I.z = Izz;

V3F error_pqr = (pqrCmd - pqr);
momentCmd = I * kpPQR * error_pqr;
```

As regards the control for the pitch and roll, this part required the implementation of another proportional controller on the rotation matrix R, used to pass from lateral accelerations in body frame to desired accelerations in world frame and vice versa. Here again, we calculated the error between the desired accelerations and the current ones and we multiplied these errors by the proportional controller $kpBank$. Since we needed to output roll and pitch rates commands ($\boldsymbol{p}_c$ and $\boldsymbol{q}_c$), we applied the following equation in order to have values in body-frame:

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

where,

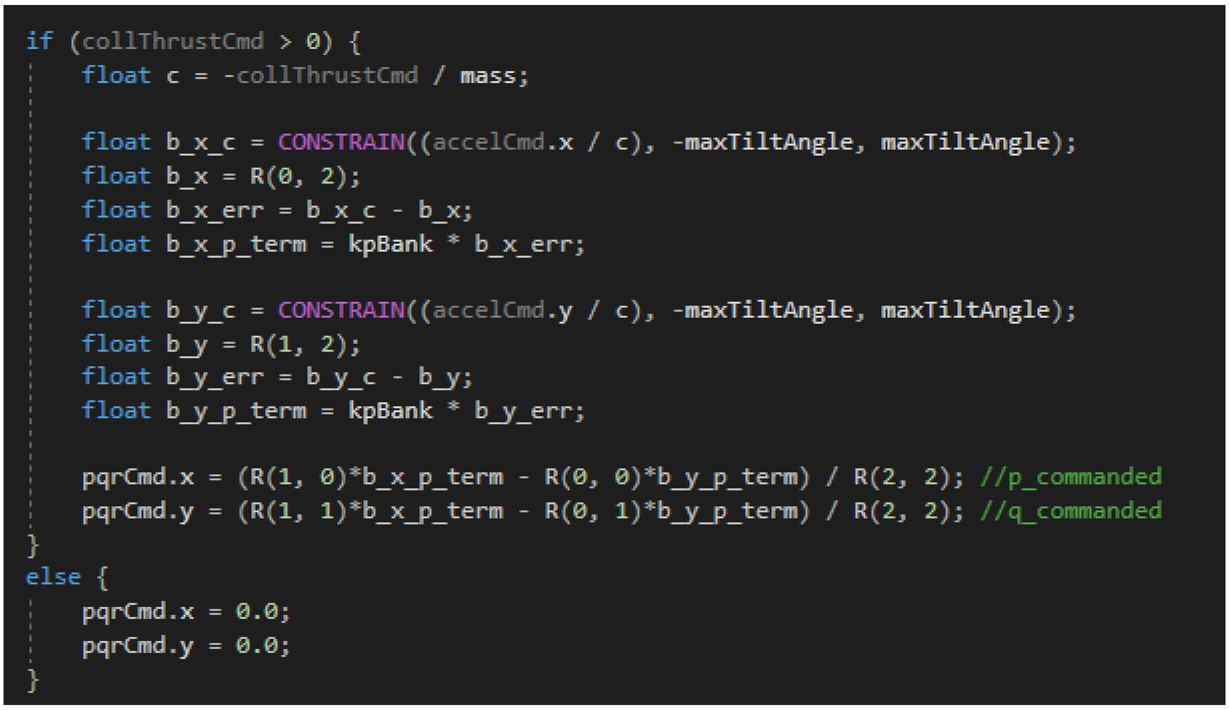
$$\dot{b}_c^x = k_p(b_c^x - b_a^x)$$

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

and

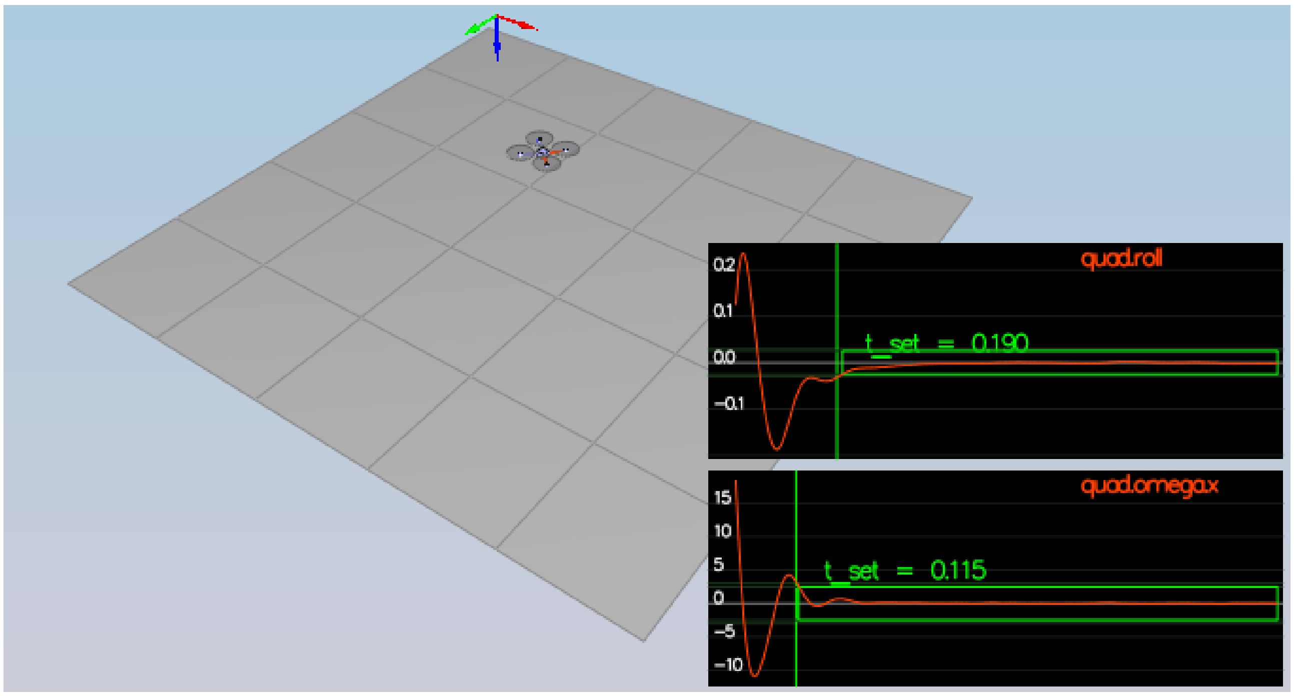
$$b_a^x = R_{13} \quad , \quad b_a^x = R_{23}$$

We also constrained the values of the target R_{13} and R_{23} to stay within the limits of the maximum tilt angle (both positive and negative).



```
if (collThrustCmd > 0) {
    float c = -collThrustCmd / mass;

    float b_x_c = CONSTRAIN((accelCmd.x / c), -maxTiltAngle, maxTiltAngle);
    float b_x = R(0, 2);
    float b_x_err = b_x_c - b_x;
    float b_x_p_term = kpBank * b_x_err;

    float b_y_c = CONSTRAIN((accelCmd.y / c), -maxTiltAngle, maxTiltAngle);
    float b_y = R(1, 2);
    float b_y_err = b_y_c - b_y;
    float b_y_p_term = kpBank * b_y_err;

    pqrCmd.x = (R(1, 0)*b_x_p_term - R(0, 0)*b_y_p_term) / R(2, 2); //p_commanded
    pqrCmd.y = (R(1, 1)*b_x_p_term - R(0, 1)*b_y_p_term) / R(2, 2); //q_commanded
}
else {
    pqrCmd.x = 0.0;
    pqrCmd.y = 0.0;
}
```

We then tuned the gain parameters $kpPQR$ and $kpBank$, so that the drone could fly within the required performance metrics.



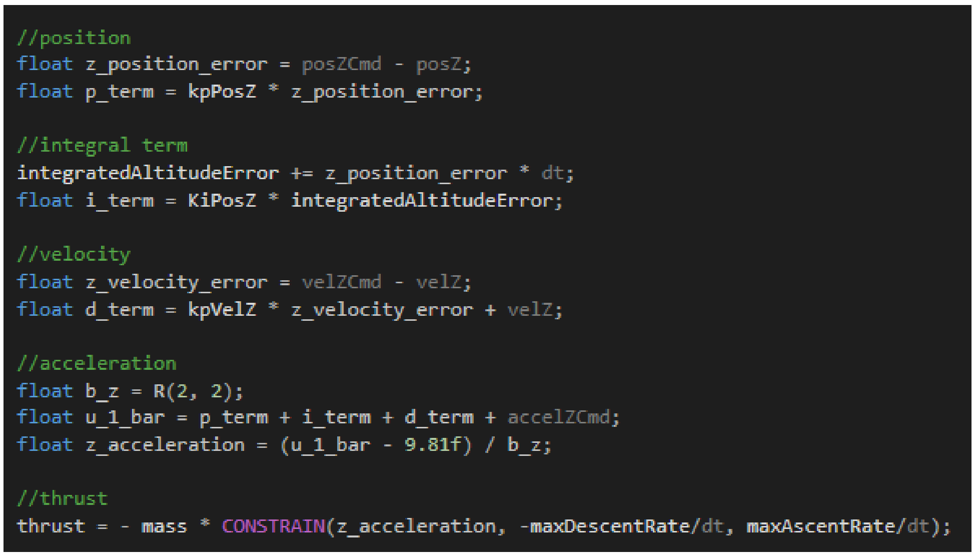
Position/velocity and yaw angle control (scenario 3)

Altitude control

Next step was to implement controller for controlling the altitude of the drone (*AltitudeControl*), its lateral position (*LateralPositionControl*) and yaw (*YawControl*). Starting with the altitude, we used a PID controller to control the vertical acceleration of the drone and return a collective thrust value based on target altitude, actual altitude, target vertical velocity, actual vertical velocity, and a vertical acceleration feed-forward command. The PID controller was composed of:

1) The proportional term, calculated by multiplying the proportional gain $kpPosZ$ by the error between the target and the actual z value (`z_position_error`);
2) The integral term, calculated by integrating the previous error over time;
3) The derivative term, calculated by multiplying the $kpVelZ$ gain with the error between the target velocity and the current velocity in z-direction.
4) Feed-forward term (`accelZCmd`)

One important point to notice is that the acceleration in z has been limited between the maximum descent rate (-ve) and the maximum ascent rate (+ve) divided by the time step dt.



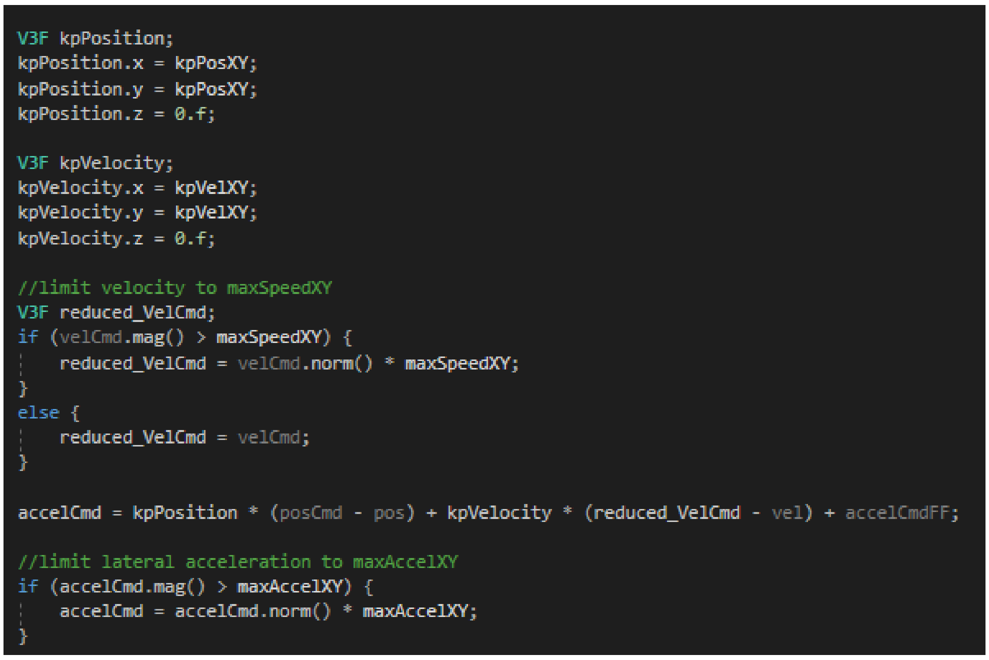
```
//position
float z_position_error = posZCmd - posZ;
float p_term = kpPosZ * z_position_error;

//integral term
integratedAltitudeError += z_position_error * dt;
float i_term = KiPosZ * integratedAltitudeError;

//velocity
float z_velocity_error = velZCmd - velZ;
float d_term = kpVelZ * z_velocity_error + velZ;

//acceleration
float b_z = R(2, 2);
float u_1_bar = p_term + i_term + d_term + accelZCmd;
float z_acceleration = (u_1_bar - 9.81f) / b_z;

//thrust
thrust = - mass * CONSTRAIN(z_acceleration, -maxDescentRate/dt, maxAscentRate/dt);
```

Lateral position control

At this point we implemented the code for the lateral position control. Here a PD controller has been implemented:

1) Proportional term, calculated by multiplying the $kpPosXY$ gain with the error between the target lateral positions (in x and y directions) and the actual ones;
2) Derivative term, calculated by multiplying the $kpVelXY$ gain with the error between the target velocities and the actual ones;
3) Feed-forward term (`accelCmdFF`). Note that the `accelCmdFF.z` was set to zero together with all the other z-components at the beginning of the code.

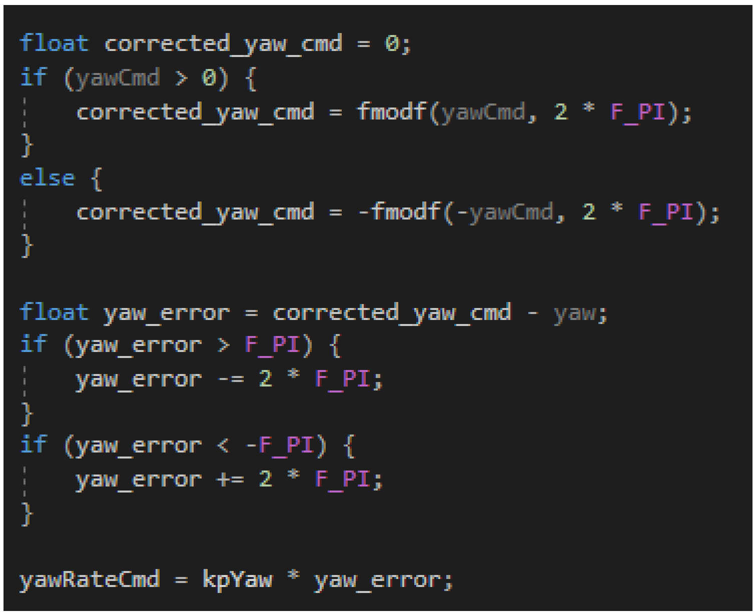
Here as well, the lateral velocities and accelerations have been limited by the maximum XY-speed (`maxSpeedXY`) and the maximum XY-acceleration (`maxAccelXY`) respectively.



```
V3F kpPosition;
kpPosition.x = kpPosXY;
kpPosition.y = kpPosXY;
kpPosition.z = 0.f;

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

//limit velocity to maxSpeedXY
V3F reduced_VelCmd;
if (velCmd.mag() > maxSpeedXY) {
    reduced_VelCmd = velCmd.norm() * maxSpeedXY;
}
else {
    reduced_VelCmd = velCmd;
}

accelCmd = kpPosition * (posCmd - pos) + kpVelocity * (reduced_VelCmd - vel) + accelCmdFF;

//limit lateral acceleration to maxAccelXY
if (accelCmd.mag() > maxAccelXY) {
    accelCmd = accelCmd.norm() * maxAccelXY;
}
```

Yaw control

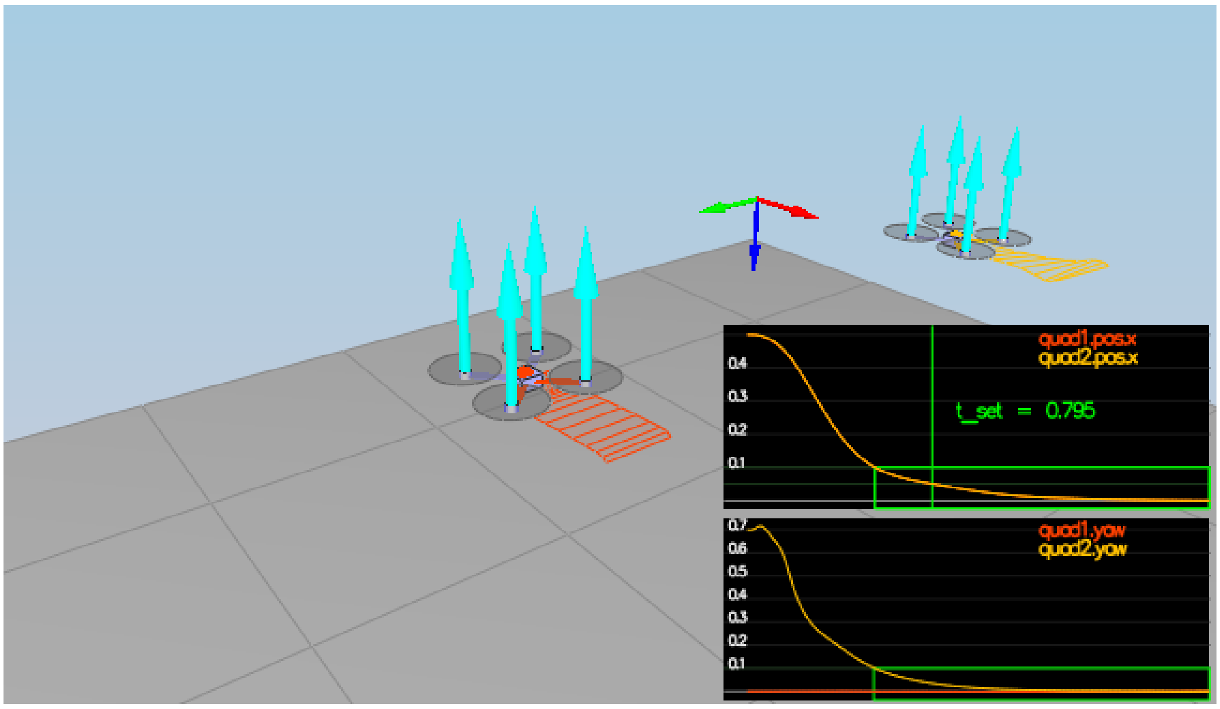
The last step was to implement the code for the yaw control. This was quite straightforward since it was a P controller obtained by multiplying the proportional gain $kpYaw$ with the error between the target and actual yaw. The only point to underline here is that we correct the yaw angle to always be between 0 and 2π.



```
float corrected_yaw_cmd = 0;
if (yawCmd > 0) {
    corrected_yaw_cmd = fmodf(yawCmd, 2 * F_PI);
}
else {
    corrected_yaw_cmd = -fmodf(-yawCmd, 2 * F_PI);
}

float yaw_error = corrected_yaw_cmd - yaw;
if (yaw_error > F_PI) {
    yaw_error -= 2 * F_PI;
}
if (yaw_error < -F_PI) {
    yaw_error += 2 * F_PI;
}

yawRateCmd = kpYaw * yaw_error;
```

Tuning

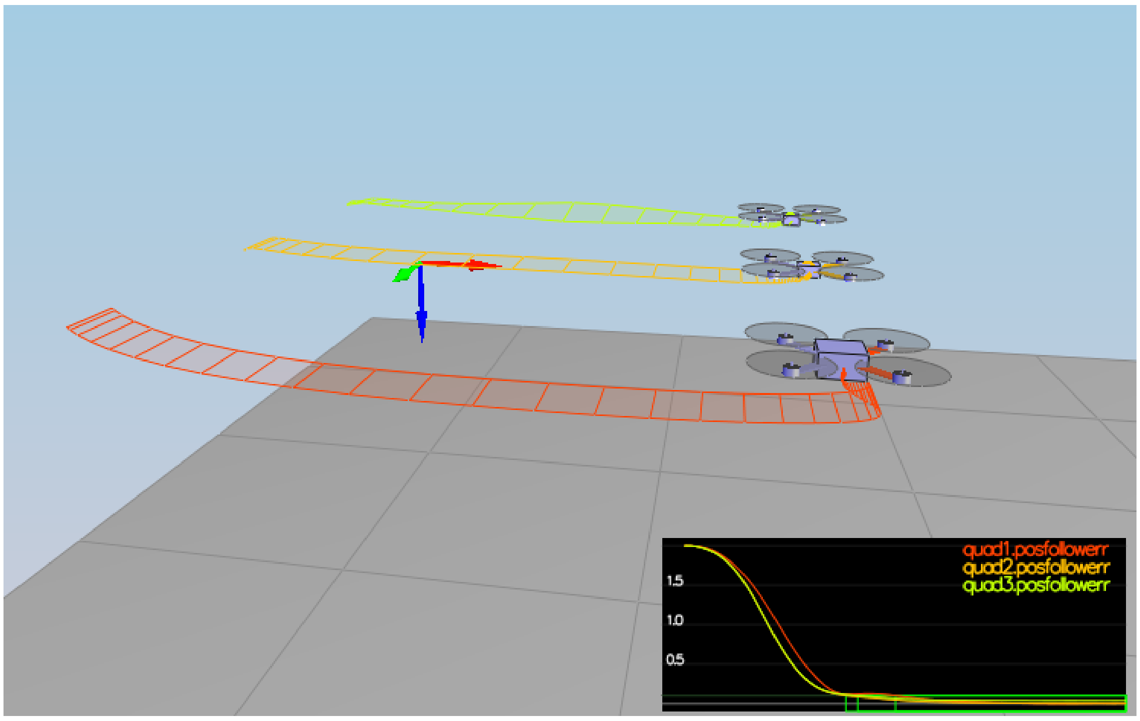
At this point, we started tuning the parameters and we refined the previous gain parameters, obtaining a successful result.



Non-idealities and robustness (scenario 4)

At this point in the project, we had all the controllers tuned and this scenario was designed to test the robustness of the controls, considering different possible non-idealities in the drone. The scenario presented 3 quads, all trying to move one meter forward, with the following non-idealities: the green quad had its centre of mass shifted back, the orange vehicle was an ideal quad, and the red vehicle was heavier than usual.

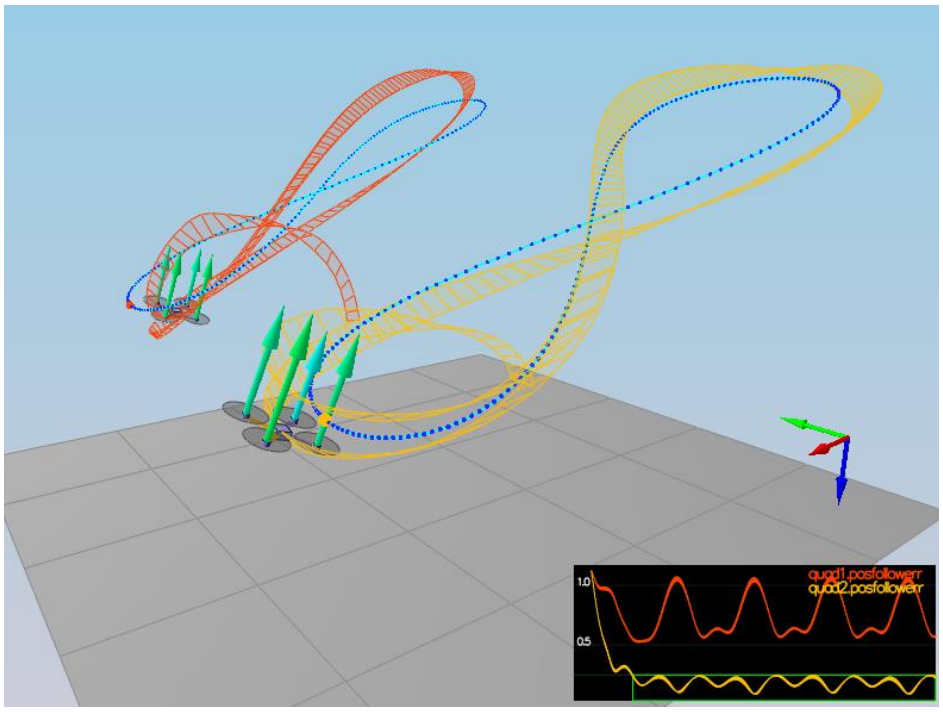
Using the same parameters' gains as before led to poor performance, especially for the red drone with a higher mass. Therefore, further tuning was performed, looking in particular at the gains for the z-position ($kpPosZ$), the z-velocity ($kpVelZ$) and the integral term ($KiPosZ$). After a quite long iterative process, the parameters were tuned leading to a successful result.



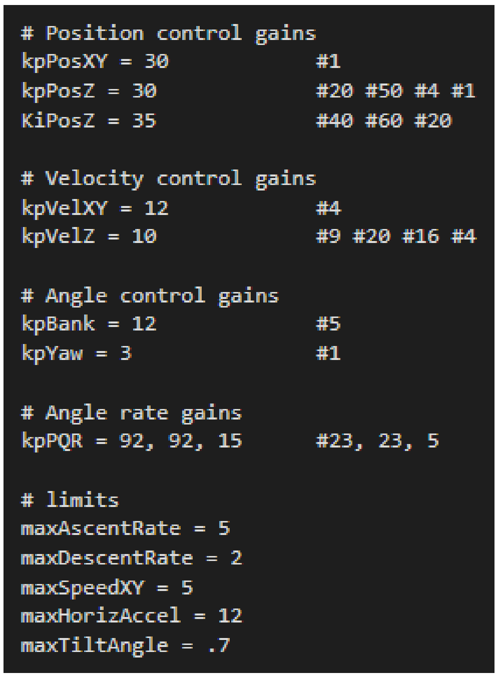
Tracking trajectories (scenario 5)

The last scenario was designed to test once more the implementation of the controllers and the drone's performances in following a given trajectory. The given trajectory was a figure of eight and in one case (`traj/FigureEight.txt`) it was only providing $[time, x_{target}, y_{target}, z_{target}]$ to the drone, while in the second case (`traj/FigureEightFF.txt`), feed-forward parameters were added, leading to a much better performance.

In particular, the txt file provided $[time, x_{target}, y_{target}, z_{target}, \dot{x}_{target}, \dot{y}_{target}, \dot{z}_{target}]$. In the figure below is possible to appreciate the difference between the trajectory without feed-forwarding (in red) and the one with feed-forward parameters (in yellow): we can see that the position tracking error is much lower in the second case (yellow).



At the end of scenario 5, all the gain parameters were tuned as follows:

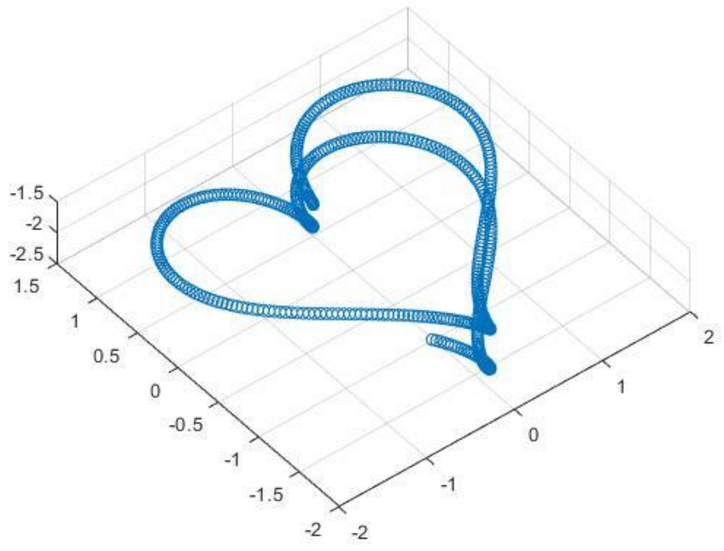


```
# Position control gains
kpPosXY = 30            #1
kpPosZ = 30             #20 #50 #4 #1
KiPosZ = 35             #40 #60 #20

# Velocity control gains
kpVelXY = 12            #4
kpVelZ = 10             #9 #20 #16 #4

# Angle control gains
kpBank = 12             #5
kpYaw = 3               #1

# Angle rate gains
kpPQR = 92, 92, 15      #23, 23, 5

# limits
maxAscentRate = 5
maxDescentRate = 2
maxSpeedXY = 5
maxHorizAccel = 12
maxTiltAngle = .7
```

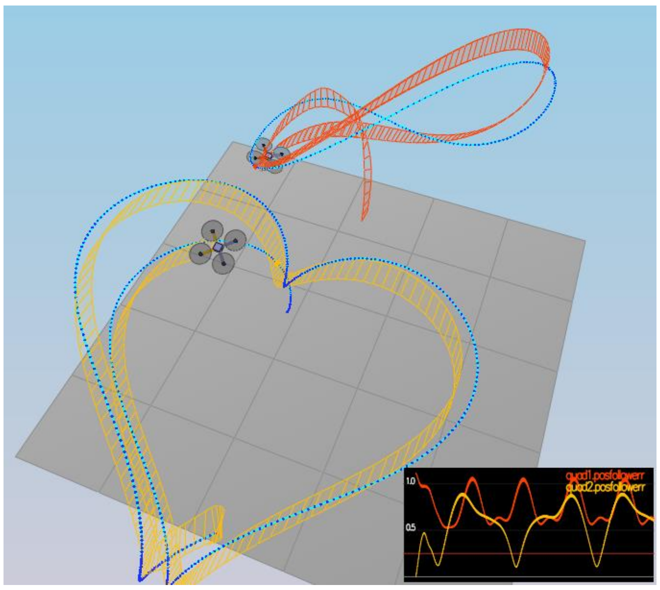
Extra challenge

As extra challenge, I personally decided to generate a trajectory having the shape of a heart to show how much I am enjoying the course. In particular, it was generated using the following functions in Matlab:

```
time_interval = 0:0.02:10;
scaling = 10;
center = [0, 0, -3];
x = (16*sin(t)^3)/scaling + center(1);
y = (13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t))/scaling + center(2);
z = 0.5*sin(2*pi*t/time_interval(end)) + center(3);
```



After recording the values in time of x, y, z and their respective feed-forward parameters, I implemented the heart trajectory in *5_TrajectoryFollow.txt* obtaining the following result.


quad1.posfollowerr
quad2.posfollowerr
1.0
0.5