Dartmouth COSC 89.18/189.02 Assignment 3

Quadrotors: simulation, control, and flight tests

February 2019

1 Introduction

In this assignment, you will learn how to design the controller of a popular robot: quadrotors. In the first part of the assignment, you will develop a rigid body simulator and implement a hierarchical PID controller in simulation. In the second part of this assignment, after you verify the performance of your controller design in simulation, you will deploy it on a popular palm-sized quadrotor: the Parrot Rolling Spider.

2 Background

2.1 The dynamic model

The world and local frames We first establish the world and local frames: our world frame is a right-handed coordinate system with the x axis pointing to the north direction, the y axis pointing to the east direction, and the z axis pointing down. The local frame is a right-handed coordinate system rigidly attached to the quadrotor: the origin of the local frame is the center of mass of the quadrotor, the x axis points along the nose of the quadrotor, the y axis points to the right side of the quadrotor, and the z axis points down. We further assume that the two frames overlap at the beginning of the simulation.

The state of the quadrotor At time t, the state of our quadrotor is fully determined by the position and orientation of our local frame (and its first-order time derivatives). Let $\mathbf{p} \in \mathbb{R}^3$ be the position of the center of mass in the world frame. Let $\mathbf{R} \in \mathbb{R}^{3\times3}$ be the rotational matrix from the local frame to the world frame. In other words, if $\mathbf{v} \in \mathbb{R}^3$ represents a vector in the local frame, then $\mathbf{Rv}$ represents the same vector in the world frame.

While $\mathbf{R}$ has 9 elements, it only has three degrees of freedom because it is a rotational matrix. Typically, $\mathbf{R}$ is defined by three *Euler angles* ϕ (roll), θ (pitch), and ψ (yaw), which represents the rotation along x, y, and z axes of the local frame. Figure 1 explains how the rotational matrix is defined from three Euler angles. In this figure, the green frame on the left is the world frame, and the red frame on the right shows the final local frame.

The actuators A quadrotor has four rotors. Each rotor spins its propeller to generate thrust and the induced torque. The order and spinning direction of our quadrotor is depicted in Figure 2. Let $l \in \mathbb{R}^+$ be the arm length (the distance from the rotor to the center of mass). We can define the position of four rotors in the local frame as follows:

$$\begin{aligned}
\mathbf{d}_1 &= [\frac{l}{\sqrt{2}}, -\frac{l}{\sqrt{2}}, 0] \\
\mathbf{d}_2 &= [\frac{l}{\sqrt{2}}, \frac{l}{\sqrt{2}}, 0] \\
\mathbf{d}_3 &= [-\frac{l}{\sqrt{2}}, \frac{l}{\sqrt{2}}, 0] \\
\mathbf{d}_4 &= [-\frac{l}{\sqrt{2}}, -\frac{l}{\sqrt{2}}, 0]
\end{aligned} \tag{1}$$

The thrust generated by rotor i is defined as $\mathbf{T}_i = [0, 0, -T_i] \in \mathbb{R}^3$. Note that there is a negative sign before T_i because I use $T_i > 0$ to represent the magnitude of thrust but our z axis points down.

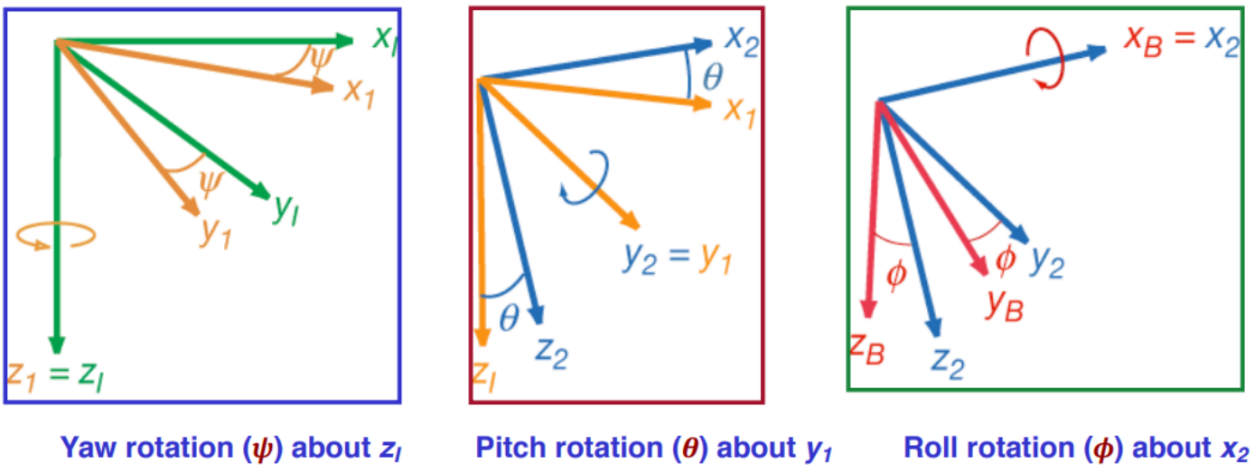

Figure 1: Euler angles. Picture credit: `http://www.stengel.mycpanel.princeton.edu/MAE331Lecture10.pdf`

The governing equations The linear and angular motions of the quadrotor is characterized by the time derivatives of its position and rotational matrix, i.e., $\dot{\mathbf{p}}$ and $\dot{\mathbf{R}}$. The linear motion is governed by the Newton's second law:

$$m\ddot{\mathbf{p}} = m\mathbf{g} + \mathbf{R}\sum_i \mathbf{T}_i \tag{2}$$

where m represents the mass and $\mathbf{g} = [0, 0, 9.81]$ is the gravitational acceleration.

The angular motion is determined by the Euler's equation:

$$\mathbf{I}\dot{\omega} + \omega \times (\mathbf{I}\omega) = \sum_i \mathbf{d}_i \times \mathbf{T}_i + \lambda_i \mathbf{T}_i \tag{3}$$

Here ω is the angular rate, which is connected to the rotational matrix by $\omega \times \mathbf{R} = \dot{\mathbf{R}}$. $\mathbf{I} \in \mathbb{R}^{3\times 3}$ is the inertial matrix in the local frame, which remains constant throughout the simulation. $\lambda_i \mathbf{T}_i$ captures the *spinning torque* induced from each rotor: if a rotor spins clockwise, it will cause the body to spin counterclockwise. $|\lambda_i|$ is the ratio of the magnitudes of the torque and the thrust, which is fully determined by the rotor itself. Since our four rotors are the same, we can use a single $\lambda = |\lambda_i|, \forall i$. In our current rotor setup, $\lambda_1 = \lambda_3 = \lambda$, and $\lambda_2 = \lambda_4 = -\lambda$.

2.2 The sensors

Our quadrotor is equipped with a gyroscope, an accelerometer, a barometer, a sonar, and a camera. All of these sensors work together to fully determine the current statues of the quadrotor. For example, to infer the altitude, we retrieve data from the sonar (the distance to the ground) and the barometer (the surrounding air pressure), and fuse them to gain a more accurate estimation of the altitude. In this assignment, we will skip the sensor fusion process and assume a clean estimate of linear and angular motion has been given, which includes the following: the position of the center of mass $\mathbf{p}$, the velocity of the center of mass $\mathbf{v}$ represented as a vector in the *local* frame (in other words, $\mathbf{R}\mathbf{v} = \dot{\mathbf{p}}$), the Euler angles ϕ, θ, ψ, and the Euler angular rate $\dot{\phi}, \dot{\theta}, \dot{\psi}$.

2.3 The controller

In this assignment, we will learn how to design a position controller that allows the quadrotor to track an arbitrary location in 3D: given any target position $(x, y, z) \in \mathbb{R}^3$, this controller will instruct the quadrotor to fly to that location and stay level as quickly as possible. To help you better understand how it works, we will use a concrete target position $\mathbf{p} = (0, 0, -1.5)$ in the following sections. Note that we implicitly require the quadrotor to hover at this position in a steady state, i.e., $\dot{\mathbf{p}} = (0, 0, 0), \phi = \theta = \psi = 0.0$, and $\dot{\phi} = \dot{\theta} = \dot{\psi} = 0$. We will walk you through a hierarchical controller design in three steps: 1) design an altitude controller to maintain its height, 2) design an attitude controller to stay level in the air, and 3) design an horizontal controller to cancel out any horizontal offsets.

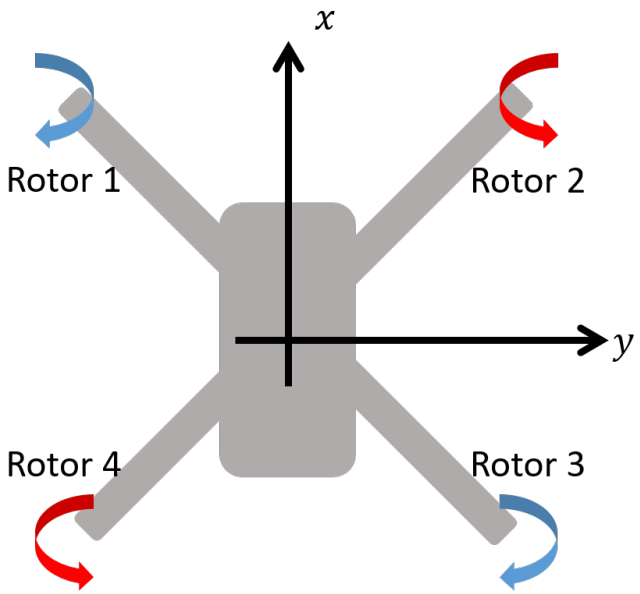

Figure 2: The four rotors and their spinning directions from a top-down view of our quadrotor.

The altitude controller Let's assume that the attitude of our quadrotor is always level ($\phi = \theta = \psi = 0$) and focus on how to ascend/descend to the desired altitude for now. Let $z_{ref} = -1.5$ be our desired height (unit: meter). Let z be the detected altitude from the sensors, which will be provided in both simulation and on the hardware platform. Keep in mind that both z_{ref} and z are negative due to the fact that the z axis points down. We use a PD controller to determine the desired net thrusts from all the four rotors. To derive the correct form, let's first introduce $h = -z$ and $h_{ref} = -z_{ref}$ to represent all the altitude as a positive number, which will make the derivation more intuitive:

$$\begin{aligned} T_z =& mg + P_z(h_{ref} - h) + D_z(\dot{h}_{ref} - \dot{h}) \\ =& mg + P_z(h_{ref} - h) - D_z\dot{h} \end{aligned} \tag{4}$$

Here $P_z > 0$ and $D_z > 0$. As a sanity check, imagine we want the quadrotor to climb, i.e., $h_{ref} - h > 0$, which predicts a T greater than the gravity mg and lift the quadrotor. Now we can replace h with z:

$$T_z = mg - P_z(z_{ref} - z) + D_z\dot{z} \tag{5}$$

This describes how much *net thrust* is needed, but how do we distribute it over four rotors? The answer is to ask each rotor to provide $T_z/4$ thrust. If you revisit the governing equation, you will understand this distribution has the benefit of decoupling altitude control from attitude control: since all four rotors always provide equal thrust, whenever we adjust the altitude, it won't cause extra rotations in any of the roll, pitch, or yaw directions. Now if you abstract our quadrotor as a mass point and ignore its orientation, this mass point can be controlled to climb/descend to desired height.

The attitude controller Let's start with the roll controller: given a desired ϕ_{ref}, we want to predict what changes to the four thrusts are needed to set $\phi = \phi_{ref}$ as fast as possible. The trick is to generate torques along the local x axis without disturbing the rotation along the other two axes and the linear motion, and here is how: consider adding δT_ϕ to T_1, T_3 and subtracting δT_ϕ from T_2, T_4. The net thrust remains unchanged, so it won't affect the linear motion. We argue this change has the desired effects on the angular motion because 1) it induces a torque $2\sqrt{2}l\delta T_\phi$ along x to change ϕ and 2) the induced torque has zero value along the y and z axes, so it won't affect the motion of pitch and yaw. The actual value of δT_ϕ can be determined by a PID controller on the error of roll ($\phi_{ref} - \phi$). We will skip the details of its implementation for simplicity.

Similarly, the pitch controller can be realized by a PID controller on the pitch error and adding the predicted δT_θ to T_1, T_2 and subtracting δT_θ from T_3, T_4. The yaw controller is implemented by adding δT_ψ to T_2, T_4 and subtracting it from T_1, T_3.

We now summarize the expressions of each T_i below:

$$\begin{aligned} T_1 =& \frac{1}{4}T_z + \delta T_\phi + \delta T_\theta - \delta T_\psi \\ T_2 =& \frac{1}{4}T_z - \delta T_\phi + \delta T_\theta + \delta T_\psi \\ T_3 =& \frac{1}{4}T_z - \delta T_\phi - \delta T_\theta - \delta T_\psi \\ T_4 =& \frac{1}{4}T_z + \delta T_\phi - \delta T_\theta + \delta T_\psi \end{aligned} \tag{6}$$

We further connect the induced torque to the change of thrust:

$$\begin{aligned} \tau_\phi =& 2\sqrt{2}l\delta T_\phi \\ \tau_\theta =& 2\sqrt{2}l\delta T_\theta \\ \tau_\psi =& 4\lambda\delta T_\psi \end{aligned} \tag{7}$$

In this assignment, we first use a PID controller to predict the desired torque, then translate them into the corresponding δT and send it to the simulator. As a concrete example, τ_ψ is defined by:

$$\tau_\psi = P_{yaw}(\psi_{ref} - \psi) - D_{yaw}\dot{\psi} \tag{8}$$

The roll and pitch controllers are defined similarly but with an additional integral controller.

We reiterate that this controller design decouples the control of height, roll, pitch, and yaw in a clean way in the sense that updating one controller signal does not affect the other three. This simple but powerful controller design is one of the core reasons why quadrotors are more popular than other types of aerial robots these days.

The horizontal controller Let's revisit our original task: by setting $z_{ref} = -1.5$ and $\phi = \theta = \psi = 0$, our controller should be able to level itself and maintain its height. Everything works fine except that the horizontal location is still uncontrolled: imagine we perturb the quadrotor by pushing it sideways a little bit, it will then gain a horizontal velocity and drift away but still stay level and maintain the altitude. As a result, the existing altitude and attitude controllers won't be able to correct itself from drifting.

To resolve this issue, we need to reuse our attitude controller in a more creative way: assuming we want to reset the location of quadrotor to somewhere in front of it, we can assign $\theta_{ref} < 0$ so that the quadrotor leans forward. This will cause the quadrotor to advance. We can gradually decrease θ_{ref} to 0 as the quadrotor gets closer to the target. In practice, θ_{ref} is usually implemented by a PD controller on the x error in the local frame. Similarly, the y error is typically corrected by setting ϕ_{ref} from a PD controller. The two PD controllers usually have the same gains (with different signs) because of the symmetric design of the quadrotor.

There is one last caveat before we finalize this idea: recall that the local and world frames may not align during the flight (e.g., after we change its heading), we need to first convert the position error from the world frame to the local frame. Since the quadrotor almost always stays level, we will simply spin the position error vector along the z axis by yaw only and gently ignore the rotations from nonzero pitch and roll. The rotational matrix is given below:

$$\mathbf{R}_\psi = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix}. \tag{9}$$

You can verify that $\mathbf{R}_\psi$ indeed converts the xy components of a vector in the world frame to the xy coordinates in the local frame, assuming we ignore the roll and pitch rotations.

Now let $\mathbf{p}_{xy} \in \mathbb{R}^2$ be the offset vector pointing from quadrotor to the target horizontal position in the world frame. In other words, $\mathbf{p}_{xy}$ is the drift we want to correct. We now explain the work flow of the x controller in detail:

- Convert $\mathbf{p}_{xy}$ into the local frame and get the error along the local x axis: $x_{err} = (\mathbf{R}_\psi \mathbf{p}_{xy})$.
- Use $-v_x = -(\mathbf{v})_x$ to approximate $\dot{x}_{err}$. Recall that $\mathbf{v}$ is the linear velocity in the local frame.
- Design a PD controller to estimate the desired θ_{ref}:

$$\theta_{ref} = -P_{xy}x_{err} - D_{xy}\dot{x}_{err} \approx -P_{xy}x_{err} + D_{xy}v_x \tag{10}$$

Here P_{xy} and D_{xy} are positive gains shared by the x and y PD controllers. Note that we use negative signs before P_{xy} and D_{xy} because we need negative θ to decrease x_{err}. The derivation of the y controller is similar, and we leave it as an exercise.

2.4 The complete loop

Given all the information above, we now summarize what happens at each time step in simulation below:

- At the beginning of each time step, retrieve the state $\mathbf{p}$, $\dot{\mathbf{p}}$, $\mathbf{R}$, and ω. These variables are only used to simulate the rigid body of the quadrotor and provide data to the sensors. In particular, our controller should never access them.
- Simulate the sensors by computing $\mathbf{v}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}$. These variables along with $\mathbf{p}$ will be fed into the controller and get the predicted thrust $\mathbf{T}_i$ for each rotor.
- Use $\mathbf{T}_i$ to compute $\ddot{\mathbf{p}}$ and $\dot{\omega}$ by following Equation (2) and (3).
- Use $\ddot{\mathbf{p}}$ and forward Euler time integration to update $\mathbf{p}$ and $\dot{\mathbf{p}}$:

$$\begin{aligned} \mathbf{p} =& \mathbf{p} + \Delta t \dot{\mathbf{p}} \\ \dot{\mathbf{p}} =& \dot{\mathbf{p}} + \Delta t \ddot{\mathbf{p}} \end{aligned} \tag{11}$$

- Use $\dot{\omega}$ and forward Euler time integration to update $\mathbf{R}$ and ω:

$$\begin{aligned} \mathbf{R} =& \texttt{AxisAngle}(\omega \Delta t)\mathbf{R} \\ \omega =& \omega + \Delta t \dot{\omega} \end{aligned} \tag{12}$$

  where `AxisAngle(n)` represents a rotation matrix which spins along the direction of $\mathbf{n}$ by an angle of $|\mathbf{n}|$.

3 Simulation

In this section, you will implement the aforementioned rigid body dynamics and the position controller in simulation. We have provided you some skeleton code, including the quadrotor model, the sensors, and the control loop. With the initial code, the virtual quadrotor simply stays at the origin forever. After you finish this part, you will see it take off from the ground, fly to a target position in 3D, and stay there stably given the existence of moderate environmental noises.

All of your implementation will be in `MultiCopter.h`.

3.1 The linear motion

The first task for you is to implement the Newton's second law as described in Equation (2) inside the `Advance_Rigid_Body` function:

```
// -- LV1 TASK: 3.1: update linear velocity --
rigid_body.position = old_p;
rigid_body.velocity = old_v;
VectorD net_force=VectorD::Zero();

// -- Your implementation starts --
// -- Your implementation ends --
```

Your task is to update `net_force` so that it matches the right hand side of Equation (2). The two `real` variables `mass` and `g` (not shown in the code snippet) define the mass and gravitational acceleration respectively. The rotational matrix $\mathbf{R}$ in the equation is stored in a `MatrixD` variable `old_R`. We have also initialized an `std::vector` called `thrust_vec` for you, where each `thrust_vec[i]` corresponds to $\mathbf{T}_i$ in the equation.

After you implement this part, you will see that the quadrotor first takes off then oscillates along the z axis with the orientation unchanged, i.e., it translates like a single mass point.

3.2 The angular motion

Next, we will add the angular motion to the quadrotor simulator based on Equation (3):

```
// -- LV1 TASK 3.2 --
rigid_body.orientation = old_orientation;
rigid_body.omega = old_omega;
VectorD net_torque = VectorD::Zero();

// -- Your implementation starts --
// -- Your implementation ends --
```

Your task is to update the `net_torque` so that it matches the right hand side of Equation (3). For each rotor, its location in the local frame $\mathbf{d}_i$ is stored in a `VectorD` variable called `rotor_pos[i]`. The value λ in the equation is stored in a `real` variable called `lambda` in the code. Keep in mind that we define $\lambda_1 = \lambda_3 = \lambda$ and $\lambda_2 = \lambda_4 = -\lambda$.

After you finish this part, you can verify it by commenting out your implementation of the linear motion. In this case, you should see the quadrotor spin along the x axis. Do not forget to uncomment your linear motion for the remaining tasks!

3.3 The altitude controller

Now we will walk you through the process of implementing the hierarchical controller to stabilize its motion. Let's start with the altitude controller described in Equation (5):

```
////LV2: PD Controller
real AltitudeController(const real total_weight, const real z_ref, const real z,
    const real z_rate)
{
  const real P_z = 0.2;
  const real D_z = 0.3;
  // -- TASK 3.3: Implement your altitude controller here --
  real total_thrust = 0.0;

  // -- Your implementation starts --
  // -- Your implementation ends --

  return total_thrust;
}
```

Here `total_weight` is a positive number equal to mg, `z_ref` is a negative number representing the reference altitude, `z` is the altitude from the sensor, and `z_rate` is the time derivative of `z`, which corresponds to $\dot{z}$ in the equation. The function should return a positive `total_thrust` that matches T_z in the equation.

Put your implementation inside this function. Once it is done, you should see your quadrotor take off, rise to the target height, and maintain its altitude. You will also notice it self spins along the z axis, which we will fix shortly. As a sanity check, you can set `D_z=0` and will notice the height of the quadrotor oscillates, a typical phenomenon of a P controller.

3.4 The attitude controller

Next, let's fix the self-spinning by implementing a yaw controller in the following function:

```
real YawController(const real yaw_ref, const real yaw, const real yaw_rate)
{
  const real P_yaw = 0.004;
  const real D_yaw = 0.3 * 0.004;
  // -- TASK 3.4 --
  // Implement your yaw controller here.
  real total_torque = 0.0;

  // -- Your implementation starts --
  // -- Your implementation ends --
  return total_torque;
}
```

You need to calculate and return the correct `total_torque` based on Equation (8). The variables `yaw_ref`, `yaw`, `yaw_rate` correspond to ψ_{ref}, ψ, and $\dot{\psi}$ in the equation respectively. We have tweaked the P and D gain (`P_yaw` and `D_yaw`) for you. Your task is to calculate τ_ψ and assign it to `total_torque`.

After you finish this function, the quadrotor should take off, maintain its altitude and heading, and counteract any horizontal drift caused by the noises (For simplicity, we have already implemented the horizontal controller for you). This concludes all your simulation tasks.

4 Lab

In this section, you will learn how to deploy your controller on a real hardware platform called Parrot Rolling Spider. We have set up the framework in Simulink and will give each group a Rolling Spider. Your task is to rewrite the controller in Simulink using the MATLAB programming language, and we will take care of flashing your code to the hardware and help you verify the performance in real flight tests. Once you successfully finish the three tasks, your controller should behave roughly the same as the one in this video.

4.1 The altitude controller

For this task, we have isolated the other components and provided you the following code skeleton in Simulink:

```
function total_thrust = altitude_controller(total_weight, z_ref, z, z_rate)
P_z = 0.2;
D_z = 0.3;
% -- STUDENT VERSION --
% Implement your altitude controller here.
total_thrust = 0.0;
% -- END OF STUDENT VERSION --
total_thrust = single(total_thrust);
```

Here `total_weight` is a positive number equal to mg, `z_ref` is a negative number representing the reference altitude, `z` is the sensed altitude, and `z_rate` is the sensed velocity, i.e., the time derivative of `z`. The function returns a positive `total_thrust`, which should be implemented the same way as defined in Equation (5).

To help you better understand the necessity of the D gain, we recorded a video where we intentionally removed the D gain from the altitude controller. As you can see in this video, with a single P controller, the quadrotor oscillated vertically in the air.

4.2 The attitude controller

For this task, we have implemented the pitch and roll controllers for you in Simulink. The remaining task for you is to implement the yaw controller by filling the following function:

```
function tau_yaw = yaw_controller(yaw_ref, yaw, yaw_rate)
P_yaw = 0.004;
D_yaw = 0.3 * 0.004;
% -- STUDENT VERSION --
% Implement your yaw controller here.
tau_yaw = 0;
% -- END OF STUDENT VERSION --
tau_yaw = single(tau_yaw);
```

Here you are given the desired yaw angle, the sensed yaw, and its rate. The return value of this function is the expected torque to steer yaw. Please refer to Equation (8) for details.

The initial yaw controller simply sets the torque to zero, meaning that it cannot counteract any perturbation to the yaw. Here is a demo showing how it behaves. Your implementation should be able to cancel out this self spinning behavior.

4.3 The horizontal controller

Finally, you need to implement the horizontal controller in the following function:

```
function pitch_roll_cmd = xy_controller(yaw, xy_ref, xy, v)
xy_error = xy_ref - xy;
% Step 1: Convert diff vector from world frame to local frame.
% -- STUDENT VERSION --
% Implement the world to local frame conversion.
R = eye(2);
% -- END OF STUDENT VERSION --
local_xy_error = R * xy_error;

% Clamp.
local_xy_error = max(-3.0, min(3.0, local_xy_error));
x_err = local_xy_error(1);
y_err = local_xy_error(2);

% Step 2: PD controller.
P = 0.24;
D = 0.1;
dx_err = -v(1);
dy_err = -v(2);
% -- STUDENT VERSION --
% Implement the desired pitch and roll angles.
pitch = 0;
roll = 0;
% -- END OF STUDENT VERSION --
pitch_roll_cmd = single([pitch, roll]');
```

This function consists of two steps. In step 1, you need to implement the rotational matrix in Equation (9) to convert the position error from the world frame to the local frame. We have initialized an 2×2 identity matrix for you. You can use `R(i,j)` to access the element in the `i`-th row and `j`-th column (`i`,`j`$\in \{1, 2\}$).

In step 2, you need to implement two PD controllers to predict the desired pitch and roll angles. The PD controllers share the same P and D gains but with possibly different signs. We have provided the expression for pitch in Equation (10), and it is your task to derive the correct expression for roll before implementing them in Simulink.

As before, we have a video to show the performance of the initial xy controller. Your implementation should be able to counteract the horizontal drift and stay roughly at the target position.