



## Aerial robotics

# Cascade control of a quadrotor

---

*The goal of this exercise is to design and implement attitude and position control of a quadrotor in the Gazebo simulator and to test position control of a real AR.Drone Parrot vehicle. Control algorithms should be implemented in Python and within Robot Operating System (ROS).*

---

NAME

JMBAG

## 1 Introduction

### 1.1 System requirements

To be able to work on this seminar, you should use a PC with Linux distribution installed (recommended Ubuntu 16.04), which supports ROS Kinetic and Gazebo simulator (version 7 default for ROS Kinetic).

Once you have set up your system, install ROS Kinetic (desktop-full version) and Gazebo simulator. Installation instructions can be found in [5]. In addition, run the following commands to install packages required by the simulator

```
$ sudo apt-get install ros-kinetic-joy ros-kinetic-octomap-ros
$ sudo apt-get install ros-kinetic-mavlink python-wstool
$ sudo apt-get install python-catkin-tools protobuf-compiler
$ sudo apt-get install ros-kinetic-control-toolbox
$ sudo apt-get install ros-kinetic-dynamic-reconfigure
$ sudo apt-get install libgoogle-glog-dev python-pip
$ sudo pip install numpy
```

Now prepare a new ROS workspace where the source code will be placed. In your home folder created a folder (usually named `catkin_ws` but you can call it as you wish), where the workspace will be stored. Enter the newly created folder and run the following commands to initialize the workspace [1]

```
$ catkin init
$ catkin init --workspace .
$ catkin config --init
$ mkdir src & catkin build
```

Add environmental variable `ROS_WORKSPACE` in your `.bashrc` file to point to the top folder of your newly created ROS workspace (the folder that you first created when initializing the ROS workspace). In your terminal execute the following command

```
$ echo 'export ROS_WORKSPACE=<path>' >> ~/.bashrc ,
```

where `< path >` is the actual path to your workspace.

You are ready to download the necessary packages for simulator and controllers from the Github repository. Make sure that you switch to the correct branch (*urs2017*) in each repository. Change your current path to the `ROS_WORKSPACE/src` folder and run

```
$ git clone https://github.com/larics/rotors_simulator.git
$ git clone https://github.com/larics/mav_comm.git
$ git clone https://github.com/larics/urs_aerial.git
$ cd rotors_simulator & git checkout urs2017
$ cd ../mav_comm & git checkout urs2017
$ cd ../urs_aerial & git checkout urs2017
```

Before building the workspace, make sure that all files in folder *urs\_solution/cfg* are executable. If they are not, make them executable by running the following command for each file

```
$ chmod u+x <file>
```

You are ready to build your workspace by running *catkin build* command in your `$ROS_WORKSPACE` folder:

```
$ catkin build
```

Make sure that you get no errors. If you get an error that you don't manage to fix in a couple of hours, please report it on the assistant's mail.

Finally, set up your environmental variable `PYTHONPATH` to indicate the path of some necessary python files

```
$ echo 'export PYTHONPATH=$PYTHONPATH:$ROS_WORKSPACE/src/
urs_aerial/urs_solution/Task1' >> ~/.bashrc
```

Source `.bashrc` file in your open terminals:

```
$ source ~/.bashrc
```

When you open a new terminal you are not required to repeat the last command, as it is automatically executed on terminal start-up.

## 1.2 Getting started

If everything is properly installed in the previous step, you are ready to start the simulator. If you run the Gazebo simulator for the first time, start *roscore* in one terminal and Gazebo server in the second one.

The first terminal:

```
$ roscore
```

The second terminal:

```
$ gzserver
```

Wait for few minutes to allow Gazebo to download models. It seems that the download procedure does not terminate automatically, so kill the process after few minutes with *Ctl-Z* command.

Now you are ready to launch the simulator and spawn the quadrotor. Run the following command in your terminal:

```
$ roslaunch urs_launch gazebo_ardrone.launch
```

You should now have a Gazebo simulator with a spawned AR.Drone quadrotor running. You can check available ROS topics using command:

```
$ rostopic list
```

You should see several topics with the names containing `/ardrone` prefix.

### 1.3 Useful ROS tools

You are encouraged to use any ROS tool available to facilitate working on this seminar. You can use `rqt_gui` as a GUI for interacting with the simulation. In particular, use plugin Visualization/Plot (type `PyQtGraph`) for plotting signals of interest and plugin Topics/Message Publisher for publishing topics of interest (e.g. reference values for controllers). To run the GUI, type:

```
$ rosrun rqt_gui rqt_gui
```

You can use `rosbag` to record data of interest. Bag files can be used to store recorded data in txt file. E.g., if you want to save the topic `/ardrone/pose`, use command:

```
$ rosbag record /ardrone/pose
```

On the other hand, if you want to record all topics, use command:

```
$ rosbag record -a
```

If you want to save the topic `/ardrone/pose`, previously recorded with `rosbag`, in a txt file, which can be easily imported in MATLAB or Octave, use:

```
$ rostopic echo -b <bag_file> -p /ardrone/pose > pose.txt
```

PID controller parameters in this seminar are dynamically reconfigurable which is useful for fine tuning. Use `rqt_reconfigure` to access and modify parameters online:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Detailed instructions on many ROS tools can be found in Lectures of the course Programming for the Robot Operating System [4].

### 1.4 Mathematical prerequisites

#### 1.4.1 Model of the quadrotor

In this section we describe the model of the quadrotor [3],[2] used in the Gazebo simulator. You will use this model in the design of quadrotor control algorithms.

First, note the coordinate system assigned to the quadrotor body (Fig. 1.4.1). As a 3D rigid body, quadrotor pose is described by 6 parameters - x,y,z position and 3 Euler angles. Throughout this seminar, when we refer to the quadrotor position, we assume the position relative to the inertial coordinate frame (i.e. global coordinates w.r.t to the Gazebo origin). For the Euler angles, we use notation yaw-pitch-roll, which means that we assume that the quadrotor is first rotated around body z-axis (this is also a global z-axis), followed by a rotation around body y-axis, and finally, a rotation around body x-axis.

A quadrotor motor in the simulator is modelled as:

$$T_m \cdot \dot{\Omega}_i + \Omega_i = \Omega_{i,ref} \quad i=1,2,3,4, \quad (1)$$

where  $\Omega_i$  is the rotational velocity of the i-th motor (in rad/sec),  $T_m$  is the motor time constant and  $\Omega_{i,ref}$  is a motor reference velocity. We assume that time constants of all motors are equal. The thrust force  $F_i$  that each motor produces is:

$$F_i = b_f \cdot \Omega_i^2 \cdot \hat{k}, \quad (2)$$

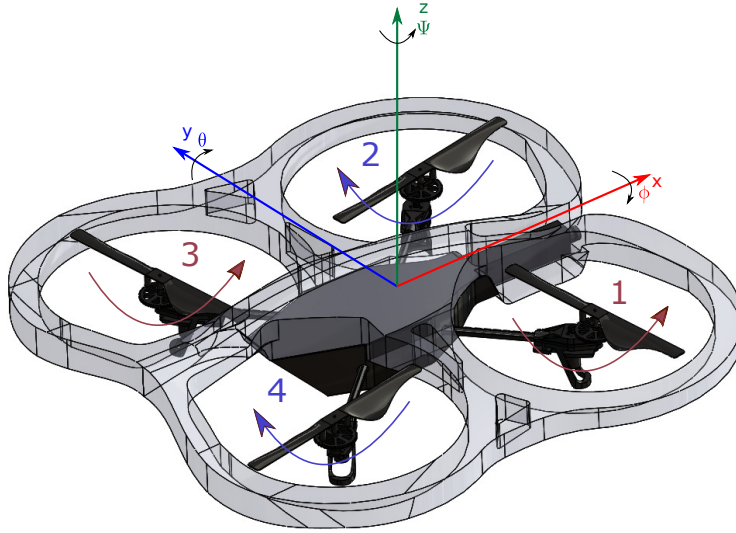


Figure 1: Coordinate frame assigned to the quadrotor body (body frame). The origin of the frame coincides with the center of the mass. Note the numbering of the motors and their direction of rotation.

$b_f$  is a motor thrust constant,  $\hat{k}$  is a unit vector in the direction of the body z-axis.

Each motor also produces moment  $M_i$  (due to induced drag):

$$M_i = \zeta_i b_f b_m \cdot \Omega_i^2 \cdot \hat{k}, \quad \zeta_i = 1 \quad (i=2,4) \text{ or } -1 \quad (i=1,3), \quad (3)$$

$b_m$  is a motor moment constant and  $\zeta_i$  indicates whether the propeller rotates clockwise ( $\zeta = 1$ ) or counter-clockwise ( $\zeta = -1$ ).

The change of the quadrotor attitude is described by the following equation:

$$\frac{d}{dt} (\mathbf{I} \cdot \boldsymbol{\omega}) = -\boldsymbol{\omega} \times (\mathbf{I} \times \boldsymbol{\omega}) + \mathbf{M}_{sum}, \quad (4)$$

where  $\mathbf{I}$  is the quadrotor tensor matrix w.r.t. the center of the mass,  $\boldsymbol{\omega}$  is the quadrotor angular velocity in the body frame (see Fig. 1.4.1),  $\mathbf{M}_{sum}$  is the sum of all external moments acting on the vehicle.

We assume that the quadrotor is symmetrical, which results in a diagonal tensor matrix (i.e.  $I_{xy} = I_{xz} = I_{yz} = 0$ ):

$$\mathbf{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \quad (5)$$

The equations describing the angular velocity change in x, y and z direction can be now written as:

$$I_{xx} \dot{\omega}_x = (I_{yy} - I_{zz}) \omega_y \omega_z + (F_2 + F_3 - F_1 - F_4) \cdot l \cos(45) \quad (6)$$

$$I_{yy} \dot{\omega}_y = (I_{zz} - I_{xx}) \omega_x \omega_z + (F_3 + F_4 - F_1 - F_2) \cdot l \cos(45) \quad (7)$$

$$I_{zz} \dot{\omega}_z = M_1 + M_2 + M_3 + M_4 \quad (8)$$

Table 1: Model parameters

Symbol	Value and unit	Description
$m$	1.477 kg	Total quadrotor mass
$I_{xx}$	0.01152 kg·m <sup>2</sup>	Quadrotor moment inertia in body x direction
$I_{yy}$	0.01152 kg·m <sup>2</sup>	Quadrotor moment inertia in body y direction
$I_{zz}$	0.0218 kg·m <sup>2</sup>	Quadrotor moment inertia in body z direction
$T_m$	0.0125 s	Time constant of a motor
$b_f$	8.54858e-06 kg · m	Thrust constant of a motor
$b_m$	0.016 m	Moment constant of a motor
$l$	0.18 m	The distance of a motor from the center of mass

Note that moments of inertia in x and y axis are equal (Table 1). Therefore, gyroscopic term in Equ. 8  $((I_{xx} - I_{yy})\omega_x\omega_y)$  equals zero and it is omitted.

As we derive the model around the hovering state, the friction in Equ. 6, 7, 8 is neglected, which is a common approach when designing a control law for a quadrotor. The change of the Euler angles (yaw -  $\psi$ , pitch -  $\theta$ , roll -  $\phi$ ) is described as:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi) \tan(\theta) & \cos(\phi) \tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \quad (9)$$

where  $\dot{\phi}, \dot{\theta}, \dot{\psi}$  are roll rate, pitch rate and yaw rate, respectively.

We derive the quadrotor position model in the inertial coordinate frame (corresponds to the Gazebo static frame):

$$m\dot{v}_x = F_z^b (\cos(\phi) \sin(\theta) \cos(\psi) + \sin(\phi) \sin(\psi)) \quad (10)$$

$$m\dot{v}_y = F_z^b (\cos(\phi) \sin(\theta) \sin(\psi) - \sin(\phi) \cos(\psi)) \quad (11)$$

$$m\dot{v}_z = F_z^b \cos(\phi) \cos(\theta) - mg \quad (12)$$

where  $m$  is the quadrotor mass,  $F_z^b$  is total thrust force in body frame (z-direction):

$$F_z^b = F_1 + F_2 + F_3 + F_4, \quad (13)$$

To get the position in the inertial coordinate frame, simply integrate the velocity:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \quad (14)$$

Parameters used in the Gazebo model can be found in Table 1.

### 1.4.2 PID control algorithm

In this section we describe a general PID control algorithm, which is the basis of the cascade control that you should implement within this seminar.

PID controller in a continuous time is given by:

$$u(t) = u_p + u_i + u_d = K_p \left( e(t) + \frac{1}{T_i} \int_0^t e(t) dt + T_d \frac{de(t)}{dt} \right), \quad (15)$$

where  $u(t)$  is computed control value,  $u_p$ ,  $u_i$  and  $u_d$  are proportional, integral and derivative components of PID algorithm,  $e(t)$  is the error signal between reference and measured values of the quantity that we want to control,  $K_p$  is proportional gain,  $T_i$  integral time constant and  $T_d$  derivative time constant.

If we use a ZOH transformation to discretize each PID component independently (parallel version of PID), we get:

$$u_p(k) = K_p \cdot e(k) \quad (16)$$

$$u_i(k) = u_i(k-1) + \frac{K_p}{T_i} e(k) T_s = u_i(k-1) + K_i \cdot e(k) \cdot T_s \quad (17)$$

$$u_d(k) = K_p T_d \frac{e(k) - e(k-1)}{T_s} = K_d \cdot \frac{de(k)}{T_s}, \quad (18)$$

where  $K_i$  is integral gain,  $K_d$  is derivative gain,  $T_s$  is the algorithm sampling time (i.e. time between two consecutive steps of the algorithm),  $de(k)$  is a difference of the error in two consecutive steps.

Note that when you design PID for a particular part of the system, some of the gains can be (and should be) set to 0 (therefore you get P, PI or PD controller).

## 2 Exercises

In this seminar your task is to design and implement a full position and attitude control of an AR.Drone quadrotor in the Gazebo simulation environment. The second part of the seminar is to design and implement a position control of a real AR.Drone vehicle. You are given template file for each exercise and your task is to complete the code with designed parameters and control algorithm. Parts of the code responsible for ROS communication is written for you.

You should complete the simulation part of the seminar on your PC, while for the second part, you will test your algorithms on a practical session. Please check course web page for the practicals schedule.

As a part of your report, you will have to complete this file with your parameters (as a text field data) and signal responses (as figure attachments). Note that if you open this file in a standard Ubuntu pdf readers, such as Okular or Evince, you won't be able to fill the text fields nor add attachments. Therefore it is recommended to open this file in Adobe Reader. As the second part of your report, send zipped folder *urs\_solution* where completed code should be contained. Again, please check course web page announcements for deadline.

### 2.1 Task 1 - PID implementation in Python

In this task you are required to implement a Python discrete version of the PID algorithm. You are given a template file *pid\_tpt.py* in package *urs\_solution* (subfolder *src/Task1*). Rename the file to *pid.py* and complete the method *compute* of the provided class *PID*. Make sure that your file *pid.py* is executable. To make it executable run:



```
$ chmod u+x pid.py
```

Once finished, run python code *test\_pid.pyc* which is located in the same folder. As a result, you should get a figure with plotted signals (error signal generated by this code and proportional, integral and derivative component generated by your PID implementation). Save the figure as a picture (.jpeg, .png or similar). As a part of your report, attach your file *pid.py* and the resultant figure in Table 2. Please note that the limits of the controller are set to 1 and  $-1$ .

To be sure that your PID implementation is valid, check the figure which you get for the following properties:

- constant  $e \rightarrow u_p$  is constant,  $u_i$  is linearly rising,  $u_d$  is zero
- linearly rising  $e \rightarrow u_p$  is linearly rising,  $u_i$  is quadratically rising,  $u_d$  is constant
- $e$  is zero  $\rightarrow u_p$  is zero,  $u_i$  is constant,  $u_d$  is zero

Table 2: Task 1 results.

pid.py	
PID test response	

## 2.2 Task 2 - Height and attitude control

The goal of this task is to implement attitude control and height control. Since it's difficult to test attitude control without the quadrotor being airborne, and as well as to tune height control without implemented attitude control, you should implement and test these two controllers more or less simultaneously.

### 2.2.1 Task 2a - Height control

The first subtask is to implement cascade PID control for quadrotor height (see Fig. 2.2.1). The common approach in designing PID gains is to first determine theoretical values based on the mathematical model, followed by fine tuning during experimental validation. Use the same approach here. First, linearize the quadrotor model around hovering state. In hovering state we assume small Euler angles, therefore you can approximate trigonometric functions in Equ. 12:

$$\cos \phi = 1 \quad (19)$$

$$\cos \theta = 1. \quad (20)$$

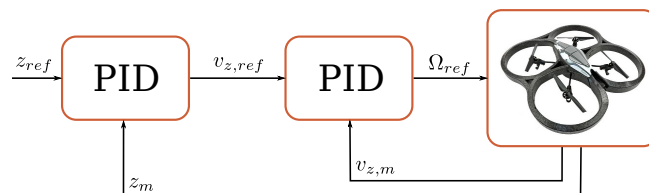


Figure 2: Cascade PID control for quadrotor height.

Next, determine the hovering motor velocity  $\Omega_0$  (the velocity of the motors at which total thrust force equals the vehicle gravitational force) from Equ. 12. Linearize the same equation around hovering velocity to get the transfer function  $\frac{\Delta v_z(s)}{\Delta \Omega_{ref}(s)}$ . One should be aware that  $\Delta \Omega_{ref}$  is the control value for this part of the system (to increase the quadrotor height we increase the each motor velocity by  $\Delta \Omega_{ref}$  and vice versa). However, note that from Equ. 1 we get (using  $\Delta \Omega(s) = \Omega(s)$  as Equ. 1 is linear) :

$$\Delta \Omega_i(s) = \frac{\Delta \Omega_{i,ref}(s)}{T_m \cdot s + 1}, \quad (21)$$

with  $\Delta \Omega_{1,ref} = \Delta \Omega_{2,ref} = \Delta \Omega_{3,ref} = \Delta \Omega_{4,ref} = \Delta \Omega_{ref}$ .

Afterwards, design gains of inner PID in Fig. 2.2.1, according to any control design method that you have learned during your study (for the definition of PID gains refer to Equ. 16, 17, 18). Report obtained values below. In the row "Control design method", write which control method you used (e.g. Ziegler-Nichols, technical optimum, symmetrical optimum, optimization in Matlab/Octave, etc.). If you have any additional note to add for you control method, please add it as an attached comment.

Table 3: Velocity ( $v_z$ ) controller design parameters

Hovering velocity $\Omega_0$ (rad/s)	
Transfer function $\frac{\Delta v_z(s)}{\Delta \Omega_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Now, use similar approach to design gains of outer PID (Fig. 2.2.1), based on the open loop tranfer function  $\frac{\Delta z(s)}{\Delta v_{z,ref}(s)}$ . Report your results in Table 4.

Table 4: Height controller design parameters

Transfer function $\frac{\Delta z(s)}{\Delta v_{z,ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Once finished with theoretical synthesis, implement your PID cascade in Python. You will of course utilize implemented *PID* class. Find a template file in ROS package *urs\_solution*, *Task2a/height\_ctl.tpt.py*. Save the file in the same directory under the name *height\_ctl.py* (check if the file is executable) and add your code in two methods of the prepared class. First, in *init* method add values of designed PID gains. Second, complete *run* method with your cascade PID implementation. To test your control, run simulator and controller:

```
$ roslaunch urs_launch gazebo_ardrone.launch
$ roslaunch urs_launch task2a.launch
```

To test your controller, you can publish height reference value to the topic */ardrone/pos\_ref*. If you set reference to 1 m, you should see the quadrotor taking-off and reaching the desired height. However, after few seconds the quadrotor will crash. Thus, it's time to implement the attitude controller.



### 2.2.2 Task 2b - Attitude control

As you could see in the previous exercise it is difficult to achieve stable flight even in a simulation where all motors have identical parameters. Therefore, it is necessary to implement attitude controller which will keep the vehicle stable.

We design attitude control in terms of Euler angles stabilization. While yaw angle is not critical for stability, the same does not hold for roll and pitch angles. Therefore, first design and implement roll and pitch controllers. The structure of each of the controller is the same as for the height controller (Fig. 2.2.1), but instead of vertical velocity here we have angular velocity (roll rate ( $\dot{\phi}$ ) and pitch rate( $\dot{\theta}$ )) and instead of z position, we have angles (roll and pitch).

For the linearization of this part of the system, use previously computed hover velocity and near hover state assumption. For trigonometric functions use the following approximations:

$$\cos \alpha = 1 \quad (22)$$

$$\sin \alpha = \alpha, \quad (23)$$

where  $\alpha = \phi$  or  $\theta$ .

For the roll rate controller, linearize Equ. 6 and 9 to get  $\frac{\Delta\dot{\phi}(s)}{\Delta\Omega_{\phi,ref}(s)}$ . In this part of the system,  $\Delta\Omega_{\phi,ref}(s)$  is the control value which we will add to the motors reference value (generated by z-cascade controller). Now we have to decide how to distribute  $\Delta\Omega_{\phi,ref}(s)$  at motors. From Equ. 6 we notice that in order to get a positive roll moment, one has to increase velocity of motors 2 and 3 and/or decrease velocity of motors 1 and 4. We will use a common quadrotor control policy - to get the positive roll moment we will increase motors 2 and 3 reference velocities by  $\Delta\Omega_{\phi,ref}(s)$  and decrease motors 1 and 4 reference velocities by the same amount. In other words, we will use the following constraint on motor velocity change:

$$\begin{aligned} \Delta\Omega_{1,\phi,ref} &= -\Delta\Omega_{\phi,ref} \\ \Delta\Omega_{2,\phi,ref} &= \Delta\Omega_{\phi,ref} \\ \Delta\Omega_{3,\phi,ref} &= \Delta\Omega_{\phi,ref} \\ \Delta\Omega_{4,\phi,ref} &= -\Delta\Omega_{\phi,ref} \end{aligned} \quad (24)$$

Now linearize Equ. 6 by neglecting gyroscopic term ( $((I_{yy} - I_{zz})\omega_y\omega_z \approx 0)$ ) and using Equ. 24.

Again you are free to choose control design method for determining PID gains. Report your results below:

Table 5: Roll rate controller design parameters

Transfer function $\frac{\Delta\dot{\phi}(s)}{\Delta\Omega_{\phi,ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Based on the open loop transfer function  $\frac{\Delta\dot{\phi}(s)}{\Delta\Omega_{\phi,ref}(s)}$  (remember that  $\frac{\Delta\dot{\phi}}{\Delta\phi} = \frac{1}{s}$ ), design the PID gains for roll controller. Report you results in Table 6.

Repeat the process for the pitch control. Again, we have to determine control policy and we will again use a common one: to get the positive pitch moment we shall increase motors 3 and

Table 6: Roll controller design parameters

Transfer function $\frac{\Delta\phi(s)}{\Delta\phi_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

4 reference velocity by  $\Delta\Omega_{\theta,ref}$  and decrease motors 1 and 2 reference velocities by the same amount. In other words, you should use the following constraint:

$$\begin{aligned}
 \Delta\Omega_{1,\theta,ref} &= -\Delta\Omega_{\theta,ref} \\
 \Delta\Omega_{2,\theta,ref} &= -\Delta\Omega_{\theta,ref} \\
 \Delta\Omega_{3,\theta,ref} &= \Delta\Omega_{\theta,ref} \\
 \Delta\Omega_{4,\theta,ref} &= \Delta\Omega_{\theta,ref}
 \end{aligned} \tag{25}$$

When linearizing Equ. 7 neglect the gyroscopic term and apply Equ. 25 to get  $\frac{\Delta\dot{\theta}(s)}{\Delta\Omega_{\theta,ref}(s)}$ . Report your results in Table 7 and 8.

Table 7: Pitch rate controller design parameters

Transfer function $\frac{\Delta\dot{\theta}(s)}{\Delta\Omega_{\theta,ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Table 8: Pitch controller design parameters

Transfer function $\frac{\Delta\theta(s)}{\Delta\theta_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

To implement your controller, open file `urs_solution/Task2b/attitude_ctl_tpt.py`. Again save it under the name `attitude_ctl.py` in the same folder and complete the code similarly as in Task2a (comment out staff required for yaw control).

To test your controller, run simulator, height controller and attitude controller

```
$ roslaunch urs_launch gazebo_ardrone.launch
$ roslaunch urs_launch task2b.launch
```

References for roll and pitch (in rad) publish on topic `/ardrone/euler_ref`. Feel free to play with PID gains to see how they effect control performance. Actually, as already mentioned, fine tune the parameters (including height controller). Don't forget that we are dealing with highly nonlinear model and that the parameters that you get are theoretically valid only for small changes in motors velocity and euler angles. However, fine tuning of PID controllers should

provide robust controller that should stabilize the vehicle in a reasonable range of roll/pitch change (e.g.  $\pm 45^\circ$ ) You will notice that for some cases it is desired to add I component to cancel static error, although it is not required according to your control design method. But, you have to be careful with I component as it can easily destabilize system.

The next task is to tune roll/pitch cascade controller to approximately get the following closed loop transfer function:

$$\frac{\Delta\phi(s)}{\Delta\phi_{ref}(s)} \approx \frac{\Delta\theta(s)}{\Delta\theta_{ref}(s)} \approx \frac{1}{0.43s + 1}. \quad (26)$$

This is the transfer function that we identified on a real AR.Drone quadrotor. Therefore, the goal is that you tune your controllers so that the closed loop angular dynamics in simulation become similar to the real vehicle dynamics. It should be useful to use ROS `rqt_reconfigure` tool here for online tuning.

List your tuned parameters below:

Table 9: Roll cascade tuned parameters

Roll PID		Roll rate PID	
$K_p$		$K_p$	
$K_i$		$K_i$	
$K_d$		$K_d$	

Table 10: Pitch cascade tuned parameters

Pitch PID		Pitch rate PID	
$K_p$		$K_p$	
$K_i$		$K_i$	
$K_d$		$K_d$	

Repeat the procedure for yaw angle (linearization of Equ. 8 and 9 + control design of a cascade controller). In this case, the control policy will be the following: to get a positive yaw moment, increase motors 2 and 4 reference velocities by  $\Delta\Omega_{\psi,ref}$  and decrease motors 1 and 3 reference velocities by the same amount. In other words:

$$\Delta\Omega_{1,\psi,ref} = -\Delta\Omega_{\psi,ref} \quad (27)$$

$$\Delta\Omega_{2,\psi,ref} = \Delta\Omega_{\psi,ref} \quad (28)$$

$$\Delta\Omega_{3,\psi,ref} = -\Delta\Omega_{\psi,ref} \quad (29)$$

$$\Delta\Omega_{4,\psi,ref} = \Delta\Omega_{\psi,ref} \quad (30)$$

Report computed parameters in Table 11 and Table 12. Implement yaw controller in file *attitude\_ctl.py*. Make sure that your controller handles any yaw reference from the range  $[-\pi, \pi]$  (the measured yaw is in the same range). Also, your controller should assure smooth transition from negative to positive references (e.g. try to set the reference to 3 *rad* and afterwards to -3 *rad* to see what happens).

Attach your functions *height\_ctl.py* and *attitude\_ctl.py* in Table 13. Finally, run the simulator, your controller and ROS node *test\_height\_attitude.pyc*. If controllers are working properly, you should see the quadrotor moving up and down, to the left and right and rotating in yaw axis. In the terminal where you run the test code you will be notified when the testing is finished. Plot the vehicle's height, roll, pitch and yaw responses that you get during the testing in separate

Table 11: Yaw rate controller design parameters







Transfer function $\frac{\Delta\psi(s)}{\Delta\Omega_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Table 12: Yaw controller design parameters

Transfer function $\frac{\Delta\psi(s)}{\Delta\dot{\psi}_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

figures. There are several options available for plotting (rqt\_gui, rosbag + MATLAB/Octave, numpy etc.). Use one of them and attach figures below.

Table 13: Task 2 final results

height_ctl.py	
attitude_ctl.py	
Height response	
Roll response	
Pitch response	
Yaw response	

### 2.3 Task 3 - Horizontal position control

In this task you will implement x and y cascade controllers. Apply the same procedure (linearization based control design) to get initial PID gains. For velocity control in x and y direction, linearize Equ. 10 and 11. Use the same assumption on small roll and pitch angles (Equ. 23). In addition, assume that yaw angle is zero ( $\psi = 0$ ). For the transfer functions  $\frac{\Delta\phi(s)}{\Delta\phi_{ref}(s)}$  and  $\frac{\Delta\theta(s)}{\Delta\theta_{ref}(s)}$  use Equ. 26 (we assume that you completed Task 2). Report your results in Tables 14 15, 16 and 17.

Implement your controllers by completing the file *urs\_solution/Task2b/horizontal\_ctl.tpt.py* (rename the file to *horizontal\_ctl.py* in the same folder). Your controller should be able to reach desired x,y position in global frame regardless of the quadrotor heading (yaw). Therefore, after you compute x and y PID cascade algorithm, you have to transform your control values (reference roll and pitch) from global to body frame using measured yaw angle.

Once finished with the implementation, run the simulator and all controllers

Table 14: Velocity ( $v_x$ ) controller design parameters

Transfer function $\frac{\Delta v_x(s)}{\Delta \theta_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Table 15: Position (x) controller design

Transfer function $\frac{\Delta x(s)}{\Delta v_{x,ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Table 16: Velocity ( $v_y$ ) controller design parameters

Transfer function $\frac{\Delta v_y(s)}{\Delta \phi_{ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

Table 17: Position (y) controller design




Transfer function $\frac{\Delta y(s)}{\Delta v_{y,ref}(s)}$	
$K_p$	
$K_i$	
$K_d$	
Control design method	

```
$ roslaunch urs_launch gazebo_ardrone.launch
```

```
$ roslaunch urs_launch task3.launch
```

Once you are satisfied with the responses, run ROS node *test\_horizontal.pyc* (make sure that all your controllers are turned on, i.e. run all nodes from previous tasks). Process acquired data and plot responses (similarly as in previous task). Attach your results in Table 18.




Table 18: Task 3 final results

horizontal_ctl.py	
X response	
Y response	

## 2.4 Task 4: Horizontal control of a real AR.Drone quadrotor

In this task you will implement and test horizontal control of a real AR.Drone quadrotor. Since you tuned angular controllers to get similar dynamics in simulation as in a real vehicle, you should be able to use herein the parameters from Task 3. Therefore, just complete the file `urs.solution/Task4/horizontal_ctl_real_tpt.py` with the same code as in Task 3 (please rename it to `horizontal_ctl_real.py`). You will have enough time on your practical session to fine tune the parameters and compare achieved performance with the simulation. For your report, attach the completed file and position responses below.

Table 19: Task 4 final results

horizontal_ctl_real.py	
X response	
Y response	

## References

- [1] Catkin tools cheat sheet, January 2018.
- [2] Robert C Leishman, John C. MacDonald, Randal W Beard, and Timothy W. McLain. Quadrotors and Accelerometers: State Estimation with an Improved Dynamic Model. *IEEE Control Systems*, 34(1):28–41, 2014.
- [3] Philippe Martin, Erwan Salaün, and E Salaun. The true role of accelerometer feedback in quadrotor control. *Robotics and Automation (ICRA), 2010 . . .*, pages 1623–1629, 2010.
- [4] D. Miklic, M. Orsag, and I. Markovic. Programming for the robot operating system: Lectures, January 2016.
- [5] W. Woodall. Ros kinetic installation instructions, January 2018.