POLYTECH
NANCY







Control Engineering
Labs

Hugues Garnier
Floriane Collin

September 2026

Contents

Introduction to the Control Engineering Labs **1**

1 PID-based line tracking control for the 3pi+ mobile robot **4**
1.1 A few short videos to start with . . . 4
1.2 Pre-lab questions . . . 5
1.3 The 3pi+ mobile robot from Pololu . . . 5
1.3.1 Hardware required . . . 5
1.3.2 Operating system and software required . . . 7
1.3.3 Identification of the physical system components of the 3pi+ mobile robot . . . 7
1.3.4 Test of a basic control in open loop . . . 10
1.3.5 Characteristics of the line following from a control system perspective . . . 12
1.3.6 Control performance requirements . . . 14
1.4 Control strategies for the robot line tracker on the basic test track . . . 14
1.4.1 Download of the files required for closed-loop control . . . 15
1.4.2 Test of a P controller in closed loop . . . 15
1.4.3 Test of a PD controller in closed loop . . . 16
1.4.4 Performance summary of the different controllers on the basic test track . . . 18
1.5 Control strategies for the robot line tracker on the racing competition track . . . 18
1.6 Summary and reflections . . . 18

2 Model-based PID controller design and tuning for rotary speed control of the QUBE-Servo 2 **19**
2.1 Methodology: Model-Based Design organized as a V-cycle . . . 19
2.2 The QUBE-Servo 2 from Quanser . . . 20
2.2.1 The QUBE servo-motor . . . 20
2.2.2 Identification of the physical system components . . . 21
2.2.3 QUARC software . . . 22
2.3 Control objectives and performance requirements . . . 24
2.4 Getting started: lab files and hardware setup . . . 24
2.5 Transfer function modelling of the rotary speed dynamics from a step test . . . 25
2.5.1 Servo motor model . . . 25
2.5.2 Recording of the step response . . . 25
2.5.3 Identification of a linear first-order model . . . 26
2.6 Rotary speed control using a simple Proportional (P) feedback controller . . . 26
2.6.1 Reminder of the theoretical results and numerical application . . . 27
2.6.2 Simulation of the simple P feedback controller in Simulink . . . 27
2.6.3 Experimental test of the simple P feedback controller on the physical QUBE-Servo 2 . . . 28
2.7 Rotary speed control using a standard PI feedback controller . . . 28
2.7.1 Reminder of the theoretical results and numerical application . . . 28
2.7.2 Simulation of the standard PI feedback controller in Simulink . . . 29
2.7.3 Experimental test of the standard PI feedback controller on the physical QUBE-Servo 2 . . . 29
2.8 Rotary speed control using an I-P feedback controller . . . 30

2.8.1 Reminder of the theoretical results and numerical application 30
2.8.2 Simulation of the I-P feedback controller in Simulink 30
2.8.3 Experimental test of the I-P feedback controller on the physical QUBE-Servo 2 31
2.9 Rejection of a load disturbance with the I-P controller 31
2.9.1 Theoretical analysis 32
2.9.2 Simulation of a load disturbance rejection in Simulink 32
2.9.3 Experimental test of a load disturbance rejection on the QUBE-Servo 2 32
2.10 Troubleshooting 34

3 Model-based PID control design for the mini-drone TELLO 35
3.1 Methodology: Model-Based Design organized as a V-cycle 35
3.2 The TELLO mini-drone 36
3.2.1 Main components 37
3.3 Understanding the basics of quadcopter control 37
3.3.1 Videos to understand the basics of drone control design 40
3.3.2 Manual control 40
3.3.3 Full closed-loop control 40
3.4 Altitude control of the TELLO mini-drone 41
3.4.1 Altitude control performance requirements 41
3.4.2 Open-loop control of the TELLO mini-drone by using its App 42
3.4.3 Download of the files required for the lab 42
3.4.4 Vertical dynamic model identification 43
3.4.5 P control for altitude tracking of a square wave signal 45
3.4.6 PD control for altitude tracking of a square wave signal 48
3.5 Troubleshooting 51
3.5.1 Connection issues 51
3.5.2 Calibration issues 51

Introduction to the Control Engineering Labs

About this manual

The purpose of this manual is to provide you with the information necessary for conducting the laboratory experiments of the Control Engineering course. You should by no means consider this manual as the sole source of information for the design and implementation process. You should rely on lectures slides, problem solving sessions, as well as your own knowledge of control engineering theory, to develop and implement your control strategies.

Objectives of the Control Engineering labs

The objectives of the labs can be summarized as follows:

- to learn how to investigate the properties of physical systems both through simulation and experimentation;
- to learn how to design and analyze feedback control systems using classical control techniques based on transfer function model and block-diagrams;
- to learn how to perform identification experiments on different physical systems;
- to learn how to simulate classical on-off and PID control in Matlab/Simulink;
- to learn how to implement classical feedback control strategies on different experimental systems;
- to develop practical skills in an experimental environment.

Pre-lab questions

The pre-lab questions of the last two labs are related to the problem solving sessions No. 4 and No. 5, respectively. Solutions to all questions should be available upon request from the lab assistant. These solutions will most likely require to be adapted to your experimental conditions.
When the lab starts, it is assumed that you have a full understanding of these solutions, and have a clear idea of the tasks that you will have to perform during the lab.

Doing your own work

While you are encouraged to discuss theory and methods with other classmates, when it comes time to analyze, design, and implement your system, it must be done on your own. As a future engineer, you must always fully understand the problem you are trying to solve. Also, remember that with experimentations, there is often no "right" answer. You should answer the question based on your observations for your simulation and experimental results from your particular system components, which may be quite different from others.

Grading

When working in an experimental environment, results are always important, but so is your ability to communicate the information with others. Any work you will submit should be neat, well-organized and easy to understand. Your report is a demonstration of your ability to understand the course you are studying as well as a reflection of your organizational skills.

Steps in the design of a control system

The various stages that lead to the design of a feedback control are detailed in Figure 1.

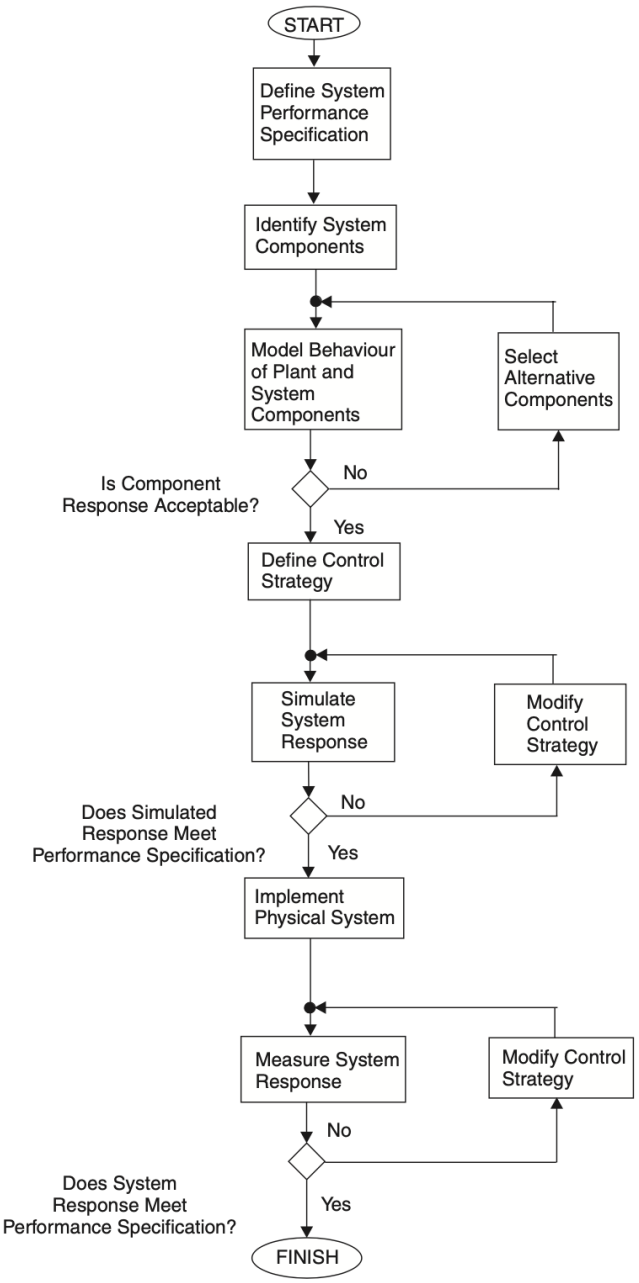


Figure 1: Steps in the design of a control system (From *S. Burns, Advanced Control Engineering, Butterworth-Heinemann, 2001)*

The methodology will be used throughout the different labs and you should always come back to it if your design is not successful. The determination of a model and the simulation of the behaviour of the closed-loop system are crucial steps for the design of a feedback control system. You will use the Matlab/Simulink environment to design, develop, and validate a variety of feedback control strategies before their implementation on the physical systems.

Use of Matlab/Simulink

MATLAB for MATrix LABoratory is a powerful software that can deal with complex problems in various scientific fields in an interactive environment. MATLAB is supplemented by various special application toolboxes, which contain additional functions and programs. One such toolbox is the Control System Toolbox which is also available in student editions.
Simulink is a very useful Graphical User Interface (GUI) tool of MATLAB for modeling closed-loop block-diagram control systems, and simulating their time response to specified inputs.
If you are not familiar with Simulink, take some time before the lab sessions to follow some tutorial introduction available such as Simulink-on-ramp from the Mathworks website.

Lab 1

PID-based line tracking control for the 3pi+ mobile robot

You are not asked to submit a report for this lab which will not therefore be marked. You are, however, encouraged to write a personal report that will be useful for your preparation to the final exam which will include some questions related to the labs.
This first lab aims to study and design different control strategies for a line following robot so that it is able to track a black line that is drawn on the surface.

1.1 A few short videos to start with

Line tracking has become the most convenient and reliable technique by which autonomous mobile robots can navigate in a controlled, usually indoor, environment. The path of the robot is demarcated with a distinguishable line or track, which the robot uses to navigate.

In the industry, vehicles are often required to carry products from one manufacturing plant to another which are usually in different buildings or separate blocks. Conventionally, carts or trucks were used with human drivers. Unreliability and inefficiency in this part of the assembly line formed the weakest link. Solutions based on automated guided vehicle following a line, instead of laying railway tracks which can be more costly, are nowadays available as shown in Figure 1.1.



Figure 1.1: Example of automated guided vehicle in the industry

You can also see the automated guided vehicle in action by watching the short video available at: `www.youtube.com/watch?v=WIlS3vNSuQ4`
Automated guided vehicles following a magnetic line have become of common use at Tesla Mega factories as explained by Elon Musk: `youtu.be/mr9kK0_7x08&t=6m43s?`

Many line following competitions have been organized by clubs of robot building enthusiasts over the last two decades. The goal of these racing competitions is usually to build an autonomous robot that does a certain number of laps of a test track the fastest. Watch, for example, the video that shows six 3pi+ robots running simultaneously on the same line-race course. Each robot was programmed

independently, but as we can see, the results were remarkably consistent. The last one on the line wins!
`www.youtube.com/watch?v=fl0CJhPiEfYt=76s`
We will try to reproduce this contest in your group by using the same type of robots.

1.2 Pre-lab questions

There are no pre-lab questions for this lab.

1.3 The 3pi+ mobile robot from Pololu

The 3pi+ 32U4 robot is a high-performance, user-programmable robot that measures just 9.7 cm in diameter. The brain of the robot is an integrated, USB-enabled, Arduino-compatible ATmega32U4 microcontroller from Atmel, clocked by a precision 16 MHz crystal oscillator. The 3pi+ robot comes preloaded with an Arduino-compatible USB bootloader which allows it to be easily programmed using the Arduino IDE.

1.3.1 Hardware required



Figure 1.2: Front view of the 3pi+ robot from Pololu

The required hardware for this lab includes:

- a 3pi+ Robot from Pololu as shown in Figure 1.2;
- a USB A to Micro-B cable to connect the robot to your computer for programming and debugging. The USB connection can be used to transmit and receive data from the computer and program the board over USB;
- four rechargeable AAA NiMH batteries;
- a battery charger (in case your batteries are discharged before the end of the lab);
- a basic track for initial test and a racing competition track which has the form of the F1 Estoril course in Portugal shown in Figure 1.3.

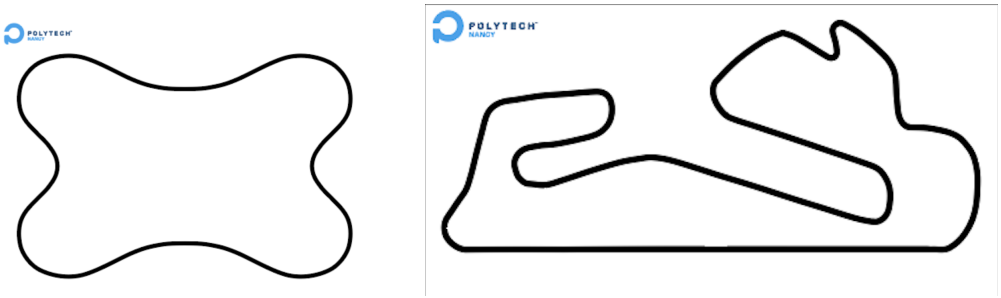


Figure 1.3: Tracks used for initial basic tests and for the racing competition

1.3.1.1 Version of the 3pi+ 32U4 robot

Three different versions of the assembled 3pi+ 32U4 robots exist. They are equipped with different motors which will impact the maximum speed of the robot. They can be identified with a sticker on the underside of the main board, visible inside the battery compartment of the 3pi+ without batteries installed. The color of the sticker indicates the gear ratio of the robot's motors:

- Green: 50:1 HP
- Blue: 75:1 HP
- Red: 100:1 HP

From the user guide available at: `www.pololu.com/docs/0J83`

- Determine the version of your 3pi+ 32U4 robot: standard, turtle or hyper edition;
- Find in the user guide the maximum speed in m/s for your 3pi+ 32U4 robot.

1.3.1.2 On/off and user pushbuttons

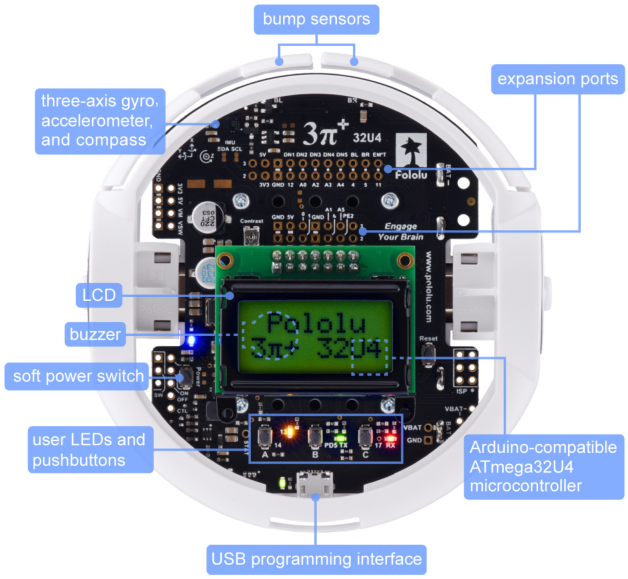


Figure 1.4: Top view of the 3pi+ robot showing its main components

The 3pi+ 32U4 control board has five pushbuttons: a power button on the left, a reset button on the right, and three user pushbuttons, labeled A, B, and C, located at the rear close to the USB programming interface as shown in Figure 1.4.

Pushbutton B will be mainly used for calibrating and running the 3pi+ robot. At the bottom left side of the LCD display, there is the soft power button to switch on or switch off the 3pi+ robot. On the right side of the LCD display, there is the reset button that can be used to reset the board (you should not need to use it).

1.3.2 Operating system and software required

1.3.2.1 Operating System required

The 3pi+ 32U4 robot can be programmed from a computer using any operating system that supports the Arduino environment. This includes Microsoft Windows 11, 10, 8.1, 8, 7, Vista, XP (with Service Pack 3), Linux, and Mac OS X.

1.3.2.2 Software required

The Arduino IDE will be used to implement and test different open-loop and closed-loop control algorithms. To get started with your 3pi+ robot, follow the instructions given below. They are more detailed in Section **6. Programming the 3pi+ 32U4** of the 3pi+ robot user guide which can be downloaded from `www.pololu.com/docs/0J83`.

1. If necessary, download and install on your PC the Arduino compiler from `www.arduino.cc/en/Main/Software`.

2. To help interface with all the on-board hardware on the 3pi+ 32U4, Pololu has provided the **Pololu3piPlus32U4** library.

   If it is not already installed, install the **"Pololu3piPlus32U4"** library by following carefully all the instructions given in `www.pololu.com/docs/0J83/6.2`.

   Depending on your Arduino version, you can also try the following: from Arduino, go to the "Sketch" menu, select "Include Library", then "Manage Libraries...", then search for **"Pololu3piPlus32U4"** and install the library.

1.3.3 Identification of the physical system components of the 3pi+ mobile robot

Prior to operating any experimental system, it is imperative to familiarize yourself with the various components of the system. Part of the process involves the identification of the individual hardware components, including its sensors and actuators. Find out from the description given below more detail about the individual hardware components of the 3pi+ robot.

1.3.3.1 The line sensors

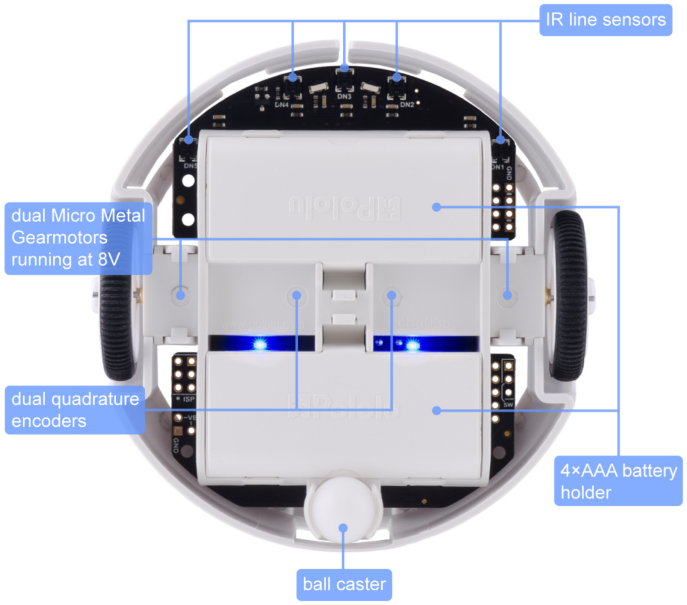


Figure 1.5: Underneath view of the 3pi+ robot. The IR line sensors are placed at the front.

The 3pi+ line sensor array is a separate board that attaches to the main underneath board, placed in front of the robot as shown in Figure 1.5. The board features five line sensors connected to the microcontroller. The five line sensors face downward and can help the 3pi+ robot distinguish a non-reflective (black) line on a reflective (white) surface. Each reflectance sensor consists of a down-facing analog infrared (IR) emitter LED (Light-Emitting Diode) paired with a phototransistor that can detect reflected IR light from the LED. Three of the five sensors are placed in the middle of the board at gaps of about 1 cm and the last two are situated at the far end of both sides such that the total sensor coverage is about 5 cm.

The line IR sensors are just an indication of surface reflectivity. The sensor value is lower when over a white surface and higher when over a dark surface. The actual values are heavily dependent on the external lighting conditions even though the sensor array is shielded. Furthermore, there is appreciable difference between the readings of various sensors due to various factors like placement, manufacturing differences, etc.
To get usable values from all the sensors, the sensor values are normalized for environmental conditions and sensor differences and the normalization values are stored in Data Flash of the microcontroller. This enables the normalization data to be used multiple times. To normalize the sensor values, the robot is placed on the track and rotated such that all the sensors pass over the black and white surfaces. The maximum and minimum readings for all the sensors are noted down and stored in non-volatile memory (Data Flash) of the microcontroller.
The `lineSensors.readLine(lineSensorValues)` function returns a value between 0 and (number_of_sensors - 1)×1000. As the 3pi+ 32U4 robot has an array of 5 sensors, the reading will be 0-4000. 0 represents the left most sensor and each increment of 1000 after that represents another sensor. 0 = first sensor, 1000 = second sensor and there are ranges in between as well. 500 means the line is between the first and second sensors, 1200 means it's between the second and third sensors but closer to the second, etc. Note that if the reading is 2000, it means that the robot is centered on the line.

This output of the IR line sensor array is fed to the microcontroller, which estimates the position of the line under the robot and calculates the error term between this estimated position and the line position setpoint (2000). The most convenient and reliable way to estimate the line position is to compute the

weighted average (centroid) of the sensor abscissas $p_i = 1000\,i$, each sensor being weighted by its normalized reading. The line position $y(k)$ and the error $\varepsilon(k)$ at time-instant k are given as follows:

$$y(k) = 1000 \times \frac{\sum_{i=0}^{4} i \times L_i(k)}{\sum_{i=0}^{4} L_i(k)}, \tag{1.1}$$

$$\varepsilon(k) = y(k) - 2000 \tag{1.2}$$

In the equations above, $L_i(k)$ refers to the normalized output of the i-th line sensor in the array. Here, 1000 is the spacing between two adjacent sensors expressed in position units: it converts the sensor index i into its abscissa $p_i \in \{0, 1000, 2000, 3000, 4000\}$, so that $y(k) \in [0\,;\,4000]$. The normalization itself is performed by the denominator, which makes the estimated position independent of the overall intensity of the readings. Note that the reading $L_0(k)$ of the leftmost sensor is taken into account even though $i \times L_0(k) = 0$ in the numerator: it appears in the denominator and pulls the centroid towards $p_0 = 0$. The subtraction by 2000, which corresponds to the line position setpoint, ensures that zero error occurs when the robot is centered on the line.

The sensors are numbered from the left to the right. An error value of zero refers to the robot being exactly on the center of the line. A positive error means that the line lies under the right part of the array, i.e., the robot has deviated to the left, and a negative error means that the robot has deviated to the right. The error can have a maximum value of ± 2000, which corresponds to the maximum detectable deviation. Note that, in accordance with the convention adopted in the Pololu library and in the program `Lab1.ino`, the error is defined throughout this lab as $\varepsilon(k) = y(k) - Y_r$, i.e., the measured line position minus the setpoint, rather than the usual $Y_r - y(k)$; the signs at the summing junction of the block diagram should be read accordingly.

The main advantage of this approach is that the five sensor readings are replaced with a **single error term**, which can be fed to the control algorithm to compute the motor speeds such that the error term becomes zero. Further, since the estimated position is a centroid, the error value is largely independent of the line width, so the robot can tackle lines of different thicknesses without needing any change in code.

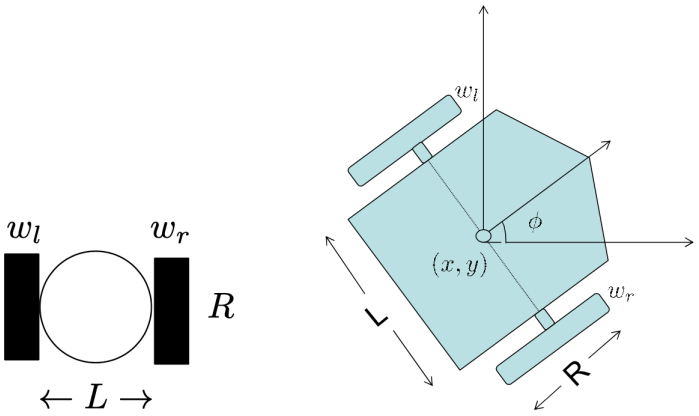
Note: since the outer sensors are farther apart than the central ones, the relationship between $\varepsilon(k)$ and the physical lateral deviation is only approximately linear. Moreover, when the line is lost, `lineSensors.readLine()` returns 0 or 4000 depending on the side where the line was last seen, so that $\varepsilon(k)$ saturates at ± 2000.

1.3.3.2 The actuators

The robot is driven by two micro metal gearmotors with extended motor shafts, coupled to wheels. It is battery powered, with the motors supplied with regulated power supply to ensure consistent motor power at all operating scenarios. The robot can reach a maximum speed of about 1.5 m/s.

A simplified diagram of the 3pi+ mobile robot is presented in Figure 1.6 where

- L is the distance between two wheels,
- R is the radius of each wheel,
- ω_l is the rate (or rotary speed) at which the left wheel is turning,
- ω_r is the rate (or rotary speed) at which the right wheel is turning.



(a) Two Wheeled Robot (b) Simplified Model

Figure 1.6: Simplified diagram of the two wheel mobile robot.

The 3pi+ robot is a **differential drive** robot. It contains two wheels each of which is equipped with a separate DC motor that controls its rotation. For this system, the two inputs are going to be ω_l and ω_r, i.e., rotary speed at which left and right wheel are turning. This robot is called **differential drive** because the steering of this system is based on difference in the turning rates/rotary speeds of the two wheels.

To move the robot forward, both motors are rotated at a given nominal speed in the forward direction. To make the robot turn to the left or to the right, the speed of one motor is reduced while the speed of the second motor is increased by the same value. The amount of turn increases as the speed difference increases. Maximum amount of turn is achieved when one motor is turned in a backward direction at maximum speed, the other in the forward direction at maximum speed. This results in maximum speed difference and the robot just spins in place.

1.3.4 Test of a basic control in open loop

The system characteristics can be better understood by considering an open-loop test on the line tracking system:

1. Assume that the robot is placed exactly on the centre of a straight line and exactly in the direction of the line. If both motors run at equal speed, the robot moves forward with zero error. This is the only special condition where zero control effort is needed.

2. If there is any discrepancy, line not being exactly straight, robot not being perfectly aligned or the existence of speed difference between the motors, control effort is needed (speed of one of the motors must be reduced while the speed of the second motor must be increased).

3. Now, assume that the robot is away from the centre of the line by some distance. In order to reduce the error, a control effort must be given as illustrated in Figure 1.7. However, the sending of a constant control effort will cause the robot to overshoot the line.

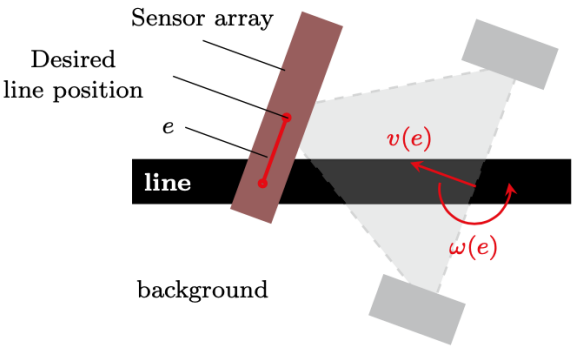


Figure 1.7: Open-loop test to illustrate the required control effort to steer the robot over the line.

We will first test how to control in open loop the 3pi+ robot by setting manually (in open-loop) the speed command of both right and left wheels. Follow the instructions below:

1. From the Arduino IDE, open the "File" menu, select "New", then enter the code below:

```
#include <Pololu3piPlus32U4.h>
#include <PololuMenu.h>

using namespace Pololu3piPlus32U4;

Motors motors;

void setup()
{
delay(5000); // wait for 5 seconds before starting the basic control
// in open loop
}
void loop()
{ // Set a nominal 50 speed command to both wheels.
// The sign determines if the robot goes forward or reverse.
int16_t rightSpeed = 50;
int16_t leftSpeed = 50;
motors.setSpeeds(leftSpeed,rightSpeed);
}
```

2. Connect the 3pi+ robot to your PC with the USB cable, without switching it on: there is no need to press the power button, since the board is powered through the USB connection during programming.

3. In the "Tools" menu, select "Board", then select "Arduino AVR Board" menu, then "Arduino Leonardo" entry.

4. In the "Tools" menu, select "Port" menu, select the port to which the 3pi+ robot is connected.

5. Press the "Upload" button to compile the sketch and upload it to the device. If everything goes correctly, you will see the message "Done uploading" appear near the bottom of the window.

6. Disconnect the 3pi+ robot and switch it on by pushing the power button of the robot, on the left of the LCD display.

7. Place the 3pi+ robot over the straight black line of the Estoril racing track. The robot should start at a fairly low speed. You might notice it is not driving in a straight line. That is the unfortunate consequence to real life: the same command to each motor might actually be driving

them at slightly different speeds or may be one wheel is making better contact with the ground. Anyway, you can see that the robot is doing pretty much exactly what you thought it would do.

8. Catch the robot and switch it off.

9. Modify the left motor speed to the nominal value 50 + 10 = 60 and the right speed to the nominal value 50 - 10 = 40.

10. Compile the program and upload it to the 3pi+ robot. Repeat the open loop control test and observe if the robot is now turning left or right? Does it make sense?

11. Set the left motor speed to the nominal value of 60 and the right speed to the nominal value of -60. Compile the program and upload it to the 3pi+ robot. The robot just spins in place.

From this simple open loop test, you should have understood how you can control the robot direction by adjusting the speed difference between the 2 wheels from a given nominal speed command. The goal will be to design different control strategies to set automatically the speed difference command from a given nominal speed command.

1.3.5 Characteristics of the line following from a control system perspective

To achieve good performance, there is a need to better examine the characteristics of the line following from a control system perspective. This should help for the tuning the different terms of the PID controller.

When the robot is placed on a straight track, it was observed that the robot was able to stay centered over the line. This indicates that the steady-state error is zero for a straight line. When the robot is placed on a curved track, it was seen that the robot overshoots the line centre by a greater amount in the direction opposite to the direction of curvature of the track. Further, it was observed that, as the curvature increases, the difference in overshoots increases. This indicates that a steady state error exists when the track is curved.
For the closed-loop system, the setpoint is always constant and set to 2000 so that the robot stays on the centre of the line. However the track does not remain straight and so the line curvature should be considered as an external disturbance for the control system. Now, it can be concluded that, when the system has zero or negligible disturbance (analogous to a step input for a simple unity feedback system), the steady-state error (position error) is zero. When a significant disturbance exists (analogous to a ramp or parabolic input for a simple unity feedback system), the steady-state error is significant. From the above discussion, it can be concluded that there is at least one integrator in the feedforward path of the system.
From the above considerations, it can be understood that the loop is integrating, i.e. control effort is being integrated by the system. The open loop response of the system thus resembles that of a servo position control system. Thus, the control techniques used for servo control may be applicable for the line tracking robot.

The closed-loop block diagram of the wheeled mobile robot based on **differential drive** mechanism is shown in Figure 1.8. The thick solid line from the sensor block represents the multiple feedback signals from the sensors.

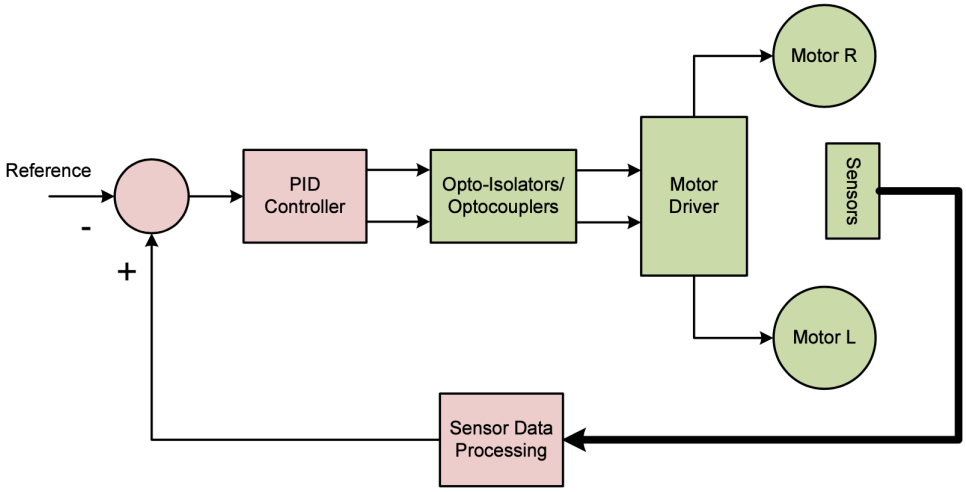


Figure 1.8: The system block diagram based on differential drive mechanism.

To handle the fact that the robot requires two inputs (the speed commands to the left and right wheels) while the PID control computes a single steering correction, we introduce a process that maps the PID output $u(t)$ into speed commands for both wheels. This is typically done using a differential drive, which adjusts the wheel velocities based on the PID correction.

The PID controller computes a single command value, $u(t)$, which represents the required adjustment to the robot's trajectory.
The left (v_L) and right (v_R) wheel commands (voltages) are determined based on:

- a base nominal speed, v_{base}: the desired forward velocity;
- the PID correction $u(t)$, represents the adjustment to the robot's angular velocity.

The wheel voltage commands are calculated as:
Left wheel voltage:

$$v_L(t) = v_{base} + u(t)$$

Right wheel voltage:

$$v_R(t) = v_{base} - u(t)$$

The nominal base speed ensures forward motion, while the correction factor $u(t)$ increases one wheel's speed while decreasing the other, creating the desired turn and ensuring steering toward the line.

From the explanations given above, complete the block-diagram of the closed-loop control for the 3pi+ robot displayed in Figure 1.9.

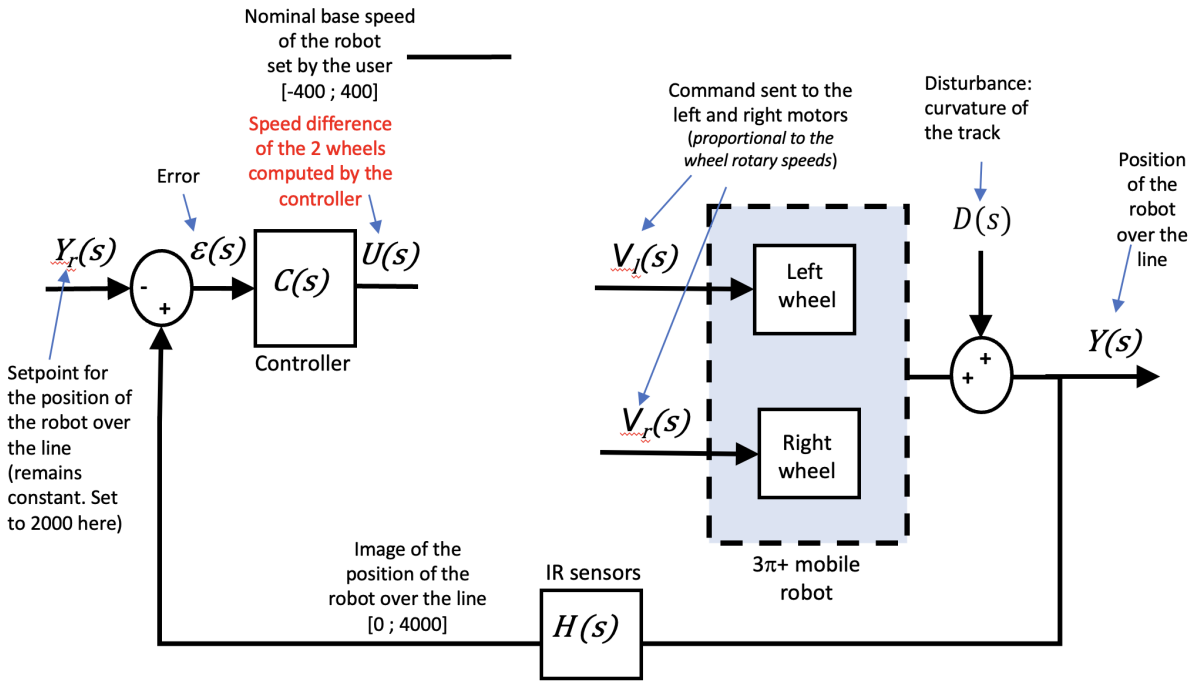


Figure 1.9: Block diagram to be completed of the line-tracking control for the 3pi+ robot (the error is defined as the measured line position minus the setpoint set to 2000 here).

1.3.6 Control performance requirements

The performance requirements for the line following control are the following:

The 3pi+ robot is expected to track the line faithfully. It should not leave the line at any instant and the movement of the robot should have minimum overshoot. All its movements should be smooth meaning that the robot must not have zigzagging and jerky movements.
In terms of performance, a first objective is to make the 3pi+ mobile robot move forward at the fastest speed (nominal base speed greater than 250) so that it can complete **one lap of the basic track** in the shortest time possible. The second objective, if time allows, is to make the 3pi+ mobile robot complete **one lap of the racing competition track** at a moderate nominal base speed of 200.

1.4 Control strategies for the robot line tracker on the basic test track

A simplified version block-diagram of the feedback control for the robot line tracker is shown in Figure 1.10.

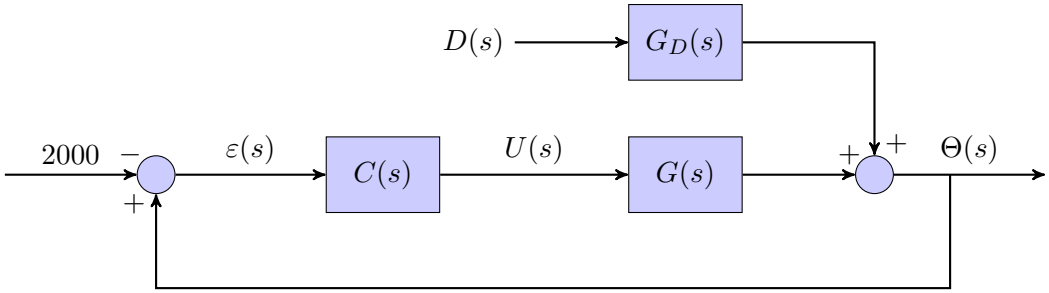


Figure 1.10: Simplified block-diagram of the feedback control (the error is defined as the measured line position minus the setpoint, i.e., $\varepsilon = \Theta - 2000$).

1.4.1 Download of the files required for closed-loop control

1. Download the file `Lab1.ino` from the course website, then save it in a Lab1 folder.
2. Do not open the `Lab1.ino` file directly in the Arduino IDE: for some unclear reasons, its content will appear empty.
3. Instead, open `Lab1.ino` with a plain text editor such as *Notepad* (Bloc-notes on a French version of Windows), copy all the lines of code, and paste them into a new sketch in the Arduino IDE.

1.4.2 Test of a P controller in closed loop

To make the 3pi+ robot follow the black line of the basic test track, we will now test a closed loop control by using a simple Proportional controller of the following form:

$$C(s) = \frac{U(s)}{\varepsilon(s)} = k_p$$

where k_p refers to the proportional gain constant.

The P controller equation in the time-domain is given as

$$u(t) = k_p \times \varepsilon(t)$$

Find in the Lab1.ino program, the line which implements the P controller equation.

Note that for this simple P control strategy, you will need to select a nominal speed ($\in [100; 300]$) at which the robot will travel. If the selected nominal speed is too high and/or the proportional gain is badly tuned, the robot will not be able to stay on the track.

Follow the instructions below:

1. From the Arduino IDE, verify that the variable **nominalSpeed** is set to 100 in the file **Lab1.ino**;
2. Connect the 3pi+ robot to your PC with the USB cable, without pressing its power button.
3. In the "Tools" menu, select "Board", then "Arduino AVR Board", then "Arduino Leonardo" entry.
4. In the "Tools" menu, select "Port" menu, select the port for the device.
5. Press the "Upload" button to compile the sketch and upload it to the device. If everything goes correctly, you will see the message "Done uploading" displayed at the right bottom of the window.
6. Disconnect the 3pi+ robot and switch it on pushing the power button on the left side of the LCD display.
7. Place the 3pi+ robot over the black line of the basic test track and press the B-button. The robot will make two turns to start the automatic sensor calibration step by rotating in place to sweep the sensors over the line.
8. Prepare your smartphone to measure the time it takes for the 3pi+ robot to complete one lap. When you are ready to start the time, press again on the B-button and let the robot follow the line. Stop the timer when the robot has completed one full lap, if it can make it! Record the time that will constitute the reference time to be improved.
   You should observe that the simple P controller works reasonably well for this relatively slow nominal speed.

9. Place now the 3pi+ robot over the black line of the racing competition track and test if the robot is able to complete one full lap.
   If this is not the case, try to reduce the nominal speed and test if the robot is able to achieve one full lap without leaving the line in the sharp curved area. To achieve better performance, there is a need to better understand how the control of the two motor speeds work.

10. From the example program and comments given inside, determine the main command to be controlled and the two variables used to steer the 3pi+ robot.

11. Give the control equation for both left and right motor speeds.

12. Determine the setpoint value for the line position.

13. Determine the role of the variable **nominalSpeed**.

14. Build a block-diagram of the simple P controller for the robot line follower, showing a block for the actuator, the robot, the sensor and the controller.

15. Modify the proportional gain of the controller so that the robot can complete one lap of the basic test track in the shortest time possible. Once the robot is following the line with good accuracy, increase the **nominalSpeed** variable to 200 and 300 (50%, and 75% of its maximum value respectively) and see if it is still able to follow the line (this might not be the case for 300). Note that the robot nominal speed affects the P controller tuning and will require its retuning as the **nominalSpeed** changes.

Gather the performance of the different P controllers by completing the first three lines of Table 1.1.

nominalSpeed	Controller	K_p gain	K_d gain	Completion time for one lap (sec)
100 (25%)	P controller		/	
200 (50%)	P controller		/	
300 (75%)	P controller		/	
200 (50%)	PD controller			
300 (75%)	PD controller			

Table 1.1: Performance of the various P and PD controllers on the basic track

1.4.3 Test of a PD controller in closed loop

One of the most popular control techniques for servo position control takes the form of a PD controller of the following form:

$$C(s) = \frac{U(s)}{\varepsilon(s)} = k_p + k_d s$$

where k_p and k_d refer to proportional and derivative gain constants respectively.

1.4.3.1 Finding best values of the PD controller

The PD controller equation in the time-domain is given as

$$u(t) = k_p \times \varepsilon(t) + k_d \times \frac{d\varepsilon(t)}{dt}$$

For digital implementation, the time-derivative of the error term can be approximated by using the backward Euler method.

$$u(k) = K_p \times \varepsilon(k) + K_d \times \big(\varepsilon(k) - \varepsilon(k-1)\big) \tag{1.3}$$

where $K_p = k_p$ and $K_d = \dfrac{k_d}{T_s}$; T_s being the sampling period. $u(k)$ denotes the command value at time-instant $t = kT_s$.
It can be noted that the sampling time, T_s, is simply eliminated from the control equation because it is embedded in the new derivative gain K_d.
This simplification assumes however a constant sampling period, whereas T_s here corresponds to the execution time of one iteration of `loop()`; any code added to the loop (e.g., display or serial communication instructions) changes T_s and may therefore require K_d to be retuned.
Finding the best values of the PD controller, i.e., K_p and K_d values, is now the most important challenge for you. There is unfortunately no simple way to get a model of the robot from an open-loop test with a nominal speed of 200, which could serve as the basis to tune the PD controller gains. The latter will therefore be tuned in this lab by a trial and error method only. These gain values are expected to be different for every group of students and for the nominalSpeed variable setting, which determines the speed at which the robot is running.

Using a "trial and error" methodology to set the gains of a PD controller has some advantages (no need to determine a model, ...). The bad side of it is that you must re-compile the program each time that you change the gains.

You might want to try this empirical method for the tuning of the PD gain controller:

- Modify the initial program to implement a PD control given in (1.3) instead of the simple P control. In the "File" menu, select "Examples" entry, then "Pololu3piPlus32U4" entry and open the LineFollowerPID.ino file available which implements a PD controller. Find and understand the line which implements the PD controller equation. Implement all the needed code in your initial program.

- Set the nominalSpeed variable to 200 (50% of its maximum value).

- Set the K_d gain to 0 and tune the K_p term alone first. If the robot cannot navigate a turn or reacts too slowly, increase its value. If the robot seems to react fast leading to a zigzagging behavior, decrease the value.

- Then set your K_d gain value to 10 to 100 times the K_p value and check if the robot can navigate in a better way than with the simple P controller. Increase the derivative gain K_d to decrease the overshoot (of the line), decrease it if the robot becomes too jerky or is not able to remain on the line.

- Once the PD controller tuning makes the 3pi+ robot follow the line with good accuracy, record the time that will constitute the best time for the PD controller at 50% of the maximum value of the robot speed.

- Once the robot is following the line with good accuracy, increase the **nominalSpeed** variable to 300 (75% of its maximum value) and see if it still is able to follow the line. Note that the robot speed affects the PD controller and will require retuning as the nominalSpeed changes.

- It is worth noting that the tuning of the PD controller gain might not be possible for the maximum speed setting without additional modification of the control algorithm so that the speed of the robot is dynamically reduced/increased depending on curvature of the track, as we do when we drive a car!

1.4.4 Performance summary of the different controllers on the basic test track

Gather the performance of the different controllers by completing Table 1.1.

1.5 Control strategies for the robot line tracker on the racing competition track

From your initial experience obtained from your test with the basic track, tune the PD controller the best you can so that your robot can traverse the Estoril racing track as fast as possible. Gather the performance and tuning value of the best PD controller in Table 1.2.
Note that the Estoril track is much more challenging than the basic track since it includes a difficult curved area which requires a low speed to keep the robot on the track. The nominal base speed of the robot should therefore be set to small or moderate value for the robot to be able to handle this curve and complete a full lap.

nominalSpeed	Controller	K_p gain	K_d gain	Completion time for one lap (sec)
100 (25%)	PD controller			
200 (50%)	PD controller			

Table 1.2: Performance of PD controller on the racing track

1.6 Summary and reflections

Summarize and reflect on what you have learned in this lab.

As you have experienced it during this lab, tuning a PID controller based on a trial and error methodology can be tedious and can lead to moderate performance. An alternative and more efficient approach would consist in determining a mathematical model which serves as the basis of the PID controller tuning. We will exploit this model-based PID control tuning approach in the next two labs.

Look over the Internet to find out more advanced control strategies to get better line tracking performance for the 3pi+ robot.

Lab 2

Model-based PID controller design and tuning for rotary speed control of the QUBE-Servo 2

Pre-lab questions. Review of Problems No. 4 and their solutions. Before coming to the lab, make sure you have a full understanding of these solutions.
Note that they might need to be slightly adapted to the experimental conditions.

In the first lab, you tuned the gains of the PD controller of the 3pi+ robot by trial and error: each new set of gains had to be tested on the robot, the results were difficult to interpret, and there was no guarantee of finding good settings in a reasonable time. This approach quickly reaches its limits as soon as the system becomes more complex, more expensive or more fragile. In industry, a controller is therefore not tuned by trial and error directly on the real system: it is designed from a model of the system, tested in simulation, and only then implemented and validated on the physical system.

This second lab aims at illustrating this design workflow, known as *Model-Based Design* and presented in Section 2.1, on the rotary speed control of the inertia disc of the QUBE-Servo 2. You will see that, once a model of the system is available, the controller gains can be computed from the performance requirements, and their effect can be predicted before any test on the real system.

2.1 Methodology: Model-Based Design organized as a V-cycle

Model-Based Design (MBD) is widely used in industry for the development of control systems, for instance in the automotive, aerospace and robotics sectors. Its key idea is to put a model of the system at the centre of the development: the model is used to design the controller and to test it in simulation before any implementation on the real system. Design decisions are thus based on a model and on quantitative requirements rather than on trial and error, which makes the development faster, safer and reproducible.
The successive stages are usually organized as a V-cycle, shown in Figure 2.1 for this lab:

- the **left branch** gathers the design stages: the **requirements** are specified (Table 2.1), a model of the process is identified from a step test (**plant modelling**), and the controllers are tuned from this model (**controller design**);
- at the **bottom of the V**, the closed loop is evaluated by **Model-In-the-Loop (MIL) simulation** in Simulink;
- the **right branch** gathers the implementation and validation stages: the controllers are executed in real time on the QUBE-Servo 2 with QUARC (**rapid control prototyping**) and their performance is checked against the requirements (**experimental validation**).

Each stage of the right branch is verified against the corresponding stage of the left branch (dashed arrows): the comparison between simulated and experimental responses assesses the quality of the model, while the experimental validation checks that the requirements are met. When a verification fails, the designer iterates, for instance by retuning the controller gains. Unlike trial and error, each iteration is then guided by the model and by a clear diagnosis of what went wrong.
In industry, the code is finally generated for an embedded target and tested in Software-In-the-Loop (SIL) and Processor-In-the-Loop (PIL) before deployment; these stages are not covered in this lab.

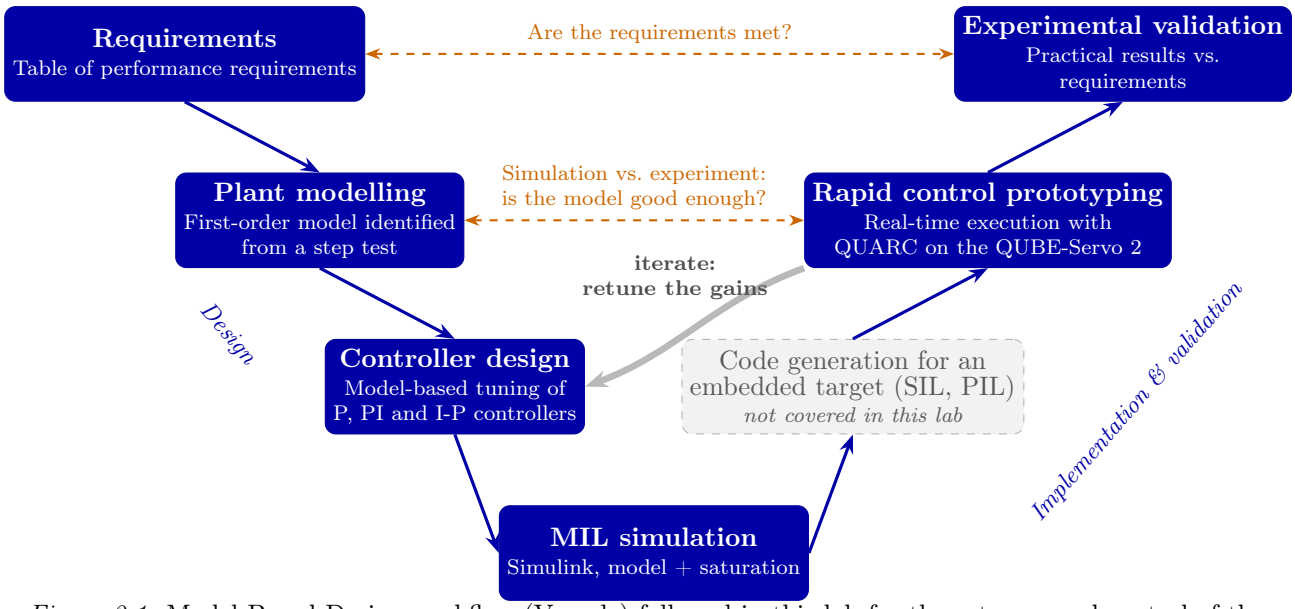


Figure 2.1: Model-Based Design workflow (V-cycle) followed in this lab for the rotary speed control of the QUBE-Servo 2

2.2 The QUBE-Servo 2 from Quanser

2.2.1 The QUBE servo-motor

The Quanser QUBE-Servo 2, pictured in Figure 2.2, is a compact rotary servo system that can be used to perform a variety of classic servo control and inverted pendulum based experiments. The QUBE can be controlled by a computer via USB connection.
The system is driven using a direct-drive 18 V brushed DC motor. The motor is powered by a built-in Pulse Width Modulation (PWM) amplifier with integrated current sense. The input of the system is the voltage $u(t)$ computed by the controller; it is limited to ± 10 V by a saturation block before being sent to the amplifier via the USB connection. Two add-on modules are supplied with the system:

- an inertia disc;
- a rotary arm+pendulum.

The two modules can be easily attached or interchanged using magnets mounted on the QUBE-Servo 2 module connector.
Encoders are used to measure the angular position of the inertia disc and pendulum. The QUBE-Servo 2 is *not* equipped with a hardware tachometer: the angular velocity of the inertia disc is estimated by a filtered differentiation of the encoder measurement:

$$\hat{\Omega}(s) = \frac{\omega_c\, s}{s + \omega_c}\, \Theta_m(s), \qquad \omega_c = 200 \text{ rad/s},$$

which corresponds to a pure derivative s followed by a first-order low-pass filter of time constant $1/\omega_c = 5$ ms.



Figure 2.2: QUBE-Servo 2 with different modules (despite the manufacturer's label, the angular velocity is not measured by a tachometer but estimated from the encoder)

During this lab, we will mainly use the QUBE-Servo 2 with the inertia disc to design and test PID-based control for the inertia disc rotary speed.

The rotary arm+pendulum module will be used in Semester 6 where you will learn how to design a more advanced state-feedback controller to balance the pendulum in its upright position.

2.2.2 Identification of the physical system components

Prior to operating any experimental system, it is imperative to familiarize yourself with the various components of the system, including its actuators and sensors. The block diagram of the open-loop chain of the QUBE-Servo 2, from the voltage computed by the controller to the estimated angular velocity of the inertia disc, is shown in Figure 2.3. The blocks on the left are implemented in software in the Simulink model executed by QUARC on the PC, while the blocks on the right are the hardware components of the QUBE-Servo 2; both parts exchange data via the USB connection.

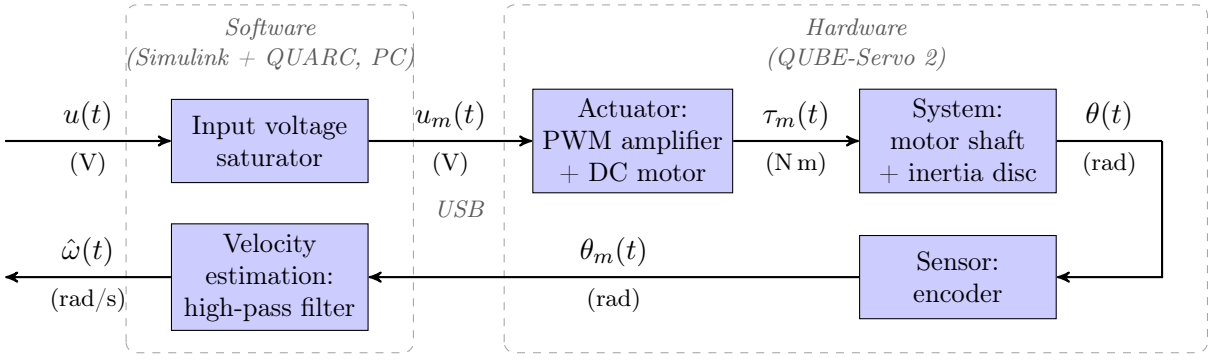


Figure 2.3: Block diagram of the open-loop chain of the QUBE-Servo 2, from the voltage $u(t)$ to the estimated angular velocity $\hat{\omega}(t)$ of the inertia disc

The role of each block is the following:

- **Input voltage saturator.** The voltage $u(t)$ computed by the controller is limited to ± 10 V before being sent to the QUBE-Servo 2, in order to protect the motor. This block is a static nonlinearity: it behaves as a unit gain, i.e. $u_m(t) = u(t)$, as long as $|u(t)| \leq 10$ V, and it has no transfer function when the voltage saturates.

- **Actuator: PWM amplifier + DC motor.** The built-in Pulse Width Modulation (PWM) amplifier applies the voltage $u_m(t)$ to the 18 V brushed DC motor, which converts this electrical energy into a motor torque $\tau_m(t)$.

- **System to be controlled: motor shaft + inertia disc.** The motor torque drives the rotation of the motor shaft and of the inertia disc attached to it (direct drive, without gearbox). The output of the system is the angular position $\theta(t)$ of the disc. The variable to be controlled is its angular velocity $\omega(t) = \dfrac{d\theta(t)}{dt}$, which is not directly measured.
- **Sensor: encoder.** An incremental optical encoder measures the angular position of the disc with a resolution of 2048 counts per revolution in quadrature mode. The number of counts is converted into the measured angle $\theta_m(t)$ in rad in the Simulink model, with a resolution of $2\pi/2048 \approx 3$ mrad.
- **Velocity estimation.** Since the QUBE-Servo 2 has no hardware tachometer, the angular velocity is estimated in the Simulink model from high-pass filtering of the measured angle:

$$\hat{\Omega}(s) = \frac{\omega_c\, s}{s + \omega_c}\, \Theta_m(s), \qquad \omega_c = 200 \text{ rad/s},$$

  i.e. a pure derivative followed by a first-order low-pass filter of time constant $1/\omega_c = 5$ ms, which limits the amplification of the encoder quantization noise.

Note that the first-order model $\dfrac{K}{1+Ts}$ identified from a step test in Section 2.5 lumps together all the blocks of Figure 2.3 in their linear range: its input is the motor voltage $u(t)$ and its output is the *estimated* angular velocity $\hat{\omega}(t)$, which is the only velocity signal available for control.

2.2.3 QUARC software

Throughout this lab, the rotary speed controller will be built as block diagrams in Simulink. The QUARC (QUanser Real-time Control) software, developed by Quanser, automatically generates C code from the Simulink diagram, compiles it into a real-time executable (.exe) and runs it on the PC. At each sampling period, this executable exchanges data via the USB cable with the data acquisition board embedded in the QUBE-Servo 2: it sends the motor voltage and reads the encoder measurements. This makes it possible to go directly from a Simulink diagram to a controller running in real time on the physical system, without writing any line of code. This approach is known as *rapid control prototyping*. Note that the QUARC blocks used to access the hardware are named *HIL* blocks (`HIL Initialize`, `HIL Read Encoder Timebase`, `HIL Write`); in Quanser's terminology, HIL simply refers to the physical hardware being part of the loop.
QUARC will therefore be used to drive the DC motor and to read the angular position of the inertia disc.
The basic steps to create a Simulink model and then exploit QUARC in order to interact with the QUBE-Servo 2 hardware are listed below. These steps are given here for information; they will be applied from Section 2.5 onwards, once the files have been downloaded and the QUBE-Servo 2 connected:

1. Make a Simulink model that interacts with the data acquisition board installed in the QUBE-Servo 2 using blocks from the *QUARC Targets* library. The required blocks have already been included in the Simulink files provided for this lab: `HIL Initialize` configures the data acquisition board, `HIL Write` sends the motor voltage to the amplifier, and `HIL Read Encoder Timebase` reads the encoder measurement.

2. Select the `HARDWARE` | `Monitor & Tune` item to build the code. This creates a QUARC executable (.exe) file. Information about the compiling process appears at the bottom left corner of the Simulink window. Various lines are also displayed in the Diagnostic Viewer window as the model is being compiled (opening this window is not mandatory).
   If the experimental procedure was followed correctly, no errors should be obtained when running the program. Time should then flow in Simulink and the LED strip at the top of the QUBE-Servo

2 should turn from red to green.
This is great news! It means you have successfully set up the connection to the QUBE-Servo 2.

3. It may happen that the connection or the compilation fails. In such a case, see the Troubleshooting section 2.10.

To know more about QUARC:

- This is not mandatory but you can watch the first 15 min of the video entitled *Getting Started with QUARC*, where Peter, an engineer at Quanser, demonstrates how to interface with the QUBE-Servo 2 hardware, build a simple Simulink model and implement it on the QUBE-Servo 2: `www.quanser.com/tutorial/quarc-essentials-hardware-interfacing/`
- Whenever necessary, you can also type `doc quarc` in Matlab to access QUARC documentation and demos.

The hands-on part of the lab starts now. From this point on, you will work on the computer and on the QUBE-Servo 2, following the stages of the V-cycle of Figure 2.1 step by step.

2.3 Control objectives and performance requirements



Figure 2.4: QUBE-Servo 2 equipped with the inertia disc

The goal is to design PID-type controllers for the rotary speed of the inertia disc attached to the Quanser QUBE-Servo 2, as shown in Figure 2.4.
The performance requirements for the rotary speed control are described in Table 2.1.

Requirement	Assessment criterion	Level
Tracking of a step setpoint for the rotary speed of the inertia disc	Steady-state error Peak overshoot Settling time at 5 % Motor input voltage	No steady-state error $D_{1\%} \approx 4.3\,\%$ $T_r^{5\%} \approx 60$ ms Limited to ± 10 V
Rejection of a constant load disturbance (light braking of the disc)	Steady-state error after the disturbance	No steady-state error

Table 2.1: Performance requirements for the rotary speed control

Speed setpoint

Unless otherwise stated, the speed setpoint $\omega_r(t)$ used in simulation and in the experimental tests is a square signal of frequency 0.2 Hz (period 5 s) switching between 0 and 50 rad/s. Each change of the setpoint is therefore a step of amplitude $\Delta\omega_r = \pm 50$ rad/s.

Rejection of a disturbance

In practice, a speed control system must not only track its setpoint but also maintain the speed when an unknown load is applied to the system. This last requirement of Table 2.1 will be assessed here for the controller, which is expected to give the best tracking performance, by braking the inertia disc lightly with a finger.

2.4 Getting started: lab files and hardware setup

To save time and help you get started in the best conditions, a set of Simulink models and Matlab scripts has been prepared for this lab. The Simulink models already include the QUARC blocks required to communicate with the QUBE-Servo 2, the ± 10 V saturation of the motor voltage and the square speed setpoint, so that you can focus on the identification, the controller design and the analysis of the results. Download these files and set up the hardware by following the procedure below:

1. Download the zipped file *Lab2.zip* from Arche, then save and unzip it into the local folder **C:/temp/**.

2. Start Matlab.

3. **Important!** By clicking on the browse for folder icon , **change the current folder of Matlab to C:/temp/Lab2**, which contains the files needed for this lab.

4. Start Simulink.

5. Connect the QUBE-Servo 2 to the PC with the USB cable, then switch it on.

2.5 Transfer function modelling of the rotary speed dynamics from a step test

Having a dynamic model of a system is crucial for effective control design. The model is used to design a control according to certain specifications and validate it before deployment on the actual system. However, the physical modelling of a complex system can be time consuming and, sometimes, not all the system parameters are known. Instead, the model of a system can be obtained experimentally by applying an input to the system and analyzing the corresponding output response.

2.5.1 Servo motor model

As shown in Problems No. 4, the motor voltage-to-angular velocity dynamics of the QUBE-Servo 2 can be described by a first-order transfer function, whose parameters can easily be estimated from a step response:

$$\frac{\Omega(s)}{U(s)} = \frac{K}{1+Ts}. \tag{2.1}$$

Since the angular velocity is not measured but estimated (see Section 2.2.1), the output of this model is in fact the estimated velocity $\hat{\Omega}(s)$. For the sake of simplicity, it will be denoted $\Omega(s)$ in the rest of this lab.

2.5.2 Recording of the step response

1. Open the file *Record_speed_step_response.slx* in Simulink.

2. Click on **Monitor & Tune** in the Hardware panel to run the step test that lasts 2.5 seconds only. The LED strip of the QUBE-Servo 2 should turn from red to green if the test succeeds.

3. Observe the rotary speed response for a positive step from 0 to 2 V applied at $t = 1$ s. The amplitude of the step can be modified according to your own reasoning and choice. To do so, click on the step block and modify the amplitude value.

The rotary speed (in rad/s) response along with the step applied to the motor voltage are recorded and saved in the *Data_speed_response_to_step.mat* file. **Note that this file is overwritten after each new step test.** An example of such a response is plotted in Figure 2.5; your own response may differ slightly.

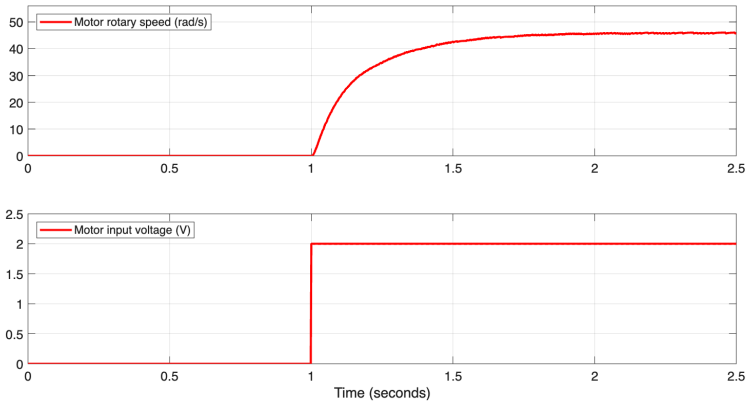


Figure 2.5: Example of rotary speed response to a 2 V step applied to the motor voltage

2.5.3 Identification of a linear first-order model

1. Open the file *plot_ speed_ response.m* in Matlab and run it. The step response recorded during your experimental test will be plotted.

2. From the rotary speed response plot, determine the steady-state gain K and the time constant T of the first-order transfer function model. The values should differ a bit from the ones obtained during the problem solving session.

3. Open the file *test_ your_ model.m* in Matlab.

4. At the beginning of the file, modify the default values of the steady-state gain K and time constant T, then run the program.

5. Compare the measured and simulated angular velocity responses. A good model should display a FIT coefficient[1] larger than 90 %.

6. If necessary, adjust the K and T model parameters to improve the fit between the measured and simulated rotary speed responses.

7. Write down the final values of K and T: they will be used throughout the rest of the lab.

2.6 Rotary speed control using a simple Proportional (P) feedback controller

The first proposed strategy is a simple proportional (P) feedback controller, shown in Figure 2.6, where $\Omega_r(s) = \mathcal{L}[\omega_r(t)]$, $\Omega(s) = \mathcal{L}[\omega(t)]$ and $U(s) = \mathcal{L}[u(t)]$.

[1]The FIT coefficient measures the agreement between the measured speed $\omega(t_k)$ and the speed $\omega_{sim}(t_k)$ simulated with the model, over the N recorded samples:

$$\text{FIT} = 100\left(1 - \frac{\sqrt{\sum_{k=1}^{N}\left(\omega(t_k) - \omega_{sim}(t_k)\right)^2}}{\sqrt{\sum_{k=1}^{N}\left(\omega(t_k) - \bar{\omega}\right)^2}}\right) \ \%,$$

where $\bar{\omega}$ is the mean value of the measured speed. FIT = 100 % corresponds to a perfect fit, while FIT = 0 % means that the model does not explain the data better than their mean value.

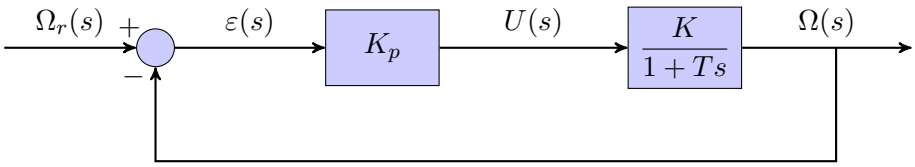


Figure 2.6: Block diagram of the simple P feedback configuration of the rotary speed control

2.6.1 Reminder of the theoretical results and numerical application

The theoretical analysis of the P controller was carried out in Problems No. 4 (section 3). The closed-loop transfer function is a first-order transfer function:

$$F_{CL}(s) = \frac{\Omega(s)}{\Omega_r(s)} = \frac{KK_p}{1 + KK_p + Ts} = \frac{K_{CL}}{1 + T_{CL}\, s} \qquad \text{with} \quad K_{CL} = \frac{KK_p}{1 + KK_p}, \quad T_{CL} = \frac{T}{1 + KK_p}$$

The closed loop is stable for any $K_p > 0$, but the steady-state tracking error for a constant setpoint ω_r is not zero: $\varepsilon(+\infty) = \omega_r/(1 + KK_p)$. Moreover, since the proportional action is applied to the error, the motor voltage jumps by $K_p\,\Delta\omega_r$ when the setpoint changes by $\Delta\omega_r$.

1. With the values of K and T identified from your step test and $K_p = 0.1$, compute K_{CL}, T_{CL}, the settling time $T_r^{5\%}$ and the steady-state tracking errors for $\omega_r = 50$ rad/s.

2. Compute the jump of the motor voltage caused by a setpoint step of $\Delta\omega_r = 50$ rad/s. Is the motor voltage requirement satisfied?

2.6.2 Simulation of the simple P feedback controller in Simulink

We will first simulate the simple P feedback controller in Simulink.

1. Open the file *Simul_ P_ control.slx* in Simulink.

2. Enter the numerical parameter values for K and T of your first-order model by clicking on the blue *Qube model* block.

3. Click on the proportional gain block and set K_p to 0.1.

4. In the **Simulation** tab, click on **Run** to simulate the feedback P control.

5. Observe the simulated rotary speed response to the square setpoint as well as the DC motor input voltage. Determine the maximum and minimum values of the input voltage. Note that the Simulink model includes the ±10 V saturation of the motor voltage.

6. Compare the simulated steady-state errors with the values computed in the previous subsection.

7. If necessary, refine the value of K_p to get the best possible answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage) with this simple P feedback control.

8. Measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage. Do not spend too much time on these measurements.

9. Close the Simulink file.

2.6.3 Experimental test of the simple P feedback controller on the physical QUBE-Servo 2

We will now proceed with the implementation and test of the P feedback controller on the QUBE-Servo 2.

1. Open the file *P_ control.slx* in Simulink.
2. Click on the proportional gain block and set K_p to 0.1.
3. In the **Hardware** tab, click on **Monitor & Tune** to execute the control on the physical platform.
4. Observe the rotary speed response to the square setpoint along with the control signal.
5. Are the experimental responses close to the simulated ones?
6. Measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage. Do not spend too much time on these measurements.
7. Are the requirement specifications fulfilled? If not, modify the P controller gain K_p to get the best answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage).
8. Conclude about the relevance of using this simple P feedback controller.

2.7 Rotary speed control using a standard PI feedback controller

The second feedback strategy considered is to use a standard PI (Proportional Integral) controller as shown in Figure 2.7, where both proportional and integral terms are applied to the error. The PI controller will be implemented in Simulink in the series form given in Figure 2.7, i.e. a gain K_p in series with the transfer function $\dfrac{1+T_i s}{T_i s}$, so that its two parameters K_p and T_i appear explicitly.

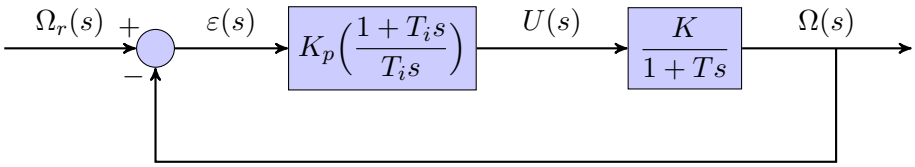


Figure 2.7: Block diagram of the classical PI feedback configuration of the rotary speed control

2.7.1 Reminder of the theoretical results and numerical application

The theoretical analysis of the standard PI controller was carried out in Problems No. 4 (section 4). The integral time constant is chosen to compensate the time constant of the process, $T_i = T$ (time-constant compensation method). The closed-loop transfer function is then a first-order transfer function with a unit steady-state gain:

$$F_{CL}(s) = \frac{1}{1+T_{CL}\,s} \qquad \text{with} \quad T_{CL} = \frac{T}{KK_p}$$

The steady-state tracking error is zero and the step response has no overshoot. However, the motor voltage still jumps by $K_p\,\Delta\omega_r$ when the setpoint changes by $\Delta\omega_r$. With the model of Problems No. 4, the value $K_p = 0.36$ gives $T_r^{5\%} = 3\,T_{CL} \approx 60$ ms.

1. With the values of K and T identified from your step test, give the value of T_i and, for $K_p = 0.36$, compute T_{CL} and the settling time $T_r^{5\%}$.

2. Compute the jump of the motor voltage caused by a setpoint step of $\Delta\omega_r = 50$ rad/s. Is the motor voltage requirement satisfied? According to the linear analysis, should the speed response display an overshoot?

2.7.2 Simulation of the standard PI feedback controller in Simulink

We will first simulate the standard PI feedback controller in Simulink.

1. Open the file *Simul_ P_ control.slx* in Simulink. Save it as *Simul_ standard_ PI_ control.slx*.
2. Modify the controller to implement the PI controller in the form given in Figure 2.7: keep the gain block K_p and add in series a *Transfer Fcn* block with numerator `[Ti 1]` and denominator `[Ti 0]`.
3. Enter the numerical parameter values for K and T of your first-order model by clicking on the blue *Qube model* block.
4. Set $K_p = 0.36$ and $T_i = T$, the time constant of your identified model.
5. In the **Simulation** tab, click on **Run** to simulate the standard PI controller.
6. Observe the simulated rotary speed response to the square setpoint.
7. Determine the maximum and minimum values of the DC motor input voltage. Does the motor voltage saturate? What is the effect of the saturation on the speed response (settling time, overshoot)?
8. If necessary, adjust the values of K_p and T_i to get the best possible answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage) with this standard PI feedback control.
9. Measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage.
10. Close the Simulink file.

2.7.3 Experimental test of the standard PI feedback controller on the physical QUBE-Servo 2

We will now proceed with the implementation and test of the PI feedback controller on the QUBE-Servo 2.

1. Open the file *P_ control.slx* in Simulink. Save it as *Standard_ PI_ control.slx*.
2. Modify the controller to implement the PI controller in the form given in Figure 2.7: keep the gain block K_p and add in series a *Transfer Fcn* block with numerator `[Ti 1]` and denominator `[Ti 0]`.
3. Set $K_p = 0.36$ and $T_i = T$, the time constant of your identified model.
4. In the **Hardware** tab, click on **Monitor & Tune** to execute the standard PI controller on the physical platform.
5. Observe the rotary speed response to the square setpoint along with the control signal.
6. Are the experimental responses close to the simulated ones?
7. Measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage.

8. The linear analysis predicts a first-order response without overshoot. Explain any overshoot observed in the speed response.

9. Compare the experimental results with the simulated ones and with the theoretical predictions.

10. Are the requirement specifications fulfilled? If not, modify the PI controller parameters K_p and T_i to get the best answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage).

11. Conclude about the relevance of using this standard PI feedback controller.

2.8 Rotary speed control using an I-P feedback controller

The third strategy is a variant of the standard PI controller in which the proportional term acts on the measured output rather than on the error, as shown in Figure 2.8. This structure is known as an I-P controller (PI controller with proportional action on the measurement). Since the setpoint only enters through the integral term, the control signal does not jump when a step setpoint is applied.

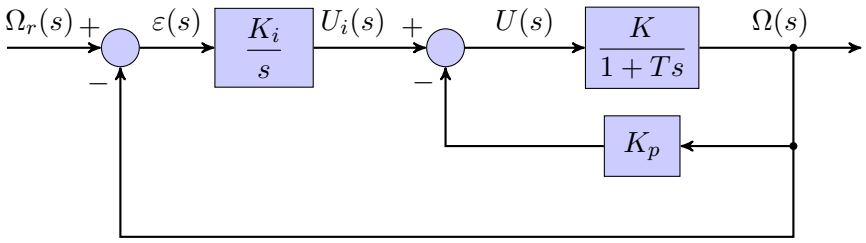


Figure 2.8: Block diagram of the I-P controller (proportional action on the output) for the rotary speed control

2.8.1 Reminder of the theoretical results and numerical application

The theoretical analysis of the I-P controller was carried out in Problems No. 4 (section 5). The closed-loop transfer function is a second-order transfer function without zero and with a unit steady-state gain:

$$F_{CL}(s) = \frac{KK_i}{Ts^2 + (1 + KK_p)\, s + KK_i}$$

The steady-state tracking error is therefore zero. The gains are tuned by pole placement by reference-model matching, so that $F_{CL}(s)$ is identical to a second-order reference model $F_{ref}(s)$ with a damping ratio z and an undamped natural frequency ω_0 derived from the performance requirements:

$$z \approx 0.707 \;\; (D_{1\%} = 4.3\,\%), \qquad \omega_0 = \frac{3}{T_r^{5\%}} = 50 \text{ rad/s} \qquad \Rightarrow \qquad K_i = \frac{T\omega_0^2}{K}, \qquad K_p = \frac{2zK_i}{\omega_0} - \frac{1}{K}$$

1. With the values of K and T identified from your step test, compute the numerical values of K_i and K_p.

2. Explain why, unlike the P and PI controllers, the motor voltage does not jump when the setpoint changes.

2.8.2 Simulation of the I-P feedback controller in Simulink

We will first simulate the I-P feedback controller in Simulink.

1. Open the file *Simul_ P_ control.slx* in Simulink. Save it as *Simul_ I-P_ control.slx*.

2. Modify the controller block to implement the I-P controller, as shown in Figure 2.8.

3. Enter the numerical parameter values for K and T of your first-order model by clicking on the blue *Qube model* block.
4. Set the values of K_p and K_i computed in the previous subsection.
5. In the **Simulation** tab, click on **Run** to simulate the I-P controller.
6. Observe the simulated rotary speed response to the square setpoint as well as the DC motor input voltage. Does the motor voltage saturate?
7. If necessary, adjust the values of K_p and K_i to get the best possible answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage) with this I-P feedback control.
8. Measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage,.
9. Close the Simulink file.

2.8.3 Experimental test of the I-P feedback controller on the physical QUBE-Servo 2

We will now proceed with the implementation and test of the I-P feedback controller on the QUBE-Servo 2.

1. Open the file *P_ control.slx* in Simulink. Save it as *I-P_ control.slx*.
2. Modify the controller block to implement the I-P controller, as shown in Figure 2.8.
3. Set the values of K_p and K_i computed from your model.
4. In the **Hardware** tab, click on **Monitor & Tune** to execute the I-P controller on the physical platform.
5. Observe the rotary speed response to the square setpoint along with the control signal.
6. Are the experimental responses close to the simulated ones?
7. By clicking on the  icon, measure the steady-state error, the peak overshoot, the settling time at 5 % and the maximum motor voltage.
8. Compare the experimental results with the simulated ones and with the theoretical predictions.
9. Are the requirement specifications fulfilled? If not, modify the I-P controller gains K_p and K_i to get the best answer to the performance requirements (overshoot, steady-state error and amplitude of the motor voltage).

2.9 Rejection of a load disturbance with the I-P controller

In practice, a speed control system must not only track its setpoint but also maintain the speed when an unknown load is applied to the system. This last requirement of Table 2.1 is assessed here for the I-P controller, which is expected to give the best tracking performance, by braking the inertia disc lightly with a finger.
The braking torque is modelled as a constant load disturbance acting at the input of the process and expressed as an equivalent motor voltage $D(s)$, as shown in Figure 2.9:

$$\Omega(s) = \frac{K}{1+Ts}\big(U(s) - D(s)\big)$$

where $D(s) = \mathcal{L}[d\,\Gamma(t)]$ and $d > 0$ is the amplitude of the disturbance.

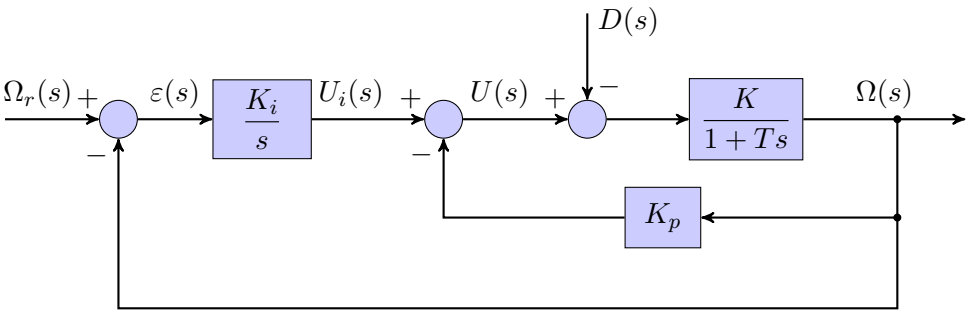


Figure 2.9: I-P closed-loop block diagram with a load disturbance acting at the process input

To have enough time to apply the disturbance on a constant setpoint, the frequency of the square setpoint is reduced to **0.02 Hz** (period 50 s) in this section: each plateau at 0 or 50 rad/s then lasts 25 s.

2.9.1 Theoretical analysis

This analysis was not covered in Problems No. 4.

1. Determine the transfer function $\Omega(s)/D(s)$ of the I-P closed loop when $\Omega_r(s) = 0$.
2. By using the final value theorem, calculate the steady-state speed variation caused by a step disturbance $d\,\Gamma(t)$. Is the disturbance rejection requirement satisfied? Which term of the controller is responsible for it?

2.9.2 Simulation of a load disturbance rejection in Simulink

5. Open your file *Simul_ I-P_ control.slx* and set the frequency of the square setpoint to 0.02 Hz.
6. Add a step disturbance $d\,\Gamma(t - t_d)$ subtracted from the motor voltage at the process input, with $d = 1$ V, the instant t_d being chosen in the middle of a 50 rad/s plateau.
7. Simulate the closed-loop responses $\omega(t)$ and $u(t)$ and check that the simulation results agree with your theoretical analysis.

2.9.3 Experimental test of a load disturbance rejection on the QUBE-Servo 2

Important. Apply only a light braking on the edge of the inertia disc, with the tip of your finger, for about 1 to 2 s. Do not stop the disc: an excessive braking saturates the motor voltage.

8. Open your file *I-P_ control.slx* and set the frequency of the square setpoint to 0.02 Hz. Execute the I-P controller on the QUBE-Servo 2.
9. Once the steady state is reached on a 50 rad/s plateau, brake the disc lightly as described above, then release it. Record and plot $\omega(t)$ and $u(t)$.
10. Determine from your plots:
    - the steady-state speed error while the disc is braked;
    - the time needed to recover the speed setpoint after the beginning and after the end of the braking;
    - the variation of the motor voltage $u(t)$ while the disc is braked.

11. Explain the evolution of the motor voltage $u(t)$ during the braking. What does it reveal about the role of the integral action?
12. Compare the experimental results with the simulated ones and with your theoretical analysis.
13. Is the disturbance rejection requirement of Table 2.1 satisfied?
14. Conclude about the relevance of using this I-P feedback controller.

2.10 Troubleshooting

How to fix the following QUARC license issue that may occur:
"Error occurred while executing External Mode MEX-file 'quarc_comm'

- Make sure that the working directory in Matlab is `C:/temp/`. If not, go to the main window of Matlab. By clicking on the browse for folder icon , **change the current folder of Matlab so that it becomes your `C:/temp/` folder**.
- Unplug and re-plug the QUBE-Servo 2 to reset the micro-controller. Try to compile the Simulink file.
- Test your Simulink file with another QUBE-Servo 2.

If this does not fix the problem, you will need to re-activate the QUARC License on your computer. Follow the steps below:

- Close Matlab and log out.
- Log in with the following login: `.\admin_IA2R`
- Ask the instructor to enter the password.
- Go to the `C:\temp\` directory.
- Double-click on the license file: `Quarc essentials Polytech Nancy`.
- Tick if necessary both options and save the file.
- Log out of the admin account.
- Log in with your student account.
- Unplug and re-plug the QUBE-Servo 2 to reset the micro-controller.
- Start Matlab again.

Lab 3

Model-based PID control design for the mini-drone TELLO

Pre-lab questions. Review Problems No. 5 and their solutions. Before coming to the lab, make sure you have a full understanding of these solutions. Note that they might need to be slightly adapted to the experimental conditions.

This third lab provides an introduction to drone control, focusing on the altitude control of the TELLO mini-drone, a small quadcopter controlled from a PC via WiFi.
As in the previous lab on the QUBE-Servo 2, the controllers will not be tuned by trial and error directly on the drone, but designed by following the *Model-Based Design* methodology organized as a V-cycle and presented in Section 3.1. A model of the vertical dynamics is first identified from experimental data, the P and PD controllers are then designed from this model to meet the performance requirements, tested in simulation, and only then implemented and validated on the real drone.
This approach is even more relevant here than for the QUBE-Servo 2: a badly tuned controller does not only give a poor response, it may make the drone hit the ceiling, the floor or the net.

Safety warning message: during this lab, you must exercise extreme caution when handling your mini-drone. Ensure that you follow all safety instructions. Unauthorized maneuvers or negligence can lead to injuries or damage to equipment.

3.1 Methodology: Model-Based Design organized as a V-cycle

The stages of the Model-Based Design methodology followed in this lab are shown in Figure 3.1. They are the same as in the lab on the QUBE-Servo 2:

- the **left branch** gathers the design stages: the **requirements** are specified (Table 3.2), a model of the vertical dynamics is identified from an open-loop test (**plant modelling**), and the P and PD controllers are tuned from this model (**controller design**);
- at the **bottom of the V**, the closed loop is evaluated by **Model-In-the-Loop (MIL) simulation** in Simulink;
- the **right branch** gathers the implementation and validation stages: the control laws are coded in Python and run on the PC, which exchanges data with the drone via WiFi (**implementation in Python** stage); the performance is then checked against the requirements during flight tests (**experimental validation** stage).

The main difference with the QUBE-Servo 2 lies in the implementation stage. There is no QUARC and no HIL blocks here: the Simulink diagram is not converted automatically into real-time code. The discrete-time control law must be *written by hand* in Python, in a loop that, at each sampling period, reads the altitude measured by the drone, computes the command and sends it via WiFi.

The implemented controller must therefore be checked carefully against the one simulated in Simulink. Moreover, the WiFi communication introduces a small time-delay that is not included in the model.

As before, each stage of the right branch is verified against the corresponding stage of the left branch (dashed arrows), and the designer iterates when a verification fails, the iterations being guided by the model.

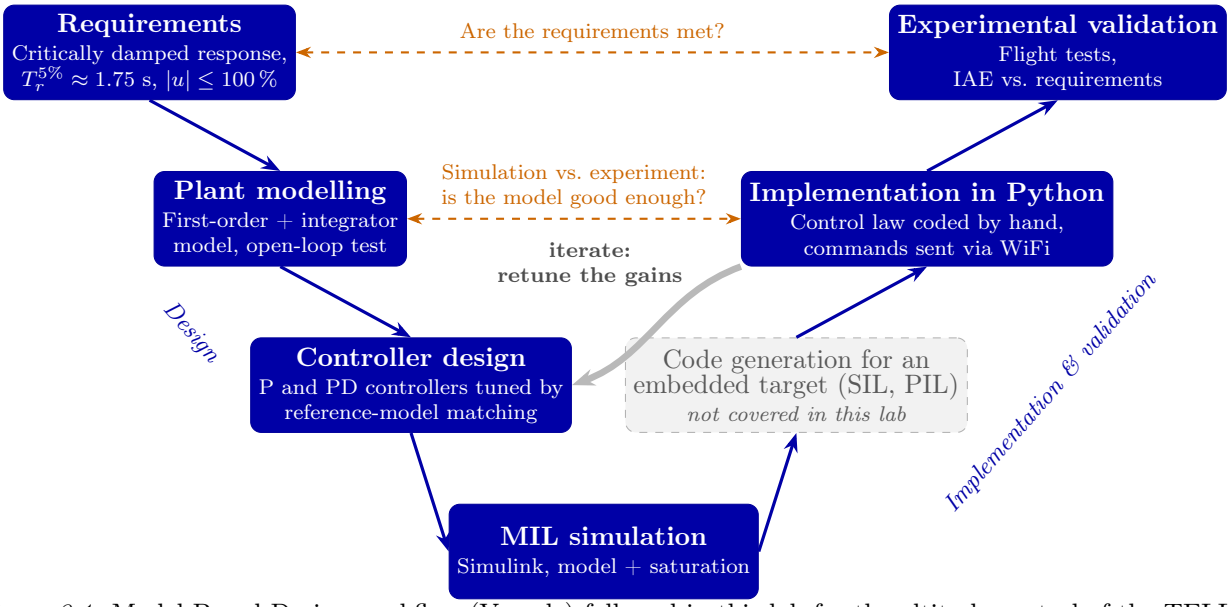


Figure 3.1: Model-Based Design workflow (V-cycle) followed in this lab for the altitude control of the TELLO mini-drone

3.2 The TELLO mini-drone

Prior to operating any experimental system, it is imperative to familiarize yourself with the various components of the system. Part of the process involves the identification of the individual hardware components, including its sensors and actuators.

The TELLO mini-drone is a small quadcopter that features a vision positioning system and an onboard camera as shown in Figure 3.2.

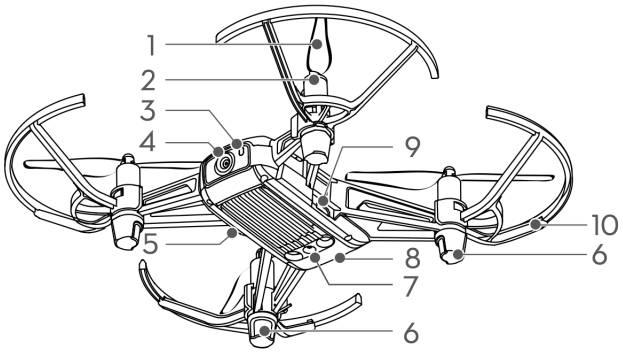


Figure 3.2: Tello mini-drone from DJI

Using its vision positioning system and advanced flight controller, it can hover in place and is suitable for flying indoors. Its maximum flight time is about 13 minutes. Whenever you can, connect the TELLO mini-drone to the USB port of your PC to recharge the battery for the next test.

3.2.1 Main components

The main components of the mini-drone are presented in Table 3.1.



1. Propellers	6. Antennas
2. Motors	7. Vision positioning system
3. Aircraft status indicator	8. Flight battery
4. Camera	9. Micro USB port
5. Power button	10. Propeller guards

Table 3.1: Main components of the TELLO mini-drone

3.3 Understanding the basics of quadcopter control

There are many types of Unmanned Aerial Vehicles (UAVs), but the TELLO mini-drone used is a quadcopter. A quadcopter is a type of helicopter which has 4 rotors. Each one of them produces thrust and torque at the same time.
To understand how a quadcopter flies, it is very important to have the concept of 6-degrees of freedom (6-DOF).
The 6 degrees of freedom is a representation of how an object moves through 3D space by either translating linearly or rotating axially as shown in Figure 3.3.

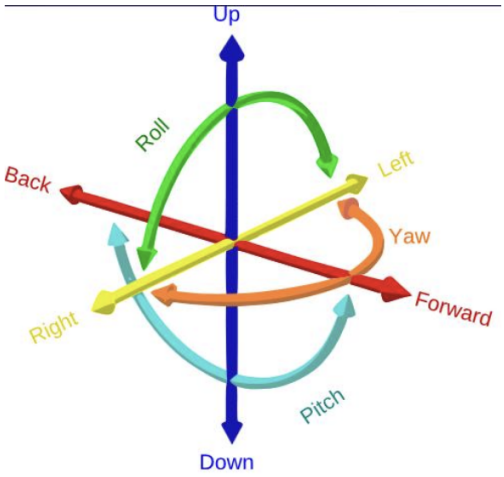


Figure 3.3: The 6 degrees of freedom

Specifically, the drone can move in three dimensions, along the X, Y and Z axes (forward/back, left/right, up/down), and can also rotate about these axes. These rotations are called yaw, roll and pitch, and are defined below:

- **Yaw (lacet):** rotation of the drone about its vertical axis passing through its center of gravity; when hovering, the drone turns on the spot while remaining level. This can be likened to shaking our head to say "no".
- **Roll (roulis):** rotation of the drone about its longitudinal (front-to-back) axis, so that one side goes down while the other goes up. This can be likened to tilting our head toward one shoulder.
- **Pitch (tangage):** rotation of the drone about its lateral (left-to-right) axis, so that the nose goes up or down. This can be likened to nodding our head to say "yes".

Impact on the motors

The drone is equipped with four rotors, two of which rotate clockwise and two of which rotate counterclockwise. Rotors spinning in the same direction are diagonally opposite.

Hovering

When all rotors spin at the same speed, the reaction torques of the clockwise and counterclockwise rotors cancel out, so the drone does not rotate. If, in addition, the total thrust produced by the four rotors equals the weight of the drone, it hovers in place, as shown in Figure 3.4.

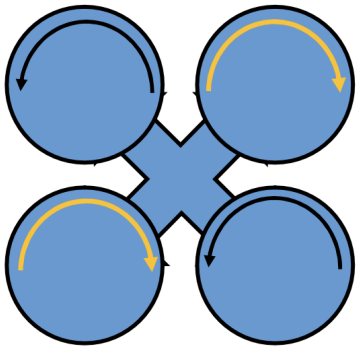


Figure 3.4: Principle of hovering

Ascending or descending

To make the drone ascend, the thrust is increased equally on all four rotors; conversely, it is decreased equally to make the drone descend, as shown in Figure 3.5.

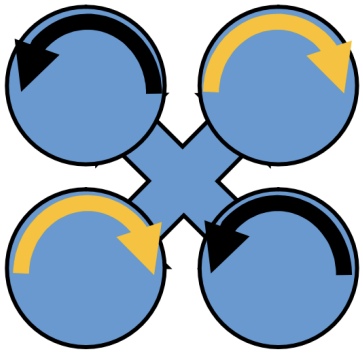


Figure 3.5: Principle of ascent and descent

Yaw

Yaw is obtained from the reaction torques of the rotors. To turn clockwise (viewed from above), the speed of the two counterclockwise rotors is increased while that of the two clockwise rotors is

decreased, so that the total thrust remains unchanged. The opposite is done to turn counterclockwise, as illustrated in Figure 3.6.

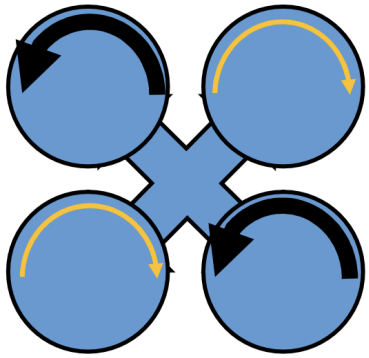


Figure 3.6: Principle of yaw

Roll and Pitch

To move forward/backward or left/right, the thrust of the rotors on the side toward which we want to move is decreased, while that of the opposite rotors is increased.
For instance, increasing the speed of the front rotors while decreasing that of the rear rotors raises the nose (pitch up), and the drone moves backward, as shown in Figure 3.7. Similarly, increasing the speed of the left rotors while decreasing that of the right rotors makes the drone roll to the right, and it moves to the right.

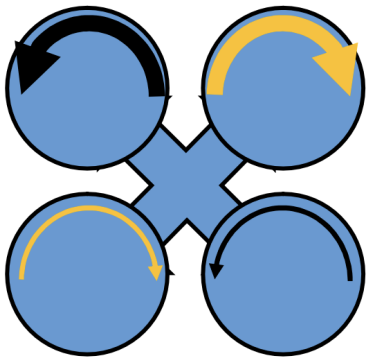


Figure 3.7: Principle of roll and pitch

Note – from attitude to translation.

A quadrotor drone is *underactuated*: its four rotors only produce a thrust force along the drone's vertical axis, plus torques. It therefore cannot translate horizontally without first changing its attitude:

- **Yaw** changes only the heading. It is obtained from the reaction torques of the rotors and produces no translation.
- **Roll** tilts the thrust vector sideways; its horizontal component makes the drone move laterally, toward the lowered side.
- **Pitch** tilts the thrust vector forward or backward, which produces forward or backward motion.

Since only the vertical component of the thrust compensates gravity, the total thrust T must be increased when the drone is tilted to maintain its altitude:

$$T \cos\phi \cos\theta = mg,$$

where ϕ and θ denote the roll and pitch angles, m the mass of the drone and g the gravitational acceleration.

3.3.1 Videos to understand the basics of drone control design

To know more about the basics of drone control, you can watch Brian Douglas's introduction videos:
Drone Simulation and Control, Part 1: Setting Up the Control Problem

Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover?

3.3.2 Manual control

From the first video, we know that by manipulating the four motor speeds in specific ways through the Motor Mixing Algorithm (MMA), thrust, roll, pitch, and yaw can be controlled directly and so we are able to control the drone in 3D space. This is the open-loop control configuration as shown in Figure 3.8 that you have used for controlling the TELLO with the App on your mobile device.

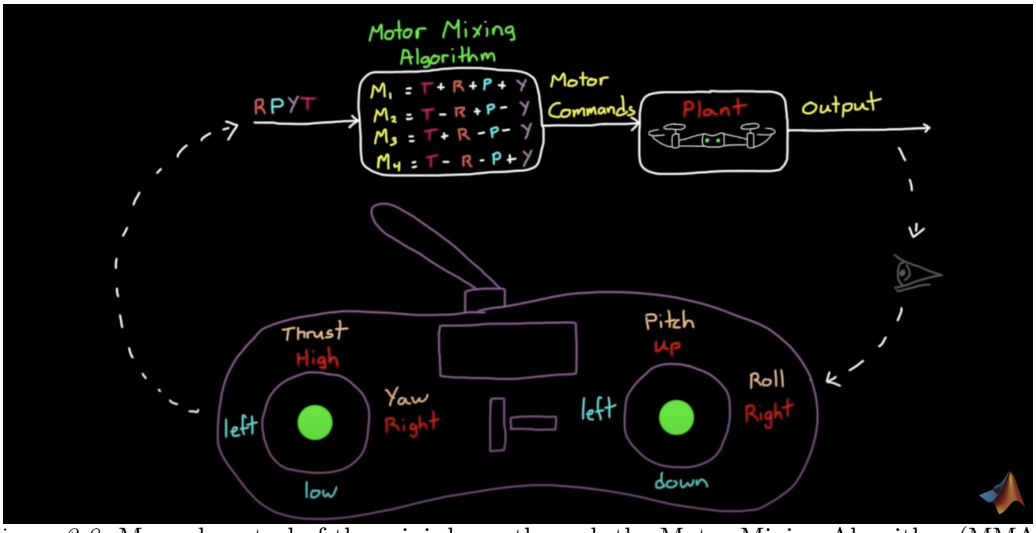


Figure 3.8: Manual control of the mini-drone through the Motor Mixing Algorithm (MMA).
From Brian Douglas's video: Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover?

3.3.3 Full closed-loop control

From the second video, we know that the full closed-loop control architecture is quite complex and looks like the one shown in Figure 3.9 with several different control loops and 6 different PID controllers.

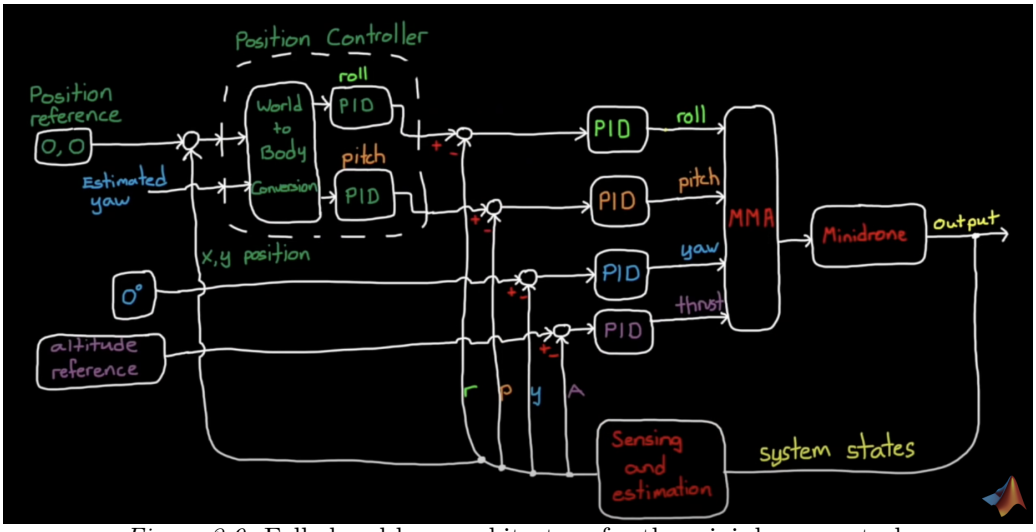


Figure 3.9: Full closed-loop architecture for the mini-drone control.
From Brian Douglas's video: Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover?

The hands-on part of the lab starts now. From this point on, you will work on the computer and on the Tello mini-drone, following the stages of the V-cycle of Figure 3.1 step by step.

3.4 Altitude control of the TELLO mini-drone

During this lab, the goal is to consider a single loop: the altitude loop. Remember, this is independent of the other loops so we can tweak and adjust altitude without affecting roll, pitch, or yaw. To make sure they are out of the equation completely, we will just set the commands to 0 for roll, pitch, and yaw as shown in Figure 3.10.

The goal is therefore first to design a control for the mini-drone that uses thrust to adjust the altitude. If we are able to measure the drone altitude then we can feed it back to compare it to an altitude reference. The resulting error is then fed into a PID controller that will determine how to increase or decrease thrust.



Figure 3.10: Altitude control architecture for the mini-drone.
From Brian Douglas's video: Drone Simulation and Control, Part 2: How Do You Get a Drone to Hover?

The input and output of the mini-drone altitude control system are:

- $u(t)$: vertical speed command in % sent to the onboard flight controller (related to the thrust generated by the 4 motors);
- $z(t)$: altitude or vertical position in cm along the Z axis of the mini-drone.

3.4.1 Altitude control performance requirements

The performance requirements for the mini-drone altitude control are described in Table 3.2.

Requirement	Assessment criterion	Level
Tracking of a square set-point	Steady-state error	No steady-state error
for the mini-drone altitude	Overshoot	No overshoot allowed (the step response should be critically damped)
	Settling time at 5 %	$T_r^{5\%} \approx 1.75$ s
	Vertical speed command	Limited to $\pm 100\,\%$

Table 3.2: Performance requirements for the mini-drone altitude control

3.4.2 Open-loop control of the TELLO mini-drone by using its App

1. Search for "Tello" on the App Store or Google Play to download the TELLO App on your mobile device.
2. Go to the room C335.
3. Press once the power button to turn the mini-drone on.
4. Place your TELLO in the middle of the net cage with the in-front camera facing the window.
5. Go outside the net cage.
6. Launch the TELLO App on your mobile device.
7. Enable the Wi-Fi on your mobile device and connect to the TELLO XXXXXX network. Connection has been established when the LED indicator blinks yellow slowly and the live camera view is shown on your mobile device.
8. Select Auto Takeoff.
9. Use the virtual joysticks in the App to test your ability to open-loop control the mini-drone. Try to control the drone to make a square trajectory of 50 cm.
10. Select Auto landing.

3.4.3 Download of the files required for the lab

1. Download the zipped file `Lab_Tello.zip` from Arche. Save and unzip it in your `Documents/Matlab/Control_Lab/` folder.
2. Start Matlab.
3. Important ! By clicking on the browse for folder icon , **change the current folder of Matlab so that it becomes your** `Documents/Matlab/Control_Lab/` **folder** that contains the files needed for this lab.
4. In the Current Folder window of Matlab, click right on the folder `Documents/Matlab/Control_Lab/Lab_Tello` and select `add to path the selected folder`.
5. Double-click on the folder `Lab_Tello` so that it becomes your current folder. You should see the different `.slx`, `.m` and `.py` files needed for this Lab.

3.4.4 Vertical dynamic model identification

The determination of a model is a crucial step for the design of a feedback control system. Unfortunately, DJI did not provide a physics-based model of the TELLO mini-drone. For this reason, the Tello mini-drone dynamics will be approached by black-box models that will be identified from input and output data.

3.4.4.1 Transfer function model of the vertical dynamic

The vertical speed command-to-altitude transfer function can be modelled as a first-order plus pure integrator model

$$\frac{Z(s)}{U(s)} = \frac{K}{s(1+Ts)} \tag{3.1}$$

$Z(s) = \mathcal{L}[z(t)]$ where $z(t)$ is the mini-drone vertical position (in cm) and $U(s) = \mathcal{L}[u(t)]$ in %.
K is the model gain and T is the model time-constant.

As the vertical velocity on the Z axis $v_z(t)$ is the time-derivative of the vertical position or altitude $z(t)$, both variables are linked in the Laplace domain by an integrator so that (3.1) can be expressed as:

$$\frac{Z(s)}{U(s)} = \frac{Z(s)}{V_z(s)} \times \frac{V_z(s)}{U(s)} = \frac{1}{s} \times \frac{K}{1+Ts} \tag{3.2}$$

where $V_z(s) = \mathcal{L}[v_z(t)]$ is the vertical velocity of the mini-UAV on the Z axis.
Identifying a system having a pure integrator is tricky and it is easier when the measure is available to identify the response between the vertical velocity and the input (the vertical speed command here) since the transfer function takes the form of a simple first-order model:

$$\frac{V_z(s)}{U(s)} = \frac{K}{1+Ts} \tag{3.3}$$

3.4.4.2 Model identification from square wave response experiments

Recording of the square wave response

This experiment is carried out in open-loop/manual mode, i.e. there is no feedback control to the altitude. As the system is marginally stable (due to the pure integrator in the model (3.2)), the input command should be designed with special care. As shown in Figure 3.11, the input command is a square wave input (a series of positive and negative steps) around a working operating point (a given altitude here). The duration of each step should be carefully chosen so that the drone vertical position increases and decreases of about 30 cm without hitting the ceiling.

Warning. Remember that the experiment is conducted in open loop. If there are disturbances, like small, invisible airflow, they will induce a little roll or pitch changes into the system, the thrust will then not only adjust altitude but also create some horizontal motions and the mini-drone will start to move away (to drift) from the centre of the room and possibly crashes into the net. It is therefore recommended **to close the door** of the classroom when the open-loop control test is carried out.

1. Go to classroom C335 with your TELLO mini-drone.
2. Log in to the PC with the following account: .\`admin_IA2R`
3. Ask the instructor to enter the password.
4. Go to the `C:/temp/Tello` folder.

5. Open Python (select if necessary version 3.10.0 in the bottom right corner). Go to the File tab and select `Open Folder Disque local (C:)/temp/Tello` (this will make sure that the .txt file are recorded in the Tello folder). Double-click on `Zcontrol_in_OL.py`.

6. Press once the power button to turn the mini-drone on.

7. Connect your mini-drone to the WIFI.
   Each mini-drone can be connected to the computer in room C335 when it is turned on via WIFI. You can identify your mini-UAV by removing the battery and looking inside the battery case. You should see written on a white paper
   WIFI:*TELLO-****XXXXXX*** where the six ***X*** represent the ID of your mini-UAV.

8. Place your Tello in the middle of the net cage with the in-front camera facing you.

9. Go outside of the cage.

10. Close the door of the room.

11. Run the python file. Click on the triangle in the top right corner or choose `Select Run without debugging` and then click on `Trust Workspace & Continue` or `Python debugger and Yes`.

12. The mini-drone should auto-take off at about 1 m and wait for 3 seconds. Then a square wave input will be sent to the vertical speed command. The drone should go up to about 1.50m and then go down to about 1m four times and finally auto-land.

13. If everything went well, a figure will be plotted showing the experimental response data that are also recorded in the file `DataOL_Zcontrol.txt`.

14. Copy the .txt file on a USB key or send it to your account by e-mail.

15. Delete the .txt file.

16. Go back to classroom C311.

Identification of the vertical dynamic transfer function model

1. Open Matlab and execute the `dataOL_plot.m` file to get a plot similar to Figure 3.11.

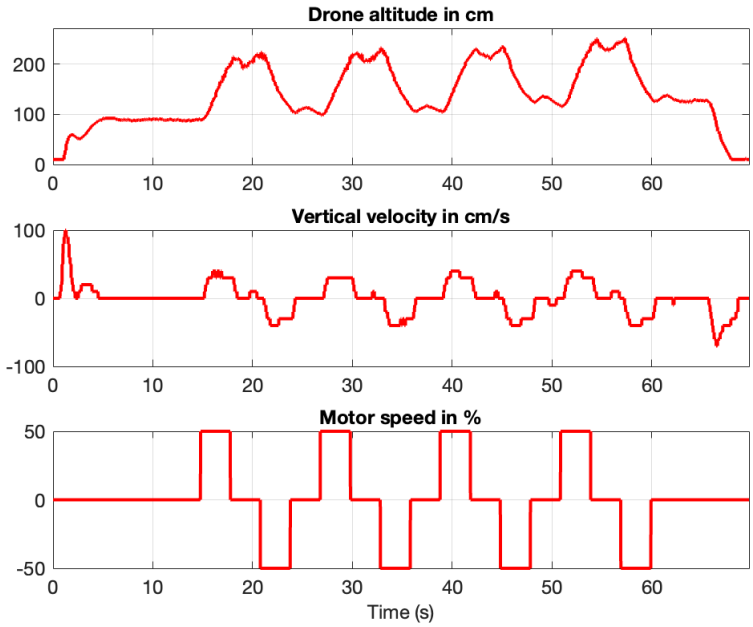


Figure 3.11: Vertical position and velocity responses of the mini-drone to a square wave input

2. Determine the quantization step size for both the altitude measure and vertical velocity measure.

3. Open and run the `test_your_model.m` file. This file will help you to adjust the first-order model parameters by using the data you recorded from initial default K and T parameter values set at the top of the file.

4. Run the file and observe the FIT percentage between the measured and simulated model vertical speed response.

5. Adjust the parameter values for K and T at the top of the .m file to get the best fit possible between the two responses.

Note that a time delay of about 0.3 s can be observed on the vertical velocity response. It is mainly due to the WiFi communication between the PC and the mini-drone, the onboard processing and the limited resolution of the velocity measurement. For simplicity, this time delay is neglected in the model and in the design of the P and PD controllers. Since it is not negligible compared with the time constant T, a slower response and an overshoot of the order of 10 to 15 % can be expected on the real mini-drone.
Now we have a relatively good model of the vertical dynamic of the drone, we can design a model-based PID control for altitude tracking.
For systems represented by a first-order transfer function plus a pure integrator, a simple P controller or a PD controller should be enough to get good tracking performance.

In the closed-loop tests, the altitude setpoint $z_r(t)$ is a square wave whose steps have an amplitude $\Delta z_r = 40$ cm around the hovering altitude.

3.4.5 P control for altitude tracking of a square wave signal

The first proposed feedback strategy is to use a simple P controller as shown in Figure 3.12.

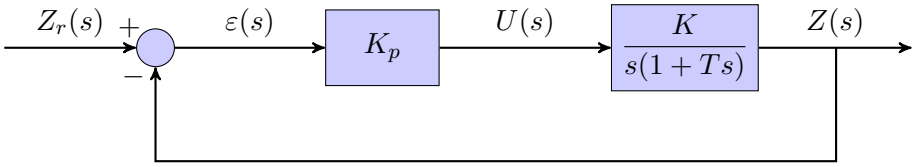


Figure 3.12: Block diagram of the simple P feedback configuration of the mini-drone altitude control

3.4.5.1 Reminder of the theoretical results and numerical application

The theoretical analysis of the P controller was carried out in Problems No. 5 (sections 1 and 3). The main results are recalled below.

Reference model. Each requirement of Table 3.2 fixes one feature of the desired closed-loop transfer function: no steady-state error gives a unit static gain, no overshoot gives a critically damped response ($z = 1$), and the settling time fixes the undamped natural frequency ω_0. The reference model is therefore

$$F_{ref}(s) = \frac{\omega_0^2}{(s+\omega_0)^2} \qquad \text{with} \qquad \omega_0 \approx \frac{4.75}{T_r^{5\%}} \approx 2.71 \text{ rad/s} \quad \text{for } T_r^{5\%} \approx 1.75 \text{ s}$$

where $\omega_0 T_r^{5\%} \approx 4.75$ holds for a critically damped second-order system. The requirement on the speed command does not appear in $F_{ref}(s)$: it must be checked once the controller is known.

Closed-loop behaviour with a P controller. With the plant model (3.1), the closed-loop transfer function is a second-order transfer function without zero and with a unit static gain:

$$F_{CL}(s) = \frac{KK_p}{Ts^2 + s + KK_p} \qquad \Rightarrow \qquad \omega_0 = \sqrt{\frac{KK_p}{T}}, \qquad z = \frac{1}{2\sqrt{K_p KT}}$$

- The closed loop is stable for any $K_p > 0$.
- The steady-state error to a step setpoint is zero: the integrator is brought by the plant itself, so that no integral action is needed in the controller (unlike the QUBE-Servo 2).
- The critically damped response ($z = 1$), i.e. the fastest response without overshoot, is obtained for

  $$K_p = \frac{1}{4KT} \qquad \Rightarrow \qquad \omega_0 = \frac{1}{2T}, \qquad T_r^{5\%} \approx \frac{4.75}{\omega_0} = 9.5\,T$$

  The settling time is therefore entirely imposed by the plant time constant T: with a single gain, only one of the two specifications (z or ω_0) can be imposed. A larger K_p speeds up the response but makes it overshoot ($z < 1$).
- The speed command is maximum just after a setpoint step of amplitude Δz_r: $u(0^+) = K_p\,\Delta z_r$.

With the model of Problems No. 5 ($K = 0.8$ (cm/s)/%, $T = 0.24$ s), this gives $K_p \approx 1.30$ %/cm, $\omega_0 \approx 2.08$ rad/s and $T_r^{5\%} \approx 2.28$ s, i.e. a response about 30 % too slow.

1. With the values of K and T identified from your own experiment, compute the gain K_p giving the critically damped response, the corresponding undamped natural frequency ω_0 and the settling time $T_r^{5\%}$.
2. Compute the speed command $u(0^+)$ just after a setpoint step of amplitude Δz_r equal to that of the square wave setpoint. Is the speed command requirement satisfied?
3. Are the performance requirements satisfied with this simple P controller? What would happen if K_p were increased to speed up the response?

3.4.5.2 Simulation of the P controller performance in Simulink

We start by testing the performance of a simple P controller using Simulink before implementing the P control in Python code on the TELLO mini-drone.

1. Open the Simulink file `Simul_P_Zcontrol.slx`.
2. Enter the parameters of your identified model K and T.
3. Set the gain K_p of the P controller.
4. Run the file.
5. Are the performance requirements satisfied in simulation ? If not adjust the gain K_p of the P controller.
6. Analyze the vertical speed command. Does it reach the saturation ?
7. Give the few Python lines to implement the P control.

3.4.5.3 Experimental test of the simple P feedback controller on the TELLO drone

Warning. The time delay of about 0.3 s observed on the vertical velocity is neglected in the model. Since it is not negligible compared with T, a slower response and an overshoot of the order of 10 to 15 % can be expected on the real mini-drone.

1. Go to classroom C335 with your mini-drone.
2. Go to the `C:/temp/Tello` folder.
3. Open the file `Zcontrol_in_CL.py`.
4. Set the P controller gain below the commented line: `Enter the P gain value below:`
5. Press once the power button to turn the mini-drone on.
6. Connect your mini-drone to the WIFI.
7. Place your TELLO in the middle of the net cage with the in-front camera facing you.
8. Go outside the cage.
9. Close the door of the room.
10. Run the python file.
11. The mini-drone should auto-take off at about 1 m and wait for 3 seconds. Then a square wave will be applied to the altitude setpoint. The drone should track the square wave setpoint and finally auto-land.
12. If everything goes well, a figure will be plotted showing the experimental response data that is also recorded in the file `DataCL_Zcontrol.txt`.
13. Copy the .txt file on a USB key or send it to you by e-mail to include it later in your report.
14. Delete the .txt file.
15. Go back to classroom C311.

16. Execute the `dataCL_plot.m` file to get a plot similar to Figure 3.13.

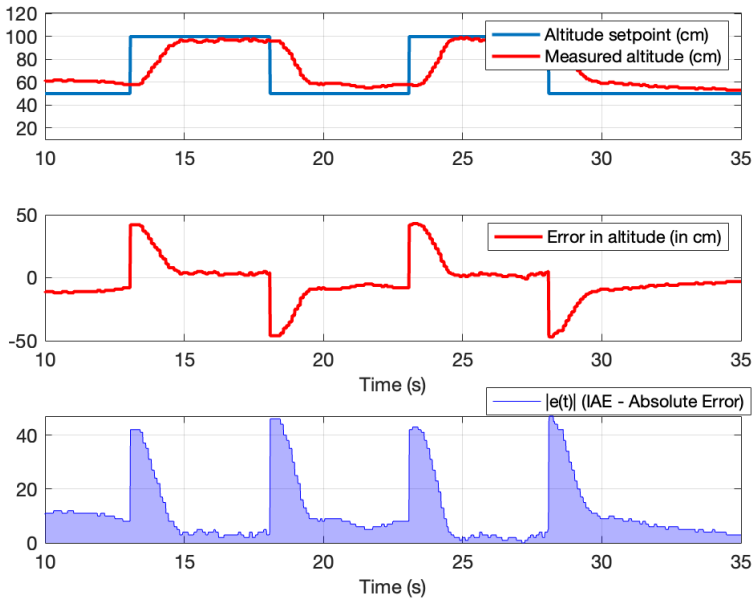


Figure 3.13: Closed-loop response of the mini-drone to a square wave altitude setpoint

To help you to compare the control performance for different K_p values, the Integral of Absolute Error (IAE) performance index over time is plotted and its final value is displayed in the command window. IAE is a standard performance index which is computed as:

$$\text{IAE} = \int_0^{T_f} |\varepsilon(t)|\, dt = \int_0^{T_f} |z_r(t) - z(t)|\, dt \tag{3.4}$$

It measures the total magnitude of errors over time without additional weighting. It is useful for understanding the overall error accumulation without bias toward time or magnitude.

17. Analyze the vertical speed command displayed in Figure 1 in Matlab. Does the command reach the saturation ?

18. Analyze the altitude tracking with the simple P controller. Are the performance requirements satisfied in practice.

19. Modify the value of K_p and repeat the closed-loop test to get the best results you can. Note the best value of IAE performance index for this simple P control.

3.4.6 PD control for altitude tracking of a square wave signal

The proposed strategy is to use a variation of the standard PD control, where unlike the standard PD where the derivative term is usually applied to the error, it is applied to the output, as shown in Figure 3.14. This PD variation presents the advantage of obtaining a closed-loop transfer function (for a first-order +pure integrator plant model) without any zero as in the desired closed-loop transfer function $F_{ref}(s)$.

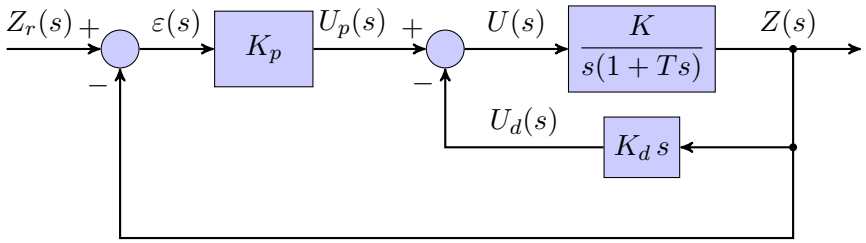


Figure 3.14: Block diagram of the PD controller with derivative action on the output for the mini-drone altitude control

3.4.6.1 Reminder of the theoretical results and numerical application

The theoretical analysis of the PD controllers was carried out in Problems No. 5 (sections 4 and 5). The main results are recalled below.

Why the derivative action is applied to the output. A PD controller provides the second tuning parameter that the P controller lacks. With the *standard* PD controller, where the derivative action is applied to the error, the closed-loop transfer function has the right denominator but keeps the zero of the controller at $s = -K_p/K_d$: the closed loop cannot behave like $F_{ref}(s)$. Moreover, the setpoint steps are differentiated, which produces a very large peak of the speed command ("derivative kick"). Applying the derivative action to the measured output removes this zero and this peak. Since $K_d\, s\, Z(s) = K_d\, V_z(s)$, the derivative action is simply a feedback of the vertical velocity, which is measured by the drone.

Closed-loop behaviour and tuning. With the plant model (3.1), the closed-loop transfer function of Figure 3.14 is a second-order transfer function without zero and with a unit static gain:

$$F_{CL}(s) = \frac{K K_p}{T s^2 + (1 + K K_d)\, s + K K_p}$$

The steady-state error to a step setpoint is again zero, thanks to the integrator of the plant. Since K_p and K_d act on two different coefficients of the characteristic polynomial, $F_{CL}(s)$ can now be made identical to the critically damped reference model $F_{ref}(s)$ with $\omega_0 \approx 2.71$ rad/s:

$$F_{CL}(s) = F_{ref}(s) \Rightarrow \begin{cases} K_p = \dfrac{T\omega_0^2}{K} \\ K_d = \dfrac{2K_p}{\omega_0} - \dfrac{1}{K} = \dfrac{2\omega_0 T - 1}{K} \end{cases}$$

Note that $K_d > 0$ requires $\omega_0 > 1/(2T)$: the derivative action is useful because the required response is faster than the critically damped P controller.

Speed command. The transfer function from the setpoint to the speed command is

$$\frac{U(s)}{Z_r(s)} = \frac{K_p\, s\, (1 + Ts)}{T s^2 + (1 + K K_d)\, s + K K_p}$$

so that, by the initial value theorem, $u(0^+) = K_p\, \Delta z_r$ for a setpoint step of amplitude Δz_r: at $t = 0^+$ the drone has not moved yet and the velocity feedback is zero. This value is the maximum of $u(t)$ during the step response. The largest setpoint step that respects the ± 100 % limitation is therefore $\Delta z_{r,\max} = 100/K_p$.
With the model of Problems No. 5, this gives $K_p \approx 2.21$ %/cm and $K_d \approx 0.38$ % s/cm, and $u(0^+) \approx 88$ % for a 40 cm setpoint step ($\Delta z_{r,\max} \approx 45$ cm).

1. With the values of K and T identified from your own experiment, compute the gains K_p and K_d of the PD controller with derivative action on the output.
2. Compute the speed command $u(0^+)$ just after a setpoint step of amplitude Δz_r equal to that of the square wave setpoint, and the largest admissible step $\Delta z_{r,\max}$. Is the speed command requirement satisfied?
3. Explain why this PD controller can be implemented in the Python program without differentiating any signal.

3.4.6.2 Simulation of the PD controller performance in Simulink

1. Open your Simulink file `Simul_P_Zcontrol.slx`.
2. Save it as `Simul_PD_Zcontrol.slx`.
3. Modify the controller to implement the PD controller.
4. Set the gains of the PD controller.
5. Run the file. Are the performance requirements satisfied in simulation ? If not adjust the gains of the PD controller.
6. Analyze the vertical speed command. Does it reach the saturation ?

3.4.6.3 Experimental test of the PD feedback controller on the TELLO drone

Warning. The time delay of about 0.3 s observed on the vertical velocity is neglected in the model. Since it is not negligible compared with T, a slower response and an overshoot of the order of 10 to 15 % can be expected on the real mini-drone.

1. Go to room C335
2. Set the PD controller gains in the beginning of the file and test the PD control implemented in Python on the TELLO mini-drone.
3. If everything goes well, a figure will be plotted showing the experimental response data that is also recorded in the file `DataCL_Zcontrol.txt`.
4. Copy the .txt file on a USB key or send it to you by e-mail.
5. Delete the .txt file.
6. Go back to the C311 room.
7. Open Matlab and execute the `dataCL_plot.m` file to get a plot of the closed-loop control performance.
8. Analyze the vertical speed command. Does it reach the saturation ?
9. Analyze the altitude tracking with the PD controller. Are the performance requirements satisfied in practice.
10. Is the PD control performance better than those of the simple P control ? You can use the IAE performance index to compare the P and PD controls.
11. If the results do not satisfy you, suggest a way to get better tracking results.

3.5 Troubleshooting

This section provides solutions to some issues you might encounter when using the TELLO drone during this lab.

3.5.1 Connection issues

If you try to connect to your drone via WIFI and it takes too long with the computer you are using, just run the Python code. If the battery level is printed, then your drone is connected to the computer.

Note that your drone will automatically switch off after 2 or 3 mn. You will need to switch it on and then connect again to the WIFI before executing the Python code.

3.5.2 Calibration issues

If the Python code returns an error message after printing the battery level, it either means that your drone battery is too low (under 10 or 20 %) or needs to be calibrated.
To calibrate your drone, you first need to download the TELLO app on your phone from this following link: `www.dji.com/uk/downloads/djiapp/tello`

Then, when you open the App, it will automatically try to connect your phone to a TELLO drone which will open your Wi-Fi interface and show you the available devices. Now, choose your drone. As it is not an internet device, your phone might ask you if you still want to connect to it. Confirm and when you are connected, go back until you see the App again.

Once you are connected to your drone and at the main interface of the App, you should see a gear on the top left side of your screen. Click on it then to "More" and to the "..." on the left side. The first option to appear should be "IMU Status". Click on "Calibrate" on the right side of this option and follow the instructions.

If you try to calibrate your drone and the App indicates that it has been calibrated without the need to place the drone in 6 different positions, close the App. Restart your drone and do the whole process again. If it is still the same, then it means your drone cannot be calibrated. Mark it and use another one.

In this calibration instructions, it is said that you should remove the propellers. There is a metallic and flat stick in the plastic bags inside the drone box. It is the Propeller Removal Tool they are talking about. The hollow part is to be used between the propeller and the motor to which it is attached.

Before removing them, mark that there is a pair with marks near the axis. The marked propellers should be at the top left and bottom right of the drone while the unmarked fill the other slots.

Note that this calibration is necessary for the Python code to be uploaded to the drone.

English to French glossary

bandwidth	:	bande passante
closed-loop system	:	système bouclé
crane	:	grue
cut-off frequency	:	fréquence (ou pulsation) de coupure
damped frequency	:	pulsation amortie
damping ratio	:	coefficient d'amortissement
drag	:	traînée
feedback	:	contre-réaction
feedback system	:	système à contre-réaction
gain	:	gain
heading angle	:	angle de cap
hoisting device	:	dispositif de levage
impulse response	:	réponse impulsionnelle
integral wind-up	:	emballement (de l'action) intégral
input	:	entrée
linear time-invariant (LTI)	:	linéaire invariant dans le temps
motor shaft	:	arbre moteur
output	:	sortie
overdamped	:	sur-amorti
overshoot	:	dépassement
pitch	:	tangage
rise time	:	temps de montée
road grade	:	inclinaison de la route
robot arm joint	:	articulation d'un bras de robot
roll	:	roulis
root locus	:	lieu des racines
setpoint	:	consigne
settling time	:	temps de réponse
steady-state gain	:	gain statique
steady-state response	:	réponse en régime permanent
steering	:	direction
step response	:	réponse indicielle
stream	:	courant
time-delay	:	retard pur
time-invariant	:	invariant dans le temps
transient response	:	réponse transitoire
throttle	:	accélérateur
undamped	:	non amorti
undamped natural frequency	:	pulsation propre non amortie
underdamped	:	sous-amorti
yaw	:	lacet