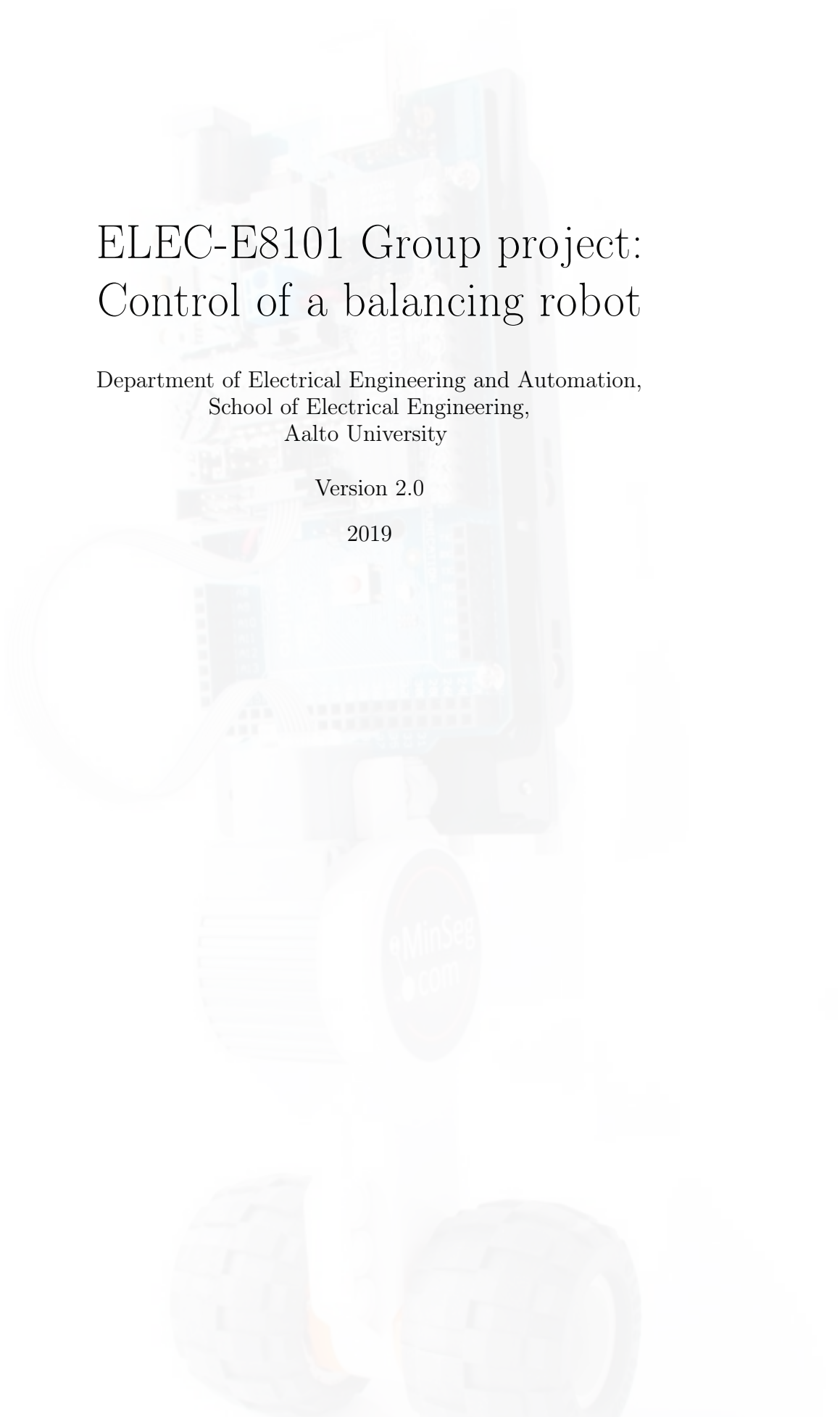
ELEC-E8101 Group project: Control of a balancing robot

Department of Electrical Engineering and Automation,
School of Electrical Engineering,
Aalto University

Version 2.0

2019



This project is based on the Labs of R7003E 2017 LP2 – Automatic Control in Luleå University. Special thanks to Prof. Damiano Varagnolo for sharing his material.

Contents

1 Introduction

1.1 What a balancing robot is and why it is an important system

A balancing robot is basically an inverted pendulum on wheels. If you see the picture in Figure 1, you immediately understand how this system works: the robot would naturally fall because of gravity, but if you accelerate the wheels properly you can make the robot stand up (or also turn around, if you have two independent wheels and motors). A balancing robot is complex enough to be fun to play with, but simple enough to be manageable in 3 labs.

The intended learning outcome is *not* just to make the balancing robot to stand. It is to learn how to design control schemes in the real world through using a simple and fun object. In practice, you will go through *some* of the fundamental steps that control engineers usually do in their job. During this project you will:

a) Do some preparatory tasks in Lab A. Here you will develop a SIMULINK simulator and do that theory that is needed before starting touching the real object[1].
b) Develop a first prototype of the control law in Lab B. Here you will test a Proportional-Integral-Derivative (PID) controller, do some experiments to validate the results of Lab A, some tuning of your simulator, and then prototype two more complex controllers.
c) Improve and validate the controller in Lab C and arrive at a complete scheme having those features that are usually needed in real world applications, like reference following blocks and structures for improving the robustness against uncertainties in the plant parameters.

Figure 1: An example of a two-wheeled balancing robot.

1.2 How we would like you to interact with your peers

Each group should be composed of *maximum* 4 persons.

If you need or absolutely want to be in 2 it is ok (you will work harder but you will also learn better). We strongly discourage you to be in 1, but you are allowed to do so if there are some particular needs. We delegate the process of creating the groups to you – in other words, try to self-organize. If somebody has some difficulty in finding a group let us know.

Please, help each other not only internally in the group but also among groups: discuss ideas and approaches, exchange information, compare results. Specially, brainstorm together and try new things. Do not cheat. First, we understand it. Second, let's be grown ups.

1.3 How we would like you to interact with us

Golden rules:

- feel free to ask for both help and clarifications;
- let us know if something in this manual is not clear / contradictory / incomplete;
- let us know if something does not work (balancing robot, computers, etc.);
- let us know your opinions and suggestions.

[1] This is a very common step: you first understand the properties of the plant before playing with it – eventually, would you wiggle a nuclear plant before seeing what happens in simulation?

2 Practical considerations

2.1 Hardware

Each group will be given a box with a MinSeg robot:

Figure 2: MinSeg robot for each group.

which you have to return back by the end of the project! In the box you will find a set of rechargeable batteries (1.2V instead of 1.5V, and lighter than normal batteries):

Figure 3: MinSeg robot for each group.

which you have to also return back by the end of the project! When the batteries drain, you can return them and get charged ones.

2.2 Installation in Windows

Follow the checklist:

1. install the Arduino IDE for Windows from https://www.arduino.cc/en/Main/Software;
2. install the Arduino support package for MATLAB as indicated in http://se.mathworks.com/help/supportpkg/arduino/ug/install-support-for-arduino-hardware.html;
3. download the Rensselaer SIMULINK library from http://homepages.rpi.edu/~hurstj2/;
4. follow the `readme` that you find in the Rensselaer library (notice that you may need to be administrator and also reboot the computer).

2.3 How to launch the code on the balancing robot

Follow the checklist:

1. check the COM port to which your robot is connected to (Windows button → Devices and printers);
2. in SIMULINK, "Code → C/C++ code → Deploy to hardware" (or, more simply, `ctrl+B`).

2.4 Datasheet

Table 1 summarizes the values of the parameters describing your balancing robot.

quantity	nominal value
g	9.8 m / s^2
b_f	0
m_b	0.463 Kg (with batteries)
l_b	0.113 m
I_b	0.00767 Kg m^2 (with batteries)
m_w	0.026 Kg
l_w	0.021 m
I_w	0.00000573 Kg m^2
R_m	4.4 Ohm
L_m	0
b_m	0
K_e	0.444 Volt sec. / radians
K_t	0.470 N m / amp.

Table 1: Notation used in this document (and in your reports).

2.5 Notation

symbol	meaning	unit
x_b	horizontal position of the center of mass of the body	m
y_b	vertical position of the center of mass of the body	m
θ_b	angular displacement of the body	radians
m_b	mass of the body	kg
l_b	wheel's center - body's center of mass distance	m
I_b	body's moment of inertia	kg m^2
x_w	horizontal position of the center of mass of the wheel	m
y_w	vertical position of the center of mass of the wheel	m
θ_w	angular displacement of the wheel	radians
m_w	mass of the wheel	kg
l_w	radius of the wheel	m
I_w	wheel's moment of inertia	kg m^2
t	time index for continuous time systems	sec.
k	time index for discrete time systems	adim.
Δ	sampling interval for discrete time systems	sec.
i_m	motor current	Amp.
v_m	motor voltage	Volt
R_m	motor electrical resistance	Ohm
L_m	motor electrical inductance	Henry
b_m	motor viscous coefficient	
T_m	motor torque	N m
T_f	friction torque	N m
$G(s)$	transfer function from v_m to θ_b	

Table 2: Notation used in this document (and in your reports).

3 Reporting your findings

3.1 How to complete the assigned tasks

The labs are divided into subsections. Each subsection in its turn contains tasks instructions, reading assignments and reporting instructions like the following ones:

Task 3.1

The **tasks** indicate what you are supposed to do in the project.

Reading 3.1

The **readings** suggest what you should know in order to perform the tasks.

Reporting 3.1

The **reportings** summarize what to report and how. Often you will be asked to report something in either *parametric* (you express the quantity as a function of some parameters) or *numeric* (you express the quantity with its numerical value) forms.

Notice that every task, reading or report is identified by an ID so that when we communicate among us we can be always on the same page of the same book.

Remark 3.1

It is recommended that every time that you use MATLAB or SIMULINK you **save the workspace** after you completed a task, giving to the saved workspace a name like `workspace_task_4.1_year_month_day_hour.mat`. In this way you will save all the information available that is needed to do the reporting comfortably at home (moreover you will not overwrite potentially important data). **And do backups of everything every time**.

Important: every group will have an ID number that is composed by **two** digits. In other words, the first group will have the ID number **01**, the second **02**, etc. This means that, for example, the SIMULINK file submitted by the first group is `group_01_Simulink.slx` (and **not** `group_1_Simulink.slx`!). Moreover, we are case sensitive, in the sense that for us `Group_01_Simulink.slx` is very different from `group_01_Simulink.slx`. Same for the reports.

3.2 The reports

The *lab reports* are very brief documents that you compile by group (thus one report for each group). We will use these documents not primarily to evaluate you, but rather to give you feedback on how you are doing. It should not take too much time to compile them (especially if you automate the production of the figures). For your help, follow this checklist:

1. download the corresponding template from MyCourses/Project/Report template;
2. **compile it**. Notice that what you are supposed to put inside the manuscript is not just a condensation of the three labs, but rather an exercise on how to write academic papers;
3. name the compiled template as `group_#MYGROUP_final_report.pdf`, with `#MYGROUP` being your group code;
4. upload it in MyCourses.

3.3 How you will be evaluated

In brief, each group needs to hand in:

1. the reports of the 3 labs;
2. the material associated to the demo.

Reports should comply with the given template. Once complete *and* complying with the requirements in the templates, reports will be evaluated by us and graded with a number ranging from 0% (worst possible) to 100% (best possible).

Every lab report will be evaluated independently following these performance metrics:

- **correctness of the results:** for a total of 50% of the points (0 = nothing is correct, 50 = no mistakes in the derivations, in the block schemes, etc.);
- **analysis of the results:** for a total of 50% of the points (0 = you analyzed nothing, 50 = you analyzed everything and correctly). Despite the lab reports will not require too much analysis, there will be some questions that challenge your knowledge. Your analysis is meant to be in this way: you obtain a specific result (for example, the PID controller behaves better in simulation than in reality). Then, *why do you think you got that specific result? What does it say to you? What are the implications?* Analyzing a result is a difficult task, and we want you to feel free to discuss what you find however you want. But the caveat is: *we want to understand if you understood, and not see what you did.* In other words, saying "we did this this and this and the result is shown in figure 7" is *not* an analysis. Instead, we prefer something like "we did this, and found that the system behaves in this way; initially we thought that it was because of this, but then we did this other experiment and found that. So we feel like this is happening, because of this this and this other reason". Notice that **it is completely OK** to say that you did not understand something / you do not know why something else is happening; but show us that you tried thinking at what you did, and not that you just monkey-made the labs.

4 Lab A

The main purpose of this lab is to develop a simulator of the balancing robot that will then be used in the next labs for rapid debugging and prototyping purposes.

4.1 Derivation of the Equations of Motion (EOM)

The Equations of Motion (EOM) are the heart of the simulation. Without them there is nothing to simulate. Furthermore, the EOM can be used for advanced filtering, for example in a Kalman filter.

The balancing robot should be modeled as in Figure 4 (check also the table for the notation on page 4): the body of the robot can be simplified as a thin pole with its mass m_b concentrated only in the center of mass of the robot, depicted as a larger dot in figure 2. The center of mass is at a distance l_b from the center of the wheel. The wheel has a radius of l_w, and mass of m_w.

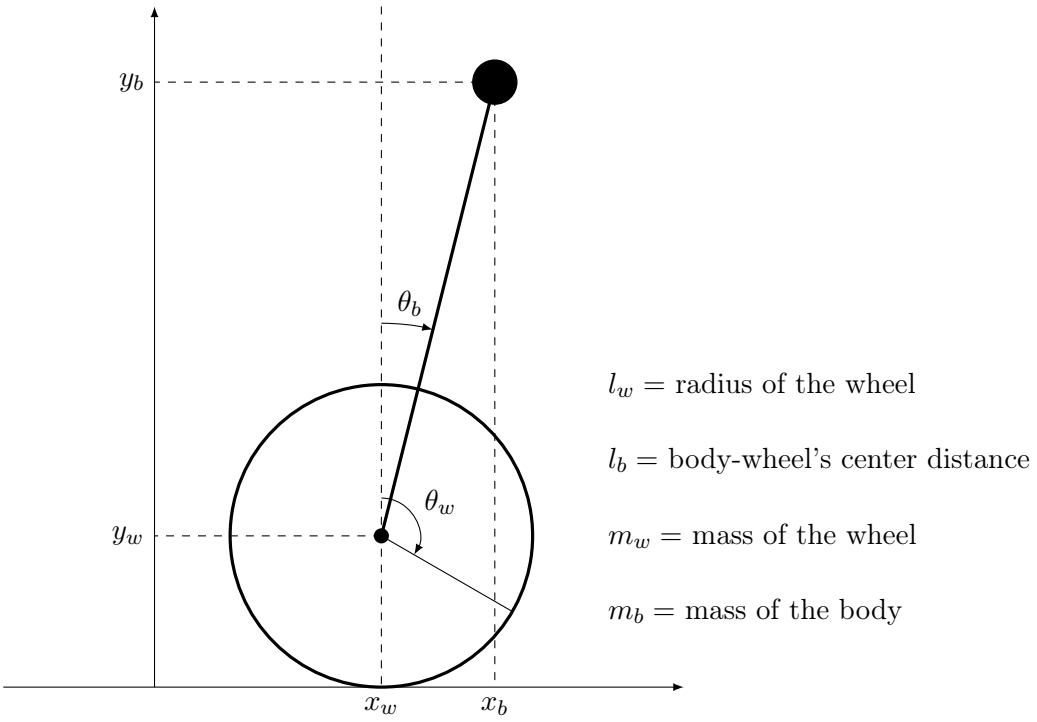
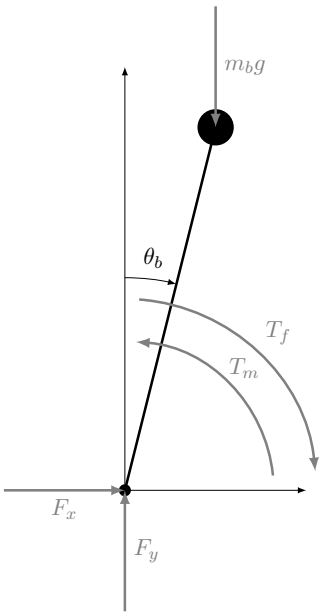


Figure 4: A simple schematic representation of a one-wheeled balancing robot where all the mass of robot (but the wheels') is concentrated in its center of mass.

The following assumptions should be made in order to simplify the problem:

1. The robot moves in a flat and horizontal environment, i.e., $\dot{y}_w = 0$ always.
2. The wheels never slip and the robot is never turned around by external factors, i.e., $x_w = l_w\theta_w$ always.
3. The aerodynamic frictions are negligible.
4. The inductance L_m and the motor viscous coefficient b_m are negligible.
5. The unique force that can be commanded is the torque applied by the motor to the wheel, and this torque is driven by the voltage that is applied to the motor.



F_y = vertical component of the tension with the wheel
F_x = horizontal component of the tension with the wheel
$m_b g$ = gravity for the body
T_m = motor torque
T_f = friction torque

Figure 5: Summary of the forces that apply to the body of the balancing robot.

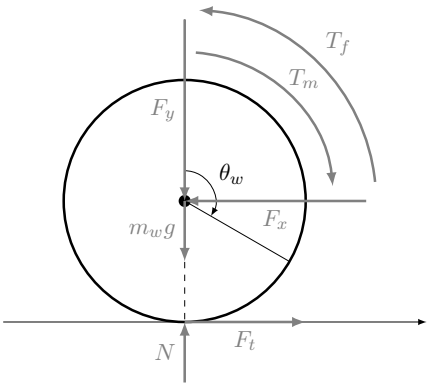
The Newton law for the linear movement of **the body** states that

$$m_b \ddot{x}_b = F_x \tag{1}$$
$$m_b \ddot{y}_b = F_y - m_b g \tag{2}$$

Notice that the gravity does not affect the x (horizontal) component since it is orthogonal to it. The Newton principle for the angular movement of the body (with rotational axis on the center of mass of the body) states that

$$I_b \ddot{\theta}_b = -T_m + T_f + F_y l_b \sin(\theta_b) - F_x l_b \cos(\theta_b) \tag{3}$$

for which the gravity does not affect the θ_b component since it does not lead to torque effects.



F_y = vertical component of the tension with the body
F_x = horizontal component of the tension with the body
$m_w g$ = gravity for the wheel
N = reaction of the plane
F_t = tractive force
T_m = motor torque
T_f = friction torque

Figure 6: Summary of the forces that apply to the wheel of the balancing robot.

The Newton law for the horizontal and vertical movements of the wheel are

$$m_w \ddot{x}_w = F_t - F_x \tag{4}$$
$$m_w \ddot{y}_w = N - m_w g - F_y = 0 \tag{5}$$

The Newton law for the angular movement of the wheel finally states that

$$I_w \ddot{\theta}_w = T_m - T_f - l_w F_t \tag{6}$$

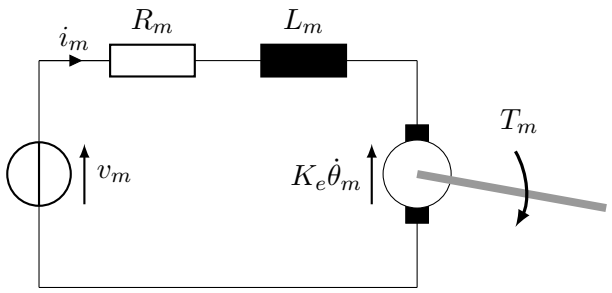


Figure 7: Schematic representation of a DC motor. Here, θ_m indicates the angle of the motor.

Analyzing the electrical circuit we get

$$L_m \frac{di_m}{dt} + R_m i_m = R_m i_m = v_m - e \tag{7}$$

where e is the back electromangetic force (emf), connected with the angular velocity of the motor through

$$e = K_e \left(\dot{\theta}_w - \dot{\theta}_b\right) = K_e \left(\frac{\dot{x}_w}{l_w} - \dot{\theta}_b\right) \tag{8}$$

where the first equality in (7) follows from the assumption that $L_m = 0$. Substituting (8) in (7) we then get

$$i_m = \frac{v_m}{R_m} - \frac{K_e}{R_m}\left(\frac{\dot{x}_w}{l_w} - \dot{\theta}_b\right) \tag{9}$$

Consider then that the torque T_m on the wheel's shaft induced by an armature's current equal to i_m is $T_m = K_t i_m$, with K_t the motor torque constant. Substituting in (9) we eventually obtain

$$T_m = \frac{K_t}{R_m} v_m - \frac{K_e K_t}{R_m}\left(\frac{\dot{x}_w}{l_w} - \dot{\theta}_b\right) \tag{10}$$

Since we have two motors, producing double the torque T_m, we replace the motor torque by

$$\hat{T}_m = 2T_m = \frac{2K_t}{R_m} v_m - \frac{2K_e K_t}{R_m}\left(\frac{\dot{x}_w}{l_w} - \dot{\theta}_b\right)$$

Remark 4.1

Be careful at the quantities that are considered in the various tasks: pay specially attention at the subscripts, and remember that w means "wheel", b means "body", m means "motor".

Remark 4.2

In order to rewrite F_t, F_x and F_y the Newton's laws for the linear movements of the wheel and of the body can be used. If everything goes as expected then there will be the need for simplifying the quantity "$\ddot{y}_b \sin(\theta_b) - \ddot{x}_b \cos(\theta_b)$". The following equivalences can be exploited for the simplification:

$$\begin{cases} x_b &= x_w + l_b \sin(\theta_b) \\ \dot{x}_b &= \dot{x}_w + \dot{\theta}_b l_b \cos(\theta_b) \\ \ddot{x}_b &= \ddot{x}_w + \ddot{\theta}_b l_b \cos(\theta_b) - \dot{\theta}_b^2 l_b \sin(\theta_b) \\ y_b &= y_w + l_b \cos(\theta_b) \\ \dot{y}_b &= \dot{y}_w - \dot{\theta}_b l_b \sin(\theta_b) = -\dot{\theta}_b l_b \sin(\theta_b) \\ \ddot{y}_b &= -\ddot{\theta}_b l_b \sin(\theta_b) - \dot{\theta}_b^2 l_b \cos(\theta_b) \end{cases} \tag{11}$$

Eliminating F_t, F_x and F_y from (3) and from (6) leads to two different equations of motion.

Equation 1: To eliminate F_x and F_y from (3) we can exploit (1) and (2) to obtain

$$I_b\ddot{\theta}_b = -\hat{T}_m + T_f + m_b l_b g \sin(\theta_b) + m_b l_b \ddot{y}_b \sin(\theta_b) - m_b l_b \ddot{x}_b \cos(\theta_b). \tag{12}$$

We don't like too much this expression, since it contains x_b and y_b terms. So we now aim to rewrite $\ddot{y}_b \sin(\theta_b) - \ddot{x}_b \cos(\theta_b)$ in a different way. Considering then Figure 4 on page 7, it follows that $\ddot{x}_b$ and $\ddot{y}_b$ are linked to $\ddot{x}_w$ and $\ddot{\theta}_b$ as in (11). Thus

$$\begin{aligned}
&+\ddot{y}_b \sin(\theta_b) - \ddot{x}_b \cos(\theta_b) = \\
&\left(-\ddot{\theta}_b l_b \sin(\theta_b) - \dot{\theta}_b^2 l_b \cos(\theta_b)\right) \sin(\theta_b) - \left(\ddot{x}_w + \ddot{\theta}_b l_b \cos(\theta_b) - \dot{\theta}_b l_b \sin(\theta_b)\right) \cos(\theta_b) = \\
&-\ddot{\theta}_b l_b \sin^2(\theta_b) - \dot{\theta}_b^2 l_b \cos(\theta_b)\sin(\theta_b) - \ddot{x}_w \cos(\theta_b) - \ddot{\theta}_b l_b \cos^2(\theta_b) + \dot{\theta}_b^2 l_b \sin(\theta_b)\cos(\theta_b) = \\
&-\ddot{\theta}_b l_b - \ddot{x}_w \cos(\theta_b)
\end{aligned} \tag{13}$$

Plugging into (12) we then obtain

$$I_b\ddot{\theta}_b = -\hat{T}_m + T_f + m_b l_b g \sin(\theta_b) - m_b l_b^2 \ddot{\theta}_b - m_b l_b \ddot{x}_w \cos(\theta_b) \tag{14}$$

and thus, rearranging,

$$\boxed{\left(I_b + m_b l_b^2\right)\ddot{\theta}_b = +m_b l_b g \sin(\theta_b) - m_b l_b \ddot{x}_w \cos(\theta_b) - \hat{T}_m + T_f} \tag{15}$$

Equation 2: Plugging (4) into (6), and using $\ddot{\theta}_w = \ddot{x}_w / l_w$ leads to

$$\frac{I_w}{l_w}\ddot{x}_w = \hat{T}_m - T_f - l_w F_x - l_w m_w \ddot{x}_w. \tag{16}$$

To eliminate F_x we then combine (1) and (11) into

$$F_x = m_b \ddot{x}_w + m_b l_b \ddot{\theta}_b \cos(\theta_b) - m_b l_b \dot{\theta}_b^2 \sin(\theta_b). \tag{17}$$

Plugging this into (16) we then obtain

$$\frac{I_w}{l_w}\ddot{x}_w = \hat{T}_m - T_f - l_w\left(m_b \ddot{x}_w + m_b l_b \ddot{\theta}_b \cos(\theta_b) - m_b l_b \dot{\theta}_b^2 \sin(\theta_b)\right) - l_w m_w \ddot{x}_w \tag{18}$$

and, rearranging,

$$\boxed{\left(\frac{I_w}{l_w} + l_w m_b + l_w m_w\right)\ddot{x}_w = -m_b l_b l_w \ddot{\theta}_b \cos(\theta_b) + m_b l_b l_w \dot{\theta}_b^2 \sin(\theta_b) - T_f + \hat{T}_m.} \tag{19}$$

4.2 Linearize the Equations of Motion

Since the goal is to design a controller for a linear system, the EOM must be linearized. The linearization point must be the equilibrium $\theta_b = 0$, so that

- $\sin(\theta_b) \approx \theta_b$;
- $\ddot{x}_w \cos(\theta_b) \approx \ddot{x}_w$;
- $\ddot{\theta}_b \cos(\theta_b) \approx \ddot{\theta}_b$.

As for $\dot{\theta}_b^2 \sin(\theta_b)$, the suggestion is to assume negligible centripetal forces (i.e., small body angle velocities), so that we can say $\dot{\theta}_b^2 \approx 0$. Thus, the linearized EOM are

$$\begin{cases}
\left(I_b + m_b l_b^2\right)\ddot{\theta}_b &= +m_b l_b g \theta_b - m_b l_b \ddot{x}_w - \dfrac{2K_t}{R_m} v_m + \left(\dfrac{2K_e K_t}{R_m} + b_f\right)\left(\dfrac{\dot{x}_w}{l_w} - \dot{\theta}_b\right) \\
\left(\dfrac{I_w}{l_w} + l_w m_b + l_w m_w\right)\ddot{x}_w &= -m_b l_b l_w \ddot{\theta}_b + \dfrac{2K_t}{R_m} v_m - \left(\dfrac{2K_e K_t}{R_m} + b_f\right)\left(\dfrac{\dot{x}_w}{l_w} - \dot{\theta}_b\right)
\end{cases} \tag{20}$$

4.3 Write the linearized EOM in state space (SS) form

We want to express the EOM in the form of a general linear time-invariant (LTI) SS system:

$$\begin{cases} \dot{\boldsymbol{x}} &= A\boldsymbol{x} + Bu \\ y &= C\boldsymbol{x} + Du \end{cases} \tag{21}$$

for state $\boldsymbol{x}$, and dynamics given by A, B, C and D. This means that you have to select the states for the system, find the equations for the first order derivatives of the states and stack everything up in a vector form like in equation (21). For now the input is the voltage applied to the motors and the measurement is the angular deviation of the balancing robot from the vertical upright position, i.e.,

$$\begin{aligned} u &= v_m \\ y &= \theta_b \end{aligned} \tag{22}$$

Task 4.1

Write the linearised EOM as in (21) considering the inputs and outputs as in (22).

Reporting 4.1

1. Say what is your choice for $\boldsymbol{x}$.
2. Write the matrices A, B, C and D in (21) in parametric form.
3. Write the matrices A, B, C and D in (21) in parametric numeric form (for this purpose use Table 1 on page 3).

Hint 4.1

It is much easier and numerically stable[a] to do as follows: write the system in parametric form as

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix} \begin{bmatrix} \ddot{x}_w \\ \ddot{\theta}_b \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \end{bmatrix} \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} u \tag{23}$$

where the various γ, α and β are functions of the various parameters in Table 1 (e.g., *and this is just an example*, $\alpha_{23} = I_b + m_r + R_m$); since

$$\begin{bmatrix} \ddot{x}_w \\ \ddot{\theta}_b \end{bmatrix} = \begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \end{bmatrix} \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} + \begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} u \tag{24}$$

you can create your A, B, C variables in MATLAB (values of the parameters given in the Datasheet in Table 1) immediately with a minimum of manipulation of the equations above (do all these computations in MATLAB! And try to learn how to use symbolic variables).

[a] In the sense that if you don't do as we suggest you may get very strange behaviors from MATLAB.

Troubleshoot 4.1

The values of your matrices should be as below. Notice that you should absolutely **NOT** manually set the matrices A and B in MATLAB as follows:

```
A = [ 0, 1,      0,    0 ;
   0,   -773.8,   -6.6 ,  16.2;
      0, 0,      0,    1 ;
 0,   3313.2,   63.1,   -69.6];
B = [ 0; 36.5980; 0; -156.7072];
```

This way of defining A and B induces very strange (and wrong) results. You have to compute A and B from the formulas; otherwise you destroy some symmetries that lead to poles / zeros cancellations and you get very wrong results. Use the above results, nonetheless, to check if you have computed the correct values.

4.4 Determine the transfer function of the LTI SS system

Task 4.2

Compute the transfer function, the poles and the zeros of the system.

Before using MATLAB, it is strongly recommended to start by getting an intuition of which poles and the zeros you will get by analyzing the structure of

$$sI - A \qquad \text{and} \qquad \begin{bmatrix} sI - A & -B \\ C & D \end{bmatrix}. \tag{25}$$

Only after this use MATLAB (and the commands in Table 3 on page 39) to compute the requested quantities both analytically and numerically. The rationale for this prior checking is this one: MATLAB (actually, any software) has numerical problems induced by the fact that they use finite arithmetic[2], and may thus make mistakes. You can then notice and correct them *only if* you developed intuitions of what you should get and what you should not.

Reporting 4.2

1. Write the transfer function in the form
$$G(s) = K \frac{\prod_i (s - z_i)}{\prod_j (s - p_j)} \tag{26}$$
2. If you experienced some numerical problem with MATLAB, describe them.

4.5 Design a PID controller stabilising the transfer function computed in Section 4.4

It is important to notice the following:

1. At this stage, only the transfer function from the motor voltage v_m to the body angle θ_b should be stabilized. The designed PID should only prevent the robot from falling over. Therefore, the reference signal for θ_b is zero.
2. The state for x_w should be ignored. It does not matter where the robot stands for now.

[2]Read, e.g., the paragraph "Zeros at Infinity" in http://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_Conversions.

3. The root locus of the linearized balancing robot indicates that there is no P controller stabilizing θ_b starting from v_m. A PI can stabilize the robot, but the stability domain is going to be very small. A PID instead works much better and gives a much bigger stability domain. Therefore, the PID version should be chosen as the first initial choice for a controller.

Task 4.3

Design the PID using the pole placement method. Since this is a simulation, you are free to choose the poles wherever you want (as long as the closed loop is stable, of course).

Reporting 4.3

1. Describe your choice for the poles, and what you want to achieve in terms of impulse response for the closed loop system.
2. Write down the values of the parameters of the PID.
3. Write down the resulting closed loop function in its numerical form.

4.6 Model the effect of disturbing the robot

It is often meaningful to analyze the effects of common disturbances on a system. In this case, it should be checked what happens when someone pokes the robot and applies an impulsive horizontal force to the center of mass of the body.

Task 4.4

Modify the EOM of your balancing robot to include an additional input that is called d and that models someone poking the robot. Derive also the linearized and state space version of the new EOM.

Reporting 4.4

1. Write down the new EOM in parametric form.
2. Write dowm the new linearized EOM in numerical state space form.

Hint 4.2

In other words, what happens if there is a term in Figure 5 that sums up with F_x?

The new disturbance can be thought as an additional input, but this new input d is different from the voltage v_m. Indeed, in the new state space representation the two columns of B are not linearly dependent, this means that d and v_m "enter" the state of the system with a "different angle" (think geometrically). In practice, poking the robot is different from disturbing the voltage v_m.

4.7 Check if everything is working as it should be

You should now have everything you need for checking if things are working as expected. The next task is to simulate what the linear system does when it is subject to a small poke.

Task 4.5

Implement a SIMULINK model for the linearized balancing robot subject to the two inputs v_m and d. Consider as outputs of the system all the states of the balancing robot, i.e., use as matrices C and D the matrices

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}. \tag{27}$$

Perform an experiment for which the external disturbance d is

$$d(t) = \begin{cases} 0.1 & \text{for } t \in [0, 0.1] \\ 0 & \text{otherwise,} \end{cases} \tag{28}$$

and check the values of the signals θ_b and x_w.

To create $d(t)$ you may use the SIMULINK block `Signal Builder`. For the linearised robot's dynamics it is instead suggested to use a `LTI System` that loads its `A`, `B`, `C` and `D` matrices from MATLAB's workspace. Be careful that here `B` includes not only the input v_m but also the input (disturbance) d, so update your definitions of C and D accordingly.

It is suggested to implement the PID through a `PID Controller` block. This block requires the `Filter coefficient (N)` to be set, and if you do not know what it does then check SIMULINK's help and try to think what happens if `N` is very big or very small.

Important: **It is a bad habit** to hard-code SIMULINK's blocks [3]. Moreover it is also suggested to automate the production of figures, e.g., as was demonstrated in Section A.2 on page 39.

Reporting 4.5

1. Plot your SIMULINK scheme.
2. Plot the realisations of $\theta_b(t)$, $x_w(t)$, $v_m(t)$ and $d(t)$.

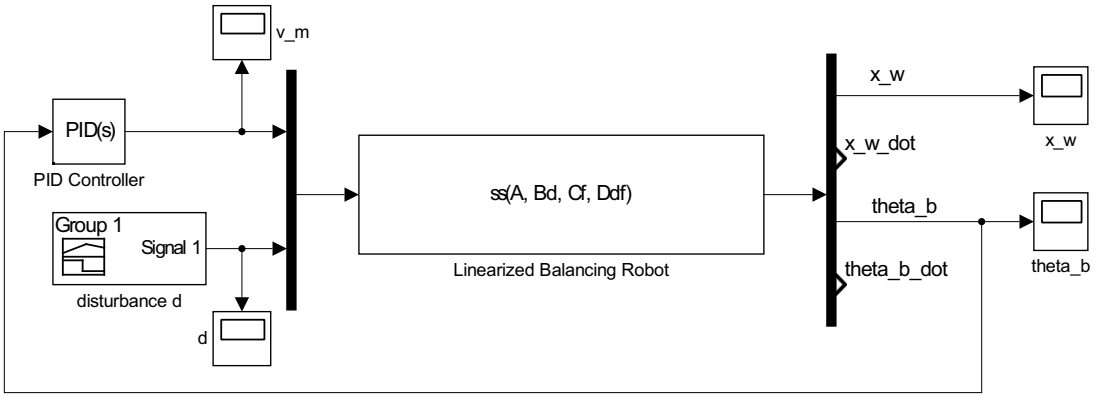


Figure 8: Our SIMULINK *solution for Task 4.7.*

The model solution is in Figures 8 (the SIMULINK diagram) and 9 (the plots of the signals). The solutions of the previous task are included because it is very important to understand what

[3] If you define the parameters within SIMULINK blocks, you need to find the block every time you need to change the value. This problem becomes more severe if you have multiple blocks using the same variables as all values need to be changed. Instead, define the block parameters by using symbols so that they will be loaded automatically from workspace.

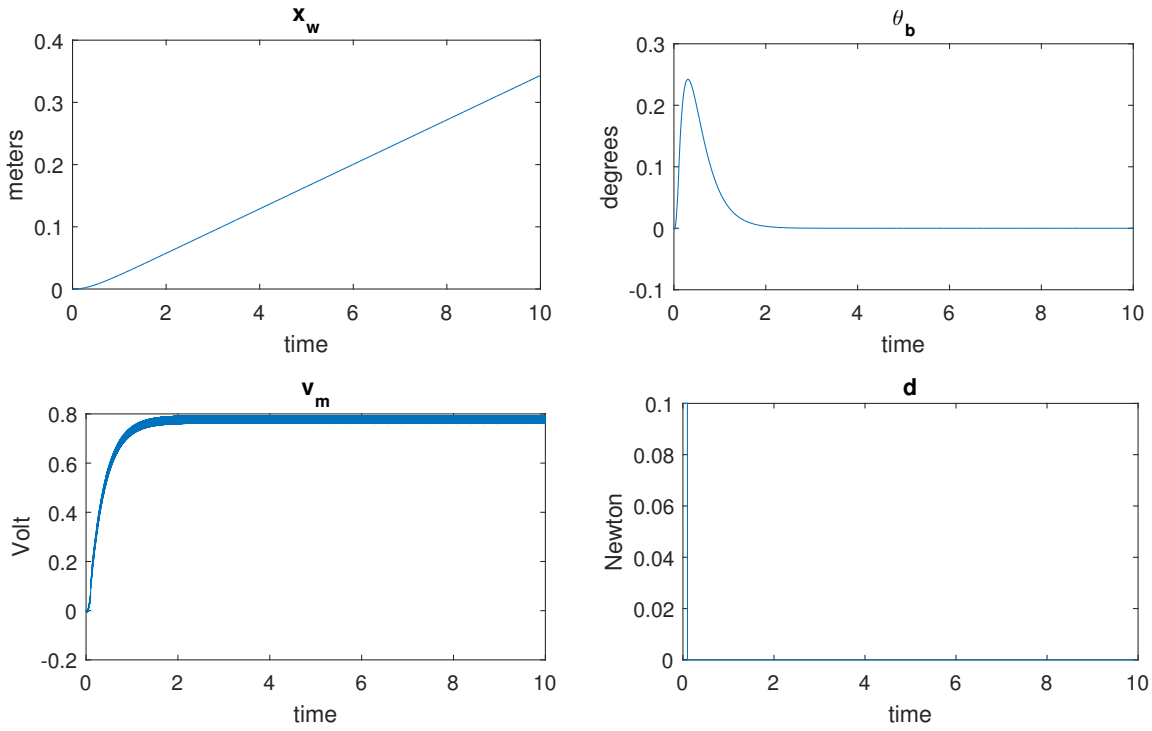


Figure 9: The signals that we obtained solving Task 4.7.

happens here: as you can notice, the system acts as an integrator on x_w. In other words, an initial poking makes the robot start moving. The controller then starts acting, but tries to regulate only θ_b: the control action indeed does not care if the state $\dot{x}_w$ is not zero. If you implement this PID on your robot, you should expect to run after it all the time. In mathematical terms, the system is actually still *unstable*.

The next question is: *Is it possible to stabilise x_b using an other PID?* The intuition is that if one could measure the wheel position x_w through some encoders then one may implement a cascaded controller that takes care of the state x_w as in Figure 10.

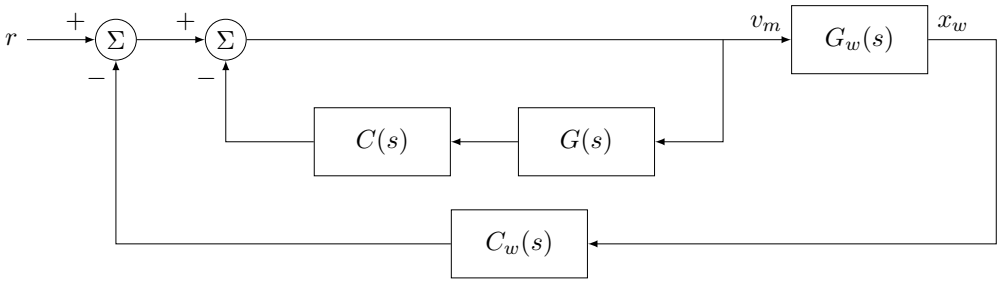


Figure 10: Alternative block schematic representation of the linearised balancing robot and the associated controllers.

The new block diagram in Figure 10 has $G(s)$ and $C(s)$ as before (i.e., the transfer functions from v_m to θ_b of Section 4.4 and the PID controller of Section 4.5), plus it has $G_w(s)$ and $C_w(s)$, respectively the transfer functions from v_m to x_w and an additional controller that needs to be designed. One could then design another PID controller and place the poles for the new transfer function $G_w(s)$. Unfortunately, $C_w(s)$ affects also $G(s)$, and $C(s)$ affects $G_w(s)$. As you may imagine, the problem becomes a bit more messy than before[4].

[4]The problem is that the new system is a multiple input multiple output (MIMO) system, since it has v_m and d as inputs, and θ_b and x_m as outputs. Controlling MIMO systems with PIDs require special techniques, which are not in the scope of this laboratory.

Therefore, this problem should be ignored for now and you should focus for now only on doing control for θ_b. A single PID controlling θ_b won't stabilize x_w. It is possible to add another PID for controlling also x_w, but the computations start becoming cumbersome. With state space notation this control problem can be solved in an easier way.

4.8 Convert the controller to the discrete time domain

The focus should now be shifted to the problem of implementing $C(s)$ with the real balancing robot. Before proceeding, notice that the Arduino board is a digital controller – in other words, you cannot implement $C(s)$ as it is: you must discretize it. There are several discretization methods. For now, the focus is on zero order hold (ZOH) methods.

Task 4.6

Compute the bandwidth of the linearised balancing robot, and decide the sampling period of the digital controller. Compute the controller in terms of *difference* equations.

Reporting 4.6

1. Report the bandwidth of the system.
2. Report the chosen sampling time.
3. Justify your choices in 1 and 2.
4. Report the discrete-time transfer function $C(z)$ of your controller in its parametric form.

4.9 Simulate the closed loop system

It is easier to prototype and validate controllers on simulators rather than on true systems. In the provided SIMULINK library you can find a block `BalancingRobotSimulator`, as the one in Figure 11.

The simulator of Figure 11 implements the non-linear EOM that you should have derived before; it thus *approximates* the behaviour of a real balancing robot with a good accuracy. The simulator allows you to:

- Play with two different disturbances: one on the mass of the body, so that you can mimic the effects of putting some additional weight on the robot, and one representing the torque disturbance d discussed above, so that you can mimic poking the robot.
- Get the whole state of the system $(x_w, \dot{x}_w, \theta_b, \dot{\theta}_b)$, so that you can try implementing full state controllers without having state observers, and test and characterize the performance of the state observers when you will implement them (more information in Lab B).

Notice that the simulated robot can fall, but – luckily – you cannot break it. So don't fear to experiment with it! To slow down/up your simulations (i.e., see the robot moving or not) set the block `Real Time Synchronization` accordingly.

Your goal is to check three things:

1. Is the controller stabilizing the non-linear dynamics?
2. Are the linearized dynamics a good approximation of the non-linear versions?
3. Is the chosen sampling time good for control purposes?

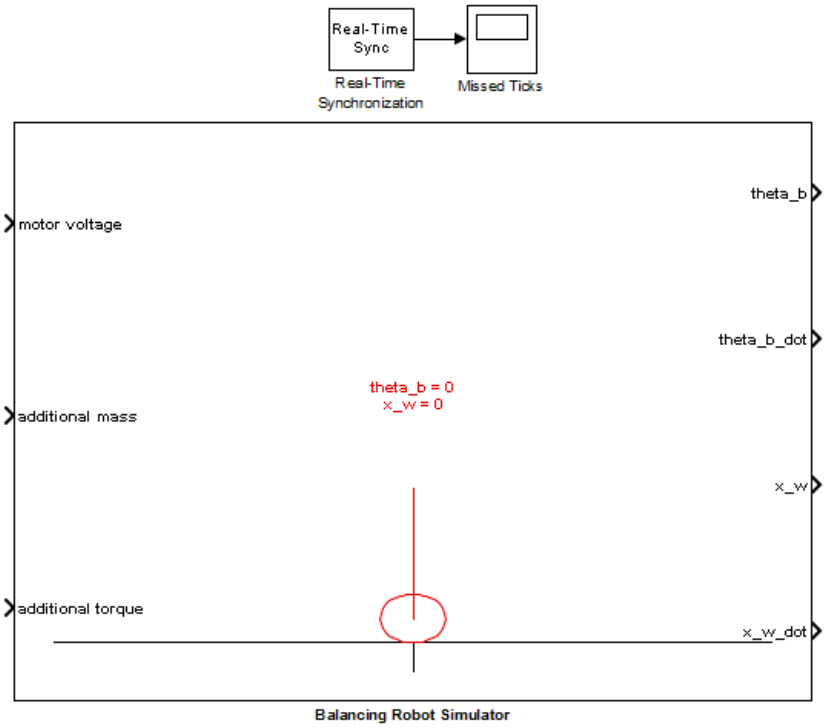


Figure 11: Our SIMULINK *balancing robot simulator. Double-clicking on it activates the mask in Figure 12, where you can set the various physical parameters describing your balancing robot.*

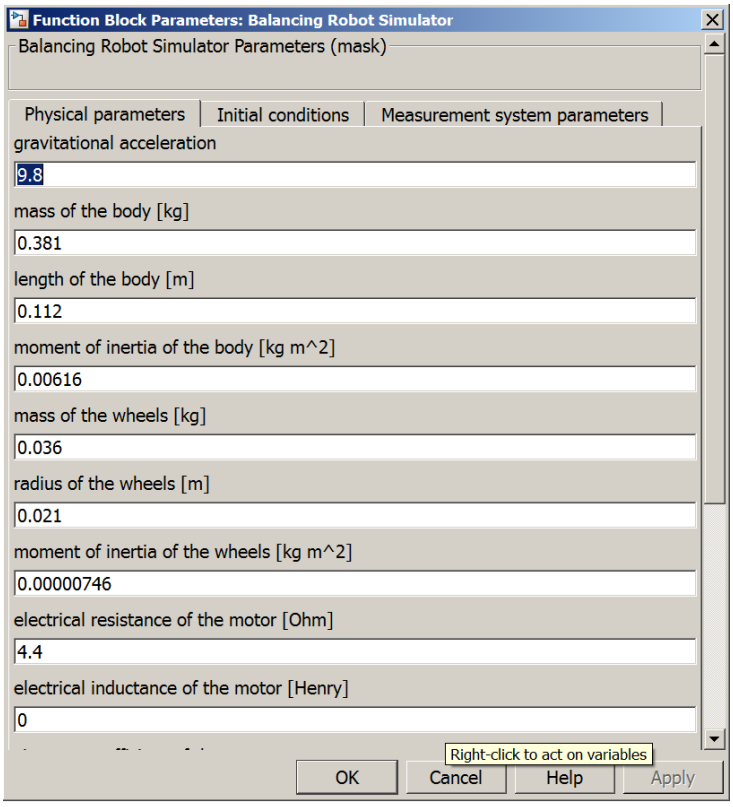


Figure 12: Mask for setting the parameters of the simulated balancing robot. The default values are the ones you should use in this lab.

Task 4.7

Using the SIMULINK block "Simulator" provided in the course website, build two SIMULINK schemes:

1. One as in Figure 13, where the simulated and linearised robots are controlled by the same continuous time PID $C(s)$ and subject to the same disturbance d.
2. One as in Figure 13, where the simulated and linearised robots are controlled by the same discrete time PID $C(z)$ and subject to the same disturbance d.

Detect for both the schemes the value of $\overline{d}$ in the disturbance signal

$$d(t) = \begin{cases} \overline{d} & \text{for } t \in [0, 0.1] \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

that makes the simulated robot fall when starting from the equilibrium. Compare the trajectories of θ_b, θ_b^{lin} and v_m for the continuous and discrete control cases by using a $\overline{d}$ in (29) with a value equal to half of the one that makes your simulated robot fall (one value for each case).

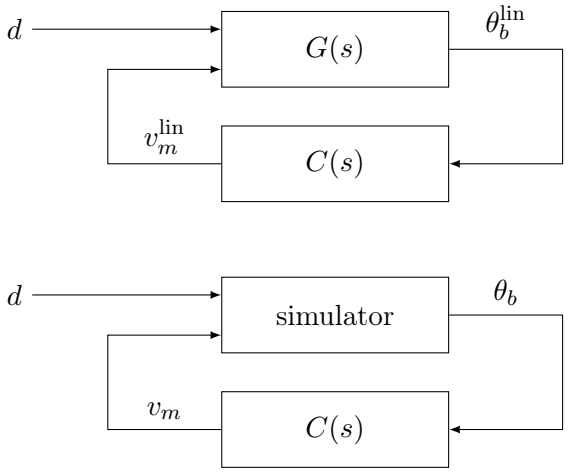


Figure 13: Block-schematic representation of how we compare the linearized dynamics with the nonlinear EOM. The disturbance d represents the same force corresponding to somebody poking the robot, while $C(s)$ is the same controller for both the linear dynamics $G(s)$ and the simulator.

IMPORTANT NOTICE! 4.1

Figure 13 represents the case of continuous time controllers. For the discrete case, the digital controller $G(z)$ should be surrounded by a sampling and a ZOH block. However, SIMULINK does this surrounding by itself. It is sufficient to change the PID block from "continuous" to "discrete" and let SIMULINK handle the conversion automatically.

Reporting 4.7

1. Report the smallest values of $\overline{d}$ that make your robot fall (one for the continuous and one for the discrete case).
2. Plot $\theta_b(t)$, $v_m(t)$, $\theta_b^{\text{lin}}(t)$ and $v_m^{\text{lin}}(t)$ (one for the continuous and one for the discrete case).
3. Discuss what you understood from these experiments, and if (and how) it helped you tuning the PID controller.

Troubleshoot 4.2

- Moving from continuous to discrete may introduce numerical problems, e.g., make a stable plant "explode"[a]. If this happens to you, then consider changing the "Discrete-time settings" of your PIDs (set, e.g., to "Backward Euler" or "Trapezoidal").
- When the SIMULINK says "Embedded Coder license missing" just press continue. The model should still work.

[a] It is because in this case there is a double integrator and numerical perturbations may be fatal for SIMULINK.

Notice that the results you got in this last step may be indicating that your controller is not good enough. Try playing with different gains, or equivalently, with different positions of the poles in the PID design step, and check how things change.

4.10 Optional: play with the simulator

This is something you are not compelled to do, but this of course is a very useful exercise. It is suggested to play around with the parameters in the simulator (i.e., make the robot heavier, put bigger wheels, lighter wheels, more powerful motors, etc.) and see if your intuitions work. In other words, try to think "what is going to happen if I do this?" and double check with experiments if you have understood how the system works. If you find something that is worth to be mentioned in the lab report, you should of course mention it.

5 Lab B

The main purpose of this lab is to start designing controllers using *continuous time* concepts. In other words, you do not start immediately from discrete considerations but rather from continuous ones, and then discretize the results.

5.1 How to do tests on the robot

There are two different ways of running tests:

- One is *External mode* (`ctrl+T` from SIMULINK), for which the robot is "commanded" by SIMULINK. In this case the robot runs up to the moment that the SIMULINK simulation finishes, or up to when you press the reset button (Figure 15). When you do this kind of tests, after the end of the simulation the robot will usually do nothing. If the motors have input other than 0 at the end of the simulation, there they will continue running as the last value is held after the communication stops. A simple press of the reset button will fix this.
- The other is *Deploy to hardware* (`ctrl+B` from SIMULINK), for which the robot is loaded with some code that will continuously run whenever the board is powered. In this situation you turn off the robot either temporarily by pressing the reset button (but when you release the robot will re-initialize its computations and start again), or by removing the power to the board (i.e., by disconnecting the USB cable and turning off the switch for the batteries highlighted in Figure 14).

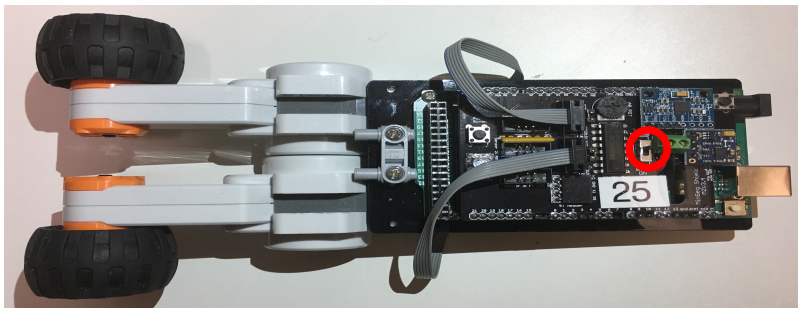


Figure 14: Batteries switch.

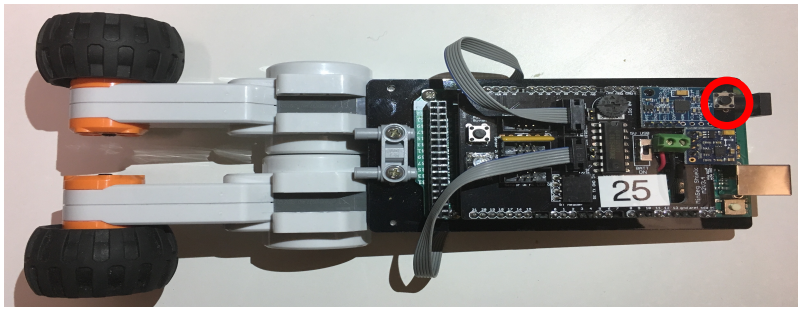


Figure 15: The reset button. When an "External" SIMULINK simulation finishes the robot remains in the last state. To stop the robot or reset the state, press the highlighted button.

Troubleshoot 5.1

In case you find some problems **please report them as soon as possible** so that they can be added to the following list:

- When SIMULINK says "access denied" when downloading some code to the robot, *sometimes* the solution is to delete the folder `NameOfTheSimulinkDiagram_rtt`. This folder contains some temporary code that SIMULINK creates for deploying the code to the hardware and it will be recreated automatically.
- If the build diagnosis report states, there was no board connected, the best option is to try to build the code again. The models do not seem to accept setting of the communication port manually.

5.2 Communicating with the balancing robot

You should start by checking if you are able to communicate with the robot. Please, follow these steps:

1. Download the code from MyCourses folder "MATLAB and SIMULINK code" for the various labs.
2. Either copy the file `C:\MATLAB\RASPLib\startup.m` to your `H:\Documents\MATLAB\startup.m` or, if you want, create `H:\Documents\MATLAB\startup.m` manually and add the following lines to it:

   ```
   addpath('C:\MATLAB\RASPlib\RASPlib')
   addpath('C:\MATLAB\RASPlib\RASPlib\src')
   addpath('C:\MATLAB\RASPlib\RASPlib\include')
   addpath('C:\MATLAB\RASPlib\RASPlib\examples')
   addpath('C:\MATLAB\RASPlib\RASPlib\blocks')
   ```

   Notice also that it is `H:\Documents` and not `H:\My Documents`: these can be different folders and MATLAB may not look into both for the startup script. Additionally, notice that Unix machines require / symbols instead of \ in the path names.
3. Open MATLAB.
4. Connect the robot and the computer with the USB cable (no need for plugging the batteries for now). MATLAB should say that an Arduino 2560 has been connected.
5. Double click the SIMULINK file `LabB_CheckCommunications.slx` and wait for SIMULINK to open.
6. Click the text "load parameters".
7. Launch a simulation in external mode (`ctrl+T`).
8. Stop manually the simulation whenever you want. Notice that after stopping SIMULINK you should also reset the robot by pressing the reset button.
9. Now you can check if you got the data from the robot in two different ways[5]:

[5]It is good practice to visualize with SIMULINK is for debugging purposes and to visualize with MATLAB for reporting. SIMULINK is able to provide live data and MATLAB offers good automation and modification of the layout. Of course you are free to do as you wish.

(a) Double-clicking on the scopes in the SIMULINK diagram (you can actually see the data on-line, TIP: open the scopes before running the code).

(b) Plot the signals in MATLAB using, e.g., the following code:

```
figure(1)
plot( squeeze(tAngleFromGyro.time), squeeze(tAngleFromGyro.signals.values), '-b' );
axis([0, max(tAngleFromGyro.time), 100, 190]);
legend('raw signal');
title('measurements from the gyroscope'); xlabel('time (sec)'); ylabel('degrees');
set(gcf, 'Units', 'centimeters');
set(gcf, 'Position', [1 1 11 9]);
set(gcf, 'PaperPositionMode', 'auto');
print('-depsc2', '-r300', 'raw_angle_from_gyro.eps');
```

Notice that you should have 4 signals in the workspace: the accelerometer, the gyro, the encoder and the motor, and all of them should contain some information.

If you get meaningful plots then you successfully communicated with the robot.

5.3 Observing that reality is far from ideality

If you successfully communicated with the robot you probably noticed that the behaviours of the accelerometer and of the gyro are very far from the ideal case. For example, the measurements from a robot lying down and still on a table probably looked like in Figure 16. *Notice that the figure from the gyroscope plots the derivative of the signal, and not the signal itself!*

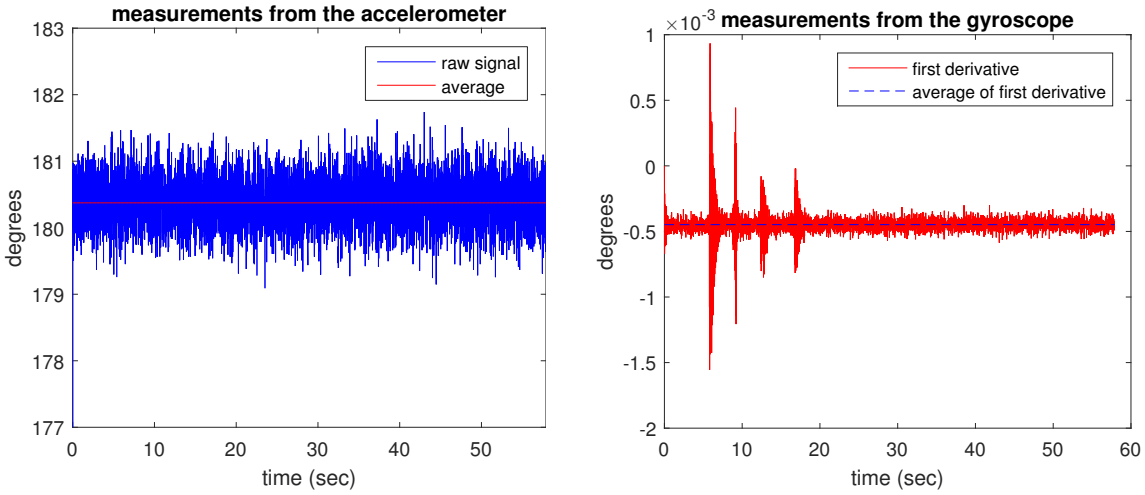


Figure 16: Examples of raw data coming from the gyroscope and the accelerometer mounted on our balancing robots.

Troubleshoot 5.2

Does the graph of the measurements from your gyroscope differ from the graph above? Did you also read the legend of the graph?

As you may suspect, using the measurements from the gyro and the accelerometer needs some filtering. The approach in this lab is to tune the bias by launching the MATLAB script `LabB_TuneTheGyro_SaveParameters.m` from MATLAB's shell (see what is in, so that you understand what it does – notice also that from now on we will assume that steps 1 - 6 of the previous procedure are done by default).

IMPORTANT NOTICE! 5.1

During the execution of this code you have to keep the robot standing still and upwards, like it should be when perfectly balancing itself.

In other words with this procedure (that lasts some 20 / 30 seconds) you are "showing" to the robot how it should stand when it will balance by itself. The result will be a variable `fGyroBias` saved in the `.mat` file `GyroBias.mat`. *Be precise and still while doing this calibration; this is a crucial step.*

5.4 Test the PID controller

Now, you should test if the developed PID controller behaves as one would wish in the real system. To do so:

1. Mount the batteries on the robot (you may want to use some tape to prevent them from falling if the robot falls over).
2. Turn the batteries switch on (see Figure 14);
3. Connect the USB cable;
4. Open the SIMULINK diagram `LabB_PIDOverRobot.slx`;
5. open the various sub-diagrams and explore how we implemented the various things;
6. open the MATLAB script `LabB_PIDOverRobot_Parameters.m`, and put in it your gains k_P, k_I and k_D plus your sampling period (you will find in the .m file our default values). **Do not touch the other variables**;
7. launch `LabB_PIDOverRobot_Parameters.m` to load these parameters (you can do it directly in SIMULINK by clicking on the `load parameters` text);
8. launch a "Deploy to hardware" simulation (`Ctrl+B`). If you completed the steps in Section 5.2 successfully you should have no problems here;
9. if the robot does not balance, try to change the PID coefficients as your intuition suggests;
10. to visualize or save data from the robot follow this procedure:
    (a) open the MATLAB script `GetDataFromSerial.m`;
    (b) edit the variable `iCOMPort` accordingly to your setup (that means, check to which COM port the robot is connected to by pressing the Windows key and checking "Devices and Printers");
    (c) the variable `fSamplingPeriod` should already be in the workspace because it should be set by the MATLAB script `LabB_PIDOverRobot_Parameters.m`;
    (d) modify (if you want) the variables:
        - `iCommunicationTime`, that dictates the duration of the experiment;
        - `fPlotsUpdatesPeriod`, that dictates how often the plots will be refreshed;
    (e) launch `GetDataFromSerial.m`. *Notice that this will cause the robot to re-start*;
    (f) at the end of the experiment you will get the information from the robot saved in the workspace in the variable `aafProcessedInformation` (open `PlotDataFromSerial.m` and look at the code inside to understand what is what).

We expect your PID not to perform too well (e.g., ours was standing for maximum about 10 seconds). So, do not lose too much time in wiggling with the PID controller.

Task 5.1

Test your PID controller as above.

Reporting 5.1

Report an example of the x_w, θ_b and u obtained with your best PID controller, and the parameters of the PID. Indicate also how long you got the robot stand before falling for this controller.

Troubleshoot 5.3

- when closing inopportunely the serial communications you may get problems in re-establishing the connections again. E.g., you may get an error of the kind:

  ```
  Opening the serial communications...Error using serial/fopen (line 72)
  Open failed: Port: COM7 is not available. Available ports: COM3, COM4,
  COM5, COM10. Use INSTRFIND to determine if other instrument objects
  are connected to the requested device.
  ```

  In this case you may solve the problem closing MATLAB, unplugging the USB cable of the robot from the computer side, restarting MATLAB and then reconnecting the robot.

5.5 Check the controllability and observability properties of the linearized system (21)

Task 5.2

Compute and discuss the controllability and observability properties of the linearized system (21) in numeric form.

Reporting 5.2

1. Write the numerical values for $\mathcal{C}$ and $\mathcal{O}$;
2. if something is not controllable or observable:
   (a) motivate why this happens;
   (b) say how you would fix the problem;
   (c) say how this affects the transfer function of the system.

5.6 Design a SS controller selecting the poles using a second order approximation

We now start the process of designing the first continuous time state-space oriented controller. The steps will be the following:

1. pretending that we have information on all the state, construct a controller by doing poles allocation;
2. since we do not measure the full state of the system, and we still do not have constructed some state observers, we must test the controllers in simulation;
3. after checking that the controllers behave as expected, we construct some observers of the state, and test also these ones in simulation;
4. when the simulations of both the controller and the observers indicate that in theory things are working, we implement the findings in the real system.

Before starting, ask yourself: relative to the models developed in Lab A, what are the inputs that you can actually command? (That means: which linearized equations shall you consider?) After thinking at this we thus continue by constructing a controller through a second order approximation of the plant, and then check its performance in simulation. As for the poles location, feel free to do some tests, but keep in mind the practical considerations.

Task 5.3

1. select the location of the poles for the closed loop system by choosing a proper dominant second order system (hint: the two poles of the open loop system that must be moved are the unstable one and the integrator. The other two stable poles are instead one very fast, relative to the motor, and one quite slower – around -5. The speed of that slower pole is a good starting point...);
2. add to the MATLAB script `LabB_ControllerOverSimulator_Continuous_Parameters.m` that MATLAB code that computes the gains matrix K. The results of your computation must be saved in the variable `K`;
3. open the SIMULINK diagram `LabB_ControllerOverSimulator_Continuous.slx` and test your controller through an "external mode" simulation (`ctrl+T`). At the end of the simulation the SIMULINK diagram will save the variables `x` and `u` in MATLAB's workspace, so that you can plot the results conveniently.

Reporting 5.3

Report:

1. the poles that you selected for the closed loop system, motivating briefly why you chose that ones, and the corresponding second order approximating system that you chose;
2. how you computed the gains matrix K, and its value;
3. the plots of θ_b and v_m obtained with the chosen controller.

5.7 Design a state observer, and add it to the simulator

The next step is to design a continuous time strategy for observing the state and test it on the simulator. We test two different strategies:

1. a full order Luenberger observer;
2. a reduced order Luenberger observer.

Both the observers are based on the assumption that we do not measure just θ_b, but also x_w, so that the C matrix of the system is now

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \Longrightarrow \qquad \begin{bmatrix} y_{x_w} \\ y_{\theta_b} \end{bmatrix} = C \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} \tag{30}$$

The tasks are the following ones:

Task 5.4

Design:

1. a full-order Luenberger observer assuming C as in (30). Use directly a poles allocation method;
2. a reduced-order state Luenberger observer assuming C as in (30), and assuming that the measurement y_{x_w} is accurate, while y_{θ_b} is not. Use again a poles allocation method.

To solve the task above edit the SIMULINK diagram `LabB_ObserverOverSimulator_Continuous.slx`. In it you find:

- the simulator;
- two already implemented block schemes of the requested observers;
- some code for visualizing the interesting signals and saving them to the workspace.

What you need to do is then to implement in `LabB_ObserverOverSimulator_Continuous_Parameters.m` some MATLAB code that computes the following quantities in the following variables[6] (or, alternatively, load these variables from the workspace if you saved them in the previous tasks):

- `kP, kI, kD` (the parameters of your continuous time PID controller);
- `A, B, C, D` (the system matrices – notice that `C` should be the C in (30) while `B` should be the B relative to just the input v_m, and thus be one single column);
- `L` (the full order observer gain);
- `M1, M2, M3, M4, M5, M6, M7` (the reduced order observer gains).

Troubleshoot 5.4

If MATLAB complains when you are using `acker` and says that A', C' may be non-controllable (that means that A, C is non-observable) then ask yourself: did you use a C that guarantees observability? (Because, as for sure you remember, `acker` works when you have full controllability/observability...)

[6] Once again, use *exactly* the notation you see below, since the SIMULINK diagram loads them from the workspace. E.g., if you do not put `kP = -50;` in the `.m` file then SIMULINK will give an error.

As for the reduced observer gains, we recall that the equations are the following ones: starting from a system $(A, B, C, 0)$, you rewrite it as

$$\begin{cases} \dot{\boldsymbol{x}} & = A\boldsymbol{x} + Bu \\ \boldsymbol{y}_{\text{acc}} & = C_{\text{acc}}\boldsymbol{x} \\ \boldsymbol{y}_{\overline{\text{acc}}} & = C_{\overline{\text{acc}}}\boldsymbol{x} \end{cases}$$

with C_{acc} in $\mathbb{R}^{p\times n}$ with p the number of accurate measurements. Select then a basis completion V so that $T^{-1} = \begin{bmatrix} C_{\text{acc}} \\ V \end{bmatrix}$ is a proper change of basis, and define $\boldsymbol{x} = T\boldsymbol{z}$ so that

$$\boldsymbol{z} = T^{-1}\boldsymbol{x} = \begin{bmatrix} C_{\text{acc}}\boldsymbol{x} \\ V\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ V\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix}.$$

This induces the new system

$$\begin{cases} \dot{\boldsymbol{z}} & = \widetilde{A}\boldsymbol{z} + \widetilde{B}u \\ \boldsymbol{y}_{\text{acc}} & = \widetilde{C}_{\text{acc}}\boldsymbol{z} \\ \boldsymbol{y}_{\overline{\text{acc}}} & = \widetilde{C}_{\overline{\text{acc}}}\boldsymbol{z} \end{cases} \qquad \text{with} \qquad \begin{cases} \widetilde{A} & = T^{-1}AT \\ \widetilde{B} & = T^{-1}B \\ \widetilde{C}_{\text{acc}} & = C_{\text{acc}}T \\ \widetilde{C}_{\overline{\text{acc}}} & = C_{\overline{\text{acc}}}T \end{cases}$$

with

$$\widetilde{C}_{\text{acc}} = C_{\text{acc}}T = C_{\text{acc}}\begin{bmatrix} C_{\text{acc}} \\ V \end{bmatrix}^{-1} = \begin{bmatrix} I_p & 0_{p\times n-p} \end{bmatrix}$$

and

$$\boldsymbol{z} = T^{-1}\boldsymbol{x} = \begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ V\boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix} \qquad \widetilde{C}_{\text{acc}} = \begin{bmatrix} I & 0 \end{bmatrix} \qquad \widetilde{C}_{\overline{\text{acc}}} = \begin{bmatrix} \widetilde{C}_y & \widetilde{C}_\chi \end{bmatrix}$$

This then implies that the new system can be rewritten as

$$\begin{cases} \begin{bmatrix} \dot{\boldsymbol{y}}_{\text{acc}} \\ \dot{\boldsymbol{\chi}} \end{bmatrix} & = \begin{bmatrix} \widetilde{A}_{yy} & \widetilde{A}_{y\chi} \\ \widetilde{A}_{\chi y} & \widetilde{A}_{\chi\chi} \end{bmatrix}\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix} + \begin{bmatrix} \widetilde{B}_y \\ \widetilde{B}_\chi \end{bmatrix} u \\ \boldsymbol{y}_{\text{acc}} & = \begin{bmatrix} I & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix} \\ \boldsymbol{y}_{\overline{\text{acc}}} & = \begin{bmatrix} \widetilde{C}_y & \widetilde{C}_\chi \end{bmatrix}\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix}, \end{cases}$$

an expression that highlights how the first and third subsystems contain partial information, while the second subsystem is instead non informative. Consider then the reduced subsystem

$$\begin{cases} \begin{bmatrix} \dot{\boldsymbol{y}}_{\text{acc}} \\ \dot{\boldsymbol{\chi}} \end{bmatrix} & = \begin{bmatrix} \widetilde{A}_{yy} & \widetilde{A}_{y\chi} \\ \widetilde{A}_{\chi y} & \widetilde{A}_{\chi\chi} \end{bmatrix}\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix} + \begin{bmatrix} \widetilde{B}_y \\ \widetilde{B}_\chi \end{bmatrix} u \\ \boldsymbol{y}_{\overline{\text{acc}}} & = \begin{bmatrix} \widetilde{C}_y & \widetilde{C}_\chi \end{bmatrix}\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \boldsymbol{\chi} \end{bmatrix} \end{cases}$$

or, equivalently,

$$\begin{cases} \dot{\boldsymbol{y}}_{\text{acc}} & = \widetilde{A}_{yy}\boldsymbol{y}_{\text{acc}} + \widetilde{A}_{y\chi}\boldsymbol{\chi} + \widetilde{B}_y u \\ \dot{\boldsymbol{\chi}} & = \widetilde{A}_{\chi y}\boldsymbol{y}_{\text{acc}} + \widetilde{A}_{\chi\chi}\boldsymbol{\chi} + \widetilde{B}_\chi u \\ \boldsymbol{y}_{\overline{\text{acc}}} & = \widetilde{C}_y\boldsymbol{y}_{\text{acc}} + \widetilde{C}_\chi\boldsymbol{\chi} \end{cases}$$

or, again equivalently, as

$$\begin{cases} \dot{\boldsymbol{\chi}} & = \widetilde{A}_{\chi\chi}\boldsymbol{\chi} + \left(\widetilde{A}_{\chi y}\boldsymbol{y}_{\text{acc}} + \widetilde{B}_{\chi}u\right) \\ \begin{bmatrix} \dot{\boldsymbol{y}}_{\text{acc}} - \widetilde{A}_{yy}\boldsymbol{y}_{\text{acc}} - \widetilde{B}_{y}u \\ \boldsymbol{y}_{\overline{\text{acc}}} - \widetilde{C}_{y}\boldsymbol{y}_{\text{acc}} \end{bmatrix} & = \begin{bmatrix} \widetilde{A}_{y\chi} \\ \widetilde{C}_{\chi} \end{bmatrix} \boldsymbol{\chi} \end{cases}$$

This implies that we moved from the full order observer structure

$$\dot{\widehat{\boldsymbol{x}}} = A\widehat{\boldsymbol{x}} + Bu + L\left(\boldsymbol{y} - C\widehat{\boldsymbol{x}}\right),$$

where we design L through `L = ( place( A', C', afPoles ) )'`, to a reduced order observer structure

$$\dot{\widehat{\boldsymbol{\chi}}} = \widetilde{A}_{\chi\chi}\widehat{\boldsymbol{\chi}} + \left(\widetilde{A}_{\chi y}\boldsymbol{y}_{\text{acc}} + \widetilde{B}_{\chi}u\right) + L\left(\begin{bmatrix} \dot{\boldsymbol{y}}_{\text{acc}} - \widetilde{A}_{yy}\boldsymbol{y}_{\text{acc}} - \widetilde{B}_{y}u \\ \boldsymbol{y}_{\overline{\text{acc}}} - \widetilde{C}_{y}\boldsymbol{y}_{\text{acc}} \end{bmatrix} - \begin{bmatrix} \widetilde{A}_{y\chi} \\ \widetilde{C}_{\chi} \end{bmatrix} \widehat{\boldsymbol{\chi}}\right)$$

for which we design L through an opportune `L = ( place( AA', CC', afPoles ) )';` where `AA` and `CC` are the new system matrices. Notice that after the design step one can split the new L in two parts, i.e.,

$$L = \begin{bmatrix} L_{\text{acc}} & L_{\overline{\text{acc}}} \end{bmatrix}$$

so that we can rewrite the expressions as

$$\begin{aligned} \dot{\widehat{\boldsymbol{\chi}}} = & \left(\widetilde{A}_{\chi\chi} - L_{\text{acc}}\widetilde{A}_{y\chi} - L_{\overline{\text{acc}}}\widetilde{C}_{\chi}\right)\widehat{\boldsymbol{\chi}} \\ & + \left(\widetilde{B}_{\chi} - L_{\text{acc}}\widetilde{B}_{y}\right)u \\ & + \left(\widetilde{A}_{\chi y} - L_{\text{acc}}\widetilde{A}_{yy} - L_{\overline{\text{acc}}}\widetilde{C}_{y}\right)\boldsymbol{y}_{\text{acc}} \\ & + L_{\overline{\text{acc}}}\boldsymbol{y}_{\overline{\text{acc}}} + L_{\text{acc}}\dot{\boldsymbol{y}}_{\text{acc}}. \end{aligned}$$

Since the term $\dot{\boldsymbol{y}}_{\text{acc}}$ may lead to numerical problems, we introduce the new state $\widehat{\boldsymbol{\chi}} = \widehat{\boldsymbol{\chi}}' + L_{\text{acc}}\boldsymbol{y}_{\text{acc}}$. Moreover we also need to changes the coordinates to go back to the original space (indeed, $\boldsymbol{x} = T\boldsymbol{z}$, thus $\boldsymbol{z} = T^{-1}\boldsymbol{x}$). We thus can eventually rewrite the dynamics of the observer as

$$\begin{aligned} \dot{\widehat{\boldsymbol{\chi}}}' = & \left(\widetilde{A}_{\chi\chi} - L_{\text{acc}}\widetilde{A}_{y\chi} - L_{\overline{\text{acc}}}\widetilde{C}_{\chi}\right)\widehat{\boldsymbol{\chi}} \\ & + \left(\widetilde{B}_{\chi} - L_{\text{acc}}\widetilde{B}_{y}\right)u \\ & + \left(\widetilde{A}_{\chi y} - L_{\text{acc}}\widetilde{A}_{yy} - L_{\overline{\text{acc}}}\widetilde{C}_{y}\right)\boldsymbol{y}_{\text{acc}} \\ & + L_{\overline{\text{acc}}}\boldsymbol{y}_{\overline{\text{acc}}} \end{aligned} \qquad \widehat{\boldsymbol{\chi}} = \widehat{\boldsymbol{\chi}}' + L_{\text{acc}}\boldsymbol{y}_{\text{acc}} \qquad \widehat{\boldsymbol{x}} = T\begin{bmatrix} \boldsymbol{y}_{\text{acc}} \\ \widehat{\boldsymbol{\chi}} \end{bmatrix}$$

or, in a compact form, as

- $\dot{\widehat{\boldsymbol{\chi}}}' = M_1\widehat{\boldsymbol{\chi}} + M_2u + M_3\boldsymbol{y}_{\text{acc}} + M_4\boldsymbol{y}_{\overline{\text{acc}}}$
- $\widehat{\boldsymbol{\chi}} = \widehat{\boldsymbol{\chi}}' + M_5\boldsymbol{y}_{\text{acc}}$
- $\widehat{\boldsymbol{x}} = M_6\boldsymbol{y}_{\text{acc}} + M_7\widehat{\boldsymbol{\chi}}$.

Reporting 5.4

Report:

1. the values of L (the gain of the full order estimator) and $M_1, \ldots, M_7$ (the gains of the reduced order estimator);
2. plots of θ_b, x_w and their estimates from both the full and reduced order estimators;
3. the maximum error committed in estimating θ_b and x_w with the full and with the reduced order observers (thus $\max_t \left|\theta_b(t) - \widehat{\theta}_b^{\text{full}}(t)\right|$ and $\max_t \left|\theta_b(t) - \widehat{\theta}_b^{\text{reduced}}(t)\right|$, and the same quantities for x_w).

5.8 Discretize both the controller and observer, and add them to the simulator

Before implementing the controller and observer in the Arduino board, we must convert our continuous time strategies into a discrete one and test that we have meaningful results. To this aim we must perform the following task:

Task 5.5

1. discretize the continuous time system $(A, B, C, 0)$ into its discrete equivalent $(A_d, B_d, C_d, 0)$ (you can easily do this in MATLAB through the function `c2d`. Notice that in the MATLAB scripts you will always find the variable `fSamplingPeriod` – change its value if you want but **keep this name!**);
2. map the poles for the controller and observer found for the continuous time case through the map $z = e^{p\Delta}$, where Δ is the sampling period;
3. compute the gains K_d, L_d, $M_{d1}, \ldots, M_{d7}$ for the discrete controller and full / reduced observers using again the functions `acker` or `place` (notice that these gains will be different since we are in a discrete time case);
4. check that in simulation things still function properly.

To solve the task above follow this procedure:

1. start adding to the MATLAB script `LabB_ControllerOverSimulator_Discrete_Parameters.m` the value for K_d, stored in the variable `Kd`. After this, simulate the SIMULINK diagram `LabB_ControllerOverSimulator_Discrete.slx`;
2. continue with `LabB_ObserverOverSimulator_Discrete_Parameters.m`, where you should:
   - compute the variables `Ad`, `Bd`, `Cd` and `Dd` relative to the discretized version of the original system (A, B, C, D);
   - compute the variable `Ld` relative to the gain of the full order discrete observer;
   - compute the variables `Md1`, ..., `Md7` relative to the gains of the reduced order discrete observer;
   - load the gains of your PID controller, i.e., `kP`, `kI` and `kD`.

   After this simulate the SIMULINK diagram `LabB_ObserverOverSimulator_Discrete.slx`;

3. end with `LabB_ObserverAndControllerOverSimulator_Discrete_Parameters.m`, where you should put all the parameters that you computed in the previous two steps. After this, simulate the SIMULINK diagram `LabB_ObserverAndControllerOverSimulator_Discrete.slx`.

Important: remember that your actuators saturate at 9 Volts. Thus if you detect that your simulated system has a u that is "too big" then you may have problems in the real device...

Reporting 5.5

Report:

1. how you derived (A_d, B_d, C_d, D_d) and their values;
2. how you derived K_d, L_d, $M_{d1}, \ldots, M_{d7}$ and their values;
3. a plot of x_w, θ_b and u for the complete control system (controller plus observer).

5.9 Test the control strategy with the real robot

We are now ready to test the control strategy on the real robot.

Task 5.6

1. edit the MATLAB script `LabB_ObserverAndControllerOverRobot_Parameters.m` by adding in it all the variables that you have computed in `LabB_ObserverAndControllerOverSimulator_Discrete_Parameters.m`.
2. open the SIMULINK diagram `LabB_ObserverAndControllerOverRobot.slx`, and explore the diagrams (i.e., how we implemented things). Take special a look at the controller and at the observer, and notice how there are actually 3 different observers:
   (a) a full order Luenberger observer;
   (b) a reduced order Luenberger observer (might not work on the robot; if not, just report it; we are aware of the issue);
   (c) a numerical observer, that works by considering all the measurements as accurate, along their numerical derivatives.

   It is possible to select which observer you want to use by clicking on the switches;
3. launch different "deploy to hardware" simulations (`ctrl+B`), and see for which observer you get the best performance (up to now we instructors got the best performance with the numerical observer). Of course you can always re-tune the various gains of the controller and observer in case you are not satisfied with the system – but it is better if you do tests on the simulators, before trying on the real robot;
4. to visualize or save the data from the robot follow the same procedure described in Section 5.4.

Reporting 5.6

Report an example of the x_w, θ_b and u obtained with your best controller, and which type of observer you used.

Troubleshoot 5.5: "My robot does stand upright, but it walks away; it doesn't go back to the original place!"

This may be caused by the drift of the gyro. For example, check Figure 17. θ_b is estimated to increase with time, and the robot compensates by actuating opportunely the wheels. The solutions to this problem are two:

- re-tune the gyro, as in Section 5.3;
- manually adjust the variable `fGyroBias` contained in `GyroBias.mat`, for example by adding to it the average bias measured in a run where the robot was not disturbed by external factors but let walk as it wanted to (something that can be computed using the MATLAB command:

  ```
  mean(diff(aafProcessedInformation(MEASURED_THETA_B_INDEX,:))/fSamplingPeriod)
  ```

  after having successfully recorded data from the serial port).

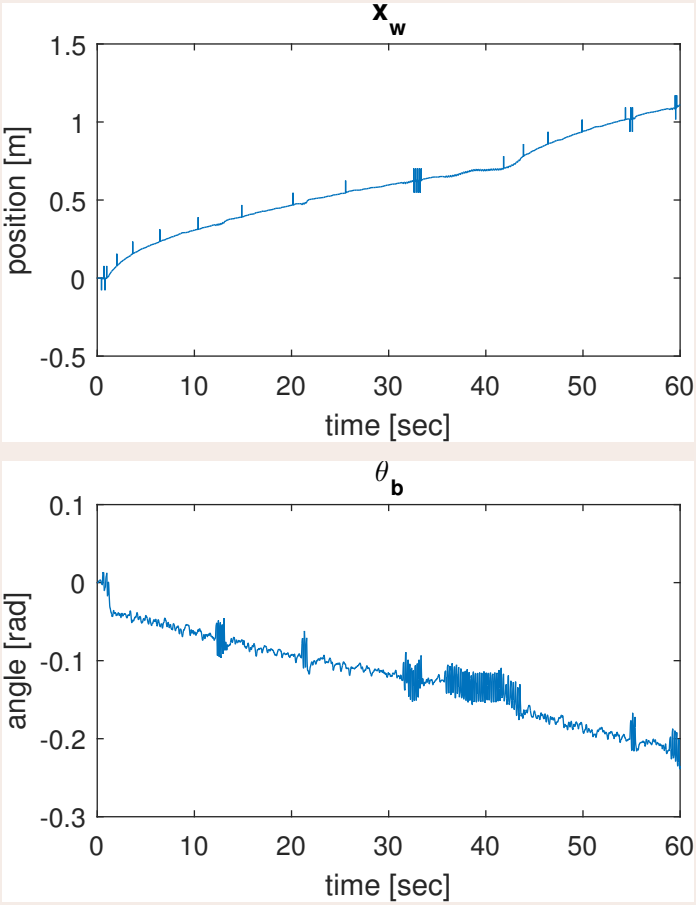


Figure 17: Example of the effect of a poorly tuned gyro: the system estimates an angle that increases / decreases with time, and compensates it by moving the wheels accordingly. The total outcome is that the robot "walks away".

6 Lab C

The main purpose of this lab is to re-design the control scheme using directly discrete time concepts, and add to the control scheme the management of the reference signal.

6.1 Compute the discrete equivalent of the original model

The *first* part of the following task is equal to Task 5.9, thus you have basically already solved it. The second part you will basically solve at the end of the lab. Here we will indeed change quite often the used sampling frequency, and test several of them up to the moment where we find "the best one" (in a sense described better later). You will thus be able to complete the second part of this task only at the end of the lab.

Task 6.1

1. Discretize the continuous time (A, B, C, D) linear model of the robot and derive its discretized version (A_d, B_d, C_d, D_d) using the sampling period used in Task 5.9;
2. find the "best sampling frequency".

Reporting 6.1

Report:

1. the "best sampling frequency" that you found at the end of the lab;
2. the numerical values for (A_d, B_d, C_d, D_d) relative to this sampling period.

6.2 Design a SS controller selecting the poles using the LQR technique

It is important to notice that the LQR formalism is defined over a system for which the output y can be "anything one likes", i.e., we are not necessarily required to use the actual $y = C\boldsymbol{x}$ defined by the system. In other words, we can pretend that our y is anything we want, so that eventually we compute the K that minimizes the continuous time performance index

$$\mathcal{J} = \int_0^{+\infty} \left(\rho \begin{bmatrix} x_w & \dot{x}_w & \theta_b & \dot{\theta}_b \end{bmatrix} \overline{C}^T \overline{C} \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} + u^2 \right) dt \tag{31}$$

where $\overline{C}$ indicates our weighting attitude towards different states. $\overline{C}$ may be, for example,

$$\overline{C} = \begin{bmatrix} 5 & 1 & 10 & 2 \end{bmatrix} \tag{32}$$

so that we are twice less happy of having $\theta_b = 1$ rather than having $x_w = 1$, and five times less happy to have $x_w = 1$ rather than having $\dot{x}_w = 1$. *In other words, $\overline{C}$ is a design parameter that you have to choose!* Choosing a specific $\overline{C}$ then basically means choosing to consider an alternative fictitious system for which $y_\star = \overline{C}\boldsymbol{x}$. Considering this fictitious system allows us to find a K that is optimal for a certain user-defined function of $\boldsymbol{x}$ (were we are using the actual $\boldsymbol{y}$, we would indeed instead have had to consider a cost function that may not reflect our preferences in how we want to weight the various components of $\boldsymbol{x}$). In (31), ρ is the relative tuning parameter between having large controls control efforts (u^2) and having a state $\boldsymbol{x}$ far from the origin (i.e. where we in this application would want it to be).

Task 6.2

1. choose $\overline{C}$ in (32);
2. draw the Symmetric Root Locus[a] (SRL) for the obtained system;
3. select ρ in (31);
4. compute the LQR gain K minimizing the cost (31);
5. implement and test the new controller as we did in Task 5.6 (comment out what's not needed anymore in `LabB_ControllerOverSimulator_Continuous_Parameters.m`);
6. compare the current control performance against the ones obtained using the pole placement scheme. Based on the simulations, decide which controller you want to use in the subsequent tasks.

[a] In case you do not know about Symmetric Root Locus, read, e.g., http://web.mit.edu/16.31/www/Fall06/1631_topic15.pdf.

Troubleshoot 6.1

If you keep getting a strange root locus, it may be because you manually set the matrices A and B, e.g., did as follows in MATLAB:

```
A = [ 0, 1,      0,    0 ;
      0, -435.0, -6.1, 9.1 ;
      0, 0,      0,    1 ;
      0, 1903.4, 62.0, -40.0 ];
B = [ 0; 20.6; 0; -90.0];
```

This way of defining A and B induces very strange (and wrong) results. You have to compute A and B from the formulas; otherwise, you destroy some symmetries that lead to zero-pole cancellations and you may get very wrong results.

Reporting 6.2

Report:

1. your choices for $\overline{C}$ and[a] ρ, with some brief justifications;
2. the SRL, and how you computed it (by the way, make the drawing show what happens around the origin; what happens around -500 is not so interesting);
3. how you computed K, and its values;
4. the plots of θ_b and v_m obtained with the chosen controller;
5. which pole placement strategy you want to use in the forthcoming tasks, and why.

[a] It is meaningful to draw the root locus with the MATLAB command `rlocus` and to select the gain ρ by using the "Data cursor" functionality in the figure plotted by MATLAB.

6.3 Design the controller using the discrete LQR technique

This section mimics Section 6.2; here, indeed we assume to know the matrix W, indicating how we weight undesirable states, and compute the K_d that minimizes the discrete time performance index

$$\mathcal{J} = \sum_{k=1}^{+\infty} \left(\rho \begin{bmatrix} x_w(k) & \dot{x}_w(k) & \theta_b(k) & \dot{\theta}_b(k) \end{bmatrix} W \begin{bmatrix} x_w(k) \\ \dot{x}_w(k) \\ \theta_b(k) \\ \dot{\theta}_b(k) \end{bmatrix} + u^2(k) \right) \tag{33}$$

Once again, both W and ρ are design parameters that you can choose. Since you already solved this selection in Section 6.2, here we use the same choices as before. Notice that, as before, you will probably re-do this task several times, one for each sampling period that you will test.

Task 6.3

1. start from a sampling frequency of 200Hz, and compute (A_d, B_d, C_d, D_d) as in Section 6.1
2. compute the LQR gain K_d minimizing the cost (33) by using the MATLAB function `dlqr`;
3. insert in the MATLAB script `LabB_ControllerOverSimulator_Discrete_Parameters.m` the new value for K_d, stored in variable `Kd`, and simulate once again the SIMULINK diagram `LabB_ControllerOverSimulator_Discrete.slx` (i.e., do as you did in Task 5.8 for testing the controller);
4. if the controller still stabilizes the simulator, reduce the sampling frequency and start over, up to the moment for which the simulated robot falls.

Reporting 6.3

Report:

1. the sampling frequency for which you lose stability;
2. the corresponding values for K_d and (A_d, B_d, C_d, D_d).

6.4 Design the observer starting from the previously constructed controller

Now that K_d is computed, we compute both the full order and the reduced order observers L_d and $M_{d1}, \ldots, M_{d7}$. To this aim we directly choose the pole locations wanting the observers to be from 2 to 6 times faster than the controller poles. Importantly, **notice that the concept "faster" is now referred in the discrete-time domain**. To clarify, in continuous time a pole located in p_s is 5 times slower than a pole located in $5p_s$. In other words, in continuous time multiplying times χ implies χ times faster. In discrete time multiplying times χ does not mean χ times faster. For example, if a pole is located in $p_z = 0.5$ (stable), then multiplying times 3 leads to something that is unstable.

To understand how to make a pole faster in the discrete-time domain, consider the following hint: we know that $p_z = e^{p_s \Delta}$, so that

$$p_z = e^{p_s \Delta} \quad \Longrightarrow \quad p_s = \frac{\ln(p_z)}{\Delta}. \tag{34}$$

At the same time, if p'_z is χ times faster than p_z it means that

$$\begin{aligned} p'_z = e^{p'_s \Delta} = e^{\chi p_s \Delta} \quad &\Longrightarrow \quad \chi p_s = \frac{\ln(p'_z)}{\Delta} \quad \Longrightarrow \quad \chi \frac{\ln(p_z)}{\Delta} = \frac{\ln(p'_z)}{\Delta} \\ &\Longrightarrow \quad \ln(p_z^\chi) = \ln\left(p'_z\right) \quad \Longrightarrow \quad p'_z = p_z^\chi. \end{aligned} \tag{35}$$

From this it is immediate how to compute p'_z as a function of p_z and χ.

Task 6.4

1. start again from a sampling frequency of 200Hz, and compute (A_d, B_d, C_d, D_d) as in Section 6.1
2. choose χ as you did in Task 5.7, and compute the gains K_d, L_d, $M_{d1}, \ldots, M_{d7}$ for the discrete controller and full / reduced observers (for the reduced order observer you can re-use the equations you used in Task 5.8);
3. insert in the MATLAB script:

   `LabB_ObserverAndControllerOverSimulator_Discrete_Parameters.m`

   the parameters of the previous controller K_d (corresponding to the right sampling time!) and of the current observers and simulate the SIMULINK diagram `LabB_ObserverAndControllerOverSimulator_Discrete.slx`;
4. as before, if the controller still stabilizes the simulator, reduce the sampling frequency and start over, up to the moment for which the simulated robot falls.

Reporting 6.4

Report:

1. the sampling frequency for which you lose stability;
2. a (brief) discussion on why the critical sampling frequency found now is different from that one found in Task 6.3;
3. the corresponding values for K_d, L_d, $M_{1d}, \ldots, M_{7d}$ and (A_d, B_d, C_d, D_d).

6.5 Perform experiments on the real balancing robot

You have now computed all the information that you already computed in Task 5.8, only following a different paradigm: instead of starting from a continuous time controller, finding the continuous time poles p_s, finding the discrete ones through the $p_z = e^{p_s \Delta}$ transformation, and then doing `acker` on them, you designed directly the p_z (and thus an other set of K_d, L_d, and $M_{d1}, \ldots, M_{d7}$) using discrete time considerations. Here, we then repeat some experiments that are similar to the ones performed in Task 5.9, with the aim of checking how different sampling frequencies affect the real balancing robot.

Task 6.5

1. Start from the critical sampling frequency that you computed in Task 6.4;
2. repeat the same tests you performed in Task 5.9 with the new controllers / observers (use the same kind of observer that you used in that task). If the robot does not stabilize, increase the sampling frequency and recompute the various controllers / observers up to the moment that the robot stabilizes. Save the traces of the x_w and θ_b obtained for that frequency (tip: use the serial communications tool, and save the workspace once you have meaningful result);
3. now increase the sampling frequency 20Hz per time up to the moment you reach 200Hz; for each of these frequencies re-do a balancing experiment, and save each time the signals x_w and θ_b that you obtain.

Reporting 6.5

Report:

1. the first sampling frequency for which your real robot balances;
2. a graph reporting how the L_2 norms[a] of the signals x_w and θ_b depend on the sampling frequency of the system. Be careful to do not forget that the signals from different experiments: *a)* have different sampling periods; *b)* may have different durations.

[a]The L_2 norm calculates the distance of the vector coordinate from the origin of the vector space. As such, it is also known as the Euclidean norm as it is calculated as the Euclidean distance from the origin. The result is a positive distance value.

6.6 Design a module for managing external references

Among the different possible strategies for managing reference signals, here we choose to implement the ones for which a step in the reference does not excite the state observer. This means that the control configuration is as in Figure 18.

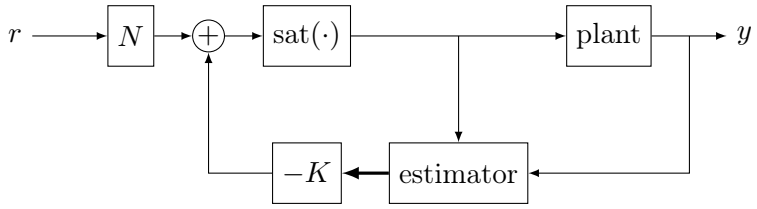


Figure 18: Chosen way of introducing the reference input.

The problem is now to compute the value for N that ensures that the DC gain from r to y is equal to 1. Important: the C that should be considered here is $C = [1\ 0\ 0\ 0]$, i.e., we should consider the C that corresponds to measuring x_w. Indeed, here we are interested in a reference for changing the x-position of the robot, and not for changing its θ_b.

Task 6.6

1. derive the formula for N that guarantees the DC gain from r to x_w to be 1 for discrete time settings;
2. compute N and put its value in the MATLAB script

   `LabC_CompensatorOverRobot_Parameters.m`,

   in the variable `Nud` (keep the variable `Nxd` and put it equal to zero). Notice that this script should also load all the variables computed for solving the tasks above;
3. open the SIMULINK diagram `LabC_CompensatorOverRobot.slx`, and explore the block `reference` inside the block `controller`. Here you may change your reference signal as you prefer.

Reporting 6.6

Report:

1. the formula you used to compute N and the value of N;
2. the plots of x_w, θ_b, v_m and r for some experiments with different reference signals. In other words, create 2 / 3 different reference signals, and for each of them report how the robots behaves;
3. (optional) compare if your robot behaves better / worse than the robots of your friends.

Notice that the nastier the reference signal is, the more likely your robot is to fall. You may consider once again to change the sampling frequency of the whole controller and see if for certain references signals a certain sampling frequency is not fast enough.

We can thus define as the best sampling frequency the lowest sampling frequency that makes your robot follow sufficiently safely the nastier reference signal that you expect it to be required to follow.

Notice that the definition is deliberately fishy. To define precisely this frequency you must define what is the nastier reference for you. Once you defined it, you can do experiments and see what is the best frequency for you. And once you did this step, you can eventually complete Task 6.1.

6.7 Final demo

The demo, that can be performed only after you finished the previous tasks of Lab C, evaluates numerically the performance of your controller. This evaluation has then two different purposes: (a) for us, to evaluate your results; (b) for you, to be motivated to do your best.

By this point, your balancing robot will be able to follow external references and will be robust against uncertainties. This means that we can test *your* controller on *our* robot, and see how well your controller performs on our system. What we need is then just your `.mat` and `.slx` files, so that we can upload the code on our balancing robot. In practice, the demo will consist of two steps:

1. you perform (and report) some experiments on *your* balancing robot with *your* controller;
2. we perform the same experiments on *our* balancing robot with *your* controller, and check that your report is meaningful.

For your help, follow this checklist:

1. do the following experiments with your balancing robot:
   (a) apply the reference signal (t in seconds, $r(t)$ in meters):
   $$r(t) = \begin{cases} 0 & t \in [0, 10) \\ 0.05(t-10) & t \in [10, 20) \\ 0.5 & t \in [20, 30] \end{cases} \quad (36)$$
   (b) apply the reference signal (t in seconds, $r(t)$ in meters):
   $$r(t) = \begin{cases} 0 & t \in [0, 10) \\ r_{\max}(t-10) & t \in [10, 20) \\ 10 r_{\max} & t \in [20, 30] \end{cases} \quad (37)$$
   where $r_{\max}$ is a factor proportional to the maximal speed that your balancing robot can follow without falling (find it by trial and error);
2. register the results and report them in the final report as indicated in the demo template;
3. when submitting the final report, submit also:
   (a) your SIMULINK simulator (i.e., your `LabC_CompensatorOverRobot.slx` and `LabC_CompensatorOverRobot_Parameters.m` files, named `group_#MYGROUP_Simulink.slx` and `group_#MYGROUP_Matlab.m` respectively, and where `#MYGROUP` is your group ID, see Section 3). We will indeed take your code and reproduce the experiments you did by our own;
   (b) your results as a `.mat` file named `group_#MYGROUP_results.mat` and containing the following variables (only these ones!):
   - `group_#MYGROUP_experiment_1_times`, a *column* vector containing the sampling instants of the data relative to $r(t)$ as in Equation (36) (in seconds, and with 0 indicating the beginning of the experiment);
   - `group_#MYGROUP_experiment_1_encoder`, a *column* vector containing the measured x_w for the experiment relative to $r(t)$ as in Equation (36) (in meters);
   - `group_#MYGROUP_experiment_1_angle`, a *column* vector containing the measured θ_b for the experiment relative to $r(t)$ as in Equation (36) (in radiants);
   - `group_#MYGROUP_experiment_1_actuation`, a *column* vector containing the v_m sent to the balancing robot's motor for the experiment relative to $r(t)$ as in Equation (36) (in Volts);
   - `group_#MYGROUP_experiment_2_times`, a *column* vector containing the sampling instants of the data relative to $r(t)$ as in Equation (37) (in seconds, and with 0 indicating the beginning of the experiment);
   - `group_#MYGROUP_experiment_2_encoder`, a *column* vector containing the measured x_w for the experiment relative to $r(t)$ as in Equation (37) (in meters);
   - `group_#MYGROUP_experiment_2_angle`, a *column* vector containing the measured θ_b for the experiment relative to $r(t)$ as in Equation (37) (in radiants);
   - `group_#MYGROUP_experiment_2_actuation`, a *column* vector containing the v_m sent to the balancing robot's motor for the experiment relative to $r(t)$ as in Equation (37) (in Volts);
   - `group_#MYGROUP_r_max`, a scalar containing which $r_{\max}$ you found for the experiment relative to $r(t)$ as in Equation (37) (in meters / seconds);

   It is important that you follow the convention for the names indicated above, since we process your data in an automatic way.

A MATLAB and SIMULINK

A.1 Useful MATLAB commands

The commands presented here are in their basic form. Please consider typing `help commmand` in your MATLAB shell.

command	description
`SYS = ss(A,B,C,D)`	creates an object `SYS` representing a state-space model
`pzplot(SYS)`	plots the pole-zero map of the dynamic system `SYS`
`pzmap(SYS)`	(alternative) plots the pole-zero map of the dynamic system `SYS`
`print('-depsc2', 'xxx.eps');`	saves the current figure as `xxx.eps`
`inv(A);`	inverse of matrix `A` (faster and more accurate than `^(-1)`)
`pid(kP, kI, kD, tf);`	creates a continuous time PID controller
`feedback(M1,M2)`	computes a closed-loop system by putting `M1` in feedback with `M2`
`bodeplot`	draws the Bode plot of a given system
`impulse`	draws the impulse response of a given system
`rlocus`	draws the root locus of a given system; may also return the locations of the closed loop roots for a specified gain
`c2d`	converts a continuous time system into a discrete one

Table 3: Useful MATLAB *commands.*

A.2 How to plot data in MATLAB / diagrams in SIMULINK

We strongly encourage to automate plotting SIMULINK's results and diagrams using the command line, rather than going and clicking with the mouse around like you were playing doom. E.g., for producing Figures 8 and 9 we used a `.m` file containing:

```
close all; clear; clc;
LoadPhysicalParameters;
LoadStateSpaceMatrices;
[afLinearizedBotZeros, afLinearizedBotPoles, fLinearizedBotGain] = ss2zp(A, B, C, D, 1);
ComputePIDGains;

open_system('../Simulink/LabA_LinearizedBot');
saveas(get_param('LabA_LinearizedBot','Handle'),'LabA_LinearizedBot_Simulink_diagram.eps');
sim('LabA_LinearizedBot');
close_system('LabA_LinearizedBot');

afFigurePosition = [1 1 10 6];

figure(1)
plot(x_w.time, x_w.signals.values);
title('x_w'); xlabel('time'); ylabel('meters')
set(gcf, 'Units', 'centimeters'); set(gcf,'Position',afFigurePosition);
set(gcf, 'PaperPositionMode', 'auto');
print('-depsc2', '-r300', 'LabA_LinearizedBot_Simulink_x_w.eps');

figure(2)
plot(theta_b.time, theta_b.signals.values * 180 / pi);
title('\theta_b'); xlabel('time'); ylabel('degrees')
set(gcf, 'Units', 'centimeters'); set(gcf,'Position',afFigurePosition);
```

```
set(gcf, 'PaperPositionMode', 'auto');
print('-depsc2', '-r300', 'LabA_LinearizedBot_Simulink_theta_b.eps');

figure(3)
plot(d.time, d.signals.values);
title('d'); xlabel('time'); ylabel('Newton')
set(gcf, 'Units', 'centimeters'); set(gcf,'Position',afFigurePosition);
set(gcf, 'PaperPositionMode', 'auto');
print('-depsc2', '-r300', 'LabA_LinearizedBot_Simulink_d.eps');

figure(4)
plot(v_m.time, v_m.signals.values);
title('v_m'); xlabel('time'); ylabel('Volt')
set(gcf, 'Units', 'centimeters'); set(gcf,'Position',afFigurePosition);
set(gcf, 'PaperPositionMode', 'auto');
print('-depsc2', '-r300', 'LabA_LinearizedBot_Simulink_v_m.eps');
```

Notice that the previous `.m` file exploits variables that have been loaded in MATLAB's workspace by the `scope` blocks in the SIMULINK diagram. For doing so you need to configure your `scope`s (the "gear" icon in the upper left corner when you open them) as in Figure 19.

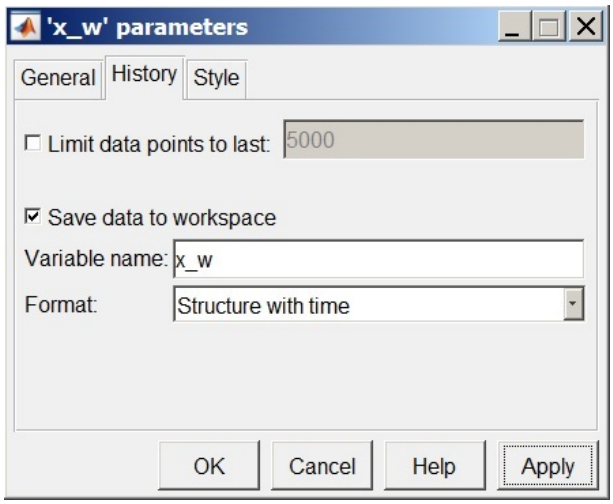


Figure 19: How to configure a scope in SIMULINK *so that it will save your data on the workspace.*

A.3 Useful SIMULINK tricks

The "tricks" presented in Table 4 help you keep your diagrams organized and easy to be debugged.

command	description
`ctrl + R` on a selected block	rotates the block
`ctrl + P`	saves the block scheme as an image
`alt + 1`	zooms to 100%
`ctrl + shift + L`	show the library browser
right-click on a block → format → hide block name	hide the label of the block
select some blocks → Ctrl + shift + x	comment / uncomment the blocks

Table 4: Useful SIMULINK *tricks.*

A.4 Useful SIMULINK blocks

The blocks presented in Table 5 instead want to help you be faster in creating your diagrams.

command	description
`Signal Builder`	for generating arbitrary signals
`LTI System`	for LTI systems (also **MIMO**)
`PID Controller`	for continuous / discrete PIDs
`Element-wise gain`	for matrix multiplication operations

Table 5: Useful Simulink *blocks.*

A.5 Managing sampling times in Simulink

Mixing continuous time with discrete time objects (or objects with different sampling times) may result in troubles. Our advice is a combination of actions:

- let your main Matlab scripts to define a variable `fSamplingPeriod`, and use it in the blocks so that you avoid hard coding (avoid hard coding especially for the sampling times matters!!);
- let every block that do not necessarily need to have a sampling time defined inherit the sampling frequency;
- let the controllers (e.g., the PIDs) define explicitly their sampling time as `fSamplingPeriod`;
- when using Simulink diagrams that will be deployed in the hardware, set the "fundamental sample time" setting in the "solver" tab of the configuration parameters of the Simulink diagram (the stuff you get when pressing `ctrl+E`) be `fSamplingPeriod`.

For debug purposes it is very useful to see the map of the sampling times of the various blocks by activating "Display → Sample Time → Colors" (sometimes you can see the map also with `ctrl+J`.