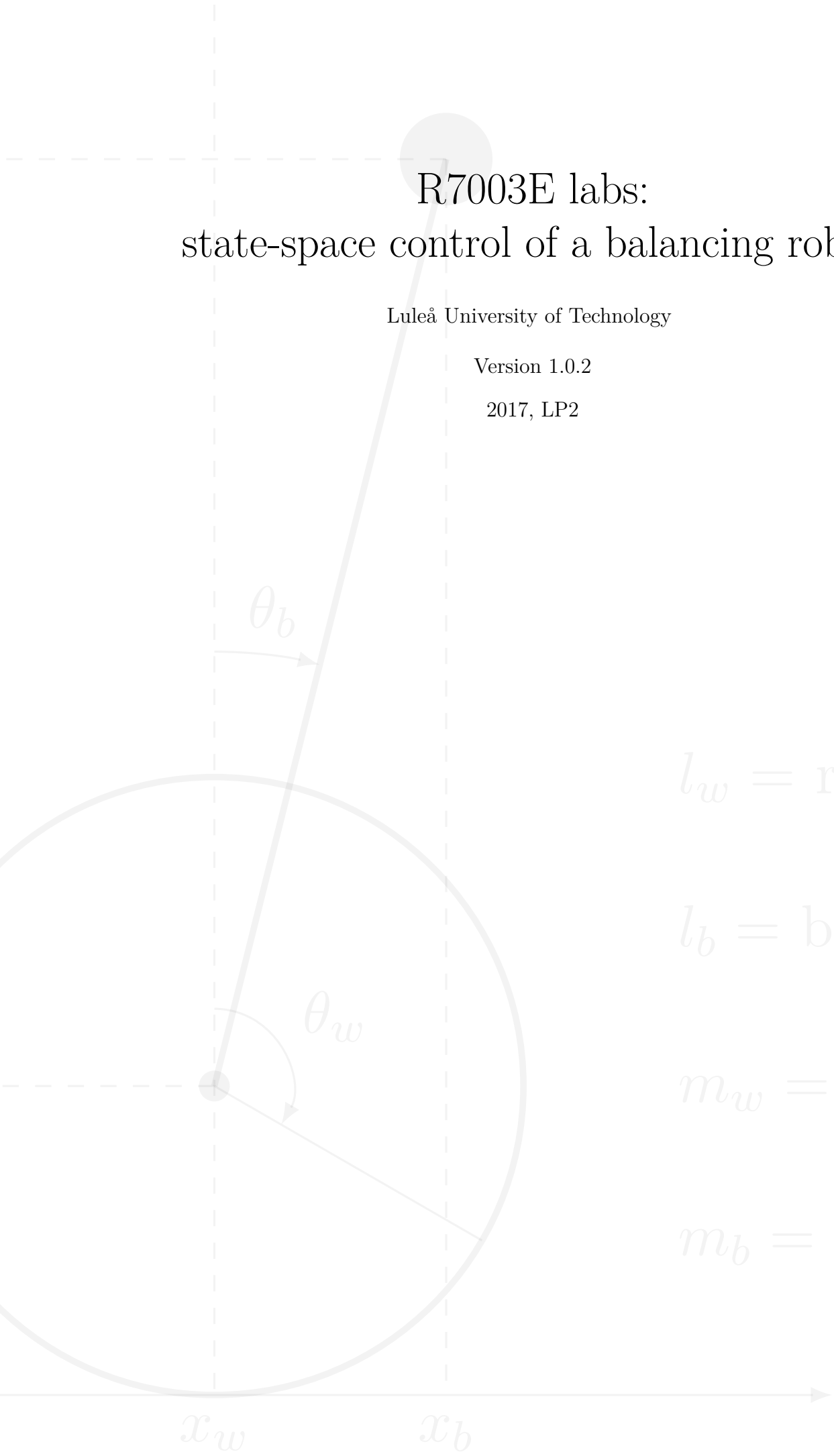
R7003E labs:
state-space control of a balancing robot

Luleå University of Technology

Version 1.0.2

2017, LP2



Contents

1 What are the labs about

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 compare the picture in Figure 1 and the scheme in Figure 2, you understand immediately how this system works: the robot would naturally fall because of gravity effects, but if you accelerate properly the wheels then 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.

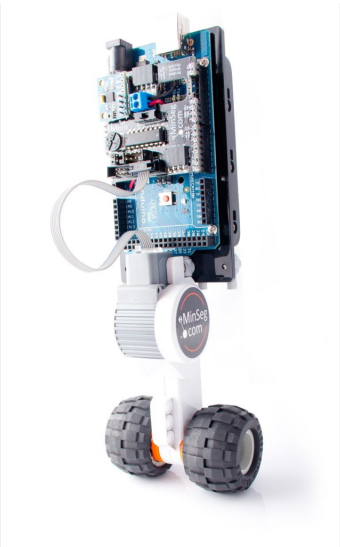


Figure 1: An example of a one-wheeled balancing robot. Actually, that specific balancing robot that you will use in this lab.

1.2 What you will do in the labs, in brief

The intended learning outcome is *not* 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 a control engineer usually does in its job. More specifically you will:

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];

develop a first prototype of the control law in lab B. Here you will test a PID, do some experiments to validate the results of lab A, some tuning of your simulator, and then prototype two more complex controllers;

improve and validate the controller in lab C and arrive at a complete scheme having that 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.

Remark 1.2.1 *Since you will receive a robot that you can bring home, you can do the labs also outside the planned time slots, in case you are sick, having problems, other impediments,*

[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?

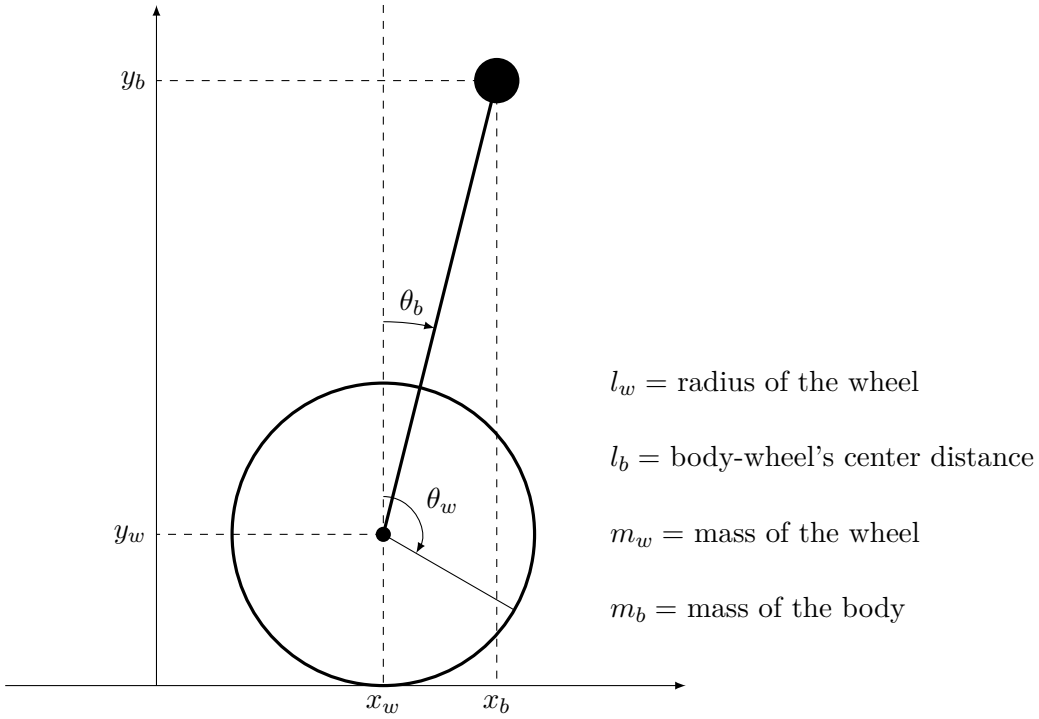


Figure 2: A simple schematic representation of a one-wheeled balancing robot where all the mass of robot (but the wheels') is concentrated in its center of mass. The notation used in this figure is the notation used throughout the whole document.

or you want to do the labs before the others. We nonetheless suggest to attend the scheduled labs, since there you will have:

- *the possibility of chatting with your peers and with the instructors;*
- *the guarantee of having people helping you with non-working hardware.*

1.3 What you will do at home, in brief

You are supposed to work a bit also at home. More specifically:

1. read **carefully** this manual. If you have questions / doubts, ask **before** doing the labs (via email, via Skype, via coming to the offices of the instructors – you have the list of all the possibilities in Section 9.9);
2. study the theory behind the assigned tasks. Below you will find some "tasks", and each of them will indicate what you should know. Thus **before** coming to the labs please do:
   (a) read the chapter relative to that lab;
   (b) go through all the various tasks;
   (c) study what they tell you to study.

1.4 How you will be evaluated, in brief

In brief, **to pass R7003E one needs to pass also the labs.** More precisely, to pass the labs one needs to hand in:

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

and get sufficient grades in the reports plus have a working demo. In details, read Section 8 and you will find all the necessary information. For now you should know that it is mandatory to hand in the reports and the demo material to have the course registered. **Notice that you need to pass the labs before doing the oral exam.**

Notice also that the labs will not contribute to the overall final grading of the whole exam, and will act as a *go / no go* flag. You can find more information on how to do your reports in Section 8.

1.5 How we would like you to interact with your peers

Each group should be composed of *maximum* 3 persons each.

Thus 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 – there exist almost[2] no stupid approach. Notice that you may also form inter-groups collaborations for solving some optional tasks (more information in Section 7).

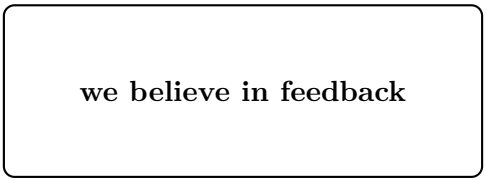
PS: do not cheat; first, we understand it. Second, let's be grown ups.

1.6 How we would like you to interact with us

Golden rules:

- feel free to ask for both help and clarifications (see Section 9.9 for our contacts);
- tell us (as soon as possible) if something in this manual is not clear / contradictory / incomplete;
- tell us (as soon as possible) if something does not work (balancing robot, computers, etc.);
- tell us (whenever you prefer) your opinions and suggestions (we care about this much more than one would expect).

To summarize these rules with a chart:



[2] Almost. A classical counterexample is putting the device on fire, something that has been proved not helping controlling it.

2 How to read this manual

The document is structured as follows:

- Sections 1 and 2 overview some general information;
- Sections 3, 4 and 5 list the tasks that you are supposed to do;
- Section 6 describes the demo you will do after the labs;
- Section 7 lists some **optional** tasks that you may want to do *if you feel like* (more info in that section);
- Section 8 describes how to report your findings;
- Section 9 presents some ancillary information and links.

2.1 How to complete the assigned tasks

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

Task 2.1.1 *do this, do that.*

Reading 2.1 *read this, study that.*

Reporting 2.1.1 *plot this, comment that.*

The **task**s indicate what you are supposed to do in the lab, the **reading**s suggest what you should know of the textbook to perform the tasks, while the **reporting**s summarize what to report and how. Often you will be asked to report something in either *parametric* or *numeric* forms (or both). This means the following:

parametric form: you express the quantity as a function of some parameters, e.g.,

$$A = \begin{bmatrix} a_{ij} \end{bmatrix}, \qquad a_{ij} = m_b g l_b + I_b \tag{1}$$

numeric form: you express the quantity with its numerical value, e.g.,

$$A = \begin{bmatrix} 3.25 \end{bmatrix} \tag{2}$$

Table 4 summarizes the values of the parameters describing your balancing robot.

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. Once again, if some task, reading or reporting is unclear, contact us as soon as possible (Section 9.9). More information on how to report your findings is also in Section 8.

Remark 2.1.1 *Our advice is to do not try to complete the reporting during the labs. Instead simply solve the tasks, and every time that you use Matlab or Simulink* ***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*** *(or use dropbox or dropbox-like services, an excellent alternative to manual backups).*

2.2 Notation

You are **imposed** to use the notation in Table 5 (page 64). Not following the notation $\Longrightarrow$ failing the labs.

3 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 an prototyping purposes. Instrumental to the development of the simulator you are supposed to start with describing mathematically the system. The reporting for this section should be done following the corresponding template (see Section 8). Notice that this lab does not require you to be physically at the University, and you can do it whenever / wherever you want (as soon as you have Matlab and Simulink installed).

3.1 Derive the Equations of Motion

Our first step towards having a simulator of the balancing robot is to derive the Equations of Motion (EOM) of the system. To this aim you should read:

Reading 3.1

- *Section 2.1*
- *Section 2.3.2*
- *Section 3.1.7*
- *Section 3.3*

of the textbook.

We model the balancing robot as in Figure 3 (check also the table for the notation on page 64): the body of the robot is a thin pole with its mass m_b concentrated only in the center of mass of the robot, and with this mass being distant l_b from the center of the wheel. In its turn the wheel has radius l_w, and mass m_r.

The aim of this subsection is then to find the dynamics for the position of the wheel x_w and the angle of the body θ_b given the voltage v_m in input to the motor. Mathematically, we need to express the system as

$$\begin{cases} \ddot{x}_w &= f_{x_w}\left(x_w, \dot{x}_w, \theta_b, \dot{\theta}_b, \ddot{\theta}_b, v_m\right) \\ \ddot{\theta}_b &= f_{\theta_b}\left(x_w, \dot{x}_w, \ddot{x}_w, \theta_b, \dot{\theta}_b, v_m\right) \end{cases} \tag{3}$$

for opportune functions f_{x_w} and f_{θ_b} (notice that we need two equations since we have two variables). To this aim we make the following assumptions:

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 unique force that can be commanded by us is the torque applied by the motor to the wheel, and this torque is driven by the voltage that is applied to the motor.

Remark 3.1.1 *There are a lot of different equivalent ways of finding EOMs of mechanical systems. In the following we* **suggest** *you to use Newton laws, but you are absolutely free to derive however you want. In case you do not feel like following the suggestions below, as an alternative task you have to describe how you derived instead.*

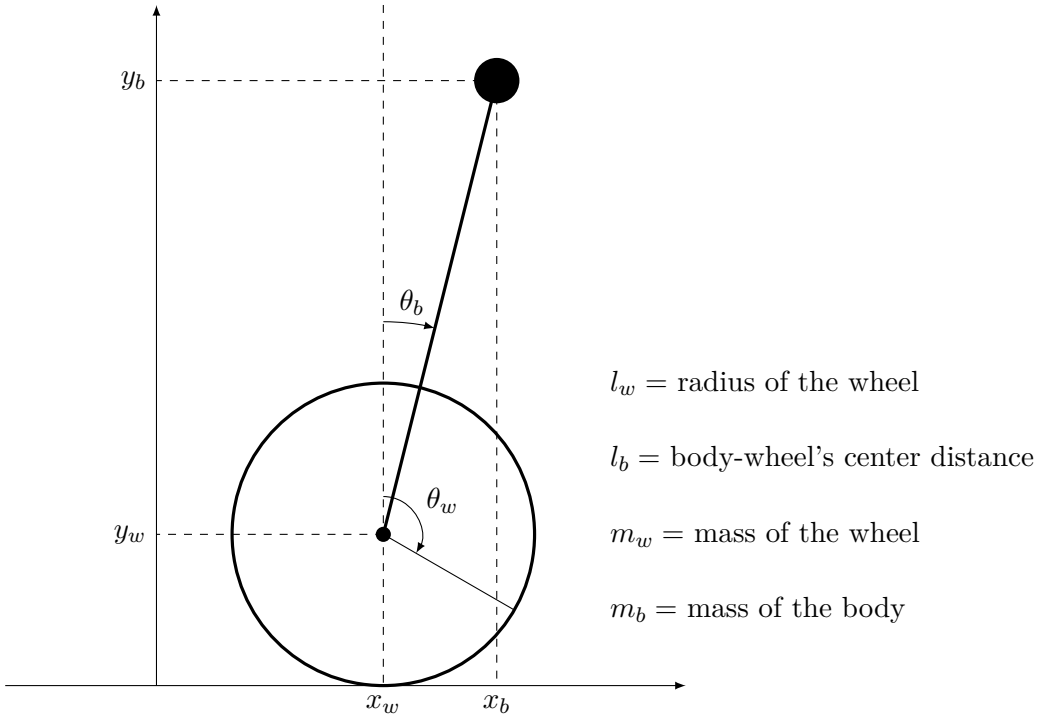


Figure 3: A simple schematic representation of a one-wheeled balancing robot where all the mass of robot (but the wheels') is concentrated in its center of mass (repetition of Figure 2 on page 5, for readability convenience).

If we use a Newton laws based approach to the modeling of the EOM, we can follow these 3 steps:

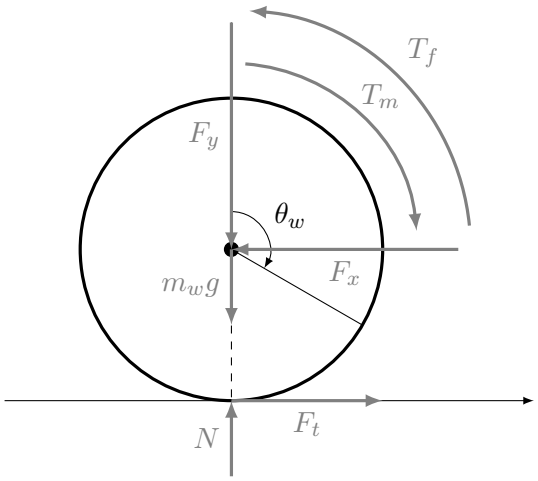
1. derive the dynamics of the wheel, i.e., how a torque applied to the wheel transforms into translation and forces applied to the body;
2. derive the dynamics of the body, i.e., how a force applied to the body transforms into dynamics of its center of mass;
3. derive the dynamics of the motor, i.e., how the voltage applied to the motor transforms into a torque applied to the wheel.

It is simple to find (3) by following this workflow: *a)* derive the dynamics of the wheel and of the body independently keeping the motor's torque T_m inside the equations; *b)* combine the two dynamics into an unique dynamics with the structure (3) but with T_m instead of the voltage v_m (that is actually our input); *c)* derive the dynamics of the torque T_m as a function of v_m; *d)* plug in the solution of *c)* in the solutions of *b)* and obtain a dynamics with the structure (3).

Remark 3.1.2 *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".*

3.1.1 Dynamics of the wheel

Consider the summary of the forces that apply to the wheel of the robot, described graphically in Figure 4 (the friction torque T_f models the effects of viscous frictions, while F_x and F_y denote the decompositions of the tension between the wheel and the body).



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

Task 3.1.1 *Find the differential equations governing the temporal evolution of the quantities of interest. I.e., express the dynamics for x_w, y_w and θ_w given the external forces $\boldsymbol{F} = \{T_m, T_f, F_x, F_y, F_t, m_b g\}$ as*

$$\begin{cases} \ddot{x}_w &= f_{x_w}\left(x_w, \dot{x}_w, \theta_w, \dot{\theta}_w, \boldsymbol{F}\right) \\ \ddot{\theta}_w &= f_{\theta_w}\left(x_w, \dot{x}_w, \theta_w, \dot{\theta}_w, \boldsymbol{F}\right) \end{cases} \tag{4}$$

for opportune functions f_{x_w} and f_{θ_w}.

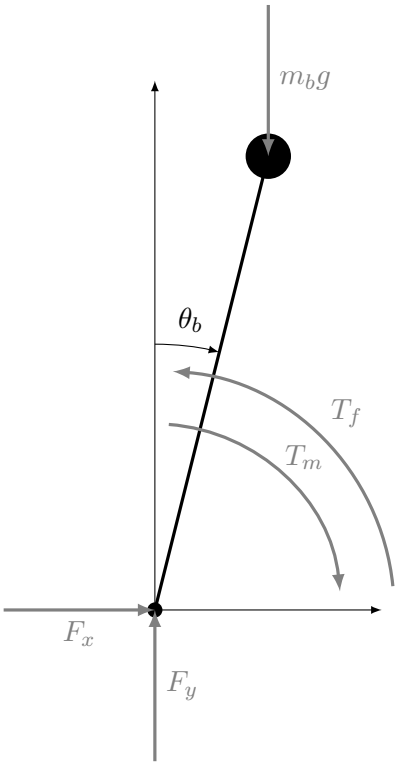
Reporting 3.1.1 *No reporting here.*

Hint 3.1.1 *Write down the Newton law for the horizontal movement of the wheel, the law for the vertical movement, and the law for the angular movement of the wheel, so to eventually have 3 separate equations.*

3.1.2 Dynamics of the body

Consider the summary of the forces that apply to the body of the robot, described graphically in Figure 5. Notice that since the motor is rigidly attached to the body, when a torque T_m is

applied to the wheels, then that torque will lead to a negative θ_b (similar considerations for the friction torque T_f).



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.

Task 3.1.2 *Find the differential equations governing the temporal evolutions of the quantities of interest. I.e., express the dynamics for x_b and θ_b given the external forces $\boldsymbol{F} = \{T_m, T_f, F_x, F_y, m_b g\}$ as*

$$\begin{cases} \ddot{x}_b &= f_{x_b}\left(x_b, \dot{x}_b, \theta_b, \dot{\theta}_b, \boldsymbol{F}\right) \\ \ddot{\theta}_b &= f_{\theta_b}\left(x_b, \dot{x}_b, \theta_b, \dot{\theta}_b, \boldsymbol{F}\right) \end{cases} \tag{5}$$

for opportune functions f_{x_b} and f_{θ_b}.

Reporting 3.1.2 *No reporting here.*

Hint 3.1.2 *As before, write down the Newton law for the horizontal movement of the body, the law for the vertical movement, and the law for the angular movement of the body, so to eventually have 3 separate equations. As for the last step, it is way easier to chose as point of rotation of the system the mass of the body instead of the center of the wheel, so that $m_b g$ will not contribute with torque, leading the angular dynamics be affected only by the torques T_m and T_f (and be careful with the signs!).*

Given the two systems (4) and (5) we now seek to get closer to (3) by finding a system where the unique variables considered are x_w and θ_b, and where the internal tension forces F_x and F_y and the tractive force F_t disappear from the dynamics. In other words, we now seek to combine (4) and (5) so to derive something like

$$\begin{cases} \ddot{x}_w &= f_{x_w}\left(x_w, \dot{x}_w, \theta_b, \dot{\theta}_b, \ddot{\theta}_b, T_m, T_f\right) \\ \ddot{\theta}_b &= f_{\theta_b}\left(x_w, \dot{x}_w, \ddot{x}_w, \theta_b, \dot{\theta}_b, T_m, T_f\right) \end{cases} \tag{6}$$

for opportune functions f_{x_w} and f_{θ_b} (notice: since we are considering two variables we need two equations). In other words, we need to eliminate all the variables that are not x_w and θ_b.

Task 3.1.3 *Combine and rewrite* (4) *and* (5) *into a system like* (6).

Reporting 3.1.3 *No reporting here.*

Hint 3.1.3 *The task is essentially to take all the variables that are not x_w and θ_b in the equations that have been obtained before and write them as functions of x_w, θ_b, T_m and T_f.*

To this aim we can use two separate facts: first,

$$\ddot{\theta}_w = \ddot{x}_w / l_w, \tag{7}$$

and this will be very useful in the Newton law for the angular movement of the wheel.

Second, to rewrite F_t, F_x and F_y we can directly use the Newton laws for the linear movements of the wheel and of the body. If everything goes as expected then there will be the need for simplifying the quantity "$\ddot{y}_b \sin(\theta_b) - \ddot{x}_b \cos(\theta_b)$". To this aim one can exploit the following equivalences:

$$\begin{cases} x_b &= x_w + l_b \sin(\theta_b) \\ \dot{x}_b &= \dot{x}_w + \dot{\theta}_b l_b \cos(\theta_b) \\ \ddot{x}_b &= \ddot{x}_w + \ddot{\theta}_b l_b \cos(\theta_b) - \dot{\theta}_b^2 l_b \sin(\theta_b) \\ y_b &= y_w + l_b \cos(\theta_b) \\ \dot{y}_b &= \dot{y}_w - \dot{\theta}_b l_b \sin(\theta_b) = -\dot{\theta}_b l_b \sin(\theta_b) \\ \ddot{y}_b &= -\ddot{\theta}_b l_b \sin(\theta_b) - \dot{\theta}_b^2 l_b \cos(\theta_b) \end{cases} \tag{8}$$

Before continuing we then consider that T_f is a torque due to mechanical frictions, and that thus is proportional to the rotational speed of the motor, i.e.,

$$T_f = b_f\left(\dot{\theta}_w - \dot{\theta}_b\right) = b_f\left(\frac{\dot{x}_w}{l_w} - \dot{\theta}_b\right). \tag{9}$$

We can thus make immediately this substitution in the solution of Task 3.1.3 and arrive at a form of the type

$$\begin{cases} \ddot{x}_w &= f_{x_w}\left(x_w, \dot{x}_w, \theta_b, \dot{\theta}_b, \ddot{\theta}_b, T_m\right) \\ \ddot{\theta}_b &= f_{\theta_b}\left(x_w, \dot{x}_w, \ddot{x}_w, \theta_b, \dot{\theta}_b, T_m\right) \end{cases} \tag{10}$$

instead of the type (6).

3.1.3 Dynamics of the motor

The final step is to express the torque T_m as a function of v_m, so that we can modify (6) to arrive to our final (3). Consider then that our motor is a Direct Current (DC) motor schematizable as in Figure 6.

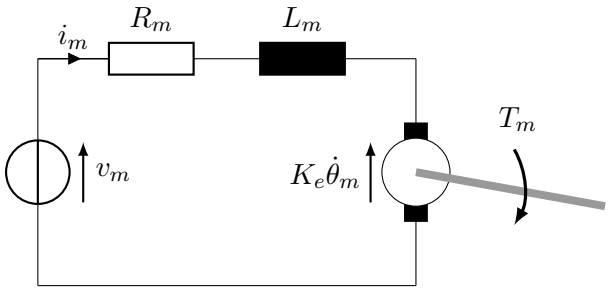


Figure 6: Schematic representation of a DC motor. Here θ_m indicates the angle of the motor (that, in our case, will be a function of θ_w and θ_b...)

Task 3.1.4 *Find the differential equations governing the temporal evolution of the quantities of interest assuming the inductance L_m and the motor viscous coefficient b_m negligible. I.e., express the relation of T_m with v_m, $\dot{x}_w$ and $\dot{\theta}_b$ as*

$$T_m = f_{T_m}\left(v_m, \dot{x}_w, \dot{\theta}_b\right) \tag{11}$$

for an opportune function f_{T_m}.

Reporting 3.1.4 *No reporting here.*

Hint 3.1.4 *Do as the textbook does.*

Now we have all the ingredients to derive (3) in a symbolic form. Notice that it is not yet time to substitute the symbols with the values provided in Table 4.

Task 3.1.5 *Plug* (9) *and* (11) *into* (6) *and obtain* (3) *in a symbolic form.*

Reporting 3.1.5 *Report the final EOMs you computed.*

3.2 Linearize the Equations of Motion

If you did everything right, you should get the following non-linear terms:

- $\sin(\theta_b)$;
- $\ddot{x}_w \cos(\theta_b)$;
- $\ddot{\theta}_b \cos(\theta_b)$;
- $\dot{\theta}_b^2 \sin(\theta_b)$.

Since up to know you know how to build controllers only for linear systems[3], we need to linearize our EOM. To do so you may need to refresh how to linearize a system:

Reading 3.2

- *Section 9.2.1;*
- *lecture notes relative to the linearization of nonlinear systems.*

Now you are ready to solve the following:

Task 3.2.1 *Linearize the EOM around an opportunely chosen point, assuming that centripetal forces are negligible.*

Reporting 3.2.1

1. *Say which linearization point you choose;*
2. *say* why *you choose that specific point;*
3. *write the linearized EOM.*

[3] If you want to know more about control of non-linear systems then consider attending also the course R7014E in 2017-2018 LP3.

3.3 Write the linearized Equations of Motion in State-Space form

Reading 3.3

- *Section 7.2.*

Since this course is focused on control based on State-Space (SS) models, we do now rewrite our EOM as

$$\begin{cases} \dot{\boldsymbol{x}} &= A\boldsymbol{x} + Bu \\ y &= C\boldsymbol{x} + Du \end{cases} \tag{12}$$

for opportune $\boldsymbol{x}$, A, B, C and D. This means that you have to select the structure of your state, i.e., if $\boldsymbol{x} = [x_1, x_2, \ldots]^T$ then you have to say what is x_1, x_2, etc., then find the single equations $\dot{x}_1 = \ldots$, $\dot{x}_2 = \ldots$, etc., and then finally stack everything up in a matrix form like the one in (12). As for the input and output, assume for now:

$$\begin{aligned} u &= v_m \\ y &= \theta_b \end{aligned} \tag{13}$$

In other words, for now our input is the voltage applied to the motor, while for now the measure is the angular deviation of the balancing robot from the vertical upright position.

Task 3.3.1 *Write the linearized EOM as in* (12) *considering the inputs and outputs as in* (13)*.*

Reporting 3.3.1

1. *Say what is your choice for* $\boldsymbol{x}$*;*
2. *write the matrices* A*,* B*,* C *and* D *in* (12) *in parametric form;*
3. *write the matrices* A*,* B*,* C *and* D *in* (12) *in parametric numeric form (for this purpose use Table 4 on page 63).*

Hint 3.3.1 *It is much easier and numerically stable*[4] *to do as follows:*

1. *write the system in parametric form as*

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

   where the various γ*,* α *and* β *are functions of the various parameters in Table 4 (e.g.,* and this is just an example*,* $\alpha_{23} = I_b + m_r + R_m$*);*

2. *write in Matlab a script where you first copy the code*

```
g   = 9.8;
b_f = 0;
m_b = 0.381;
l_b = 0.112;
I_b = 0.00616;
m_w = 0.036;
l_w = 0.021;
I_w = 0.00000746;
R_m = 4.4;
L_m = 0;
b_m = 0;
K_e = 0.444;
K_t = 0.470;
```

and then you write down all the various $\alpha_{ij} = \ldots$. In this way you will compute your γs, αs and βs immediately;

3. *since*

$$\begin{bmatrix} \ddot{x}_w \\ \ddot{\theta}_b \end{bmatrix} = \begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \end{bmatrix} \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} + \begin{bmatrix} \gamma_{11} & \gamma_{12} \\ \gamma_{21} & \gamma_{22} \end{bmatrix}^{-1} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} u \tag{15}$$

you can create your A, B, C variables in Matlab 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).

3.4 Determine the transfer function relative to system (12)

Here you need again:

Reading 3.4

- *Section 7.4.*

Task 3.4.1 *Compute the transfer function, the poles and the zeros of the system.*

Before using Matlab, we suggest you to start by getting an intuition of which poles and the zeros you will get by analyzing the structure of

$$sI - A \qquad \text{and} \qquad \begin{bmatrix} sI - A & -B \\ C & D \end{bmatrix}. \tag{16}$$

Only after this use Matlab (and the commands in Table 3 on page 62) 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[5], 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 3.4.1

1. *Write the transfer function in the form*

$$G(s) = K \frac{\prod_i (s - z_i)}{\prod_j (s - p_j)} \tag{17}$$

2. *describe, in case you saw them, whether you experienced some numerical problem with Matlab.*

Once again we suggest you to use Matlab and the workspace loaded before.

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

3.5 Design a PID controller stabilizing the transfer function computed in Section 3.4

Reading 3.5

- *Section 4.3.*

It is important to notice the following things:

1. at this stage we just want to stabilize the transfer function from the motor voltage v_m to the body angle θ_b; in other words, we are designing a PID that makes the robot do not fall. Thus our reference signal for θ_b is zero;
2. nothing is said about x_w. For now we are ignoring what happens to the position (in the x axis) of the robot;
3. plotting 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, but the stability domain is going to be very small. A PID instead works much better and gives a much bigger stability domain. We thus use a PID as our first initial choice.

Task 3.5.1 *Design the PID using the poles allocation method. Since we are doing simulations, you are free to choose to set the poles wherever you want (as soon as the closed loop is stable, of course).*

Reporting 3.5.1

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 the values of the parameters of the PID;*
3. *write the resulting closed loop function in its numerical form.*

Hint 3.5.1 *If you don't remember how to do poles allocation, then check the corresponding video in the webpage of the course.*

3.6 Model the effect of disturbing the robot

Reading 3.6 *No readings here!*

When controlling a system it may be meaningful to analyze the effects of that disturbances that will affect the normal operations of the plant. In our case we may want to check what happens if somebody pokes the robot, i.e., applies an impulsive horizontal force to the center of mass of the body of the balancing robot. In practice, we would like to know what happens if there is a term in Figure 5 that sums with F_x.

Task 3.6.1 *Modify the EOM of your balancing robot so to include an other additional input that is called d and that models somebody poking the robot. Derive also the linearized and state space versions of these EOM.*

Reporting 3.6.1

1. *Write the novel EOM in parametric form;*
2. *write the novel linearized EOM in state space in numerical form.*

Hint 3.6.1 *The poke affects the body, thus it affects the Newton laws of the dynamics of the body. Moreover, the poke is "horizontal"...*

You should now notice that the novel disturbance can be thought as an additional input, but that 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, and 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.

3.7 Check if everything is working as it should be

Reading 3.7 *No readings here!*

We have now all the ingredients to start checking if things are working as expected. The task now is to simulate what the linear systems does when it is subject to a small poke.

Task 3.7.1 *Implement a Simulink simulator 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*

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}. \tag{18}$$

Perform an experiment for which the external disturbance d is

$$d(t) = \begin{cases} 0.1 & \text{for } t \in [0, 0.1] \\ 0 & \text{otherwise,} \end{cases} \tag{19}$$

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 linearized robots dynamics we instead suggest 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 (as above). We suggest to implement the PID through a `PID Controller` block; this block requires to set the `Filter coefficient (N)`, and if you do not know what it does then check Simulink's help and try to think at what happens if `N` is very big or very small.

Important: we **strongly discourage** you to hard-code Simulink's blocks[6]. Moreover we suggest also to automate the production of figures, e.g., as we did in Section 9.3 on page 59.

Reporting 3.7.1

1. *Plot your Simulink scheme;*
2. *plot the realizations of $\theta_b(t)$, $x_w(t)$, $v_m(t)$ and $d(t)$.*

Our solution is in Figures 7 (the Simulink diagram) and 8 (the plots of the signals). We included the solution of the previous task because it is very important to understand what happens here: as you can notice, the system acts as an integrator on x_w. In other words,

[6]With this we mean the following: assume that you have to implement a P controller in Simulink. In Matlab you computed $k_P = 10$. Hard-coding means that in Simulink you create a gain and you put as its parameter "10". This means that if you later realize that it is better to use $k_P = 15$, you need to go to that P block in Simulink and change things by hand. Probably you realize that it is better at this point to let the Simulink block load, as its k_P, the variable that you used in Matlab to compute that value.

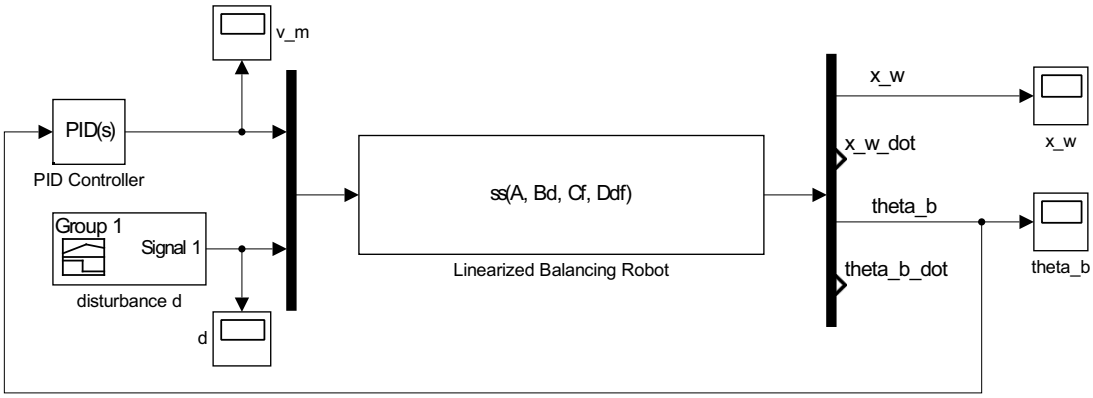


Figure 7: Our Simulink solution for Task 3.7.1.

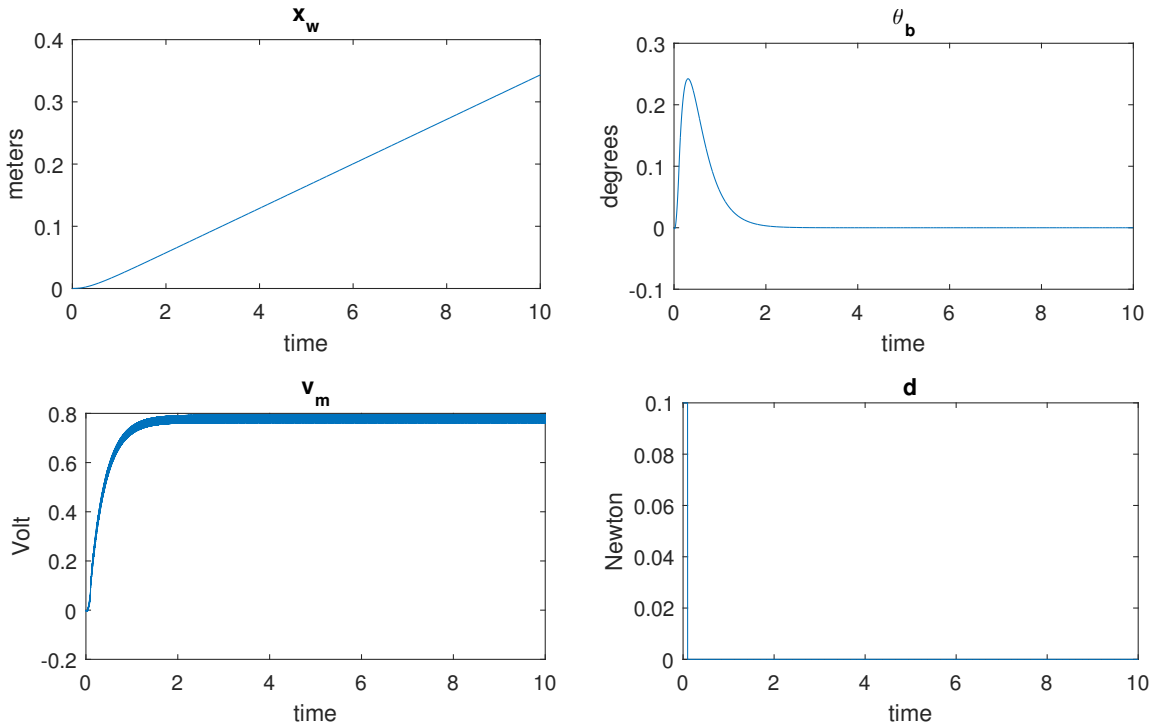


Figure 8: The signals that we obtained solving Task 3.7.1.

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 null. If you implement this PID in your robot thus you should expect to run after it all the time. In mathematical terms, the system is actually still *unstable*.

One may then wonder: *may we stabilize also 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 9.

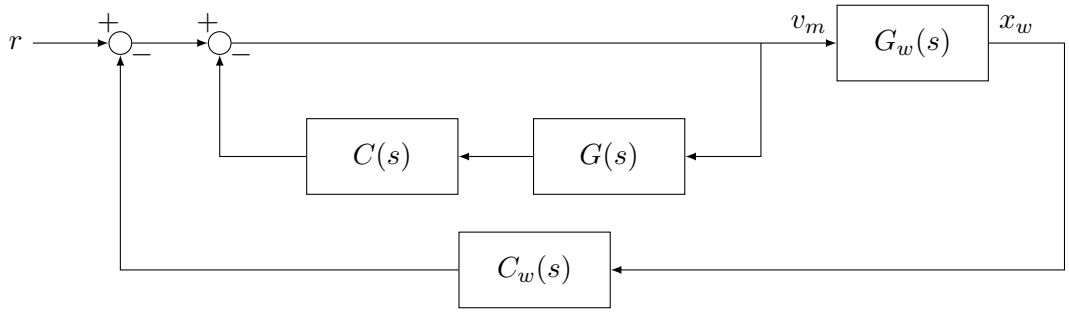


Figure 9: Alternative block schematic representation of the linearized balancing robot and the associated controllers.

The novel block diagram in Figure 9 has $G(s)$ and $C(s)$ as before (i.e., the transfer functions from v_m to θ_b of Section 3.4 and the PID controller of Section 3.5), plus 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 may then want to design a further PID controller and allocate the poles for the novel 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[7].

Let's thus ignore this problem, stop pursing it here, and focus for now only on doing control for θ_b: we will indeed see how with state space formulations we can solve this control problem much more easily. Summarizing, the message of this section wants to be: a single PID controlling θ_b won't stabilize x_w; one may add an other PID for controlling also x_w, but the computations start becoming cumbersome; with state space notation this control problem will become easier.

[7]The problem is that the novel system is Multi Input Multi Output (MIMO), since it has v_m and d as inputs, and θ_b and x_m as outputs. Controlling MIMO systems with PIDs requires special techniques; if you want to know more then consider following R70xxE in 2017 LP1.

3.8 Convert the controller to the discrete domain

Let's then focus to the problem of implementing $C(s)$ in our real balancing robot. Before proceeding, we notice that the Arduino board is a digital controller – in other words, you cannot implement $C(s)$ as it is; you must discretize it. How? Let's start by reading the following:

Reading 3.8

1. *Pages 335-336;*
2. *Section 8.3.2;*
3. *Section 8.3.6.*

As you know from there are several discretization methods. For now we focus on Zero Order Hold (ZOH) methods, and defer more considerations for Lab C.

Task 3.8.1 *Compute the bandwidth of the linearized balancing robot, and decide the sampling period of the digital controller. Compute then the controller in terms of difference equations.*

Reporting 3.8.1

1. *Report the bandwidth of the system;*
2. *report the chosen sampling time;*
3. *motivate your choice;*
4. *report the transfer function $C(z)$ of your controller in its parametric form.*

3.9 Simulate the closed loop system

Reading 3.9 *No readings here!*

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 10.

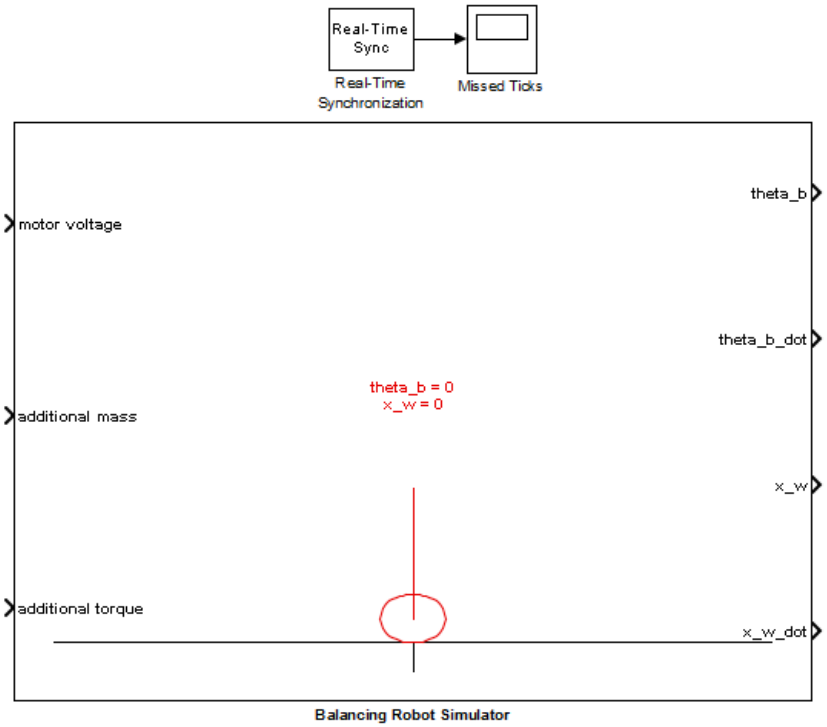


Figure 10: Our Simulink balancing robot simulator. Double-clicking on it activates the mask in Figure 11, where you can set the various physical parameters describing your balancing robot.

The simulator of Figure 10 implements the nonlinear EOM that you should have derived before; it thus *approximates* the behavior 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 won't break it. So don't fear to experiment with it! (You will get a different recommendation when playing with the real device.) To slow down/up your simulations (i.e., see the robot moving or not) set opportunely the block `Real Time Synchronization`.

Our aim is now to check three things:

1. is the controller stabilizing the nonlinear dynamics?

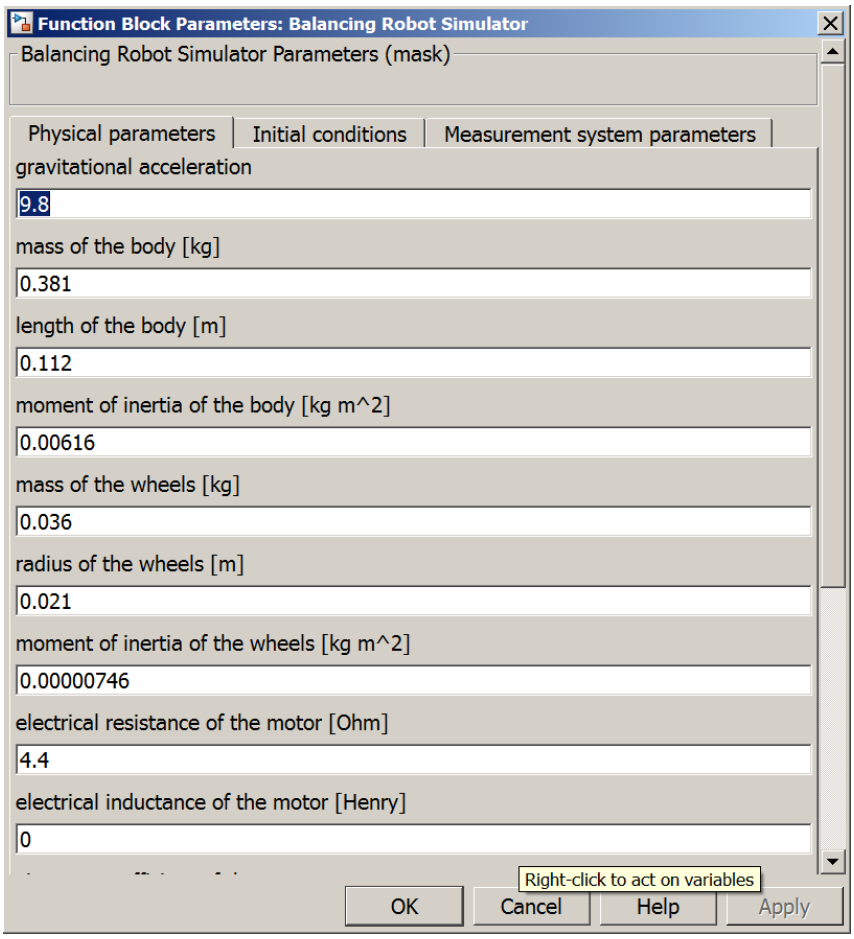


Figure 11: Mask for setting the parameters of the simulated balancing robot. The default values are the ones you should use in this lab.

2. is the linearized dynamics a good approximation of the nonlinear ones?

3. is the chosen sampling time good for control purposes?

Task 3.9.1 *Implement two Simulink schemes:*

1. *one as in Figure 12, where the simulated and linearized robots are controlled by the same continuous time PID $C(s)$ and subject to the same disturbance d;*

2. *one as in Figure 12, where the simulated and linearized robots are controlled by the same discrete time PID $C(z)$ and subject to the same disturbance d.*

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

$$d(t) = \begin{cases} \overline{d} & \text{for } t \in [0, 0.1] \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

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 two different cases (continuous and discrete control) for a $\overline{d}$ in (20) equal to half of that one that makes your simulated robot fall.

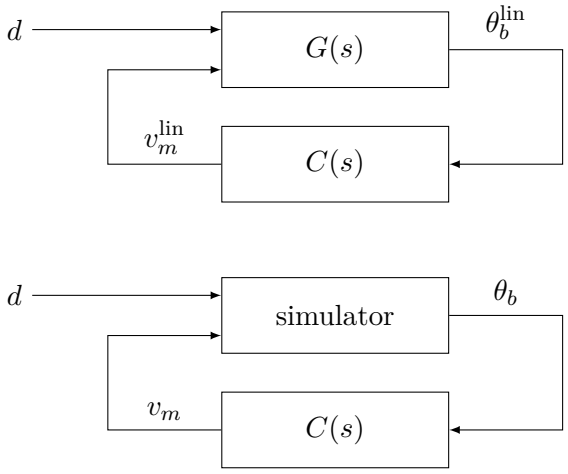


Figure 12: Block-schematic representation of how we compare the linearized dynamics with the nonlinear EOM. The disturbance d represents the same force corresponding to somebody poking the robot, while $C(s)$ is the same controller for both the linear dynamics $G(s)$ and the simulator.

Important: Figure 12 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 – check Figures 8.1 and 8.6 in the textbook if you are unsure of what we are talking about. Simulink then does this surrounding by itself. In other words, it is sufficient to change the PID block from "continuous" to "discrete", and it will be like surrounding the controller with the necessary blocks.

Reporting 3.9.1

1. *Plot your Simulink schemes;*

2. *report the smallest values $\overline{d}$ that make your robot fall (they will be two, one for the continuous and one for the discrete case);*

3. *plot $\theta_b(t)$, $v_m(t)$, $\theta_b^{lin}(t)$ and $v_m^{lin}(t)$ (again, one for the continuous and one for the discrete case);*

4. *discuss what you understood from these experiments, and if (and how) it helped you tuning the PID controller.*

Troubleshooting:

- managing the sampling times in Simulink may be a hassle. Consider reading Section 9.6 before moving from continuous time domains to discrete time domains;
- moving from continuous to discrete may also introduce numerical problems, e.g., make a stable plant "explode"[8]. If this happens to you, then consider changing the "Discrete time settings" of your PIDs (set, e.g., to "Backward Euler" or "Trapezoidal". We will understand better what this means towards the end of the course).

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.

[8]It is because in our case we have a double integrator and numerical perturbations may be fatal for Simulink.

3.10 (Optional) play with the simulator

This is something you are not compelled to do, but that of course is (we think) a very useful exercise. Our suggestion is 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. (And if you find something that is worth to be mentioned in the lab report, of course mention it.)

4 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.

4.1 How to do tests on the robot

Reading 4.1 *No readings here!*

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 14). When you do this kind of tests, after the end of the simulation the robot will do nothing;
- the other is *Deploy to hardware* (`ctrl+B` from Simulink), for which the robot is loaded with some code that is continuously run as soon as 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 13).

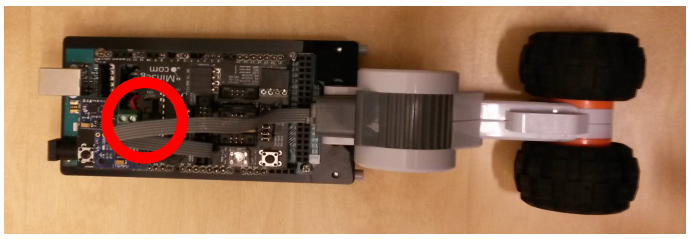


Figure 13: Batteries switch.

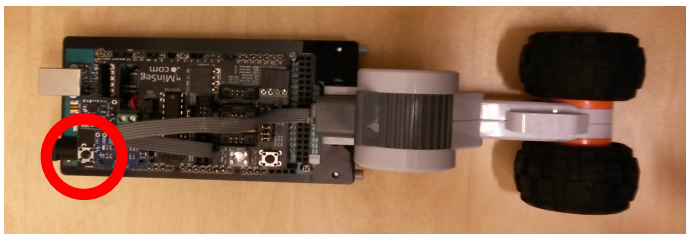


Figure 14: Reset button. When a "External" Simulink simulation finishes the robot "freezes" its state – for example, if the motors were receiving 7 volts then they will continue receiving 7 volts ad libitum. *To stop the robot press the highlighted button.*

Troubleshooting: our systems configurations have been working with the given instructions, but you know that Murphy is omnipresent. In case you find some problems **please report them as soon as possible** so that we will complete the following list:

- when Simulink says "access denied" when downloading some code to the robot, *sometimes* the solution is to cancel 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.

4.2 Communicating with the balancing robot

Reading 4.2 *No readings here!*

We start by checking if we are able to communicate with the robot. To this aim follow these steps (notice that these instructions refer to the lab rooms A15xx in campus Luleå; for the installation in your own computer please follow also the instructions in Section 9.1):

1. download from http://staff.www.ltu.se/~damvar/R7003E-2017-LP2.html the code for the various labs (it is a `.zip` file, unzip it wherever you prefer);
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;
3. open Matlab;
4. connect the robot and the computer with the USB cable (no need for plugging the batteries at this moment now). Matlab should say that an Arduino 2560 has been connected;
5. open Simulink (type `simulink` in Matlab's command line and then "enter"), and then open the Simulink file `LabB_CheckCommunications.slx`;
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 highlighted in Figure 14 on page 30);
9. now you can check if you got the data from the robot in two different ways[9]:
   (a) double-clicking on the scopes in the Simulink diagram (you can actually see the data on-line);
   (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');
   ```

[9] In our opinion visualizing with Simulink is more useful for debugging purposes; visualizing with Matlab is instead better for reporting ones, since you can automate saving and moving figures as you like. Of course you are free to do as you wish.

```
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: from 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.

4.3 Checking that reality is far from ideality

Reading 4.3 *No readings here!*

If you successfully communicated with the robot you probably noticed that the behaviors of the accelerometer and of the gyro are very far from the ideal case. For example, our measurements from a robot lying down and still on a table was like in Figure 15.

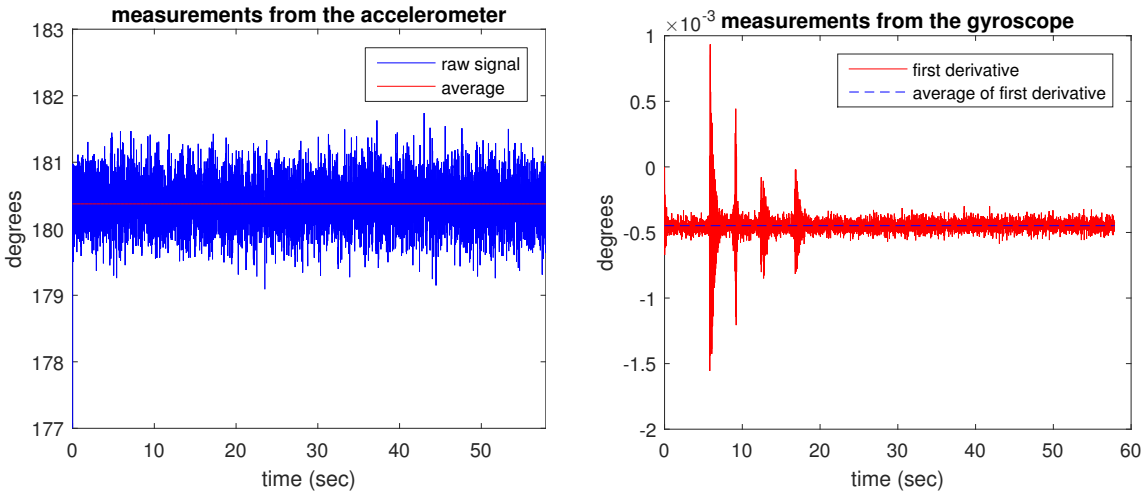


Figure 15: Examples of raw data coming from the gyroscope and the accelerometer mounted on our balancing robots.

As you may suspect, using the measurements from the gyro and the accelerometer needs some opportune filtering. Our approach 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). Very important:

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.*

4.4 Test the PID controller

Reading 4.4 *No readings here!*

We now 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 do not have them fall all the time);
2. turn on the batteries switch (i.e., towards "batt on", see Figure 13);
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 4.2 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 to do not perform too well (for example, ours was standing maximum some 10 seconds). So do not lose too much time in wiggling with the PID controller.

Task 4.4.1 *Test your PID controller as above.*

Reporting 4.4.1 *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.*

Troubleshooting:

- 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.

4.5 Check the controllability and observability properties of the linearized system (12)

To this aim it is better if you refresh your knowledge in:

Reading 4.5

- *Section 7.4;*
- *Section 7.7.*

Task 4.5.1 *Compute and discuss the controllability and observability properties of the linearized system* (12) *in numeric form.*

Reporting 4.5.1

1. *Write the numerical values for $\mathcal{C}$ and $\mathcal{O}$;*
2. *write the controllability and observability indexes;*
3. *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.*

4.6 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 two controllers:
   - one by doing poles allocation;
   - an other one by using a Linear-Quadratic Regulator (LQR) approach;
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.

Reading 4.6

- *Section 7.5.1*
- *Section 7.6.2*

We thus start from constructing a controller through a second order approximation of the plant, and checking its performance in simulation. As for the poles location, feel free to do some tests, but keep in mind the practical considerations in Sections 7.5.1 and 7.6.1 in the textbook.

Task 4.6.1

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 4.6.1 *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.*

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

Reading 4.7

- *Section 7.6.2 (also the examples!)*

We now proceed by computing that K that minimizes of the continuous time performance index

$$\mathcal{J} = \int_0^{+\infty} \left(\rho \begin{bmatrix} x_w & \dot{x}_w & \theta_b & \dot{\theta}_b \end{bmatrix} W \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} + u^2 \right) dt \tag{21}$$

where W is a matrix that indicates how we weight undesirable states. *Notice that W is a design parameter that you have to choose!* For example, you may choose

$$W = \begin{bmatrix} 5 & & & \\ & 1 & & \\ & & 10 & \\ & & & 2 \end{bmatrix} \tag{22}$$

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 $\dot{x}_w = 1$. ρ instead, as you know, weights the relative unhappiness of having to use big control efforts (u^2) or having a state $\boldsymbol{x}$ far from the origin (i.e., where we would like it to be). (Remember that you can always change your mind after having observed the outcomes of your tests, and that if you code in a good way then it is fast to change numbers; so do not fear to start by throwing in some numbers without caring too much.)

Task 4.7.1 *1. choose W in (22);*

2. *draw the Symmetric Root Locus (SRL) for the obtained system (recall that $W = C_1^T C_1$);*
3. *select ρ in (21);*
4. *compute the LQR gain K minimizing the cost (21);*
5. *implement and test the novel controller as did in Task 4.6.1 (comment out the things that are not needed anymore in* `LabB_ControllerOverSimulator_Continuous_Parameters.m`*);*
6. *compare the current control performance against the ones obtained using the second order approximation poles allocation scheme. Based on the simulative results, decide which controller you want to use in the subsequent tasks.*

Reporting 4.7.1 *Report:*

1. *your choices for W and[10] ρ, with some brief motivations;*

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 poles allocation strategy you want to use in the forthcoming tasks, and why.*

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

Reading 4.8

- *Section 7.7*

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

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad \Longrightarrow \qquad \begin{bmatrix} y_{x_w} \\ y_{\theta_b} \end{bmatrix} = C \begin{bmatrix} x_w \\ \dot{x}_w \\ \theta_b \\ \dot{\theta}_b \end{bmatrix} \tag{23}$$

The tasks are the following ones:

Task 4.8.1

1. *design a full order Luenberger observer assuming C as in* (23)*. Use directly a poles allocation method where the poles of the observer are from 2 to 6 times faster than the poles of the closed loop system computed in Task 4.7.1;*
2. *design a reduced order state Luenberger observer assuming C as in* (23)*, and assuming that the measurement y_{x_w} is accurate, while y_{θ_b} is not. Use again a poles allocation method where the poles of the observer are from 2 to 6 times faster than the slowest poles of the closed loop system computed in Task 4.7.1.*

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[11] (or, alternatively, load these variables from the workspace if you saved them in the previous tasks):

[11] 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.

- `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 (23) 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).

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 novel system matrices. Notice that after the design step one can split the novel 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

- $\dot{\widehat{\boldsymbol{\chi}}}' = M_1\widehat{\boldsymbol{\chi}} + M_2 u + M_3\boldsymbol{y}_{\text{acc}} + M_4\boldsymbol{y}_{\overline{\text{acc}}}$

- $\widehat{\boldsymbol{\chi}} = \widehat{\boldsymbol{\chi}}' + M_5 \boldsymbol{y}_{\text{acc}}$
- $\widehat{\boldsymbol{x}} = M_6 \boldsymbol{y}_{\text{acc}} + M_7 \widehat{\boldsymbol{\chi}}$.

Reporting 4.8.1 *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 $\max_t \left|\theta_b(t) - \widehat{\theta}_b^{full}(t)\right|$ and $\max_t \left|\theta_b(t) - \widehat{\theta}_b^{reduced}(t)\right|$, and the same quantities for x_w).*

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

Reading 4.9

- *Section 8.2*
- *Section 8.3*

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 4.9.1

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 4.9.1 *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).*

4.10 Test the control strategy with the real robot

Reading 4.10 *No readings here!*

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

Task 4.10.1

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;*
   (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 4.4.*

Reporting 4.10.1 *Report an example of the x_w, θ_b and u obtained with your best controller, and which type of observer you used.*

Troubleshooting:

- "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 16. 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 4.3;
  – 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).

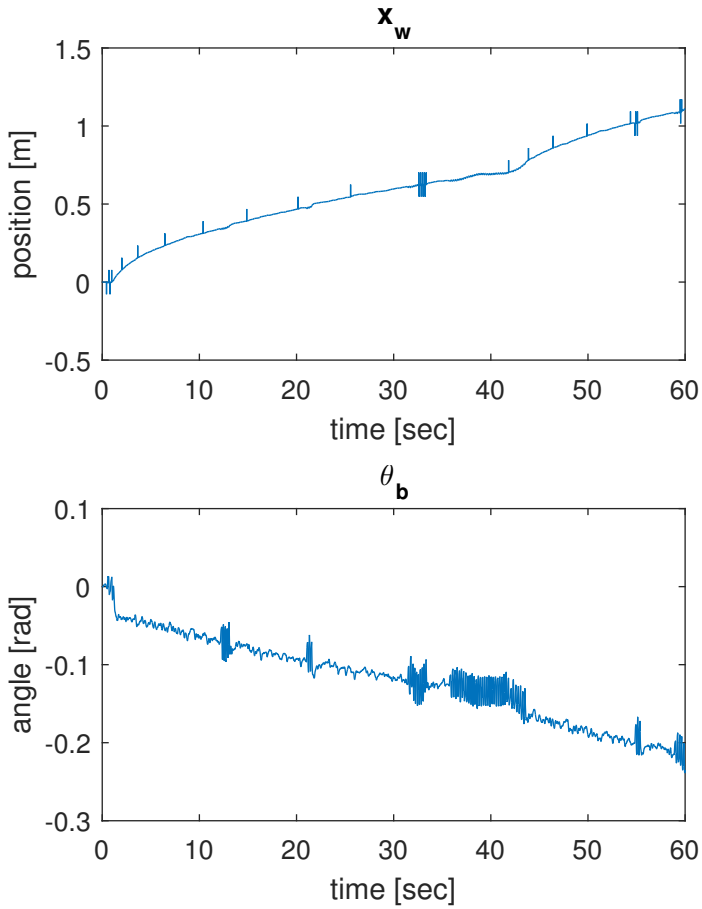


Figure 16: Example of the effect of a poorly tuned gyro: the system estimates an angle that increases / decreases with time, and compensates it by moving the wheels accordingly. The total outcome is that the robot "walks away".

5 Lab C

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

5.1 Compute the discrete equivalent of the original model

Reading 5.1

- *Section 8.5;*
- *Section W8.7 (www.fpe7e.com).*

The *first* part of the following task is equal to Task 4.10.1, thus you basically already solved it. The second no, and you will basically solve it 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 5.1.1

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 4.10.1;*
2. *find the "best sampling frequency".*

Reporting 5.1.1 *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.*

5.2 Design the controller using the discrete LQR technique

Reading 5.2

- *Section 7.6.2 (being careful that now you are working in discrete-time settings).*

This section mimics Section 4.7; here indeed we assume to know the matrix W, indicating how we weight undesirable states, and compute that K_d that minimizes of the discrete time performance index

$$\mathcal{J} = \sum_{k=1}^{+\infty} \left(\rho \begin{bmatrix} x_w(k) & \dot{x}_w(k) & \theta_b(k) & \dot{\theta}_b(k) \end{bmatrix} W \begin{bmatrix} x_w(k) \\ \dot{x}_w(k) \\ \theta_b(k) \\ \dot{\theta}_b(k) \end{bmatrix} + u^2(k) \right) \tag{24}$$

Once again, both W and ρ are design parameters that you can choose. Since you already solved this selection in Section 4.7, here we use the same choices of before. Notice that, as before, you will probably re-do this task several times, one for each sampling period that you will test.

Task 5.2.1

1. *start from a sampling frequency of 200Hz, and compute* (A_d, B_d, C_d, D_d) *as in Section 5.1*
2. *compute the LQR gain* K_d *minimizing the cost* (24) *by using the Matlab function* `dlqr`*;*
3. *insert in the Matlab script* `LabB_ControllerOverSimulator_Discrete_Parameters.m` *the novel value for* K_d*, stored in the variable* `Kd`*, and simulate once again the Simulink diagram* `LabB_ControllerOverSimulator_Discrete.slx` *(i.e., do as you did in Task 4.9.1 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 5.2.1 *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)*.*

5.3 Design the observer starting from the previously constructed controller

Reading 5.3 *No readings here!*

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 poles 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 domain consider then the following hint: we know that $p_z = e^{p_s \Delta}$, so that

$$p_z = e^{p_s \Delta} \quad \Longrightarrow \quad p_s = \frac{\ln(p_z)}{\Delta}. \tag{25}$$

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

$$p'_z = e^{p'_s \Delta} = e^{\chi p_s \Delta} \quad \Longrightarrow \quad \chi p_s = \frac{\ln(p'_z)}{\Delta} \quad \Longrightarrow \quad \chi \frac{\ln(p_z)}{\Delta} = \frac{\ln(p'_z)}{\Delta}. \tag{26}$$

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

Task 5.3.1

1. *find the formula that describes how to make a discrete pole χ times faster;*
2. *start again from a sampling frequency of 200Hz, and compute (A_d, B_d, C_d, D_d) as in Section 5.1*
3. *choose χ as you did in Task 4.8.1, 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 4.9.1);*
4. *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`*;*
5. *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 5.3.1 *Report:*

1. *the formula that describes how to make a discrete pole χ times faster;*
2. *the sampling frequency for which you lose stability;*
3. *a (brief) discussion on why the critical sampling frequency found now is different from that one found in Task 5.2.1;*
4. *the corresponding values for K_d, L_d, $M_{1d}, \ldots, M_{7d}$ and (A_d, B_d, C_d, D_d).*

5.4 Perform experiments on the real balancing robot

Reading 5.4 *No readings here!*

You have now computed all the information that you already computed in Task 4.9.1, 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 4.10.1, with the aim of checking how different sampling frequencies affect the real balancing robot.

Task 5.4.1

1. *Start from the critical sampling frequency that you computed in Task 5.3.1;*

2. *repeat the same tests you performed in Task 4.10.1 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 x_w and θ_b signals that you obtain.*

Reporting 5.4.1 *Report:*

1. *the first sampling frequency for which your real robot balances;*

2. *a graph reporting how the L_2 norms 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.*

5.5 Design a module for managing external references

Reading 5.5 *No readings here!*

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 17, where the reference enters as a feedforward term so that the poles of the observer are canceled out.

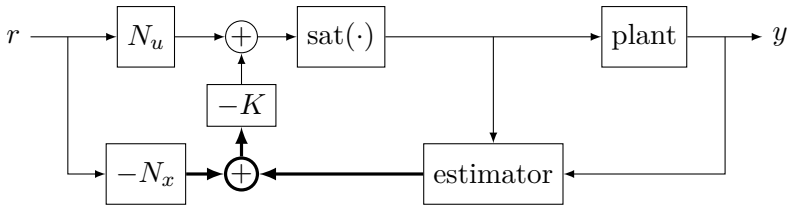


Figure 17: Chosen way of introducing the reference input.

The problem is now to compute the values for N_u and N_x that ensure that the DC gain from r to y is equal to 1. We recall that in continuous time frameworks the formula that guarantees this condition is

$$\begin{bmatrix} N_x \\ N_u \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} \tag{27}$$

with C here being $C = [1\ 0\ 0\ 0]$, i.e., that C corresponding 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.

Shall we then use formula (27) for our discrete time system? No, obviously, since in discrete time settings the condition for guaranteeing a DC gain equal to 1 starts from a different definition of equilibrium. Indeed for discrete time systems the equilibrium is:

$$\begin{aligned} N_{xd}y_{ss} &= A_d N_{xd} y_{ss} + B_d N_{ud} y_{ss} \\ y_{ss} &= C_d N_{xd} y_{ss} + D_d N_{ud} y_{ss}. \end{aligned}$$

To derive the formula you can then modify opportunely the computations performed in class.

Task 5.5.1

1. *derive the formula for N_x and N_u that guarantees the DC gain from r to y to be 1 for discrete time settings;*
2. *compute N_x and N_u, and put their values in the Matlab script* `LabC_CompensatorOverRobot_Parameters.m`*, in the variables* `Nxd` *and* `Nud`*, respectively. 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 5.5.1 *Report:*

1. *the formula for N_x and N_u for the discrete time system, plus the numerical values for these variables that you obtained applying that formula;*
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 here 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 5.1.1.

6 Final demo

The demo, that can be performed only after you finished lab C, evaluates numerically the performance of your controller. This evaluation has then two different purposes:

1. for us, to evaluate your results;
2. for you, to be motivated to do your best.

6.1 What it is

Assume that you completed Lab C; then 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 thus the demo will be composed of two steps:

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

6.2 What you are supposed to do

Follow this checklist:

1. do the following experiments with your balancing robot:
   (a) apply the reference signal (t in seconds, $r(t)$ in meters):

$$r(t) = \begin{cases} 0 & t \in [0, 10) \\ 0.05(t-10) & t \in [10, 20) \\ 0.5 & t \in [20, 30] \end{cases} \tag{28}$$

   (b) apply the reference signal (t in seconds, $r(t)$ in meters):

$$r(t) = \begin{cases} 0 & t \in [0, 10) \\ r_{\max}(t-10) & t \in [10, 20) \\ 10 r_{\max} & t \in [20, 30] \end{cases} \tag{29}$$

   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 (see Section 8.2 for more information);
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 (2 digits!), see Section 8). 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 (28) (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 (28) (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 (28) (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 (28) (in Volts);
- `group_#MYGROUP_experiment_2_times`, a *column* vector containing the sampling instants of the data relative to $r(t)$ as in Equation (29) (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 (29) (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 (29) (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 (29) (in Volts);
- `group_#MYGROUP_r_max`, a scalar containing which $r_{\max}$ you found for the experiment relative to $r(t)$ as in Equation (29) (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.

6.3 How you will be evaluated

The demo is not subject to grading, but follows instead a *go / no go* concept. More precisely, you get a *go* when all the following requirements are simultaneously met:

- the results in your `group_#MYGROUP_results.mat` show that your balancing robot is behaving as we expect it to behave (e.g., be stable, have a sufficiently big $r_{\max}$, be sufficiently fast, etc.);
- the data in your `group_#MYGROUP_results.mat` reproduces the data that we get when applying your `group_#MYGROUP_Simulink.slx` to our unperturbed balancing robot.

6.4 The R7003E 2017 LP2 International Balancing Robot Race

If you can and want, you can participate to our **R7003E 2017 LP2 International Balancing Robot Race**!

How does it work?

The race will happen on Monday January 15 2018, 10:00 Central European Time in the Control Engineering Group lab (and everybody is invited). Prior starting the race, you have to upload a code on your balancing robot so that it will go forward as fast as possible (thus, come to the race with your robot already prepared). We will then check how fast each robot is by manually measuring the time the robots take to do a 4 meters long straight path (we will do 2 trials per robot and then take the best time). The 3 fastest robots will then compete in the final run, where the 1st, 2nd and 3rd prizes will be assigned. If your balancing robot is the winning one, then you will get the utterly prestigious prize shown in Figure 18 (and there are prizes also for the second and third fastest balancing robot!).

Figure 18: The prototype for the 1^{st} prize of the R7003E 2017 LP2 International balancing robot Race. Colors and text still subject to changes!

7 Optional tasks

If you had fun up to now, then you may want to do also some other tasks (that are not just very fun to do, but that also lead to learning very useful skills). Notice: no additional points for completing these tasks, but we ensure you that you will get a lot of gratification. Some of the additional tasks are actually quite hard; in case you are interested we can make them become a Master Thesis project.

- make the balancing robot start moving only when "poked";
- make the balancing robot stop moving when "poked" once again;
- add an other motor to the balancing robot, and add the possibility of steering it;
- visualize the motion of the balancing robot through our Vicon MoCap system;
- write a phone app so that you get the data from the balancing robot via Bluetooth;
- write a phone app so that you can draw a path in your phone and make the robot follow it;
- add a camera to the balancing robot and make it an exploring balancing robot.

8 Reporting your findings

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. **Clear?**

8.1 The labs reports

8.1.1 What they are

They are very brief documents that you compile by group (thus one report for each group) and that you upload in Canvas. 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 (specially if you automate the production of the figures).

8.1.2 What you are supposed to do

For every report (A, B or C), follow this checklist:

1. download the corresponding template from http://staff.www.ltu.se/~damvar/R7003E-2017-LP2.html;
2. compile the templates following the instructions that they contain;
3. name the compiled template as `group_#MYGROUP_lab_#LAB.pdf`, with `#MYGROUP` being your two digits group code, and `#LAB` being either `A`, `B` or `C`, accordingly;
4. upload the template in Canvas (notice that it is the first time that we teachers are using Canvas so we have still to understand how this works; nonetheless we will solve this issue soon and inform accordingly).

Important: the reports **must be** in a `.pdf` format. If you do things in Word or stuff like that then export your file in a `.pdf`. We will **not** evaluate reports that are not in a `.pdf` format. Thus "no .pdf" $\implies$ failing the lab.

8.1.3 How you will be evaluated

First of all, every report should be complete: we will not evaluate documents that miss reporting things that have been asked to be reported. Second, reports should comply with the given template. For example, if the maximum number of pages allowed is 6, then you have to submit a document that is at most 6 pages. 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), with 60 indicating the sufficiency. If you get an insufficiency in some report then you will have to improve it up to the moment that you get a grading ≥ 60.

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

correctness of the results, for a total of 50 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 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.

8.2 The final report

8.2.1 What it is

It is a document that you send us via Canvas, and that is used not just to evaluate you, but also to train you writing scientific articles.

The final report has to be 6 pages long and follow the `ieeeconf` template. Not one more / one less page, not other fonts or formatting style rather than the indicated ones. Not complying with this $\implies$ having to fix and then re-submit + making the teachers quite upset.

8.2.2 What you are supposed to do

Follow this checklist:

1. download the corresponding template from http://staff.www.ltu.se/~damvar/R7003E-2017-LP2.html;
2. **compile it following the hints that it contains**. 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 two digits group code;
4. upload it in Canvas.

Important (en gång till): the reports **must be** in a `.pdf` format. If you do things in Word or stuff like that then export your file in a `.pdf`. We will not evaluate reports that are not in a `.pdf` format.

8.2.3 How you will be evaluated

Like as in Section 8.1.3, plus an eye for your capacity of understanding what is important and what is not. Notice that the final report will contain information on your experiments for the demo.

9 Appendix

9.1 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).

9.2 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`).

9.3 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 Figures7 and 8 we used a `.m` file containing:

```
close all; clear all; 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.

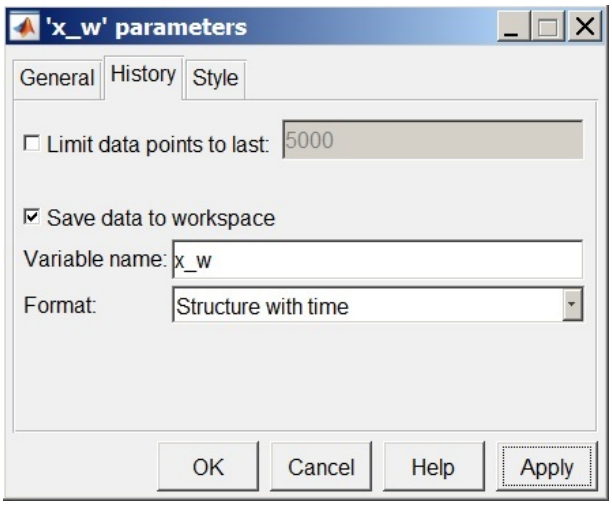


Figure 19: How to configure a scope in Simulink so that it will save your data on the workspace.

9.4 Useful Simulink tricks

The "tricks" presented in Table 1 want to help you keeping your diagrams organized and easy to be debugged.

9.5 Useful Simulink blocks

The blocks presented in Table 2 instead want to help you be faster in creating your diagrams.

9.6 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:

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 1: Useful Simulink tricks.

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 2: Useful Simulink blocks.

- 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`.

9.7 Useful Matlab commands

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

9.8 Useful bash scripts

Matlab notoriously produces "bad" `.eps` files. In this document we cropped all the `.eps` files into `.pdf` ones through the following bash script (*of course* we are using Linux[12]):

```
#!/bin/bash

# for every file
```

[12]When making documents. Unfortunately this year we have Simulink libraries for communicating with the robots that work only under Windows (and this bothers us quite a lot, actually). Porting them to other OSs is one of our priorities for 2017, though.

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.

```
for originalFileInEps in *.eps
do
# find the names of the various files
nonCroppedFileInPdf=${originalFileInEps%.eps}.pdf
croppedFileInPdfWithBadName=${nonCroppedFileInPdf%.pdf}-crop.pdf
croppedFileInPdfWithGoodName=$nonCroppedFileInPdf

# perform the cropping and name changing operations
epstopdf $originalFileInEps
pdfcrop $nonCroppedFileInPdf
mv $croppedFileInPdfWithBadName $croppedFileInPdfWithGoodName
done
```

9.9 Contacts

- Damiano Varagnolo, room = A2550 (campus Luleå), email = `damiano.varagnolo@ltu.se`, Skype = damianovar, phone = +46720216047;
- Riccardo Lucchese, room A2569 (campus Luleå), email = `riccardo.lucchese@ltu.se`, phone = +46739923014.

9.10 List of the acronyms

EOM Equations of Motion

DC Direct Current

SS State-Space

LQR Linear-Quadratic Regulator

ZOH Zero Order Hold

SRL Symmetric Root Locus

MIMO Multi Input Multi Output

9.11 Datasheet

quantity	nominal value
g	9.8 m / s^2
b_f	0
m_b	0.381 Kg (with batteries)
l_b	0.112 m
I_b	0.00616 Kg m^2
m_w	0.036 Kg
l_w	0.021 m
I_w	0.00000746 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 4: Notation used in this document (and in your reports).

9.12 Notation

symbol	meaning	unit
x_b	horizontal position of the center of mass of the body	cm
y_b	vertical position of the center of mass of the body	cm
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	cm
x_w	horizontal position of the center of mass of the wheel	cm
y_w	vertical position of the center of mass of the wheel	cm
θ_w	angular displacement of the wheel	radians
m_w	mass of the wheel	kg
l_w	radius of the wheel	cm
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 5: Notation used in this document (and in your reports).

10 FAQ

- once I computed the EOM, how can I be sure that they are correct? - Which criterion use to choose the poles? - the θ_b measured by the gyroscope changes and "decreases" or "increases" in time even if the robot is not moving; why? - I have numbers like "something $\cdot e^{-15}$" or stuff like that. Is it a sign of some problems? - I measured something from the robot; how can I know if I am doing things correctly?

1. When formulating the state space equations, does the order of the state vectors matter in the state vector? I remember that I had formulated the state vector differently than you and therefore had a different matrix for A and B.

2. What are some general recommendations about placing the poles using pole allocation method for the PID controller?

3. When converting the PID controller to the discrete domain, how can you calculate the transfer function C(z) of the controller? Do you simply copy the formula from the Simulink PID block once you click 'discrete-time'? There are also formulae in the textbook for converting to a ZOH discrete time transfer function, but they are somewhat confusing.

4. Even after doing the bias step, my measurements don't look like those in section 4.3 of the lab manual for the gyroscope. Is this normal? I now realize the gyro is fine, as it works with state space control, but maybe there was an extra step of Matlab code to get the unbiased signal displayed for these figures?

5. Should you expect your PID to be able to balance the robot for close to 10 seconds? I now realize it's possible, but not without a lot of manual tuning.

1. The gain, check negative feedback. 2. The Nyquist plots. We had a bit of trubble understaning our resulting Nyquist plot and what we where supposed to expect. 3. In retrospect we might have had needed some intuition on how the matrices A,B,C and D relates to the EOMs so that we could see if we have the correct equations or not. Perheps where to expect values and where not to. 3. The pole allocation method. One tip we found usefull and helped us was to do it symbolically. 4. Tips and/or pointers on how one can determine the gain, K-value.