Laboratory activity 4: Longitudinal state–space control of the balancing robot

(Laboratory assignments)

Riccardo Antonello* Francesco Ticozzi*

May 13, 2025

## 1 Activity goal

The purpose of this laboratory activity is to design and test a longitudinal state–space controller for the two–wheeled balancing robot (also referred as "two–wheeled inverted pendulum robot", or "Segway–like robot") available in laboratory. The controller is designed to simultaneously stabilize the robot body to its upward vertical position, and the robot base to a desired longitudinal position set-point. The design is performed by resorting to a simplified model of the robot dynamics, obtained by assuming that the motion occurs along a straight line (i.e. the lateral or heading–angle dynamics is ignored).

**There is no challenge for this LAB.**

*Dept. of Information Engineering (DEI), University of Padova; email: {antonello, ticozzi}@dei.unipd.it

# 4 Laboratory assignments: numerical simulations

## 4.1 Simulink model of the balancing robot (longitudinal dynamics only)

**(S1)** Implement a Simulink model of the nonlinear electromechanical dynamics of the balancing robot derived in Sec. 2.1 and 2.2 – see (54)–(55). A possible Simulink implementation is shown in Fig. 1. It basically consists of rewriting (54) as follows:

$$\ddot{\boldsymbol{q}} \;=\; \boldsymbol{M}^{-1}(\boldsymbol{q})\left[-\boldsymbol{C}(\boldsymbol{q},\,\dot{\boldsymbol{q}})\,\dot{\boldsymbol{q}} \,-\, \boldsymbol{F}_v'\,\dot{\boldsymbol{q}} \,-\, \boldsymbol{g}(\boldsymbol{q}) \,+\, \boldsymbol{\tau}'\right] \tag{1}$$

and then using the "chain of integrators" approach to derive an equivalent block diagram representation of the (nonlinear) differential equation. The state–dependent matrices $\boldsymbol{M}(\boldsymbol{q})$, $\boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}})$ and $\boldsymbol{g}(\boldsymbol{q})$ can be implemented as shown in Fig. 1b–1d. In particular:

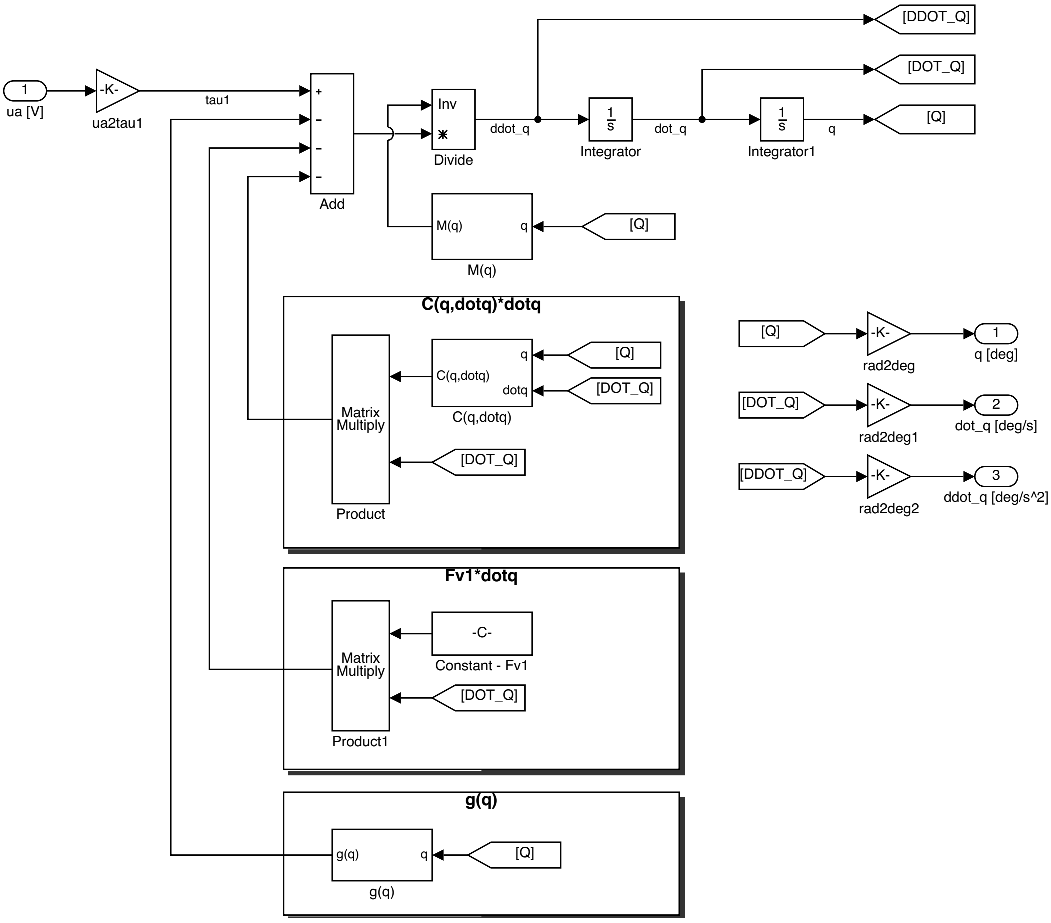
- use the `User-Defined Functions` → `Fcn` block to implement the state–dependent elements of the aforementioned matrices. For the state–independent (i.e. constant) elements, use the `Sources` → `Constant` block.
- use the `Signal Routing` → `Mux` block to combine the single matrix elements into matrix columns.
- use the `Math Operations` → `Matrix Concatenate` block to form a single matrix by concatenation of its columns.
- use the `Math Operations` → `Product` block to multiply a matrix by a column vector. In the block options, select `Matrix` among the `Multiplication` options: this instructs Simulink to compute the "row–by–column" product, instead of the conventional "element–by–element" product.

The torque input $\boldsymbol{\tau}'$ can be obtained by multiplying the scalar input $u_a$ by the column vector gain $(2Nk_t/R_a)\,[1,\,-1]^T$.

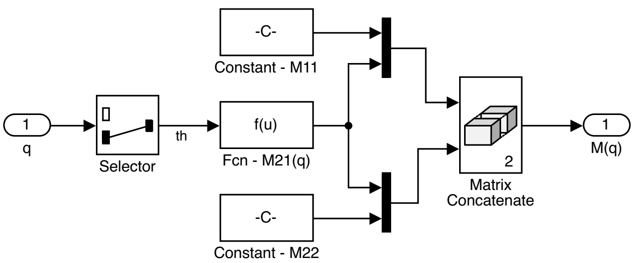
An alternative Simulink implementation of the nonlinear dynamics (54)–(55) consists of using MATLAB *S-Functions*. A possible implementation based on S–Function is reported in Appendix 6.1.

**(S2)** Implement a Simulink model of the Motion Processing Unit (MPU), according to the mathematical model derived in Sec. 2.3 – see (64)–(65). A possible Simulink implementation is shown in Fig. 2. In particular:

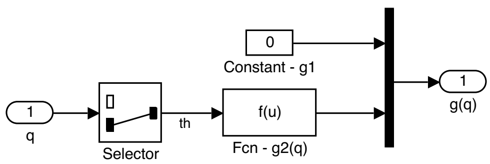
- use the `User-Defined Functions` → `Fcn` block to compute the $x$ and $z$ axes components of the accelerometer output vector (64).
- use a `Discrete` → `Zero-Order Hold` block to sample the accelerometer and gyroscope outputs with a sampling time equal to `Ts` (see line 4 in Listing 1), which corresponds to the controller sampling time $T = 0.01\,\mathrm{s}$.
- the accelerometer outputs are in $[\mathfrak{g}]$ units ($1\,\mathfrak{g} = 9.81\,\mathrm{m/s^2}$). Therefore, use a `ms22g` gain to convert the accelerometer outputs from $[\mathrm{m/s^2}]$ to $[\mathfrak{g}]$ units. On the other hand, the gyroscope output is in $[\mathrm{deg/s}]$ units, so that if the gyroscope input is expressed in such units, no extra units conversion is required.
- use a `Sources` → `Random Number` block to model the normally (Gaussian) distributed noise affecting the accelerometer and gyroscope outputs. Use `sens.mpu.acc.noisevar` and `sens.mpu.`



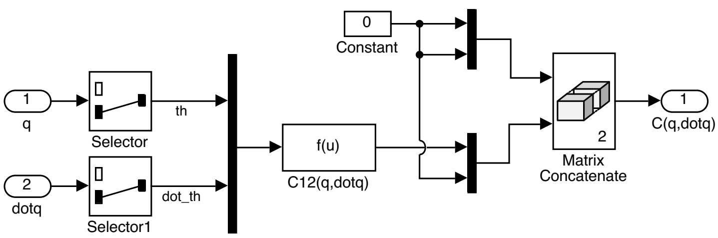

(a) Electromechanical dynamics




(b) $\boldsymbol{M}(\boldsymbol{q})$ matrix




(c) $\boldsymbol{g}(\boldsymbol{q})$ vector




(d) $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ matrix

Figure 1: Simulink model implementation details: electromechanical dynamics (implemented by using the "chain of integrators" approach).

gyro.noisevar (see lines 210 and 223 in Listing 1) as the noise variances for, respectively, the accelerometer and gyroscope output noises, and Ts as the sampling time.

- use a Discontinuities $\to$ Quantizer block to model the finite resolution of the accelerometer and gyroscope outputs. Use sens.mpu.acc.LSB2g and sens.mpu.gyro.LSB2degs (see line 207 and 219 of Listing 1) as the quantization steps for, respectively, the accelerometer and gyroscope output quantizations.

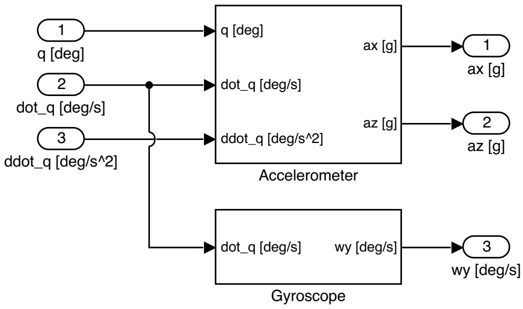
**(S3)** Implement a Simulink model of the incremental encoder used to measure the motor shaft position. A possible Simulink implementation is shown in Fig. 3. In particular:

- the incremental encoder measures the angular displacement $\Delta\vartheta_{rot} = \vartheta_{rot} - \vartheta$ of the rotor with respect to the stator (rigidly connected to the robot chassis). It holds that $\Delta\vartheta_{rot} = N(\gamma - \vartheta)$, where $\gamma - \vartheta$ is the angular displacement of the wheel with respect to robot body, and $N$ is the gearbox ratio.
- use a Discrete $\to$ Zero-Order Hold block to sample the encoder output with a sampling time equal to Ts.
- use a Discontinuities $\to$ Quantizer block to model the finite resolution of the encoder. Use sens.enc.pulse2deg (see line 189 in Listing 1) as the quantization step (provided that the rotor angular displacement $\Delta\vartheta_{rot}$ is computed in $[\mathrm{deg}]$ units).
- the encoder output is in $[\mathrm{pulses}]$ units. Therefore, use a sens.enc.deg2pulse gain to convert the encoder output from $[\mathrm{deg}]$ to $[\mathrm{pulses}]$ units.

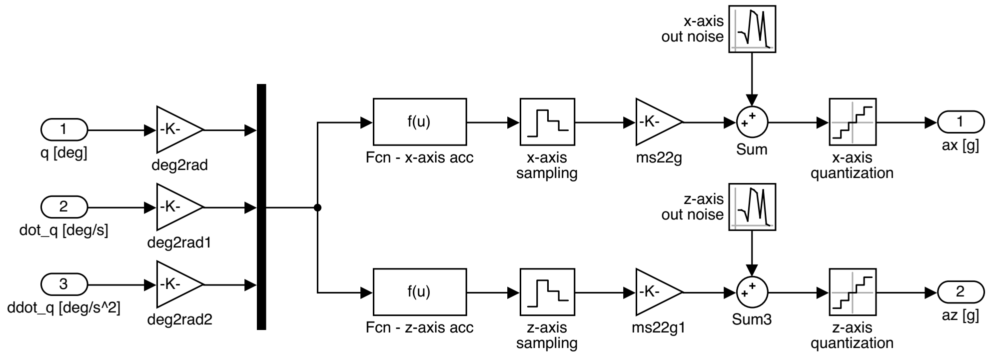
**(S4)** Implement a Simulink model of the motor voltage driver. A possible Simulink implementation is shown in Fig. 4. In particular:

- the PWM command to the voltage driver consists of a byte, plus a sign flag. Hence, the voltage driver input can be considered as an integer number in the range $[-255, 255]$. The maximum value corresponds to apply the maximum voltage to the motor armature, which is equal to the battery nominal voltage, i.e. $11.1\,\mathrm{V}$. Use a Discontinuities $\to$ Saturation block to limit the driver input to the the voltage range specified above. Use $\pm$drv.dutymax (see line 114 in Listing 1) as the saturation levels.
- use a Math Operations $\to$ Rounding Function to convert the driver input into an integer number. Select fix as the rounding method.
- the driver output is the voltage applied to the motor armature. Therefore, use a drv.duty2V gain to convert the driver voltage command into a voltage signal in $[\mathrm{V}]$ units. Since the driver output voltage can never exceed the battery nominal voltage, consider to insert a saturation block to limit the driver output. Use $\pm$drv.Vbus (see line 110 in Listing 1) as the saturation levels.
- use a Discrete $\to$ Zero-Order Hold block to hold the voltage command within each sampling period (equal to Ts).

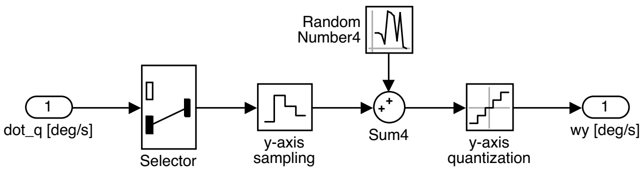
**(S5)** Combine all the models prepared in points **(S1)–(S4)**, to form a Simulink model of the whole balancing robot hardware equipment (i.e. electromechanical system + motor driver + sensors). A possible Simulink implementation is shown in Fig. 5.



(a) Motion Processing Unit (MPU)



(b) Accelerometer



(c) Gyroscope

Figure 2: Simulink model implementation details: MPU.

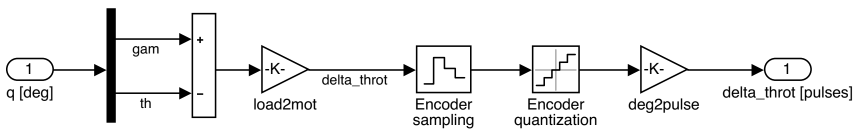


Figure 3: Simulink model implementation details: encoder.

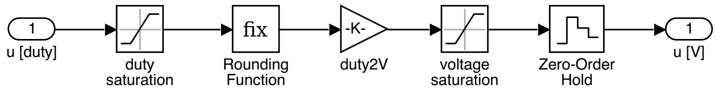


Figure 4: Simulink model implementation details: voltage driver.

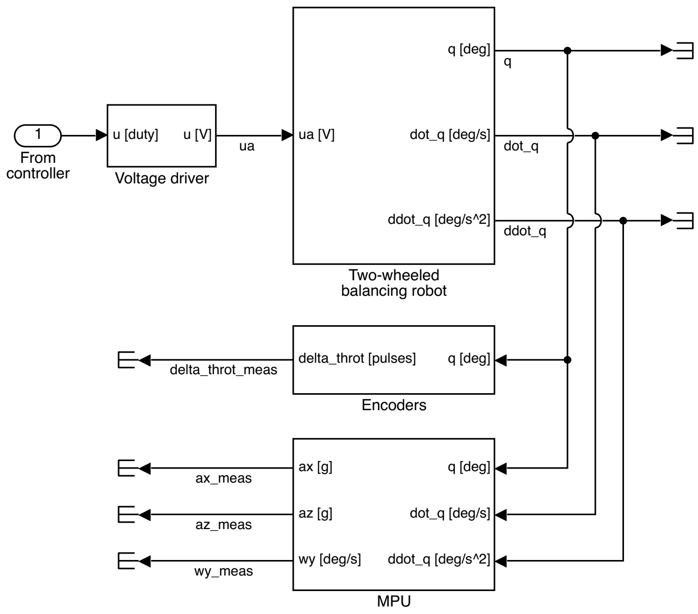



Figure 5: Simulink model implementation details: overall balancing robot hardware equipment.

Listing 1: `balrob_params.m`

```
%% General parameters and conversion gains

%   controller sampling time
Ts = 1e-2;

%   gravity acc [m/s^2]
g = 9.81;

%   conversion gains
rpm2rads = 2*pi/60;         %   [rpm]   -> [rad/s]
rads2rpm = 60/2/pi;         %   [rad/s] -> [rpm]
rpm2degs = 360/60;          %   [rpm]   -> [deg/s]
degs2rpm = 60/360;          %   [deg/s] -> [rpm]
deg2rad = pi/180;           %   [deg]   -> [rad]
rad2deg = 180/pi;           %   [rad]   -> [deg]
g2ms2 = g;                  %   [acc_g] -> [m/s^2]
ms22g = 1/g;                %   [m/s^2] -> [acc_g]
ozin2Nm = 0.706e-2;         %   [oz*inch] -> [N*m]

%   robot initial condition
x0 =[ ...
    0, ...                  %   gam(0)
    5*deg2rad, ...          %   th(0)
    0, ...                  %   dot_gam(0)
    0];                     %   dot_th(0)

%% DC motor data

%   motor id: brushed DC gearmotor Pololu 30:1 37Dx68L mm

%   electromechanical params
mot.UN   = 12;                        %   nominal voltage
mot.taus = 110/30 * ozin2Nm;          %   stall torque @ nom voltage
mot.Is   = 5;                         %   stall current @ nom voltage
mot.w0   = 350 * 30 * rpm2rads;       %   no-load speed @ nom voltage
mot.I0   = 0.3;                       %   no-load current @ nom voltage

```

```
mot.R    = mot.UN/mot.Is;                        %   armature resistance
mot.L    = NaN;                                  %   armature inductance
mot.Kt   = mot.taus/mot.Is;                      %   torque constant
mot.Ke   = (mot.UN - mot.R*mot.I0)/(mot.w0);     %   back-EMF constant
mot.eta  = NaN;                                  %   motor efficiency
mot.PN   = NaN;                                  %   nominal output power
mot.IN   = NaN;                                  %   nominal current
mot.tauN = NaN;                                  %   nominal torque

%   dimensions
mot.rot.h   = 30.7e-3;                           %   rotor height
mot.rot.r   = 0.9 * 17e-3;                       %   rotor radius

mot.stat.h  = 68.1e-3;                           %   stator height
mot.stat.r  = 17e-3;                             %   stator radius

%   center of mass (CoM) position
mot.rot.xb  = 0;                                 %   (left) rot CoM x-pos in body frame
mot.rot.yb  = 42.7e-3;                           %   (left) rot CoM y-pos in body frame
mot.rot.zb  = -7e-3;                             %   (left) rot CoM z-pos in body frame

mot.stat.xb = 0;                                 %   (left) stat CoM x-pos in body frame
mot.stat.yb = 52.1e-3;                           %   (left) stat CoM y-pos in body frame
mot.stat.zb = -7e-3;                             %   (left) stat CoM z-pos in body frame

%   mass
mot.m       = 0.215;                             %   total motor mass
mot.rot.m   = 0.35 * mot.m;                      %   rotor mass
mot.stat.m  = mot.m - mot.rot.m;                 %   stator mass

%   moment of inertias (MoI) wrt principal axes
mot.rot.Ixx  = mot.rot.m/12 * (3*mot.rot.r^2 + mot.rot.h^2);    %   MoI along r dir
mot.rot.Iyy  = mot.rot.m/2 * mot.rot.r^2;                       %   MoI along h dim
mot.rot.Izz  = mot.rot.Ixx;                                     %   MoI along r dir

mot.stat.Ixx = mot.stat.m/12 * (3*mot.stat.r^2 + mot.stat.h^2); %   MoI along r dir
mot.stat.Iyy = mot.stat.m/2 * mot.stat.r^2;                     %   MoI along h dir
mot.stat.Izz = mot.stat.Ixx;                                    %   MoI along r dir

%   viscous friction coeff (motor side)
mot.B  = mot.Kt*mot.I0/mot.w0;

%%  Gearbox data

gbox.N = 30;        %   reduction ratio
gbox.B = 0.025;     %   viscous friction coeff (load side)

%%  Battery data

%   electrical data
batt.UN = 11.1;     %   nominal voltage

%   dimensions
batt.w = 136e-3;    %   battery pack width
batt.h = 26e-3;     %   battery pack height
batt.d = 44e-3;     %   battery pack depth

%   center of mass (CoM) position
batt.xb = 0;        %   CoM x-pos in body frame
batt.yb = 0;        %   CoM y-pos in body frame
batt.zb = 44e-3;    %   CoM z-pos in body frame

%   mass
```

```
batt.m = 0.320;

%   moment of inertias (MoI) wrt principal axes
batt.Ixx = batt.m/12 * (batt.w^2 + batt.h^2);   %   MoI along d dim
batt.Iyy = batt.m/12 * (batt.d^2 + batt.h^2);   %   MoI along w dim
batt.Izz = batt.m/12 * (batt.w^2 + batt.d^2);   %   MoI along h dim

%%  H-bridge PWM voltage driver data

drv.Vbus = batt.UN;                     %   H-bridge DC bus voltage
drv.pwm.bits = 8;                       %   PWM resolution [bits]
drv.pwm.levels = 2^drv.pwm.bits;        %   PWM levels
drv.dutymax = drv.pwm.levels-1;         %   max duty cycle code
drv.duty2V = drv.Vbus/drv.dutymax;      %   duty cycle code (0-255) to voltage
drv.V2duty = drv.dutymax/drv.Vbus;      %   voltage to duty cycle code (0-255)

%%  Wheel data

%   dimensions
wheel.h = 26e-3;                %   wheel height
wheel.r = 68e-3/2;              %   wheel radius

%   center of mass (CoM) position
wheel.xb = 0;                   %   (left) wheel CoM x-pos in body frame
wheel.yb = 100e-3;              %   (left) wheel CoM y-pos in body frame
wheel.zb = 0;                   %   (left) wheel CoM z-pos in body frame

%   mass
wheel.m = 50e-3;

%   moment of inertias (MoI) wrt principal axes
wheel.Ixx = wheel.m/12 * (3*wheel.r^2 + wheel.h^2);     %   MoI along r dim
wheel.Iyy = wheel.m/2 * wheel.r^2;                      %   MoI along h dim
wheel.Izz = wheel.Ixx;                                  %   MoI along r dim

%   viscous friction coeff
wheel.B = 0.0015;

%%  Chassis data

%   dimensions
chassis.w = 160e-3;       %   frame width
chassis.h = 119e-3;       %   frame height
chassis.d = 80e-3;        %   frame depth

%   center of mass (CoM) position
chassis.xb = 0;           %   CoM x-pos in body frame
chassis.yb = 0;           %   CoM x-pos in body frame
chassis.zb = 80e-3;       %   CoM x-pos in body frame

%   mass
chassis.m = 0.456;

%   moment of inertias (MoI) wrt principal axes
chassis.Ixx = chassis.m/12 * (chassis.w^2 + chassis.h^2);   %   MoI along d dim
chassis.Iyy = chassis.m/12 * (chassis.d^2 + chassis.h^2);   %   MoI along w dim
chassis.Izz = chassis.m/12 * (chassis.w^2 + chassis.d^2);   %   MoI along h dim

%%  Body data

%   mass
body.m = chassis.m + batt.m + 2*mot.stat.m;

```

```
%   center of mass (CoM) position
body.xb = 0;                                            %   CoM x-pos in body frame
body.yb = 0;                                            %   CoM y-pos in body frame
body.zb = (1/body.m) * (chassis.m*chassis.zb + ...      %   CoM z-pos in body frame
    batt.m*batt.zb + 2*mot.stat.m*mot.stat.zb);

%   moment of inertias (MoI) wrt principal axes
body.Ixx = chassis.Ixx + chassis.m*(body.zb - chassis.zb)^2 + ...       %   MoI along d dim
    batt.Ixx + batt.m*(body.zb - batt.zb)^2 + ...
    2*mot.stat.Ixx + ...
    2*mot.stat.m*(mot.stat.yb^2 + (body.zb - mot.stat.zb)^2);

body.Iyy = chassis.Iyy + chassis.m*(body.zb - chassis.zb)^2 + ...       %   MoI along w dim
    batt.Iyy + batt.m*(body.zb - batt.zb)^2 + ...
    2*mot.stat.Iyy + ...
    2*mot.stat.m*(body.zb - mot.stat.zb)^2;

body.Izz = chassis.Izz + batt.Izz + ...                                 %   MoI along h dim
    2*mot.stat.Izz + 2*mot.stat.m*mot.stat.yb^2;

%%  Sensors data - Hall-effect encoder

%   Hall-effect encoder
sens.enc.ppr = 16*4;    %   pulses per rotation at motor side (w/ quadrature decoding)
sens.enc.pulse2deg = 360/sens.enc.ppr;
sens.enc.pulse2rad = 2*pi/sens.enc.ppr;
sens.enc.deg2pulse = sens.enc.ppr/360;
sens.enc.rad2pulse = sens.enc.ppr/2/pi;

%%  Sensors data - MPU6050 (accelerometer + gyro)

%   center of mass (CoM) position
sens.mpu.xb = 0;
sens.mpu.yb = 0;
sens.mpu.zb = 13.5e-3;

%       MPU6050 embedded accelerometer specs
sens.mpu.acc.bits = 16;
sens.mpu.acc.fs_g = 16;                                                 %   full-scale in "g" units
sens.mpu.acc.fs = sens.mpu.acc.fs_g * g2ms2;                            %   full-scale in [m/s^2]
sens.mpu.acc.g2LSB = floor(2^(sens.mpu.acc.bits-1)/sens.mpu.acc.fs_g);  %   sensitivity [LSB/g]
sens.mpu.acc.ms22LSB = sens.mpu.acc.g2LSB * ms22g;                      %   sensitvity [LSB/(m/s^2)]
sens.mpu.acc.LSB2g = sens.mpu.acc.fs_g/2^(sens.mpu.acc.bits-1);         %   out quantization [g/LSB]
sens.mpu.acc.LSB2ms2 = sens.mpu.acc.LSB2g * g2ms2;                      %   out quantization [ms2/LSB]
sens.mpu.acc.bw = 94;                                                   %   out low-pass filter BW [Hz]
sens.mpu.acc.noisestd = 400e-6*sqrt(100);                               %   output noise std [g-rms]
sens.mpu.acc.noisevar = sens.mpu.acc.noisestd^2;                        %   output noise var [g^2]

%       MPU6050 embdedded gyroscope specs
sens.mpu.gyro.bits = 16;
sens.mpu.gyro.fs_degs = 250;                                            %   full scale in [deg/s (dps)]
sens.mpu.gyro.fs = sens.mpu.gyro.fs_degs * deg2rad;                     %   full scale in [rad/s]
sens.mpu.gyro.degs2LSB = floor(2^(sens.mpu.gyro.bits-1)/sens.mpu.gyro.fs_degs); %   sensitivity [LSB/degs]
sens.mpu.gyro.rads2LSB = sens.mpu.gyro.degs2LSB * rad2deg;              %   sensitivity [LSB/rads]
sens.mpu.gyro.LSB2degs = sens.mpu.gyro.fs_degs/2^(sens.mpu.gyro.bits-1);    %   out quantization [degs/LSB]
sens.mpu.gyro.LSB2rads = sens.mpu.gyro.LSB2degs * deg2rad;              %   out quantization [rads/LSB]
sens.mpu.gyro.bw = 98;                                                  %   out low-pass filter BW [Hz]
sens.mpu.gyro.noisestd = 5e-3*sqrt(100);                                %   output noise std [degs-rms]
sens.mpu.gyro.noisevar = sens.mpu.acc.noisestd ^2;                      %   output noise var [degs^2]
```

## 4.2 Balance–and–position state–space control using LQR methods

**(S6)** For the implementation of a state–space balance–and–position controller, it is first necessary to estimate the robot state $\boldsymbol{x} = [\gamma,\, \vartheta,\, \dot{\gamma},\, \dot{\vartheta}]^T$ from the measurements provided by the onboard sensors, namely the incremental encoder and the MPU (accelerometer and gyroscope). For such purpose, consider to implement a "simple" state observer as follows:

– for estimating the robot body tilt angle $\vartheta$, use the complementary filtering approach described in Sec. 3. Consider to use a pair of first–order complementary filters such as (72). The filters must be discretized, since the whole control system operates in the discrete–time domain. Say $H(z)$ the discrete equivalent of $H(s)$, obtained with any discretization method. From (71) it follows that

$$\hat{\vartheta} \;=\; H(z)\,\hat{\vartheta}_a \,+\, [1 - H(z)]\,\hat{\vartheta}_g \;=\; \hat{\vartheta}_g \,+\, H(z)\,(\hat{\vartheta}_a \,-\, \hat{\vartheta}_g) \tag{2}$$

where $\hat{\vartheta}_a$ is computed as specified in (67), and $\hat{\vartheta}_g$ by discrete–time integration of the gyroscope output $y_g$. A possible Simulink implementation of the complementary filtering (2) is shown in Fig. 6a. A tentative value for the filter cut–off frequency $f_c = 1/(2\pi\, T_c)$ is $0.35\,\mathrm{Hz}$.

Notes:

a) it is worth to notice here that if both $H(s)$ and the integrator in (69) are discretized with the *Backward Euler* discretization method, then the generic implementation (2) can be further simplified as follows. The Backward Euler discretization of $H(s)$ yields

$$H(z) \;=\; \frac{C}{1 - (1 - C)\, z^{-1}} \qquad \text{with} \qquad C = \frac{T}{T_c + T} \tag{3}$$

where $T$ denotes the sampling time. Hence, from (2) it follows that

$$\hat{\vartheta} \;=\; \frac{C}{1 - (1 - C)\, z^{-1}}\,\hat{\vartheta}_a \,+\, \frac{(1 - C)\,(1 - z^{-1})}{1 - (1 - C)\, z^{-1}} \cdot \frac{T}{1 - z^{-1}}\, y_g \tag{4}$$

where it has been used the fact that $\hat{\vartheta}_g$ is obtained by integration of $y_g$, using the discrete–time integrator $T/(1 - z^{-1})$ (Backward Euler discretization of the continuous–time integrator $1/s$). In time domain, the expression (4) becomes

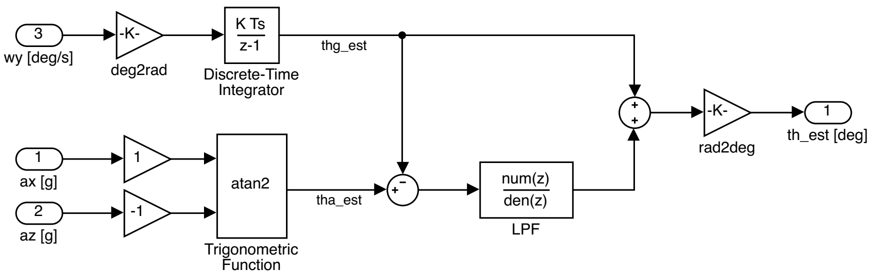
$$\begin{aligned} \hat{\vartheta}[k] \;&=\; (1 - C)\,\hat{\vartheta}[k-1] \,+\, C\,\hat{\vartheta}_a[k] \,+\, (1 - C)\,T\,y_g[k] \\ &=\; C\,\hat{\vartheta}_a[k] \,+\, (1 - C)\,\Big(\hat{\vartheta}[k-1] \,+\, T\,y_g[k]\Big) \end{aligned} \tag{5}$$

A possible Simulink implementation of the complementary filtering (5) is shown in Fig. 6b.

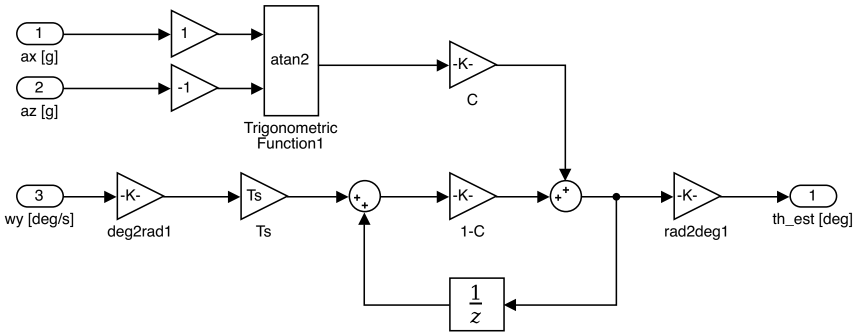
b) when using the balancing robot model implemented with the "chain of integrators" approach, an algebraic loop could arise if the complementary filters are implemented either as in Fig. 6a, and the transfer function $H(z)$ of the discretized low–pass filter is not strictly proper, or as in Fig. 6b.

In fact, in these situations, the estimated tilt angle depends instantaneously on the acceleration output of the robot model; but the robot acceleration depends instantaneously on the control command, which in turns depends instantaneously (through the multiplication by the feedback gain) from the estimated tilt angle, and hence an algebraic loop originates[1].

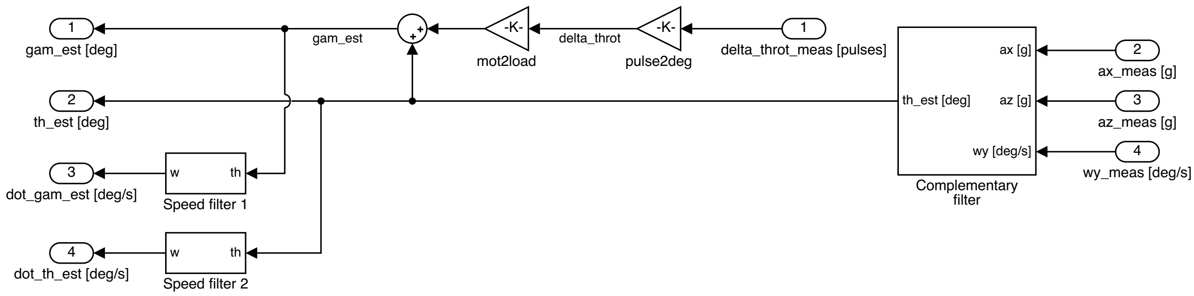
[1]The real reason for the existence of the algebraic loop is that both the models of the motor voltage driver (see




(a) Tilt estimation by complementary filtering (implementation based on (2))




(b) Tilt estimation by complementary filtering (implementation based on (5))




(c) "Simple" state observer

Figure 6: Possible Simulink implementation of the "simple" state observer.

To avoid the algebraic loop issue when using the robot model based on the "chain of integrators" implementation, consider to use the complementary filtering scheme shown in Fig. 6a, and choose a discrete–time low–pass filter with a strictly proper transfer function (e.g. the filter obtain by discretizing a continuous–time, first–order low–pass filter with the Forward Euler method).

– for estimating the wheel angle $\gamma$, use the identity $\Delta\vartheta_{rot} = N(\gamma - \vartheta)$ to obtain the estimate

$$\hat{\gamma} \;=\; \Delta\vartheta_{rot}/N \,+\, \hat{\vartheta} \tag{6}$$

where $\Delta\vartheta_{rot}$ is the rotor angular displacement measured by the encoder. A possible Simulink implementation of (6) is shown in Fig. 6c.

Fig. 4) and the inertial sensors (see Fig. 2) have no dynamics, i.e. the outputs instantaneously depend on the inputs. This idealized situation never occurs in practice, and typically both the actuators and sensors have a low-pass dynamics that should be taken into account when modelling them for simulation purposes. Indeed, this is what has been done for voltage driver model of the Quanser DC servomotor used in the previous laboratory activities. Unfortunately, in the case of the balancing robot, there is no explicit knowledge about the voltage driver bandwidth, and this was the original reason that motivated the choice of neglecting the driver dynamics in the model proposed in this handout.

– use a "real derivative" filter of the type

$$H_\omega(z) = \frac{1 - z^{-N}}{N\,T} \qquad \text{with} \qquad N = 3 \tag{7}$$

to obtain the angular speeds estimates $\dot{\hat{\vartheta}}$ and $\dot{\hat{\gamma}}$ from $\hat{\vartheta}$ and $\hat{\gamma}$.

**(S7)** Design a discrete–time state–space controller that simultaneously stabilizes the robot body to its upward vertical position, and guarantees the *nominal* perfect tracking of a constant wheel angle position set–point $\gamma^*$. For such purpose, consider first to discretize the continuous–time plant model (57) with the exact discretization method, and sampling time equal to the controller sampling time $T = 0.01\,\mathrm{s}$; let

$$\begin{cases} \boldsymbol{x}[k+1] \;=\; \boldsymbol{\Phi}\,\boldsymbol{x}[k] \,+\, \boldsymbol{\Gamma}\,u[k] \\ \qquad y[k] \;=\; \boldsymbol{H}\,\boldsymbol{x}[k] \end{cases} \tag{8}$$

denote the discretized plant model. For the design of the tracking controller, the model output $y$ has to be equal to the signal to track, namely the wheel angle position $\gamma$. Therefore, the matrix $\boldsymbol{H}$ is chosen equal to $\boldsymbol{H} = [1,\,0,\,0,\,0]$.

The discrete–time state–space control law has the following structure:

$$u[k] \;=\; N_u\,r[k] \,-\, \boldsymbol{K}\,(\hat{\boldsymbol{x}}[k] - \boldsymbol{N}_x\,r[k]) \;=\; -\boldsymbol{K}\,\boldsymbol{x}[k] \,+\, \underbrace{(N_u \,+\, \boldsymbol{K}\,N_x)}_{=N_r}\,r[k] \tag{9}$$

where $r = \gamma^*$ is the wheel angle reference signal, and $\boldsymbol{x}$ is the state vector estimated with the simple state observer designed in point **(S6)**. As explained in the handout of laboratory activity 2, the feedforward gains $\boldsymbol{N}_x$ and $N_u$ are determined by solving the following set of linear equations:

$$\begin{bmatrix} \boldsymbol{\Phi} - \boldsymbol{I} & \boldsymbol{\Gamma} \\ \boldsymbol{H} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{N}_x \\ N_u \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ 1 \end{bmatrix} \tag{10}$$

The feedback gain $\boldsymbol{K}$ can instead be computed with LQR methods, in order to minimise the discrete–time quadratic cost function

$$J \;=\; \sum_{k=0}^{+\infty} \boldsymbol{x}^T[k]\,\boldsymbol{Q}\,\boldsymbol{x}[k] \,+\, \rho\,r\,u^2[k] \tag{11}$$

Select the cost weights $\boldsymbol{Q}$ and $r$ according to the *Bryson's rule*. At steady–state, it is desired to have:

$$|\gamma - \gamma^*| \,<\, \pi/36\;(5\,\mathrm{deg})\,, \qquad |\vartheta| \,<\, \pi/360\;(0.5\,\mathrm{deg})\,, \qquad |u| \,<\, 1\,\mathrm{V} \tag{12}$$

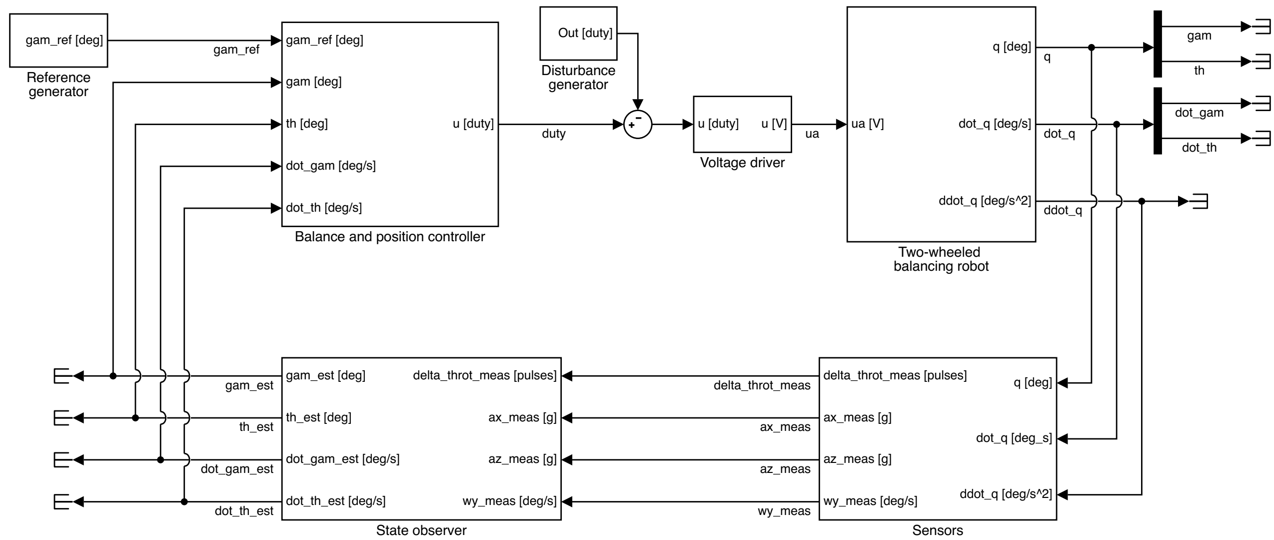
Therefore, according to the Bryson's rule, the cost weights $\boldsymbol{Q}$ and $r$ are selected as follows:

$$\boldsymbol{Q} \;=\; \mathrm{diag}\left\{\frac{1}{\bar{\gamma}^2},\; \frac{1}{\bar{\vartheta}^2},\; 0,\; 0\right\}, \qquad r \;=\; \frac{1}{\bar{u}^2} \tag{13}$$

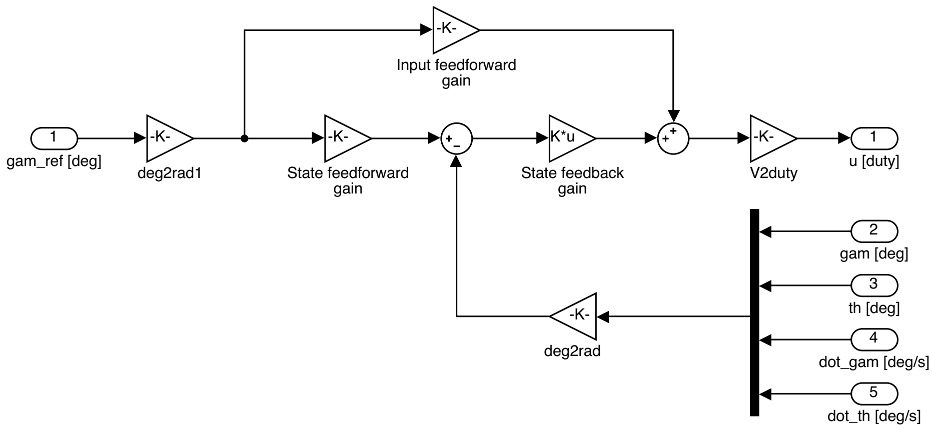
where

$$\bar{\gamma} \;=\; \pi/18\,, \qquad \bar{\vartheta} \;=\; \pi/360\,, \qquad \bar{u} \;=\; 1 \tag{14}$$

In the $\boldsymbol{Q}$ matrix, note that the weights of the two angular velocities $\dot{\gamma}$ and $\dot{\vartheta}$ have been set equal to zero. The extra weight $\rho$ in (11) is used to adjust the relative weighting between the state and input



(a) Balance–and–position state–space control system



(b) State–space controller

Figure 7: Possible Simulink implementation of the balance–and–position state–space control system (for the *nominal* perfect tracking of constant position set–points).

contributions to the total cost function value. Consider the following choices for such weight:

$$\rho \in \{\, 500,\ 5000 \,\} \tag{15}$$

For the computation of the feedback matrix, use the routine `dlqr` of the Control System Toolbox (CST).

**(S8)** Validate the design of point **(S7)** in simulation, by using the Simulink model of the balancing robot developed in Sec. 4.1. A possible Simulink implementation of the whole control system is shown in Fig. 7. In the model of Fig. 7a, the state observer is implemented as described in point **(S6)**. Test the controller in the following situations:

– initial state $\boldsymbol{x}(0) = [0,\, \pi/36,\, 0,\, 0]^T$ (i.e. initial body tilt angle equal to $\vartheta(0) = 5\,\mathrm{deg}$), reference input equal to zero, and no load disturbance (i.e. disturbance entering at the plant input).

  This test is aimed to verify that the controller is capable of restoring the balance after that the robot is released from a position off the upward vertical equilibrium.

- initial state $\boldsymbol{x}(0) = [0, 0, 0, 0]^T$, step reference input applied at $t = 0$ with amplitude $\gamma^* = 0.1/r\,\mathrm{rad}$, where $r$ is the wheel radius (this choice corresponds to a longitudinal position displacement of $10\,\mathrm{cm}$), and no load disturbance.

  This test is aimed to verify that the controller guarantees perfect tracking of the constant position set–point in the nominal case, when no external disturbances are present.

- initial state and reference input as in the previous point, and a load disturbance of amplitude $5.0/k_{duty \to V} \approx 115$ (where $k_{duty \to V}$ is the voltage driver input–to–output conversion gain), applied at the voltage driver input at $t = 10\,\mathrm{s}$. This is equivalent to a voltage disturbance of $5\,\mathrm{V}$ applied at the motor input.

  This test is aimed to verify that the controller is unable to guarantee the perfect tracking of the constant position set–point when an external disturbance is present. In this case, the disturbance is considered as an equivalent constant voltage disturbance entering at the plant input. Such disturbance could result from the application of a constant longitudinal force to the robot, similarly to what happens when the robot is forced to climb an inclined plane.

**(S9)** Repeat the design of point **(S8)** by introducing the integral action in the controller, in order to achieve *robust* tracking of constant wheel angle position set-points.

By introducing the integral action in (9), the control law becomes

$$\begin{cases} x_I[k+1] = x_I[k] + (\,y[k] - r[k]\,) \\ u[k] = N_u\, r[k] - \boldsymbol{K}\,(\boldsymbol{x}[k] - \boldsymbol{N}_x\, r[k]) - K_I\, x_I[k] \end{cases} \tag{16}$$

where $x_I$ is the integrator state variable[2]. After introducing the augmented state vector $\boldsymbol{x}_e = [x_I,\, \boldsymbol{x}]^T$ of the augmented state system

$$\Sigma_e \,:\, \begin{bmatrix} x_I[k+1] \\ \boldsymbol{x}[k+1] \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \boldsymbol{H} \\ \boldsymbol{0} & \boldsymbol{\Phi} \end{bmatrix}}_{\triangleq\, \boldsymbol{\Phi}_e} \underbrace{\begin{bmatrix} x_I[k] \\ \boldsymbol{x}[k] \end{bmatrix}}_{\triangleq\, \boldsymbol{x}_e} + \underbrace{\begin{bmatrix} 0 \\ \boldsymbol{\Gamma} \end{bmatrix}}_{\triangleq\, \boldsymbol{\Gamma}_e} u[k] - \begin{bmatrix} 1 \\ \boldsymbol{0} \end{bmatrix} r[k] \tag{17}$$

the control law (16) can be rewritten as follows:

$$u[k] = -\boldsymbol{K}_e\, \boldsymbol{x}_e[k] + (N_u + \boldsymbol{K}\,\boldsymbol{N}_x)\, r[k] = -\boldsymbol{K}_e\, \boldsymbol{x}_e[k] + N_r\, r[k] \tag{18}$$

The augmented state feedback matrix $\boldsymbol{K}_e = [K_I,\, \boldsymbol{K}]^T$ can be designed again by resorting to LQR methods. Compared to the design of point **(S7)**, in this case the cost matrix $\boldsymbol{Q}$ contains an extra weight $q_{11}$ for the integrator state variable $x_I$. This weight must be chosen different from zero, otherwise the LQR cannot be designed. However, the weight cannot be chosen with the Bryson's rule, since there is no reasonable and immediate way to identify the maximum deviation of the integrator state from its steady–state value. In practice, the weight $q_{11}$ must be chosen by trial and error. Two possible choices to consider for such weight are:

$$q_{11} \in \{0.1,\ 1\} \tag{19}$$

[2] In (16), note that the tracking error is defined as $e[k] = y[k] - r[k]$.

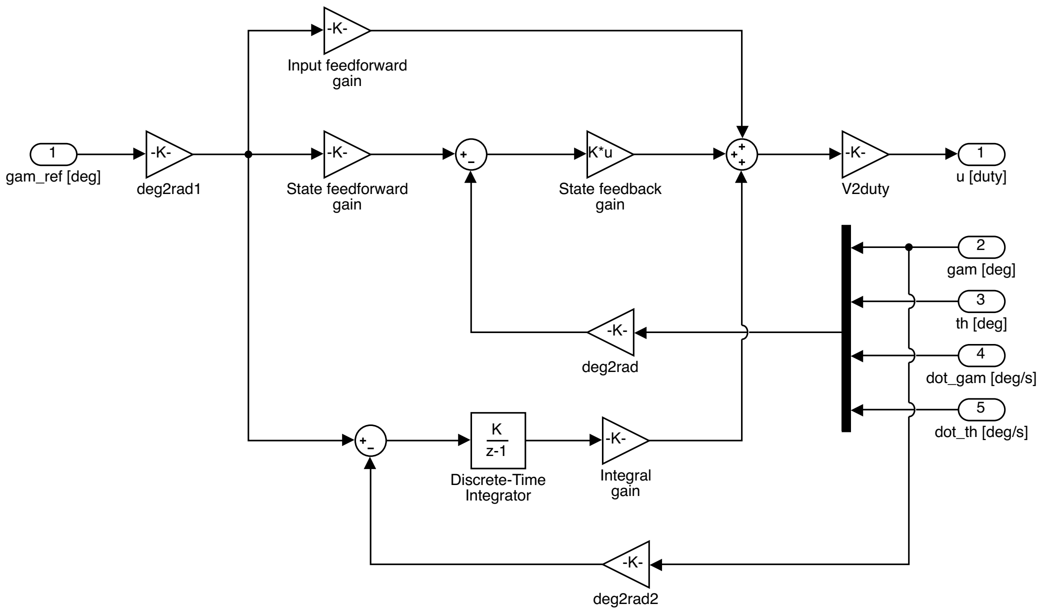



Figure 8: Possible Simulink implementation of the state–space controller, for the balance control and the *robust* perfect tracking of constant position set–points, using the integral action approach.

Choose the remaining weights as suggested in point **(S7)**.

A possible Simulink implementation of the control law (16) is shown in Fig. 8. In (16), note that the integrator state $x_I$ consists of the discrete–time *accumulation* of the tracking error $e[k] = y[k] - r[k]$. Therefore, the `Integrator method` property of the `Discrete-Time Integrator` block in Fig. 8 must be set as `Accumulation:Forward Euler` (i.e. the integrator has to be implemented as a discrete–time Forward Euler accumulator).

**(S10)** Validate the design of point **(S9)** in simulation, using the Simulink model of the whole control system prepared in point **(S8)** (simply replace the controller block of Fig. 7 with that of Fig. 8).

Repeat the same tests of point **(S8)**. In particular, verify that the controller is capable of perfectly tracking a constant position reference at steady state, even in presence of a constant load disturbance.

# 5 Laboratory assignments: experimental tests

**(E1)** Prepare a Simulink model for testing the balance–and–position controllers designed in Sec. 4.2 on the balancing robot available in laboratory. For such purpose, it is sufficient to proceed as follows:

– make a copy of the Simulink model prepared in point **(S8)** of Sec. 4.2 for the numerical simulations (see also Fig. 7).

– replace the models of the balancing robot and the sensors (encoder and MPU) with the blocks of the **Balancing Robot Toolbox** (BRT) that allows to interface with the robot hardware.

– configure the model parameters to enable the execution on the balancing robot micro–controller unit (MCU), according to the details provided in the introductory guide to the experimental setup (laboratory guide 2).

  In particular, in the `Hardware Implementation` settings, select *Arduino Mega 2560* as the `Hardware Board`, and `Automatically` as the detection method for the `Host-board connection` port. Regarding the Solver parameters, choose a `Fixed-step discrete (no continuous states)` solver, and a sample time (fixed–step size) equal to $T = 0.01\,\mathrm{s}$.

A possible implementation is shown in Fig. 9. In addition to the base controller, the proposed implementation includes some extra "logic", to enable the generation of the position reference and the load disturbance via the two pushbuttons available on the robot. The logic operates as follows:

– the controller is enabled by pressing either the pushbutton 1 or 2.

  This logic is implemented by using two `BRT` → `Utilities` → `Up-Counter (with upper bound)` counters to detect when the pushbuttons are pressed for the first time. The upper bounds of the two counters are both set equal to $1$. The controller is enabled when at least one counter output is equal to $1$.

  Note that the two up–counters (with upper bound set to $1$) behave as two set–reset (SR) latches, with the S inputs driven by the two pushbuttons.

– a constant reference signal is generated with a delay after pressing the pushbutton 1.

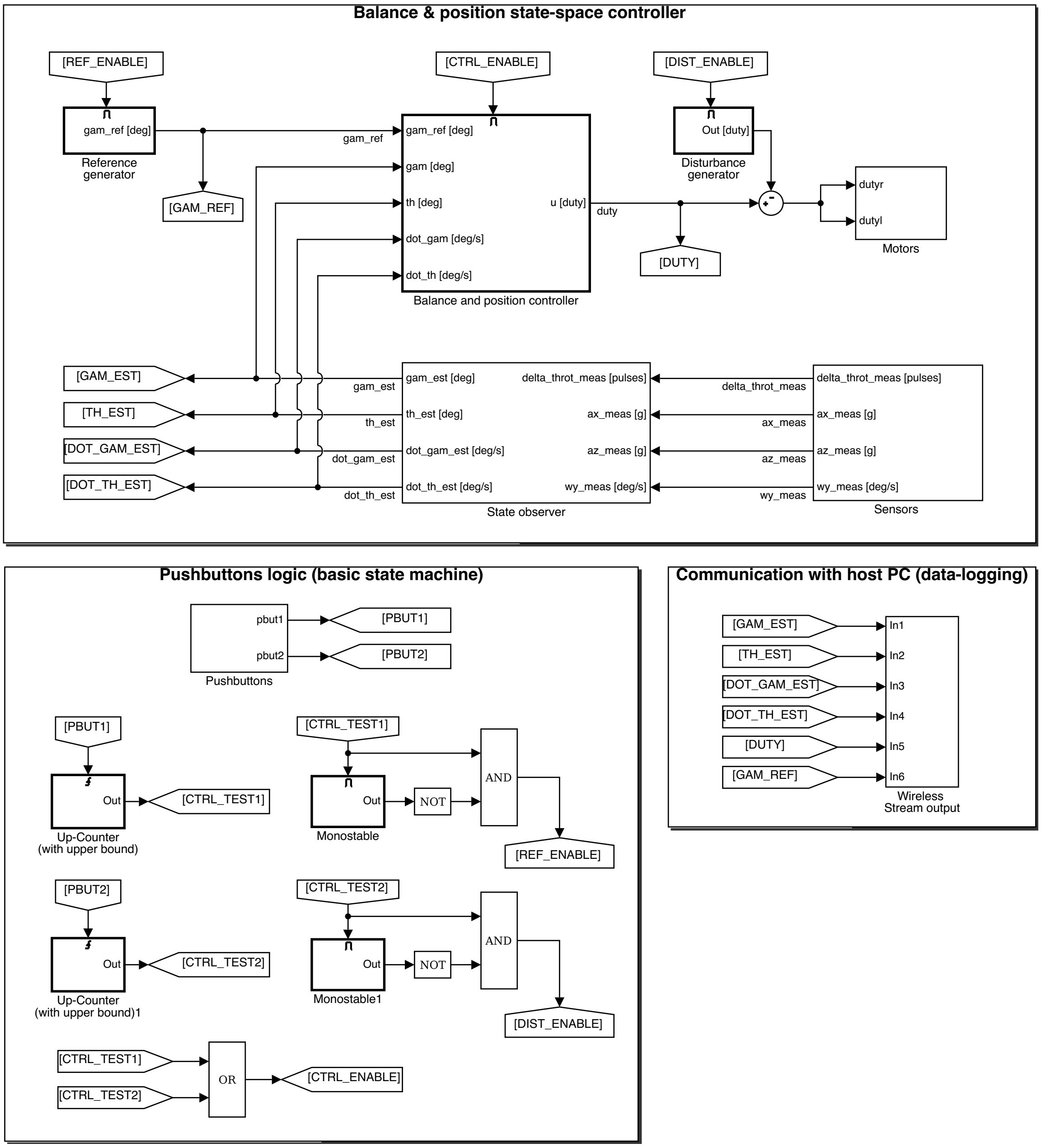
  This logic is implemented by enabling a `BRT` → `Utilities` → `Monostable` block with the output of the up–counter triggered by the push–button 1. In this way, a pulse of specified duration is generated after pressing the pushbutton. Before this event, or after the pulse expiration, the output of the monostable is equal to zero. Therefore, the condition for enabling the generation of the position reference is that the pushbutton 1 has been pressed (i.e. the output of the corresponding up–counter is equal to $1$), and the output of the monostable is not equal to $1$.

  Consider to set a pulse duration of at least $10\,\mathrm{s}$ in the monostable: this amount of time should be sufficient for the robot to reach a stable vertical balance, before moving according to the provided position reference.

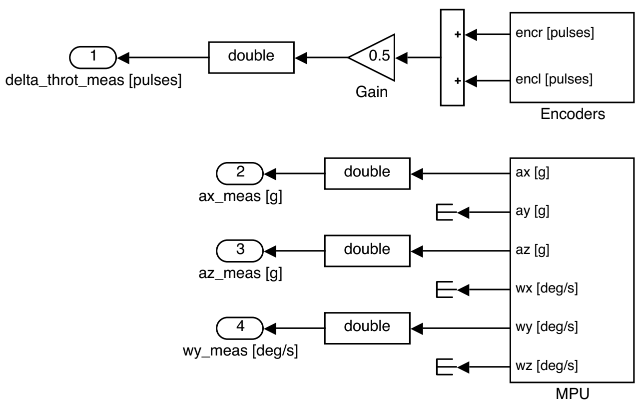
– a constant load disturbance is generated with a delay after pressing the pushbutton 2.

  This logic is identical to that of the previous point (for enabling the generation of the position reference), except for the pushbutton used.

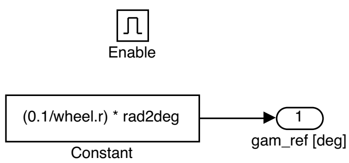
Both the balance–and–position controller, and the reference/disturbance generator are *enabled* blocks (use a `Ports & Subsystems` → `Enable` to create a subsystem with an enable port). When working



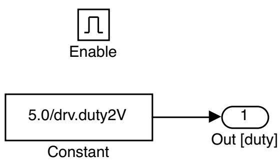

(a) Balance–and–position state–space control system




(b) Sensors interface




(c) Reference generator




(d) Disturbance generator

Figure 9: Possible Simulink implementation of the balance–and–position controller to be uploaded on the balancing robot MCU.

with an enabled block, it is necessary to specify how to set its outputs when the block is disabled. For each output, this can be done by setting the properties of the `Sinks` → `Out` port appropriately (double–click on the port block to access its parameters). In particular, the `Output when disabled` option specifies to either hold or reset the output port value when the block is disabled. If the `held` option is selected, then an initial output value has to be provided (in the `Initial output` field). In the Simulink model of Fig. 9, the option `reset` is selected for the outputs of both the controller and the reference/disturbance generator.

In the sensors interface implementation of Fig. 9b, note that the angular displacement $\Delta\vartheta_{rot}$ of the rotor with respect to the stator (in the the planar robot approximation) is obtained as the average of the angular displacements $\Delta\vartheta_{rot,r}$ and $\Delta\vartheta_{rot,l}$ measured by the two encoders:

$$\Delta\vartheta_{rot} = \frac{\Delta\vartheta_{rot,r} + \Delta\vartheta_{rot,l}}{2} \tag{20}$$

Moreover, note that a `Signal Attributes` → `Data Type Conversion` has been used to convert the outputs of the `BRT` → `Sensors` → `Encoders` and `BRT` → `Sensors` → `MPU` blocks from their original data types (`uint32` and `single`, respectively) to the `double` type (in the block settings, select the option `double` in the `Output data type` drop–down list), which is the internal type adopted in the controller implementation. This choice guarantees the best numerical accuracy; however, it is also more demanding in terms of memory occupation and computational effort, which are both scarce resources on a typical embedded system. The `single` floating point data type can be used to save space and computational resources.

**(E2)** Test the controller designed in point **(S7)** of Sec. 4.2 on the balancing robot, using the model prepared in point **(S6)**.

For the tests, use a constant position reference equal to $\gamma^* = 0.1/r\,\mathrm{rad}$, where $r$ is the wheel radius, which corresponds to a longitudinal position displacement of $10\,\mathrm{cm}$, and a load disturbance equal to $u_d = 5.0/k_{duty\to V} \approx 115$, where $k_{duty\to V}$ is the PWM duty–cycle to voltage conversion gain, which corresponds to a voltage disturbance of $5.0\,\mathrm{V}$. The position reference and/or the load disturbance should be enabled at least $10\,\mathrm{s}$ after turning the controller on, to leave enough time to the robot to initially stabilize on the vertical position.

**(E3)** Test the controller designed in point **(S9)** of Sec. 4.2 on the balancing robot, using the model prepared in point **(S6)**.

For the tests, use the same position reference and load disturbance of the previous point **(E3)**.

**(E4)** (**optional**) From the experimental tests of points **(E2)** and **(E3)** it can be noticed that the balancing robot has the tendency of drifting laterally, and in practice it never moves along a perfectly straight line. This problem is caused by the fact that even if the two motors are driven by the same voltage command, they do not necessarily move by the same angle, because of unavoidable differences in the motor parameters, and the presence of friction and backlash in the mechanical transmission (gearboxes).

To avoid the lateral drift motion, an extra controller for the robot heading angle (yaw angle) $\psi$ is required. The design of such controller is beyond the scope of this laboratory activity, and will be addressed in a possible follow–up (i.e. "Combined longitudinal and heading–angle state–space control of the balancing robot"). In the following, a simple implementation based on a PI controller is presented, for the only purpose of illustrating how to test the longitudinal controller under "more stable" working conditions. The proposed implementation is shown in Fig. 10 and 11. It consists of controlling the two motors with the following two voltage commands (for the right and left motors, respectively):

$$u_r = u_\Sigma + u_\Delta \,, \qquad u_l = u_\Sigma - u_\Delta \tag{21}$$

where $u_\Sigma$ is a "common–mode" command generated by the longitudinal controller, and $u_\Delta$ a "differential–mode" command generated by the heading angle (yaw) controller. The longitudinal controller is the state–space controller designed in point **(S7)** or **(S9)** of Sec. 4.2. The heading angle controller is instead a simple PI regulator, as shown in Fig. 10c. For the purpose of this laboratory activity, the following proportional and integral gains can be used:

$$K_P = 3.3 \,, \qquad K_I = 0.7 \tag{22}$$

The heading angle (yaw) $\psi$ is estimated in the block of Fig. 11b by using the expression (10), under the assumption that $\psi(0) = 0$, namely

$$\psi = \frac{r}{w} \left(\vartheta_r - \vartheta_l\right) \tag{23}$$

where $\vartheta_r$ and $\vartheta_l$ are the wheels angles derived from the encoders measurements. From (6) and (7), these quantities are equal to:

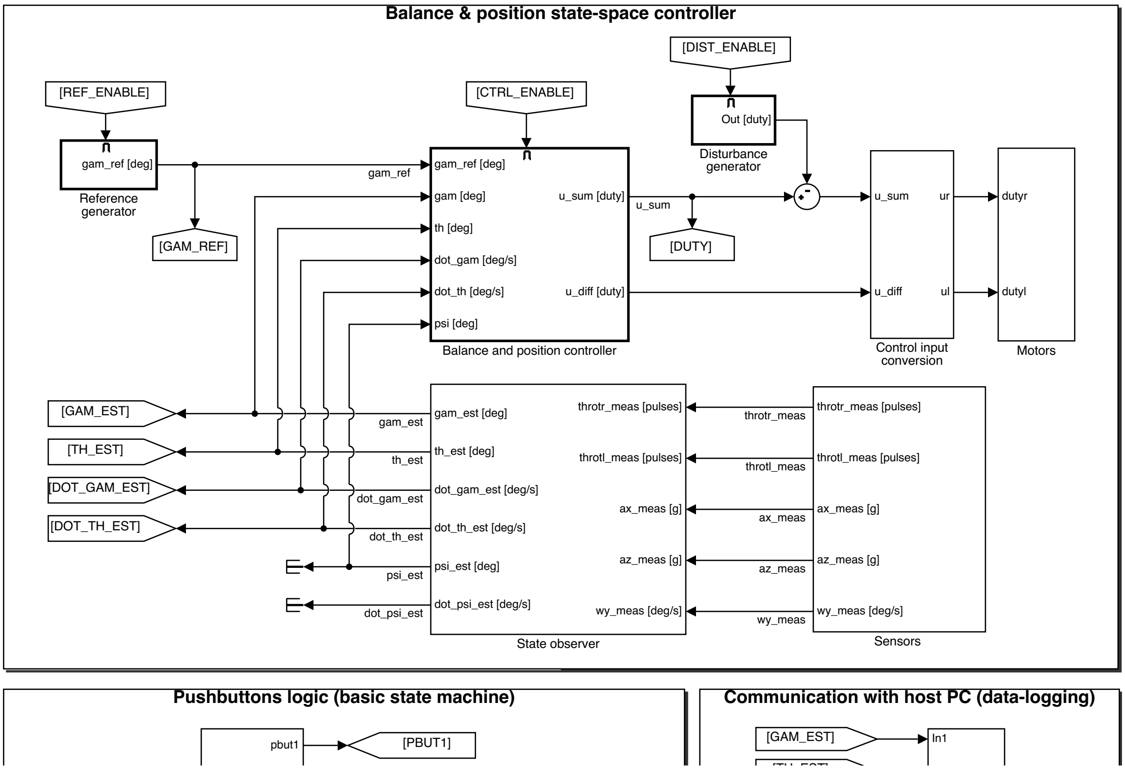
$$\vartheta_l = \frac{\Delta\vartheta_{rot,l}}{N} + \vartheta \,, \qquad \vartheta_r = \frac{\Delta\vartheta_{rot,r}}{N} + \vartheta \tag{24}$$

where $\Delta\vartheta_{rot,l}$ and $\Delta\vartheta_{rot,r}$ are the measurements provided by the two encoders.

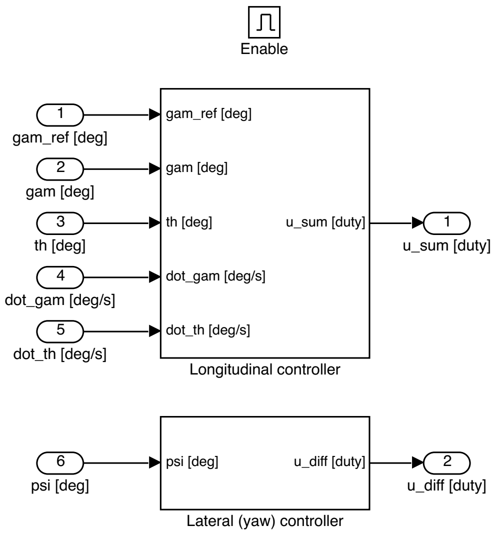
Instead, the variable $\gamma$ is estimated by simply considering the average value of the two wheels angles, namely

$$\gamma = \frac{\vartheta_r + \vartheta_l}{2} \tag{25}$$

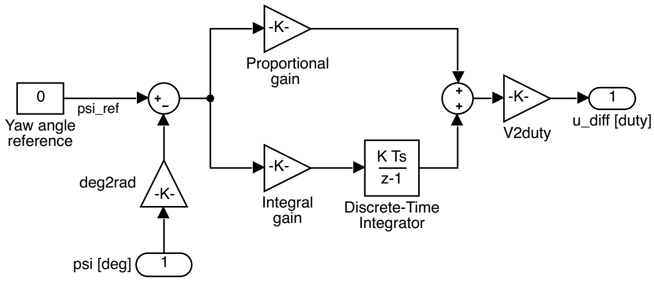
Consider to repeat the experimental tests of points **(E2)** and **(E3)** with the controller configuration proposed above.



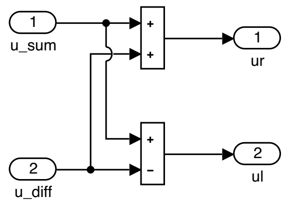

(a) Balance–and–position state–space control system




(b) Balance–and–position controller

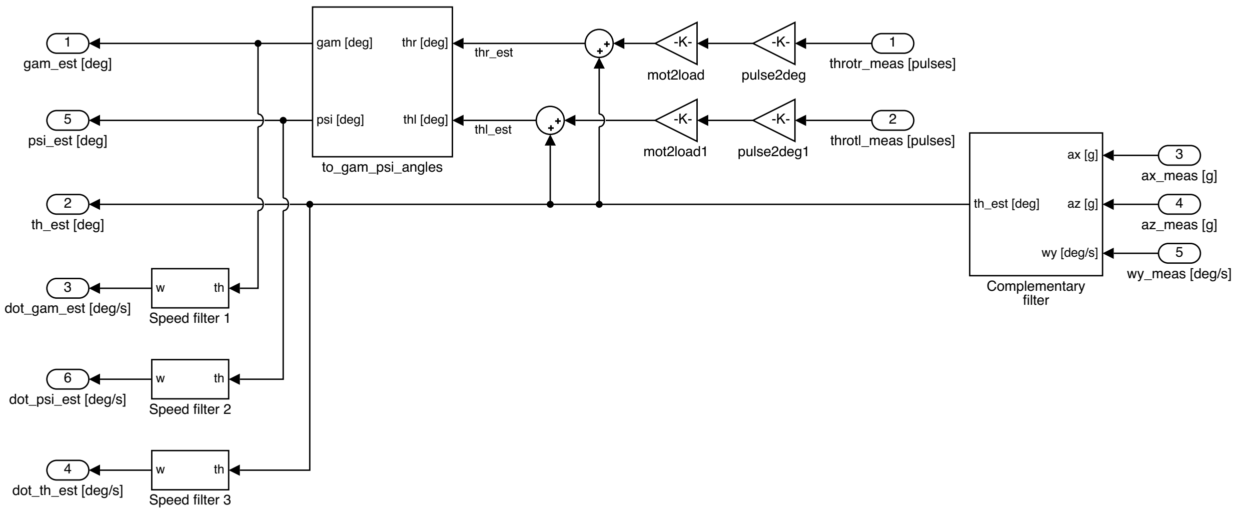



(c) Lateral (yaw) controller

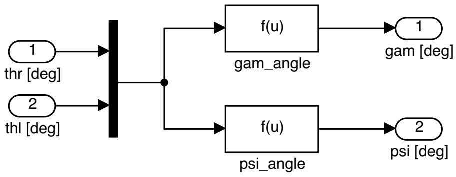



(d) Control input conversion

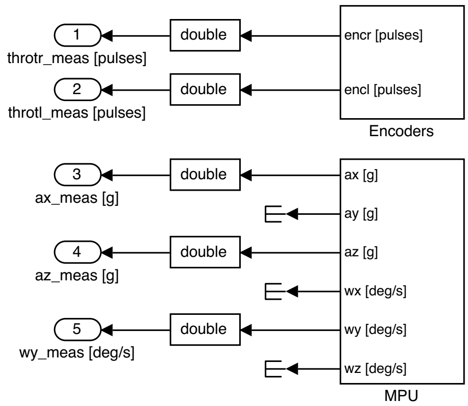
Figure 10: Possible Simulink implementation of a combined longitudinal and heading–angle controller for the balancing robot.




(a) State observer




(b) "to_gam_psi_angles" block




(c) Sensors interface

Figure 11: Possible Simulink implementation of a combined longitudinal and heading–angle controller for the balancing robot (cont'd).

# 6 Appendix

## 6.1 Simulink implementation of the electromechanical dynamics with *S–Functions*

The nonlinear dynamical model (54)–(55) can be implemented in Simulink using a **MATLAB S–Function**. Consult the Simulink documentation for the details regarding how to write a MATLAB S–Function, using either the *Level–1* or *Level–2* API (Application Programming Interface).

A possible Level–2 MATLAB S–Function implementation is reported in Listing 2. The corresponding block to be used in the Simulink model is the `User-Defined Functions` → `Level-2 MATLAB S-Function` block. The block has two parameters: the *S–function name* (required parameter) is the name of the MATLAB script containing the implementation of the S–function (i.e. `sfun_balrob_long_dyn.m`), while *Parameters* (optional parameter) are the extra parameters to be passed to the S–function (if required by the implementation). The Simulink model based on the S–function of Listing 2 is shown in Fig. 12.

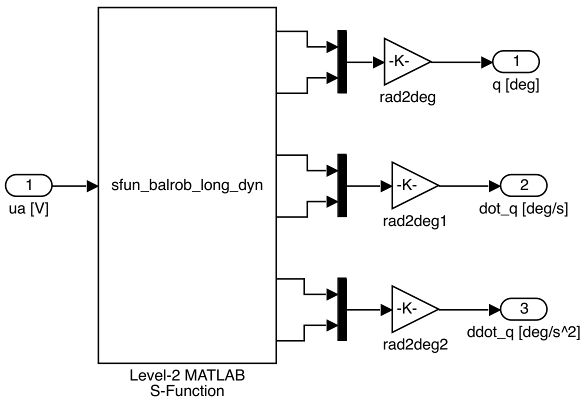



Figure 12: Simulink model implementation details: electromechanical dynamics (implemented by using a MATLAB S–Function).

Note that the S–function implementation specifies a block with 1 input (line 11), 6 outputs (line 12), 4 continuous states (line 15) and 5 parameters (line 37). Accordingly, the S–function block in the Simulink model is shown with 1 input (the armature voltage $u_a$) and 6 outputs (the vector of generalized variables $\boldsymbol{q}$, and its first two time derivatives $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$). The 5 expected parameters are the vector of initial conditions `x0` (line 72), and the four data structures `body`, `mot`, `gbox` and `wheel` (lines 145–148). The 4 continuous–state variables are the generalized variables in $\boldsymbol{q}$, and their derivatives in $\dot{\boldsymbol{q}}$. The acceleration vector $\ddot{\boldsymbol{q}}$ is computed by the function `get_acc` (lines 140–242) according to the expression derived in (1).

Listing 2: `sfun_balrob_long_dyn.m`

```
1  function sfun_balrob_long_dyn(block)
2
3  setup(block);
4
5  end %function
6
7
8  function setup(block)
9
10 % Register number of ports
11 block.NumInputPorts  = 1;
12 block.NumOutputPorts = 2*3;
13
14 % Register number of continuous states
15 block.NumContStates = 2*2;
16
17 % Setup port properties to be inherited or dynamic
18 block.SetPreCompInpPortInfoToDynamic;
19 block.SetPreCompOutPortInfoToDynamic;
20
21 % Override input port properties
22 for k = 1:block.NumInputPorts,
23     block.InputPort(k).Dimensions        = 1;
24     block.InputPort(k).DatatypeID        = 0;  % double
25     block.InputPort(k).Complexity        = 'Real';
26     block.InputPort(k).DirectFeedthrough = false;
27 end;
28
29 % Override output port properties
30 for k = 1:block.NumOutputPorts,
31     block.OutputPort(k).Dimensions  = 1;
```

```
    block.OutputPort(k).DatatypeID  = 0; % double
    block.OutputPort(k).Complexity  = 'Real';
end;

% Register parameters
block.NumDialogPrms = 5;

% Register sample times
%  [0 offset]            : Continuous sample time
block.SampleTimes = [0 0];

% Specify the block simStateCompliance.
%    'DefaultSimState', < Same sim state as a built-in block
block.SimStateCompliance = 'DefaultSimState';

%   Register block methods
block.RegBlockMethod('Start',                       @Start);
block.RegBlockMethod('SetInputPortSamplingMode',    @SetInputPortSamplingMode);
block.RegBlockMethod('Outputs',                     @Outputs);      % Required
block.RegBlockMethod('Derivatives',                 @Derivatives);
block.RegBlockMethod('Terminate',                   @Terminate);    % Required

for k = 1:block.NumOutputPorts,
    block.OutputPort(k).SamplingMode = 0;
end;

end % setup


function SetInputPortSamplingMode(block, port, mode)

block.InputPort(port).SamplingMode = mode;

end % SetInputPortSamplingMode


function Start(block)

%%  Get init state

x0  = block.DialogPrm(1).Data;      %   get init state

%%  Set init state

block.ContStates.Data(1) =  x0(1);  %   gam(0)
block.ContStates.Data(2) =  x0(2);  %   th(0)

block.ContStates.Data(3) =  x0(3);  %   dot_gam(0)
block.ContStates.Data(4) =  x0(4);  %   dot_th(0)

end % Start


function Outputs(block)

%%  Extract state components

gam     = block.ContStates.Data(1);
th      = block.ContStates.Data(2);

dot_gam = block.ContStates.Data(3);
dot_th  = block.ContStates.Data(4);

```

```matlab
%%  Get accelerations

[ddot_gam, ddot_th] = get_acc(block);

%%  Set outputs

block.OutputPort(1).Data = gam;
block.OutputPort(2).Data = th;

block.OutputPort(3).Data = dot_gam;
block.OutputPort(4).Data = dot_th;

block.OutputPort(5).Data = ddot_gam;
block.OutputPort(6).Data = ddot_th;

end % Outputs


function Derivatives(block)

%%  Extract state components

dot_gam = block.ContStates.Data(3);
dot_th  = block.ContStates.Data(4);

%%  Get accelerations

[ddot_gam, ddot_th] = get_acc(block);

%%  Set state derivative

block.Derivatives.Data(1) = dot_gam;
block.Derivatives.Data(2) = dot_th;

block.Derivatives.Data(3) = ddot_gam;
block.Derivatives.Data(4) = ddot_th;

end % Derivatives


function Terminate(block)

end % Terminate


function [ddot_gam, ddot_th] = get_acc(block)

%%  Get inputs and parameter structs

%   parameters
body  = block.DialogPrm(2).Data;    %   body data struct
mot   = block.DialogPrm(3).Data;    %   mot data struct
gbox  = block.DialogPrm(4).Data;    %   gbox data struct
wheel = block.DialogPrm(5).Data;    %   wheel data struct

%   input voltages
ua = block.InputPort(1).Data;       %   armature voltage (right/left motor)

%%  Extract state components

th      = block.ContStates.Data(2);
dot_gam = block.ContStates.Data(3);
dot_th  = block.ContStates.Data(4);
```

```
%% Extract params

% body params
l       = body.zb;
mb      = body.m;
Ibyy    = body.Iyy;

% wheel params
w       = 2*wheel.yb;
r       = wheel.r;
mw      = wheel.m;
Iwyy    = wheel.Iyy;

% (motor) rotor params
zbrot   = mot.rot.zb;
mrot    = mot.rot.m;
Irotyy  = mot.rot.Iyy;

% gear ratio
n   = gbox.N;

% friction params
bw  = wheel.B;      % wheel viscous fric coeff
bm  = mot.B;        % motor viscous fric coeff (motor side)
bg  = gbox.B;       % gbox viscous fric coeff (load side)
b   = n^2*bm+bg;    % motor+gbox viscous fric coeff (load side)

% gravity acc
g = 9.81;

%% Get motor torques

% back-EMFs
ue = mot.Ke * n*(dot_gam-dot_th);

% motor torque (single motor)
tau = n*mot.Kt * (ua-ue)/mot.R;

%% Evaluate accelerations of generalised coords

% inertia matrix
MM = zeros(2,2);

MM(1,1) = 2*Iwyy + 2*Irotyy*n^2 + (mb + 2*(mrot+mw))*r^2;
MM(1,2) = 2*(1-n)*n*Irotyy + r*(l*mb + 2*mrot*zbrot)*cos(th);

MM(2,1) = MM(1,2);
MM(2,2) = Ibyy + 2*(1-n)^2*Irotyy + mb*l^2 + 2*mrot*zbrot^2;

% Coriolis + centrifugal terms matrix
CC = zeros(2,2);

CC(1,1) = 0;
CC(2,1) = 0;

CC(2,2) = 0;
CC(1,2) = -r*(mb*l + 2*mrot*zbrot)*sin(th)*dot_th;

% viscous friction matrix
Fv = zeros(2,2);

Fv(1,1) = 2*(b+bw);
```

```
Fv(1,2) = -2*b;

Fv(2,1) = Fv(1,2);
Fv(2,2) = 2*b;

%   gravity loading
 GG = zeros(2,1);
GG(2) = -g*(mb*l + 2*mrot*zbrot)*sin(th);

%   generalized actuator forces
TT = zeros(2,1);
TT(1) = 2*tau;
TT(2) = -2*tau;

%   get accelerations of generalised coords (q = [gam, th].')
dotq = [dot_gam, dot_th].';
ddotq = MM \ (TT - CC*dotq - Fv*dotq - GG);

ddot_gam = ddotq(1);
ddot_th  = ddotq(2);

end % get_acc
```