

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:

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

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

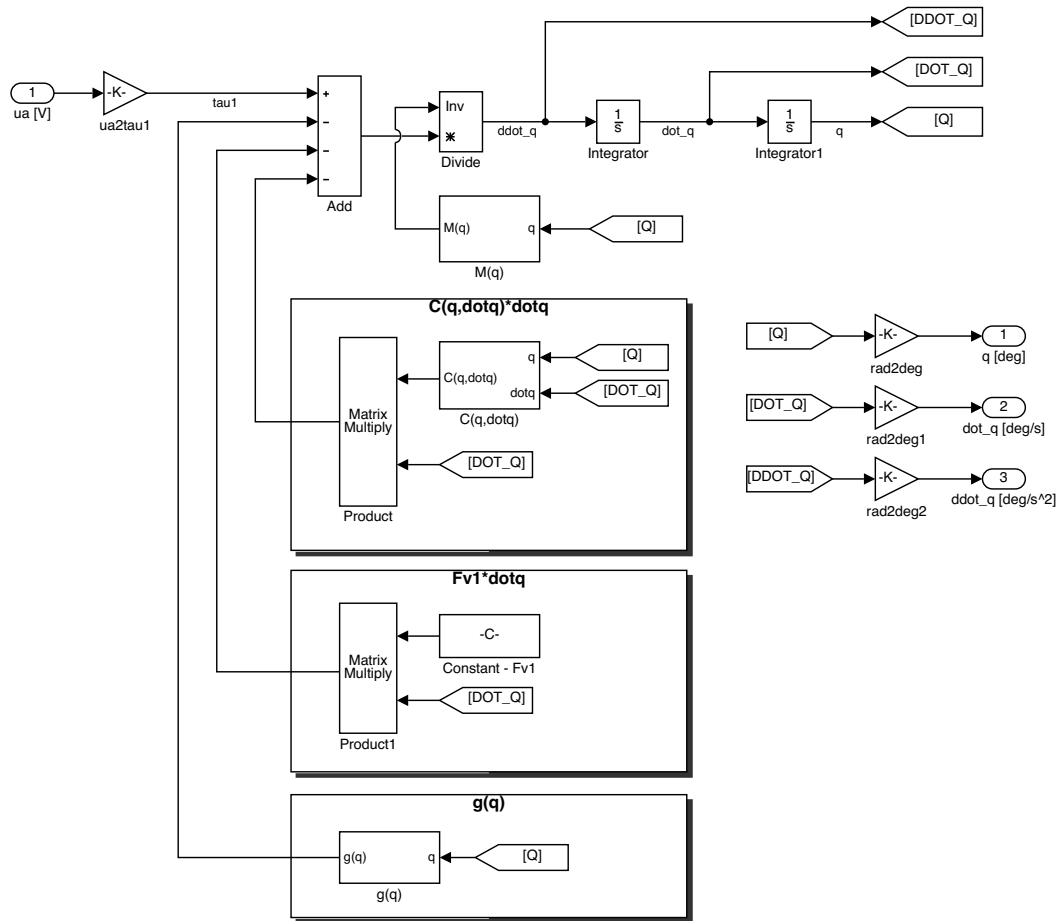
- use the User-Defined Functions \rightarrow Fcn block to implement the state-dependent elements of the aforementioned matrices. For the state-independent (i.e. constant) elements, use the Sources \rightarrow Constant block.
- use the Signal Routing \rightarrow Mux block to combine the single matrix elements into matrix columns.
- use the Math Operations \rightarrow Matrix Concatenate block to form a single matrix by concatenation of its columns.
- use the Math Operations \rightarrow 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 $(2Nk_t/R_a) [1, -1]^T$.

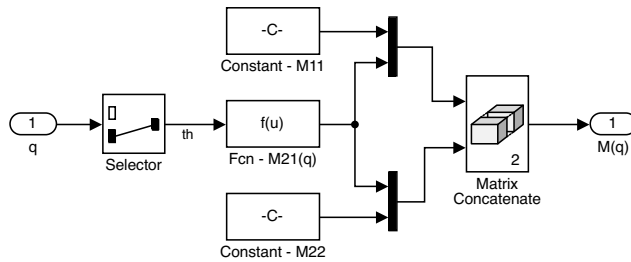
An alternative Simulink implementation of the nonlinear dynamics (54)–(55) consists of using MATLAB *S-Functions*. A possible implementation based on S-Function is reported in Appendix 6.1.

(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:

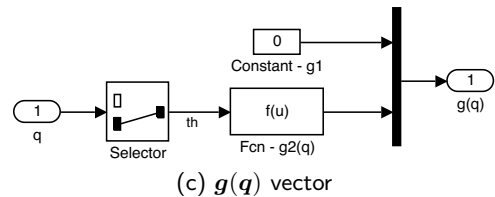
- use the User-Defined Functions \rightarrow Fcn block to compute the x and z axes components of the accelerometer output vector (64).
- use a Discrete \rightarrow Zero-Order Hold block to sample the accelerometer and gyroscope outputs with a sampling time equal to T_s (see line 4 in Listing 1), which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



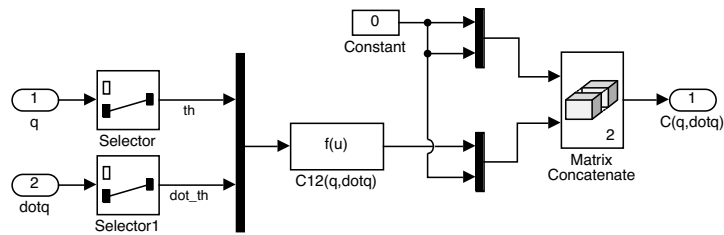
(a) Electromechanical dynamics



(b) $M(q)$ matrix



(c) $g(q)$ vector



(d) $C(q, \dot{q})$ 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 T_s as the sampling time.

- use a `Discontinuities` → `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.

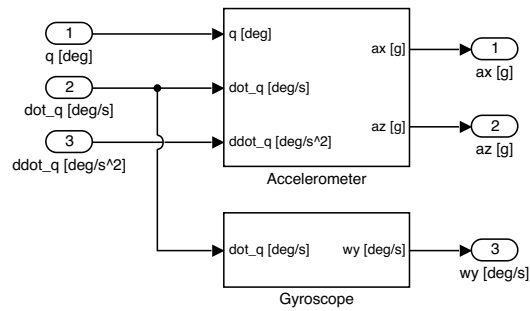
(S3) Implement a Simulink model of the incremental encoder used to measure the motor shaft position. A possible Simulink implementation is shown in Fig. 3. In particular:

- 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` → `Zero-Order Hold` block to sample the encoder output with a sampling time equal to T_s .
- use a `Discontinuities` → `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 [deg] units).
- the encoder output is in [pulses] units. Therefore, use a `sens.enc.deg2pulse` gain to convert the encoder output from [deg] to [pulses] units.

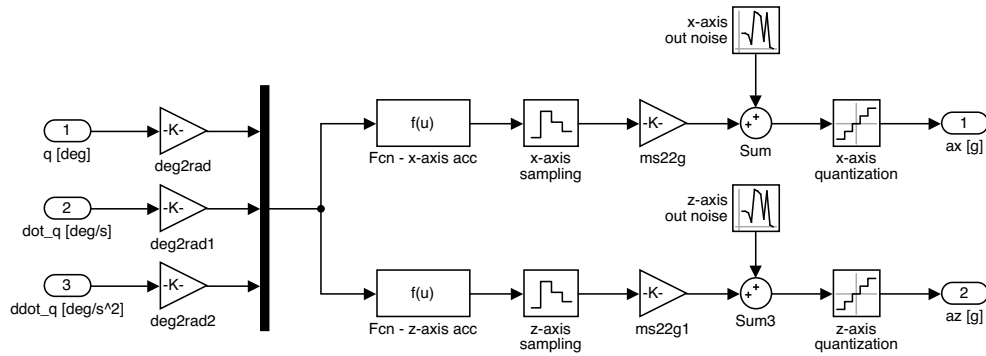
(S4) Implement a Simulink model of the motor voltage driver. A possible Simulink implementation is shown in Fig. 4. In particular:

- 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 V. Use a `Discontinuities` → `Saturation` block to limit the driver input to the voltage range specified above. Use $\pm\text{drv.dutymax}$ (see line 114 in Listing 1) as the saturation levels.
- use a `Math Operations` → `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 [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 $\pm\text{drv.Vbus}$ (see line 110 in Listing 1) as the saturation levels.
- use a `Discrete` → `Zero-Order Hold` block to hold the voltage command within each sampling period (equal to T_s).

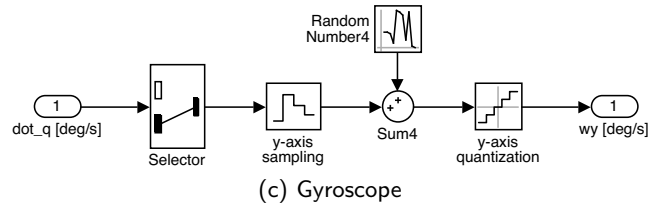
(S5) Combine all the models prepared in points **(S1)–(S4)**, to form a Simulink model of the whole balancing robot hardware equipment (i.e. electromechanical system + motor driver + sensors). A possible Simulink implementation is shown in Fig. 5.



(a) Motion Processing Unit (MPU)



(b) Accelerometer



(c) Gyroscope

Figure 2: Simulink model implementation details: MPU.

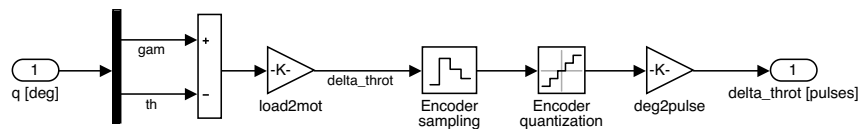


Figure 3: Simulink model implementation details: encoder.

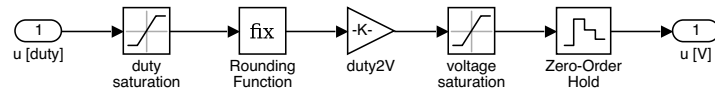


Figure 4: Simulink model implementation details: voltage driver.

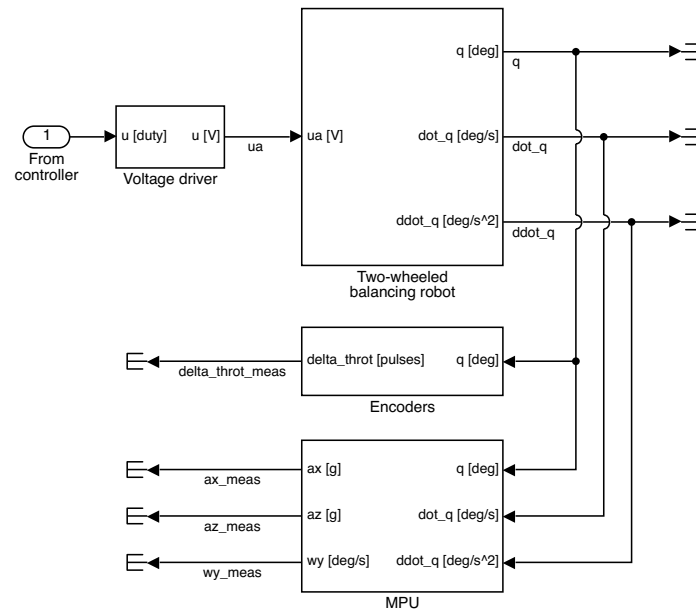


Figure 5: Simulink model implementation details: overall balancing robot hardware equipment.

Listing 1: balrob_params.m

```

1 %% General parameters and conversion gains
2
3 % controller sampling time
4 Ts = 1e-2;
5
6 % gravity acc [m/s^2]
7 g = 9.81;
8
9 % conversion gains
10 rpm2rads = 2*pi/60; % [rpm] -> [rad/s]
11 rads2rpm = 60/2/pi; % [rad/s] -> [rpm]
12 rpm2degs = 360/60; % [rpm] -> [deg/s]
13 degs2rpm = 60/360; % [deg/s] -> [rpm]
14 deg2rad = pi/180; % [deg] -> [rad]
15 rad2deg = 180/pi; % [rad] -> [deg]
16 g2ms2 = g; % [acc_g] -> [m/s^2]
17 ms2g = 1/g; % [m/s^2] -> [acc_g]
18 ozin2Nm = 0.706e-2; % [oz*inch] -> [N*m]
19
20 % robot initial condition
21 x0 = [ ...
22     0, ... % gam(0)
23     5*deg2rad, ... % th(0)
24     0, ... % dot_gam(0)
25     0]; % dot_th(0)
26
27 %% DC motor data
28
29 % motor id: brushed DC gearmotor Pololu 30:1 37Dx68L mm
30
31 % electromechanical params
32 mot.UN = 12; % nominal voltage
33 mot.taus = 110/30 * ozin2Nm; % stall torque @ nom voltage
34 mot.Is = 5; % stall current @ nom voltage
35 mot.w0 = 350 * 30 * rpm2rads; % no-load speed @ nom voltage
36 mot.I0 = 0.3; % no-load current @ nom voltage
37

```

```

38 mot.R      = mot.UN/mot.Is;           % armature resistance
39 mot.L      = NaN;                     % armature inductance
40 mot.Kt      = mot.taus/mot.Is;        % torque constant
41 mot.Ke      = (mot.UN - mot.R*mot.I0)/(mot.w0); % back-EMF constant
42 mot.eta     = NaN;                     % motor efficiency
43 mot.PN      = NaN;                     % nominal output power
44 mot.IN      = NaN;                     % nominal current
45 mot.tauN    = NaN;                     % nominal torque
46
47 % dimensions
48 mot.rot.h   = 30.7e-3;                 % rotor height
49 mot.rot.r   = 0.9 * 17e-3;             % rotor radius
50
51 mot.stat.h  = 68.1e-3;                 % stator height
52 mot.stat.r  = 17e-3;                   % stator radius
53
54 % center of mass (CoM) position
55 mot.rot.xb  = 0;                       % (left) rot CoM x-pos in body frame
56 mot.rot.yb  = 42.7e-3;                 % (left) rot CoM y-pos in body frame
57 mot.rot.zb  = -7e-3;                   % (left) rot CoM z-pos in body frame
58
59 mot.stat.xb = 0;                       % (left) stat CoM x-pos in body frame
60 mot.stat.yb = 52.1e-3;                 % (left) stat CoM y-pos in body frame
61 mot.stat.zb = -7e-3;                   % (left) stat CoM z-pos in body frame
62
63 % mass
64 mot.m       = 0.215;                   % total motor mass
65 mot.rot.m   = 0.35 * mot.m;           % rotor mass
66 mot.stat.m  = mot.m - mot.rot.m;      % stator mass
67
68 % moment of inertias (MoI) wrt principal axes
69 mot.rot.Ixx = mot.rot.m/12 * (3*mot.rot.r^2 + mot.rot.h^2); % MoI along r dir
70 mot.rot.Iyy = mot.rot.m/2 * mot.rot.r^2; % MoI along h dim
71 mot.rot.Izz = mot.rot.Ixx; % MoI along r dir
72
73 mot.stat.Ixx = mot.stat.m/12 * (3*mot.stat.r^2 + mot.stat.h^2); % MoI along r dir
74 mot.stat.Iyy = mot.stat.m/2 * mot.stat.r^2; % MoI along h dir
75 mot.stat.Izz = mot.stat.Ixx; % MoI along r dir
76
77 % viscous friction coeff (motor side)
78 mot.B = mot.Kt*mot.I0/mot.w0;
79
80 %% Gearbox data
81
82 gbox.N = 30; % reduction ratio
83 gbox.B = 0.025; % viscous friction coeff (load side)
84
85 %% Battery data
86
87 % electrical data
88 batt.UN = 11.1; % nominal voltage
89
90 % dimensions
91 batt.w = 136e-3; % battery pack width
92 batt.h = 26e-3; % battery pack height
93 batt.d = 44e-3; % battery pack depth
94
95 % center of mass (CoM) position
96 batt.xb = 0; % CoM x-pos in body frame
97 batt.yb = 0; % CoM y-pos in body frame
98 batt.zb = 44e-3; % CoM z-pos in body frame
99
100 % mass

```

```

101 batt.m = 0.320;
102
103 % moment of inertias (MoI) wrt principal axes
104 batt.Ixx = batt.m/12 * (batt.w^2 + batt.h^2); % MoI along d dim
105 batt.Iyy = batt.m/12 * (batt.d^2 + batt.h^2); % MoI along w dim
106 batt.Izz = batt.m/12 * (batt.w^2 + batt.d^2); % MoI along h dim
107
108 %% H-bridge PWM voltage driver data
109
110 drv.Vbus = batt.UN; % H-bridge DC bus voltage
111 drv.pwm.bits = 8; % PWM resolution [bits]
112 drv.pwm.levels = 2^drv.pwm.bits; % PWM levels
113 drv.dutymax = drv.pwm.levels-1; % max duty cycle code
114 drv.duty2V = drv.Vbus/drv.dutymax; % duty cycle code (0-255) to voltage
115 drv.V2duty = drv.dutymax/drv.Vbus; % voltage to duty cycle code (0-255)
116
117 %% Wheel data
118
119 % dimensions
120 wheel.h = 26e-3; % wheel height
121 wheel.r = 68e-3/2; % wheel radius
122
123 % center of mass (CoM) position
124 wheel.xb = 0; % (left) wheel CoM x-pos in body frame
125 wheel.yb = 100e-3; % (left) wheel CoM y-pos in body frame
126 wheel.zb = 0; % (left) wheel CoM z-pos in body frame
127
128 % mass
129 wheel.m = 50e-3;
130
131 % moment of inertias (MoI) wrt principal axes
132 wheel.Ixx = wheel.m/12 * (3*wheel.r^2 + wheel.h^2); % MoI along r dim
133 wheel.Iyy = wheel.m/2 * wheel.r^2; % MoI along h dim
134 wheel.Izz = wheel.Ixx; % MoI along r dim
135
136 % viscous friction coeff
137 wheel.B = 0.0015;
138
139 %% Chassis data
140
141 % dimensions
142 chassis.w = 160e-3; % frame width
143 chassis.h = 119e-3; % frame height
144 chassis.d = 80e-3; % frame depth
145
146 % center of mass (CoM) position
147 chassis.xb = 0; % CoM x-pos in body frame
148 chassis.yb = 0; % CoM x-pos in body frame
149 chassis.zb = 80e-3; % CoM x-pos in body frame
150
151 % mass
152 chassis.m = 0.456;
153
154 % moment of inertias (MoI) wrt principal axes
155 chassis.Ixx = chassis.m/12 * (chassis.w^2 + chassis.h^2); % MoI along d dim
156 chassis.Iyy = chassis.m/12 * (chassis.d^2 + chassis.h^2); % MoI along w dim
157 chassis.Izz = chassis.m/12 * (chassis.w^2 + chassis.d^2); % MoI along h dim
158
159 %% Body data
160
161 % mass
162 body.m = chassis.m + batt.m + 2*mot.stat.m;
163

```



```

164 % center of mass (CoM) position
165 body.xb = 0; % CoM x-pos in body frame
166 body.yb = 0; % CoM y-pos in body frame
167 body.zb = (1/body.m) * (chassis.m*chassis.zb + ... % CoM z-pos in body frame
168     batt.m*batt.zb + 2*mot.stat.m*mot.stat.zb);
169
170 % moment of inertias (MoI) wrt principal axes
171 body.Ixx = chassis.Ixx + chassis.m*(body.zb - chassis.zb)^2 + ... % MoI along d dim
172     batt.Ixx + batt.m*(body.zb - batt.zb)^2 + ...
173     2*mot.stat.Ixx + ...
174     2*mot.stat.m*(mot.stat.yb^2 + (body.zb - mot.stat.zb)^2);
175
176 body.Iyy = chassis.Iyy + chassis.m*(body.zb - chassis.zb)^2 + ... % MoI along w dim
177     batt.Iyy + batt.m*(body.zb - batt.zb)^2 + ...
178     2*mot.stat.Iyy + ...
179     2*mot.stat.m*(body.zb - mot.stat.zb)^2;
180
181 body.Izz = chassis.Izz + batt.Izz + ... % MoI along h dim
182     2*mot.stat.Izz + 2*mot.stat.m*mot.stat.yb^2;
183
184 %% Sensors data — Hall-effect encoder
185
186 % Hall-effect encoder
187 sens.enc.ppr = 16*4; % pulses per rotation at motor side (w/ quadrature decoding)
188 sens.enc.pulse2deg = 360/sens.enc.ppr;
189 sens.enc.pulse2rad = 2*pi/sens.enc.ppr;
190 sens.enc.deg2pulse = sens.enc.ppr/360;
191 sens.enc.rad2pulse = sens.enc.ppr/2/pi;
192
193 %% Sensors data — MPU6050 (accelerometer + gyro)
194
195 % center of mass (CoM) position
196 sens.mpu.xb = 0;
197 sens.mpu.yb = 0;
198 sens.mpu.zb = 13.5e-3;
199
200 % MPU6050 embedded accelerometer specs
201 sens.mpu.acc.bits = 16;
202 sens.mpu.acc.fs_g = 16; % full-scale in "g" units
203 sens.mpu.acc.fs = sens.mpu.acc.fs_g * g2ms2; % full-scale in [m/s^2]
204 sens.mpu.acc.g2LSB = floor(2^(sens.mpu.acc.bits-1)/sens.mpu.acc.fs_g); % sensitivity [LSB/g]
205 sens.mpu.acc.ms22LSB = sens.mpu.acc.g2LSB * ms22g; % sensitivity [LSB/(m/s^2)]
206 sens.mpu.acc.LSB2g = sens.mpu.acc.fs_g/2^(sens.mpu.acc.bits-1); % out quantization [g/LSB]
207 sens.mpu.acc.LSB2ms2 = sens.mpu.acc.LSB2g * g2ms2; % out quantization [ms2/LSB]
208 sens.mpu.acc.bw = 94; % out low-pass filter BW [Hz]
209 sens.mpu.acc.noisestd = 400e-6*sqrt(100); % output noise std [g-rms]
210 sens.mpu.acc.noisevar = sens.mpu.acc.noisestd^2; % output noise var [g^2]
211
212 % MPU6050 embedded gyroscope specs
213 sens.mpu.gyro.bits = 16;
214 sens.mpu.gyro.fs_degs = 250; % full scale in [deg/s (dps)]
215 sens.mpu.gyro.fs = sens.mpu.gyro.fs_degs * deg2rad; % full scale in [rad/s]
216 sens.mpu.gyro.degs2LSB = floor(2^(sens.mpu.gyro.bits-1)/sens.mpu.gyro.fs_degs); % sensitivity [LSB/degs]
217 sens.mpu.gyro.rads2LSB = sens.mpu.gyro.degs2LSB * rad2deg; % sensitivity [LSB/rads]
218 sens.mpu.gyro.LSB2degs = sens.mpu.gyro.fs_degs/2^(sens.mpu.gyro.bits-1); % out quantization [degs/LSB]
219 sens.mpu.gyro.LSB2rads = sens.mpu.gyro.LSB2degs * deg2rad; % out quantization [rads/LSB]
220 sens.mpu.gyro.bw = 98; % out low-pass filter BW [Hz]
221 sens.mpu.gyro.noisestd = 5e-3*sqrt(100); % output noise std [degs-rms]
222 sens.mpu.gyro.noisevar = sens.mpu.gyro.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 $x = [\gamma, \vartheta, \dot{\gamma}, \dot{\vartheta}]^T$ 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 ϑ , 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

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

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 Hz.

Notes:

- 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

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

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

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

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

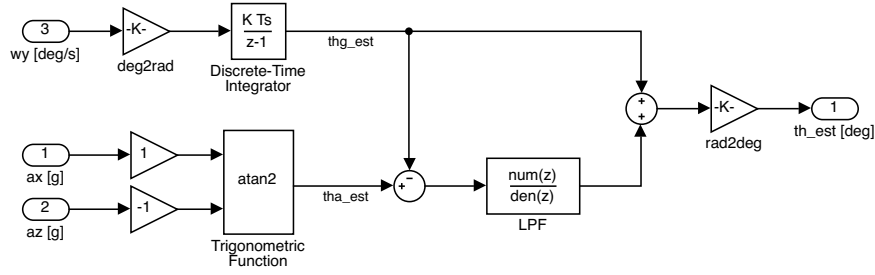
$$\begin{aligned} \hat{\vartheta}[k] &= (1 - C) \hat{\vartheta}[k - 1] + C \hat{\vartheta}_a[k] + (1 - C) T y_g[k] \\ &= C \hat{\vartheta}_a[k] + (1 - C) (\hat{\vartheta}[k - 1] + T y_g[k]) \end{aligned} \quad (5)$$

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

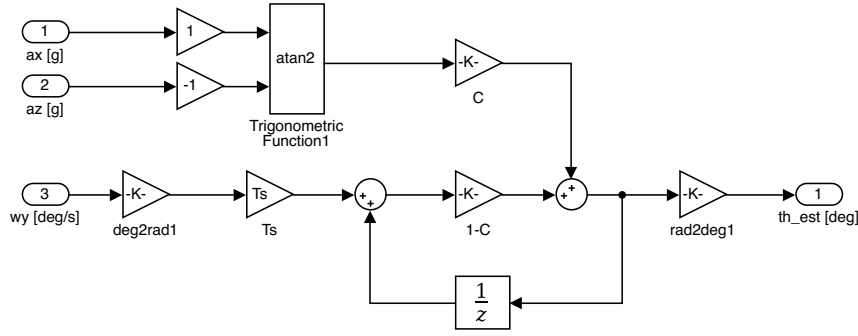
- when using the balancing robot model implemented with the “chain of integrators” approach, an algebraic loop could arise if the complementary filters are implemented either as in Fig. 6a, and the transfer function $H(z)$ of the discretized low-pass filter is not strictly proper, or as in Fig. 6b.

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 originates¹.

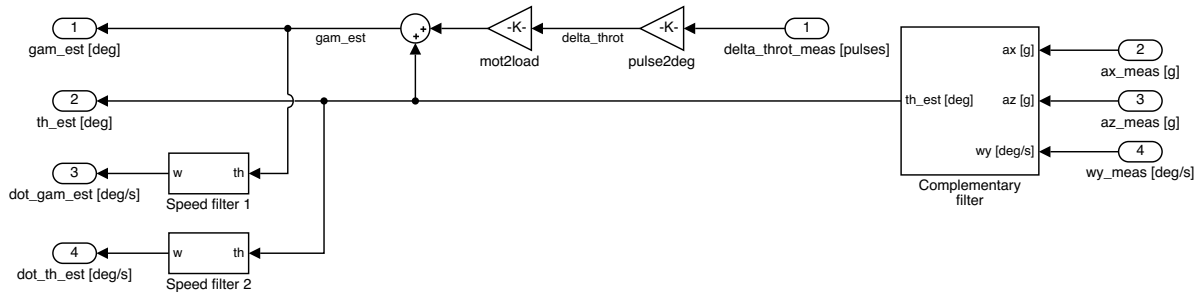
¹The real reason for the existence of the algebraic loop is that both the models of the motor voltage driver (see



(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 γ , use the identity $\Delta\vartheta_{rot} = N(\gamma - \vartheta)$ to obtain the estimate

$$\hat{\gamma} = \Delta\vartheta_{rot}/N + \hat{\vartheta} \quad (6)$$

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

$$H_\omega(z) = \frac{1 - z^{-N}}{NT} \quad \text{with} \quad N = 3 \quad (7)$$

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 γ^* . 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 $T = 0.01$ s; let

$$\begin{cases} \mathbf{x}[k+1] = \mathbf{\Phi} \mathbf{x}[k] + \mathbf{\Gamma} u[k] \\ y[k] = \mathbf{H} \mathbf{x}[k] \end{cases} \quad (8)$$

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 γ . Therefore, the matrix \mathbf{H} is chosen equal to $\mathbf{H} = [1, 0, 0, 0]$.

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

$$u[k] = N_u r[k] - \mathbf{K} (\hat{\mathbf{x}}[k] - \mathbf{N}_x r[k]) = -\mathbf{K} \mathbf{x}[k] + \underbrace{(N_u + \mathbf{K} \mathbf{N}_x)}_{= N_r} r[k] \quad (9)$$

where $r = \gamma^*$ is the wheel angle reference signal, and \mathbf{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 N_x and N_u are determined by solving the following set of linear equations:

$$\begin{bmatrix} \mathbf{\Phi} - \mathbf{I} & \mathbf{\Gamma} \\ \mathbf{H} & 0 \end{bmatrix} \begin{bmatrix} N_x \\ N_u \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} \quad (10)$$

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

$$J = \sum_{k=0}^{+\infty} \mathbf{x}^T[k] \mathbf{Q} \mathbf{x}[k] + \rho r u^2[k] \quad (11)$$

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

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

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

$$\mathbf{Q} = \text{diag} \left\{ \frac{1}{\bar{\gamma}^2}, \frac{1}{\bar{\vartheta}^2}, 0, 0 \right\}, \quad r = \frac{1}{\bar{u}^2} \quad (13)$$

where

$$\bar{\gamma} = \pi/18, \quad \bar{\vartheta} = \pi/360, \quad \bar{u} = 1 \quad (14)$$

In the \mathbf{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 ρ in (11) is used to adjust the relative weighting between the state and input

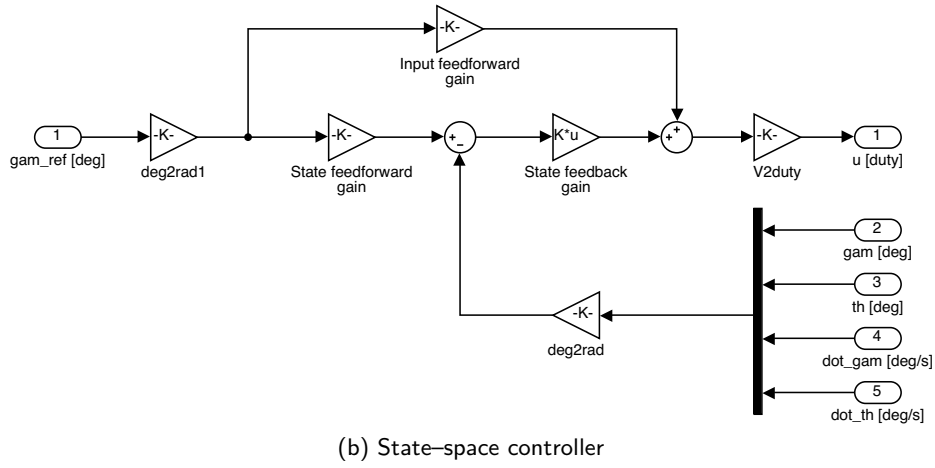
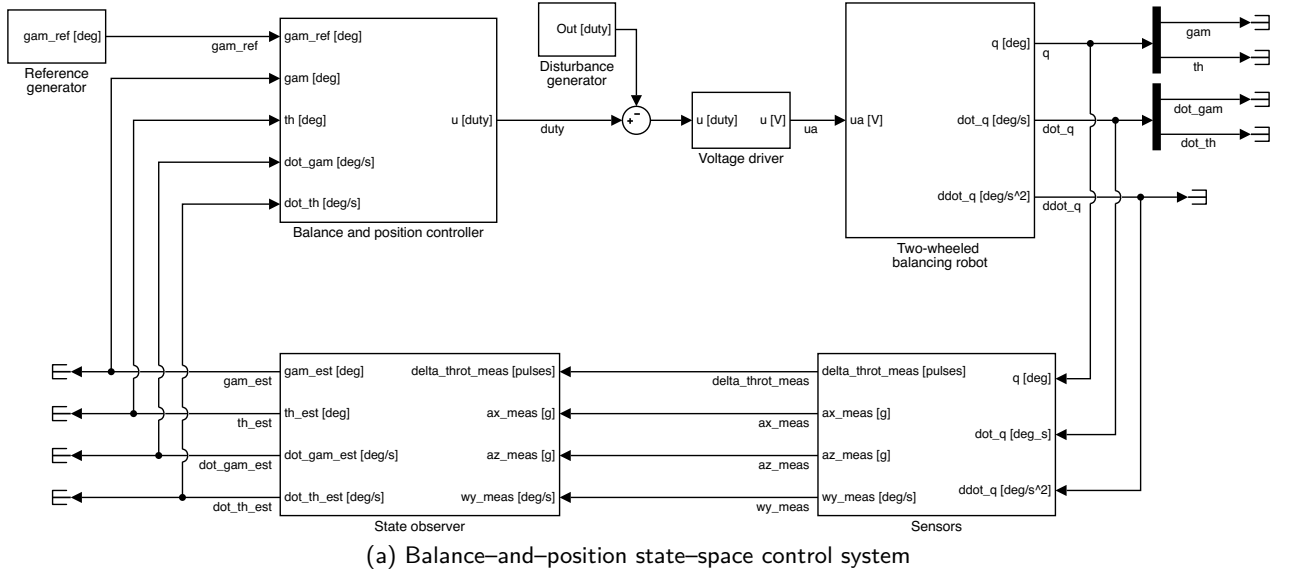


Figure 7: Possible Simulink implementation of the balance-and-position state-space control system (for the *nominal* perfect tracking of constant position set-points).

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

$$\rho \in \{ 500, 5000 \} \quad (15)$$

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 $x(0) = [0, \pi/36, 0, 0]^T$ (i.e. initial body tilt angle equal to $\vartheta(0) = 5\text{deg}$), 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 $\mathbf{x}(0) = [0, 0, 0, 0]^T$, step reference input applied at $t = 0$ with amplitude $\gamma^* = 0.1/r$ rad, where r is the wheel radius (this choice corresponds to a longitudinal position displacement of 10 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 $5.0/k_{duty \rightarrow V} \approx 115$ (where $k_{duty \rightarrow V}$ is the voltage driver input-to-output conversion gain), applied at the voltage driver input at $t = 10$ s. This is equivalent to a voltage disturbance of 5 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

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

where x_I is the integrator state variable². After introducing the augmented state vector $\mathbf{x}_e = [x_I, \mathbf{x}]^T$ of the augmented state system

$$\Sigma_e : \begin{bmatrix} x_I[k+1] \\ \mathbf{x}[k+1] \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \mathbf{H} \\ \mathbf{0} & \Phi \end{bmatrix}}_{\triangleq \Phi_e} \underbrace{\begin{bmatrix} x_I[k] \\ \mathbf{x}[k] \end{bmatrix}}_{\triangleq \mathbf{x}_e} + \underbrace{\begin{bmatrix} 0 \\ \Gamma \end{bmatrix}}_{\triangleq \Gamma_e} u[k] - \begin{bmatrix} 1 \\ \mathbf{0} \end{bmatrix} r[k] \quad (17)$$

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

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

The augmented state feedback matrix $\mathbf{K}_e = [K_I, \mathbf{K}]^T$ can be designed again by resorting to LQR methods. Compared to the design of point **(S7)**, in this case the cost matrix \mathbf{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:

$$q_{11} \in \{0.1, 1\} \quad (19)$$

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

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:

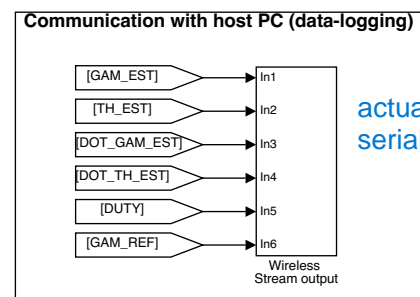
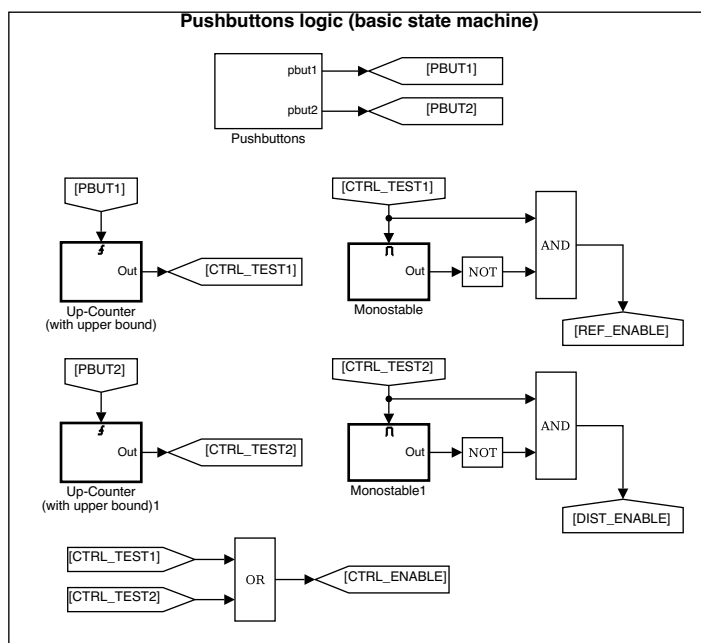
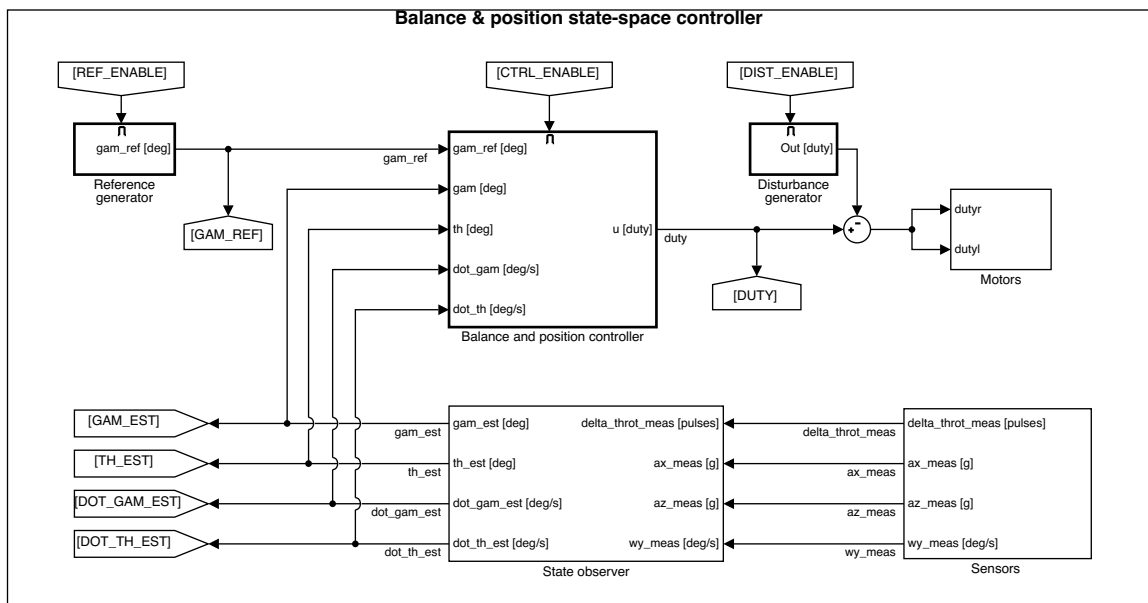
- make a copy of the Simulink model prepared in point (S8) of Sec. 4.2 for the numerical simulations (see also Fig. 7).
- replace the models of the balancing robot and the sensors (encoder and MPU) with the blocks of the **Balancing Robot Toolbox** (BRT) that allows to interface with the robot hardware.
- 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 $T = 0.01$ s.

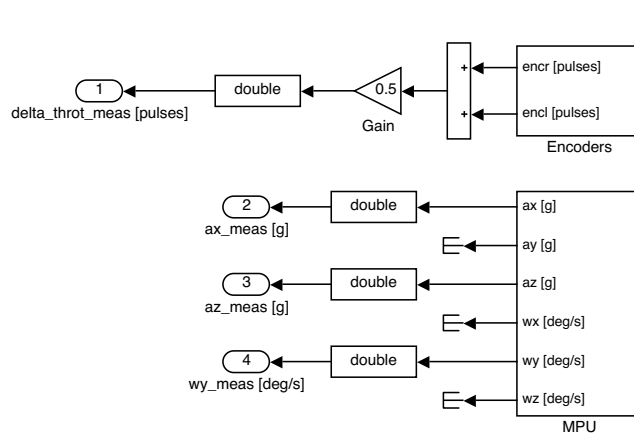
A possible implementation is shown in Fig. 9. In addition to the base controller, the proposed implementation includes some extra “logic”, to enable the generation of the position reference and the load disturbance via the two pushbuttons available on the robot. The logic operates as follows:

- 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 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.

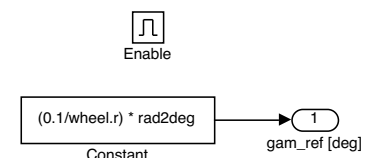
Both the balance-and-position controller, and the reference/disturbance generator are *enabled* blocks (use a Ports & Subsystems → Enable to create a subsystem with an enable port). When working



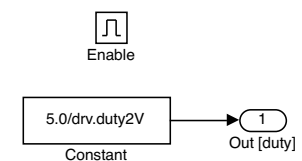
(a) Balance-and-position state-space control system



(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 \rightarrow 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 hold 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 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:

$$\Delta\vartheta_{rot} = \frac{\Delta\vartheta_{rot,r} + \Delta\vartheta_{rot,l}}{2} \quad (20)$$

Moreover, note that a Signal Attributes \rightarrow Data Type Conversion has been used to convert the outputs of the BRT \rightarrow Sensors \rightarrow Encoders and BRT \rightarrow Sensors \rightarrow 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 $\gamma^* = 0.1/r$ rad, where r is the wheel radius, which corresponds to a longitudinal position displacement of 10 cm, and a load disturbance equal to $u_d = 5.0/k_{duty \rightarrow V} \approx 115$, where $k_{duty \rightarrow V}$ is the PWM duty-cycle to voltage conversion gain, which corresponds to a voltage disturbance of 5.0 V. The position reference and/or the load disturbance should be enabled at least 10 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) ψ 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):

$$u_r = u_\Sigma + u_\Delta, \quad u_l = u_\Sigma - u_\Delta \quad (21)$$

where u_Σ is a “common-mode” command generated by the longitudinal controller, and u_Δ 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:

$$K_P = 3.3, \quad K_I = 0.7 \quad (22)$$

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

$$\psi = \frac{r}{w} (\vartheta_r - \vartheta_l) \quad (23)$$

where ϑ_r and ϑ_l are the wheels angles derived from the encoders measurements. From (6) and (7), these quantities are equal to:

$$\vartheta_l = \frac{\Delta\vartheta_{rot,l}}{N} + \vartheta, \quad \vartheta_r = \frac{\Delta\vartheta_{rot,r}}{N} + \vartheta \quad (24)$$

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

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

$$\gamma = \frac{\vartheta_r + \vartheta_l}{2} \quad (25)$$

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

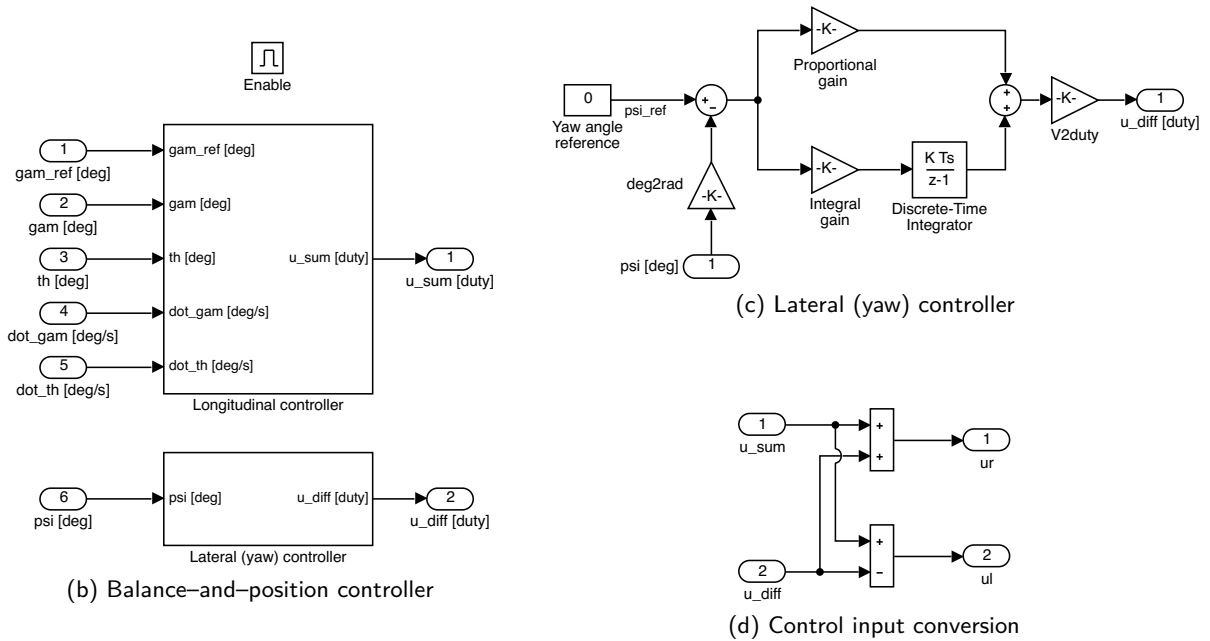
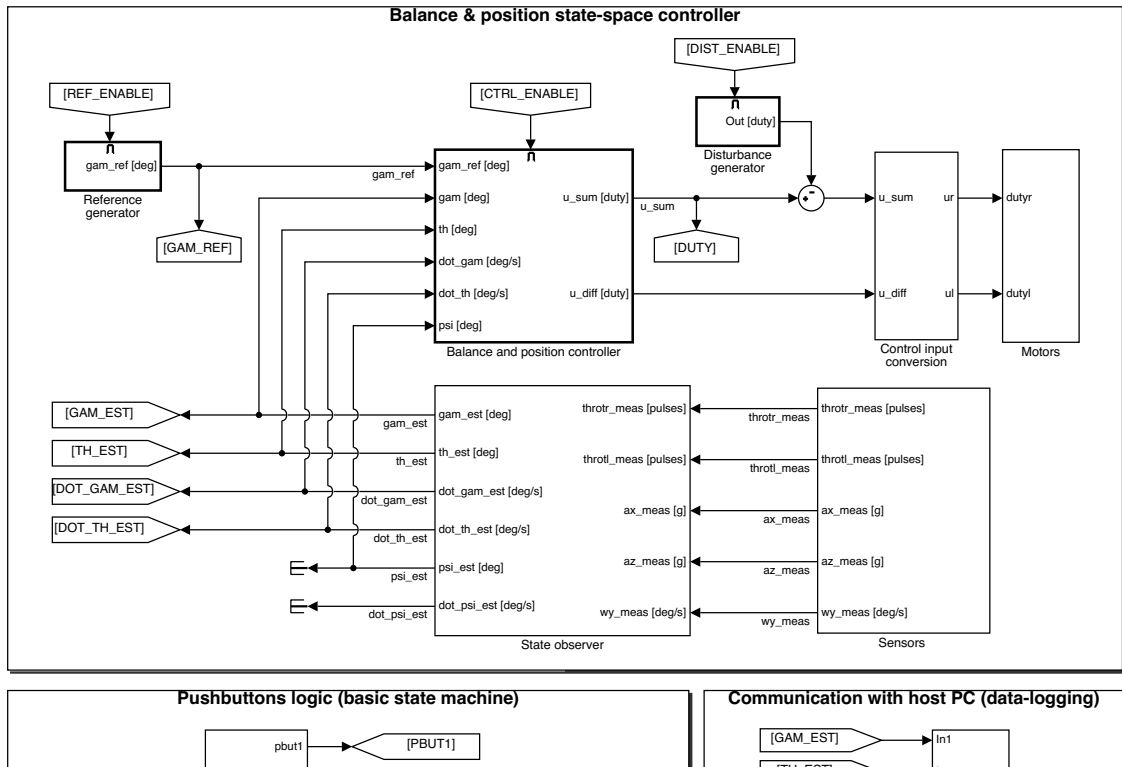
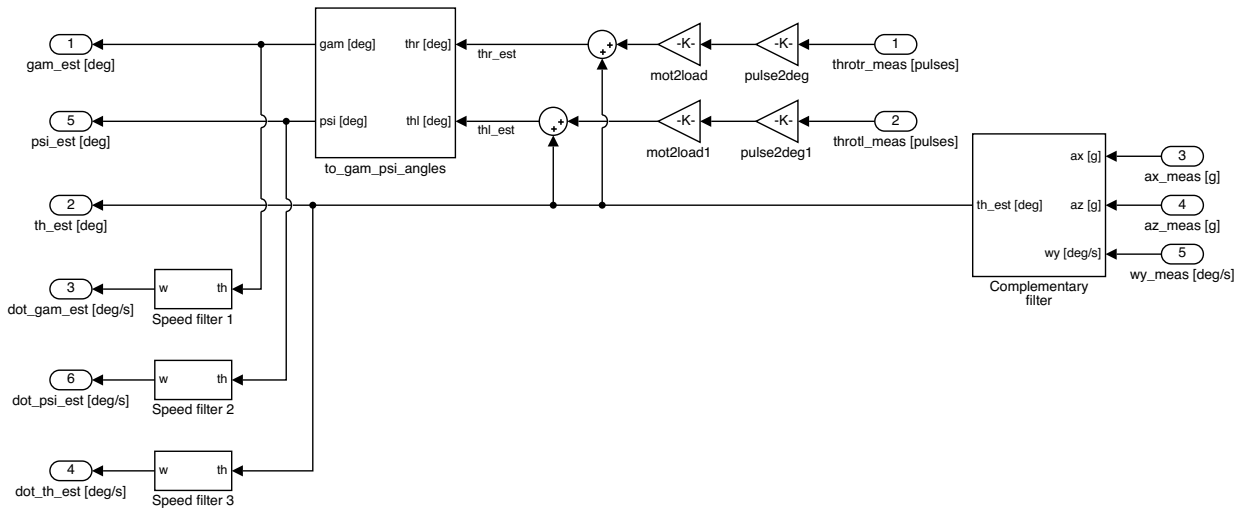
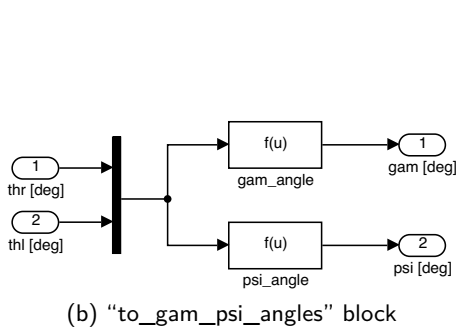


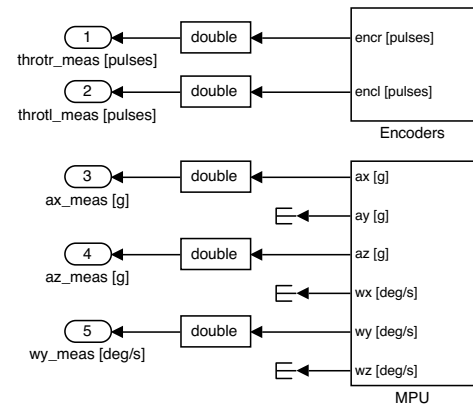
Figure 10: Possible Simulink implementation of a combined longitudinal and heading-angle controller for the balancing robot.



(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.

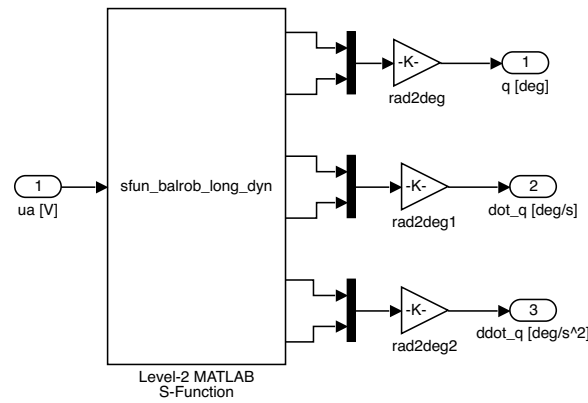


Figure 12: Simulink model implementation details: electromechanical dynamics (implemented by using a MATLAB S-Function).

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 \mathbf{q} , and its first two time derivatives $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$). The 5 expected parameters are the vector of initial conditions \mathbf{x}_0 (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 \mathbf{q} , and their derivatives in $\dot{\mathbf{q}}$. The acceleration vector $\ddot{\mathbf{q}}$ 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.SetPreCompInPortInfoToDynamic;
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;

```

```

32     block.OutputPort(k).DatatypeID = 0; % double
33     block.OutputPort(k).Complexity = 'Real';
34 end;
35
36 % Register parameters
37 block.NumDialogPrms = 5;
38
39 % Register sample times
40 % [0 offset] : Continuous sample time
41 block.SampleTimes = [0 0];
42
43 % Specify the block simStateCompliance.
44 % 'DefaultSimState', < Same sim state as a built-in block
45 block.SimStateCompliance = 'DefaultSimState';
46
47 % Register block methods
48 block.RegBlockMethod('Start', @Start);
49 block.RegBlockMethod('SetInputPortSamplingMode', @SetInputPortSamplingMode);
50 block.RegBlockMethod('Outputs', @Outputs); % Required
51 block.RegBlockMethod('Derivatives', @Derivatives);
52 block.RegBlockMethod('Terminate', @Terminate); % Required
53
54 for k = 1:block.NumOutputPorts,
55     block.OutputPort(k).SamplingMode = 0;
56 end;
57
58 end % setup
59
60
61 function SetInputPortSamplingMode(block, port, mode)
62
63 block.InputPort(port).SamplingMode = mode;
64
65 end % SetInputPortSamplingMode
66
67
68 function Start(block)
69
70 %% Get init state
71
72 x0 = block.DialogPrm(1).Data; % get init state
73
74 %% Set init state
75
76 block.ContStates.Data(1) = x0(1); % gam(0)
77 block.ContStates.Data(2) = x0(2); % th(0)
78
79 block.ContStates.Data(3) = x0(3); % dot_gam(0)
80 block.ContStates.Data(4) = x0(4); % dot_th(0)
81
82 end % Start
83
84
85 function Outputs(block)
86
87 %% Extract state components
88
89 gam = block.ContStates.Data(1);
90 th = block.ContStates.Data(2);
91
92 dot_gam = block.ContStates.Data(3);
93 dot_th = block.ContStates.Data(4);
94

```

```

95 %% Get accelerations
96
97 [ddot_gam, ddot_th] = get_acc(block);
98
99 %% Set outputs
100
101 block.OutputPort(1).Data = gam;
102 block.OutputPort(2).Data = th;
103
104 block.OutputPort(3).Data = dot_gam;
105 block.OutputPort(4).Data = dot_th;
106
107 block.OutputPort(5).Data = ddot_gam;
108 block.OutputPort(6).Data = ddot_th;
109
110 end % Outputs
111
112
113 function Derivatives(block)
114
115 %% Extract state components
116
117 dot_gam = block.ContStates.Data(3);
118 dot_th = block.ContStates.Data(4);
119
120 %% Get accelerations
121
122 [ddot_gam, ddot_th] = get_acc(block);
123
124 %% Set state derivative
125
126 block.Derivatives.Data(1) = dot_gam;
127 block.Derivatives.Data(2) = dot_th;
128
129 block.Derivatives.Data(3) = ddot_gam;
130 block.Derivatives.Data(4) = ddot_th;
131
132 end % Derivatives
133
134
135 function Terminate(block)
136
137 end % Terminate
138
139
140 function [ddot_gam, ddot_th] = get_acc(block)
141
142 %% Get inputs and parameter structs
143
144 % parameters
145 body = block.DialogPrm(2).Data; % body data struct
146 mot = block.DialogPrm(3).Data; % mot data struct
147 gbox = block.DialogPrm(4).Data; % gbox data struct
148 wheel = block.DialogPrm(5).Data; % wheel data struct
149
150 % input voltages
151 ua = block.InputPort(1).Data; % armature voltage (right/left motor)
152
153 %% Extract state components
154
155 th = block.ContStates.Data(2);
156 dot_gam = block.ContStates.Data(3);
157 dot_th = block.ContStates.Data(4);

```



```

158
159 %% Extract params
160
161 % body params
162 l      = body.zb;
163 mb      = body.m;
164 Ibyy    = body.Iyy;
165
166 % wheel params
167 w      = 2*wheel.yb;
168 r      = wheel.r;
169 mw      = wheel.m;
170 Iwyy    = wheel.Iyy;
171
172 % (motor) rotor params
173 zbrot   = mot.rot.zb;
174 mrot    = mot.rot.m;
175 Irotty  = mot.rot.Iyy;
176
177 % gear ratio
178 n      = gbox.N;
179
180 % friction params
181 bw = wheel.B;      % wheel viscous fric coeff
182 bm = mot.B;        % motor viscous fric coeff (motor side)
183 bg = gbox.B;       % gbox viscous fric coeff (load side)
184 b  = n^2*bm+bg;    % motor+gbox viscous fric coeff (load side)
185
186 % gravity acc
187 g = 9.81;
188
189 %% Get motor torques
190
191 % back-EMFs
192 ue = mot.Ke * n*(dot_gam-dot_th);
193
194 % motor torque (single motor)
195 tau = n*mot.Kt * (ua-ue)/mot.R;
196
197 %% Evaluate accelerations of generalised coords
198
199 % inertia matrix
200 MM = zeros(2,2);
201
202 MM(1,1) = 2*Iwyy + 2*Irotty*n^2 + (mb + 2*(mrot+mw))*r^2;
203 MM(1,2) = 2*(1-n)*n*Irotty + r*(l*mb + 2*mrot*zbrot)*cos(th);
204
205 MM(2,1) = MM(1,2);
206 MM(2,2) = Ibyy + 2*(1-n)^2*Irotty + mb*l^2 + 2*mrot*zbrot^2;
207
208 % Coriolis + centrifugal terms matrix
209 CC = zeros(2,2);
210
211 CC(1,1) = 0;
212 CC(2,1) = 0;
213
214 CC(2,2) = 0;
215 CC(1,2) = -r*(mb*l + 2*mrot*zbrot)*sin(th)*dot_th;
216
217 % viscous friction matrix
218 Fv = zeros(2,2);
219
220 Fv(1,1) = 2*(b+bw);

```

```

221 Fv(1,2) = -2*b;
222
223 Fv(2,1) = Fv(1,2);
224 Fv(2,2) = 2*b;
225
226 % gravity loading
227 GG = zeros(2,1);
228 GG(2) = -g*(mb*l + 2*mrot*zbroth)*sin(th);
229
230 % generalized actuator forces
231 TT = zeros(2,1);
232 TT(1) = 2*tau;
233 TT(2) = -2*tau;
234
235 % get accelerations of generalised coords (q = [gam, th].')
236 dotq = [dot_gam, dot_th].';
237 ddotq = MM \ (TT - CC*dotq - Fv*dotq - GG);
238
239 ddot_gam = ddotq(1);
240 ddot_th = ddotq(2);
241
242 end % get_acc

```