Haptic Human-Robot Interfaces: Lab 1 - answers

3 Lab instructions

3.2 Hall sensor calibration

a) Over a full rotation of the magnet, the Hall-effect sensor measures a magnetic field intensity that seems to follow a sine wave. This was expected, since the rotation of the magnet makes a rotating magnetic field, while only the vertical component is measured by the sensor.

It is not possible to use this setup to compute the angular position of the magnet over 360°, because there are two angles associated to each voltage. The "voltage to angle" function is not bijective.

A solution to this issue is to add a second Hall-effect sensor, at a 90° angle with respect to the other one. There is then enough information to compute the angle over 360°, using the atan2() function. Note that this method is not sensitive to the magnet distance and strength, since the direction of the magnetic field is directly measured. Ready-to-use integrated circuits are available and are usually called "magnetic encoders".

b) After recording a few seconds of the Hall-effect sensor voltage, we can compute the standard deviation. It should typically be around 0.78 mV, as measured on one of the paddles.

c) We first record the encoder angle and the Hall voltage, while moving the paddle over the full range. To fit the model to the acquired data, there are at least these three options on MATLAB:

- Use the "Curve fitting" app, if the "Curve fitting toolbox" is installed. Then choose the "polynomial model", degree 1.
- Use the fit() function (the "Curve fitting toolbox" should also be installed):
  `[fitresult, ~] = fit(x, y, fittype('poly1'));`
- Use the backslash operator, which solves a problem of the type $Ac = b$ with the command:
  `c = A\b`
  We require the model to be: $y = c_1 x + c_0$ (with x the Hall voltage, and y the paddle angle) which be written in the matrix form:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = c_1 \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} + c_0 \quad \Leftrightarrow \quad \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \quad \Leftrightarrow \quad b = Ac$$

This can be solved by MATLAB:

```
log3_2_d = hri_load_logfile('q3_2_d.csv');
t = log3_2_d.timestamp__s_;
x = log3_2_d.hall_voltage__V_;
y = log3_2_d.encoder_paddle_pos__deg_;

coefs = [ones(size(x)) x] \ y; % Resolve A*x=b with x=A\b.

plot(t, y, t, coefs(2)*x + coefs(1));
legend('Encoder angle', 'Calibrated Hall angle');
xlabel('Time [s]');
```

```
ylabel('Angle [deg]');
```

We get typically $y = c_1 x + c_0$ with: $\begin{cases} c_1 = -33.0037\ °/\mathrm{V} \\ c_0 = 81.9710\ ° \end{cases}$ but note that these numerical values can vary depending on your Hall sensor's type, its distance from the magnet, and the relative rotation of the magnet with respect to the paddle shaft.

d) By computing the maximum difference between the encoder angle and the calibrated Hall angle, we get the maximum linearity error. On the paddle, it should typically be around 1~3°, when the paddle is far from the center position. Compared to the movement range of ~100°, this makes a full-scale error of 1%, which is good enough for most applications. If this linear approximation is not enough, a higher-degree model could be fitted.

e) A simple implementation in the firmware is:

```
hapt_hallPaddleAngle = hapt_hallVoltage * (-33.0037f) + 81.9710f;
```

3.3 Velocity and acceleration computation

3.3.1 Hall-effect sensor

a) The speed can be computed in the firmware:

```
hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPos) / dt;
hapt_paddlePrevPos = hapt_paddleAngle;
```

We obtain 97°/s of noise when still.

b) The speed can be computed in the firmware in a similar way:

```
hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPrevPos) / (2.0f*dt);
hapt_paddlePrevPrevPos = hapt_paddlePrevPos;
hapt_paddlePrevPos = hapt_paddleAngle;
```

For the noise, we obtain 49°/s when still.

c)

```
hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPos) / dt;
hapt_paddleAccel = (hapt_paddleSpeed - hapt_paddlePrevSpeed) / dt;
hapt_paddlePrevPos = hapt_paddleAngle;
hapt_paddlePrevSpeed = hapt_paddleSpeed;
```

We obtain a standard deviation of 480000°/s^2 when still.

d)

```
hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPrevPos) / (2.0f*dt);
hapt_paddleAccel = (hapt_paddleSpeed - hapt_paddlePrevPrevSpeed) / (2.0f*dt);
hapt_paddlePrevPrevPos = hapt_paddlePrevPos;
hapt_paddlePrevPrevSpeed = hapt_paddlePrevSpeed;
hapt_paddlePrevPos = hapt_paddleAngle;
hapt_paddlePrevSpeed = hapt_paddleSpeed;
```

We obtain a standard deviation of 240000°/s^2 when still. Again, we get around half of the values obtained previously.

e) The noise comes from the analog noise, which is due to the environment noise, the Hall-effect sensor's intrinsic noise, and the board amplifier noise. This noise exists even when the paddle is not moving.

3.3.2 Incremental encoder

a) When the paddle is still, there is no noise at all, so the speed is also zero without noise. (But note that the speed signal is noisy when you move the paddle)

b) With the differentiation over two samples, the noise when standing still is 0 again. But the noise when moving will be visibly lower (by a factor of almost 2).

c) Same for acceleration, except that the noise when moving is even more.

d) Again, the noise when moving is lower when differentiated over 2 sampling periods.

e) The noise comes from the fact we differentiate a signal that has discrete values. Intuitively, if the speed is very small, we may alternatively get 0 or 1 encoder steps during a sampling period. Therefore, this noise only appears when we are moving the paddle. This is also known as quantization noise[1].

3.3.3 Conclusion

f) The Hall-effect sensor is inherently noisy. While this is acceptable to measure the angle of the paddle, this noise gets amplified when differentiating the signal, which makes the speed and acceleration extremely noisy, and practically unusable.
However, this sensor has two advantages compared to the motor encoder:
- It is absolute, as opposed to the incremental encoder which needs to be initialized properly at startup, using a reference position.
- It measures the paddle angle directly (at the output of the transmission), even if the steel cable slips or breaks.

[1] Note that with our setup, quantization noise also exists for the Hall effect sensor, because we are reading the "analog" output of this sensor using an analog-to-digital converter (ADC), and the output value of the ADC is ultimately digital and therefore discrete. But this noise is negligible compared to the analog noise.

3.4 Filters implementation

a) An example of implementation can be found in the Appendix 4.2.
The cutoff frequency for the speed filter was set to 100 Hz, and set to 10 Hz for the acceleration filter. Some typical results are displayed here:

Case	RMS noise before filtering	RMS noise after filtering
Hall sensor, speed	107°/s	18.0°/s
Hall sensor, acceleration	541000 °/s^2	6820 °/s^2
Encoder, speed	13.7 °/s	2.31°/s
Encoder, acceleration	66300 °/s^2	867 °/s^2

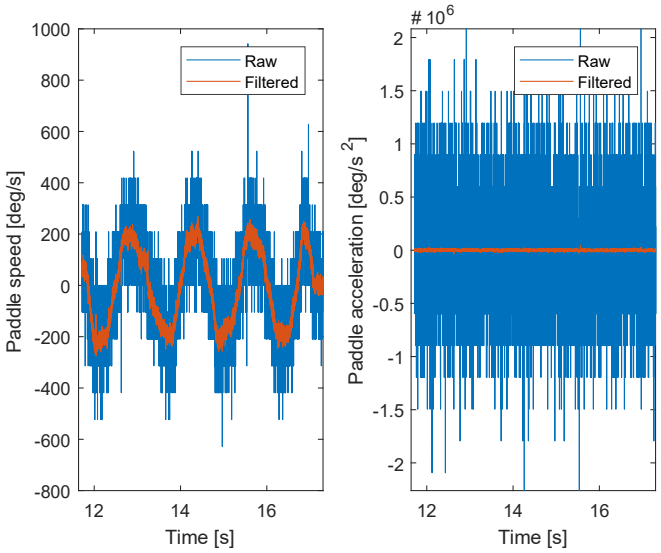

Figure 1: filtering results with the Hall-effect sensor

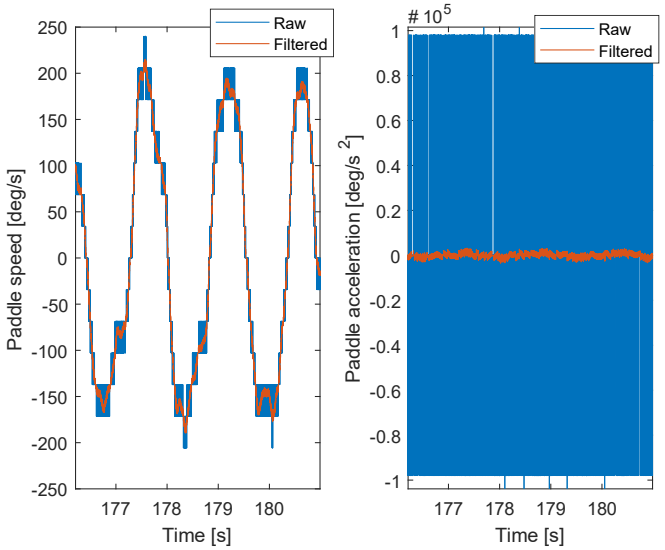

Figure 2: filtering results with the encoder

For both the Hall sensor and the encoder, the filtered speed is smooth enough to be usable for the control. However, the filtered accelerations remain extremely noisy, even if the cutoff frequency was set very low. In particular, the filtered acceleration from the Hall is completely unusable.

b) By plotting all 3 signals together (raw signal, filtered at 200 Hz, filtered at 20 Hz) you can see that the filtered speeds are much smoother (and the lower the cut-off frequency, the smoother the signal); however, the filtering adds a delay to the filtered signal (which is most conspicuous in the 20 Hz filter), and also when changing the speed very quickly, the filtered signals have attenuated peaks. The advantage of filtering using this low-pass filter is that it helps to get rid of the very high-frequency noise generated in numerical differentiation. The disadvantage is that it will also warp the actual value of the signal, and the stronger the filtering, the more severe this warping becomes[2]. Therefore, when choosing the cut-off frequency, there is a trade-off to be considered between the smoothness of the signal and its accuracy both in time and in values compared to the original one.

3.5 Motor characteristics

a) When subject to a constant torque, the rotor accelerates quickly, and then seems to stabilize at a steady-state speed.
The plot of the speed is shown on Figure 3. At the beginning of the movement (0.05 s and before), its acceleration is constant, its velocity is a ramp, and the angle is parabolic.
After 0.05 s, the acceleration decreases and goes to 0, and the speed saturates to the no-load maximum speed of the motor at 24V.

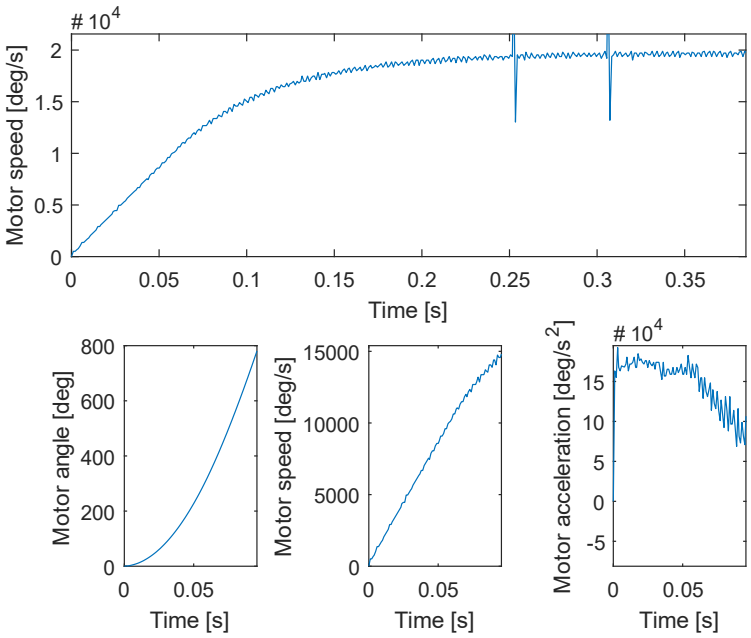

Figure 3: measured motor angle, speed and acceleration over time.

b) Dynamical model of the system:

$$\sum T_{ext} = I\,\ddot{\theta} \quad \Rightarrow \quad T_{motor} = I_{rotor}\,\ddot{\theta} \quad \Leftrightarrow \quad \ddot{\theta} = \frac{T_{rotor}}{I_{rotor}}$$

[2] Note that this is a very simple filter. There are more advanced higher-order filters in which these disadvantages are less pronounced, but for the purposes of our labs this filter is sufficient.

We can then integrate this equation twice to predict the angle over time:

$$\ddot{\theta} \text{ constant} \quad \Rightarrow \quad \dot{\theta}(t) = \ddot{\theta}t \quad \Rightarrow \quad \theta(t) = \frac{\ddot{\theta}}{2}t^2 = \frac{1}{2}\frac{T_{motor}}{I_{rotor}}t^2$$

The simulated and measured angle trajectories are shown in Figure 4. While both curves are close, the error increases over time. This is because the simulation model is simplistic: in the real motor, there is a viscous friction component that is acting against the motor movement, and its magnitude is proportional to the motor speed. This is why the simulated system accelerates faster than the actual one, and the actual system comes to a steady-state speed (at which the viscous friction term becomes equal to the applied motor torque).

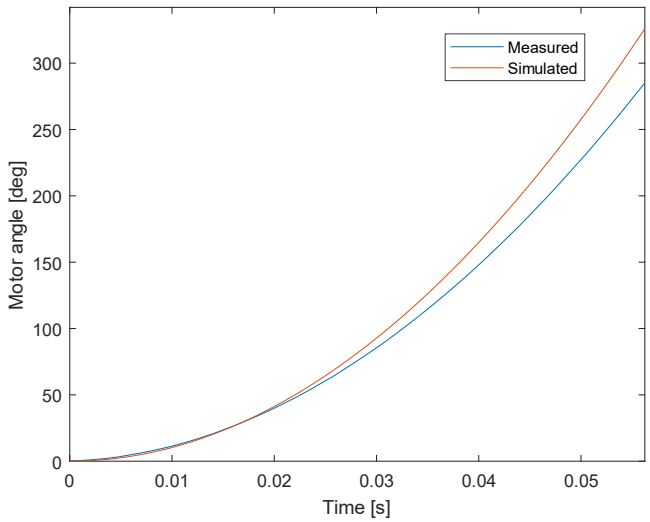

Figure 4: comparison of the simulated and measured motor angle trajectory

The MATLAB code for generating the simulated angle is:

```
% Constants.
motorTorque = 0.005; % [N.m].
motorRotorInertia = (11 + 1.44) * 1e-3 * 1e-4; % [kg.m^2].
time = linspace(0, 0.05); % [s].

% Simulate the movement.
simuAngle = rad2deg(motorTorque / motorRotorInertia * (time.^2) / 2); % [deg].
```

4 Appendix

4.1 Firmware source code for the part 3.2

```
#include "haptic_controller.h"
#include "communication.h"
#include "drivers/adc.h"
#include "drivers/incr_encoder.h"
#include "drivers/hall.h"
#include "drivers/callback_timers.h"
#include "lib/utils.h"
#include "torque_regulator.h"

#define DEFAULT_HAPTIC_CONTROLLER_PERIOD 350 // Default control loop period [us].

volatile uint32_t  hapt_timestamp; // Time base of the controller, also used to timestamp the samples
sent by streaming [us].
volatile float32_t hapt_hallVoltage; // Hall sensor output voltage [V].
volatile float32_t hapt_encoderPaddleAngle; // Paddle angle measured by the incremental encoder [deg].
volatile float32_t hapt_paddleAngle; // Paddle angle [deg].
volatile float32_t hapt_paddleSpeed; // Paddle speed [deg/s].
volatile float32_t hapt_paddleAccel; // Paddle acceleration [deg/s^2].
volatile float32_t hapt_paddlePrevPos; // Previous paddle angle [deg].
volatile float32_t hapt_paddlePrevPrevPos; // Sample before the previous paddle angle [deg].
volatile float32_t hapt_paddlePrevSpeed; // Previous paddle speed [deg/s].
volatile float32_t hapt_paddlePrevPrevSpeed; // Sample before the previous paddle speed [deg/s].
volatile float32_t hapt_motorTorque; // Motor torque [N.m].

void hapt_Update(void);

/**
  * @brief Initializes the haptic controller.
  */
void hapt_Init(void)
{
    hapt_timestamp = 0;
    hapt_motorTorque = 0.0f;
    hapt_paddlePrevPos = 0.0f;
    hapt_paddlePrevSpeed = 0.0f;
    hapt_paddlePrevPrevPos = 0.0f;
    hapt_paddlePrevPrevSpeed = 0.0f;

    // Make the timers call the update function periodically.
    cbt_SetHapticControllerTimer(hapt_Update, DEFAULT_HAPTIC_CONTROLLER_PERIOD);

    // Share some variables with the computer.
    comm_monitorUint32Func("timestep [us]", cbt_GetHapticControllerPeriod,
                           cbt_SetHapticControllerPeriod);
    comm_monitorFloat("motor_torque [N.m]", (float32_t*)&hapt_motorTorque, READWRITE);
    comm_monitorFloat("encoder_paddle_pos [deg]", (float32_t*)&hapt_encoderPaddleAngle, READONLY);
    comm_monitorFloat("paddle_pos [deg]", (float32_t*)&hapt_paddleAngle, READONLY);
    comm_monitorFloat("paddle_speed [deg/s]", (float32_t*)&hapt_paddleSpeed, READONLY);
    comm_monitorFloat("paddle_accel [deg/s^2]", (float32_t*)&hapt_paddleAccel, READONLY);
    comm_monitorFloat("hall_voltage [V]", (float32_t*)&hapt_hallVoltage, READONLY);
}

/**
  * @brief Updates the haptic controller state.
  */
void hapt_Update()
{
    float32_t motorShaftAngle; // [deg].

    // Compute the dt (uncomment if you need it).
    float32_t dt = ((float32_t)cbt_GetHapticControllerPeriod()) / 1000000.0f; // [s].

    // Increment the timestamp.
    hapt_timestamp += cbt_GetHapticControllerPeriod();

    // Get the Hall sensor voltage.
    hapt_hallVoltage = hall_GetVoltage();

    // Compute the paddle angle from the desired sensor.
```

```
    hapt_paddleAngle = hapt_hallVoltage * (-33.0037f) + 81.9710f;
    //hapt_paddleAngle = enc_GetPosition() / REDUCTION_RATIO;

    // Compute the speed and accleration from the Hall paddle angle.
    hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPrevPos) / (2.0f*dt);
    hapt_paddleAccel = (hapt_paddleSpeed - hapt_paddlePrevPrevSpeed) / (2.0f*dt);

    hapt_paddlePrevPrevPos = hapt_paddlePrevPos;
    hapt_paddlePrevPrevSpeed = hapt_paddlePrevSpeed;
    hapt_paddlePrevPos = hapt_paddleAngle;
    hapt_paddlePrevSpeed = hapt_paddleSpeed;

    // Get the encoder position.
    motorShaftAngle = enc_GetPosition();
    hapt_encoderPaddleAngle = motorShaftAngle / REDUCTION_RATIO;

    // Compute the motor torque, and apply it.
    //hapt_motorTorque = 0.0f;
    torq_SetTorque(hapt_motorTorque);
}
```

4.2 Firmware source code for the part 3.4

```
#include "haptic_controller.h"
#include "communication.h"
#include "drivers/adc.h"
#include "drivers/incr_encoder.h"
#include "drivers/hall.h"
#include "drivers/callback_timers.h"
#include "lib/utils.h"
#include "torque_regulator.h"

#define DEFAULT_HAPTIC_CONTROLLER_PERIOD 350 // Default control loop period [us].

volatile uint32_t  hapt_timestamp; // Time base of the controller, also used to timestamp
the samples sent by streaming [us].
volatile float32_t hapt_hallVoltage; // Hall sensor output voltage [V].
volatile float32_t hapt_encoderPaddleAngle; // Paddle angle measured by the incremental
encoder [deg].
volatile float32_t hapt_paddleAngle; // Paddle angle measured by the calibrated Hall sensor
[deg].
volatile float32_t hapt_paddleSpeed; // Paddle speed measured by the calibrated Hall sensor
[deg/s].
volatile float32_t hapt_paddleAccel; // Paddle acceleration measured by the calibrated Hall
sensor [deg/s^2].
volatile float32_t hapt_paddlePrevPos; // Previous paddle angle [deg].
volatile float32_t hapt_paddlePrevSpeed; // Previous paddle speed [deg/s].
volatile float32_t hapt_paddleSpeedFilt; // Filtered paddle speed [deg/s].
volatile float32_t hapt_paddleAccelFilt; // Filtered paddle acceleration [deg/s].
volatile float32_t hapt_motorTorque; // Motor torque [N.m].

const float32_t filterCutoffFreqSpeed = 100.0f; // Cutoff frequency for the
speed/acceleration filter [Hz].
const float32_t filterCutoffFreqAccel = 10.0f; // Cutoff frequency for the
speed/acceleration filter [Hz].

void hapt_Update(void);
float32_t hapt_LowPassFilter(float32_t previousFilteredValue, float32_t input,
                             float32_t dt, float32_t tau);

/**
  * @brief Initializes the haptic controller.
  */
void hapt_Init(void)
{
```

```
    hapt_timestamp = 0;
    hapt_motorTorque = 0.0f;
    hapt_paddlePrevPos = 0.0f;
    hapt_paddlePrevSpeed = 0.0f;
    hapt_paddleSpeedFilt = 0.0f;
    hapt_paddleAccelFilt = 0.0f;

    // Make the timers call the update function periodically.
    cbt_SetHapticControllerTimer(hapt_Update, DEFAULT_HAPTIC_CONTROLLER_PERIOD);

    // Share some variables with the computer.
    comm_monitorUint32Func("timestep [us]", cbt_GetHapticControllerPeriod,
                           cbt_SetHapticControllerPeriod);
    comm_monitorFloat("motor_torque [N.m]", (float32_t*)&hapt_motorTorque, READWRITE);
    comm_monitorFloat("encoder_paddle_pos [deg]", (float32_t*)&hapt_encoderPaddleAngle,
READONLY);
    comm_monitorFloat("paddle_pos [deg]", (float32_t*)&hapt_paddleAngle, READONLY);
    comm_monitorFloat("paddle_speed [deg/s]", (float32_t*)&hapt_paddleSpeed, READONLY);
    comm_monitorFloat("paddle_accel [deg/s^2]", (float32_t*)&hapt_paddleAccel, READONLY);
    comm_monitorFloat("paddle_speed_filt [deg/s]", (float32_t*)&hapt_paddleSpeedFilt,
READONLY);
    comm_monitorFloat("paddle_accel_filt [deg/s^2]", (float32_t*)&hapt_paddleAccelFilt,
READONLY);
    comm_monitorFloat("hall_voltage [V]", (float32_t*)&hapt_hallVoltage, READONLY);
}

/**
  * @brief Updates the haptic controller state.
  */
void hapt_Update()
{
    float32_t motorShaftAngle; // [deg].

    // Compute the dt (uncomment if you need it).
    float32_t dt = ((float32_t)cbt_GetHapticControllerPeriod()) / 1000000.0f; // [s].

    // Increment the timestamp.
    hapt_timestamp += cbt_GetHapticControllerPeriod();

    // Get the Hall sensor voltage.
    hapt_hallVoltage = hall_GetVoltage();

    // Compute the paddle angle from the Hall sensor voltage.
    hapt_paddleAngle = hapt_hallVoltage * (-33.0037f) + 81.9710f;
    //hapt_paddleAngle = enc_GetPosition() / REDUCTION_RATIO;

    // Compute the speed and accleration from the Hall paddle angle.
    hapt_paddleSpeed = (hapt_paddleAngle - hapt_paddlePrevPos) / dt;
    hapt_paddleAccel = (hapt_paddleSpeed - hapt_paddlePrevSpeed) / dt;

    hapt_paddlePrevPos = hapt_paddleAngle;
    hapt_paddlePrevSpeed = hapt_paddleSpeed;

    // Filter the speed and acceleration.
    hapt_paddleSpeedFilt = hapt_LowPassFilter(hapt_paddleSpeedFilt,
                                              hapt_paddleSpeed, dt,
                                              filterCutoffFreqSpeed);
    hapt_paddleAccelFilt = hapt_LowPassFilter(hapt_paddleAccelFilt,
                                              hapt_paddleAccel, dt,
                                              filterCutoffFreqAccel);

    // Get the encoder position.
    motorShaftAngle = enc_GetPosition();
    hapt_encoderPaddleAngle = motorShaftAngle / REDUCTION_RATIO;
```

```
    // Compute the motor torque, and apply it.
    //hapt_motorTorque = 0.0f;
    torq_SetTorque(hapt_motorTorque);
}

/**
 * @brief Filters a signal with a first-order low-pass filter.
 * @param previousFilteredValue the previous filtered value.
 * @param input the filter input (the current sample of the signal to filter).
 * @param dt the time elapsed since the last call of this function [s].
 * @param cutoffFreq the cutoff frequency of the filter [Hz].
 * @return the new output of the filter.
 */
float32_t hapt_LowPassFilter(float32_t previousFilteredValue, float32_t input,
                             float32_t dt, float32_t cutoffFreq)
{
    float32_t tau = 1.0f / (2.0f * PI * cutoffFreq); // Rise time (time to reach 63% of the
steady-state value).
    float32_t alpha = dt / (tau+dt); // Smoothing factor.
    return alpha * input + (1.0f - alpha) * previousFilteredValue;
}
```