Haptic Human-Robot interfaces: Lab 0 - answers

2 First steps with the board

2.3 Interaction with the board

2.3.1 Basic remote control with the GUI

When setting the motor torque to a value of 3 mN.m, the paddle moves until it reaches the end stop. When moving the paddle, it does not really feel like a spring, because the force is constant, and is then not proportional to the angle. Examples of (almost) constant forces are:

- The gravity acting on a constant mass.
- A constant-force spring.
- A pre-loaded spring with a very low stiffness, so that over the paddle movement range, it does not deform enough to change its force noticeably.

2.4 Exercises

2.4.1 Spring effect

1. See 3.1 for the full implementation.
2. When moving the paddle with a high spring stiffness (0.02 N.m/deg), we obtain:

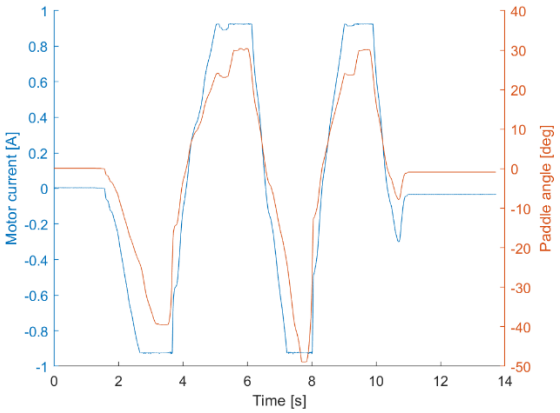

Figure 1: plot of the current and the paddle position

We can notice that the motor current saturates around 0.92 A, which is when the motor torque reaches its nominal value, even if the angle of the paddle keeps increasing. This is because the board automatically limits the current to prevent the motor from overheating.

3. We set the spring stiffness to a lower value, such that we make sure that the motor current never saturates. Then, we pull and release the paddle, and record the oscillation.

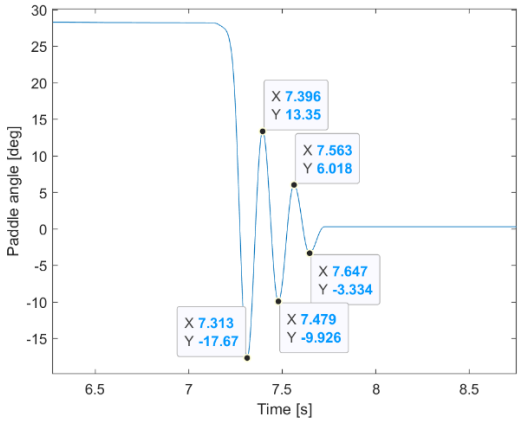

Figure 2: paddle oscillation after release

From the plot, we can read the oscillation period: 0.175 s, or 5.71 Hz.
The frequency of an ideal mass-spring oscillator is:

$$f = \frac{1}{2\pi}\sqrt{\frac{k}{I}}$$

With:

- f = 5.71 Hz: the oscillation frequency
- k = 0.01 N.m/deg = 0.573 N.m/rad: the spring stiffness
- I [kg.m^2]: the system inertia

We can then compute the system inertia (paddle part + rotor + worm screw inertia):

$$I = \frac{k}{(2\pi f)^2} = 4.45 \cdot 10^{-4}\ \text{kg.m}^2$$

We now have to compute the theoretical paddle inertia, considering the dominant inertias: the paddle part itself, the rotor of the motor and the worm screw. From the hardware documentation:

- Paddle part inertia: 2.054e-4 kg.m^2.
- Rotor inertia: 11 g.cm^2 = 1.1e-6 kg.m^2.
- Worm screw inertia: 1.44 g.cm^2 = 0.144e-6 kg.m^2.
- Reduction ratio: 15.

The total equivalent inertia around the rotation axis of the paddle is:

$$I_{eq} = I_{paddle} + R^2(I_{rotor} + I_{worm\,screw}) = 4.85 \cdot 10^{-4}\ \text{kg.m}^2$$

This theoretical value is very close to the one computed from the measurements. (What do you think is the major source of error in this calculation?)

2.4.2 Damping effect

1. When moving the paddle slowly, we notice that the resolution of the speed estimate is very poor (~17°/s).

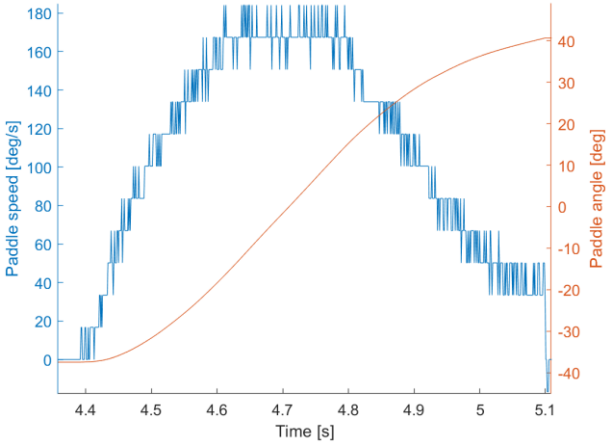

Figure 3: speed estimate at dt = 350 us.

From the hardware documentation, the encoder has a resolution of 4096 steps per motor turn, so 4096 x 15 = 61440 steps per paddle turn, or 170.67 steps/°.
The speed resolution corresponds to the smallest value that can be measured. In this case, this would correspond to 1 encoder step per sampling period (350 us), or 1/170.67° / 350e-6s = 16.74°/s.

2. When increasing the sampling period of the haptic controller to 10 ms, the resolution of the speed estimate is dramatically improved. This time, the resolution increases to 1/170.67° / 10e-3s = 0.59°/s.

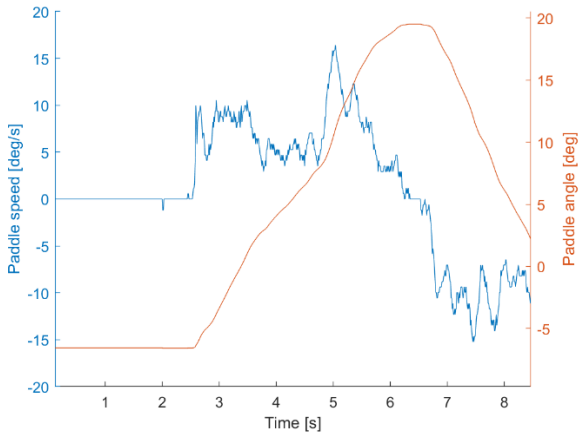

Figure 4: speed estimate at dt = 10000 us.

3. See 3.2 for the full implementation.
   When the paddle moves into the center area, it feels like it is moving into a viscous liquid. It requires very little force to go through it at low speed, but more force is needed to move faster.
   Note that if the damping factor is too high, the motor current may saturate, and the controller could even be unstable.

3 Appendix: source codes

3.1 Spring effect

```
#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_motorTorque; // Motor torque [N.m].
volatile float32_t hapt_springStiffness; // Spring stiffness [N.m/deg].
volatile float32_t hapt_springRestAngle; // Spring rest angle [deg].

void hapt_Update(void);

/**
  * @brief Initializes the haptic controller.
  */
void hapt_Init(void)
{
    hapt_timestamp = 0;
    hapt_motorTorque = 0.0f;
    hapt_springStiffness = 0.0f;
    hapt_springRestAngle = 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("hall_voltage [V]", (float32_t*)&hapt_hallVoltage, READONLY);
    comm_monitorFloat("spring_stiffness [N.m/deg]", (float32_t*)&hapt_springStiffness, READWRITE);
    comm_monitorFloat("spring_rest_angle [deg]", (float32_t*)&hapt_springRestAngle, READWRITE);
}

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

    // 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();

    // Get the encoder position.
```

```
    motorShaftAngle = enc_GetPosition();
    hapt_encoderPaddleAngle = motorShaftAngle / REDUCTION_RATIO;

    // Compute the spring torque.
    paddleTorque = -(hapt_encoderPaddleAngle - hapt_springRestAngle) * hapt_springStiffness;

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

3.2 Damping effect

```
#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 10000 // 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_motorTorque; // Motor torque [N.m].

volatile float32_t hapt_damping; // Spring stiffness [N.m/(deg/s)].
volatile float32_t hapt_prevPaddleAngle; // Previous paddle angle [deg].
volatile float32_t hapt_paddleSpeed; // Paddle speed [deg/s].

void hapt_Update(void);

/**
  * @brief Initializes the haptic controller.
  */
void hapt_Init(void)
{
    hapt_timestamp = 0;
    hapt_motorTorque = 0.0f;
    hapt_damping = 0.0f;
    hapt_prevPaddleAngle = 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("hall_voltage [V]", (float32_t*)&hapt_hallVoltage, READONLY);
    comm_monitorFloat("damping [N.m/(deg/s)]", (float32_t*)&hapt_damping, READWRITE);
    comm_monitorFloat("paddle_speed [deg/s]", (float32_t*)&hapt_paddleSpeed, READONLY);
}

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

    // 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();

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

    // Compute the paddle speed.
    hapt_paddleSpeed = (hapt_encoderPaddleAngle - hapt_prevPaddleAngle) / dt;
    hapt_prevPaddleAngle = hapt_encoderPaddleAngle;

    // Compute the paddle damping torque.
    if(fabsf(hapt_encoderPaddleAngle) < 10.0f)
        paddleTorque = -hapt_paddleSpeed * hapt_damping;
    else
        paddleTorque = 0.0f;

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