Real-Time Systems

Laboratory Exercise 3 - Distance Education Version

Embedded Control of a Rotating DC Servo

Department of Automatic Control LTH
Lund University
March 2020

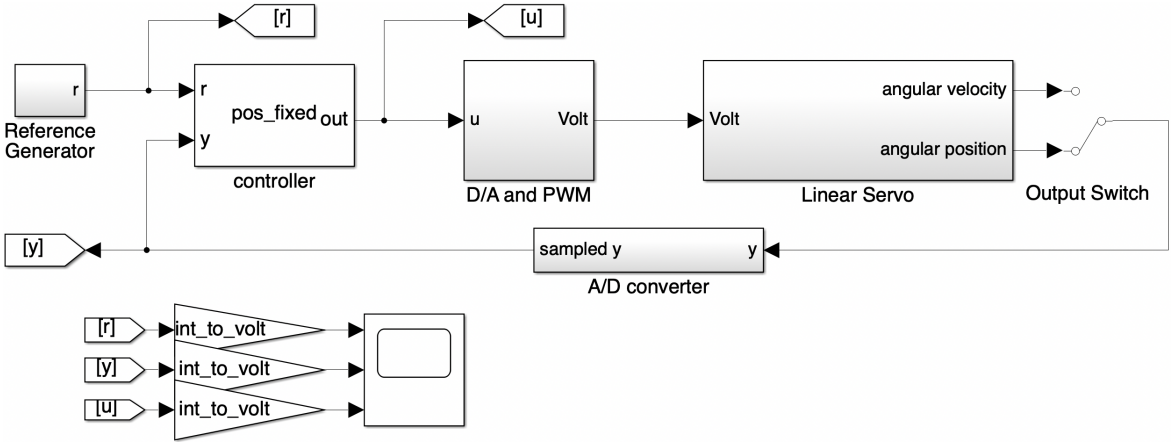

Figure 1: Simulink model for control of a rotating DC servo.

Preparations

Make sure you have reviewed Lecture 11 on computer arithmetic before starting this lab. Also reviewing Problem Solving Exercise 3 (on design of state feedback and observers in MATLAB) can be useful.

1 Introduction

The purpose of this "virtual" laboratory exercise is to develop an embedded controller for a rotating DC servo. By embedded we mean that the controller is implemented in the low-level language C, and we account for the A/D and D/A conversion of signals. In the physical version of the lab you would have used an ATMEL AVR microcontroller to run the C code and control the DC servo motor. Here you will develop the same code for the controller but test it within the Matlab simulation environment Simulink. The

`.slx` file contains the Simulink model: In Matlab, navigate to the folder that contains the model and open it. This model uses the *CCaller* block to execute the C code that you will write. This block is available only since Matlab 2019, therefore the Matlab version 2019 is needed to run it[1]. The mentioned Simulink model is shown in Figure 1.

You will develop one controller for the angular velocity of the rotating servo and one controller for the angular position. A PI controller will be used for velocity control. The position instead will be controlled using state feedback, and therefore a state-observer is required as well. Also, you will use the estimation of a constant load disturbance to obtain integral action. Both the velocity and position controllers will be implemented using floating-point arithmetic at first, and then using only fixed-point arithmetic.

1.1 The ATMEL AVR Microcontroller

Programs for the AVR are usually written in C and sometimes in assembler. The AVR, like most microcontrollers, has no built-in support for floating-point arithmetic. If the C program uses `floats`, then all operations on them are emulated using calls to a floating-point library. Since the floating-point library is very slow and consumes a lot of program memory, it is often much more efficient to use fixed-point arithmetic. It can also be noted that the AVR has no hardware instruction for (integer) division.

Memory words in the AVR have the size of 16 bits. Therefore memory read and write are optimized for variables of this size. As a consequence, to write code that is most efficient, it is better to use only `int16_t` variables. On the other side, `int32_t` variables can still be used for intermediate results that are not stored in any variable. In fact intermediate results will only be stored in the CPU registers which have 32 bits.

Analog input is performed using the built-in A/D converter, while analog output is emulated using **pulse-width modulation** (PWM). This technique is used when the voltage source cannot be controlled to obtain an arbitrary number of volts; it can only be turned on (giving maximum voltage) or turned off (giving zero voltage). With PWM, the desired analog signal is generated through an high-frequency square wave, i.e., a series of pulses at the maximum voltage available. Those pulses are then filtered using an analog filter to obtain the desired signal. Wider pulses will generate a higher signal, while narrower ones will generate a lower signal. In this context, the **duty cycle** of the square wave (i.e., the percentage of the period that the signal is high) is used as the control variable. In this way we will be able to generate the desired voltage to be applied to the DC motor. Timers are set in the AVR microcontroller such that the controller is executed every 50 ms, and the square wave of the PWM has a period of 10 ms.

NOTE: You will not be asked to implement the PWM generation. This explanation is just for you to understand the implementation of control systems that rely on the generation of analog voltages.

Both the analog inputs and outputs of the AVR controller take values in the range $[-10\text{ V}, +10\text{ V}]$, with $n = 10$ bits of precision. This means that the control output y

[1] You can download Matlab from `http://program.ddg.lth.se/en/` . You need to include Simulink and Control System Toolbox in the installation.

and the control input u are `int16_t` variables that can take values only between -512 (corresponding to -10 V) and $+511$ (corresponding to 10 V)[2]. Thus, values outside of this range will cause overflow.

2 The Simulink Model

In this section we describe the Simulink model and how to use it. The Simulink model relies on some parameters that are defined in the Matlab script `initialize.m`; this script is run automatically by Simulink before every simulation. The Simulink model is executed by clicking on the green play button on top of the window. If you double click on the Scope block in the bottom part of the model then a plot of the reference r, the control output y and the control input u is shown. All the signals are plotted in volts. The reference is a square wave with a period of 10 s and an amplitude of 5 V. You can also click on the legend of the different signals on the top right of the plot to hide them. You do not need to change the simulink model for completing the lab but, of course, you are encourage to do so to try things and verify your understanding.

We will now go into the details of each of the blocks.

2.1 The Controller

The controller block is the one representing your controller, and it is the most important one in this lab. If you double click on it, a window opens, and you can select which controller function you want to use. Each controller is implemented as one C function. Once you have selected the desired function, its name will appear on the block (for example, in Figure 1 the `pos_fixed` function is selected).

In the folder `src` you will find four `.c` files. Each of those contains the definition of a C function for one of the four different controllers that you have to implement. **These are the files you have to edit to do this lab**.

The control action is written to the D/A converter with the `return` statement. This statement also ends the function call. This means that the code that is written after the return statement will not be executed. As a consequence, in this environment, the processing time of the controller cannot be minimized by updating the state after having actuated the new control action[3]. The state needs to be updated before the `return`. This is an artifact of this simiulaiton environment. In the physical version of the lab (and on real controllers) you would use another function (e.g., `write_ctrl_action()`) to write the new control signal to the D/A converter.

2.1.1 Important Remarks on Writing C Code

- In C programs, constants are usually defined as **macros**. Macros are usually defined at the beginning of a C file and are commands to the compiler, they are not part of the program that will be executed by the target. They are very

[2] Review the two's complement representation if this asymmetry is not clear.

[3] On the other side, in these simulations, the execution of the code is considered "instant", and its duration does not really affect the system's performance.

useful to write code that is readable, portable and efficient. In practice, define the constants at the beginning of the file with the syntax `#define K 99.99` to define $K = 99.99$ in the whole file. In this way, the compiler, even before starting to compile the program, will swap all the `K` symbols with `99.99`. A common bug is to write define statements with an equal sign and/or a semicolon at the end. E.g., "`#define K = 1.234;`" is wrong and should be "`#define K 1.234`". These type of mistakes might give very strange compilation errors.

- Some variables you will want to be the same in the different executions of the controller, namely the *states*. To prevent that the variables are re-allocated every time (and therefore erased) you can use the keyword `static` (e.g. `static float I;`). This keyword allows for static allocation of the given variable across the different instances of the C function. An alternative could be to use global variables (i.e. variables defined outside of the function). But using the gloal scope should be limited only to those variables that need to be read by different functions, this is apparently not the case for our controller functions.

- In the fixed point controller implementations you are not allowed to store variables that are not `int16_t`, but intermediate results can be `int32_t`. To perform an `int32_t` operation between two `int16_t` variables, it is sufficient to cast one operand beforehand, the casting of the other operand will be automatically included by the compiler. For example (also assuming that `K`, `r` and `y` have 3 fractional bits):

```
int16_t u = ( (int32_t)K * (r - y) ) >> 3;
```

- Multiplication and division by two and its powers are easily implemented with the bit-shift operator. E.g., to implement $a = a * 2^{-3}$ you can write `a = a >> 3`.

- Be careful with the operator precedence – especially between additions and shifts. For example "`u = (temp >> 13) + I;`" is correct, "`u = temp >> 13 + I;`" is not.

- **When you write C code, you write assignments and not equations.** Therefore the order in which you write the lines of code is very important and has to be carefully taken into account.

2.2 D/A (PWM)

This block implements the PWM modulation of generating the analog voltage signal. You can see how it is implemented by double clicking on it. The square wave generation is implemented with the common method of comparing the signal to a triangular wave. **Tip:** open the scope on the right to see how the low pass filter "transforms" the square wave into the desired analog signal.

2.3 Linear Servo

This block implements a linear model simulating the dynamics of the DC servo. By double clicking on the output switch on its right you can choose which signal is used as output. Obviously you should use the angular velocity for the velocity control and the angular position for the position control.

2.4 A/D Converter

This block implements the analog to digital conversion of the output signal. **Tip:** If you double click on it and open the scope inside it you can see three different signals: the original output signal, the quantized signal, and the quantized and sampled signal. You will need to zoom into the signal to distinguish between the three of them.

3 Laboratory Assessment

You can check if the controller parameters and their fixed point representation are correct using the following form `https://docs.google.com/forms/d/e/1FAIpQLScHDXuzuYHGhQUix9Z7QNG51m33iPFXbZ9FKVzj1zxOw-0GXQ/viewform?usp=sf_link` . Insert your answers and submit your form, you will be able to see which are correct and which are not. **Note:** we will not check your answers to this form, it is just for you to make sure you have the correct control parameters.

The lab is individual, and each student has to submit his/her own code. Submit the four `.c` files from the `src` folder in the canvas page. Clearly comment **EVERY SINGLE** line of code! It is not sufficient to implement the correct solution—you also have to be able to explain how you came up with that solution. In the comments you have to make clear why you made the different choices (e.g., why do you define a variable with a certain type, why you do a casting, why you do a bit-shift of that specific number of bits). Do not be afraid of stating the obvious.

4 Control of the Angular Velocity

4.1 Control Design

The transfer function from the control input u to the measured angular velocity y is given by

$$P(s) = \frac{2.25}{s+0.12}$$

To control the velocity process, we will use a simple PI controller,

$$C(s) = K\left(1 + \frac{1}{sT_i}\right)$$

4.1.1 Assignment 1

Determine K and T_i so that the (continuous-time) closed-loop poles are placed in $-3 \pm 2i$. ◇

4.2 Implementation Structure

The PI controller shall be implemented in discrete time as

$$u(k) = K\beta r(k) - Ky(k) + I(k)$$
$$I(k+1) = I(k) + \tfrac{Kh}{T_i}\big(r(k) - y(k)\big)$$

with reference weighting $\beta = 0.5$ and the sampling interval $h = 0.05$.

4.3 Floating-Point Implementation

4.3.1 Assignment 2.

Implement the velocity controller in C using floating-point constants and `float` variables.

- The measurement and the reference signal are the inputs of the C function `int16_t`. Those are integer values in the range $[-512, +511]$.
- The output shall be in the range $[-512, +511]$. Otherwise overflow might happen.

When implementing the floating-point control algorithm, you can simply ignore the fact that the input and output can only assume integer values. When adding or multiplying an `int` and a `float` in C, the `int` is first converted to a `float`.

Simulate the model to test your controller.

Note the performance level (e.g. amplitude of ripple on the signals) so you can compare it with the fixed-point implementation later. ◇

Tip: once you have a working controller you should look at the scopes inside the D/A and A/D convertes to see how they convert the signals.

4.4 Fixed-Point Implementation

4.4.1 Assignment 3

For the fixed-point implementation, we will use 16-bit wordlength ($N = 16$). Study the controller coefficients and select a suitable number of fractional bits n. For simplicity, we will use the same number of fractional bits for all coefficients. Convert the controller coefficients to their fixed-point representation and verify it on the google form.

Choose the number of fractional bits to maximize their number and improve precision. At the same time, make sure that no integers bits are missed during the computations: i.e. that overflow doesn't happen. *Remember that you are allowed to use 32-bit integers for the intermediate results.*

Instead of converting β, convert $K\beta$ to fixed-point. Why? Which are the other coefficients that should be converted? ◇

4.4.2 Assignment 4

Make a fixed-point implementation of the velocity controller in C using only integer constants and `int16_t` and `int32_t` variables (the latter only for the intermediate results).

- Remember that the input, output, and reference variables are integers, i.e., they have zero fractional bits.
- For simplicity, use zero fractional bits also for the controller state I. This means that addition and subtraction can be done without scaling.

Simulate the model to test your controller, remember to change the selected C function. If everything is done correctly, you should achieve very similar performance to the floating point implementation. ◇

5 Control of the Angular Position

5.1 Control Design

For control of the angular position, we will work in the state-space domain. A state-space model of the position process is given by

$$\frac{dx(t)}{dt} = \underbrace{\begin{bmatrix} -0.12 & 0 \\ 5 & 0 \end{bmatrix}}_{A} x(t) + \underbrace{\begin{bmatrix} 2.25 \\ 0 \end{bmatrix}}_{B} u(t),$$
$$y(t) = \underbrace{\begin{bmatrix} 0 & 1 \end{bmatrix}}_{C} x(t),$$

where u is the control input, x_1 is the angular velocity, x_2 is the angular position, and y is the measured position. The variables in the model are in the scale of 10 bits integers, so you can directly implement the control parameters that you obtain for this model to the real implementation. To make the control problem more interesting, we assume that the velocity is not directly measurable on the process.

5.1.1 Assignment 5

Using MATLAB, sample the process description with the interval $h = 0.05$ and then design a state-feedback controller

$$u(k) = k_r r(k) - Kx(k), \quad K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}$$

so that the (discrete-time) closed-loop poles are placed in $0.8 \pm 0.1i$ and so that the static gain from r to y is 1. $\diamond$

To obtain a controller with integral action, we assume that there is a constant disturbance v acting on the input of the process. The augmented sampled system then becomes

$$\begin{bmatrix} x(k+1) \\ v(k+1) \end{bmatrix} = \underbrace{\begin{bmatrix} \Phi & \Gamma \\ 0 & 1 \end{bmatrix}}_{\Phi_e} \underbrace{\begin{bmatrix} x(k) \\ v(k) \end{bmatrix}}_{x_e(k)} + \underbrace{\begin{bmatrix} \Gamma \\ 0 \end{bmatrix}}_{\Gamma_e} u(k)$$

$$y(k) = \underbrace{\begin{bmatrix} C & 0 \end{bmatrix}}_{C_e} \begin{bmatrix} x(k) \\ v(k) \end{bmatrix}$$

5.1.2 Assignment 6

Using MATLAB, design an observer

$$\hat{x}_e(k+1) = \Phi_e \hat{x}_e(k) + \Gamma_e u(k) + L_e\big(y(k) - C_e \hat{x}_e(k)\big)$$

for the augmented system so that the (discrete-time) observer poles are placed in $0.6 \pm 0.2i$ and 0.55. $\diamond$

5.2 Implementation Structure

Using the information from the observer, we have the augmented control law

$$u(k) = k_r r(k) - K\hat{x}(k) - \hat{v}(k)$$

Now let

$$\Phi = \begin{bmatrix} \phi_{11} & \phi_{12} \\ \phi_{21} & \phi_{22} \end{bmatrix}, \quad \Gamma = \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix}, \quad K = \begin{bmatrix} k_1 & k_2 \end{bmatrix}, \quad L_e = \begin{bmatrix} l_1 \\ l_2 \\ l_v \end{bmatrix}$$

The complete controller to be implemented can then be written as

$$\begin{aligned} u(k) &= k_r r(k) - k_1 \hat{x}_1(k) - k_2 \hat{x}_2(k) - \hat{v}(k) \\ \varepsilon(k) &= y(k) - \hat{x}_2(k) \\ \hat{x}_1(k+1) &= \phi_{11}\hat{x}_1(k) + \phi_{12}\hat{x}_2(k) + \gamma_1\big(u(k) + \hat{v}(k)\big) + l_1\varepsilon(k) \\ \hat{x}_2(k+1) &= \phi_{21}\hat{x}_1(k) + \phi_{22}\hat{x}_2(k) + \gamma_2\big(u(k) + \hat{v}(k)\big) + l_2\varepsilon(k) \\ \hat{v}(k+1) &= \hat{v}(k) + l_v\varepsilon(k) \end{aligned}$$

5.3 Floating-Point Implementation

5.3.1 Assignment 7

Implement the position controller in C using floating-point constants and `float` variables. Remember to switch the output switch to the angular position and test your controller. ◇

5.4 Fixed-Point Implementation

5.4.1 Assignment 8

For the fixed-point implementation, we will use 16-bit wordlength ($N = 16$). Study the controller coefficients and select a suitable number of fractional bits n (the same for all coefficients). Convert the controller coefficients to fixed-point representation. ◇

5.4.2 Assignment 9

Make a fixed-point implementation of the position controller in C using integer constants and `int16_t` and `int32_t` variables (the latter only for the intermediate results). Again, for simplicity, assume zero fractional bits for all the controller states.

Execute the model and test your controller. Can you notice any performance difference from the floating-point case? ◇