Real-Time Systems

Laboratory Exercise 3

Embedded Control of a Rotating DC Servo

Department of Automatic Control LTH
Lund University
November 2010

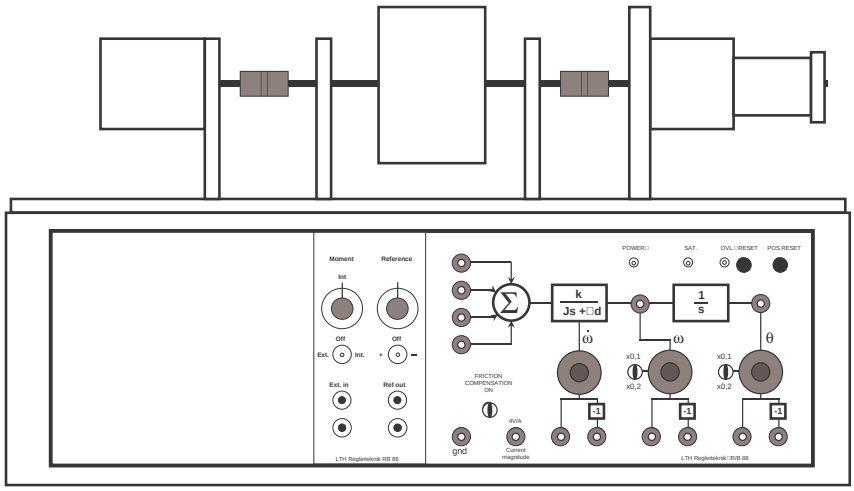

Figure 1 The rotating DC servo.

Preparations

- Review Lecture 11 (on computer arithmetic).
- Review Problem Solving Exercise 3 (on design of state feedback and observers in MATLAB).
- Study this manual carefully, including the code skeleton in Appendix B. Make sure that you understand the structure of the program.
- Solve the Preparatory Assignments 1, 3, 5, 6 and 8.

1. Introduction

The purpose of the laboratory exercise is to control the rotating DC servo in Figure 1 using an AVR microcontroller. For velocity control, a simple PI controller will be used. The position will be controlled using state feedback from an observer, with estimation of a constant input disturbance to obtain integral action.

The velocity and position controllers will be implemented using both floating-point and fixed-point arithmetic. The control programs are edited and compiled on the PC and then uploaded to the AVR. User commands to the AVR are sent via a terminal window connected to the serial port. The control and measurement signals can be plotted in a simple Java GUI on the PC.

1.1 The ATMEL AVR Microcontroller

During the lab you will use either an ATmega8 or an ATmega16 AVR.

The ATmega16 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. It provides, among other things, 16 Kbyte of programmable flash program memory, 512 byte EEPROM, 1 Kbyte SRAM, 32 general purpose I/O lines, three flexible timer/counters, four PWM channels, a serial programmable USART, and a 10-bit A/D converter.

The ATmega8 is very similar to the ATmega16, but has a little less functionality. For example it only has 8 Kbyte flash, 23 general purpose I/O lines and three PWM channels.

In our laboratory setup, the AVR is clocked at 14.7456 MHz and provides up to 14.7456 MIPS throughput, since most instructions are single cycle. 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.

1.2 Implementing Feedback Controllers on the AVR

The simplest way to implement a periodic controller in the AVR is to set up a periodic timer interrupt and then execute the control algorithm in the interrupt handler. The ATmega8/ATmega16 provides a 16-bit high-resolution timer that can be used for this purpose. Every time the timer expires, the handler `ISR(TIMER2_COMP_vect)` will be called.

Analog input is performed using the built-in A/D converter, while analog output can be emulated using the built-in pulse-width modulation (PWM) generator. User communication over an RS-232 serial connection can be performed via the USART interface. Characters can be written to the serial connection using the `putchar` standard IO function.

For the controller implementation, a code skeleton is given in Appendix B. Go through the code and make sure you understand the structure of the program. In the `main()` function, the following initialization is performed:

- The directions of the I/O ports are set, and the A/D converter is enabled.
- Timer 1 is programmed to produce a PWM (pulse-width modulation) output. The duty cycle of the PWM is used as the control variable.
- Timer 2 is programmed to give an interrupt every 10 ms, invoking the handler `ISR(TIMER2_COMP_vect)`. Every fifth time (i.e., every 50 ms), the handler runs the controller.
- The serial communication interface is configured to invoke the handler `ISR(USART_RXC_vect)` every time a character is received.
- The interrupts are enabled and the timers are started.

Tip! You will write four different programs based on the code skeleton `DCservo.c` in Appendix B (available on the home page). To manage the different versions, make four copies of `DCservo.c` named `velfloat.c`, `velfixed.c`, `posfloat.c` and `posfixed.c`. Remember to change the compile commands accordingly.

2. Control of the Angular Velocity

2.1 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,

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

Assignment 1 (Preparatory Assignment). Determine K and T_i so that the (continuous-time) closed-loop poles are placed in $-3 \pm 2i$. ◇

2.2 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 $h = 0.05$.

2.3 Floating-Point Implementation

Assignment 2. Implement the velocity controller in C using floating-point constants and `float` variables. Use the code skeleton in Appendix B (available on the home page).

- The input is read using the function `int16_t readInput(char chan)`. This gives a value in the range $[-512, +511]$. The velocity measurement is read from input channel 0.
- The output is written using the function `void writeOutput(int16_t val)`. The value should be in the range $[-512, +511]$.
- The reference value is available in the global variable `int16_t r` and is compatible with the measured input.
- In C, constants are conveniently declared using the `#define` directive, e.g. `#define K 2.6133`

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`.

Compile and upload the controller to the AVR and test it against the DC servo. Instructions for compiling and running are given in Appendix A. What is the size of the uploaded code? (Look at the size of the `.sr` file)

Note the performance level (e.g. amplitude of ripple on the signals) so you can compare it with the fixed-point implementation later. See how the controller handles load disturbances by e.g. touching the rotating mass or applying a torque with the moment knob (see Appendix A.5): ◇

Tip! Open simcom and Opcom in separate terminal windows. Then you can keep them running during the entire lab and don't have to restart them for each experiment.

2.4 Fixed-Point Implementation

Assignment 3 (Preparatory Assignment). 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 fixed-point representation. Instead of converting β, convert $K\beta$ to fixed-point. Why? Which are the other coefficients that should be converted? ◇

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 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.

Compile and upload the controller to the AVR and check that it works. Can you notice any performance difference from the floating-point case? What is the size of the uploaded code now? ◇

3. Control of the Angular Position

3.1 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

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

$$y(t) = \underbrace{\begin{pmatrix} 0 & 1 \end{pmatrix}}_{C} x(t)$$

where u is the control input, x_1 is the angular velocity, x_2 is the angular position, and y is the measured position. (To make the control problem more interesting, we assume that the velocity is not directly measurable on the process.)

Assignment 5 (Preparatory Assignment). Using MATLAB, sample the process description with the interval $h = 0.05$ and then design a state-feedback controller

$$u(k) = l_r r(k) - Lx(k)$$

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. ◇

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

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

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

Assignment 6. (Preparatory Assignment) Using MATLAB, design an observer

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

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

3.2 Implementation Structure

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

$$u(k) = l_r r(k) - L\hat{x}(k) - \hat{v}(k)$$

Now let

$$\Phi = \begin{pmatrix} \phi_{11} & \phi_{12} \\ \phi_{21} & \phi_{22} \end{pmatrix}, \quad \Gamma = \begin{pmatrix} \gamma_1 \\ \gamma_2 \end{pmatrix}, \quad L = \begin{pmatrix} l_1 & l_2 \end{pmatrix}, \quad K_e = \begin{pmatrix} k_1 \\ k_2 \\ k_v \end{pmatrix}$$

The complete controller to be implemented can then be written as

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

3.3 Floating-Point Implementation

Assignment 7. Implement the position controller in C using floating-point constants and `float` variables. The position is measured on input channel 1. Upload the controller to the AVR and test it against the DC servo. What is the size of the uploaded code? ◇

3.4 Fixed-Point Implementation

Assignment 8. (Preparatory Assignment) 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. ◇

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 for the intermediate results). Again, for simplicity, assume zero fractional bits for all the controller states.

Upload the controller to the AVR and check that it works. Can you notice any performance difference from the floating-point case? What is the size of the uploaded code now? ◇

A. Compiling and Running Against the Process

A.1 Connecting the AVR

In the lab a box containing either an ATmega16 or an ATmega8 will be used.

Connect the ATmega16 AVR to the PC using the programming cable, the serial cable, and the USB cable. Note that the USB connection is only used as a power supply.

The ATmega8 is connected the same way, except for that the power is supplied by a 12 V power adaptor.

Connect the process in the following way:

DC Servo	ATmega8	ATmega16	PC I/O Interface	Cable Color
Angular velocity (ω)	AI2	AI0	AI0	Yellow
Angular position (θ)	AI3	AI1	AI1	Blue
Control input (Σ)	AO0	AO1	AI2	Red
Ground	Ground	Ground	Ground	Black

A.2 Compiling and downloading to the AVR

Assume that the program source code is called `DCservo.c`.

- To compile for the
  - ATmega8 AVR:
    ```
    > avr-gcc -mmcu=atmega8 -O -g -Wall -o DCservo.elf DCservo.c
    ```
  - ATmega16 AVR:
    ```
    > avr-gcc -mmcu=atmega16 -O -g -Wall -o DCservo.elf DCservo.c
    ```
- To upload the executable to the
  - ATmega8 AVR:
    ```
    > avr-objcopy -Osrec DCservo.elf DCservo.sr
    > avrdude -e -p atmega8 -P usb -c avrisp2 -U flash:w:DCservo.sr:a
    ```
  - ATmega16 AVR:
    ```
    > avr-objcopy -Osrec DCservo.elf DCservo.sr
    > avrdude -e -p atmega16 -P usb -c avrisp2 -U flash:w:DCservo.sr:a
    ```
- To view the generated assembler code, you can type
  ```
  > avr-objdump -S DCservo.elf
  ```

A.3 Command interface

Commands can be sent to the AVR by sending 8-bit characters over the serial port.

Open a new terminal window and open a 38400 bps serial connection to the AVR:

```
> simcom -38400 /dev/ttyS0
```

To send commands to the AVR, type one of the following characters in the terminal window:

- s: start the controller
- t: stop the controller
- r: switch sign of the reference

If the connection works properly, the AVR will send back the same character that it received, and this character will be printed in the terminal window.

A.4 Plots

Download `Opcom.zip` from the course homepage and unzip it. Start it in a separate terminal window using

```
> java Opcom
```

This starts a Java GUI that measures the the angular velocity, the angular position and the control signal, and plots them over time.

A.5 DC Servo

A DC servo driving a rotating mass will be used in this lab.

Before you start the experiments, make sure that the Moment switch on the left side of the interface panel is set to Internal and the knob turned to 0. If the knob is turned, a second servo applies a load torque to the rotating mass. Also make sure that the friction compensation switch is in the off position when you start. When you perform your experiments you can introduce load disturbances my modifying the settings of the moment knob and the friction compensation switch.

If the OVL (overload) LED on the upper right part of the interface panel is lit, press the RESET button to eneble the servo again.

A.6 Troubleshooting

When running programs on an embedded system such as the AVR, it might not be obvious how to troubleshoot a malfunctioning program. You can start by asking yourself the following questions:

- Did the build and upload succeed?
- Are the commands written to `simcom` echoed (shown on screen) as they are supposed to?
- What values are shown in the plots in `OpCom`?
- Do the measurements in the plots match the process?

Also, the serial port can be used to send small debug messages to `simcom`.

B. Code Skeleton – DCservo.c

```
/**
 * AVR program for control of the DC-servo process.
 *
 * User communication via the serial line. Commands:
 *   s: start controller
 *   t: stop controller
 *   r: change sign of reference (+/- 5.0 volt)
 *
 * To compile for the ATmega8 AVR:
 *   avr-gcc -mmcu=atmega8 -O -g -Wall -o DCservo.elf DCservo.c
 *
 * To upload to the ATmega8 AVR:
 *   avr-objcopy -Osrec DCservo.elf DCservo.sr
 *   avrdude -e -p atmega8 -P usb -c avrisp2 -U flash:w:DCservo.sr:a
 *
 * To compile for the ATmega16 AVR:
 *   avr-gcc -mmcu=atmega16 -O -g -Wall -o DCservo.elf DCservo.c
 *
 * To upload to the ATmega16 AVR:
 *   avr-objcopy -Osrec DCservo.elf DCservo.sr
 *   avrdude -e -p atmega16 -P usb -c avrisp2 -U flash:w:DCservo.sr:a
 *
 * To view the assembler code:
 *   avr-objdump -S DCservo.elf
 *
 * To open a serial terminal on the PC:
 *   simcom -38400 /dev/ttyS0
 */

#include <avr/io.h>
#include <avr/interrupt.h>

/* Controller parameters and variables (add your own code here) */

int8_t on = 0;                      /* 0=off, 1=on */
int16_t r = 255;                    /* Reference, corresponds to +5.0 V */

/**
 * Write a character on the serial connection
 */
static inline void put_char(char ch){
  while ((UCSRA & 0x20) == 0) {};
  UDR = ch;
}

/**
 * Write 10-bit output using the PWM generator
 */
static inline void writeOutput(int16_t val) {
  val += 512;
  OCR1AH = (uint8_t) (val>>8);
  OCR1AL = (uint8_t) val;
}

/**
```

```
 * Read 10-bit input using the AD converter
 */
static inline int16_t readInput(char chan) {
  uint8_t low, high;
  ADMUX = 0xc0 + chan;             /* Specify channel (0 or 1) */
  ADCSRA |= 0x40;                  /* Start the conversion */
  while (ADCSRA & 0x40);           /* Wait for conversion to finish */
  low = ADCL;                      /* Read input, low byte first! */
  high = ADCH;                     /* Read input, high byte */
  return ((high<<8) | low) - 512;  /* 10 bit ADC value [-512..511] */
}

/**
 * Interrupt handler for receiving characters over serial connection
 */
ISR(USART_RXC_vect){
  switch (UDR) {
  case 's':                        /* Start the controller */
    put_char('s');
    on = 1;
    break;
  case 't':                        /* Stop the controller */
    put_char('t');
    on = 0;
    break;
  case 'r':                        /* Change sign of reference */
    put_char('r');
    r = -r;
    break;
  }
}

/**
 * Interrupt handler for the periodic timer. Interrupts are generated
 * every 10 ms. The control algorithm is executed every 50 ms.
 */
ISR(TIMER2_COMP_vect){
  static int8_t ctr = 0;
  if (++ctr < 5) return;
  ctr = 0;
  if (on) {
    /* Insert your controller code here */

  } else {
    writeOutput(0);     /* Off */
  }
}

/**
 * Main program
 */
int main(){

  DDRB = 0x02;    /* Enable PWM output for ATmega8 */
  DDRD = 0x20;    /* Enable PWM output for ATmega16 */
  DDRC = 0x30;    /* Enable time measurement pins */
  ADCSRA = 0xc7;  /* ADC enable */
```

```
  TCCR1A = 0xf3;  /* Timer 1: OC1A & OC1B 10 bit fast PWM */
  TCCR1B = 0x09;  /* Clock / 1 */

  TCNT2 = 0x00;   /* Timer 2: Reset counter (periodic timer) */
  TCCR2 = 0x0f;   /* Clock / 1024, clear after compare match (CTC) */
  OCR2 = 144;     /* Set the compare value, corresponds to ~100 Hz */

  /* Configure serial communication */
  UCSRA = 0x00;   /* USART: */
  UCSRB = 0x98;   /* USART: RXC enable, Receiver enable, Transmitter enable */
  UCSRC = 0x86;   /* USART: 8bit, no parity */
  UBRRH = 0x00;   /* USART: 38400 @ 14.7456MHz */
  UBRRL = 23;     /* USART: 38400 @ 14.7456MHz */

  TIMSK = 1<<OCIE2; /* Start periodic timer */

  sei();          /* Enable interrupts */

  while (1);
}
```