

5. 근 궤적 법

5.5 실험 실습: Lab 5

5.5.1 아날로그 다이내믹 시뮬레이터 대한 PD 제어기

이 실험에서는 4.7.2절에서 사용한 아날로그 다이내믹 시뮬레이터를 사용한다. 상수 이득값을 가지는 제어기에 미분 항을 추가해서 PD 제어기를 적용해 본다. 그림 5-51은 아날로그 다이내믹 시뮬레이터에 PD 제어기를 적용한 블록 선도이다.

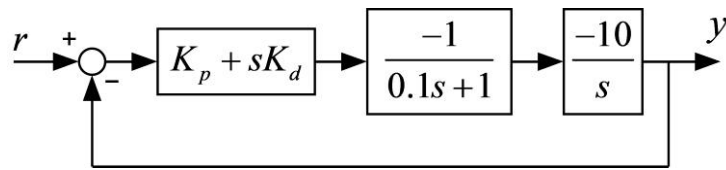


그림 5-51 아날로그 다이내믹 시뮬레이터의 PD 제어

제어 오차는 기준 입력에서 출력을 뺀 값으로 다음 식과 같이 정의한다.

$$e(t) = r(t) - y(t) \quad (5.143)$$

PD 제어기의 제어 신호는 다음과 같다.

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} \quad (5.144)$$

그림 5-51 시스템의 전달 함수는 다음과 같다.

$$\frac{Y(s)}{R(s)} = \frac{10(sK_d + K_p)}{0.1s^2 + s + 10(sK_d + K_p)} \quad (5.145)$$

위의 전달 함수는 2개의 극점과 1개의 영점을 가지고 있다. 전달 함수의 영점은 시스템의 안정도에 영향을 미치지 않지만 기준 입력 신호의 종류에 따라서 시스템 출력의 오버슈트를 크게 증가시킬 수 있다. 만약 기준 입력 신호가 계단 함수라면 계단 함수의 미분 성분이 출력 신호에 포함될 수 있다. 계단 함수의 미분은 크기가 무한대인 임펄스 함수이므로 출력 신호에 오버슈트를 증가시키는 요인으로 작용할 수 있다. 이와 같은 상황을 피하기 위해서 PD 제어기가 적용된 시스템을 그림 5-52와 같이 변형할 수 있다.

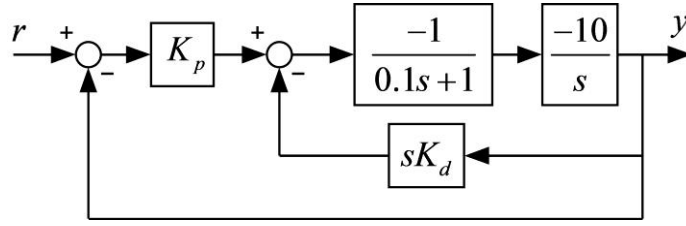


그림 5-52 변형된 PD 제어기가 적용된 시스템

그림 5-52의 PD 제어기의 제어 신호는 다음과 같다.

$$u(t) = K_p e(t) - K_d \frac{dy(t)}{dt} \quad (5.146)$$

그림 5-52의 페루프 시스템의 전달 함수는 다음과 같다.

$$\frac{Y(s)}{R(s)} = \frac{10K_p}{0.1s^2 + s + 10(sK_d + K_p)} = \frac{100K_p}{s^2 + (10 + 100K_d)s + 100K_p} \quad (5.147)$$

식 (5.147)의 전달 함수는 식 (5.145)의 전달 함수와 분모는 같지만 영점이 없다는 것을 알 수 있다. 이 실험에서는 식 (5.146) 형태의 PD 제어기를 구현한다. 페루프 시스템의 감쇠비 (damping ratio)는 다음과 같이 계산할 수 있다.

$$\zeta = \frac{10 + 100K_d}{2\omega_n} = \frac{10 + 100K_d}{2\sqrt{100K_p}} \quad (5.148)$$

시스템의 응답 데이터를 데이터 어레이에 저장하면 컴퓨터로 전송해서 파일에 저장할 수 있다. STM32F429 디스커버리 보드의 파란색 사용자 버튼 스위치를 누르면 인터럽트를 발생해서 데이터 캡처를 시작하도록 할 수 있다.

STM32CubeIDE에서 4장 Lab4의 **4.7.2 아날로그 다이내믹 시뮬레이터의 피드백 제어**와 동일한 방법으로 프로젝트를 만든다. 여기에 추가할 설정은 사용자 버튼의 인터럽트 발생이다. 사용자 버튼은 PA0 핀에 연결되어 있으며 그림 5-53과 같이 외부 입력으로 사용될 수 있도록 설정한다. 그림 5-53은 System Core 항목에서 GPIO 항목을 선택하면 볼 수 있는 화면이며, 이 화면에서 PA0/WKUP 항목을 클릭하면 GPIO mode 선택 창이 아래에 나온다. 이 창에서 External Interrupt Mode with Rising edge trigger detection 항목을 선택한다.

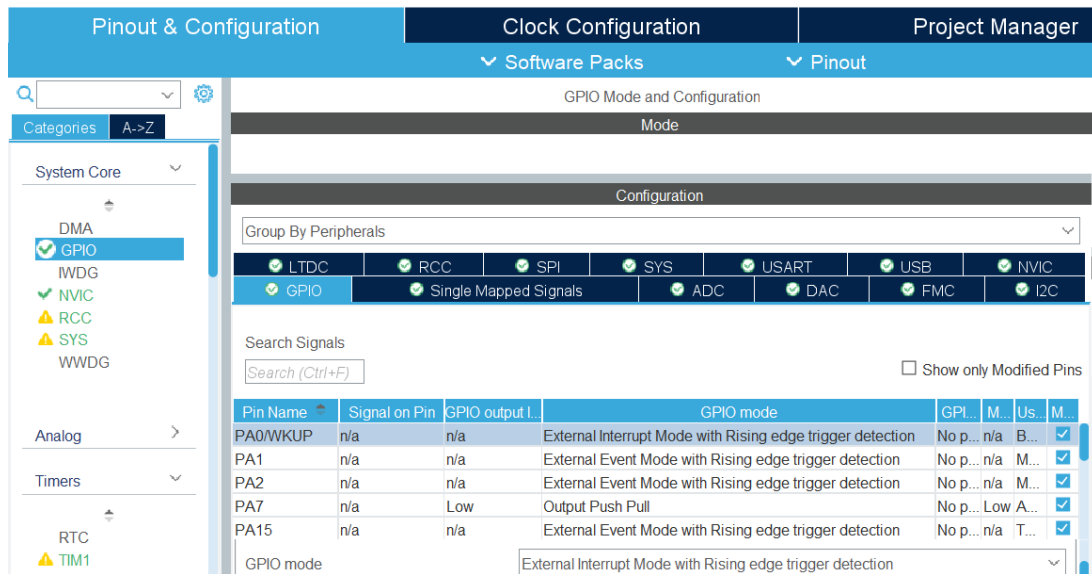


그림 5-53 PA0을 외부 인터럽트 모드로 설정

다음으로 그림 5-54와 같이 NVIC 탭을 선택해서 EXTI line0 interrupt 항목의 Enabled를 체크해서 외부 입력 인터럽트를 활성화 한다.

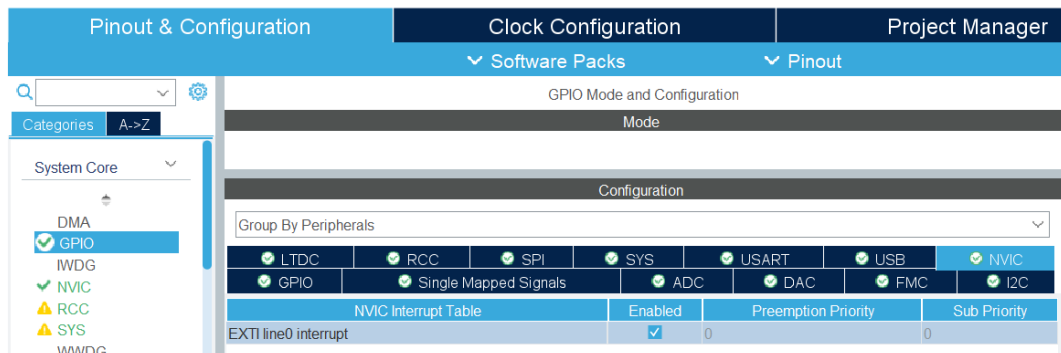


그림 5-54 Enable EXTI line0 interrupt

코드 생성을 진행한 후 아래의 코드를 입력한다.

코드 5.1

```
/* USER CODE BEGIN Includes */
#include "stdio.h"
/* USER CODE END Includes */
```

코드 5.2

```
/* USER CODE BEGIN PV */
float Kp,Kd,control;
int32_t y,oldy,ref,interrupt_counter,sampling_frequency,data_counter;
int16_t data[8001];
volatile uint8_t data_flag,data_done;
/* USER CODE END PV */
```

코드 5.3

```
/* USER CODE BEGIN 0 */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
PUTCHAR_PROTOTYPE
{
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}
/* USER CODE END 0 */
```

코드 5.4

```
/* USER CODE BEGIN 1 */
Kp=5.0;Kd=0.2;
sampling_frequency=1000;
/* USER CODE END 1 */
```

코드 5.5

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim10);
HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
/* USER CODE END 2 */
```

코드 5.6

```
/* USER CODE BEGIN WHILE */
while (1)
{
    if(data_done == 1) {
        for (int i=0; i < 4000 ;i++){
            printf("%d %d\r\n",i,data[i]);
        }
        data_flag=0;
        data_done=0;
        HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_14);
    }
}
/* USER CODE END WHILE */
```

코드 5.7

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
    data_flag=1;
}
/* USER CODE END 4 */
```

코드 5.8

```

/* USER CODE BEGIN Callback 0 */
int32_t da_value, ad_value, sum;
if (htim->Instance == TIM10) {
    sum=0;
    for (int i=0; i<20 ; i++) {
        HAL_ADC_Start(&hadc1);
        if (HAL_ADC_PollForConversion(&hadc1, 10000) == HAL_OK) {
            ad_value = HAL_ADC_GetValue(&hadc1);
            sum += ad_value;
        }
    }
    y = sum/20 - 2048;
    interrupt_counter++;
    if (interrupt_counter >= sampling_frequency*4) {
        interrupt_counter=0;
        if (data_flag==1) {
            data_counter=0;
            data_flag=2;
        }
        ref=205;
    }
    if (interrupt_counter >= sampling_frequency*2) {
        ref=0;
    }
    if (data_flag==2) {
        if (data_counter<=sampling_frequency*4) {
            data[data_counter++]= (int16_t)y;
        }
        else {
            data_done=1;
        }
    }
    control = Kp*(float)(ref-y)-(float)sampling_frequency*Kd*(float)(y-oldy);
    oldy=y;
    if (control > 2047) control = 2047;
    if (control < -2048) control = -2048;
    da_value = control + 2048;
    HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, (uint32_t)(da_value));
}
/* USER CODE END Callback 0 */

```

위의 코드에서 PD 제어기의 계수는 다음과 같다.

$$K_p = 5.0, K_d = 0.2 \quad (5.149)$$

그림 5-55는 계단 응답을 보여준다. 그림 4-53의 응답과 비교해 보면 PD 제어기가 감쇠비를 증가시키고 오버슈트를 감소시키는 것을 볼 수 있다.

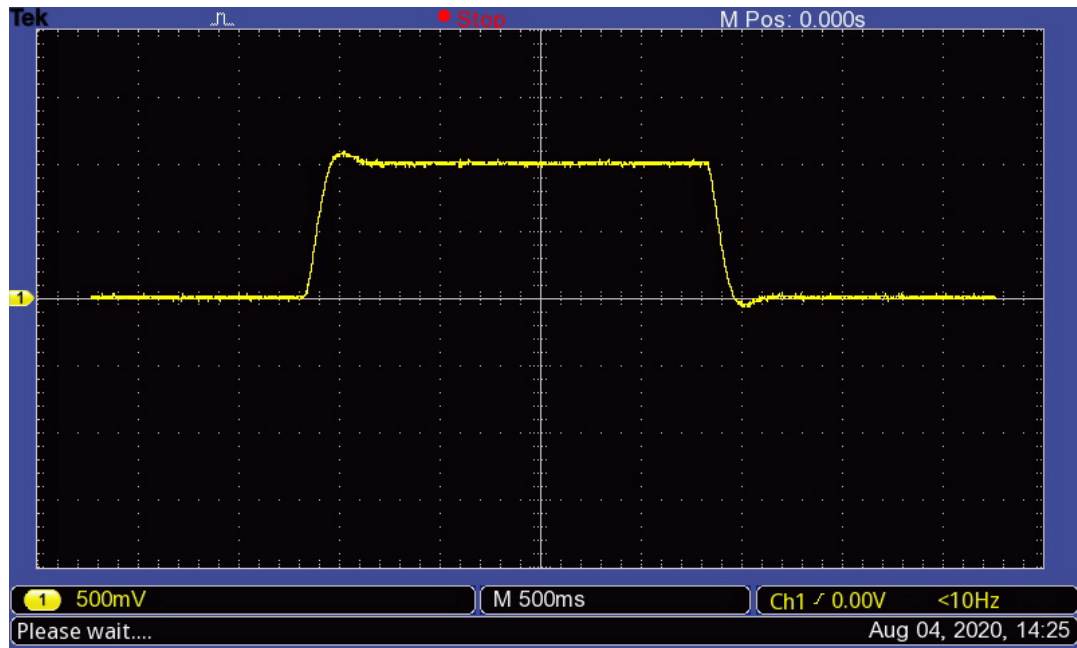


그림 5-55 계단 응답

응답 데이터를 저장해서 컴퓨터로 전송한 후 그래프를 그려본다. 먼저 SmarTTY 시리얼 터미널을 연후, 파란색 버튼 스위치를 누르고 잠시 기다린다(4초 이상). 그러면 시리얼 터미널 스크린에 데이터가 프린트 되는 것을 볼 수 있다. 각 줄의 첫째 숫자는 샘플링 카운트 값이고 두번째 숫자는 다이내믹 시뮬레이터 출력을 A/D 컨버터로 읽은 값이다. 샘플링 주파수는 1KHz이고 4초 동안의 데이터를 캡처 하므로 총 4000개의 데이터가 화면에 프린트 된다. 데이터 캡처가 끝나면 SmarTTY 메뉴의 디스켓 모양의 버튼을 눌러서 그림 5-56과 같이 데이터 파일에 저장한다.

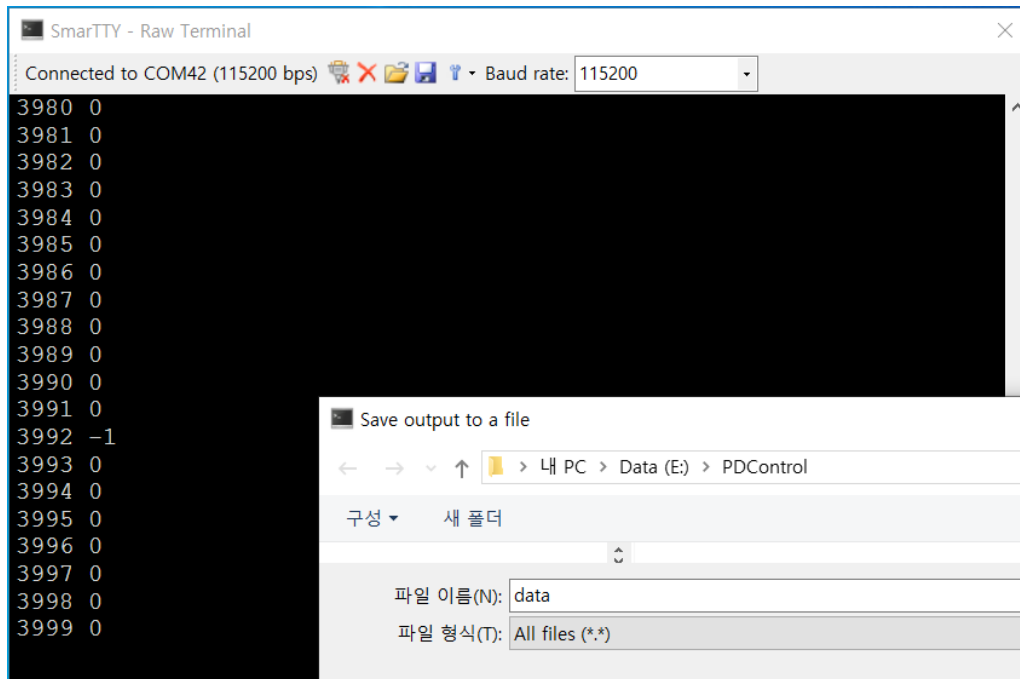


그림 5-56 캡처 데이터의 저장

아래의 코드는 데이터 파일을 읽어서 그래프를 그리는 MATLAB 코드이다. 그림 5-57은 MATLAB으로 그린 응답 데이터 그래프이다.

코드 5.9

```
clear
clf
sf=1000;
load -ascii data
for i=1:4*sf
    x(i)=data(i,1)/sf;
    y(i)=data(i,2)/205;
end
figure(1)
plot(x,y)
axis([0 4 -1 2])
xlabel('Time(sec)');
ylabel('Output/(Volt)');
grid on
```

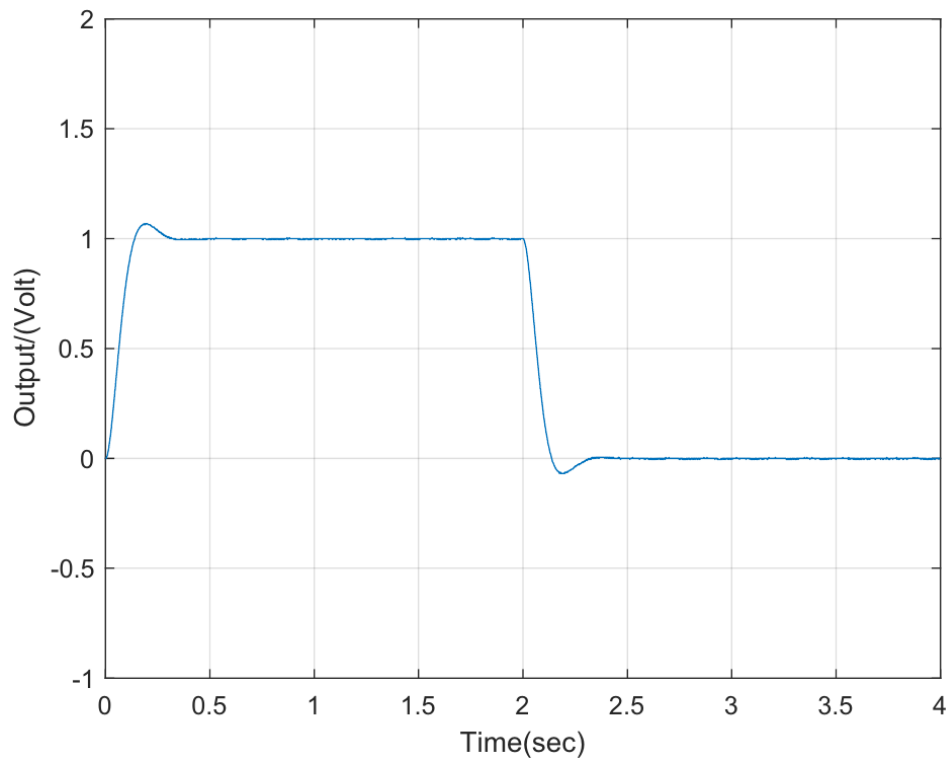


그림 5-57 MATLAB으로 그린 응답 그래프

5.5.2 DC 모터의 위치 제어기

이 실험에서는 3장 Lab3의 3.9절에서 사용한 DC 모터의 위치 제어기를 구성해 본다. 3장의 3.9.2절에 모터 모델 계수를 측정하기 위한 실험이 있었으며, 이 절에서 만드는 프로젝트의 설정은 모터 모델 계수 측정 프로젝트와 동일하다. STM32CubeIDE에서 3장의 **3.9.2 DC 모터 모델의 계수 측정**과 동일한 설정으로 새로운 프로젝트를 만든다. 코드 생성을 완료한 후 아래의 코드를 입력한다.

코드 5.10

```
/* USER CODE BEGIN Includes */
#include "stdio.h"
/* USER CODE END Includes */
```

코드 5.11

```
/* USER CODE BEGIN PV */
float Kp,Kd,control;
int32_t y,oldy,ref,interrupt_counter,data_counter,sampling_frequency;
int16_t data[2000];
volatile uint8_t data_flag,data_done;
/* USER CODE END PV */
```


코드 5.12

```
/* USER CODE BEGIN 0 */
#ifdef __GNUC__
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
PUTCHAR_PROTOTYPE
{
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}
/* USER CODE END 0
```

코드 5.13

```
/* USER CODE BEGIN Init */
sampling_frequency=1000;
Kp=8.0; Kd=0.00;
/* USER CODE END Init */
```

코드 5.14

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim10);
HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_1);
HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
/* USER CODE END 2 */
```

코드 5.15

```
/* USER CODE BEGIN WHILE */
interrupt_counter=0;
data_counter=0;
data_flag=0;
data_done=0;
while (1)
{
    if(data_done == 1) {
        for (int i=0; i < 2000 ;i++){
            printf("%d %d\r\n",i,data[i]);
        }
        data_flag=0;
        data_done=0;
        HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_14);
    }
}
/* USER CODE END WHILE */
```

코드 5.16

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
    data_flag = 1;
    interrupt_counter = sampling_frequency*2;;
}
/* USER CODE END 4 */
```

코드 5.17

```
/* USER CODE BEGIN Callback 0 */
int32_t da_value;
if (htim->Instance == TIM10) {
    y = TIM2->CNT;
    interrupt_counter++;
    if (interrupt_counter >= sampling_frequency*2) {
        interrupt_counter=0;
        if (data_flag==1) {
            data_counter=0;
            data_flag=2;
            ref = 192;
        }
        else {
            ref = 0;
        }
    }
    if (interrupt_counter >= sampling_frequency*1) {
        ref=0;
    }
    if (data_flag==2) {
        if (data_counter<=sampling_frequency*2) {
            data[data_counter++]= (int16_t)y;
        }
        else {
            data_done=1;
        }
    }
    control = Kp*(float)(ref-y)-(float)sampling_frequency*Kd*(float)(y-
oldy);
    oldy=y;
    if (control > 2047) control = 2047;
    if (control < -2048) control = -2048;
    da_value = control + 2048;
    HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R,
(uint32_t)(da_value));
}
/* USER CODE END Callback 0 */
```

실습용 보드에서 프로그램이 실행되면 파란색 사용자 버튼 스위치를 누르기 전까지 모터는 정지한 상태에서 기다린다. 사용자 버튼을 누르기 전에 SmarTTY 시리얼 터미널을 열어서

보드에서 데이터를 전송 받을 준비를 한다. 사용자 버튼을 누르면 모터는 기준 입력 값(0.5 회전)까지 회전한 후, 1초 후에 처음의 위치로 돌아온다. 모터가 최초의 위치에서 정지한 후 시리얼 터미널에는 모터의 엔코더 값이 프린트 된다. 프린트가 끝나는 것을 기다린 후 SmarTTY의 파일 저장 버튼을 눌러서 파일에 저장한다.

데이터가 저장된 파일이 준비되면 아래의 MATLAB 코드를 실행해서 응답 그래프를 그릴 수 있다.

코드 5.18

```
clear
clf
sf=1000;
load -ascii data
for i=1:2*sf
    x(i)=data(i,1)/sf;
    y(i)=data(i,2)/384;
end
figure(1)
plot(x,y)
axis([0 .2 0 1])
xlabel('Time(seconds)');
ylabel('Revolution');
grid on
```

그림 5-58은 PD 제어기 계수 $K_p=8.0$ 과 $K_d=0.0$ 에 대한 계단 응답을 보여준다. 그림 5-58의 응답은 모터의 회전 수를 출력 값으로 그린 그래프이다. 기준 입력 값이 0.5 회전이며, 정상 상태에서 모터의 출력이 기준 입력 값인 0.5에 수렴하는 것을 볼 수 있다. PD 제어기에서 미분 항에 대한 계수가 0이므로 오버슈트가 크게 나타나는 것을 볼 수 있다.

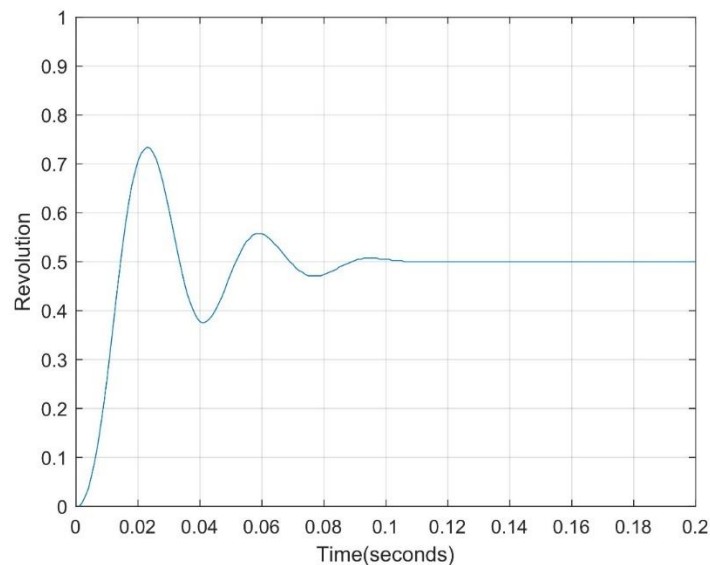


그림 5-58 $K_p=8.0$ 과 $K_d=0.0$ 에 대한 계단 응답

오버슈트를 줄이기 위해서 PD 제어기에 미분 항을 추가한다. 그림 5-59와 그림 5-60은 각각 PD 제어기 계수 $K_p=8.0$, $K_d=0.02$ 와 $K_p=8.0$, $K_d=0.04$ 에 대한 계단 응답을 보여준다. 예상한 대로 미분 항의 계수가 증가됨에 따라 오버슈트가 줄어드는 것을 볼 수 있다.

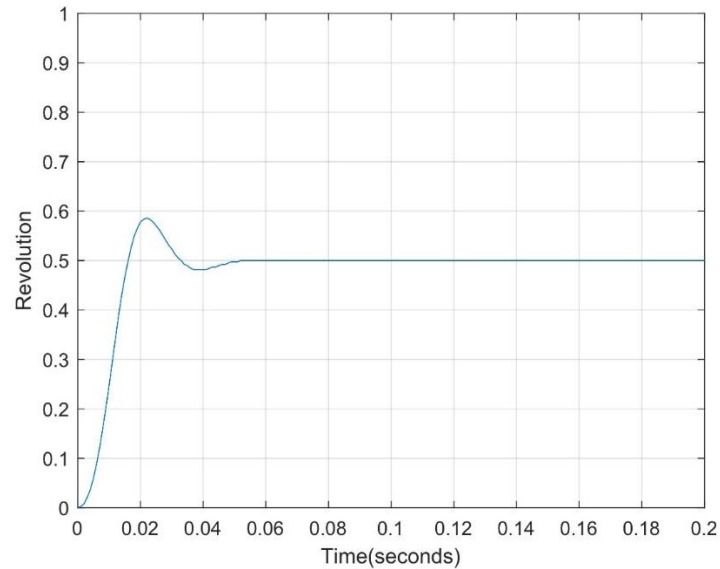


그림 5-59 $K_p=8.0$ 과 $K_d=0.02$ 에 대한 계단 응답

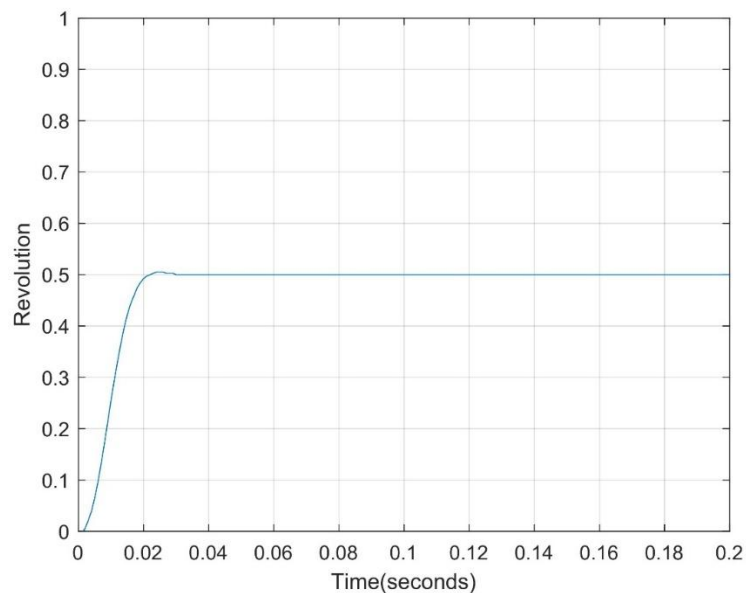


그림 5-60 $K_p=8.0$ 과 $K_d=0.04$ 에 대한 계단 응답

3장의 3.9.2절의 실습에서 구한 DC 모터의 모델을 이용해서 MATLAB 시뮬레이션을 실행해 보고 위의 실험에서 얻은 결과와 비교해 볼 수 있다. 그림 5-61은 이 실험에서 사용된 DC

모터의 위치 제어 시스템의 블록 선도이다. 실험 결과와 비교하기 위해서 D/A 변환기와 엔코더의 이득 값이 포함된 것을 주목한다. 모터의 한 회전에 해당하는 엔코더 카운트 값은 384이므로 엔코더의 카운트 값을 회전수로 변환하기 위해서는 $384/(2\pi)$ 를 곱해야 한다. 또한 제어기의 계산 결과 값을 D/A 변환기로 출력할 때, 파워 앰프에는 D/A 변환기에 전달하는 값이 아닌 전압으로 입력이 되므로 전압 값으로 변환하기 위해서 $(10/2048)$ 의 값을 곱해야 한다.

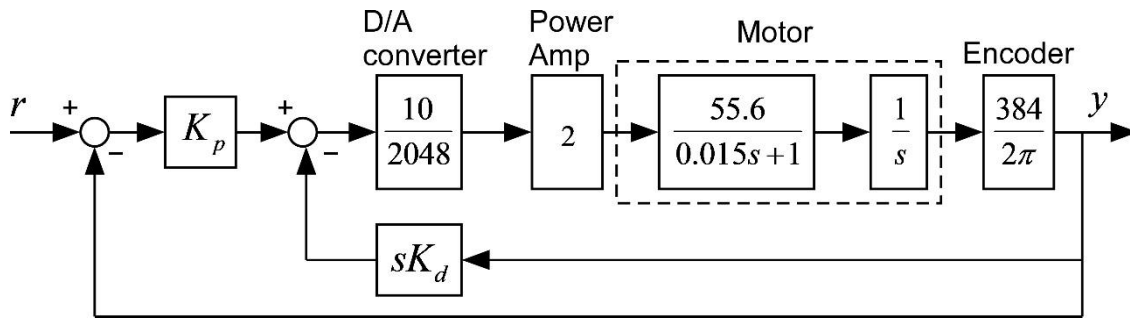


그림 5-61 DC 모터 위치 제어 시스템의 블록 선도

아래의 코드는 그림 5-61의 블록 선도의 MATLAB 시뮬레이션 코드이다.

코드 5.19

```
Kp=8;Kd=0.0;Ki=0;
D=Kp+tf([Ki],[1 0])
G=tf([2*55.6*384/(2*3.14*204.8)],[0.015 1 0])
G1=feedback(G, tf([Kd 0],[1]))
Gt=feedback(D*G1,1)
t=0:0.001:0.2;
R=0.5*ones(size(t));
figure(2)
lsim(Gt,R,t)
pole(Gt)
axis([0 .2 0 1])
xlabel('Time');
ylabel('Revolution');
grid on
```

그림 5-62, 그림 5-63, 그림 5-64는 위에서 실시한 DC 모터 위치 제어 실험과 동일한 PD 제어기 계수에 대해서 시뮬레이션을 실행한 결과이다. 앞에서 언급한 바와 같이 실제 모터에는 비선형 마찰이 존재하지만 시뮬레이션 모델에는 비선형 마찰이 포함되어 있지 않다. 따라서 시뮬레이션의 응답에 비해서 실제 실험의 응답은 대체로 오버슈트가 작고 감쇠 속도가 빠른 것을 관찰할 수 있다. 이와 같은 차이가 있음에도 불구하고 실험 결과와 시뮬레이션은 PD 제어기의 계수 변환에 따른 응답의 변화 경향은 유사한 특징을 보여주는 것을 알 수 있다.

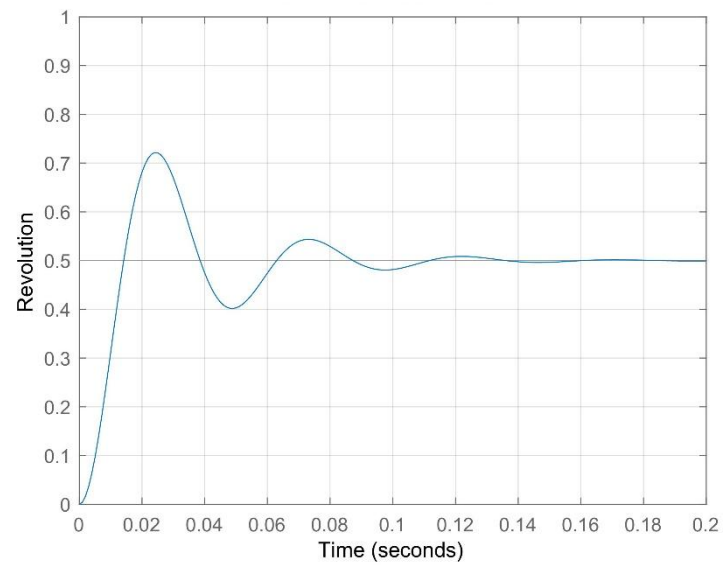


그림 5-62 $K_p=8.0$ 과 $K_d=0.0$ 에 대한 스텝 응답 시뮬레이션

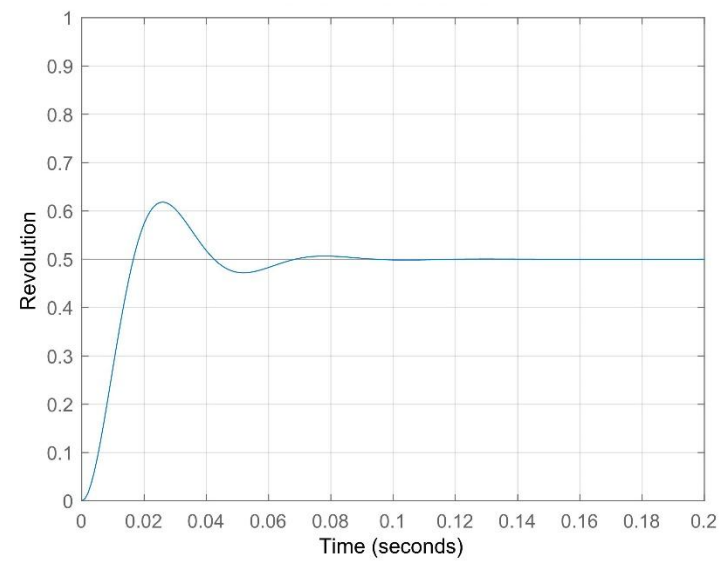


그림 5-63 $K_p=8.0$ 과 $K_d=0.02$ 에 대한 스텝 응답 시뮬레이션

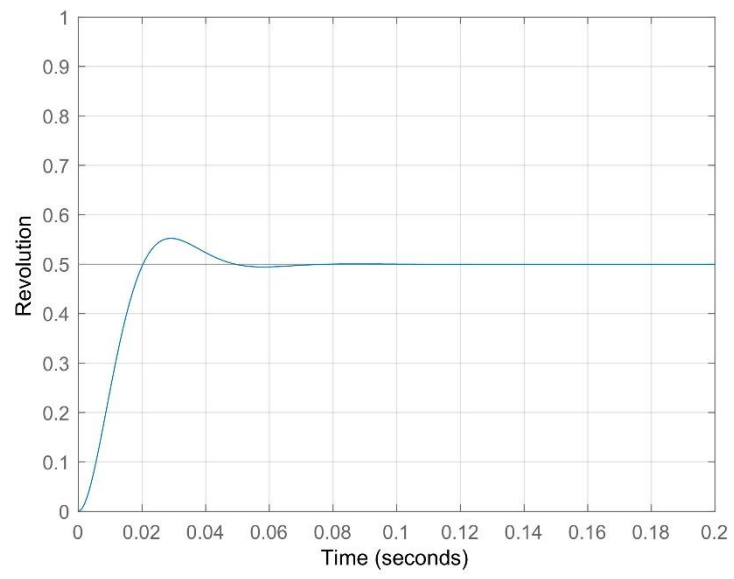


그림 5-64 $K_p=8.0$ 과 $K_d=0.04$ 에 대한 스텝 응답 시뮬레이션