ME 586: Lab 7 Report
Servo Table PID Control

Harsh Savla & TJ Wiegman

2022-11-08

Abstract

In this lab, we created C programs for the STM32 microcontroller and tested them on a STM32F100RB. These programs aimed to control a servo table system, which took in an analog voltage in order to set the servo position and output an analog voltage to feedback the current postion. Using a PID controller, we were able to make the servo step to a new position and stabilize within 2% of the goal value within 500 ms while experiencing minimal overshoot.

1 Lab Checkpoints

1.1 System Calibration

Overview

The first task was to construct a simple Servo Table Amplifier and calibrate the system per the lab manual's requirements.

Procedure

The servo table system was constructed following the instructions in the lab manual. The amplifier box was kept switched off while flashing any code to the STM32F100RB in order to avoid any damge to the system. DAC output range was measured in order to ensure the minimum and maximum output voltages to be within the safe range of $-10\,\text{V}$ to $10\,\text{V}$.

A C program, as shown in Appendix A.1, was developed to send out 20 equally spaced DAC counts from 1000 to 3000 through channel AO0. After a key press in the serial terminal, the DAC output and ADC input were measured, as shown in Table 1,.

Results and Discussion

The servo table system was calibrated successfully. As shown in Table 1, the maximum and minimum DAC output voltages were nearly $4.6\,\text{V}$ and $-5.1\,\text{V}$, respectively. The corresponding maximum and minimum servo table angles were +81 degrees and -83 degrees. In addition, both the DAC and the ADC had significantly finer resolution than one degree, by at least an order of magnitude.

1.2 System Identification

Overview

The second task of this lab was to use the data collected in the previous section to identify the open-loop response of the servo table system.

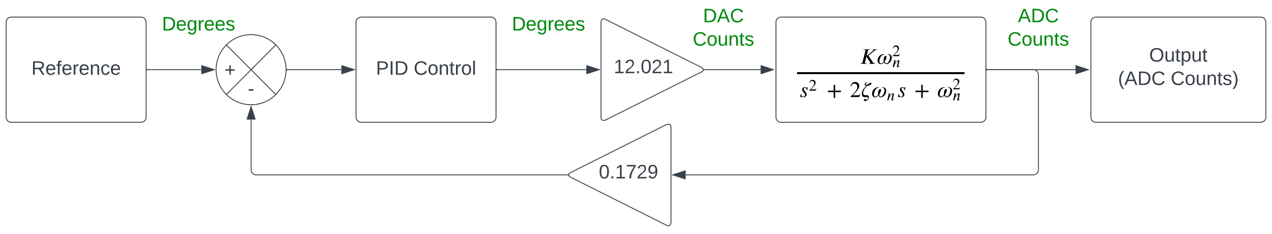
Table 1: Calibration data recorded via the program given in Appendix A.1.

DAC (Counts)	DAC (Voltage)	Angle (Degrees)	ADC (Voltage)	ADC (Counts)
1000	-5.106	-83	1.353	2604
1200	-4.135	-67	1.607	2709
1400	-3.163	-53	1.823	2797
1600	-2.189	-35	2.086	2904
1800	-1.216	-22	2.252	2973
2000	-0.241	-3	2.521	3082
2200	0.73	16	2.787	3190
2400	1.705	29	2.966	3262
2600	2.676	49	3.236	3375
2800	3.651	67	3.492	3477
3000	4.624	81	3.717	3571

Procedure

Initially, the system was modeled as shown in Figure 1. In this model, the PID controller operated entirely in degrees space, which simplified the analysis and simulation of the system. With only two unit conversions required, to go from degrees to DAC counts before the plant and from ADC counts to degrees after it, the plant's gain K was in units of ADC counts per DAC count. To calculate the other parameters of the system, such as ω_n and ζ, each trial was analyzed using the Matlab code given in Appendix A.2 and the mean of each parameter was recorded across all trials (after removing any obvious outliers).

Figure 1: The flow diagram for the original system model, which used a PID controller that operated entirely in degrees space.



Results and Discussion

The following values were obtained for the system parameters: $K = 0.0883$, $\omega_n = 36.3599$, and $\zeta = 0.0744$. The two unit conversions, from desired degrees to DAC counts and from ADC counts to measured degrees, were determined by linear regression of the calibration data collected in the previous section, as shown in Figures 2 and 3.

1.3 Controller Implementation

Overview

The third task was to implement the PID controller that was developed and simulated in Homework 8, as well as to observe how the controller reacted to external disturbances.

Figure 2: A linear regression showing the conversion from the desired angle, in degrees, to DAC counts.

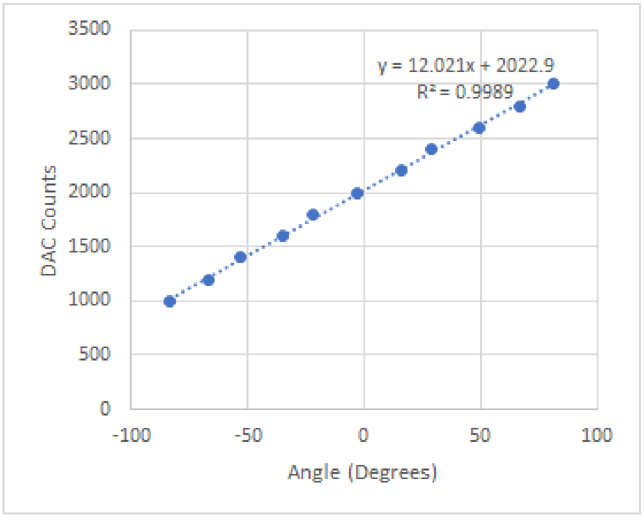
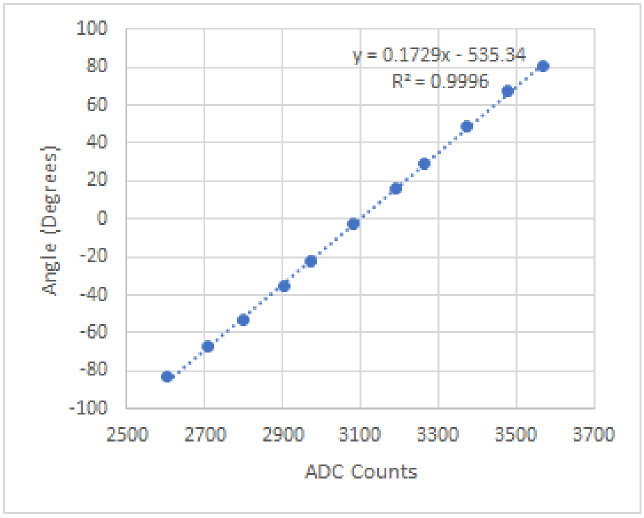


Figure 3: A linear regression showing the conversion from the measured angle, in ADC counts, to degrees.



Procedure

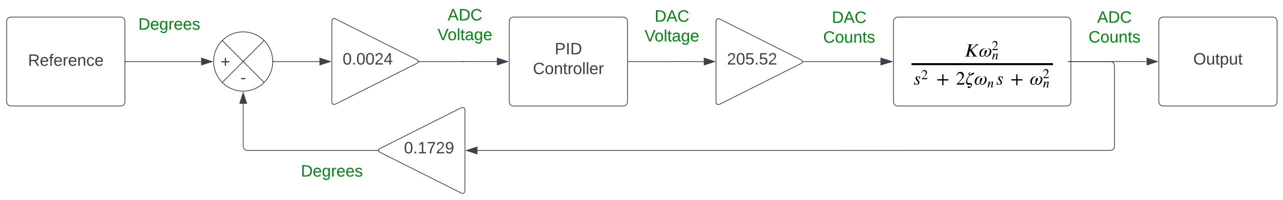
The code for the controller used a UI developed earlier, for Homework 7, which allowed the user to change the set point and PID gains, run the controller, and display data measured during operation. The DAC output channel of STM32F100RB was connected to the servo table's motor drive input, and the feedback potentiometer output was connected as an input to the ADC on the microcontroller.

Once a stable control system was verified, the servo was manually disturbed in both positive and negative directions and the response was measured for each case.

Results and Discussion

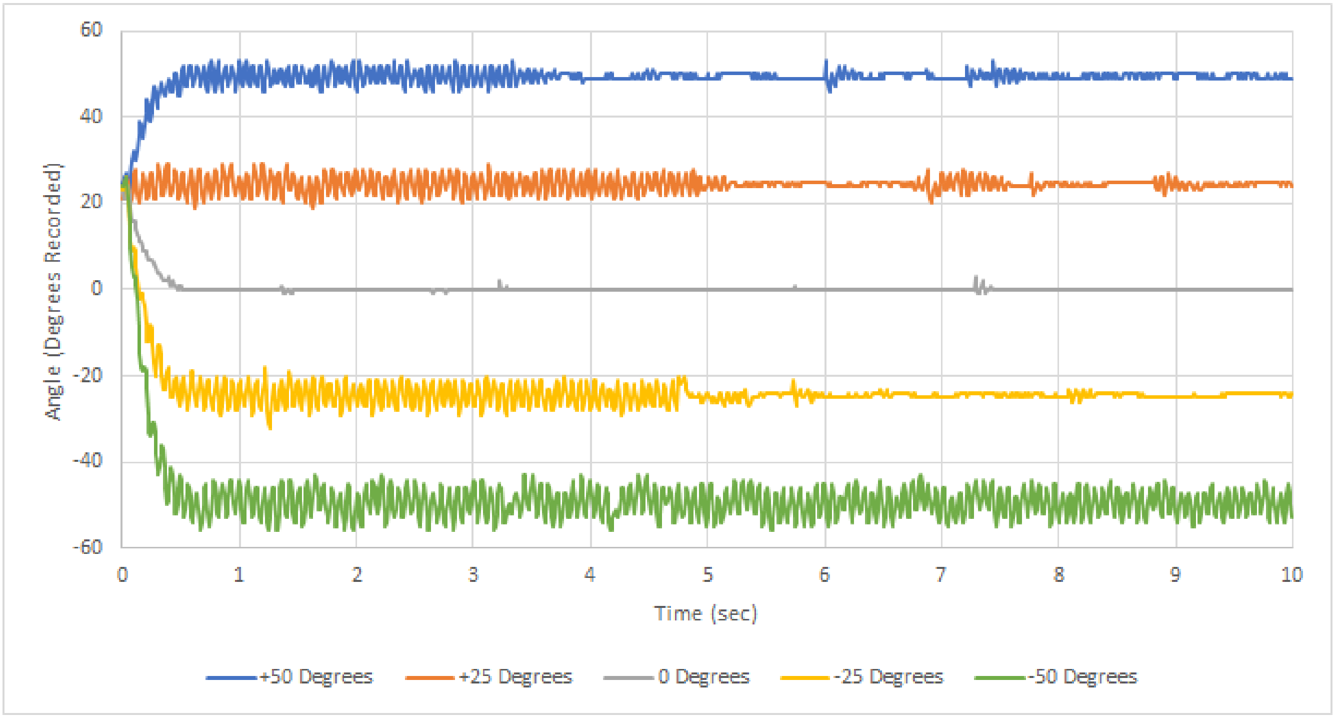
Initially, the controller developed for this task operated entirely in degrees space, using the steady-state mappings between DAC counts, ADC counts, and the degrees output by the system. While this model worked well in Simulink, it did not accurately predict the behavior of the real system: when that controller was implemented on hardware, the controller was unstable and would rapidly decay into chaotic oscillations. With more careful thought, the controller was changed to more accurately match the physical devices in the hardware system, as shown in Figure 4.

Figure 4: The flow diagram for the final system model, which used a PID controller that took in the voltage measured by the ADC and output a voltage for the DAC.



With this model, the appropriate gains were re-calculated to match the calibration data, giving the following values for the PID controller: $K_p = 0.42496$, $K_i = 111.71$, and $K_d = 0.93048$. With these gains, running on the PID algorithm given in Appendix A.3 with a 10 ms period, the system was tested with five different set points. The results from those tests are shown in Figure 5.

Figure 5: The system's response to, from top to bottom, +50, +25, 0, -25, and -50 degree input set points.



That set of experiments showed that the controller was stable and responsive for a wide range of angles. One wrinkle in the results is that the table seems to start itself at about 25 degrees, rather than the 0 degrees that were intended, but that may simply be due to a calibration error. Additionally, the device tended to make many small, noisy oscillations about the set point once it reached that position. While quite small in amplitude–within only a few degrees–they are quite obvious in the plotted data. This likely occurred because the gains were tuned quite high: even the smallest disturbance or noise would be amplified enough to break the servo's friction for a brief instant before being put back in its place by the controller.

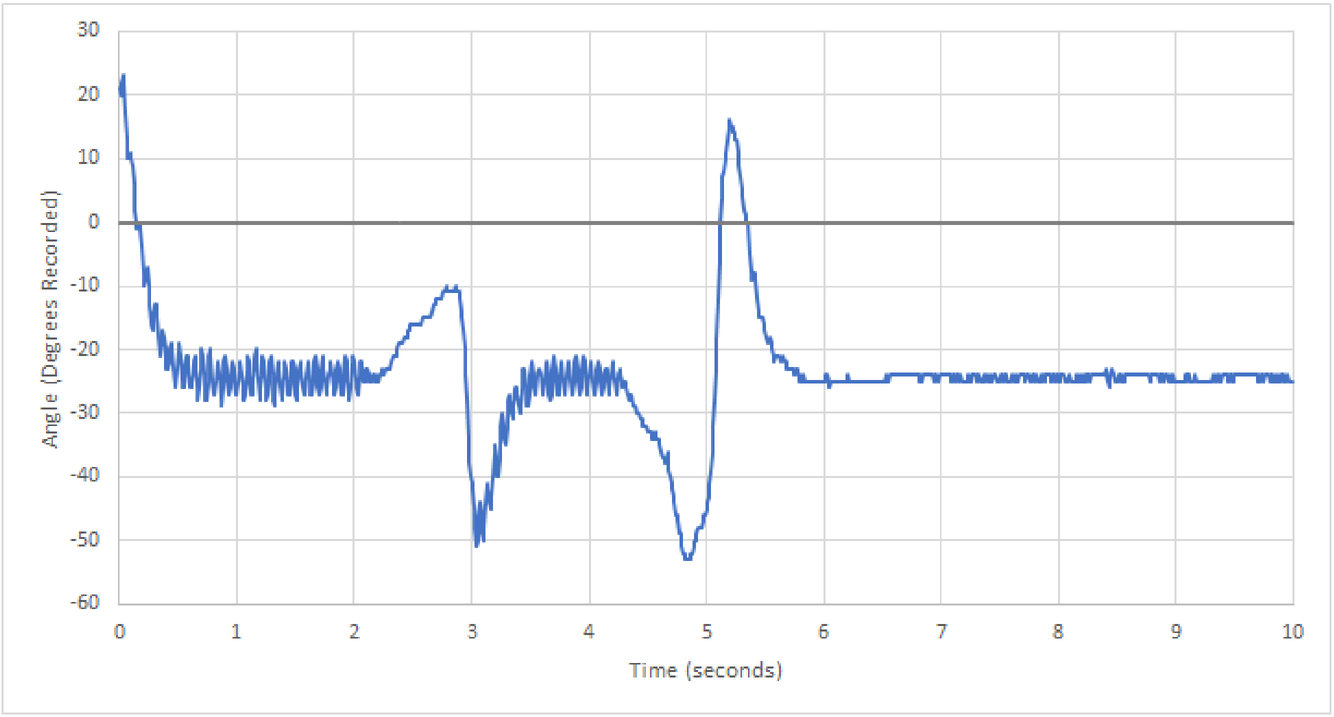
With the stability of the controller established, the final test for the system was to manually disturb the servo and measure its response. A sample of such behavior is shown in Figure 6.

2 Conclusion

This lab taught us many things about plant system identification and control system design. We had quite a bit of trouble in developing our PID controller at first, because our initial system identification was not accurate enough, so our model could not predict and compensate for the plant's behavior correctly. After we switched to a more physically accurate model (see Figure 4 for what we eventually settled on), and removed an extra offset term that had been mistakenly included in the PID transfer function code, everything went much more smoothly.

For future projects of this kind, it may be better to develop the control system incrementally, testing pieces atomically one at a time, rather than simply collecting all the data at the beginning and then analyzing it all afterwards. With luck, that sort of system would prevent some of the multi-faceted errors that made this lab much more difficult for us this time.

Figure 6: The system's response to a positive disturbance (at about $t = 2.5$) and a negative disturbance (at about $t = 4.5$) prove that it can self-correct for disturbances and errors, even large ones, in either direction. The set point in this experiment was -25 degrees.



A Appendix: Code

A.1 servo_calib.c

```c
#include "ME586.h"

int main(void) {
    int value = 1000;
    initcom();
    initadc();
    initdac();

    printf("DAC:\t ADC:\n\r");

    while(1){
        WaitForKeypress();
        d_to_a(0, value);
        shownum(value);
        putchar('\t');

        WaitForKeypress();
        shownum(a_to_d(1));
        printf("\n\r");

        value = value + 200;
        if(value > 3000){
            break;
        }
    }
}
```

A.2 systemID.m

```matlab
function [K, zeta, wn, settle, osp, data, finalAngle] = secondOrderAnalyze(data, time, stepDAC)
    zeroAngle = mean(data(1:150)); % get average ADC of initial angle
    data = data - zeroAngle; % normalize by initial angle
    finalAngle = mean(data(700:1000)); % assuming settled after 7 seconds

    % Get static gain, in units of ADC counts per DAC count above 2048
    K = finalAngle / (stepDAC - 2048);

    % Get 2-percent settling time
    settle = time(find(abs(data - finalAngle) >= 0.02 * abs(finalAngle), 1, 'last'));

    % Get overshoot percentage
    osp = (max(abs(data)) - abs(finalAngle)) / abs(finalAngle);

    % Get zeta
    zeta = abs(log(osp))/sqrt(log(osp)^2 + pi^2);

    % Get damped frequency (fourier transform to find peak)
    destep = [data(1:200); data(201:1000) - finalAngle];
    L = length(time);
    P2 = abs(fft(destep)/L);
    P1 = P2(1:L/2+1);
    P1(2:end-1) = 2*P1(2:end-1);
    P1 = P1(5:end-5); % ignore outliers on edges
    fAngular = 2*pi * 100*(0:(L/2))/L;
    wd = fAngular(find(P1 == max(P1), 1));

    % Get natural frequency
    % algebra from wd = wn*sqrt(1 - zeta^2) and 2% settle time = 4 / zeta*wn
    wn = sqrt((wd^2)*(settle^2) + 16) / settle;
end
```

A.3 PID Controller (Timer Interrupt)

```c
void timehand(void) {
   // Setup
   int measured_ADC;
   float measured_angle;
   float error;
   float derivative;
   float output;

   // Get position from ADC input
   measured_ADC = a_to_d(1);
   measured_angle = 0.1729*measured_ADC - 535.34; // convert ADC count to degrees
   if (stored_actual < 1000) {
      ram_ptr[stored_actual] = measured_angle; // store degrees to memory
      stored_actual = stored_actual + 1;
   }

   // PID Controller
   error = setpoint - measured_angle;
   error = 0.0024*error;  // convert degrees to ADC voltage (no offset inside PID)
   integral = integral + (error * (time_period / 1000)); // convert ms to seconds
   derivative = (error - error_prior) / (time_period / 1000);
   output = (Kp * error) + (Ki * integral) + (Kd * derivative);

   if (output > OUTPUT_MAX) output = OUTPUT_MAX;
   if (output < OUTPUT_MIN) output = OUTPUT_MIN;
   error_prior = error;
   output = output*205.52 + 2049.8; // convert voltage to DAC
   d_to_a(0, output);
}
```