Ball Hopper – Optimal Control

White Paper **Updated – Mar 19, 2023**

Phi Lam, Group Portugal, ECE, University of BC, Vancouver BC, Canada

Abstract

Our project is designed to be a tennis ball hopper and an important part of this robotic claw system is the control system that enables the robot to smoothly operate in different states. The claw is controlled by one PID controller model that is simulated in Simulink. The code in the MCU and the control system were designed with the idea of achieving the RCGs stated in table 1. The code compilation is done with the Arduino IDE and the selected MCU is the ESP32-Wroom-32. We created our PID with the input and output be based on the position of the

Part one of this paper indicates the five RCGs selected and maintained throughout the project term. Part two discusses the control system model and design and part three discusses the MCU and code portion to achieve our RCGs.

Nomenclature

PID	Proportional, Integral, Derivative controller
RCG	Requirement, Constraint, Goal
MCU	Microcontroller Unit

1. RCGs

For the optimal controls portion of this project, we require the claw to move with precision and smoothly to each of its states (origin, grasping, moving, and releasing). With this information in mind, the following RCGs are made to open and close the claw accurately without overshooting and with as minimal settling time.

RCGs	Requirement	Constraint	Goal
Settle Time	Settle time must be less than 2s	Settle time can't be too fast that it will cause something to break	Minimize settle time as much as possible
Overshoot	Overshoot must be less than 10%	Cannot run faster than the control frequency	Minimize overshoot as much as possible

ISR	ISR frequency must be smaller than 12x the most dominant pole	Frequency cannot be faster than 10x the most dominant pole	Frequency needs to be as close to 10x the most dominant pole
State Indication	Code must exemplify what state of the design is in at a 1Hz frequency	Timer needs to have enough bits to run at 1Hz	Code needs to indicate all states as visible as possible
Rise time	Rise time must be less than 1s	Rise time can't be faster than settling time	Minimize rise time as much as possible

Table 1: RCGs for the optimal control portion

2. Controller System

For our control system, we have two inputs that perform at different times, one with the goal to close the claw and one with the goal to open the claw. This feeds into our PID with $K_p, K_i,$ and K_d parameters that is then fed into the second order amplifier. After this, we feed the signal into our electrical admittance then apply a K_m gain to it and apply a 60-ratio gain value, as the gear ratio is 60 to 1. The electrical admittance outputs a torque value which becomes the input of our mechanical admittance subsystem and outputs speed. This signal gets fed back into the admittance motor loop and a torque constant gain K_t.

2.1 Amplifier

To enable the motor to have enough torque and power to perform its role, we use an H-bridge. We obtained our second order amplifier transfer function from using an oscilloscope to obtain data and used the MATLAB script in Appendix A to obtain the transfer function shown below (1). The step response graph of this model is also shown down in Appendix B.

$$\frac{8748}{s^2 + 54\,s + 729} \quad (1)$$

2.2 Motor

Our electrical admittance was modelled by measuring the inductance and resistance of the motor. Through the electrical admittance model, we convert voltage to current. The represented transfer function is shown below in (2).

$$\frac{1}{Ls+R} = \frac{1}{0.001996s+2.182} \quad (2)$$

2.3 Tuning

Combining all of these in Simulink we can get the corresponding system shown in Figure 1.

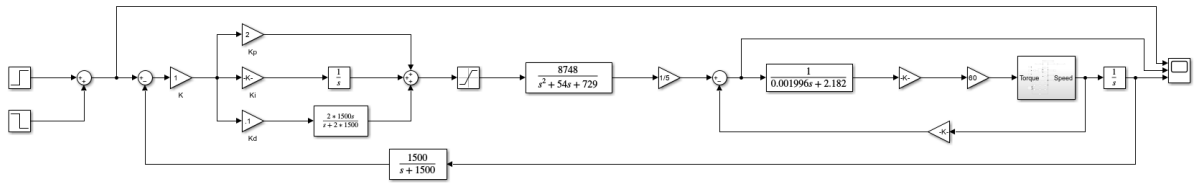


Figure 1: Control system of claw

Tuning the PID to the desired move in Figure 1, I use the method of manually tuning the PID following the steps noted down in my references [2]. I first set the integral and derivative gains to zero before I change the proportional gain to a small oscillating wave. Then, I increase the integral gain a little bit to stabilize the proportional gain without too much overshoot. Then, I increase the derivative gain to dampen the signal and then increase the proportional gain again until my signal settles at the input line. This results in the red line shown in Figure 2. I use the method of manually tuning the PID incorporating non-linearities of the mechanical model can help achieve more accurate gains and because I did not receive the mechanical admittance values to replace the subsystem with a transfer function.

Overall, combing all the components we use in this model, we were able to tune the graph to fit into our RCGs with less than 10% overshoot and roughly 0.5s of settle time. Rise time also takes about 0.5s.

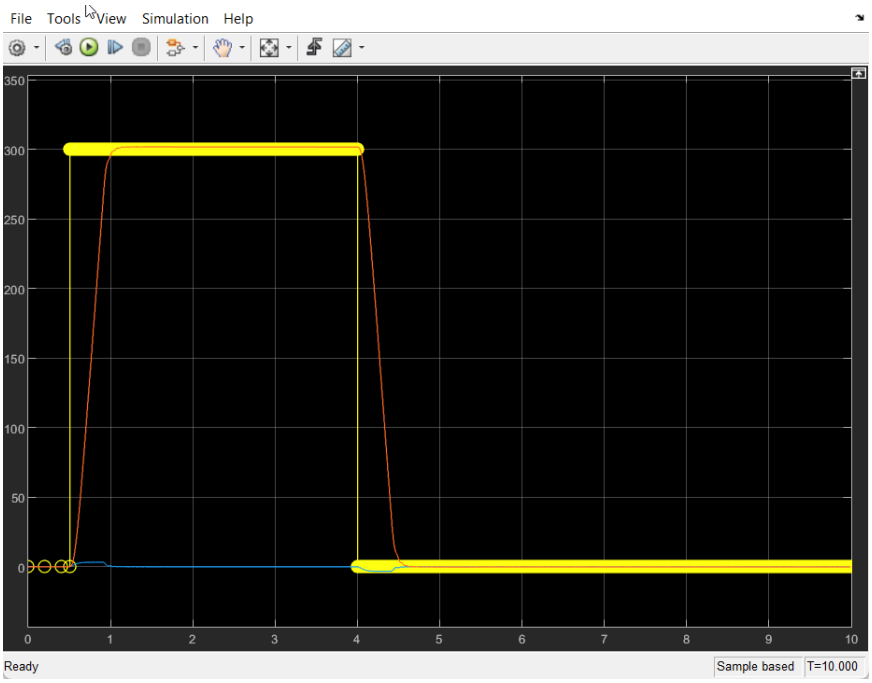


Figure 2: PID Position output

2. Microcontroller

2.1 MCU Selection

The selected MCU is an ESP32-WROOM-32. It has a 32 pin layout with the flexibility to allow any GPIO output pins to be made into PWM pins. We selected this MCU to allow us to run more ISRs as there are 2 hardware timers available with 64 bit each.

2.2 ISR

We run one internal ISR for the PID, one external ISR to open the claw when the metal sensor detects a magnet. The frequency for this ISR is set to be 10x the most dominant pole of our control system in Figure 1, 1.5kHz or more. So using the code in Figure 3, I configure the frequency of our motor PID for the ESP32 to be roughly 15kHz. The overall clock frequency of the ESP32 is 80MHz. Taking this into mind, we first need to use a prescalar of 1 and a timer overflow of 5333 to have our PID ISR run at 15kHz.

```
// ISRs
pidTimer = timerBegin(2, 1, true); // pidTimer number 2, divider = 1 bc no prescalar needed
timerAttachInterrupt(pidTimer, &PID, true);
timerAlarmWrite(pidTimer, 5333, true); // Every 15Khz PID will run 80MHz/5333 = 15Khz
motorPID.SetMode(AUTOMATIC);
```

Figure 3: PID frequency configuration

2.3 State Indication

To clearly identify the state of the system, I ran the LED at a 1Hz frequency as shown in the code below in figure 4. I run the led at different PWMs at each state, 70% at its normal state, 10% while closing, 40% while travelling at 100% while opening the claw. I model the output of the LED to make sure the PWM at each state corresponds correctly and it can be seen below in Figure 5.

```
#define LEDFREQ 1
#define PRESETPOINT 8544

/* PINS */
#define MOTORPIN 10
#define IN1PIN 11
#define IN2PIN 12
#define IN1CHANNEL 1
#define IN2CHANNEL 2
#define LEDPIN 32
#define LEDCHANNEL 0
```

```
// LED states at 1Hz PWM and 16 bit resolution at channel 0
ledcAttachPin(LEDPIN, LEDCHANNEL);
ledcSetup(LEDCHANNEL, LEDFREQ, 16);
```

```
ledcWrite(LEDCHANNEL, TENPERC);
```

Figure 4: Code for state indication

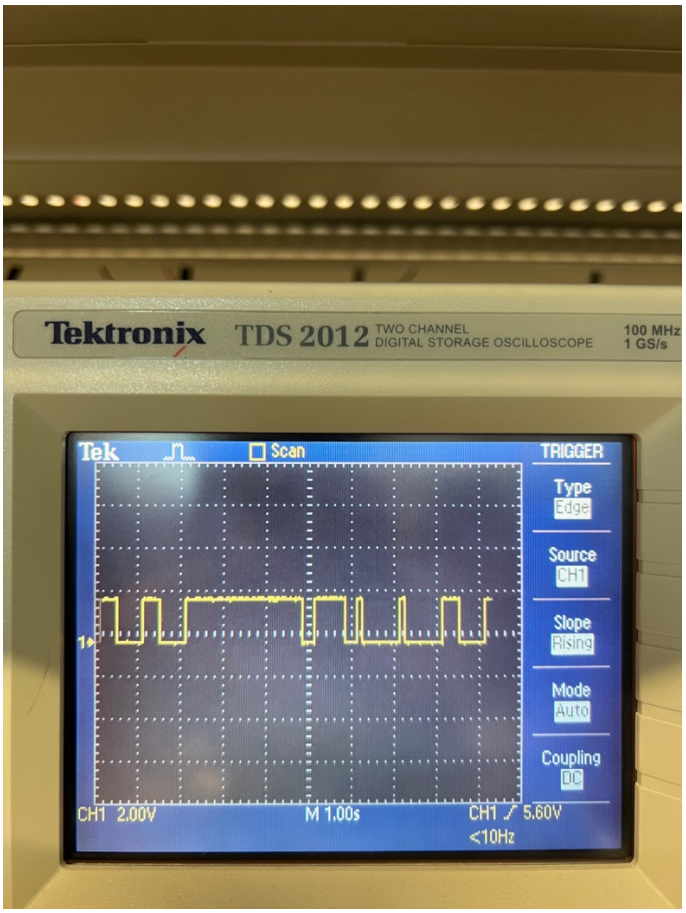


Figure 5: PWM model of LEDs 70 -> 10 ->40 ->100

References

[1] ESP32 Series Datasheet, 2023,
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

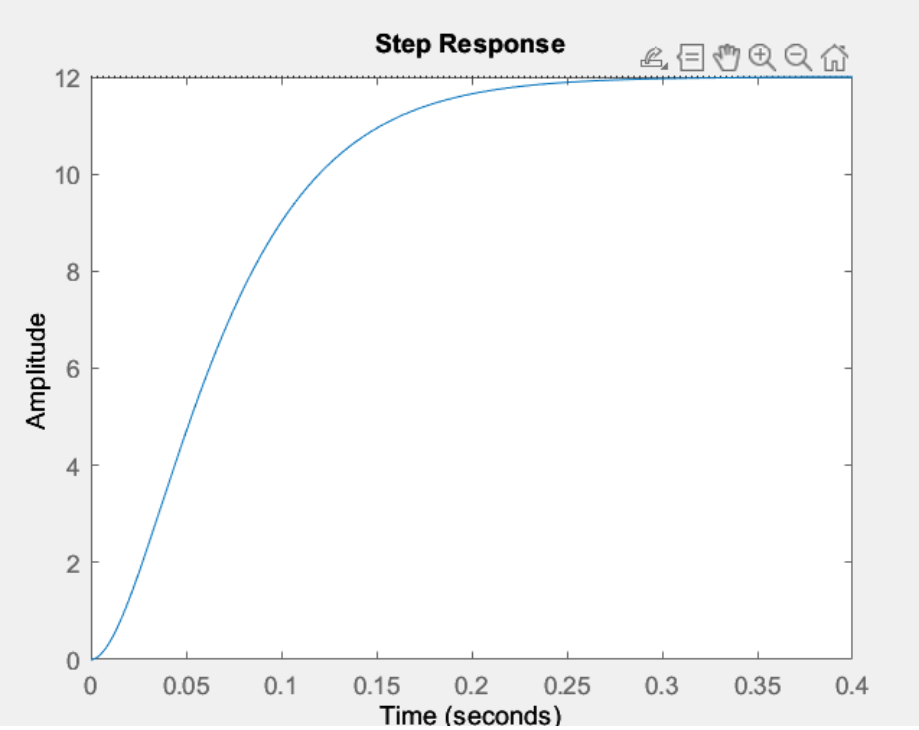
[2] Driver PID Settings, 1999-2023,
https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=9013

Appendix:

Appendix A:

```
function transferFunction = getTransferFunction(naturalFrequency, gain, dampingCoefficient) %#ok<DEFNU>
    transferFunction = minreal(tf((((naturalFrequency)^2) * gain), [1 (2 * dampingCoefficient * naturalFrequency) ((naturalFrequency)^2)]));
end
```

Appendix B:



Appendix C:

```
/* IMPORTED LIBRARIES */
#include <PID_v1.h>
#include <HCSR04.h>

/* MISC CONSTANTS */
#define PLDCOMM 10
#define MAXPULSE 16383 // 14-bit
#define MOTORFREQ 30000
#define LEDFREQ 1
#define PRESETPOINT 8544

/* PINS */
#define MOTORPIN 10
#define IN1PIN 11
#define IN2PIN 12
#define IN1CHANNEL 1
#define IN2CHANNEL 2
#define LEDPIN 32
#define LEDCHANNEL 0
#define SENSOR2PIN 33
#define ECHOPIN 26
#define TRIGGERPIN 27
```

```
/* LED DUTY CYCLES */
#define TENPERC 6554
#define FORTYPERC 26214
#define ONEHUNPERC 65536
#define SEVENTYPERC 45876

/* GLOBAL VARIABLE DECLARATION */
double setPosition;
double input, output;
double Kp = 0.5, Ki = 0.1, Kd = 0.5;
int pulse = 0;
int pulseOverflow = 0;
bool home = true;
bool flag = false;

/* INITIALIZE PID TIMER, MOTOR PID MODEL, AND SENSOR */
hw_timer_t *pidTimer = NULL;
UltraSonicDistanceSensor distanceSensor(TRIGGERPIN, ECHOPIN);
PID motorPID(&input, &output, &setPosition, Kp, Ki, Kd, DIRECT);

void setup()
{
  Serial.begin(9600);

  pinMode(MOTORPIN, OUTPUT);
  pinMode(IN1PIN, OUTPUT);
  pinMode(IN2PIN, OUTPUT);
  pinMode(SENSOR2PIN, INPUT);

  // LED states at 1Hz PWM and 16 bit resolution at channel 0
  ledcAttachPin(LEDPIN, LEDCHANNEL);
  ledcSetup(LEDCHANNEL, LEDFREQ, 16);

  // Motor PWM setup 30kHz at channel 1 and 2 at 8 bit resolution
  ledcAttachPin(IN1PIN, IN1CHANNEL);
  ledcSetup(IN1CHANNEL, MOTORFREQ, 8);
  ledcAttachPin(IN2PIN, IN2CHANNEL);
  ledcSetup(IN2CHANNEL, MOTORFREQ, 8);

  // ISRs
  pidTimer = timerBegin(2, 1, true); // pidTimer number 2, divider = 1 bc no prescalar
needed
  timerAttachInterrupt(pidTimer, &PID, true);
  timerAlarmWrite(pidTimer, 5333, true); // Every 15Khz PID will run 80MHz/5333 =
15Khz
  motorPID.SetMode(AUTOMATIC);

  attachInterrupt(digitalPinToInterrupt(SENSOR2PIN), openClaw, RISING);
```

```
}

void loop()
{
  ledcWrite(LEDCHANNEL, SEVENTYPERC);
  digitalWrite(MOTORPIN, LOW);
  float distance = distanceSensor.measureDistanceCm();

  Serial.print("Distance from object: ");
  Serial.println(distance);

  if (distance <= 15.0 && distance >= 0 && home)
  {
    closeClaw();
  }
}

void closeClaw()
{
  Serial.println("Entering closeClaw state");
  ledcWrite(LEDCHANNEL, TENPERC);
  setPosition = 8544;
  flag = true;

  // Initialize motor pin without PWM
  digitalWrite(MOTORPIN, HIGH);
  ledcWrite(IN2PIN, 0);

  // Start PID
  home = false;
  timerAlarmEnable(pidTimer);

  // Should not get to this case, but goes to next state if PID > 30s
  delay(30000);

  // Next state: Travel Claw
  travelClaw();
}

void PID()
{
  if (!home)
  {
    int pulseTemp = 0b0;
    bool pin;

    for (int i = 0; i < PLDCOMM; i++)
    {
```

```
      // PIN D0 - D9 (D0-D9)
      pulseTemp = pulseTemp << 1 | int(digitalRead(i));
    }

    Serial.print("Current pulse, prev pulse, overall pulse, max pulse: ");
    Serial.println(pulseTemp);
    Serial.println(pulseOverflow);
    Serial.println(pulse);
    Serial.println(MAXPULSE);

    // Motor running backwards
    //  case that pulse count overflows backwards from 200 -> 15950
    //  or motor goes backward 800 -> 400, chose 1000 as buffer as
    //  pulses should not be greater than that with PID ISR
    if ((pulseTemp)*0.5 > pulseOverflow || (pulseOverflow - pulseTemp < 1000 &&
pulseOverflow - pulseTemp > 0))
    {
      pulse = pulse - (MAXPULSE + pulseOverflow - pulseTemp);
      pin = IN2PIN;
      Serial.print("Motor running backwards");
    }
    else
    {
      // Motor running forwards
      //  case if pulse is regularly larger or is less than prev pulse
      pulse = pulse + (pulseTemp - pulseOverflow);
      pin = IN1PIN;
      Serial.print("Motor running forwards");
    }

    pulseOverflow = pulseTemp;
    input = pulse;
    motorPID.Compute();

    ledcWrite(pin, 255 * abs((setPosition - output) / PRESETPOINT));

    Serial.print("New overall pulse, PID output: ");
    Serial.println(pulse);
    Serial.println(output);

    // PID settles close enough
    if (output <= setPosition + 1 && output >= setPosition - 1)
    {
      timerAlarmDisable(pidTimer);

      if (flag)
      {
        travelClaw();
```

```
      }
      else
      {
        openClaw();
      }
    }
  }
}

void travelClaw()
{
  Serial.println("Entering travelClaw state");
  home = true;
  ledcWrite(LEDCHANNEL, FORTYPERC);

  // Motor stop spin, sanity check
  ledcWrite(IN1PIN, 0);
  ledcWrite(IN2PIN, 0);

  // Should not get to this case, but goes to next state if
  // it doesn't see sensor for > 30s
  delay(30000);

  // Next state: openClaw()
  openClaw();
}

// Last state (ISR):
void openClaw()
{
  Serial.println("Entering openClaw state");
  // ISR sense
  ledcWrite(LEDCHANNEL, ONEHUNPERC);
  setPosition = 0;
  flag = false;

  ledcWrite(IN1PIN, 0);

  // Start PID
  home = false;
  timerAlarmEnable(pidTimer);

  // Should not get to this case, but goes to next state if PID > 30s
  delay(30000);

  // Next state: Home
  home = true;
}
```