Haptic Mouse Bungee

Eric Wang

December 8, 2020

Background/Product Description

Recently, I have noticed that my gaming mouse's braided USB cable frequently catches on nearby objects or the edge of my mousepad since my desk is often cluttered. Mouse bungees attempt to solve this problem by securing the mouse wire on an elastic platform (fig. 1). However, I find that these wired mouse accessories are unwieldy to use because they are designed to apply a larger restoration force the further the mouse is dragged away from the bungee. Furthermore, the spring's equilibrium angle and stiffness cannot be adjusted to account for mouse location bias. Overall, these factors fail to replace intermittent cable snags with decreased resistance (as if the mouse were wireless), but instead make the user fight against an uncomfortable elastic force. My haptic mouse bungee (fig. 2) is designed to solve these issues by using motorized feedback to simulate a ratcheted spring, allowing the device to adjust its desired position based on proprioceptive feedback and center the "spring's" equilibrium angle to best suit the mouse's current position. In addition, it gives the user the option to easily alter the effective virtual spring stiffness.

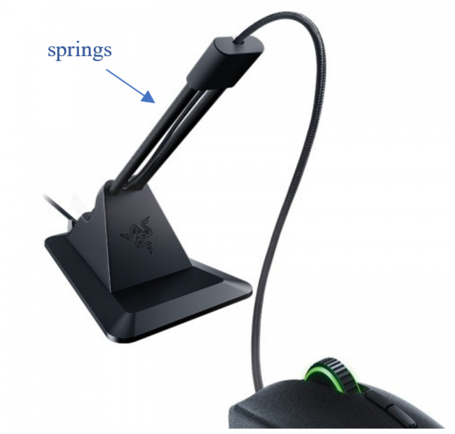


Figure 1: Conventional Mouse Bungee

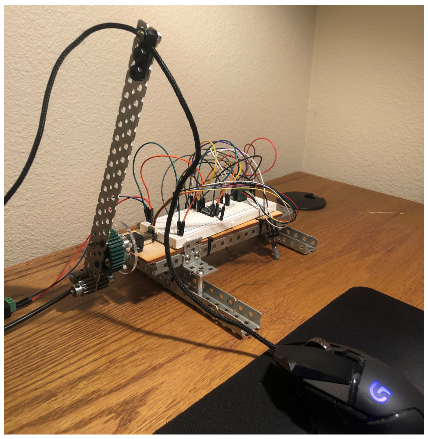


Figure 2: Haptic Mouse Bungee

Electromechanical Design

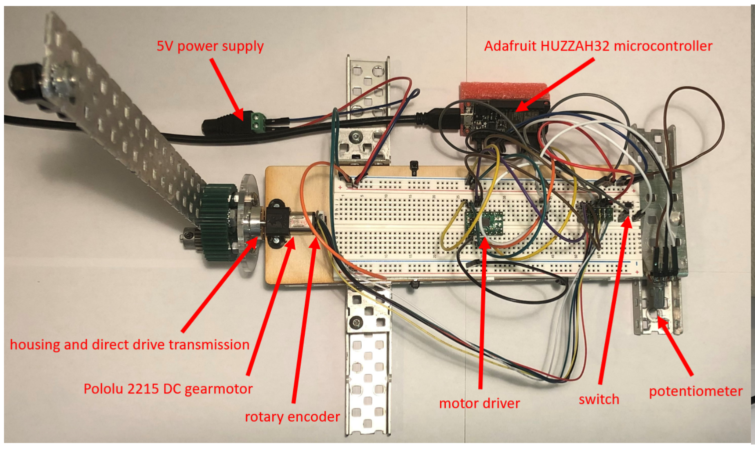


Figure 3: Main Body

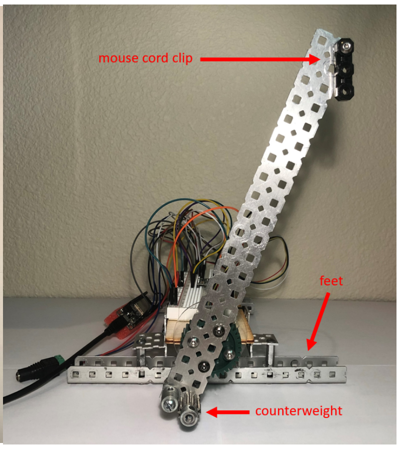


Figure 4: Bungee

This prototype is made from lab kit elements, VEX robotics structural pieces, as well as scavenged household components. The added VEX parts include hole-punched aluminum sheet metal and C-channels, threaded hex screws, keps nuts, steel standoffs, zip-ties, screw-in collars, steel square-crossection axles, and gears.

Main Body: The main body of the device contains the circuitry and provides a rigid interface for the motor housing and the breadboard (fig. 3). Power for the actuator is supplied through a 5V connector and power for the microcontroller is done via USB, since this device is primarily used in conjunction with a computer mouse. A button switch and potentiometer are provided on the right for the user to switch between "unlocked" and "locked" states (explained in FSM section) and adjust spring stiffness.

Bungee: A direct drive transmission was chosen over a geared or linkage transmission for two reasons: (1) proprioceptive feedback requires minimal backlash and friction and (2) a linkage would produce a variable mechanical advantage that is dependent on crank angle and would make it difficult to simulate a linear spring. A counterweight is added to counter the large lever arm so that the motor does not require a feedforward torque to fight gravity. The clip holds a mouse cord in place at the end of the lever.

Circuit

The circuit contains three peripheral sensors (button switch, potentiometer, and rotary encoder) and one actuator (Pololu 2215 motor) which is powered by a 5V power source. A description of the circuit components is as follows:

- **button switch:** The switch is connected to a GPIO digital input with a pullup resistor. The pin is programmatically tied to an interrupt so that the embedded system does not need to constantly check the state of the switch.

- **potentiometer:** The potentiometer is connected to an Analog-to-Digital (ADC) converter pin which converts the analog voltage across the variable resistor to a digital signal. One end of the resistor is connected to the 3V pin.
- **rotary encoder:** The encoder, like the potentiometer is powered via the microcontroller's 3V pin. The two channels are connected to separate GPIO input pins each with 45kΩ pullup resistors.
- **motor driver/motor:** Unlike the auxiliary powered components, the motor is powered by the 5V power supply and controlled via pulse width modulation (PWM) through the motor driver, which contains an H-bridge so that it is backdrivable.

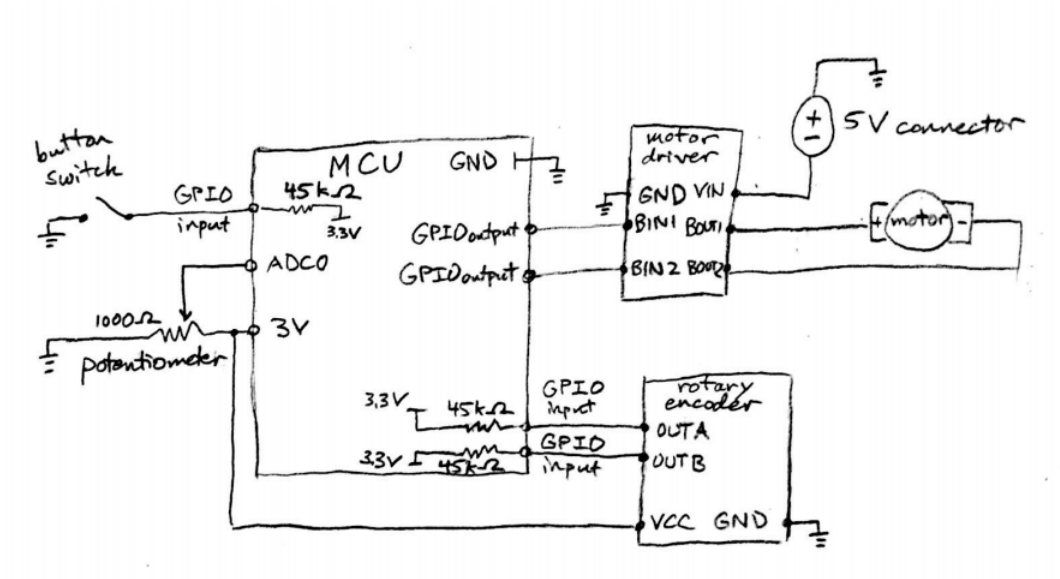
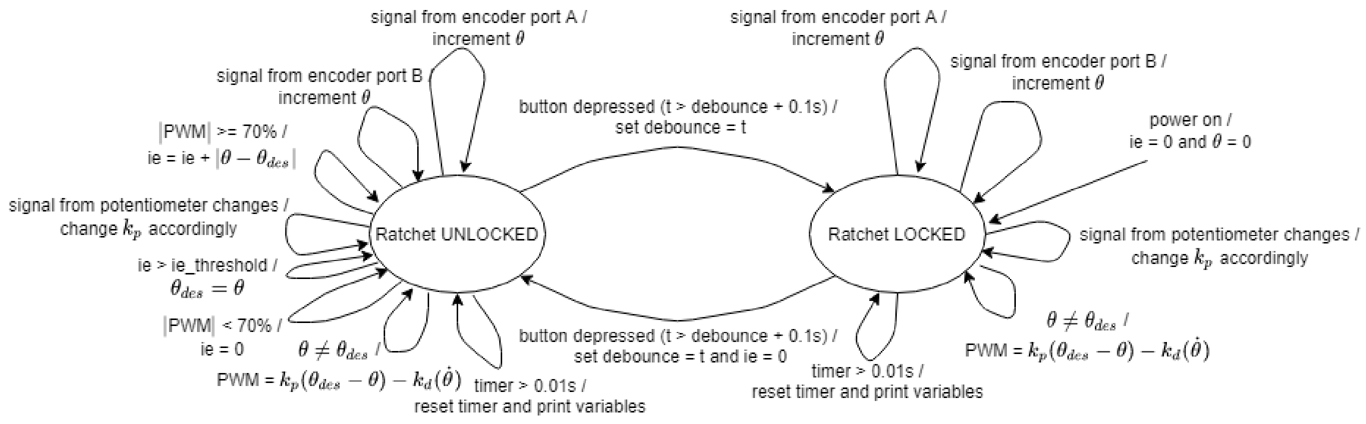


Figure 5: Circuit Diagram

Finite State Machine

The heart of this project lies within the integration of feedback with the mechanical design to produce interactive haptic behavior. There are two states - "locked" and "unlocked" which are toggled by depressing the switch. The "locked" state fixes the equilibrium position of the spring, notated as θ_{des}, and acts like a pure spring damper. The "unlocked" state allows the equilibrium position of the spring to change if an external torque causes the motor to exert more than 70% PWM for ~150ms. This indicates that the user is tugging at the mouse unnecessarily and θ_{des} shifts to the current angle θ. Thus, in the "unlocked" state, the device acts as a spring loaded ratchet. The finite state diagram is shown below:



signal from encoder port A / increment θ
signal from encoder port B increment θ
$|PWM| >= 70\%$ / ie = ie + $|\theta - \theta_{des}|$
signal from potentiometer changes / change k_p accordingly
ie > ie_threshold / $\theta_{des} = \theta$
$|PWM| < 70\%$ / ie = 0
$\theta \neq \theta_{des}$ / PWM = $k_p(\theta_{des} - \theta) - k_d(\dot{\theta})$
timer > 0.01s / reset timer and print variables
Ratchet UNLOCKED
button depressed (t > debounce + 0.1s) / set debounce = t
button depressed (t > debounce + 0.1s) / set debounce = t and ie = 0
Ratchet LOCKED
signal from encoder port A / increment θ
signal from encoder port B / increment θ
power on / ie = 0 and θ = 0
signal from potentiometer changes / change k_p accordingly
$\theta \neq \theta_{des}$ / PWM = $k_p(\theta_{des} - \theta) - k_d(\dot{\theta})$
timer > 0.01s / reset timer and print variables

Figure 6: Finite State Diagram

An example of this ratchet behavior is shown below. The red line indicates the angle of the bungee, θ and the blue line indicates the θ_{des}. The shaded blue regions show the time intervals at which the PWM supplied by the motor exceeds 70%, causing the integral variable "ie" to increment. Once "ie" exceeds a threshold, $\theta_{des} = \theta$, making the ratchet perform one step. While this is occurring, the PD controller produces a local spring behavior in the region about θ_{des} and this can be shown around 8s, where the measured angle overshoots and settles at some steady state value.

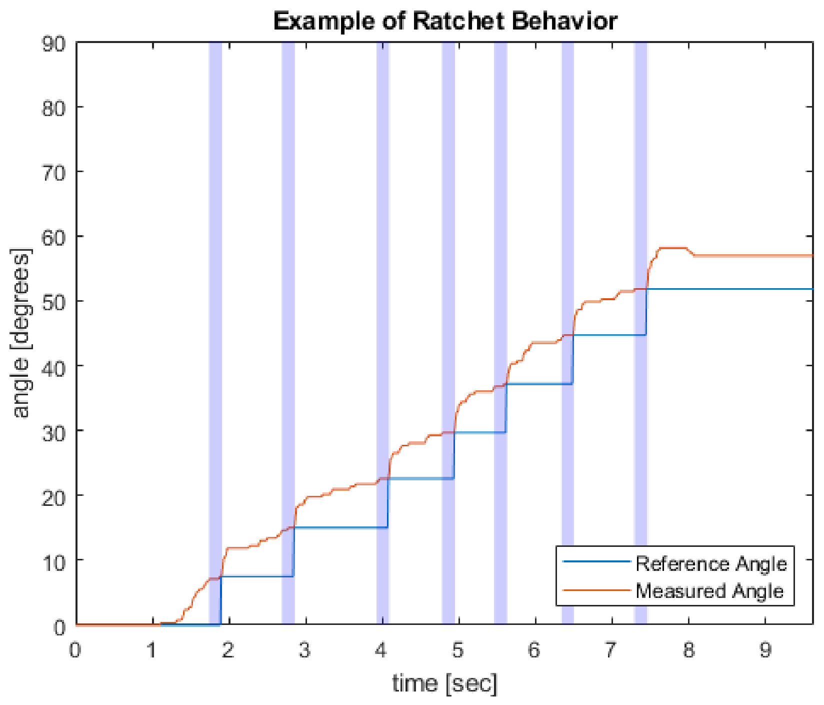


Figure 7: Ratchet Behavior

Appendix I: Arduino Code

```
#include <ESP32Encoder.h>
// sensor and output pins
#define b1 21
#define b2 17
#define buttonInput 16
#define ea 18
#define eb 19
int sensorPin = A2;

// constants
int debounce = 100;
struct Button {
  const uint8_t PIN;
  bool pressed;
};
Button button1 = {buttonInput, false};
double t_Δ = 0.01;
double RPM_per_tick = 0.396/360*60/t_Δ;
double tick_per_degree = 910/360;
hw_timer_t * timer = NULL;
ESP32Encoder encoder;
float kp = 4;
float ki = kp*0.02;
float kd = kp/3;
float ie_max = 300;
int shiftPwm = 170;

// variables
int sensorValue = 0;
int resistance = 0;
int pwm = 0;
unsigned long timeM;
int counter = 0;
unsigned long count = 0;
double rot_speed = 0;
int ticks = 0;
bool toPrint;
float w_des = 0;
float th_des = 0;
int effort = 0;
float ie = 0;
float dp = 0;
float di = 0;

// Timer interrupt for updating encoder counter and ...
    integral error
void IRAM_ATTR onTimer(){
  // updates rotational speed
  rot_speed = (double)(encoder.getCount()-ticks)*RPM_per_tick;
  // updates position
  ticks = encoder.getCount();
  // updates integral error
  if (abs(th_des-ticks) > shiftPwm/kp) {
    ie = ie + abs(th_des-ticks);
  } else {
    ie = 0;
  }
  // saturates integral error
  ie = min(ie, ie_max);
  toPrint = true;
}

// Interrupt function called when switch is depressed
void IRAM_ATTR isr() {
  // checking debounce condition
  if (millis() - timeM > debounce) {
    button1.pressed = !button1.pressed;
    timeM = millis();
  }
}

// Interrupt function for encoder ticks
void IRAM_ATTR isrE() {
  // increments counter each time interrupt is called
  counter++;
}

// Function to compute control effort
int control(float th_des, float ticks) {
  effort = kp*(th_des - ticks) - kd*rot_speed;
  effort = min(max(effort, -255), 255);
  return effort;
}


void setup() {
  // initializes serial
  Serial.begin(115200);

  // sets up PWM output pins
  ledcAttachPin(b1, 1);
  ledcSetup(1, 12000, 8);
  ledcAttachPin(b2, 2);
  ledcSetup(2, 12000, 8);

  // sets button input
  pinMode(buttonInput, INPUT_PULLUP);
  attachInterrupt(button1.PIN, isr, RISING);
  // sets encoder input
  ESP32Encoder::useInternalWeakPullResistors=UP;
  encoder.attachFullQuad(ea, eb);
  encoder.setCount(37);
  encoder.clearCount();

  // sets potentiometer inputs and initial map
  sensorValue = analogRead(sensorPin);
  resistance = map(sensorValue,0,4095,0,1000);

  // starts timers
  timeM = millis();
  timer = timerBegin(0, 80, true);
  timerAttachInterrupt(timer, &onTimer, true);
  timerAlarmWrite(timer, t_Δ*1000000, true);
  timerAlarmEnable(timer);

  // initializes variables
  pwm = 0;
  counter = 0;
  rot_speed = 0;
  toPrint = false;
  w_des = 0;
  ie = 0;
}

void loop() {
  // reads potentiometer value and maps it to PWM values
  sensorValue = analogRead(sensorPin);
  resistance = map(sensorValue,0,4095,0,1000);
  // calling PI controller function
  pwm = control(th_des, ticks);
```

```
// mapping potentiometer values to desired effective ...
   stiffnesses
kp = map(resistance, 0, 1000, 10, 20);
ki = kp*0.02;
kd = kp/3;
// logic to account for negative pwm values
if (pwm > 0) {
  ledcWrite(1, pwm);
  ledcWrite(2, 0);
} else {
  ledcWrite(1, 0);
  ledcWrite(2, -pwm);
}

// main FSM logic
switch (button1.pressed){
  case true:
    // adaptive desired angle
    if (ie == ie_max) {
      th_des = ticks;
    }
    break;
  default:
    // locked state
    ie = 0;
    toPrint = false;
    break;
}
switch (toPrint) {
  case true:
    // prints data to copy and paste into text file
    Serial.print(millis());
    Serial.print(" ");
    Serial.print(String(th_des));
    Serial.print(" ");
    Serial.print(String(ticks));
    Serial.print(" ");
    Serial.print(String(rot_speed));
    Serial.print(" ");
    Serial.print(String(pwm));
    Serial.print(" ");
    Serial.print(String(kp));
    Serial.print(" ");
    Serial.println(String(ie));
    toPrint = false;
```

```
        break;
      default:
        break;
    }
}
```

Listing 1: Full code implementation

Appendix II: MATLAB Processing Code

```
close all, clear all
%% Initialization
fileNames = {'Tuning.txt', 'Ratchet1.txt'};
dt = 0.01;
d_per_tick = 360/910;

%% Tests
data = load(fileNames{1});
t = data(:,1)/1000;
t = t - t(1);
th_des = data(:,2);
th = data(:,3);
w = data(:,4);
duty = data(:,5)/255*100;
figure, plot(t, th_des), hold on, plot(t, th), plot(t, duty);
xlim([min(t) max(t)]);
xlabel('time [sec]');
legend({'Desired Angle', 'Measured Angle', 'PWM'}, 'Location', 'SE');
title('Testing Behavior');

%% Rachet Tests
data = load(fileNames{2});
t = data(:,1)/1000;
t = t - t(1);
th_des = data(:,2)*d_per_tick;
th = data(:,3)*d_per_tick;
w = data(:,4);
duty = data(:,5)/255*100;
kp = data(:,6);
ie = data(:,7);
shadedRegion = find(ie > 0);
figure, plot(t, th_des), hold on, plot(t, th);
if ¬isempty(shadedRegion)
    for ii = 1:length(shadedRegion)
        ind = shadedRegion(ii);
        patch([t(ind) t(ind)+dt t(ind)+dt t(ind)],[0 0 90 ...
            90],'b','FaceAlpha',0.2,'EdgeColor','none');
    end
end
```

```
xlim([min(t) max(t)]);
xlabel('time [sec]');
ylabel('angle [degrees]');
legend({'Reference Angle', 'Measured Angle'}, 'Location', 'SE');
title('Example of Ratchet Behavior');
```

Listing 2: processData.m