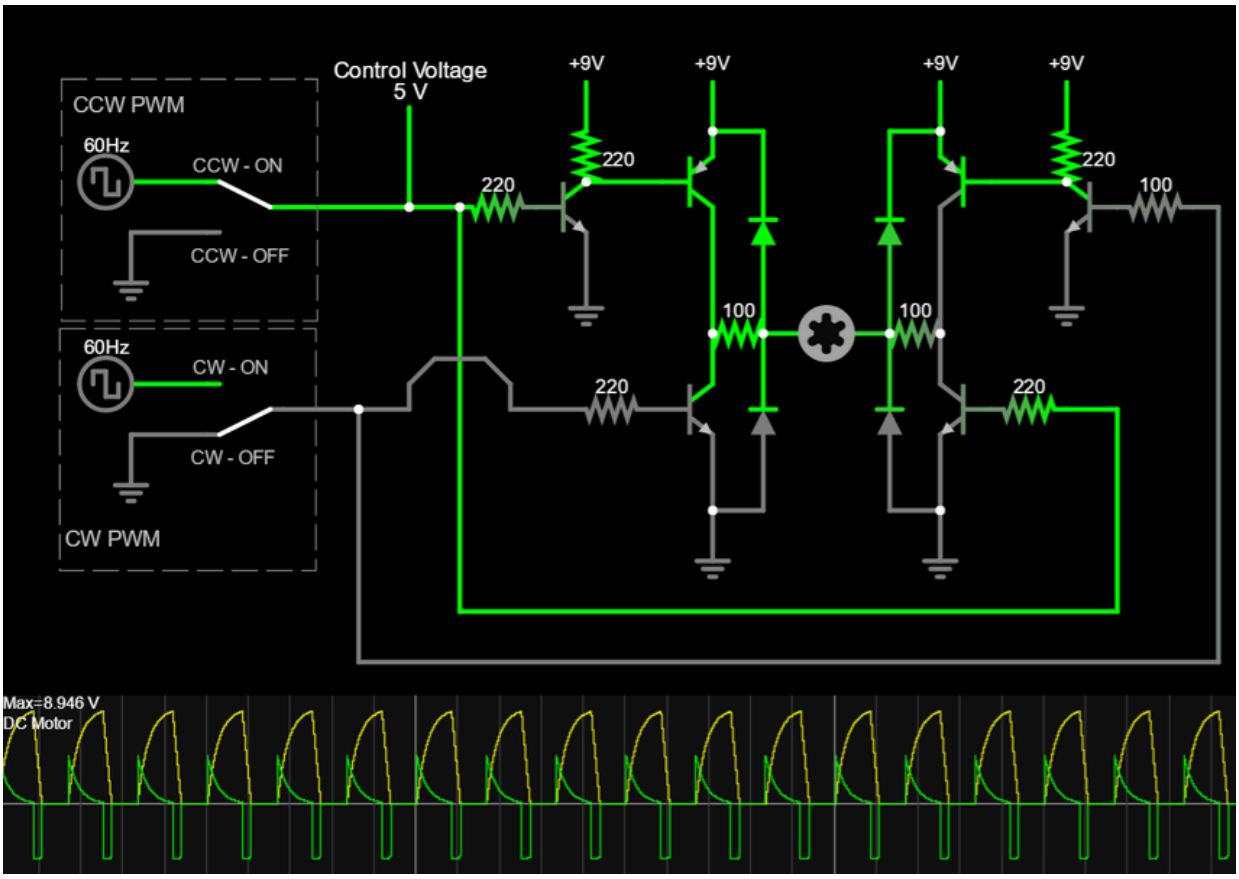
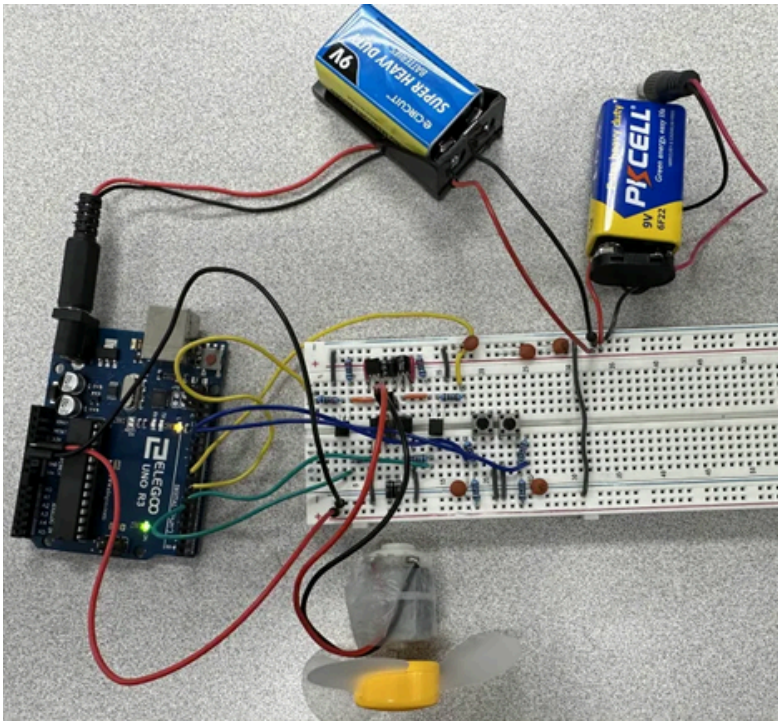
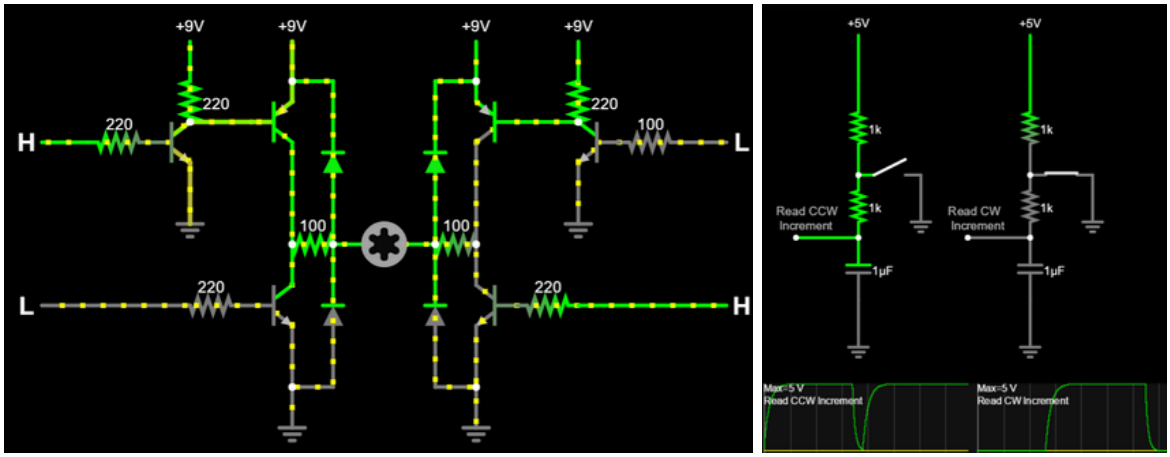
## 9V H-BRIDGE CONTROLLED WITH 5V — OCTOBER 2023

I was tasked with building an H-Bridge controlled with PWM from an Arduino. The challenge here was driving the motor with 9V when the Arduino outputted at 5V

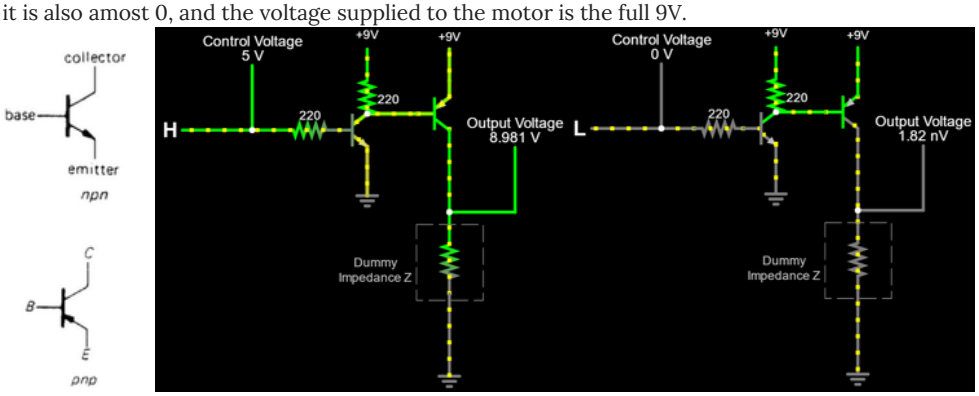


Below, it should be more clear how the motor is driven with 9V, yet is controlled with 5V. This is because it has modifications from a normal H-Bridge. The transistors are named clockwise starting from the left middle Q1, Q2, Q3, Q4. Q1 and Q2 are PNP transistors, and Q3 and Q4 are NPN. The two transistors on the far left and right are named Q5 and Q6 respectively, and are also NPN transistors.




When the Arduino outputs a high PWM signal, Q1 saturates, because in NPN transistors, if the voltage between its base and emitter will typically saturate at a V_be >= 0.7V, of which its emitter is grounded, thus 5V-0V > 0.7V. This means that the voltage drop across it is almost 0V, and therefore, the voltage at the base of Q1 is 0V. Because Q1 is a PNP transistor, and it saturates at a difference between its V_eb at <= 0.7V, and its emitter is at 9V and its base is at 9V, then the voltage drop across it is also amost 0, and the voltage supplied to the motor is the full 9V.




Shown to the right is the simple Arduino code. There are buttons to change the speed and direction that the motor spins at. As the left button is pressed, it increments the counterclockwise speed until it hits its maximum. It could also slow down the motor if it was moving clockwise.

To prevent the user from holding down the button for too long and hitting the max speed unexpectedly, there are conditional statements when buttons are pressed. This allows the button to detect if it was unpressed before it got pressed, before deciding to update the speed of the motor.

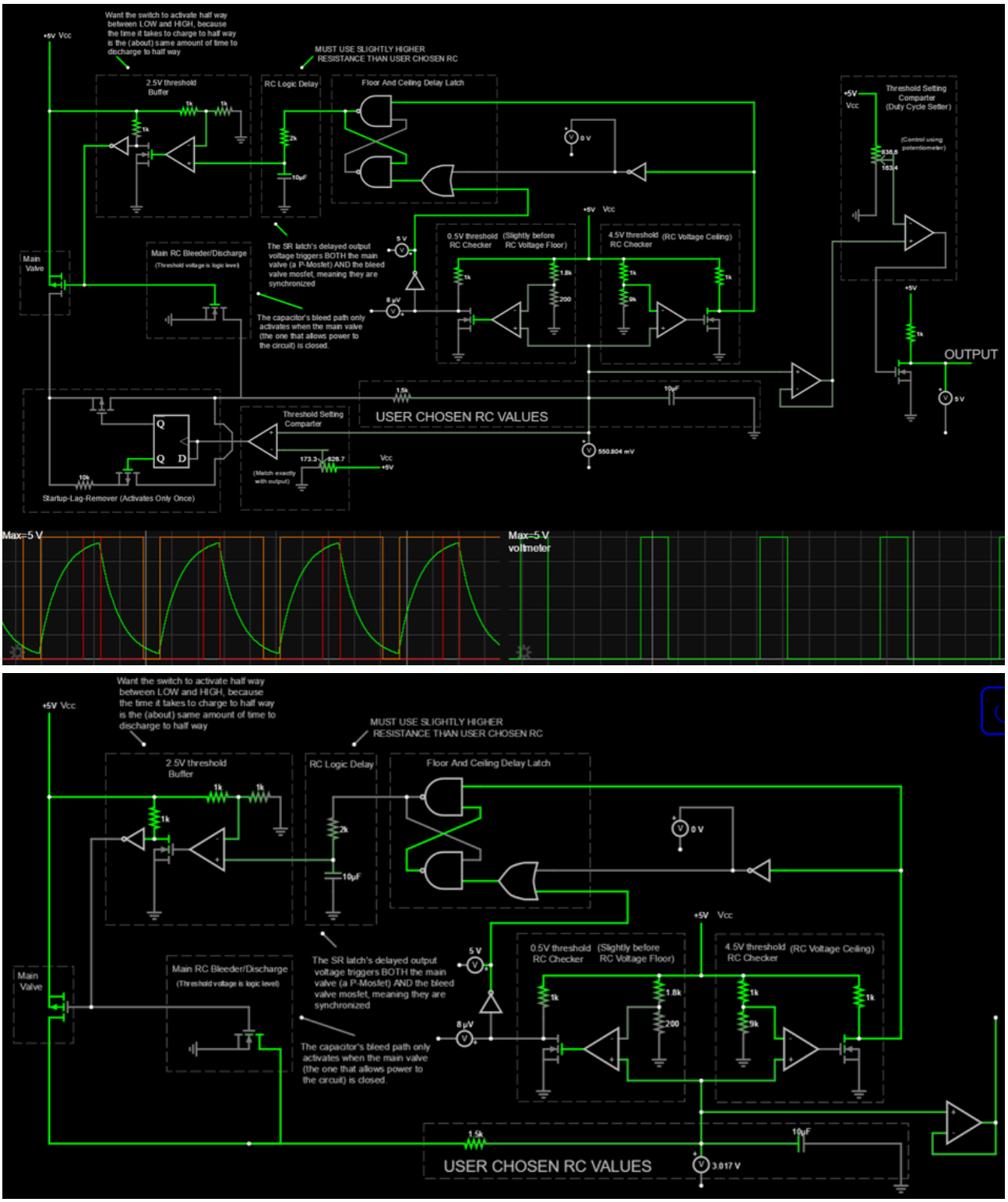
```
int ccwOnOrOff_Pin = 7;//Is actually ON when LOW, and OFF when HIGH
int cwOnOrOff_Pin = 8;//Is actually ON when LOW, and OFF when HIGH
int ccwPWM_Pin = 6;
int cwPWM_Pin = 5;
int ccwIncrement_Pin = 12;
int cwIncrement_Pin = 13;
int increments = 5; // 5 increments going one way, 5 going the other, and a middle for 0, in total, 11 levels.
int level = 0;
int ccwIncrement_AllowPress = 1;//1 is allow press
int cwIncrement_AllowPress = 1;//1 is allow press
int minPWM = 75;

void setup() {
  Serial.begin(9600);
  pinMode(ccwOnOrOff_Pin, OUTPUT);
  pinMode(cwOnOrOff_Pin, OUTPUT);
  pinMode(ccwIncrement_Pin, INPUT);
  pinMode(cwIncrement_Pin, INPUT);
}

void loop() {
  digitalWrite(ccwOnOrOff_Pin, 1 - level > 0);
  digitalWrite(cwOnOrOff_Pin, 1 - (level < 0));//original was (level > 0) * (abs(level) * 255/(increments) )
  analogWrite(ccwPWM_Pin, (level > 0) * (abs(level-(minPWM>0) ) * (255 - minPWM)/(increments-(minPWM>0)) + minPWM) );
  analogWrite(cwPWM_Pin, (level < 0) * (abs(level+(minPWM>0)) * (255 - minPWM)/(increments-(minPWM>0)) + minPWM) );
  //the (minPWM>0) = 1 if true, and 0 if false, so if you set a minPWM, it will start at that. Otherwise, starts at first increment

  Serial.println((level > 0) * (abs(level-(minPWM>0) ) * (255 - minPWM)/(increments-(minPWM>0)) + minPWM));
  Serial.println((level < 0) * (abs(level+(minPWM>0)) * (255 - minPWM)/(increments-(minPWM>0)) + minPWM));
  Serial.print("Level: ");
  Serial.println(level);

  if(digitalRead(ccwIncrement_Pin) == HIGH && ccwIncrement_AllowPress == 0){
    ccwIncrement_AllowPress = 1;
  }
  if(digitalRead(ccwIncrement_Pin) == LOW && ccwIncrement_AllowPress == 1 && level < increments){
    level = level + 1;
    ccwIncrement_AllowPress = 0;
  }
  if(digitalRead(cwIncrement_Pin) == HIGH && cwIncrement_AllowPress == 0){
    cwIncrement_AllowPress = 1;
  }
  if(digitalRead(cwIncrement_Pin) == LOW && cwIncrement_AllowPress == 1 && level > -1 * increments){
    level = level - 1;
    cwIncrement_AllowPress = 0;
  }
}
```

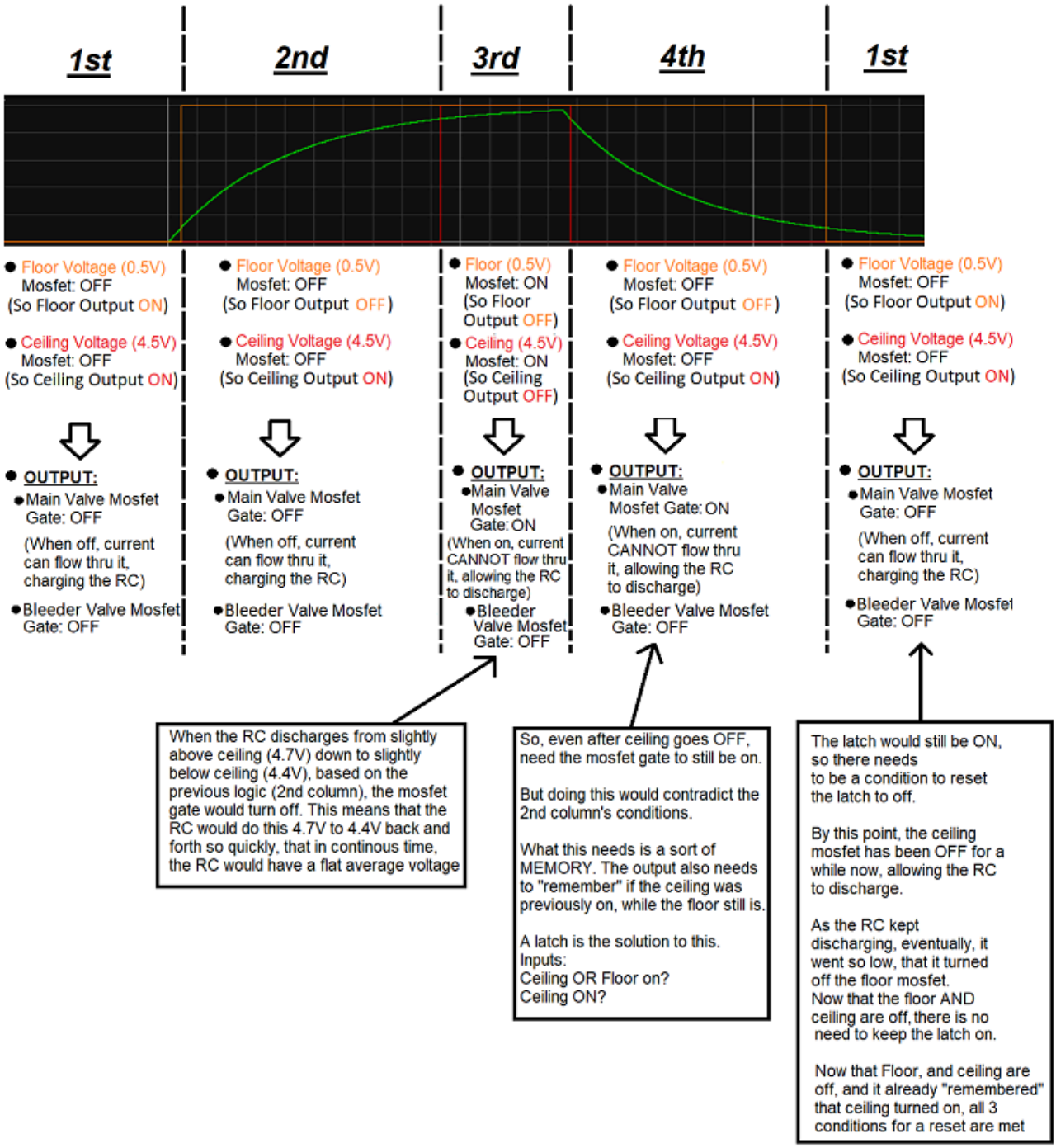






This was borne out of trying to build several kinds of oscillators. Thinking that this problem would be simple, it ended up blowing up, and helped me understand why square wave generators work without dampening out.

The heart of this circuit is an RC circuit that turns itself on and off again based on the level of voltage at the measured node. Starting off, the capacitor would charge from 0V up until 5V. If the voltage reached high enough (4.5V), it would shut off the voltage supplied to itself. This detection uses comparters for two reasons: Firstly, they do not draw any current away from the RC circuit, thus keeping the math the same as any normal RC circuit. Secondly, it allows for easy threshold setting. For example, the right comparter would remain at 0V output until the RC circuit hit that desired 4.5V. The left one is off until the RC circuit dips below 0.5V. Combined together, these control the on and off time.

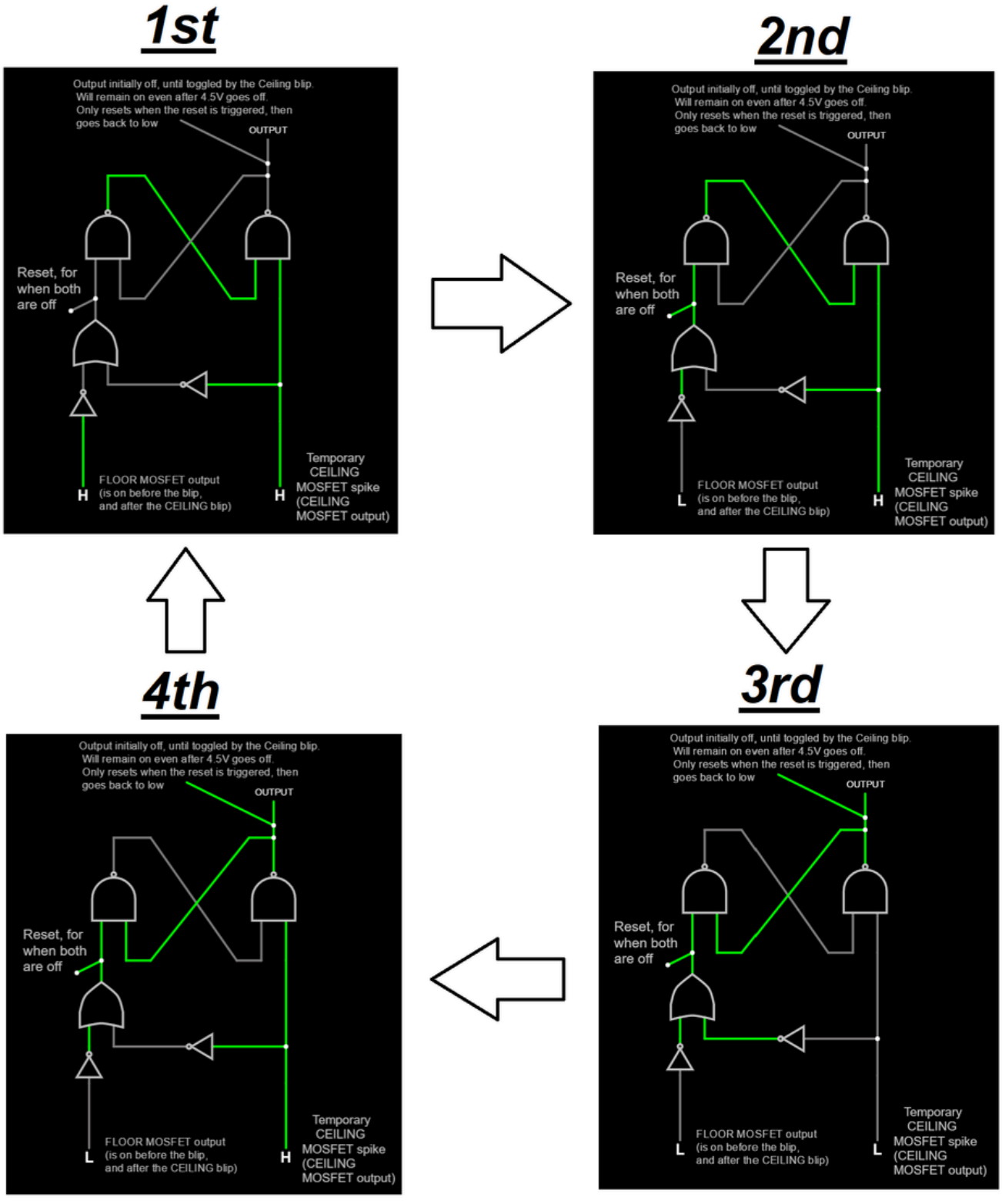



This problem stumped me for a while, because the states seemed contradictory. The “Main Valve” should be on when the timer starts. Ideally, the capacitor is at 0V when starting. This means that the Main valve should remain on. The RC circuit will shut off the main valve (and allow the capacitor to discharge by closing the “bleeder valve”) once the capacitor reaches 4.5V, at which point it starts discharging. When the capacitor discharges to a low enough voltage (0.5V), the main valve will be turned back on again. However, this raises a contradiction. When the RC circuit is discharging from 5V down to 0V, it will cross 4.5V and dip below, meaning the 4.5V comparter would shutoff again, making the condition for the main valve to open difficult, since it would switch on and off so fast around this area. This is why there needs to be some memory - the timer needs to remember what state it was in previously to turn on or off the main valve.

The main valve would start ON (it is on when the gate of the main valve is OFF), so the RC can charge up. In this state, all it needs to remember is that the 4.5V threshold wasn’t hit yet. When the 4.5V threshold is met, the main valve gate will turn ON, shutting OFF the main valve. Note, that there is a delay between the 4.5V threshold being reached, and the main valve being shut off. This is due to the memory mechanism - the SR latch. This is the price paid by allowing the mechanism to keep the main valve off as the capacitor dips below 4.5V, without immediately turning it back on again. Due to this latch, the circuit can “remember” that it had already hit the threshold of 4.5V, and that it is not allowed to turn the main valve off. This memory is reset when the capacitor dips below 0.5V, which then allows the main valve to open again (notice the delay again).

The diagram below describes the state of the latch, as it corresponds with each stage in the timer's cycle. The "FLOOR MOSFET" is set LOW when the capacitor reaches 0.5V (using the comparter). The "CEILING MOSFET" is set LOW when the capacitor reaches 4.5V (using the other comparter). The MOSFETs are set HIGH when their voltage conditions aren't met. The OUTPUT node is what determines if the main valve is open or closed, and if the bleeder valve is closed or open.

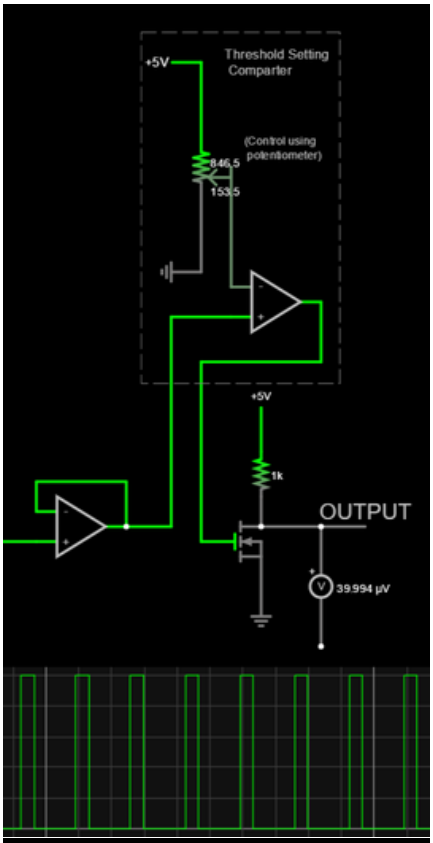
Initially, as the capacitor is at 0V, neither threshold is met, and so both inputs to the latch are HIGH. This means the output is OFF, and so the main valve is closed, and the bleeder valve is open. As soon as the capacitor hits 0.5V, the floor MOSFET's out put is set LOW. But this isn't enough to set the OUTPUT node to high. This only happens once the capacitor reaches 4.5V, at which point, the ceiling MOSFET's output is LOW, thus, finally turning the OUTPUT ON. This opens the main valve, and allows the capacitor to discharge through the closed bleeder valve. Then, the capacitor discharges below 4.5V, turning the ceiling MOSFET's output HIGH again. But this doesn't yet reset the latch. This only once the capacitor dip below 0.5V. This sets the OUTPUT OFF, thus, opening the bleeder valve, and closing the main valve, allowing the RC circuit to charge up, and the cycle repeats.

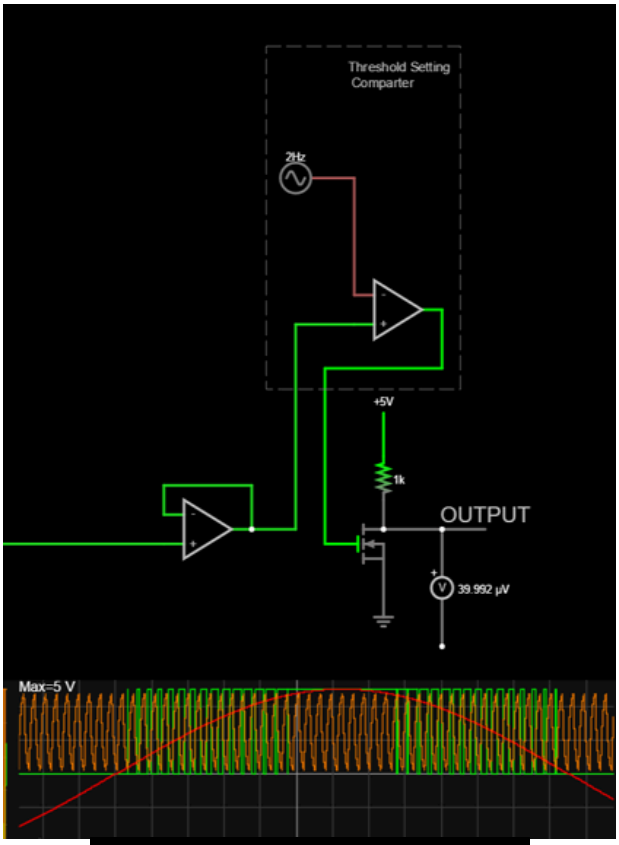


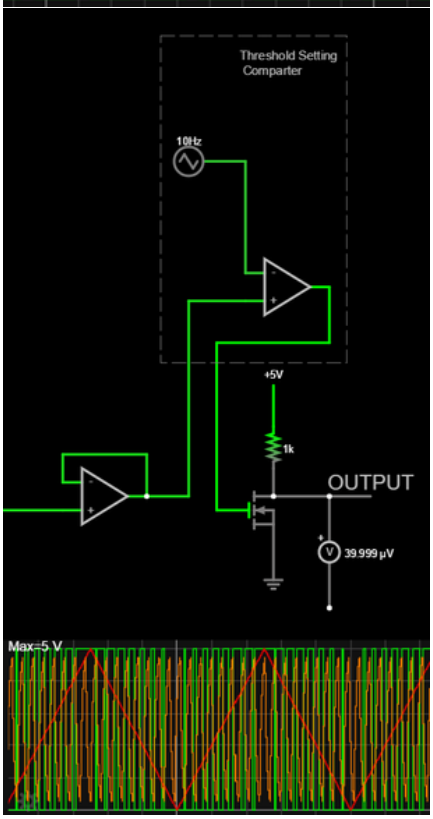
## RC SQUARE WAVE TIMER (W.I.P) — CONT

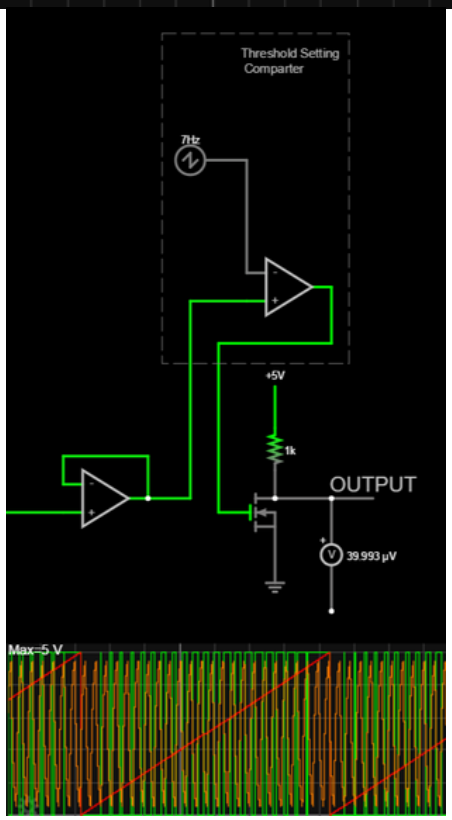
To determine the duty cycle, the actual voltage of the RC circuit is compared to some threshold. A follower op-amp is used to ensure that when sampling the voltage of the capacitor, it draws no current from it, thus, leaving the dynamics and math of the RC circuit to remain the same.

This voltage is compared to whatever input the user desires. For a square wave, the duty cycle can be set with a simple voltage divider. For a sine wave, the RC circuit needs to have a high frequency (not cutoff frequency) to accurately PWM the signal. This also goes for triangle and sawtooth waveforms.



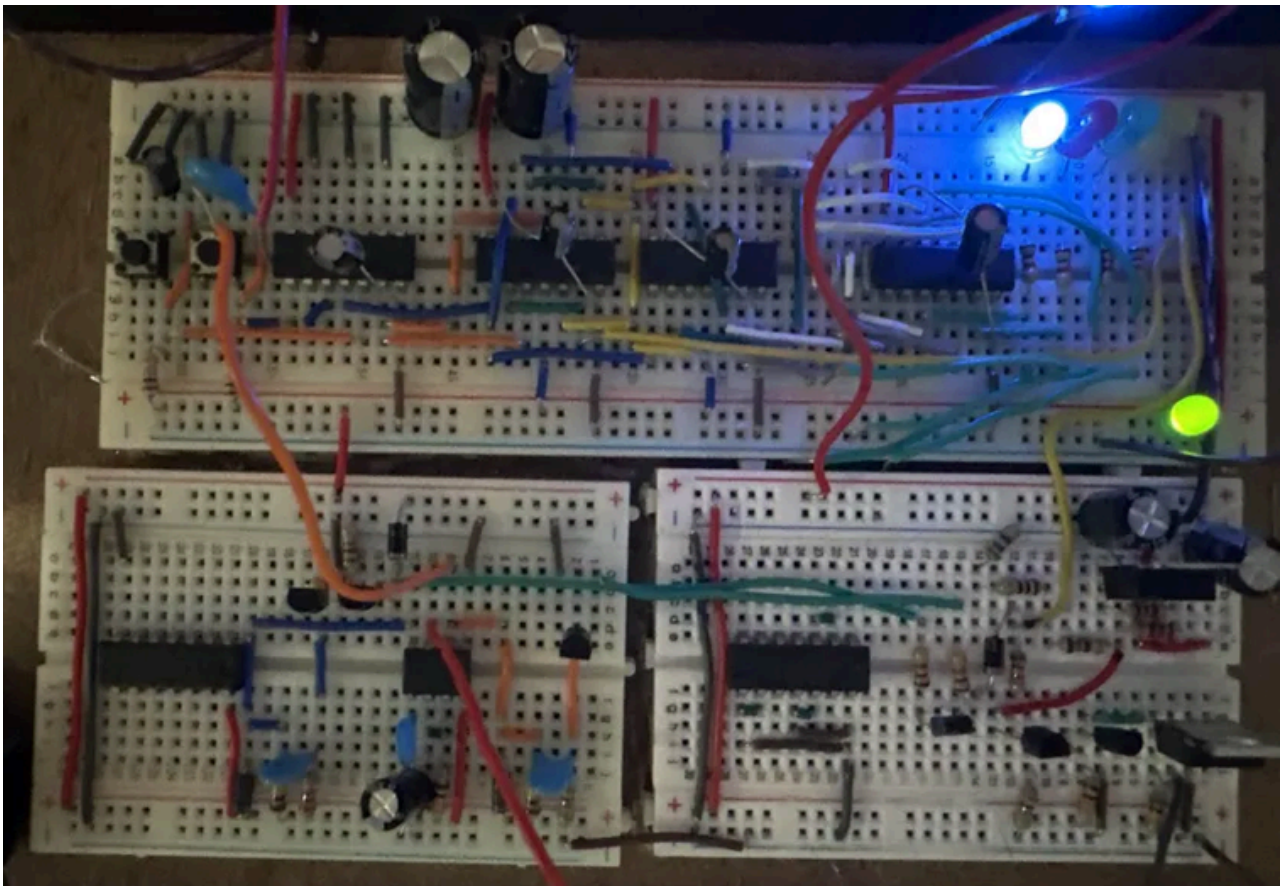



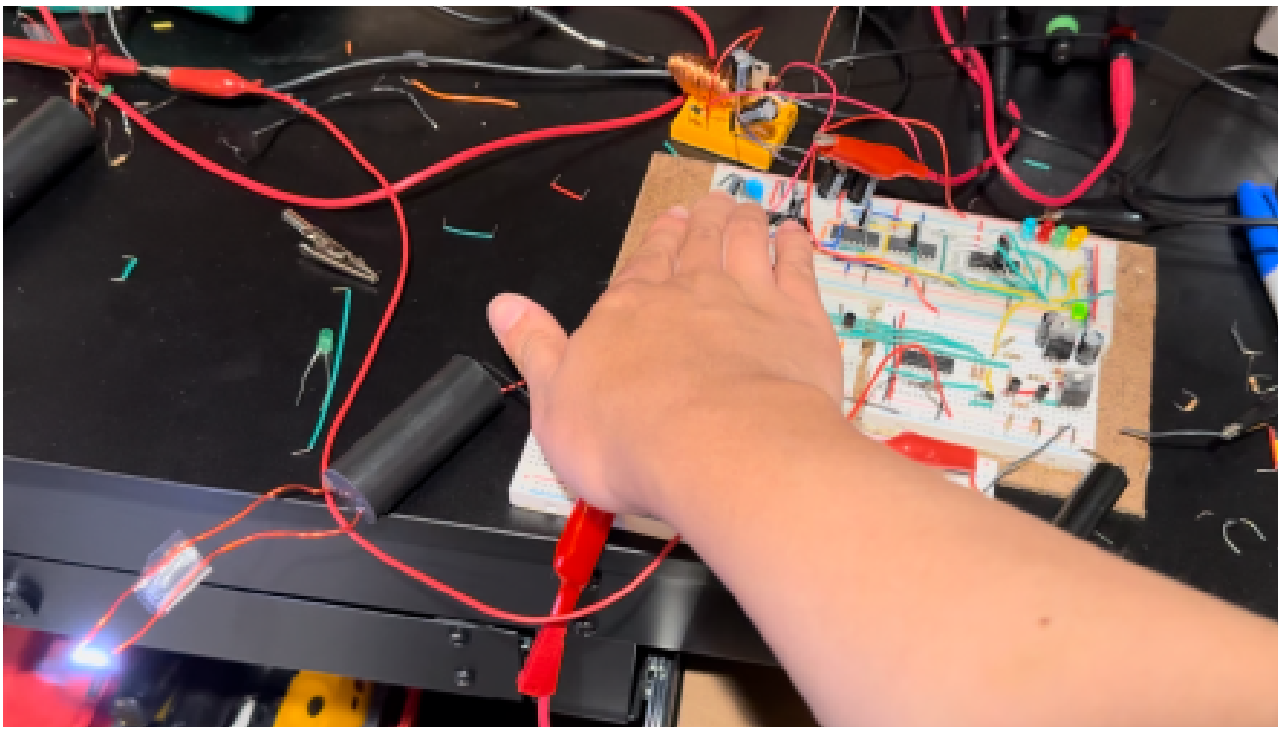
## MOCK LOGIC GATE IGNITION SYSTEM (W.I.P) — OCT 2022 - PRESENT

I wanted to recreate the distributor cap in old car ignition systems with a purely digital and electronic system. The only mechanical part would have been the shaft and some magnets to trigger a reed switch, which would control when the ignition system charged and discharged. The LEDs on the top right indicate which cylinder is to be fired. If the green LED at the bottom is on, it means that the coil has discharged, igniting the spark.



To make testing easier, the ignitoin coil was replaced with a cheap high voltage spark generator from amazon. It is more akin to a tazer than an actual ignition coil, since it continously gives off sparks, instead of one. However, they work on the same principle of cutting off the voltage across one coil of a transformer, inducing a massive voltage on the other side.

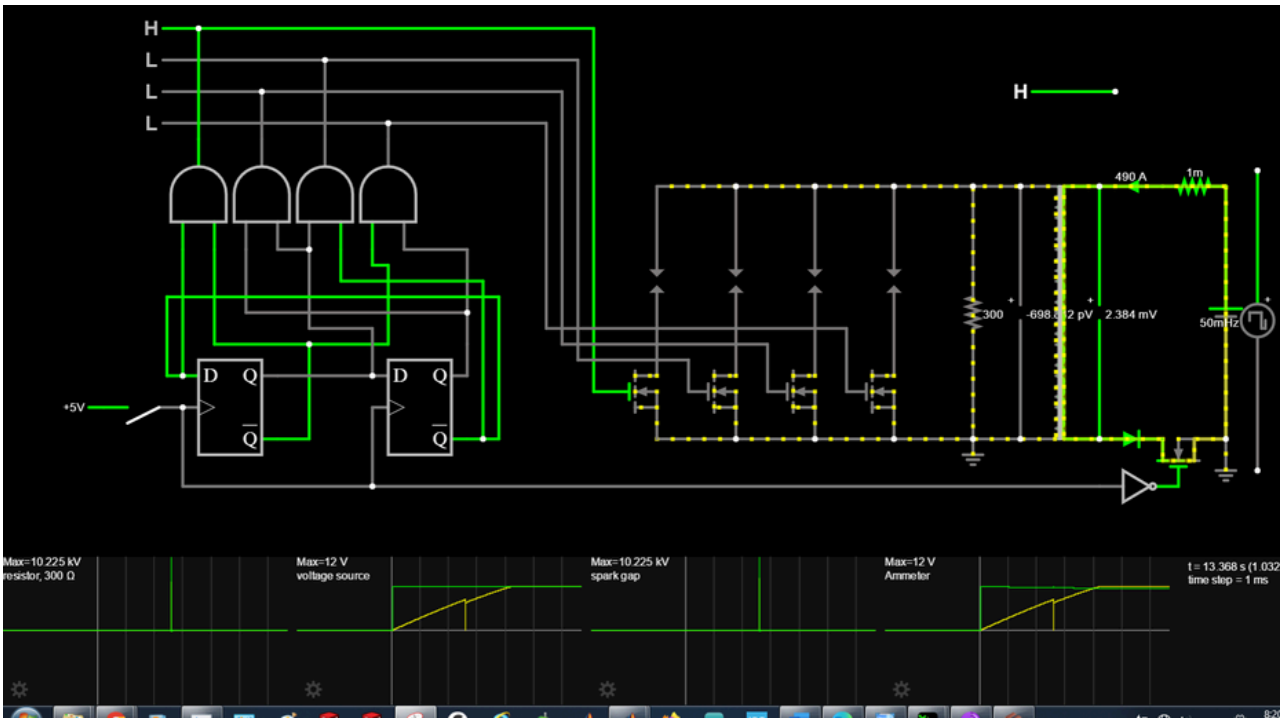
The reed switch was replaced with a simple push button that was debounced using an RC circuit and schmitt trigger.



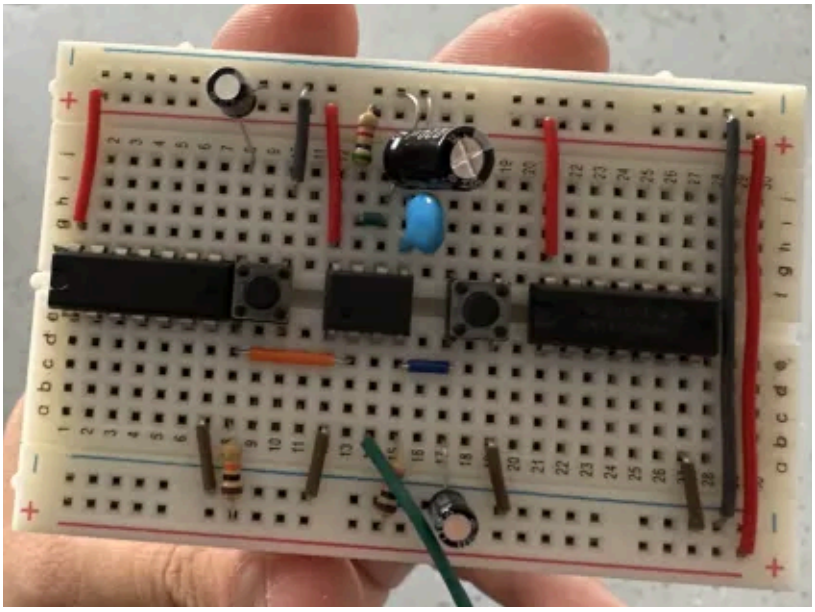
## MOCK LOGIC GATE IGNITION SYSTEM (W.I.P) — CONT

The distributor cap is replaced with two parts: a Johnson counter, and IGBTs. There is only one ignition coil, so the Johnson counter decides which cylinder to fire. This counter is toggled by a reed switch.

The first stage consists of letting the battery run current through the transformer. At the same time, the IGBT of the first cylinder is turned on. Next, after the reed switch is toggled again, the Johnson counter still stays on the first cylinder, but it then abruptly breaks the circuit of the battery and transformer. This induces a huge voltage across the other side of the transformer, and thus, the spark plugs. The IGBTs determine which spark plugs can actually discharge. Then reed switch is toggled again, and now, the next IGBT is turned on, and the battery is once again connected to the transformer.




I had issues during testing, particularly when the leads between the spark generator were too far apart. This wouldn't allow the coil to discharge, and so the voltage supplied to the transformer by the battery or power supply would essentially be feeding DC through an inductor. This eventually burnt out my power supply, since it would max out at 4.6A, or greater. This problem would have still occurred if I had used a regular ignition coil - since this could still occur if the Johnson counter's toggle to discharge the coil is triggered too late



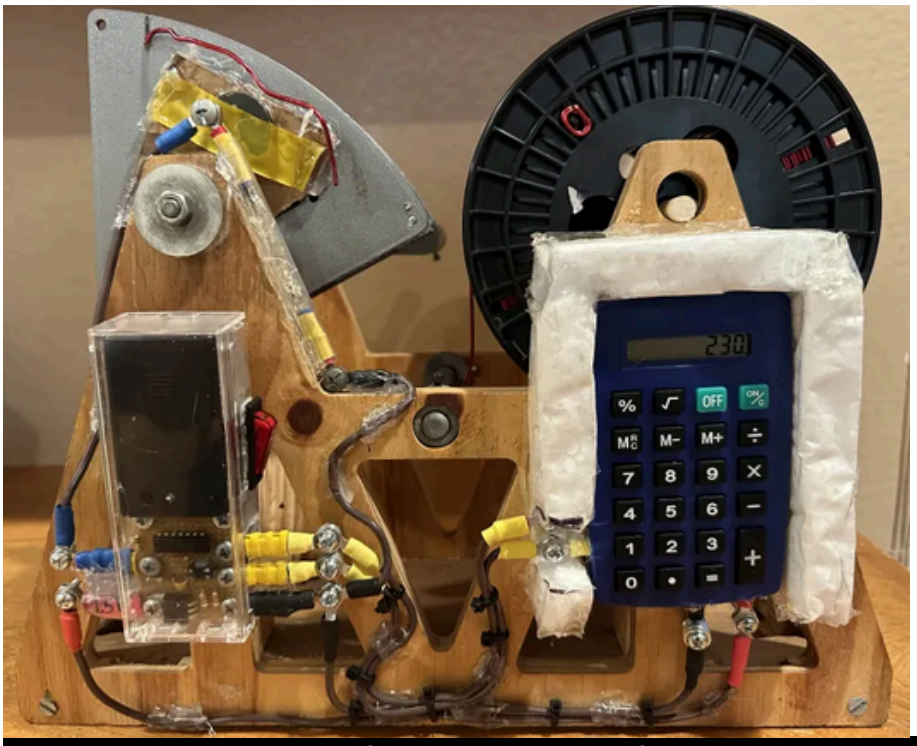
To solve this, I used a 555 timer in monostable mode, along with AND gates. When the battery is connected to the transformer, the 555 timer would start high, and after a certain length of time, say 0.5s, it would stay low until reset. But this timer should only start when the transformer needs to have current running through it, not on the disconnect. This is where the AND gate comes in. It takes the state of the Johnson counter, and decides whether or not to trigger the start on the 555 timer This start is triggered by a reset pulse from the AND gate that is connected to an RC circuit.

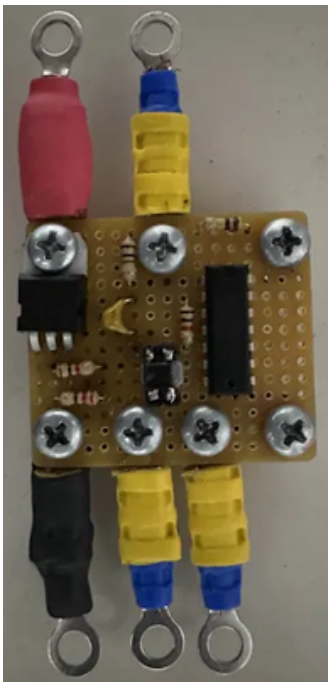
## IMPROVED COIL COUNTING CIRCUIT — JAN 2025

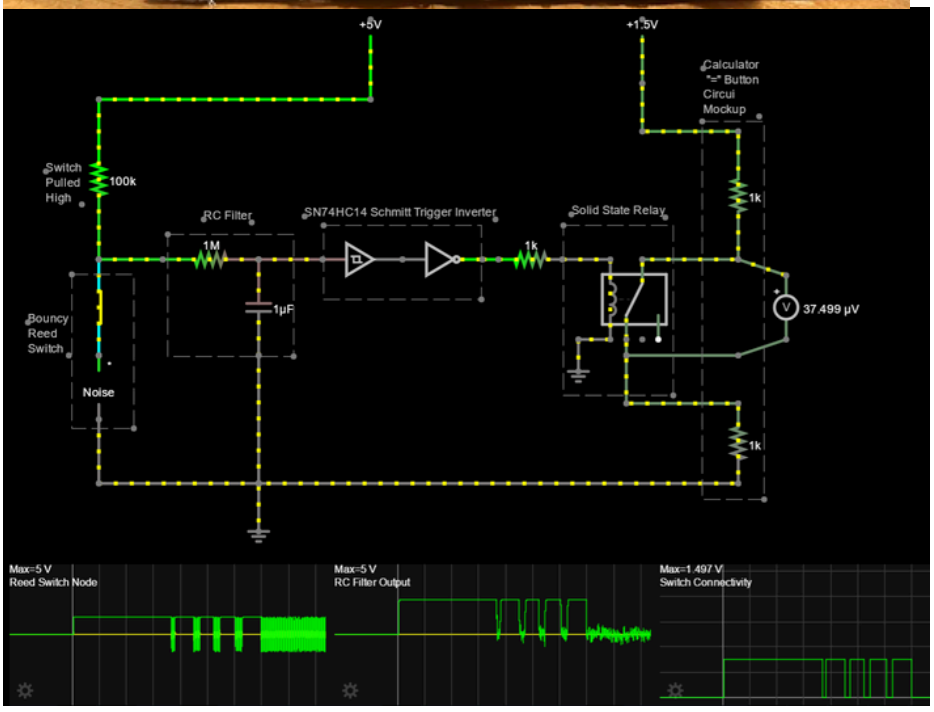
In 2019, I was interested in winding coils for a generator, and I wanted to know how many turns of wire per coil. If you press “+” then “1” on a calculator, and you keep hitting the “=” button after that, the calculator increments the count by 1 each time you press “=”. Initially, all I did was attach the “=” button leads to a reed switch, so that when a magnet connected to she spool/shaft passed by it, it would complete the circuit, and “press the button”, incrementing the count by 1. However, this was very bouncy.

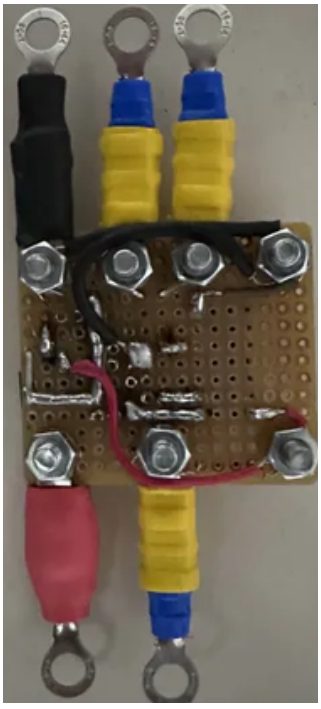
In early 2025, I finally understood why my previous debouncing efforts would fail, and I decided to tackle the problem again. The issue is that the leads of the “=” button are connected to unknown nodes on a calculator, so all that needs to be done is for the leads to be shorted, but without the bounciness. An RC circuit and Schmitt trigger weren’t enough, as I previously thought. In previous attempts, I would just connect the high side of the RC circuit to one lead of the “=” and ground on the other.

This time, I used the RC circuit combined with a Schmitt trigger to turn on and off a solid state relay. The Schmitt trigger filtered RC output would produce a clean, stable signal, while the SSR simply connected the leads.

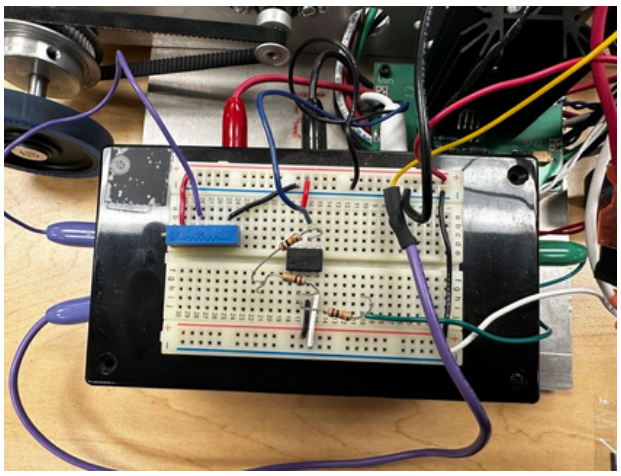






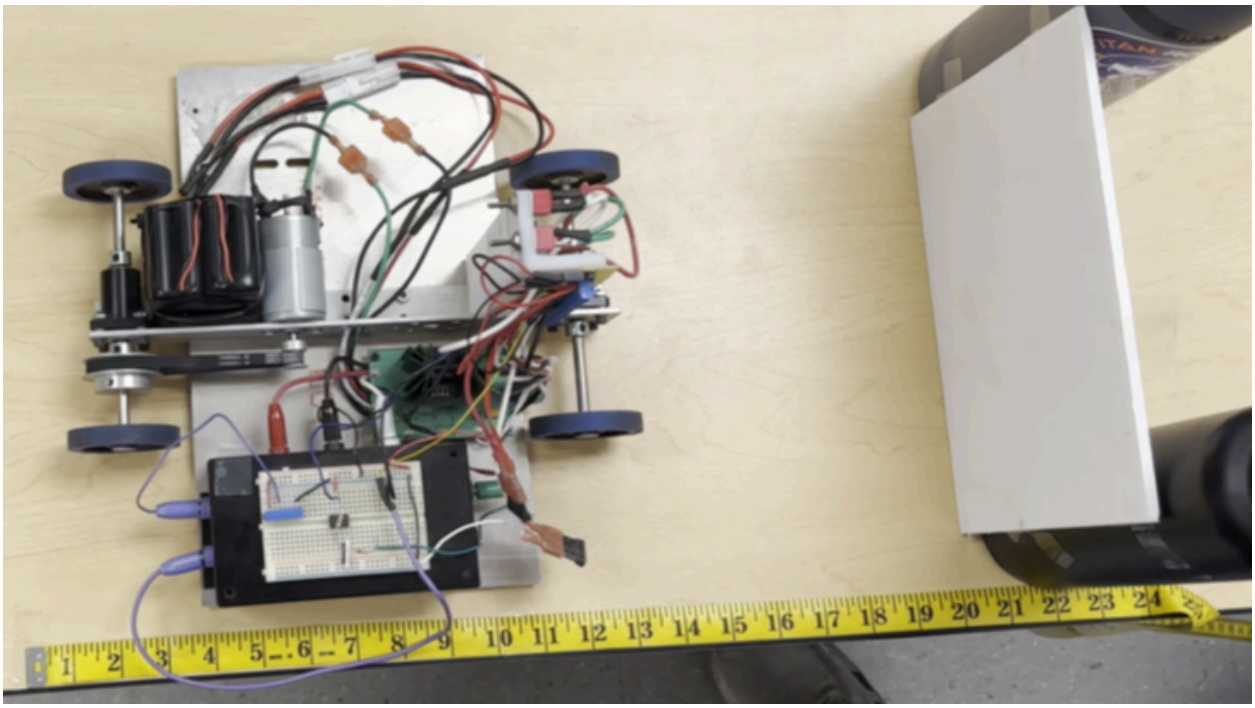



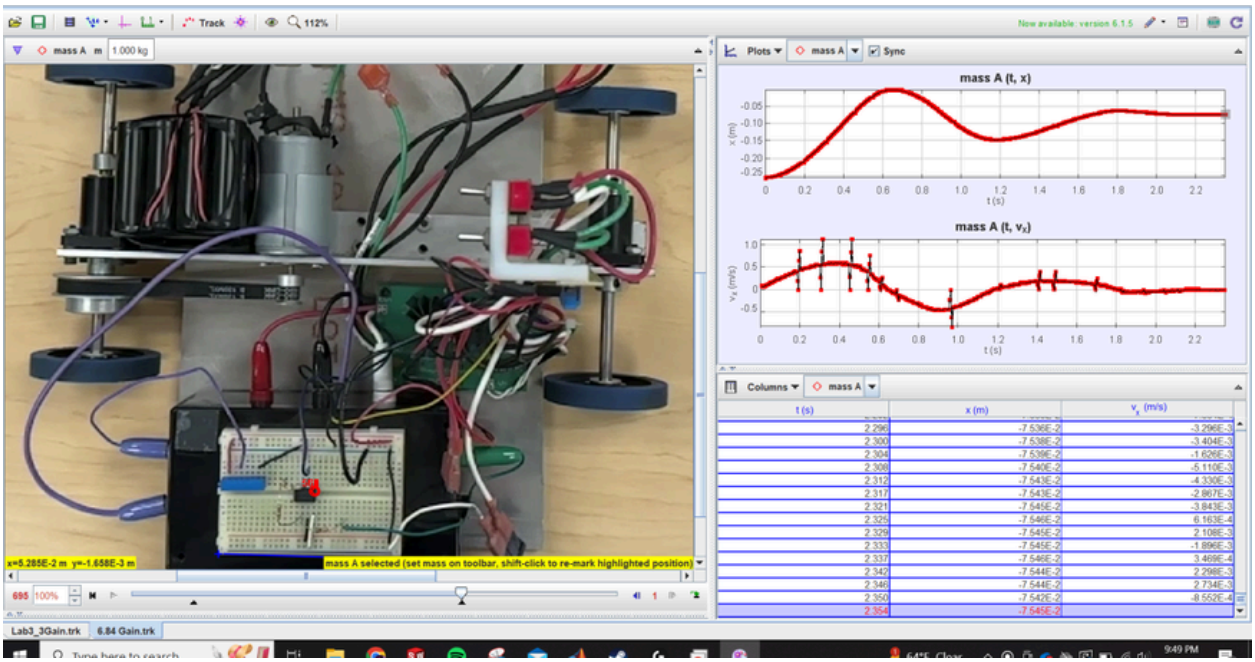
This was the main assignment in my Dynamics and Controls Systems lab. We were tasked with developing a PID Op-Amp controller to control the motor on a cart (shown to the right). We needed to get the cart to stop a certain distance from a white panel (as shown below), and we needed to use resistors and capacitors in a way (which controlled the P and D terms) so that an optimal response was reached: the car would be responsive enough (rise time) without having too much overshoot, and it would settle at an appropriate time to have the shortest possible total response time.



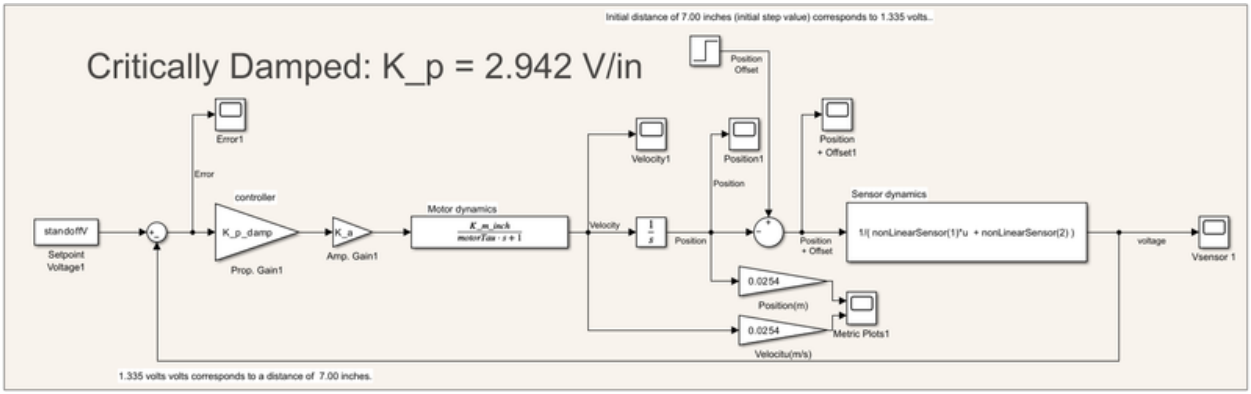
The cart gauged distance using an infrared sensor, and the motor's supply voltage was controlled using an Op-Amp, before being fed into the power electronics.

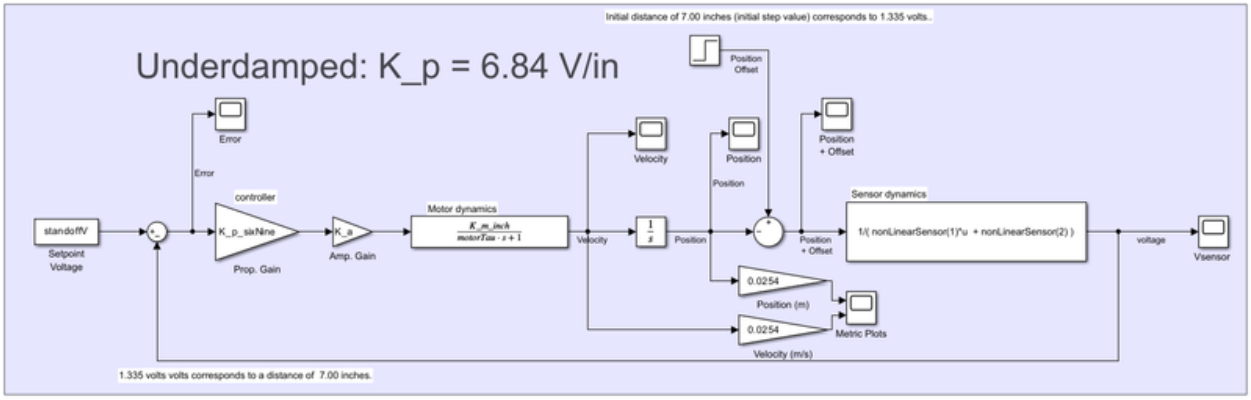
We tracked the cart's position in real time, and ran data analysis on it. We also used Simulink to tune the PD controller

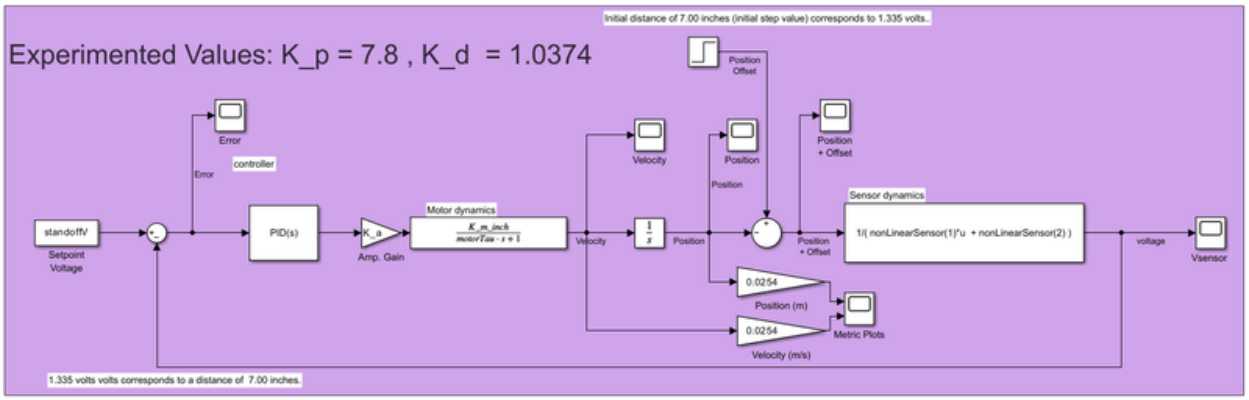


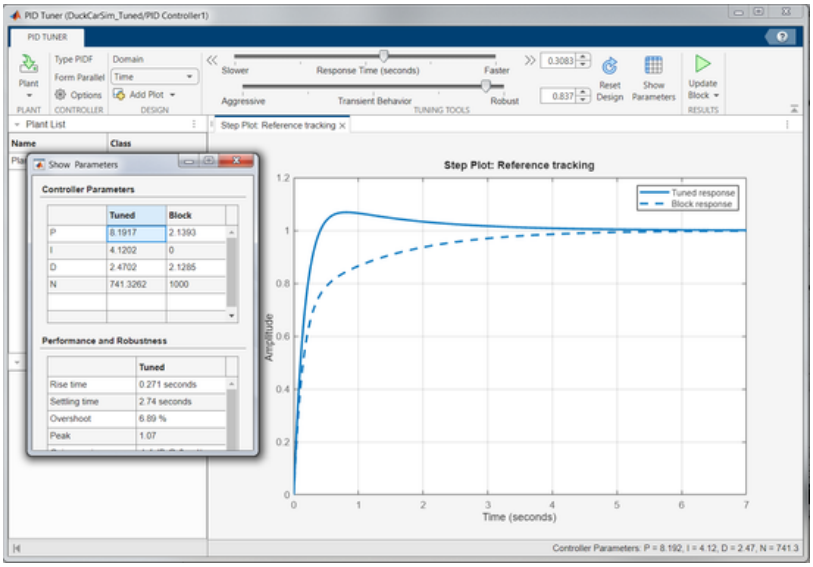


We were taught how to model electromechanical systems. We were able to neglect the electrical side of the motor, as its time constant was negligible in comparison to the motor's mechanical time constant. As shown, the motor is fed by the Op- Amp. Feeding into the op amp are the set point voltage, which is the voltage the IR sensor reads when it is at stand still from the panel (which is set via a potentiometer), and the actual read distance, which is outputted voltage from the IR sensor.




Tuning the PD controller was a matter of both using Simulink's PID Tuner, and experimenting in real life. Because the cart outputted distance, it had an integrator term, and therefore, was a type 1 system. This meant it had no steady state error, and integral control wasn't required.

So I had to manually set the I term to 0. I then used the P and D terms as launching points to determine the optimal P and D values, given the resistors and capacitors available. This was all very experimental.

Eventually, as shown on the next page, our system produced almost no overshoot, and settled very quickly because of it. It also had a fast rise time.

We then compared real life to our Simulink models. We calculated the damping coefficient and natural frequency both ways, using log-decrement, and also testing our linearized equations and plugging in constants. As seen, our Simulink model wasn't too close to real life. This may have been because the electrical side was non-negligible, especially the power electronics.

$$\zeta_{log-decrement} = -\frac{log(\frac{maxPercentOvershoot}{100})}{\sqrt{\pi^2+log(\frac{maxPercentOvershoot}{100})^2}} \qquad \zeta_{experimentally} = \frac{1}{2\tau\omega_n}$$

$$\omega_{n_{log-decrement}} = \frac{\pi}{(T)\sqrt{1-(\zeta_{log-decrement})^2}} \qquad \omega_{n_{experimentally}} = \sqrt{\frac{K_p K_a K_m K_s}{\tau}}$$

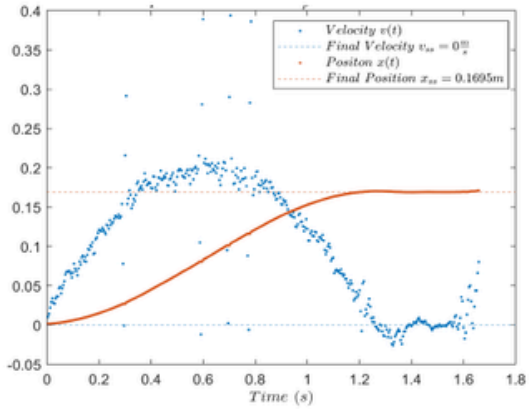



Figure 1: Actual Motion at $K_p = 2.942$

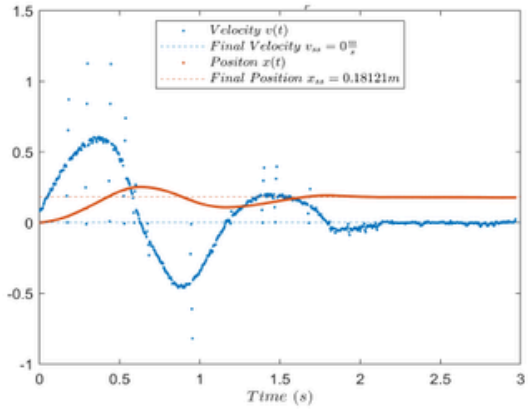



Figure 2: Actual Motion at $K_p = 6.84$

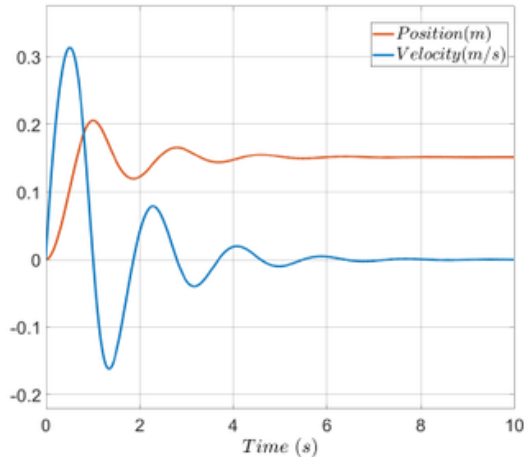



Figure 3: SIMULINK's Model at $K_p = 2.942$

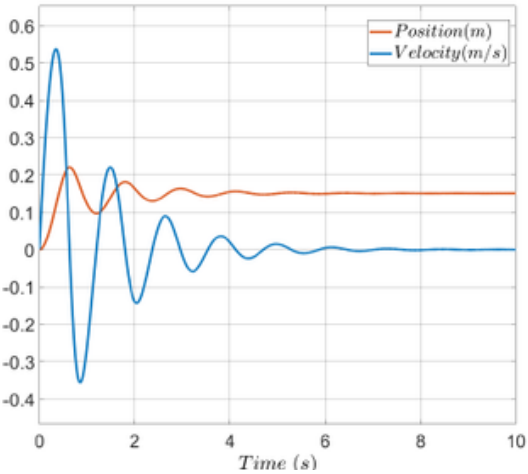



Figure 4: SIMULINK's Model at $K_p = 6.84$

My experimented and tuned PD values produced a very good result, even with the Simulink model being a little off

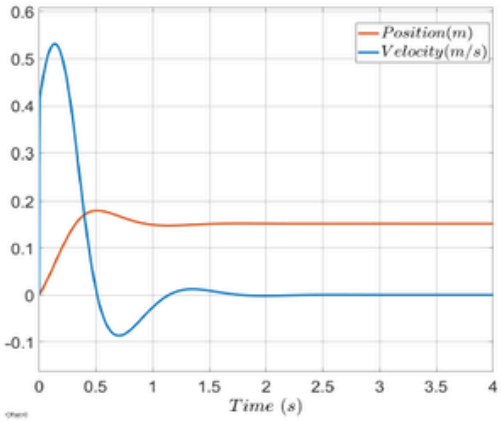



Figure 5: SIMULINK'S Model at $K_p = 7.8, K_d = 1.0374$

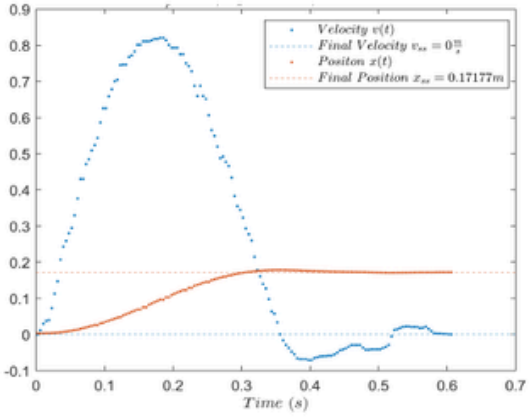
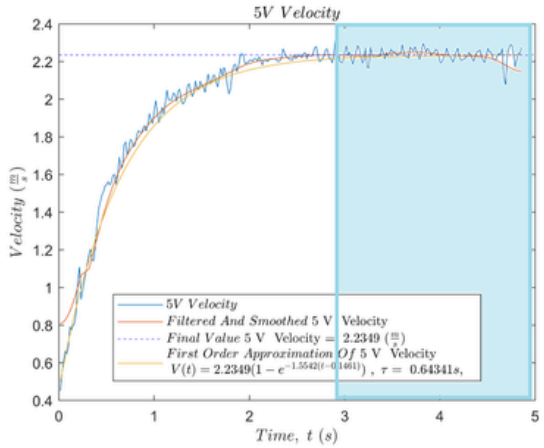


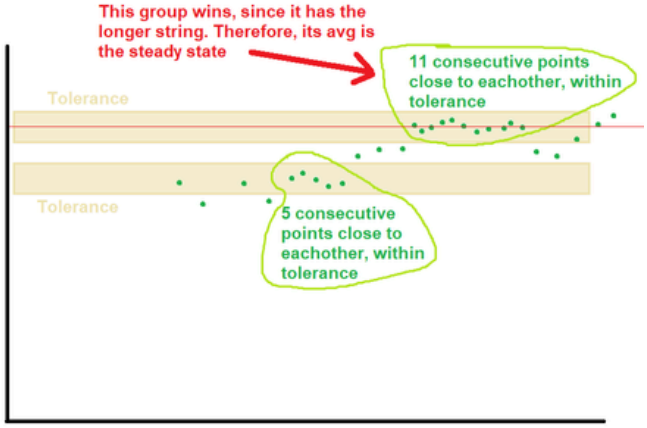

Figure 6: Actual Motion at $K_p = 7.8, K_d = 1.0374$

Because our tracking software could be a little buggy at times, it produced NaN positions for certain time points. This caused problems with the data, and the NaNs had to be removed

```
function [dataWithoutNaNs] = NaNRemover(dataInput)
    dataWithoutNaNs = dataInput; % can't pass by reference :(
    [a,~] = find(isnan(dataWithoutNaNs));
    a = unique(a(:).'); %removes any duplicate values, incase there is a NaN twice in one row
    for i = 1:length(a)
        dataWithoutNaNs(a(i),:) = [];
        a = a - 1;
    end
end
```

Manually finding the steady state position and velocity would have been subject to a lot of human bias, and calculation error. So that is why I had MATLAB find the steady state for me. Zooming in on a chosen window (usually the last third of the x axis), the program looks for a continuous string of values that are close to each other (they are given a tolerance). There could be many of these strings though, and so it looks for the longest consecutive string, and whatever those values are, they are averaged to output the steady state.




```
function [finalValue] = finalValueFinder(dataInputted,steadyStateTolerance,timeSection)
    %will only look at last set of time values for final value
    lastSectionOfTime = [floor( timeSection *height(dataInputted)),ceil( timeSection *height(dataInputted))];
    if lastSectionOfTime(1) == 0
        lastSectionOfTime(1) = 1;
    end
    a = diff(dataInputted(lastSectionOfTime(1):lastSectionOfTime(2),2))./diff(dataInputted(lastSectionOfTime(1):lastSectionOfTime(2),1));
    steadyestLastIndecies = find(abs(a)<=steadyStateTolerance) + lastSectionOfTime(1) - 1;
    %checks for any changes smaller than the tolerance, so that change is close to 0
    consecutiveSteadyIndecies = [;]; %  [ start1,start2 ; end1,end2 ; ... ]
    itt = 1;
    %will now check for the longest streak of consecutive indecies that fall within tolerance
    while itt <= length(steadyestLastIndecies)
        a = steadyestLastIndecies(itt);
        iitt = itt+1; %check for consecutive
        while iitt <= length(steadyestLastIndecies) && steadyestLastIndecies(iitt) - steadyestLastIndecies(itt) < 2
            iitt = iitt+1;
            itt = itt + 1;
        end
        b = steadyestLastIndecies(itt);
        consecutiveSteadyIndecies = [consecutiveSteadyIndecies ; [a,b]];
        itt = itt+1;
    end
    consecutiveSteadyIndecies;
    longestStreakPairs = find(max( consecutiveSteadyIndecies(:,2) - consecutiveSteadyIndecies(:,1) )); %might return multiple indecies if there is a tie
    longestStreakPairs = longestStreakPairs(end); %use the last longest steady streak
    finalValue = mean(dataInputted(consecutiveSteadyIndecies(longestStreakPairs,1):consecutiveSteadyIndecies(longestStreakPairs,1),2));
    %runs these indecies through the filtered function, and averages the values at these indecies
end
```

As seen, the data was also noise filtered to produce smoother curves, so that a more accurate equation of motion could be fit onto it.

```
function [filteredData] = dataNoiseRemover(dataIn,cutoffFreq)
    %Runs a low pass filter for each non-time column of the data, to filter
    %out nise
    dataIn = NaNRemover(dataIn);
    samplesCount = height(dataIn);
    filteredData = zeros(height(dataIn),width(dataIn));
    filteredData(:,1) = dataIn(:,1);%all time entries for position and velocity are the same
    for i = 2:width(dataIn)
        filteredData(:,i) = lowpass(dataIn(:,i),cutoffFreq,samplesCount);
    end
end
```