Duck Car Lab Final Report

Nathan De La Santos
Ian Mark Bautista
James Varghese
Cal State Fullerton, Department of Mechanical Engineering
EGME 476A Dynamic Systems and Controls Laboratory
November 10, 2023

Table of Contents

.

Abstract

The Ducky Car labs are an instrument for teaching control systems, offering a user-friendly platform to analyze and manipulate system responses through various input parameters. This report explores the impact of proportional control, specifically the proportional gain (Kp), on the Ducky Car's behavior. Additionally, we implement a Proportional-Derivative (PD) controller to enhance control characteristics such as settling time and percent overshoot. The experimental data is compared with simulations in SIMULINK, revealing disparities between idealized models and real-world responses. The report discusses sources of error, including sensor delays and environmental factors, highlighting the significance of considering these complexities in practical experiments. Our findings underscore the need for additional control elements like derivative control to achieve desired system behavior. The Ducky Car emerges as a valuable educational tool for comprehending control systems and their impact on system responses in real-world scenarios.

Introduction/Background

The Ducky Car labs are a useful demonstration of controls, with a user-friendly and customizable input system, designed to facilitate analyses of system responses through multiple varying inputs. This platform enables students to manipulate a diverse array of components, including resistors, capacitors, etc. thereby exerting direct control over pivotal system parameters such as gain, percent overshoot, and settling time. The tailoring of these components create a range of controllers which allow manipulation of key characteristics of the system like the Kp, Km, and Ka values. Understanding the effects of these values grants students the ability to influence fundamental aspects of system dynamics, shaping key metrics that characterize the response of the system like its rise time, settling time, percent overshoot, etc. These changing values defining the response of the system were once again manipulated within the multiple models created in SIMULINK modeling the Ducky Car. This synergy between a simplified input interface and the modeling capabilities offered by SIMULINK foster an enhanced insight into the effect of PD controllers and their ability to change a system's response.

Method

1. Duck Car Properties Data Collection

In preparation for Duck Car Lab 4 (The PD Controller Implementation), the following steps were performed:

1. Letting the Duck Car run freely at 3V, 4V, and 5V, and recording its velocity over time. The video was uploaded to and analyzed by a software called "Tracker"
    a. Using its steady state velocity at each applied voltage, the motor constant was found $K_m = \frac{Vel_{ss}}{V_a}$, and was set to the average between the three runs, $K_m = 0.38518 \frac{(m/s)}{V}$
    b. The time constant of the motor τ_m was found by taking the negative reciprocal of the slope of its dimensionless velocity $\phi(t) = ln\left(\frac{Vel_{ss} - Vel(t)}{Vel_{ss}}\right)$

Where τ_{motor} was set to the average of the three runs, $\tau_{motor} = 0.6459\ s$. Note, only the most "linear part" of the data was taken into consideration.

i. Note that another method to find the motor time constant was used, in which the reciprocal of the "initial slope" of the data was taken. However, because of the nature of the "initial" slope, only a few points in the beginning of the data could be used, which was noisy. This produced inconsistent results, and therefore, it was decided to stick with dimensionless velocity, which was more accurate because the slope fit of the data included more points to work with.

2. Sensor Voltage Data
   a. A white card was placed in front of the duck car's infrared sensor from 3 to 11 inches at increments of 0.5inches, and the voltage reading at the output of the sensor was recorded and graphed
   b. Linearized approximations, K_s of the sensor gain was taken at 7 inches, which was later set as the "standoff" distance, where $K_s = 0.15252\ \frac{V}{in}$

2. Proportional Control Metrics

To set the metrics to beat in Duck Car Lab 4 (The PD Controller Implementation), the following steps were performed:

1. The duck car was now given proportional control using an op-amp, in which different gains, K_p were set by using resistors only.
   a. Two gain values were tested: Where the system seemed critically damped, $K_p = 2.942$, and one where the system was underdamped, $K_p = 6.84$
2. In each test, a white card was first placed at 7 inches from the duck car, in which it was in "standoff" (not moving). This card was then removed to reveal another white card 13 inches away from the duck car, in which it tried to settle 7 inches away from the final card (6 inches from its starting position).
3. A video was recorded in slow motion for each test, in which they were uploaded and analyzed by Tracker.
   a. When $K_p = 6.84$, the rise time to react to the sudden change was quick, however, it resulted in sever overshoot, in which $PO = 39.5939\%$
   b. The dampened gain, $K_p = 2.942$ resulted in a slow rise, but in the best settling time, $T_s = 1.0333s$ as it did not overshoot.
   c. Because these were second order systems (where the first lab was approximated as a first order system), only the two percent settling time method was able to be used to find the settling time, and the time constant of the entire run.
   d. The two percent settling time was not used in the first lab, because when the car ran freely on rough outside terrain, causing a lot of noisy and wobbly data, unlike here, where the duck car's response was done over a short distance on a smooth table.

3. Proportional-Derivative Controller Implementation

The duck car was now equipped with Proportional-Derivative control using an op-amp, a proportional gain set by resistors, and a derivative gain set by capacitors and resistors. This new proportional control was able to change percent overshoot without changing settling time, as there were two different gains with independent parameters. The new duck car had to have a faster settling time than $T_s = 1.0333s$ (from the critically damped gain), while also having a lower percent overshoot than $PO = 39.5939\%$ (from the underdamped gain)

1. A simulink model of the duck car was used, in which a PID block was used to determine the appropriate K_p and K_d values. (K_i was set to 0, as there was no integral control)
2. Through trial and error in the actual lab however, it was found that $K_p = 7.8$ and $K_d = 1.0374$ had the best results, at $PO = 4.0592\%$ and $T_s = 0.3625s$
3. Tracker was used to analyze the recorded videos to come up with the new settling time and percent overshoot, as well as its natural frequency and damping value.
    a. Again, the two percent settling time method was used.

Through the experiment, multiple parameters which describe the characteristics of the system were found using the methods and equipment listed above. These insights provide analysis into the electronic components' effect on the car but also their effects within the Simulink model. Specifically from this lab, measurements and calculations were performed in order to return values for the system's dampening coefficient and natural frequency. Each of these parameters were found in two unique ways, one being the log-decrement method and one being extrapolated from the lab data. The log-decrement method was done in the MATLAB code and called for use of the Max Percent Overshoot variable calculated within the script, its use can be seen in the following equation

$$\zeta_{log-decrement} = -\frac{log(\frac{maxPercentOvershoot}{100})}{\sqrt{\Pi^2+log(\frac{maxPercentOvershoot}{100})^2}} \quad (1)$$

Where the value for maxPercentOvershoot was calculated via data from the tracking software in MATLAB. The log decrement equation for determining natural frequency was determined as follows

$$\omega_{n_{log-decrement}} = \frac{\pi}{(T)\sqrt{1-(\zeta_{log-decrement})^2}} \quad (2)$$

Where the values for damping (zeta) were also determined mathematically in MATLAB. T, the period, was replaced with rise time. In contrast, the other method of determining these values were performed using the following equations utilizing experimental values

$$\omega_{n_{experimentally}} = \sqrt{\frac{K_p K_a K_m K_s}{\tau}} \quad (3)$$

Where K_p represents gain from the controller's resistor values, K_a the default gain from the power circuitry of the system, Km represents the motor constant averaged out from the values gained in Lab 1, and Ks also represents a gain value. Finally, the damping coefficient was experimentally obtained via the equation

$$\zeta_{experimentally} = \frac{1}{2\tau\omega_n} \quad (4)$$

Where τ is the time constant and ω_n is the natural frequency found above. The garnering of these values provide students with parameters obtained independently of each other, allowing for an analysis of not only the validity of each method, but the validity of their experiment, the lab equipment used, and the procedure of which they adhered to. These values also provide key insights into the characteristics of the system and how each parameter may affect the response.

Results/Discussion

The modeling of the system within SIMULINK provides key insights into the interaction of each component within the Duck Car. This analysis allows students to manipulate and observe gain values, affecting key characteristics of the system. However, the theoretical model version does not always provide completely accurate results. In the plots below, the multiple responses of the system under different gain configurations is shown; these values vary for the system with one being set at K_p = 2.942, a critically damped response, and one being set to a value of K_p = 6.84. These gain configurations were modeled and analyzed through SIMULINK and then through experimental procedures; their responses were plotted above and demonstrate a clear difference between the experimental and modeled version in SIMULINK.

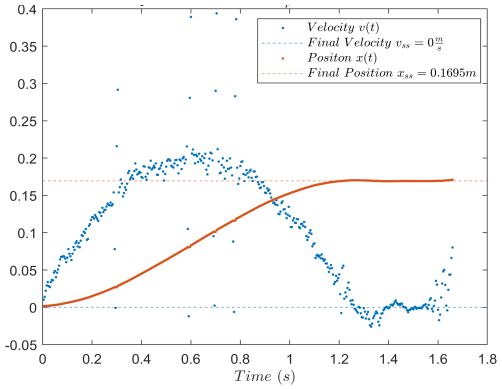


Figure 1: *Actual Motion at* $K_p = 2.942$

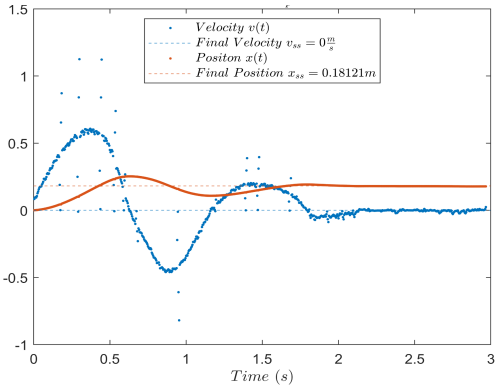


Figure 2: *Actual Motion at* $K_p = 6.84$

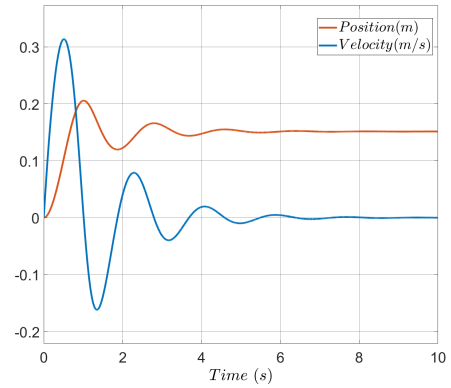


Figure 3: *SIMULINK's Model at* $K_p = 2.942$

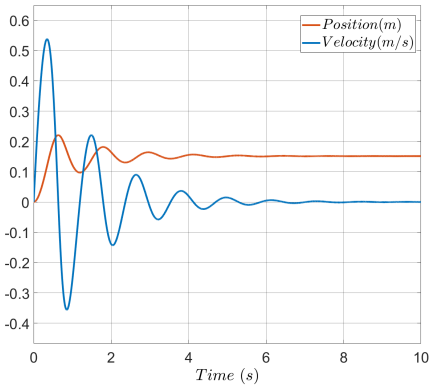


Figure 4: *SIMULINK's Model at* $K_p = 6.84$

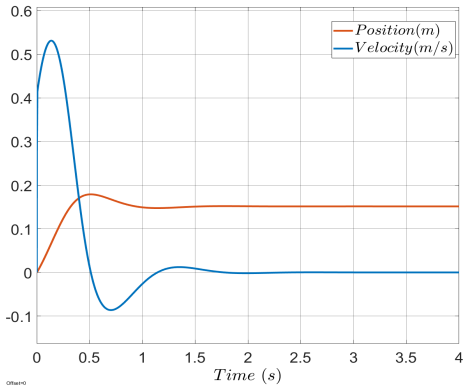


Figure 5: *SIMULINK'S Model at* $K_p = 7.8,\ K_d = 1.0374$

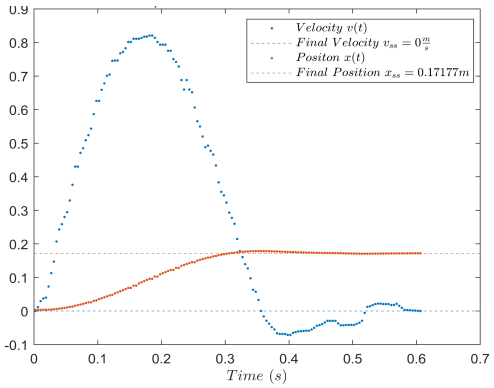


Figure 6: *Actual Motion at* $K_p = 7.8,\ K_d = 1.0374$

Because of internal and external friction, the later oscillations within the time range disappear and flatline. This happens for a multitude of reasons with one of them being the internal resistance that the motor experiences at those low voltages. This is most noticeable when analyzing figures 2 and 4 where the gain configuration of Kp = 6.84 is seen uniformly for both the SIMULINK and experimental run. When looking at the SIMULINK model, there are multiple oscillations within the time period (0 to 2 seconds) that the plot in real life did not exhibit, this is a perfect example of the unmodeled internal resistance of the motor. In SIMULINK, the gain of the vehicle is enough to cause multiple overshoots of the target distance, oscillating a total of four times within two seconds until finally reaching its position. This can be compared to figure 2 where in the actual motion, it is seen that the vehicle stops oscillating sooner due to the hindrance of internal friction, reaching its final position/velocity much quicker than the SIMULINK model. The effect of varying gain can be clearly seen through these four plots, the PD controller can be seen altering multiple key characteristics of the system response like its rise and settling time, but it doesn't give a complete picture of the analysis.

The implementation of the PD controller was a necessity in tailoring the response of the vehicle. In order to manipulate this controller correctly, multiple configurations of resistors and capacitors were used to garner the desired output and alter the time constant, rise time, and other characteristics. From these findings, calculations were made via experimental data to further the analysis and obtain values for metrics like the damping coefficient and natural frequency, these are tabulated below.

For $K_p = 6.84$	Log Decrement Method	Experimentally Obtained/Calculated
ζ (Damping Coefficient)	0.2829	0.183
ω_n (Natural Frequency)	5.2581 $\frac{rad}{s}$	5.7559 $\frac{rad}{s}$

Table 1: *Damping Coefficient and Natural Frequency through Log − Decrement and*

The results from the table return an expectedly similar outcome with the natural frequencies of the system returning a small percentage difference, this difference increases with the damping coefficient but can be explained through possible sources of error within the experiment or data. Since these values returned a somewhat similar response, the difference between the experimental data and SIMULINK model can be clearly seen but not significant enough to be deemed inaccurate, suggesting that both these methods for analysis of the response of the vehicle are valid and return a somewhat similar output. One important note to make about these values is the calculations and values obtained beforehand; some of these parameters were affected by values determined in Lab 1. From the three experiments, it was determined that $K_m = 0.38518$ (m/s)/V (the average Km between the three experiments). Also, determined from Lab 2 was the linearized Sensor Gain, K_s, at 7 inches where $K_s = 0.15252$ V/in. Finally, $K_a = 1.2$ by default for all duck cars. Note, there had to be conversions done in the calculations, since sensor gain was in V/in, while K_m was all metric.

Although the PD controller gives control over some parameters within the system, it's not perfect. Proportional control, represented by the proportional gain (Kp), can be limited in achieving the desired control behavior of reducing percent overshoot and settling time in control systems. This limitation arises from the inherent characteristics of proportional control. Firstly, proportional control has limited control authority. It adjusts the control input in proportion to the error signal, which is effective for quickly trying to bring the system to the desired output (setpoint), but lacks the capability to bring about rapid, finely tuned changes in the system's behavior. This limitation results in overshoot, where the system response surpasses the setpoint before settling, and it can lead to undesirable oscillations. Additionally, proportional control alone typically leads to slower settling times since it lacks the ability to introduce additional dynamics. The system's response relies solely on the proportional gain, possibly causing overcorrection, and will result in oscillations, since the proportional constant that the error is multiplied by will never really get it exactly to match the set point.This is because proportional control lacks inherent damping, making it challenging to reduce oscillations and attain critically damped or underdamped responses. To address these limitations, additional control elements, such as derivative control (D) in a PID (Proportional-Integral-Derivative) controller, are often introduced. Derivative control considers the rate of change of the error signal and helps reduce overshoot and accelerate settling time, thus improving the system's overall performance. In the context of our lab experiment, the introduction of the PD (Proportional-Derivative) controller is intended to address these issues by enhancing the control behavior and achieving better results. It This is where the rapid response fine tuning comes from.

Although the controllers can theoretically provide accessibility to changing a wide range of parameters in an accurate and predictable fashion, there will always be some sort of unpredictability within the experiment. The calculated gain for critical damping, as explained during the Lab 3, may not perform as expected in practical experiments due to real-world complexities. Factors like friction, nonlinearity, delays, and noise, not accounted for in the theoretical model, can influence results. Additionally, uncertainties in system parameters and environmental conditions can impact the effectiveness of the calculated gain, necessitating practical adjustments or control strategies to achieve the desired critical damping behavior.

Possible Sources of Error

Potential sources of error in our experiment must be carefully considered as they can substantially impact the reliability of our results. Sensor accuracy and calibration are paramount, as sensors can introduce inaccuracies due to their precision limitations or calibration issues. Sensor delays, meanwhile, can affect system stability by introducing phase shifts, resulting in response deviations. Non-ideal components like friction and nonlinearities, often overlooked in theoretical models, can introduce unexpected variations in system behavior.

Variability in system parameters, inherent in real systems, can lead to deviations from assumed values. External disturbances, such as temperature fluctuations or interference, can significantly affect system behavior. Lastly, hardware constraints, limiting control inputs due to factors like actuator precision or maximum voltages, need to be addressed. Recognizing these complexities is essential for understanding potential errors in our experimental data and optimizing our control strategies for practical applications.

Environmental factors, such as temperature variations, humidity levels, external vibrations, and dust presence, significantly influence the results in Lab 1. These factors impact equipment properties and the studied system, leading to variations in measurements. High humidity can affect sensor performance, while external vibrations introduce noise. Dust can also impact mechanical components like bearings and gears. Additionally, factors related to setup and data collection, including recording angles and road irregularities, must be carefully considered to ensure data accuracy and reliability.

Measurement errors in Lab 1 stem from various sources, including inaccuracies in sensor calibration, limitations in instrument precision, and human errors during data recording. Sensor noise and external interference, such as electromagnetic or vibrational disturbances, can further distort measurements. Uncertainties in reference objects, parallax errors, and inaccuracies in data processing also contribute to measurement discrepancies. Mitigating these specific error sources is vital to ensuring the reliability and precision of the experimental data collected in Lab 1.

The system under study in Lab 1 exhibits additional dynamics that may not be immediately apparent. One of these less obvious dynamics is the presence of a delay in the sensor's response. This delay is documented in the spec sheet for the sensor used in the experiment. The sensor's response time, or delay, can introduce a time lag between the actual physical changes in the system and the corresponding measurements recorded by the sensor.
This time delay can lead to discrepancies between the expected and observed behavior of the system. It is crucial to consider this sensor delay when interpreting the experimental results and accounting for any deviations between the measured data and the system's true dynamics. Understanding and accommodating for this sensor delay is essential for achieving accurate and reliable conclusions from the experimental data obtained in Lab 1.

Conclusion

In conclusion, our Ducky Car experiment serves as a dynamic learning platform for control systems, fostering a deeper understanding of their practical application. The investigation into the influence of proportional control, specifically the proportional gain (Kp), revealed that relying solely on Kp has limitations in achieving the desired control behavior. This limitation arises from the inherent characteristics of proportional control, which primarily adjusts control inputs proportionally to the error signal but may result in overshoot and slow settling times.

Our results, which encompass both real-world experimental data and SIMULINK simulations, showed significant disparities between the idealized models and practical responses. These differences emphasized the significance of acknowledging real-world complexities, such as friction, nonlinearity, sensor delays, and environmental factors.

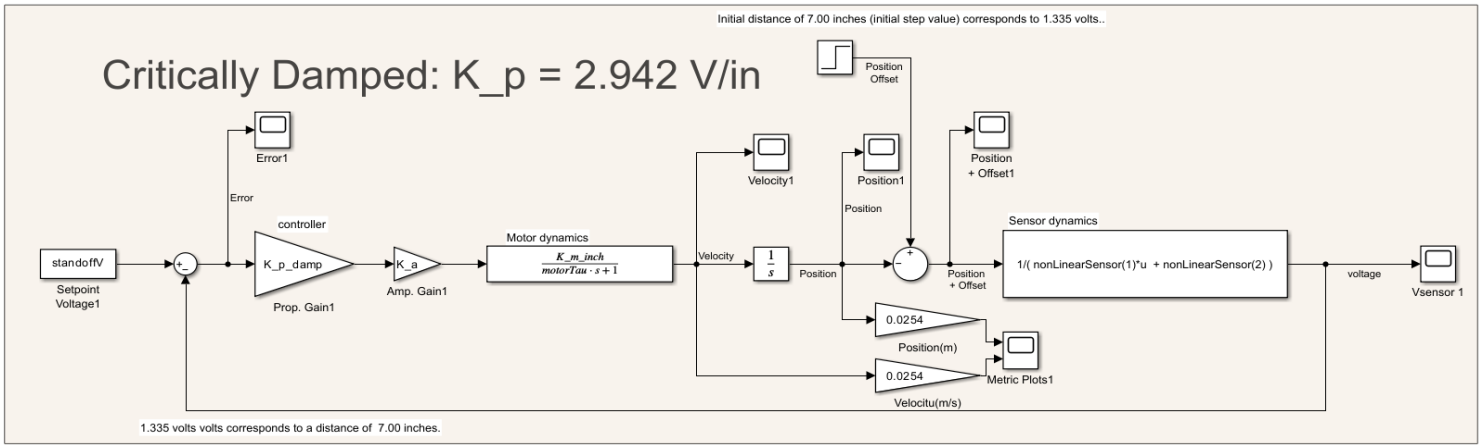
Furthermore, the introduction of derivative control through the PD controller proved valuable in mitigating some of the limitations of proportional control. Derivative control considers the rate of change of the error signal, thereby reducing overshoot and accelerating settling time, ultimately improving the system's overall performance.

These findings have essential implications for practical applications of control systems. Recognizing and addressing sources of error, including sensor delays and environmental influences, are critical for achieving the desired control behavior in real-world systems. The experiment demonstrated that the Ducky Car is an invaluable educational tool for bridging the gap between theoretical knowledge and hands-on experience, enriching the learning process in dynamic systems and controls.

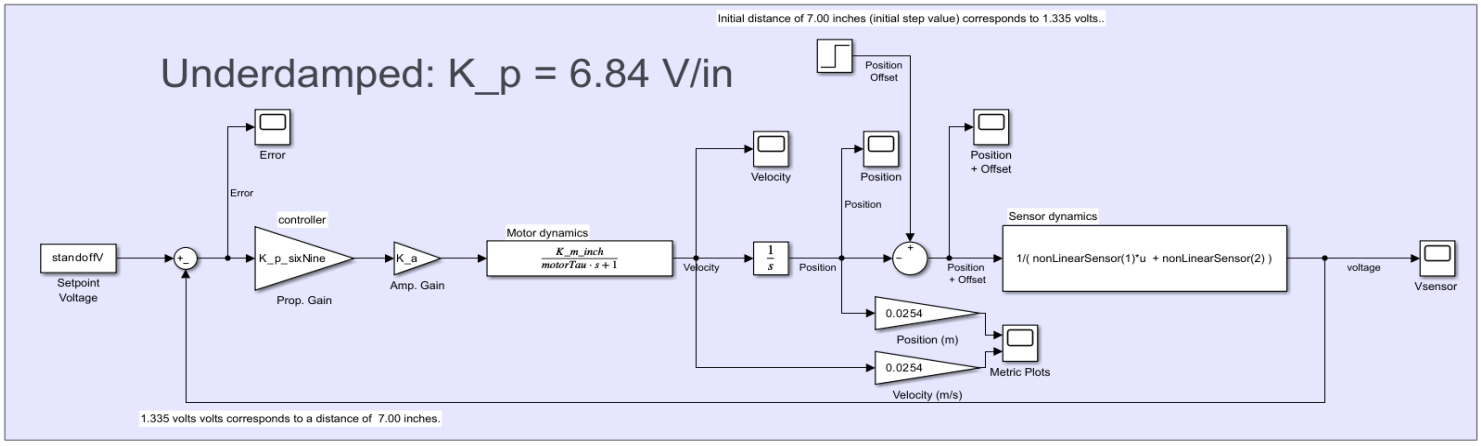
Appendix

SIMULINK Models

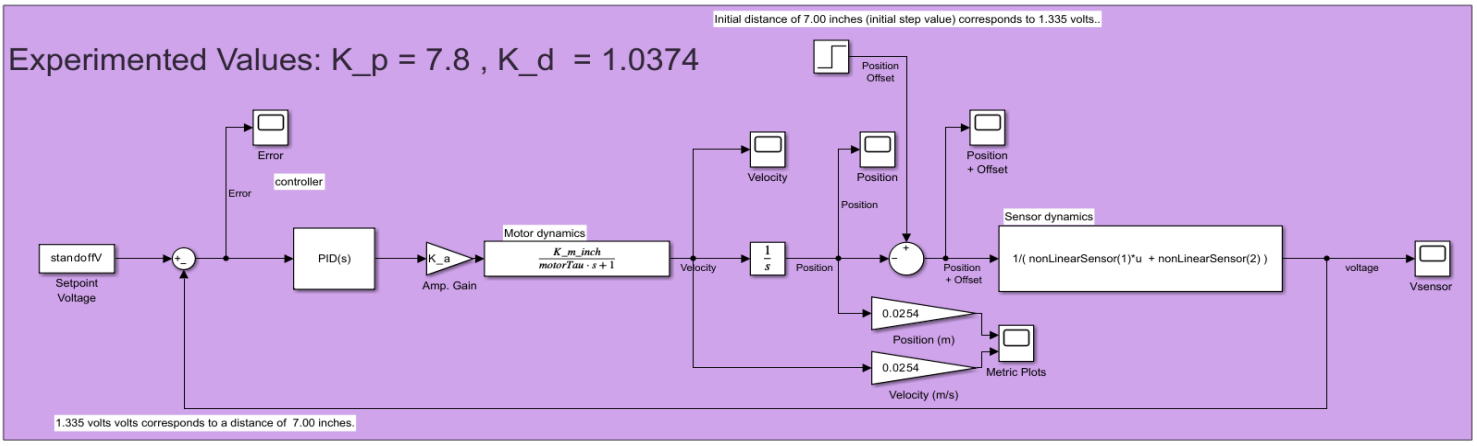
Proportional Control When K_p = 2.942



Proportional Control When K_p = 6.84



Tuned PD Values



MATLAB Code

Lab 1 Code: Finding Motor Constants And Motor Time Constants Through Steady State Velocity

```
% Duck Car Lab 1
% Ian Bautista, James Varghese, Nathan Delos Santos
clear;
clc;
close all;
ThreeV_PosData = readtable("3V_Ducky_Position.txt");
FourV_PosData = readtable("4v_Ducky_position.txt");
ThreeVData =
table2array([readtable("3V_Ducky_Velocity.txt"),ThreeV_PosData(:,2)]); %both
pos and vel have same timestamps
FourVData = table2array([readtable("4v_Ducky_velocity.txt"),
FourV_PosData(:,2) ]);
FiveVData = table2array(readtable("Duck Car5V.txt"));
clearvars ThreeV_PosData FourV_PosData; %clears them from memory
%only "filter" applied is removing NaNs
filteredThreeVData = NaNRemover(ThreeVData);
filteredFourVData = NaNRemover(FourVData);
filteredFiveVData = NaNRemover(FiveVData);
%first 4 data points of 4V is when stationary. Will remove them
filteredFourVData = filteredFourVData(4:end,:);
ThreeV_fvVel = finalValueFinder(filteredThreeVData,1*10^(-1),[2/3,1]); %
Smoothed Out Data, Tolerance, What sections of the graph you want
FourV_fvVel = finalValueFinder(filteredFourVData,1*10^(-1),[0.5,1]);
FiveV_fvVel = finalValueFinder(filteredFiveVData,1*10^(-1),[0.5,1]);
ThreeV_TimeConst =
timeConstantFinder(filteredThreeVData,ThreeV_fvVel,0.06,0.13,"$3 V \
Dimnensionless \ Velocity$","$Dimnensionless \ Velocity$","3V")
FourV_TimeConst =
timeConstantFinder(filteredFourVData,FourV_fvVel,0.075,0.475,"$4 V \
Dimnensionless \ Velocity$","$Dimnensionless \ Velocity$","4V")
FiveV_TimeConst =
timeConstantFinder(filteredFiveVData,FiveV_fvVel,0.075,0.45,"$5 V \
Dimnensionless \ Velocity$","$Dimnensionless \ Velocity$","5V")
[ThreeV_FirstOrderApprox,ThreeV_Shift] =
firstOrderApproximation(filteredThreeVData,filteredThreeVData,ThreeV_fvVel,Thr
eeV_TimeConst);
[FourV_FirstOrderApprox,FourV_Shift] =
firstOrderApproximation(filteredFourVData,filteredFourVData,FourV_fvVel,FourV_
TimeConst);
[FiveV_FirstOrderApprox,FiveV_Shift] =
firstOrderApproximation(filteredFiveVData,filteredFiveVData,FiveV_fvVel,FiveV_
TimeConst);
[~,ThreeV_initSlope_Tau,ThreeV_InitLine] =
initialSlopeTimeConstantFinder(filteredThreeVData,0,0.0125);
[~,FourV_initSlope_Tau,FourV_InitLine] =
initialSlopeTimeConstantFinder(filteredFourVData,0.01,0.09);
[~,FiveV_initSlope_Tau,FiveV_InitLine] =
initialSlopeTimeConstantFinder(filteredFiveVData,3/height(filteredFiveVData),8
/height(filteredFiveVData));
ThreeV_initSlope_Tau
FourV_initSlope_Tau
FiveV_initSlope_Tau
rawAndFilteredPlotter({filteredThreeVData,ThreeV_FirstOrderApprox,ThreeV_InitL
ine},[".","",'--'],"$3 V \ Velocity$","$Velocity \ ( \frac{m}{s}
```

```
)$",ThreeV_fvVel,ThreeV_TimeConst,ThreeV_Shift,"3V")
rawAndFilteredPlotter({filteredFourVData,FourV_FirstOrderApprox,FourV_InitLine
},[".","",'--'],"$4 V \ Velocity$","$Velocity \ ( \frac{m}{s}
)$",FourV_fvVel,FourV_TimeConst,FourV_Shift,"4V")
rawAndFilteredPlotter({filteredFiveVData,FiveV_FirstOrderApprox,FiveV_InitLine
},[".","",'--'],"$5 V \ Velocity$","$Velocity \ ( \frac{m}{s}
)$",FiveV_fvVel,FiveV_TimeConst,FiveV_Shift,"5V")
% %must ALWAYS skip first and last velocity entry, since are NaN, and so if
want to
% %plot it, also omit those entries for position
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
function [smoothedOutData] =
dataSmoother(dataIn,sandingPasses,movingAvgWindow)
   %Runs several moving averages on the noise-removed data in one call, to
   %further smooth it out.
   %think of the movingAvgWindow as the grit of sandpaper
   smoothedOutData = dataIn; %No way to pass by reference :(
   for i = 1:sandingPasses
       smoothedOutData = [smoothedOutData(:,1),movmean(smoothedOutData(:,2)...
           ,movingAvgWindow) , movmean(smoothedOutData(:,3),movingAvgWindow)];
   end
end
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
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(dataInpu
tted(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):consecutiveS
teadyIndecies(longestStreakPairs,1),2));
   %runs these indecies through the filtered function, and averages the values
at these indecies
end
function [timeConstant] =
timeConstantFinder(dataIn,finalVelocity,firstSectionPercentage,endSectionPerce
ntage,givenTitle,measuredQuantity,voltageString)
   dimensionlessVelocity = real(log((finalVelocity -
dataIn(:,2))/finalVelocity));%log without subscript in matlab is ln
   graphSection = [floor(firstSectionPercentage * length(dataIn(:,1)) ) ,
ceil(endSectionPercentage * length(dataIn(:,1)) )];
   if graphSection(1) == 0
       graphSection(1) = 1;
   end
   slope = polyfit( dataIn(graphSection(1):graphSection(2),1), ...
       dimensionlessVelocity(graphSection(1):graphSection(2)) ,1 );
   %using polyfit to get the slope of the linear part of the plot
   timeConstant = -1/slope(1);
   figure;
   plot(dataIn(:,1),dimensionlessVelocity,".");
   hold on;
   t = linspace(dataIn(1,1),dataIn(end,1),100);
   plot(t,t*slope(1));
   hold off;
   title(givenTitle,'Interpreter','latex')
   xlabel("$ Time, \ t \ (s) $",'Interpreter','latex')
   ylabel(measuredQuantity,'Interpreter','latex')
   legendStuff = [givenTitle, strcat("$Estimated \ " , givenTitle, ", \
\tau_{motor} = \ ", num2str(timeConstant), "s \ $"),strcat("$Final \ Value \ "
, givenTitle, " \ $")];
   lgnd= legend(legendStuff,'Location','best');
   set(lgnd, 'Interpreter','latex')
   plot1 = strcat("RAW Dimensionless Velocity And Time Constant For ",
voltageString ,  " vs time");
   print('-r600','-dpng',plot1);
end
function [firstOrder,shiftPoint] =
firstOrderApproximation(filteredData,originalData,finalVal,timeConst)
```

```
   pointAtInitTime = originalData(2,2); %is NaN velocity at t=0,
   %y0 = fv * (1 - e^(-(1/tau) * (t - x0) ) ) ----> 1 - (y0/fv) = e^(-(1/tau)
* (t - x0) )
   %ln( 1 - (y0/fv) ) / (-1/tau) = (t - x0) ----> x0 = t + tau * ln( 1 -
(y0/fv) )
   shiftPoint = originalData(2,1) + timeConst*log( 1 -
pointAtInitTime/finalVal);
   %Accounts for the fact that the recorded data from tracker may not have
   %started at 0 velocity, so just shifts the graph left or right to try
   %getting the fit better.
   t = linspace(filteredData(1,1),filteredData(end,1),1000);
   firstOrderApprox = finalVal * (1 - exp( (-1/timeConst) * (t - shiftPoint) )
);
   firstOrder = [t',firstOrderApprox'];
end
function
rawAndFilteredPlotter(dataList,marker,givenTitle,measuredQuantity,finalVal,tim
eConst,timeShift,voltageString)
   figure;
   for i = 1:length(dataList)
       plot(dataList{i}(:,1),dataList{i}(:,2),marker(i))
       hold on;
   end
   % plot(originalData(:,1),originalData(:,2),".")
   % hold on;
   % yline(finalVal,"--b");
   % hold on;
   % plot(firstOrder(:,1),firstOrder(:,2));
   % hold off;
   yline(finalVal,"--b");
   hold off;

   ylim([0,finalVal*1.05]);
   title(givenTitle,'Interpreter','latex')
   xlabel("$ Time, \ t \ (s) $",'Interpreter','latex')
   ylabel(measuredQuantity,'Interpreter','latex')
   legendStuff = [givenTitle,strcat("$First \ Order \ Approximation \ Of \ "
,...
       givenTitle , " \ $" , newline + " $ V(t) = " ,  num2str(finalVal)," (1
- e^{ " , num2str(-1/timeConst),...
       " (t + " , num2str(timeShift)," ) } ) \ , \" , " \tau_{motor} = \ ",
num2str(timeConst), "s, \ $"),...
       strcat("$Initial \ Slope \ Approximation $"),...
       strcat("$Final \ Value \ " , givenTitle, " \ = \ ", num2str(finalVal)
," \ ( \frac{m}{s} ) $" )];
   lgnd= legend(legendStuff,'Location','best');
   set(lgnd, 'Interpreter','latex')
   plot1 = strcat("RAW All Velocity Plots For", voltageString ,  " vs time");
   print('-r600','-dpng',plot1);
end
function [initSlope,tau_initSlope,plottableLine] =
initialSlopeTimeConstantFinder(dataIn,firstSectionPercentage,endSectionPercent
age)
   graphSection = [floor(firstSectionPercentage * length(dataIn(:,1)) ) ,
ceil(endSectionPercentage * length(dataIn(:,1)) )];
   if graphSection(1) == 0
       graphSection(1) = 1;
   end
   initSlope =
polyfit(dataIn(graphSection(1):graphSection(2),1),dataIn(graphSection(1):graph
```

```
Section(2),2),1);
   tau_initSlope = 1/initSlope(1);

   t = linspace(dataIn(1,1),dataIn(end,1),100);
   plottableLine = [t;initSlope(1)*(t - dataIn(graphSection(1),1) ) +
dataIn(graphSection(1),2)]';
end
```

Lab 2 Code: Sensor Linearization

```
clear;
clc;
close all;
sensorData = table2array(readtable("Ducky Lab 2
Data.xlsx"','Range','A2:B18'));
setPoint = sensorData(1,:);%( distance in inches x_0 , voltage reading v_so )
x = linspace(sensorData(1,1),sensorData(end,1),100);
%%Non-Linear Curve Fit
%is actually done by linear curve fitting x vs 1/V_s
%(sensorData(:,1),1./sensorData(:,2))
%if you plot this ^^^, you can see it actually becomes quite linear...
inv = polyfit(sensorData(:,1),1./sensorData(:,2),1);
curveApprox = 1./( inv(1)*x + inv(2) );
metricInv = polyfit(sensorData(:,1),1./(0.0254*sensorData(:,2)),1);
metricCurveApprox = 1./( metricInv(1)*x + metricInv(2) );
%%Linear Approximation About A Point
[fiveIn_linearApprox,five_Ks] = linearizeAboutPoint(curveApprox,x,5,10^-1);
[sevenIn_linearApprox,seven_Ks] = linearizeAboutPoint(curveApprox,x,7,10^-1);
[tenIn_linearApprox,ten_Ks] = linearizeAboutPoint(curveApprox,x,10,10^-0.75);
plot(sensorData(:,1),sensorData(:,2),".")
hold on;
plot(x,curveApprox)
hold on;
plot(x,fiveIn_linearApprox,"--")
hold on;
plot(x,sevenIn_linearApprox,"--")
hold on;
plot(x,tenIn_linearApprox,"--")
title("$Sensor \ Voltage \ V_{s} \ vs \ Distance \ (in)
$",'Interpreter','latex')
xlabel("$ Distance \ (in) $",'Interpreter','latex')
ylabel("$Sensor \ Voltage \ V_{s}$",'Interpreter','latex')
legendStuff = ["$Raw \ Data$", strcat("$Rational Approximation, V_{s}(x) = \
\frac{1}{ " ...
, num2str(inv(1)) , " x + " , num2str(inv(1)), " } $"),...
strcat("$Linear \ Approximation \ at \ 5in, \ K_s = " , num2str(five_Ks) ,
"$"),...
strcat("$Linear \ Approximation \ at \ 7in, \ K_s = " , num2str(seven_Ks) ,
"$"),...
strcat("$Linear \ Approximation \ at \ 10in, \ K_s = " , num2str(ten_Ks) ,
"$")];
lgnd= legend(legendStuff,'Location','best');
set(lgnd, 'Interpreter','latex')
%hold on
%plot(sensorData(:,1),1./sensorData(:,2),".") %if you plot this, you can
```

```
%see it actually becomes quite linear...
plot1 = strcat("Sensor Voltage vs Distance");
print('-r600','-dpng',plot1);
hold off;
function [linearApproximation,slope] =
linearizeAboutPoint(y_out,x_in,x_point,tol)
      deriv = diff(y_out)./diff(x_in);
      ind = find( abs(x_in - x_point) <= tol );
      ind = ind(end);
      slope = deriv(ind);
      linearApproximation = y_out(ind) + deriv(ind) * (x_in - x_point);
end
```

Lab 3 Code: Critically Damped Vs Underdamped Performance

```
clear;
clc;
close all;
%%Experiment Constants
standOffDistInch = 7.00;
standoffV = 1.335;
finalDistInch = 13.00;
K_a = 1.2;
K_m = mean([0.3297,0.3789,0.44694]);
K_m_inch = K_m/0.0254;
motorTau = mean([0.6578,0.6471,0.6328]);
nonLinearSensor = [0.0864192841833102 , 0.140817267512297];
nonLinearSensor_metric = [3.4023,5.5440];
%%Importing Data
dampGainData = table2array(readtable("2.942gain.txt"));
sixNineGainData = table2array(readtable("6.84gain.txt"));
%need to reorder columns of damp data, since velocity is in last column
tempVel = dampGainData(:,2);
dampGainData(:,2) = dampGainData(:,3);
dampGainData(:,3) = tempVel;
clearvars tempVel
K_p_damp = 2.942;
K_p_sixNine = 6.84;
dampGainData = NaNRemover(dampGainData);
sixNineGainData = NaNRemover(sixNineGainData);
dampPosFinalValue = rawDataFinalValueFinder(dampGainData(:,3),[0.75,1]);
dampVelFinalValue =
round(rawDataFinalValueFinder(dampGainData(:,2),[0.75,0.8]) ,2);%is
practically 0 in set, and should be 0 in real life
sixNinePosFinalValue = rawDataFinalValueFinder(sixNineGainData(:,3),[0.6,1]);
sixNineVelFinalValue =
round(rawDataFinalValueFinder(sixNineGainData(:,2),[0.65,0.75]) ,1);%is
practically 0 in set, and should be 0 in real life
%%finding time constants, settling time, peak time, etc...
[tau_xDamp,Ts_xDamp] =
TwoPercentSettlingTimeTau(dampGainData,3,dampPosFinalValue,10^-2,0.175)
[tau_xSixNine,Ts_xSixNine] =
TwoPercentSettlingTimeTau(sixNineGainData,3,sixNinePosFinalValue,10^-2,0.175)
[sixNineMax,sixNinePeakTime,sixNineMaxOvershoot,sixNineMaxPO] =
overshootFinder(sixNineGainData,3,sixNinePosFinalValue)
dampingValue = -log(sixNineMaxPO/100) / sqrt( pi^2 + (log(sixNineMaxPO/100))^2
)
```

```
natFreq = pi / ( sixNinePeakTime * sqrt( 1 - dampingValue^2 ) )
%%plotting
sameSetDataPlotter(dampGainData,[dampVelFinalValue,dampPosFinalValue],strcat("
$ \ Dampened \ Gain \ Value \ K_{p} = " ,...
num2str(K_p_damp)," \ Raw \ Data \ $"), "$ Time \ (s) $" ,...
["$Velocity \ v(t) $", strcat("$Final \ Velocity \ v_{ss} = "
,num2str(dampVelFinalValue) ," \frac{m}{s} $") ,...
"$Positon \ x(t)$" ,strcat("$Final \ Position \ x_{ss} = "
,num2str(dampPosFinalValue) ," m $") ],"Dampened Gain Movement")
sameSetDataPlotter(sixNineGainData,[sixNineVelFinalValue,sixNinePosFinalValue]
,strcat("$ \ Overshoot \ Gain \ Value \ K_{p} = ",...
num2str(K_p_sixNine)," \ Raw \ Data \ $"),"$ Time \ (s) $",...
["$Velocity \ v(t) $", strcat("$Final \ Velocity \ v_{ss} = "
,num2str(sixNineVelFinalValue) ," \frac{m}{s} $") ,...
"$Positon \ x(t)$" ,strcat("$Final \ Position \ x_{ss} = "
,num2str(sixNinePosFinalValue) ," m $") ], "Overshoot Gain Movement")

%%functions
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

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

      function [rawDataFinalValue] =
      rawDataFinalValueFinder(dataInputted,timeSection)
      %will only look at last set of time values for final value
      lastSectionOfTime = [floor( height(dataInputted)*timeSection(1)) , ceil(
      height(dataInputted)*timeSection(2))];
      rawDataFinalValue =
      mean(dataInputted(lastSectionOfTime(1):lastSectionOfTime(2)));
      %runs these indecies through the filtered function, and averages the
      values at these indecies
end

function
sameSetDataPlotter(datas,finalValues,titleStuff,xAxisLabel,legendStuff,name)
      figure;
      for i = 2:width(datas) %final values length is the same as datas width
            pointPlot = plot(datas(:,1),datas(:,i),".");
            hold on;
            %just gets it the same color
```

```
            yline(finalValues(i-1),"--",'color',get(pointPlot,'color'))
            hold on;
      end
      hold off
      title(titleStuff,'Interpreter','latex')
      xlabel(xAxisLabel,'Interpreter','latex')
      lgnd= legend(legendStuff,'Location','best');
      set(lgnd, 'Interpreter','latex')
      plot1 = name;
      print('-r600','-dpng',plot1);
end

function [settlingTimeVersion_Tau,settlingTime] =
TwoPercentSettlingTimeTau(dataIn,dataColumn,finalVal,tol,streakFraction)
minContinuousLength = floor( streakFraction * length(dataIn) ); %has to be at
least spanning 3% of the graph
%minContinuousLength = floor( 0.03 * length(dataIn) ); %has to be at least
spanning 3% of the graph
      % will find the first continous set of values within 2% that is at
      % least "minContinuousLength" indecies long
      chosenIndecies = find(abs(finalVal*0.982 - dataIn(:,dataColumn)) <=
      tol); %if the data is 98% of the final val within a tolerance
      consecutiveToleranceIndecies = [;]; % [ start1,start2 ; end1,end2 ; ...
      ]
      itt = 1;
      %will now check for the longest streak of consecutive indecies that fall
      within tolerance
      while itt <= length(chosenIndecies)
            a = chosenIndecies(itt);
            iitt = itt+1; %check for consecutive
                  while iitt <= length(chosenIndecies) &&
                  chosenIndecies(iitt) - chosenIndecies(itt) < 2
                  iitt = iitt+1;
                  itt = itt + 1;
            end
            b = chosenIndecies(itt);
            consecutiveToleranceIndecies = [consecutiveToleranceIndecies ;
            [a,b]];
            itt = itt+1;
      end
      consecutiveToleranceIndecies;

      % will find the first continous set of values within 2% that is at
      % least "minContinuousLength" indecies long
      for i = 1:height(consecutiveToleranceIndecies)
                  if consecutiveToleranceIndecies(i,2) -
                  consecutiveToleranceIndecies(i,1) >= minContinuousLength
                  chosenIndecies =
                  consecutiveToleranceIndecies(i,1):1:consecutiveToleranceInd
                  ecies(i,2);
                  break;
            end
      end
      chosenIndecies;
      %Uses the first index of that streak to get the time
      settlingTime = dataIn(chosenIndecies(1),1);
      settlingTimeVersion_Tau = settlingTime/4;
end

      function [maximumValue,peakTime,overshoot,percentOvershoot] =
```

```
      overshootFinder(dataIn,dataColumn,finalVal)
      [maximumValue , indOfMax] = max(dataIn(:,dataColumn) );
      maximumValue;
      peakTime = dataIn(indOfMax,1);
      overshoot = (maximumValue - finalVal);
      percentOvershoot = overshoot/finalVal *100;
end
```

Lab 4 Code: PD Control Performance

```
clear;
clc;
close all;
%%Experiment Constants
standOffDistInch = 7.00;
standoffV = 1.335;
finalDistInch = 13.00;
K_a = 1.2;
K_m = mean([0.3297,0.3789,0.44694]);
K_m_inch = K_m/0.0254;
motorTau = mean([0.6578,0.6471,0.6328]);
nonLinearSensor = [0.0864192841833102 , 0.140817267512297];
nonLinearSensor_metric = [3.4023,5.5440];
%%Importing Data
labGainData = table2array(readtable("Lab4_slomo_vel_pos.txt"));
K_p_desired = 7.8;
K_d_desired = (78*10^3)*(13.3*10^-6);
TS_desired = 1.0312;
tau_desired = TS_desired/4;
labGainData = NaNRemover(labGainData);
posFinalValue = rawDataFinalValueFinder(labGainData(:,3),[0.75,1]);
velFinalValue = round(rawDataFinalValueFinder(labGainData(:,2),[0.75,0.8])
,1);%is practically 0 in set, and should be 0 in real life
%%finding time constants, settling time, peak time, etc...
[tau,Ts] = TwoPercentSettlingTimeTau(labGainData,3,posFinalValue,10^-2,0.175)
[posMax,posPeakTime,posMaxOvershoot,posMaxPO] =
overshootFinder(labGainData,3,posFinalValue)
dampingValue = -log(posMaxPO/100) / sqrt( pi^2 + (log(posMaxPO/100))^2 )
natFreq = pi / ( posPeakTime * sqrt( 1 - dampingValue^2 ) )
%%plotting
sameSetDataPlotter(labGainData,[velFinalValue,posFinalValue],strcat("$ \ K_{p}
= " ,...
num2str(K_p)," , " , " K_{d} = " ,num2str(K_d), ", \ Raw \ Data \ $"), "$ Time
\ (s) $" ,...
["$Velocity \ v(t) $", strcat("$Final \ Velocity \ v_{ss} = "
,num2str(velFinalValue) ," \frac{m}{s} $") ,...
"$Positon \ x(t)$" ,strcat("$Final \ Position \ x_{ss} = "
,num2str(posFinalValue) ," m $") ])

%%functions
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

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

function [rawDataFinalValue] =
rawDataFinalValueFinder(dataInputted,timeSection)
      %will only look at last set of time values for final value
      lastSectionOfTime = [floor( height(dataInputted)*timeSection(1)) , ceil(
      height(dataInputted)*timeSection(2))];
      rawDataFinalValue =
      mean(dataInputted(lastSectionOfTime(1):lastSectionOfTime(2)));
      %runs these indecies through the filtered function, and averages the
      values at these indecies
end
      function
      sameSetDataPlotter(datas,finalValues,titleStuff,xAxisLabel,legendStuff)
      figure;
      for i = 2:width(datas) %final values length is the same as datas width
            pointPlot = plot(datas(:,1),datas(:,i),".");
            hold on;
            %just gets it the same color
            yline(finalValues(i-1),"--",'color',get(pointPlot,'color'))
            hold on;
      end
      hold off
      title(titleStuff,'Interpreter','latex')
      xlabel(xAxisLabel,'Interpreter','latex')
      lgnd= legend(legendStuff,'Location','best');
      set(lgnd, 'Interpreter','latex')
      plot1 = "Experimental Data";
      print('-r600','-dpng',plot1);
end

function [settlingTimeVersion_Tau,settlingTime] =
TwoPercentSettlingTimeTau(dataIn,dataColumn,finalVal,tol,streakFraction)
      minContinuousLength = floor( streakFraction * length(dataIn) ); %has to
      be at least spanning 3% of the graph
      %minContinuousLength = floor( 0.03 * length(dataIn) ); %has to be at
      least spanning 3% of the graph
      % will find the first continous set of values within 2% that is at
      % least "minContinuousLength" indecies long
      chosenIndecies = find(abs(finalVal*0.982 - dataIn(:,dataColumn)) <=
      tol); %if the data is 98% of the final val within a tolerance
      consecutiveToleranceIndecies = [;]; % [ start1,start2 ; end1,end2 ; ...
      ]
      itt = 1;
      %will now check for the longest streak of consecutive indecies that fall
```

```
      within tolerance
      while itt <= length(chosenIndecies)
            a = chosenIndecies(itt);
            iitt = itt+1; %check for consecutive
            while iitt <= length(chosenIndecies) && chosenIndecies(iitt) -
            chosenIndecies(itt) < 2
                  iitt = iitt+1;
                  itt = itt + 1;
            end
            b = chosenIndecies(itt);
            consecutiveToleranceIndecies = [consecutiveToleranceIndecies ;
            [a,b]];
            itt = itt+1;
      end
      consecutiveToleranceIndecies;
      % will find the first continous set of values within 2% that is at
      % least "minContinuousLength" indecies long
      for i = 1:height(consecutiveToleranceIndecies)
                  if consecutiveToleranceIndecies(i,2) -
                  consecutiveToleranceIndecies(i,1) >= minContinuousLength
                  chosenIndecies =
                  consecutiveToleranceIndecies(i,1):1:consecutiveToleranceInd
                  ecies(i,2);
                  break;
            end
      end
      chosenIndecies;
      %Uses the first index of that streak to get the time
      settlingTime = dataIn(chosenIndecies(1),1);
      settlingTimeVersion_Tau = settlingTime/4;
end

      function [maximumValue,peakTime,overshoot,percentOvershoot] =
      overshootFinder(dataIn,dataColumn,finalVal)
      [maximumValue , indOfMax] = max(dataIn(:,dataColumn) );
      maximumValue;
      peakTime = dataIn(indOfMax,1);
      overshoot = (maximumValue - finalVal);
      percentOvershoot = overshoot/finalVal *100;
end
```

PLOTS w/ Additional Data

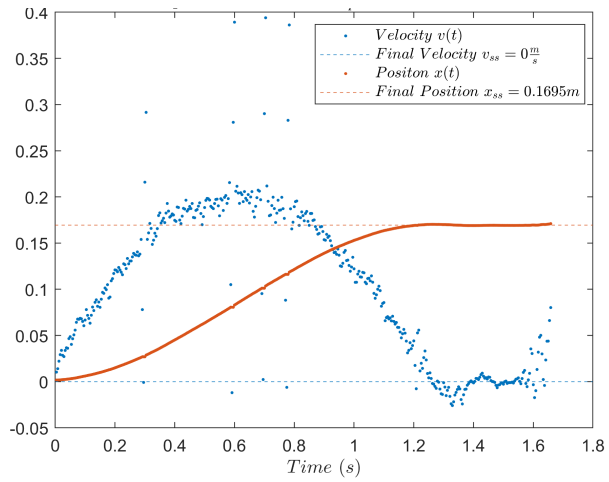


Figure 1: *Actual Motion at* $K_p = 2.942$

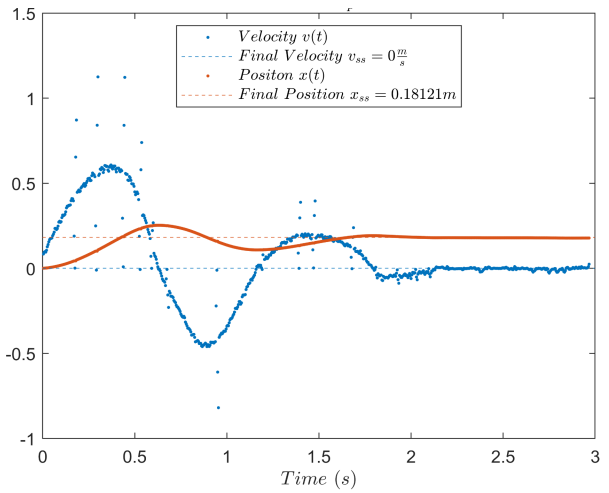


Figure 2: *Actual Motion at* $K_p = 6.84$

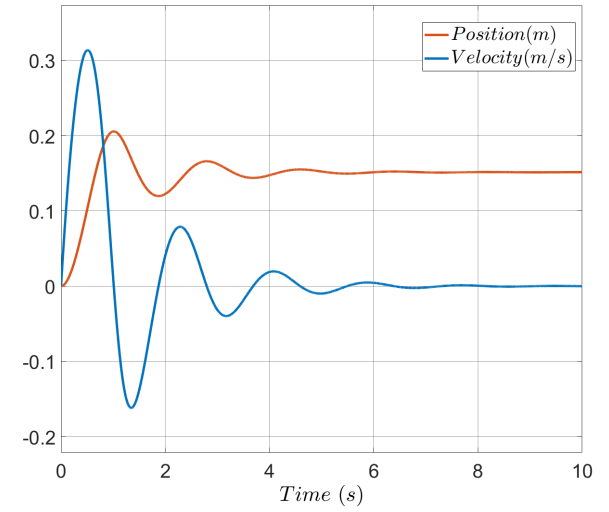


Figure 3: *SIMULINK's Model at* $K_p = 2.942$

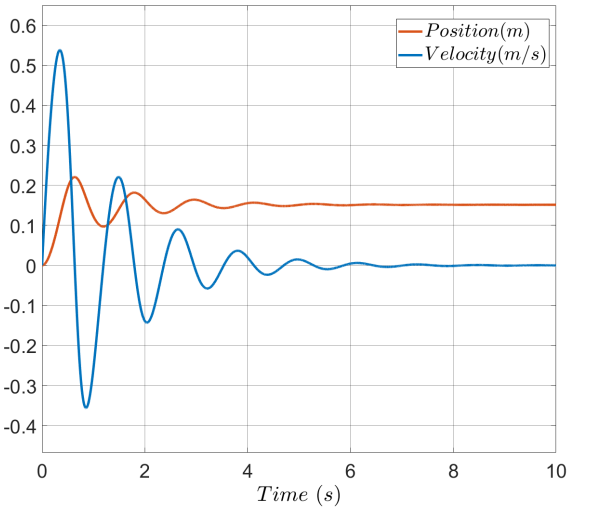


Figure 4: *SIMULINK's Model at* $K_p = 6.84$

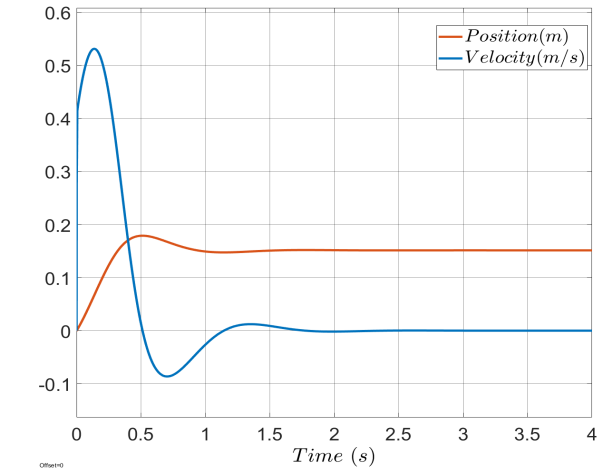


Figure 5: *SIMULINK'S Model at* $K_p = 7.8,\ K_d = 1.0374$

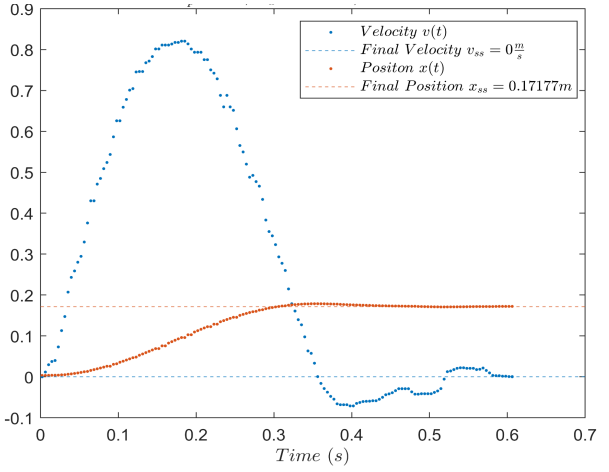


Figure 6: *Actual Motion at* $K_p = 7.8,\ K_d = 1.0374$

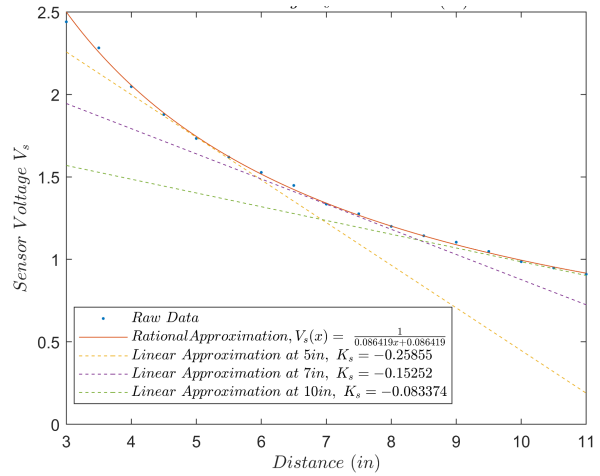


Figure 7: Sensor Voltage Vs Distance, with Linearizations, K_s at 5 in, 7 in, 10 in

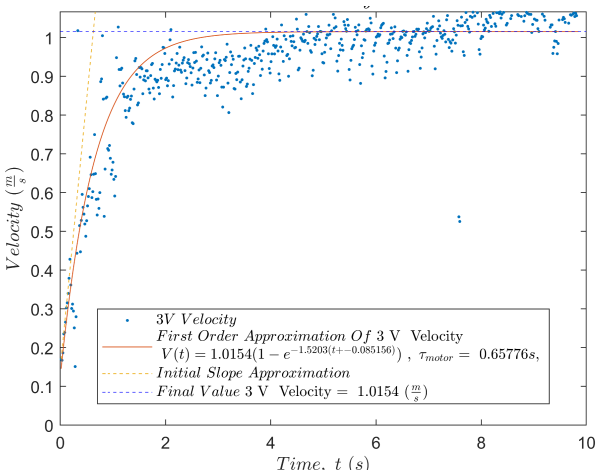


Figure 8: Letting The Car Run Freely At 3V

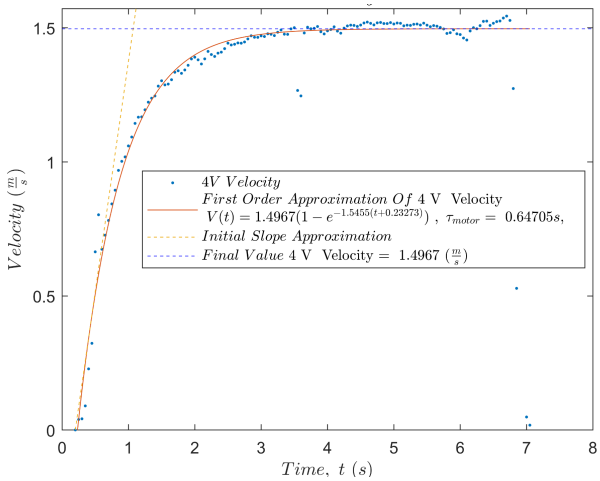


Figure 9: Letting The Car Run Freely At 4V

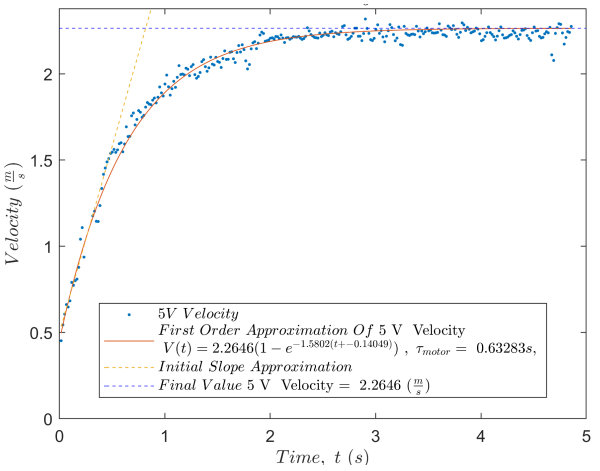


Figure 10: Letting The Car Run Freely At 5V

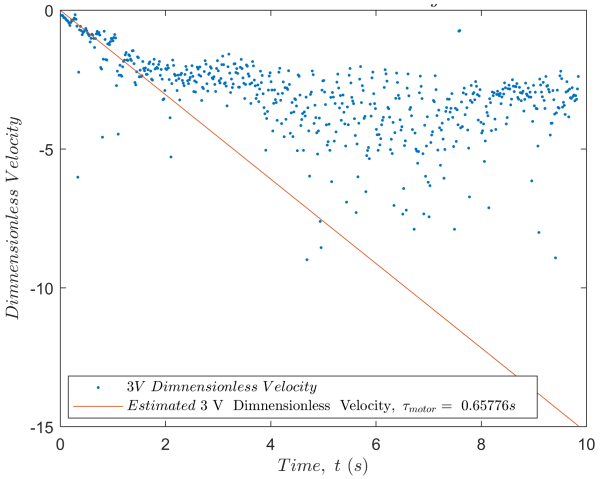


Figure 11: Dimensionless Velocity From Letting The Car Run Freely At 3V

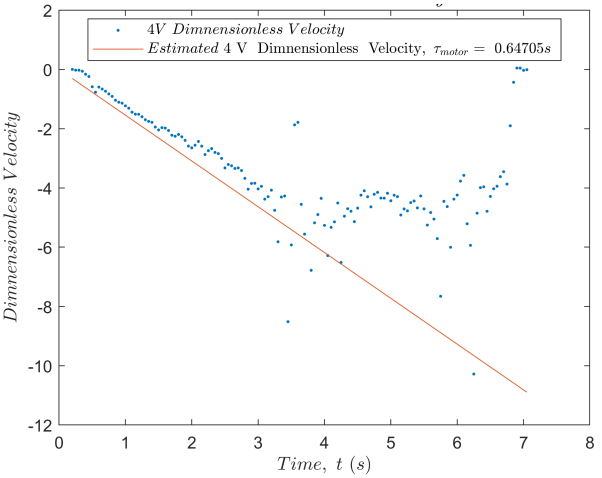


Figure 12: Dimensionless Velocity From Letting The Car Run Freely At 4V

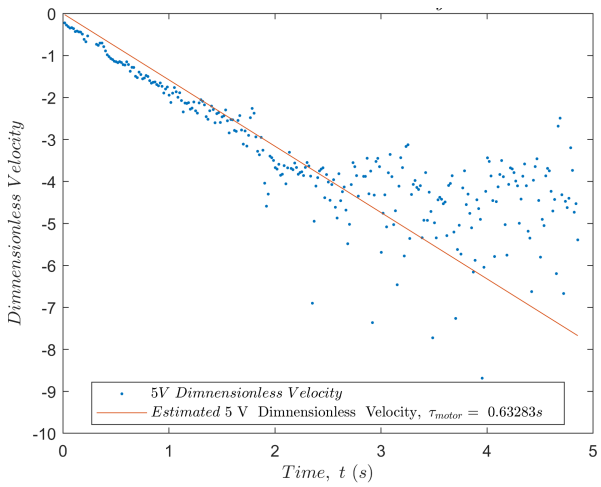


Figure 13: Dimensionless Velocity From Letting The Car Run Freely At 5V

Applied Voltage (V)	$Vel_{ss}\ \frac{m}{s}$	$K_m = \frac{Vel_{ss}}{V_a}\ (\frac{m/s}{V})$	$\tau_{motor}\ (s)$
3	0.98929	0.3297	0.6578
4	1.516	0.3789	0.6471
5	2.2347	0.44694	0.6328
		Mean: 0.38518	Mean: 0.6459