

December 15, 2017

Dr. Barry Fussell and Dr. Alireza Ebadi

Dept. of Mechanical Engineering

University of New Hampshire

33 Academic Way

Durham, NH 03824

Dear Professor Fussell and Dr. Ebadi,

With the anticipation of the UNH’s LunaCats competing at the NASA Robotics Mining Competition our team has decided to develop PID control for both the velocity of the Mars Bot and the Mars Cat’s auger velocity.

For the purpose of this experiment a theoretical model for a smaller former battlebot called ‘Another Brick in the Wall’ is developed to compare theoretical results qualitatively to experimental results for different control values. Another model is made for the Mars Cat’s auger to be controlled by a PID controller based on the robot’s position from a wall at 90° to simulate the Mars Bot’s approach to the mining area. Both system models are created in Simulink, and MATLAB is used to for the analysis of their system constants and evaluating proper controller values.

Regards,

Team Velocity and Position Control of a Robot Vehicle and Auger Motor

Members:

Zhangxi ‘Jesse’ Feng

Michael Locke

Simon Popecki

James Skinner

Matthew Westbrook



|  |  |
| --- | --- |
| Course Number and Name:  ME 747  Experimental Measurement and Modeling of Complex Systems | |
| Semester and Year:  Fall 2017 | Name of Lab Instructor:  Barry Fussell  Alireza Ebadi |
| Lab Section and Meeting Time:  Lab Section 2B/3B  Tuesdays/Wednesdays 2:00-5:00PM | Report Type:  External Group Report |
| Title of Experiment:  Velocity Control of a Robot Vehicle and Auger Motor  -Final Laboratory- | |
| Date Experiment Performed:  15 November – 15 December 2017 | Date Report Submitted:  15 December 2017 |
| Names of Group Members:  Zhangxi ‘Jesse’ Feng  Michael Locke  Simon Popecki  James Skinner  Matthew Westbrook | Grader's Comments: |
| Grade: |

TABLE OF CONTENTS

[OBJECTIVES 1](#_Toc501104172)

[EXECUTIVE SUMMARY 2](#_Toc501104173)

[1 THEORY AND EXPERIMENTAL METHODS 3](#_Toc501104174)

[1.1 Auger Motor – Velocity PID Control 3](#_Toc501104175)

[1.2 Robot Drive Train – Position PID Control of Velocity 5](#_Toc501104176)

[1.3 Proportional-Integral-Derivative Control Theory 8](#_Toc501104177)

[2 RESULTS AND DISCUSSION 11](#_Toc501104178)

[2.1 Velocity Control of the Auger Motor 11](#_Toc501104179)

[2.2 Position Control of the Robot’s Velocity 17](#_Toc501104180)

[3 CONCLUSIONS 19](#_Toc501104181)

[3.1 Zhangxi ‘Jesse’ Feng 19](#_Toc501104182)

[3.2 Michael Locke 19](#_Toc501104183)

[3.3 Simon Popecki 19](#_Toc501104184)

[3.4 James Skinner 19](#_Toc501104185)

[3.5 Matthew Westbrook 19](#_Toc501104186)

[4 REFERENCES 21](#_Toc501104187)

[5 APPENDIX 21](#_Toc501104188)

[A: Equipment List 21](#_Toc501104189)

[B: MatLab Codes 21](#_Toc501104190)

LIST OF FIGURES

[Figure 1: Velocity Control Process Diagram 4](file:///D:\ME747\Final\final_report.docx#_Toc501109762)

[Figure 2: Wiring Diagram for Experimental Setup 4](file:///D:\ME747\Final\final_report.docx#_Toc501109763)

[Figure 3: Experimental Prototypes (Left: Original, Right: Final) 5](file:///D:\ME747\Final\final_report.docx#_Toc501109764)

[Figure 4: Robot “Another Brick in the Wall” with direct drive DC brushed gear motors 6](file:///D:\ME747\Final\final_report.docx#_Toc501109765)

[Figure 5: Robot motor step response to PWM signal 6](file:///D:\ME747\Final\final_report.docx#_Toc501109766)

[Figure 6: DC Motor electromechanical model 7](file:///D:\ME747\Final\final_report.docx#_Toc501109767)

[Figure 7: Motor system block diagram 7](file:///D:\ME747\Final\final_report.docx#_Toc501109768)

[Figure 8: System block diagram for the auger motor 9](file:///D:\ME747\Final\final_report.docx#_Toc501109769)

[Figure 9: Calibration of DC Vex 393 motor with varying PWM inputs 11](file:///D:\ME747\Final\final_report.docx#_Toc501109770)

[Figure 10: Velocity of motor with PWM step input 12](file:///D:\ME747\Final\final_report.docx#_Toc501109771)

[Figure 11: Step input with proportional gain of 0.5 12](file:///D:\ME747\Final\final_report.docx#_Toc501109772)

[Figure 12: Step input to motor with I-control 13](file:///D:\ME747\Final\final_report.docx#_Toc501109773)

[Figure 13: Step input to motor with PI-control 13](file:///D:\ME747\Final\final_report.docx#_Toc501109774)

[Figure 14: Step input to motor with PID-control 14](file:///D:\ME747\Final\final_report.docx#_Toc501109775)

[Figure 15: Root locus of motor with P-control 15](file:///D:\ME747\Final\final_report.docx#_Toc501109776)

[Figure 16: Root locus of motor with I-control. 15](file:///D:\ME747\Final\final_report.docx#_Toc501109777)

[Figure 17: Root locus of motor with PI-control 16](file:///D:\ME747\Final\final_report.docx#_Toc501109778)

[Figure 18: Root locus of motor with PID-control 16](file:///D:\ME747\Final\final_report.docx#_Toc501109779)

[Figure 19: The resulting output voltage of the robot driving along the table 17](file:///D:\ME747\Final\final_report.docx#_Toc501109780)

LIST OF TABLES

[Table 1: Auger motor analysis summary 2](#_Toc501104134)

[Table 2: Another Brick in the Wall system parameters 16](#_Toc501104135)

# OBJECTIVES

The goal of this project is to determine the appropriate constants of two PID controllers for a robot’s DC brush motors and an auger motor. The controller for the robot receives the position data from an ultrasonic sensor and produces a velocity output to the motors. The controller for the motor receives a step voltage input and produces an output that optimizes both fast response and minimal oscillation from overshoot.

# EXECUTIVE SUMMARY

Analysis of the auger motor was done to find system parameters which were then used to compare different methods of velocity control. Parameters were found by putting a step input of known velocities into the system. The output was then compared to the input signal to find the system gain and time constant. Velocity was then controlled using four different methods; proportional (P), integral (I), proportional-integral (PI) and proportional-integral-derivative (PID) control. The P control had the fastest response but was the only control method that had a steady-state error. The PID control had the second fastest response and did not have any steady-state error. Results from analysis of the auger motor are summarized in **Table 1** below.

Table : Auger motor analysis summary

|  |  |
| --- | --- |
| **Time Constant, τ (sec)** | 0.35 |
| **System Gain, K** | 0.77 |
| **P-control steady-state error (%)** | 16.00 |
| **P-control speed of response (sec)** | 0.35 |
| **I-control steady-state error (%)** | 0 |
| **I-control speed of response (sec)** | 5.60 |
| **PI-control steady-state error (%)** | 0 |
| **PI-control speed of response (sec)** | 3.13 |
| **PID-control steady-state error (%)** | 0 |
| **PID-control speed of response (sec)** | 2.42 |

# 1 THEORY AND EXPERIMENTAL METHODS

The UNH LunaCats designs, builds, and competes with a mining robot at NASA’s Robotics Mining Competition in Kennedy Space Center at Cape Canaveral every May. The robot needs to traverse through terrain with random obstacles to reach a mining area. The material must be mined from the mining area only, therefore it is beneficial for the robot to reach the area as quickly as possible with no overshoot such that the robot does not waste time traveling any extra distance. This year’s LunaCats robot will use an auger as the mining apparatus. The auger consisted of a rotary motor connected to auger tip that acts like an Archimedes’ screw. For this project, the auger motor was modeled using a VEX robotics gearbox motor as shown in **Figure 2**; the drive train was modeled using the robot named Another Brick in the Wall as shown in **Figure 4**.

## 1.1 Auger Motor – Velocity PID Control

For the auger subsystem, proportional-integral-derivative (PID) velocity control was developed in this experiment and tested on a prototype platform to evaluate the control algorithm. The goal of this platform and algorithm is to develop an understanding of P, I, PI, and PID control using sensor feedback and a dedicated microcontroller to process the sensor inputs and motor outputs. To start, a basic process diagram was developed to understand the velocity control process, in which four main parts to this experiment were determined. First, a means of delivering a step input via user input was needed. Second, a microcontroller was needed to take this step input and control a motor accordingly. Third, a motor to be tested was needed that would provide sufficient output speed. Fourth, a sensor that could be used to provide velocity feedback from the motor to the microcontroller was needed to act as the basis for the control data. **Figure 1** demonstrates this process flow:

Knowing the overall process of the velocity control, components were selected that would satisfy each of the four components shown in **Figure 1** and that were easily accessible. An Arduino Uno served at the microcontroller that would process the inputs, outputs, and velocity control algorithm. A basic push button served as the means of user input to initiate a step input. A VEX robotics gearbox motor served as the tested motor for its desirable working speed of about 350RPM. Lastly, an encoder was chosen to provide feedback to the Arduino. Since the encoder measures position and not velocity, the position data was converted to velocity data by taking the difference in positional encoder ticks (rotational resolution) and dividing by the corresponding change in time measured using an internal timer on the microcontroller.

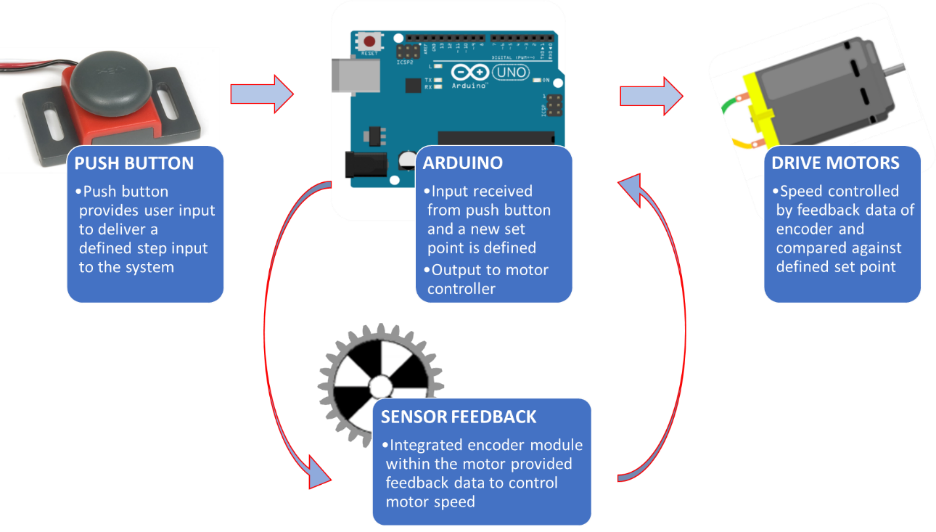


Figure 1: Velocity Control Process Diagram

The next step was to combine all the components. A wiring diagram was created to define how each component would be connected and to serve as a reference when creating the circuit. **Figure 2** shows the velocity control experimental circuit:

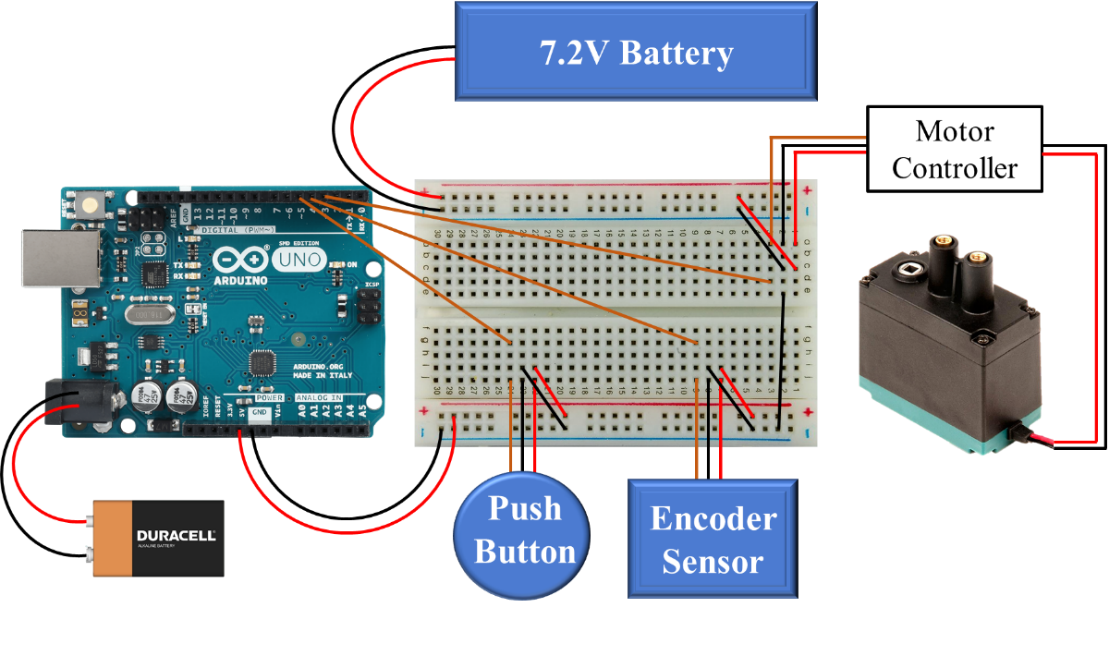


Figure 2: Wiring Diagram for Experimental Setup

From the wiring diagram, note that an extra 7.2-volt battery was used to provide enough power to the motor controller and motor. Using this layout, the experimental setup could finally be constructed. The first experimental prototype and final experimental prototype can be seen below:

As shown in **Figure 3**, multiple experimental setups transpired throughout this project. The first version contained an Arduino microcontroller, separate quadrature encoder, a motor, and push button. The final version contained a VEX Robotics microcontroller, integrated encoder module with the motor housing, a motor, and push button. The reason for this change in equipment was due to numerous issues with components burning out and not being able to easily determine a way for the Arduino to capture the rapid change of encoder rotary ticks to track position. The VEX Robotics microcontroller has built in encoder features to accurately track encoder position, so it was used in the final setup.

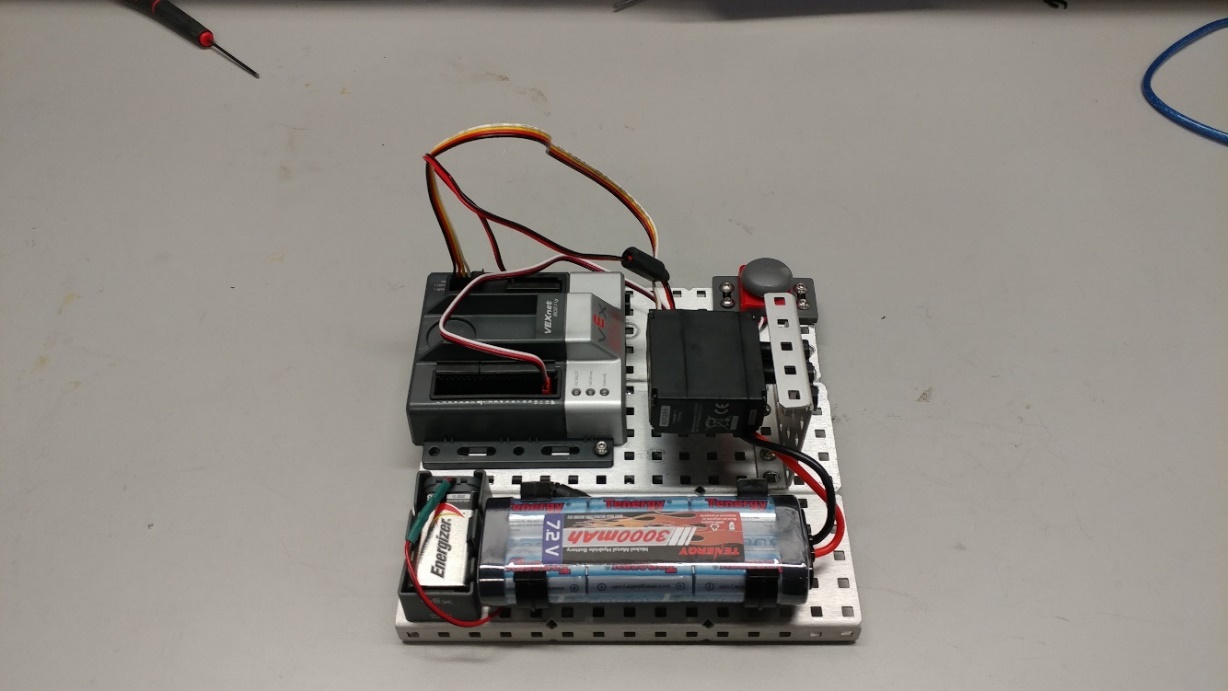
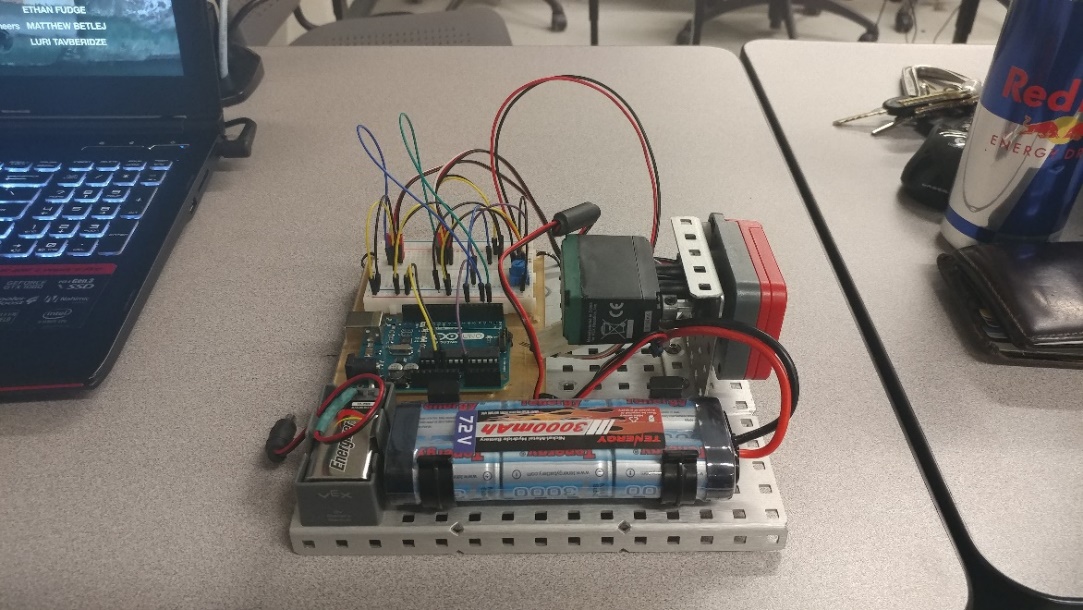


Figure 3: Experimental Prototypes (Left: Original, Right: Final)

## 1.2 Robot Drive Train – Position PID Control of Velocity

For the drive train subsystem, a proportional-integral-derivative (PID) position-feedback velocity control was also investigated for validity and necessity. The system parameters were experimentally determined.

Each of the wheels on the robot was directly driven by a DC brush motor. Motors on each side were soldered together for tank drive. The motors were connected to an electronic speed controller (ESC), Sabretooth 2X12-RC, that uses pulse width modulated (PWM) signal to maximize the efficiency of the motors. The robot’s position was measured by an ultrasonic sensor. The components were connected and controlled via an Arduino Mega board. The robot and its components are shown in **Figure 4**.

The system time constant was determined by analyzing the motor’s step response curve (**Figure 5**) to a set PWM duty cycle signal from the ESC.



Figure 4: Robot “Another Brick in the Wall” with direct drive DC brushed gear motors

Electronic Speed Controller (ESC)

Wheel motors

Arduino Mega

Ultrasonic sensor



Figure 5: Robot motor step response to PWM signal

The battery on the robot provided an analogue DC voltage. The ESC eliminated the voltage-related loss of torque or efficiency by pulsing the input voltage on and off at a fixed duty cycle of 5 kHz. The vertical lines were results of the input voltage being switched off. The response curve also experienced the effect of the PWM signal, but overall, a step-response could be seen and the time constant as well as the system gain were extracted from the plot.

The motors were modeled as shown in **Figure 6**. The pulse-modulated input voltage is converted into torque via the rotor. The motor constants were calculated from the motor’s specification sheet where it listed the input voltage, no-load speed, and stall torque [1].

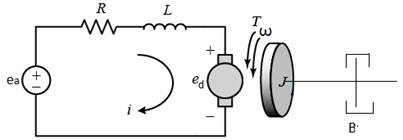


Figure 6: DC Motor electromechanical model

The motor system can be modeled with the block diagram shown in **Figure 7**, where is a conversion constant that converts the position error into an input voltage for the motor to generate a desired speed. The position conversion constant was experimentally determined.

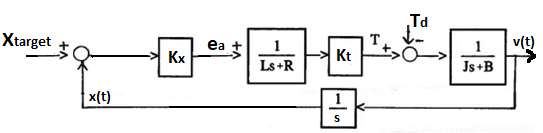


Figure 7: Motor system block diagram

The motor torque constant, , can be found using:

Equation 1

The resistance is measured with a digital multi-meter. Since two motors were soldered together, the measured resistance is the parallel equivalent resistance. Assuming the motors have the same resistance, using parallel resistance theory, it was determined that

Equation 2

The inductance of the motor is considered to be negligible for the analyses. The moment of inertia, however, cannot be ignored since it would also reflect the effect of the robot’s mass at 1.68 kg. The open-loop transfer function of the motor from the block diagram is:

Equation 3

From equation 3 the equations for gain, , and time constant, , were found to be:

Equation 4

Equation 5

## 1.3 Proportional-Integral-Derivative Control Theory

With the system parameters determined and experiment setup completed, the control algorithm now needed to be created to control the motor velocity of the system. The three control methods investigated in this experiment were proportional, integral, and derivative control (and combinations of PI and PID). The basis for each control method can be seen in equations 6 to 8 below [2]:

Equation 6

Equation 7

Equation 8

For proportional control, the difference between a desired set point velocity and the feedback velocity of the motor is considered. For integral control, the summation of error over a time period is considered, so the error will start to increase rapidly over time if the error is large and the control system does not see the input approaching the set point. Finally, for derivative control, the rate at which the feedback velocity is approaching the setpoint velocity is considered. For a full PID control, the sum of these three errors controls the response of the motor. Combining equations 6 to 8 gives:

Equation 9

Where , , and are the respective proportional, integral, and derivative gain values that tune the system response for different responses. Knowing these control algorithms, a block diagram of the system can be developed [3]:

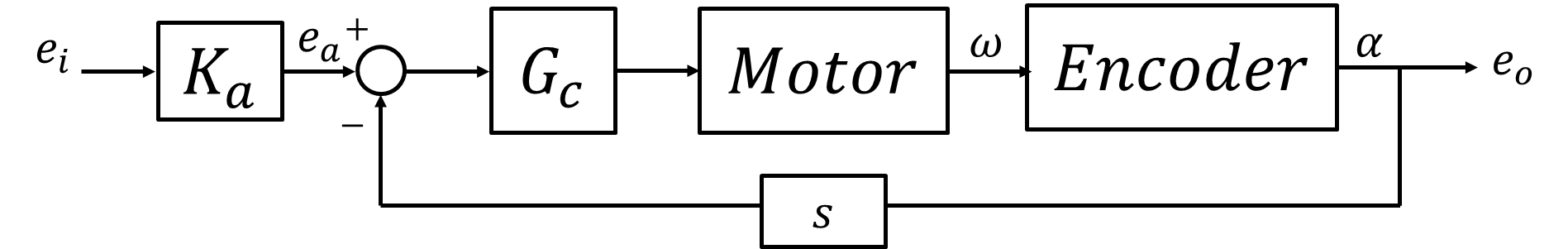


Figure 8: System block diagram for the auger motor

Where is the motor controller gain, and is the respective controller being used. Note that the encoder signal needed to be differentiated to obtain a velocity signal. Using this block diagram, a transfer function was derived that describes the complete PID control system:

Equation 10

Where and are the motor gain and motor system response, respectively. These values were going to be found experimentally. Once all values of the transfer function were determined, root loci of each control system were determined to better understand the range of system parameters and system responses.

The original setup was first tested, and the initial analysis started with the calibration of the pulse-width-modulation (PWM) input to the VEX motor controller. The motor controller specification sheet called for an input PWM signal of 1.5-2.0ms pulse widths, so this band was tested to discover the optimal operating range. Once the best operating range was found, an initial proportional control step input was examined. The Arduino was not able to accurately sample the data, and lousy data was obtained due to aliasing. Due to this, the Arduino microcontroller was switched to a VEX Robotics microcontroller, and testing resumed. This new platform operated on sending a power signal of -127 to 127 to the motor (with -127 being full reverse, 0 being stop, and 127 being full forward). A desired starting velocity was calibrated to a corresponding motor power input, and a mapping equation was created to proportionally convert future velocities to the correct, corresponding motor power:

Equation 11

The different control algorithms were then implemented into the system and tested against a set step impulse of a desired set point. P, I, PI, and PID control were all examined and compared against theoretical step responses. Resulting settling times were used to evaluate each system.

Furthermore, the effects of increasing the PID gain values are as follows [4]:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CL RESPONSE** | **RISE TIME** | **OVERSHOOT** | **SETTLING TIME** | **S-S ERROR** |
| **Kp** | Decrease | Increase | Small Change | Decrease |
| **Ki** | Decrease | Increase | Increase | Decrease |
| **Kd** | Small Change | Decrease | Decrease | No Change |

# 2 RESULTS AND DISCUSSION

## 2.1 Velocity Control of the Auger Motor

Analysis of the velocity controlled auger started with calibration of the DC motor with PWM input. **Figure 9** below shows the velocity of the motor (calculated using the encoder) versus the PWM input from the Arduino. The motor must overcome a stall torque to rotate and after a certain PWM input the motor is at its maximum velocity. From this data, step inputs were used from 184 to 250 rpm which is within the increasing linear region of the calibration plot.

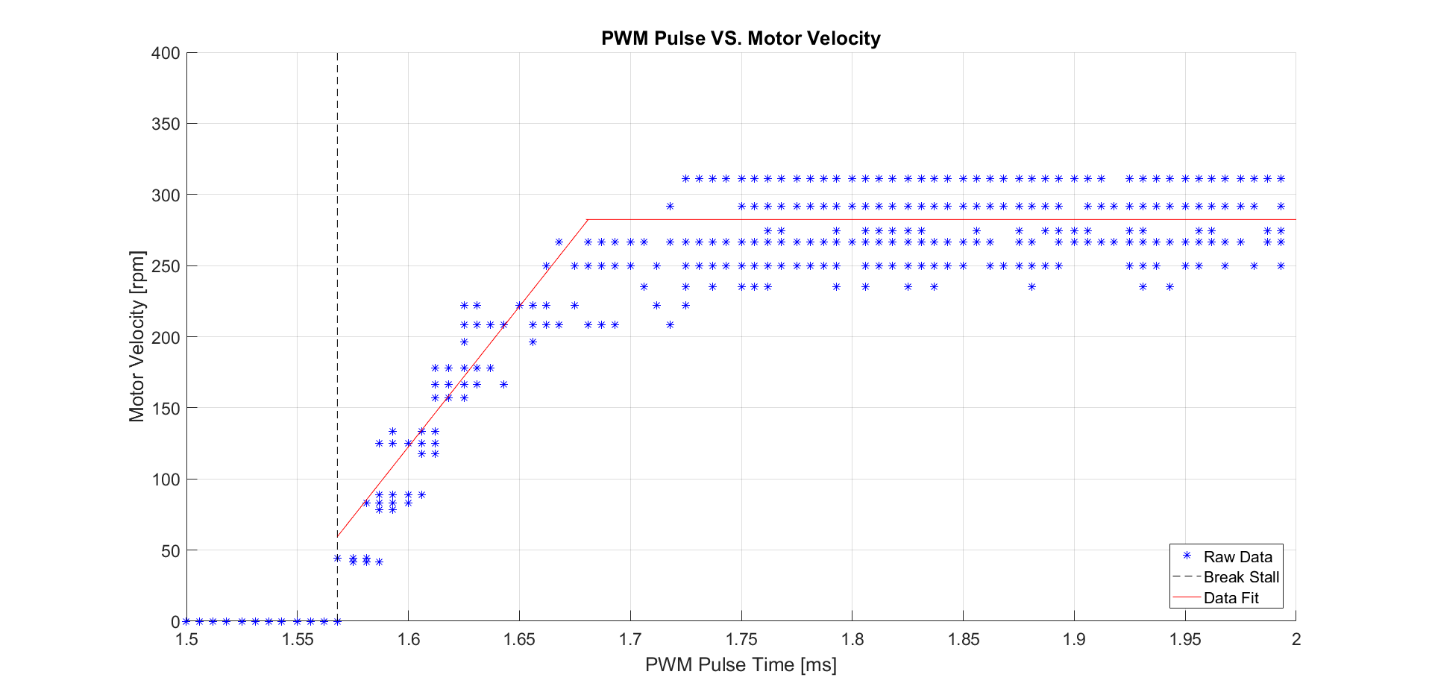


Figure 9: Calibration of DC Vex 393 motor with varying PWM inputs from Arduino microcontroller

Step inputs were initially done using an Arduino with PWM. The step response of the motor with proportional gain is shown in **Figure 10**. The PWM input combined with the Arduino’s inability to sample fast enough resulted in very noisy data. It was not feasible to analyze this response which led to using the Vex controller.

The motor was then analyzed with a step input to the Vex controller using proportional control. This resulted in **Figure 11**. This data was used to find the gain of the motor which is 0.7729. The time constant of the motor is 0.35s which was used for theoretical control analysis. The steady-state error of the motor with P-control is 16% of the target speed.

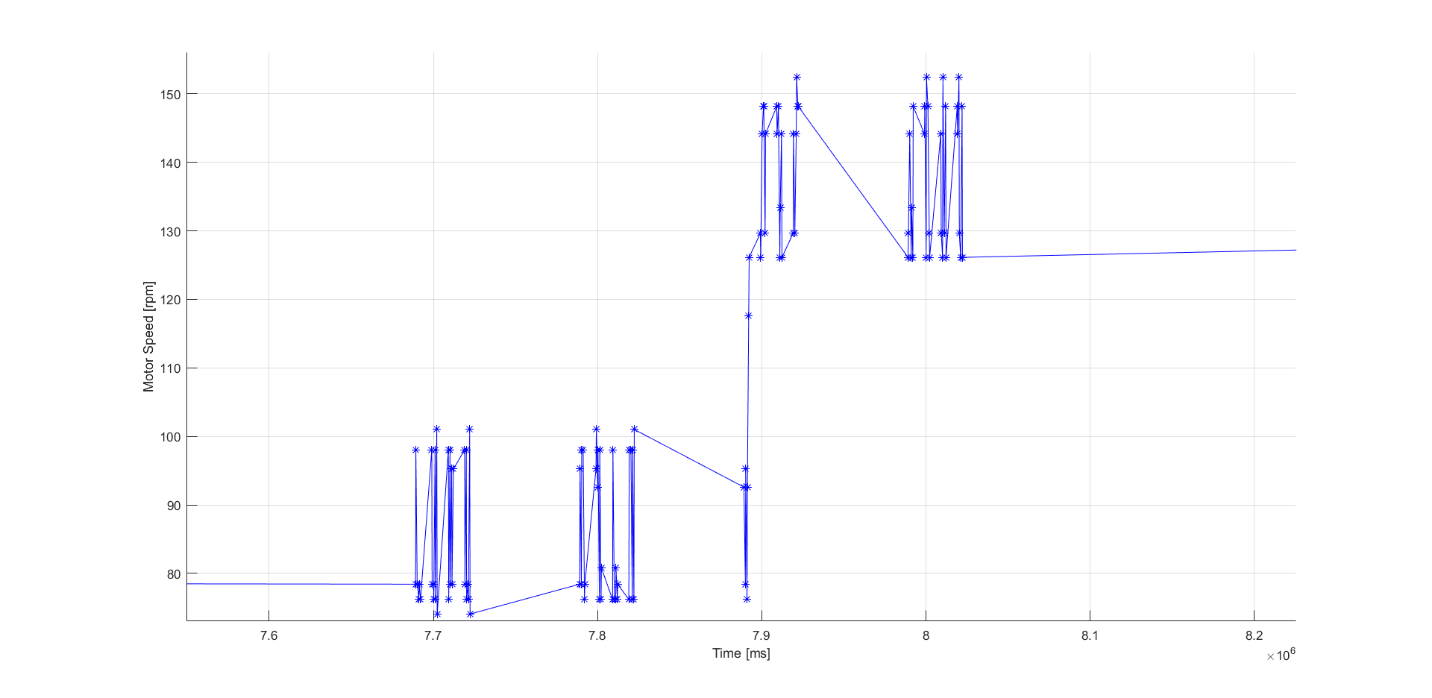


Figure 10: Velocity of motor with PWM step input

150

Time (ms)

Motor Speed (rpm)

140

x106

8.1

8.2

8.0

7.8

7.7

7.6

7.9

90

100

80

110

120

130



Figure 11: Step input with proportional gain of 0.5

The motor was tested with integral control shown in **Figure 12**. There was no steady-state error in the response, but the oscillations cause an increased settling time of approximately 5.6s. This is greater than the theoretical settling time of 3.1s with integral control.



Figure 12: Step input to motor with I-control



Figure 13: Step input to motor with PI-control

Proportional and integral control were implemented on the motor as shown in **Figure 13**. Proportional-Integral control had no steady-state error and a much faster settling time than the integral control at approximately 3.1s experimentally and 2.2s theoretically.

Derivative control was added to the system as shown in **Figure 14** for Proportional-Integral-Derivative control. The response had nearly no steady-state error and settled faster experimentally at approximately 2.4s. The theoretical response was very close to experimental with a settling time of 2.7s.



Figure 14: Step input to motor with PID-control

Root loci were also performed for the motor with each velocity control shown in **Figures 15 -** **18**. These root loci show how the gains can be altered to change the response of the system. The proportional control can be adjusted for a faster response, but this will change the steady-state error. The integral control cannot be adjusted for a faster response time. Both PID and PI control can be adjusted to respond faster but PID will be better assuming the system is accurate in calculating the derivative.



Figure 15: Root locus of motor with P-control



Figure 16: Root locus of motor with I-control.



Figure 17: Root locus of motor with PI-control



Figure 18: Root locus of motor with PID-control

## 2.2 Position Control of the Robot’s Velocity

In order to determine the system parameters of the robot the Arduino’s code was set to go at maximum speed, ignoring any obstacles sensed by the ultrasonic sensor. **Figure 19** below shows output voltage from the ESC PWM signal before it reaches the motors.



Figure 19: The resulting output voltage of the robot driving along the table

Using equations 1, 2, 4, and 5, with the gain and time constant obtained from **Figure 19**, the calculated system parameters were shown in **Table 2**:

Table : Another Brick in the Wall system parameters

|  |  |
| --- | --- |
| **Time Constant, τ (sec)** | 0.0244 |
| **System Gain, K** | 0.9719 |
| **Motor Resistance, R (Ω)** | 4.0000 |
| **Stall Torque, Td (oz-in)** | 31.2300 |
| **Torque Constant, Kt (oz-in/A)** | 20.8200 |
| **Motor Voltage Constant, Ke (V/rad/s)** | 0.1470 |
| **Damping, B (oz-in/(rad/s))** | 4.5905 |
| **Moment of Inertia, J (oz-in-s2)** | 0.1306 |

The robot’s motors were mounted in-place. Although they could be removed for testing, reassembly of the robot would be near-impossible due to time limitations and shortage of materials required for installing the components back together such as carbon fiber and special adhesives.

Using the system parameters above for the robot, the system can be simulated with Simulink. The Simulink model is displayed in figure (**INSERT FIGURE NUMBER OF SIMULINK MODEL HERE**).

# 3 CONCLUSIONS

## 3.1 Zhangxi ‘Jesse’ Feng

## 3.2 Michael Locke

Investigation of position PID control and velocity PID control in this experiment demonstrated the effectiveness of each control algorithm when subjected to a certain step input. The tested prototype platforms for each controller consisted of a four-wheeled robot chassis with an ultrasonic sensor for the positional PID controller and a fixtured motor with an integrated encoder for the velocity PID control. Feedback from the ultrasonic sensor and the integrated encoder sensor provided the foundation for the control system.

The measurements taken of the robot chassis distance using an ultrasonic sensor and output motor velocity using an encoder senor demonstrated the speed of response of each implemented system. This experiment is an excellent example of many real-world systems, such as distance to a part in an automated assembly process or speed response of cruise control. Whatever the system may be, the goal is all the same. There is always a desired value that needs to be obtained, and a system that needs to control the respective response.

The findings of this analysis demonstrated that one of the most important components to achieving a desired control response are the gain values of each controller (, , and ). For the PID position control, these gains may dictate whether the robot crashes into a wall or takes a very long time to reach its desired location. For PID velocity control, these gain values may dictate if the output velocity is completely unstable and is experiencing unrecoverable oscillations. Once the control algorithm is tuned however, the results demonstrate very fast and desirable response speeds. The results established, too, that the proportional-integration-derivative control demonstrated the fastest response due to the controller implementing three methods of error control.

Future studies should include additional analysis of PID control gain tuning. With the time constraints of this project, full efforts could not have been focused on tuning the control algorithm. This would improve the response overshoot and response time significantly, so when implemented on the final senior project robot, the performance would be exactly as desired.

## 3.3 Simon Popecki

The primary objectives of this experiment were to implement PID control in a real-world application. PID is often used to precisely control the approach of a machine or system to a target limit, whether it be the surface of Mars, a train station, or in this case- a wall. PID control systems can be used with inputs that are different from the output. In this case motor RPM was controlled with distance, and robot distance from the wall was controlled by the time between sonar pings from the ultrasonic sensor.

A methodical engineering approach was taken to create a solution to the objective of this experiment – a robot was modified for use with microprocessor control, software was written to make the robot move, and then that software was modified to control the robot with a prescribed control system. System parameters were obtained from spec sheets or testing, and a model was made to represent the robot mathematically. Once a model was made, a computer and modern system engineering methods were used to implement the most efficient control system possible. Analysis of the control system told the design team how effective the combination of software and materials was.

Improvements to the system could be made by making a more accurate model (measuring constants more precisely, taking measurements instead of relying on spec sheets), and by improving the electric and mechanical quality of the robot.

## 3.4 James Skinner

## 3.5 Matthew Westbrook

The goal of the analyzing the auger motor was to find its defining system parameters and implement velocity control to quickly reach the desired input. The initial use of an Arduino posed problems with PWM and a slow sampling time affecting the data. We were able to find the best range to operate the motor and then used a Vex controller for further analysis. Parameters of the motor were found which allowed for experimental and theoretical comparisons of velocity control.

From analysis of the motor with velocity control, PID was the best combination of response time and no steady-state error. To use PID control the system needs to be very faset and precise in its calculations of velocity. The initial setup with the Arduino would not have been reliable with PID because of noise in the system and slow sampling time. With the vex controller it made the system settle faster than using PI control alone.

In the future the system could be optimized to have a better response with PID control. Our data showed the PID performing better than other control methods but with proper optimization from the root locus it could be altered to have less overshoot and oscillations before settling to the steady state value.

# 4 REFERENCES

[1] RobotCombat, “Motors, Beetle B16 Gearmotor 16:1”, last accessed: December 15, 2017, from: <http://www.robotcombat.com/products/0-B16.html>

[2] Ogata, K., (2004), *System Dynamics,* 4th ed*.,* Upper Saddle River, NJ.: Person Prentice Hall.

[3] Figliola, R. and Beasley, D., (2011), *Theory and Design for Mechanical Measurements,*5th ed., Hoboken, NJ.: John Wiley & Sons.

[4] University of Michigan, “Introduction: PID Controller Design”, last accessed: December 15, 2017, from: <http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID>

# 5 APPENDIX

## A: Equipment List

* Station #2 – Kingsbury S221
* Arduino Uno
* Arduino Mega
* Sabertooth 2x12 ESC
* Beetle B16 gearmotors (Henkwell)
* Robot Chassis Platform with Motors
* Ultrasonic Sensor HC-SR04
* Vex Robotics Microcontroller
* VEX 393 Motor with Integrated Encoder Module
* NI Eight Slot Chassis (NI PXIe-1062Q)
* Control Board (NI PXIe-8360)
* Function Generator (NI PXI-5412)
* Digital Multimeter (NI PXI-4065)
* Digital Oscilloscope (NI PXI-5142)
* DC Power Supply (NI PXI-4110)
* DMM Soft Front Panel Software
* LabVIEW 2016
* MATLAB R2016B

## B: MatLab Codes

Auger motor Codes:

%% Final Project

clc; clear variables; close all

% control gain values (user input)

Kp = .5;

Ki = 37;% increasing this seems to match experimental results much better

Kd = 0.5;

%% P-control and Motor Parameters

% import experimental data

header = 1;

data\_test = csvread('P\_CONTROL\_TEST\_SLAB.csv',1,0);

time\_test = data\_test(:,2);

out\_test = data\_test(:,3);

smooth\_test = wsmooth1(out\_test,time\_test,.5);%smooth output(increase last number for more smoothing)

in\_test = data\_test(:,4);

data\_P = csvread('P\_Control\_VEX\_MRL.csv',1,0);

time\_P = data\_P(:,2);

out\_P = data\_P(:,3);

smooth\_P = wsmooth1(out\_P,time\_P,1);

in\_P = data\_P(:,4);

base\_in = find(in\_P>184,1);% find index of first step input up

end\_in = base\_in+find(in\_P(base\_in:end)<250,1);% find index ofstep back down

tau\_in = find(smooth\_P>smooth\_P(base\_in)+.632\*(smooth\_P(end\_in)-smooth\_P(base\_in)),1);% find index of time constant

tau\_P = time\_P(tau\_in)-time\_P(base\_in);% calculate time constant

K\_P = (smooth\_P(end\_in)-smooth\_P(base\_in))/(66);% gain of system with proportional gain

K\_motor = K\_P/Kp;% motor gain

Tau\_motor = tau\_P;% motor time constant

motor = tf([K\_motor],[Tau\_motor 1]);% motor transfer function

P\_sys = Kp\*motor;% P-control transfer function

Tf\_P = time\_P(end\_in)-time\_P(base\_in);% time constraint of simulation

[th\_v\_P,th\_t\_P] = step(66\*P\_sys,Tf\_P);% simulate P-control system

th\_t\_P = th\_t\_P+time\_P(base\_in);% theoretical output

th\_v\_P = th\_v\_P+smooth\_P(base\_in);

error = 100\*(max(in\_P)-max(smooth\_P))/max(in\_P);

%% I-Control

data\_I = csvread('I\_Control\_VEX\_MRL.csv',1,0);

time\_I = data\_I(:,2);

out\_I = data\_I(:,3);

smooth\_I = wsmooth1(out\_I,time\_I,1);

in\_I = data\_I(:,4);

base\_in = find(in\_I>184,1);

end\_in = base\_in+find(in\_I(base\_in:end)<250,1);

% take data from step down:

% -this came from the data only having oscillations on the step down,

% possibly from a force against the motor on step increases

% -done for I,PI and PID

% -data is flipped and shifted to step at t=0

start = base\_in+floor(.5\*(end\_in-base\_in));

new\_time\_I = time\_I(start:end)-time\_I(end\_in);

new\_smooth\_I = -smooth\_I(start:end)+smooth\_I(end\_in)+smooth\_I(base\_in);

new\_in\_I = -in\_I(start-2:end-2)+max(in\_I)+min(in\_I);

I = tf([Ki],[1 0]);

I\_sys = I\*motor;

sim('I\_model.slx');

th\_v\_I = I\_simout.data+smooth\_I(base\_in);

shift\_I = I\_simout.time(find(th\_v\_I>smooth\_I(base\_in),1));

th\_t\_I = I\_simout.time-shift\_I;

%% PI-Control

data\_PI = csvread('PI\_Control\_VEX\_MRL.csv',1,0);

time\_PI = data\_PI(:,2);

out\_PI = data\_PI(:,3);

smooth\_PI = wsmooth1(out\_PI,time\_PI,1);

in\_PI = data\_PI(:,4);

base\_in = find(in\_PI>184,1);

end\_in = base\_in+find(in\_PI(base\_in:end)<250,1);

start = base\_in+floor(.5\*(end\_in-base\_in));

new\_time\_PI = time\_PI(start:end)-time\_PI(end\_in);

new\_smooth\_PI = -smooth\_PI(start:end)+smooth\_PI(end\_in)+smooth\_PI(base\_in);

new\_in\_PI = -in\_PI(start-2:end-2)+max(in\_PI)+min(in\_PI);

PI = tf([Kp Ki],[1 0]);

PI\_sys = PI\*motor;

sim('PI\_model.slx');

th\_v\_PI = PI\_simout.data+smooth\_PI(base\_in);

shift\_PI = PI\_simout.time(find(th\_v\_PI>smooth\_PI(base\_in),1));

th\_t\_PI = PI\_simout.time-shift\_PI;

%% PID-Control

data\_PID = csvread('PID\_Control\_VEX\_MRL.csv',1,0);

time\_PID = data\_PID(:,2);

out\_PID = data\_PID(:,3);

smooth\_PID = wsmooth1(out\_PID,time\_PID,.5);

in\_PID = data\_PID(:,4);

base\_in = find(in\_PID>184,1);

end\_in = base\_in+find(in\_PID(base\_in:end)<250,1);

start = base\_in+floor(.5\*(end\_in-base\_in));

new\_time\_PID = time\_PID(start:end)-time\_PID(end\_in);

new\_smooth\_PID = -smooth\_PID(start:end)+smooth\_PID(end\_in)+smooth\_PID(base\_in);

new\_in\_PID = -in\_PID(start-2:end-2)+max(in\_PID)+min(in\_PID);

PID = tf([Kd Kp Ki],[1 0]);

PID\_sys = PID\*motor;

PID\_sys2 = tf([Ki\*Kp 0],[0 (Kp+0) (2\*Ki+Kp)])\*motor;

sim('PID\_model.slx');

th\_v\_PID = PID\_simout.data+smooth\_PID(base\_in);

shift\_PID = PID\_simout.time(find(th\_v\_PID>smooth\_PID(base\_in),1));

th\_t\_PID = PID\_simout.time-shift\_PID;

fit = (249.9811-183.681)/(3.0647+.01)\*linspace(-.01,3.0647,177)+183.681;

th\_v\_PID(100:276) = fit;

%% Plotting

figure (1)

plot(time\_P-4.0181,smooth\_P,time\_P-4.0181,in\_P,th\_t\_P-4.0181,th\_v\_P)

xlabel ('Time (s)')

ylabel ('Speed (rpm)')

title ('Motor with P-Control')

legend ('Output Signal','Target Signal','Theoretical')

figure (2)

plot(new\_time\_I,new\_smooth\_I,new\_time\_I,new\_in\_I,th\_t\_I,th\_v\_I)

xlabel ('Time (s)')

ylabel ('Speed (rpm)')

title ('Motor with I-Control')

legend ('Experimental','Target Signal','Theoretical')

figure (3)

plot(new\_time\_PI,new\_smooth\_PI,new\_time\_PI,new\_in\_PI,th\_t\_PI,th\_v\_PI)

xlabel ('Time (s)')

ylabel ('Speed (rpm)')

title ('Motor with PI-Control')

legend ('Output Signal','Target Signal','Theoretical')

figure (4)

plot(new\_time\_PID,new\_smooth\_PID,new\_time\_PID,new\_in\_PID,th\_t\_PID,th\_v\_PID)

xlabel ('Time (s)')

ylabel ('Speed (rpm)')

title ('Motor with PID-Control')

legend ('Output Signal','Target Signal','Theoretical')

figure (5)

rlocus(P\_sys)

title('Root Locus of Motor With P Velocity Control')

figure (6)

rlocus(I\_sys)

title('Root Locus of Motor With I Velocity Control')

figure (7)

rlocus(PI\_sys)

title('Root Locus of Motor With PI Velocity Control')

figure (8)

rlocus(PID\_sys)

title('Root Locus of Motor With PID Velocity Control')

function [zs,L] = wsmooth(z,x,y,L)

%WSMOOTH 1D and 2D robust smoothing.

% ZS = WSMOOTH(Z,X,L) smoothes the signal Z(X) using the smoothing

% parameter L. Z and X must be vector arrays of same length. L must be a

% negative or positive real scalar (the program actually uses 10^L). The

% larger L is, the smoother the output will be. Typical values for L are

% in the range [0,10].

%

% ZS = WSMOOTH(Z,X,Y,L) smoothes the surface Z(X,Y). X and Y must be

% vector arrays whose lengths match the size of matrix Z, i.e.:

% LENGTH(X) = SIZE(Z,2) and LENGTH(Y) = SIZE(Z,1)

%

% ZS = WSMOOTH(Z,L) uses unit spacings for X and/or Y.

%

% ZS = WSMOOTH(Z,X,Y) or ZS = WSMOOTH(Z,X) or ZS = WSMOOTH(Z): if L is

% omitted, it is automatically determined using the generalized cross-

% validation (GCV) method (see details in the program, line 260). Because

% eigenvalues are calculated, this may take a few seconds with large

% arrays. [ZS,L] = WSMOOTH(...) also returns the calculated value for L

% so that you can fine-tune the smoothing subsequently if wanted.

%

% Notes:

% -----

% WSMOOTH also works if some data are missing (see examples); data are

% considered missing if they are not finite (NaN or Inf). WSMOOTH can

% thus be used as an alternative to INPAINT\_NANS (a John D'Errico's

% function): Z = INPAINT\_NANS(ZNAN) and Z = WSMOOTH(ZNAN,L) with small L

% give nearly similar results. Smoothing level, however, can be adjusted

% with WSMOOTH if necessary.

%

% This program is based on the Whittaker's smoother. The 1D algorithm has

% been well described by Eilers in Anal. Chem. 2003, 75, 3631-3636. This

% algorithm has been here adapted to 2D. Note that a 3D update could be

% easily written (but beware the "out of memory" errors).

%

% Examples:

% --------

% % 1-D example

% x = 0:100;

% y = cos(x/10)+(x/50).^2;

% yn = y + 0.2\*randn(size(y));

% ys = wsmooth(yn,2);

% plot(x,yn,'.-',x,ys)

%

% % 1-D example with non uniform steps and automatic L selection

% x = sort(rand(1,1000)\*100);

% y = cos(x/10)+(x/50).^2;

% yn = y + 0.2\*randn(size(y));

% [ys,L] = wsmooth(yn,x);

% plot(x,yn,'.',x,ys,'r','Linewidth',2)

%

% % 2-D example

% [xp,yp] = deal(0:.02:1);

% [x,y] = meshgrid(xp,yp);

% f = exp(x+y) + sin((x-2\*y)\*3);

% fn = f + (rand(size(f))-0.5);

% fs = wsmooth(fn,2);

% subplot(121), surf(xp,yp,fn), zlim([0 8])

% subplot(122), surf(xp,yp,fs), zlim([0 8])

%

% % 2-D example with missing data and non-uniform grid

% xp = [0 sort(rand(1,48)) 1]; yp = [0 sort(rand(1,48)) 1];

% [x,y] = meshgrid(xp,yp);

% f = exp(x+y) + sin((x-2\*y)\*3);

% f(round(rand(1,100)\*2500)) = NaN; f(20:30,20:40) = NaN;

% fn = f + (rand(size(f))-0.5);

% fs = wsmooth(fn,xp,yp,2);

% subplot(121), surf(xp,yp,fn), zlim([0 8])

% subplot(122), surf(xp,yp,fs), zlim([0 8])

%

% % comparison with inpaint\_nans when filling NaN

% [x,y] = meshgrid(0:.02:1);

% [znan,z0] = deal(exp(x+y));

% znan(10:25,20:35) = NaN;

% znan(15:45,1:5) = NaN;

% znan(35:37,20:45) = NaN;

% z1 = inpaint\_nans(znan);

% z2 = wsmooth(znan,0);

% subplot(121), surf(znan), zlim([1 8])

% subplot(122), surf(z2), zlim([1 8]), title('wsmooth')

%

% See also SMOOTH, INPAINT\_NANS.

%

% -- Damien Garcia -- 2008/03

%% Check input arguments

% ---

if ndims(z)~=2

error('MATLAB:wsmooth:WrongZArray',...

'First input must be a vector or a matrix.')

end

siz0 = size(z);

if nargin==1 % wsmooth(z)

if isvector(z) % 1D

z = z(:)';

nx = length(z);

x = 1:nx; y = 1;

else % 2D

[ny,nx] = size(z);

x = 1:nx; y = 1:ny;

end

elseif nargin==2

if isscalar(x) && ~isvector(z) % wsmooth(z,L), 2D

L = x;

[ny,nx] = size(z);

x = 1:nx; y = 1:ny;

elseif isscalar(x) && isvector(z) % wsmooth(z,L), 1D

L = x;

z = z(:)';

nx = length(z);

x = 1:nx; y = 1;

elseif isvector(x) && isvector(z) % wsmooth(z,x), 1D

z = z(:)';

y = 1;

else

error('MATLAB:wsmooth:WrongInputs',...

'Wrong input arguments: type ''help wsmooth''.')

end

elseif nargin==3

if isvector(z) && isvector(x) && isscalar(y) % wsmooth(z,x,L), 1D

L = y;

z = z(:)';

y = 1;

elseif ~isvector(z) && isvector(x) && isvector(y) % wsmooth(z,x,y), 2D

% Nothing to declare for the moment

else

error('MATLAB:wsmooth:WrongInputs',...

'Wrong input arguments: type ''help wsmooth''.')

end

else % wsmooth(z,x,y,L), 2D

if isvector(z) || ~isvector(x) || ~isvector(y) || ~isscalar(L)

error('MATLAB:wsmooth:WrongInputs',...

'Wrong input arguments: type ''help wsmooth''.')

end

end

% Check size matching and monotonic properties

[ny,nx] = size(z);

if ~isequal(length(x),nx) || ~isequal(length(y),ny)

error('MATLAB:wsmooth:XYLengthMismatch',...

'Number of elements in X and/or Y is inadequate.')

elseif any(sign(diff(x))~=1) && any(sign(diff(x))~=-1)

error('MATLAB:wsmooth:NonMonotonicX',...

'X must be strictly monotonic.')

elseif ~isequal(y,1) && any(sign(diff(y))~=1) && any(sign(diff(y))~=-1)

error('MATLAB:wsmooth:NonMonotonicY',...

'Y must be strictly monotonic.')

end

% Nothing to do if dims(z)<3

if (isequal(y,1) && length(z)<3) || (~isequal(y,1) && any([nx ny]<3))

zs = reshape(z,siz0);

return

end

x = x(:);

y = y(:);

class0 = class(z);

z = double(z);

% Warning messages may appear in some atypical cases

warn0 = warning('query','MATLAB:nearlySingularMatrix');

warning('off','MATLAB:nearlySingularMatrix')

%% Create the difference matrix

%--

% 1D

if isequal(y,1)

% Create the X-difference matrix

% FDM: 2nd derivative, X-central difference, 2nd order

h0 = x(2:end-1)-x(1:end-2);

h1 = x(3:end)-x(2:end-1);

D = spdiags([2./h0./(h0+h1) -2./h0./h1 2./h1./(h0+h1)],...

[0 1 2],nx-2,nx)\*mean(h0.^2);

D = D'\*D;

%--

% 2D

else

% -----

% The Whittaker's method is adapted to 2D. We solve:

% (I + lambda(Dx' Dx + Dy' Dy))zs = z,

% where Dx and Dy are the difference matrices related to x- and y-

% direction, respectively.

% -----

% Create the X-Difference matrix

% FDM: 2nd derivative, X-central difference, 2nd order

n = (nx-2)\*ny;

[Jd,Dx] = deal(zeros(3,n));

zI = true(ny,nx);

zI(:,[1 nx]) = false;

[I,J] = find(zI);

Jd(1,:) = I+(J-2)\*ny;

Jd(2,:) = I+(J-1)\*ny;

Jd(3,:) = I+J\*ny;

h0 = x(J)-x(J-1);

h1 = x(J+1)-x(J);

Dx(1,:) = 2./h0./(h0+h1);

Dx(2,:) = -2./h0./h1;

Dx(3,:) = 2./h1./(h0+h1);

Id = repmat(1:n,[3 1]);

D = sparse(Id,Jd,Dx,n,nx\*ny)\*mean(h0.^2);

clear Dx

D = D'\*D;

% Create the Y-Difference matrix

% FDM: 2nd derivative, Y-central difference, 2nd order

n = nx\*(ny-2);

[Jd,Dy] = deal(zeros(3,n));

zI = true(ny,nx);

zI([1 ny],:) = false;

[I,J] = find(zI);

Jd(1,:) = I-1+(J-1)\*ny;

Jd(2,:) = I+(J-1)\*ny;

Jd(3,:) = I+1+(J-1)\*ny;

h0 = y(I)-y(I-1);

h1 = y(I+1)-y(I);

Dy(1,:) = 2./h0./(h0+h1);

Dy(2,:) = -2./h0./h1;

Dy(3,:) = 2./h1./(h0+h1);

Id = repmat(1:n,[3 1]);

Dy = sparse(Id,Jd,Dy,n,nx\*ny)\*mean(h0.^2);

clear I\* J\* h\*

D = D + Dy'\*Dy;

clear Dy

end

%% Weight function (0 for NaN/Inf data, 1 otherwise)

% --

% NaN or Inf data are considered missing data

I = isfinite(z);

z(~I) = 0;

W = spdiags(double(I(:)),0,nx\*ny,nx\*ny);

%% Solve the linear system

if exist('L','var')

% L parameter is given

% --------------------

zs = (W + 10^L\*D)\z(:);

else

% Automatic determination of L parameter

% --------------------------------------

% The generalized cross-validation (GCV) method is used. We determine L

% that minimizes the GCV score. If zs = H\*z (with z containing N

% elements) then the GCV score is written as:

% GCVs = 1/N\*sum((zs-z).^2) / (1 - Tr(H)/N)^2

% Tr(H) (trace of H) is required. Here, H = (I + L\*D)^(-1). I+L\*D is

% hermitian positive semi-definite, I+L\*D can be thus decomposed as

% I+L\*D = U\*S\*U' with U orthogonal and, S diagonal and containing the

% eigenvalues. The eigenvalue decomposition gives: H = U\*S^(-1)\*U^(-1).

% Finally: Tr(H) = Tr(inv(S)) = sum(diag(1./S)) = sum(1./eig(I+L\*D)),

% with eig(I+L\*D) = 1+L\*eig(D)

% Note: the weight function is not taken into account in order to

% simplify the algorithm and make it faster. This does not change the

% results significantly.

eigD = eig(D); % this may take a couple of seconds if D is large

L = fminsearch(@(L) gcv(L),1,...

optimset('TolX',0.1,'Display','off','Maxiter',50));

L = round(L\*10)/10;

zs = (W + 10^L\*D)\z(:);

end

zs = reshape(zs,siz0);

zs = cast(zs,class0);

warning(warn0)

function score = gcv(L)

% GCV score as a function of L

L = L(1);

trH = sum(1./(1+10^L\*eigD));

zs = (W + 10^L\*D)\z(:);

score = sum((zs(:)-z(:)).^2)/nx/ny/(1-trH/nx/ny)^2;

end

end

% James Skinner

% ME 747

% Final Project

close all; clear all;

%% Experimental data With Load Full Speed

nheaderlines = 31;

dataexp = importdata('Motors on table with wires.lvm','\t',nheaderlines);

time = dataexp.data(:,1);

time = time(3146:end);

time = time-time(1);

volt = dataexp.data(:,2);

volt = volt(3146:end);

voltnew = volt;

timenew = time;

j = 1;

for i = 1:length(time)

holder = volt(i);

if (holder>(-6) || holder<(-7.27))

indexes(j) = i;

j = j+1;

end

end

voltnew(indexes) = [];

timenew(indexes) = [];

voltnew = voltnew(1:577);

voltnew(218) = [];

voltnew(411) = [];

voltnew(429) = [];

timenew = timenew(1:577);

timenew(218) = [];

timenew(411) = [];

timenew(429) = [];

Vtau = voltnew(1)+0.632\*(voltnew(end)-voltnew(1));

indexVtau = find(voltnew>Vtau-.02 & voltnew<Vtau+.02);

tau = mean(time(indexVtau));

figure

plot(timenew,voltnew,tau,Vtau,'ro')

xlabel('Time (sec)')

ylabel('Voltage (V)')

title('Output Voltage With Load at Full speed')

legend('Experimental Data','Time Constant, \tau')

set(gca,'fontname','Times','fontsize',12)

xlim([0 0.4])

grid on

box on

voltsmooth = smooth(voltnew);

figure

plot(timenew,voltsmooth,tau,Vtau,'ro')

xlabel('Time (sec)')

ylabel('Voltage (V)')

title('Output Voltage With Load at Full Speed')

legend('Smoothed Experimental Data','Time Constant, \tau')

set(gca,'fontname','Times','fontsize',12)

xlim([0 0.4])

grid on

box on

ei = 7.41; % Input Voltage as measured from battery

voltss = voltnew(end); % Steady State Voltage

K = abs(voltss)/ei; % Gain

R = 4; % ohm, measured 8 ohm from 2 motors

Td = interp1([6,12],[27,45],7.2); % oz-in, stall torque for 7.2V input

Kt = Td\*R/6; % oz-in/A

Ke = Kt/141.6119; % (V/rad/s)

B = ((Kt/K)-Ke\*Kt)/R; % oz-in/rad/s

J = (tau\*(R\*B + Ke\*Kt))/R; % oz-in-s^2

% PID Controller constants

Kp = 75; % 295.5; % P-Control

Ki = 35\*40; % 5162; % I-Control

Kd = 1; % 2.951; % D-Control

Target = 10; % Distance from the wall to stop

%% Experimental data with load half speed

clear all;

nheaderlines = 31;

dataexp = importdata('Motors on table with wires slower.lvm','\t',nheaderlines);

time = dataexp.data(:,1);

time = time(1837:2700);

time = time-time(1);

volt = dataexp.data(:,2);

volt = volt(1837:2700);

voltnew = volt;

timenew = time;

j = 1;

for i = 1:length(time)

holder = volt(i);

if (holder>(-6.99) || holder<(-7.32))

indexes(j) = i;

j = j+1;

end

end

voltnew(indexes) = [];

timenew(indexes) = [];

voltnew(193) = [];

voltnew(276) = [];

voltnew(344) = [];

timenew(193) = [];

timenew(276) = [];

timenew(344) = [];

Vtau = voltnew(1)+0.632\*(voltnew(end)-voltnew(1));

indexVtau = find(voltnew>Vtau-.004 & voltnew<Vtau+.004);

tau = mean(time(indexVtau));

figure

plot(timenew,voltnew,tau,Vtau,'ro')

xlabel('Time (sec)')

ylabel('Voltage (V)')

title('Output Voltage With Load at Half Speed')

legend('Experimental Data','Time Constant, \tau')

set(gca,'fontname','Times','fontsize',12)

xlim([0 timenew(end)])

grid on

box on

voltsmooth = smooth(voltnew);

figure

plot(timenew,voltsmooth,tau,Vtau,'ro')

xlabel('Time (sec)')

ylabel('Voltage (V)')

title('Output Voltage With Load at Half Speed')

legend('Smoothed Experimental Data','Time Constant, \tau')

set(gca,'fontname','Times','fontsize',12)

xlim([0 timenew(end)])

grid on

box on

ei = 7.41; % Input Voltage as measured from battery

voltss = voltnew(end); % Steady State Voltage

K = abs(voltss)/ei; % Gain

R = 4; % ohm, measured 8 ohm from 2 motors

Td = interp1([6,12],[27,45],7.2); % oz-in, stall torque for 7.2V input

Kt = Td\*R/6; % oz-in/A

Ke = Kt/141.6119; % (V/rad/s)

B = ((Kt/K)-Ke\*Kt)/R; % oz-in/rad/s

J = (tau\*(R\*B + Ke\*Kt))/R; % oz-in-s^2

%% Originall plot

clear all;

nheaderlines = 31;

dataexp = importdata('Motors on table with wires slower.lvm','\t',nheaderlines);

figure;

hold on;

plot(dataexp.data(1800:2500,1),dataexp.data(1800:2500,2));

xlabel('Time (s)','FontSize',12);

ylabel('Voltage (V)','FontSize',12);

ylim([-8 -6]);