2022

Integrating Experimental Parameter Space for Granular Flow within Polygon Tumblers

Michaela Mcmahon
michaela.j.mcmahon@gmail.com

ABSTRACT

INTEGRATING EXPERIMENTAL PARAMETER SPACE FOR GRANULAR FLOW WITHIN POLYGON TUMBLERS

Michaela McMahon, MS
Department of Mechanical Engineering
Northern Illinois University, 2022
Dr. Nicholas Pohlman, Director

Granular flows are complex phenomena bridging both fluid and solid mechanics and observed frequently in the natural world. The transitional dynamics of acceleration from solid body motion to rapid flow can affect many aspects including volume flow rates and mixing behaviors. In order to study the combination of both variables together within polygon-shaped tumblers, control systems are necessary with a corresponding user interface. This thesis reports an interface for changing variable inputs to the system. Predictions of rotation rates with corresponding interfaces with hardware connections are described. Results of image analysis verify output performance that incorporates the following variable changes: 1. Number of sides of the polygon, 2. Number of oscillations between vertices, 3. Phase of acceleration or deceleration at start, 4. Long-time performance in time-based digital data acquisition controls. The mechanisms for operation offer suggestions on the next improvements and preparing the software and hardware for future experiments in granular segregation and velocity measurements.

NORTHERN ILLINOIS UNIVERSITY
DE KALB, ILLINOIS

AUGUST 2022

INTEGRATING EXPERIMENTAL PARAMETER SPACE FOR GRANULAR FLOW WITHIN POLYGON TUMBLERS

BY

MICHAELA MCMAHON

A THESIS SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE
MASTER OF SCIENCE

DEPARTMENT OF MECHANICAL ENGINEERING

Thesis Director:
Dr. Nicholas Pohlman

ACKNOWLEDGEMENTS

I would like to first acknowledge and thank my thesis director, Dr. Nicholas Pohlman, for his supervision, expertise, and enthusiasm during my thesis process. Thank you for the countless hours of discussion and problem solving. Thank you for your patience with me and your encouragement of my hard work.

I would also like to thank my committee members, Dr. Schait Butail and Dr. Jifu Tan, for their support, their particular expertise, and their willingness to participate in my thesis as committee members.

I extend a special thanks to Mike Reynolds and NIU Machine Shop for their assistance with the fabrication of physical testing components.

I would like to thank the College of Engineering and Engineering Technology at Northern Illinois University as a whole, along with all its faculty and staff members who have helped make the education I received possible and who have helped ignite and encourage my passion and love for engineering.

My dearest thanks to the Newman Catholic Student Center and my community there for its support academically, professionally, and personally. A particular thanks to Fr. Robert Gonnella IV and Aaron Beck for their mathematical and technical collaboration on this project and to Rebekah Reynolds and Jordan Huff for their accompaniment and direction with filming equipment set up.

To my family and friends, thank you for all your support, encouragement, and prayers as I have worked to accomplish my academic goals. It was your confidence in me, your encouragement of me, and your excitement for me that enabled me to continue through frustration time and time again and achieve all that I have to date.

DEDICATION

To the Blessed Virgin Mary, to the love of my life, Jesus Christ, and to Michael J. McPheters, without all of whom I would not be the woman I am today.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

Fluid flow is an important field of study. Liquids and gasses are present in every day life from human blood flow to a boiling kettle. Fluid flow has been long studied and contains many sub fields. One such sub-field is called granular flow. Granular flow is flow in which discrete particles move relative to one another with fluid-like motions. Granular flows are ubiquitous and exist in both the natural and industrial worlds [3]. In nature they are seen in things like moving sand, avalanches and deposition of soil [2]. Some industrial examples include corn or grain silos, transportation of coal dust, and mixing [4].

System repeatability is necessary when studying granular flow in tilted chutes, exit area of silos, or rotating tumblers [5]. Rotating tumblers, drums with axial rotation, are of particular interest. A benefit of rotating tumblers is their capacity for continuous operation without the need to replenish granular source. Additionally they are predictable and repeatable systems making them ideal for study. Rotating tumblers have many industrial applications such as mixing, polishing and refining [3] [4]. There has been considerable study of the flow patterns found in rotating tumblers as a result of accumulated segregation of small and large particles. Within tumblers different flow regimes exist. There are four general kinds of flow: avalanching, continuous, cataracting, and centrifuging [6].

The four kinds of flow occur at different rotation rates. At low rotation rates particles build up along one edge of the tumbler and then cascade down suddenly. This is avalanching flow. As rotation rate is increased, the time between cascades decreases until the flow eventually becomes continuous. During continuous flow the free surface length and angle of repose of flowing layer are relatively constant. If the rotation rate is increased further,

the free surface becomes S-shaped and cataracting flow is present due to material launching from the higher rotation rate. At high enough rotation rates, the particles in the tumbler will centrifuge and are expelled to the sides of the drum where they experience solid body rotation. [6]

A challenge of granular materials is their opaque appearance. It is impossible to view internal dynamics without high power 3D imaging equipment [7]. 3D tumblers have been studied using magnetic resonance imaging (MRI); however, due to high equipment cost simplifications are often made [8] [3]. To simplify 3D tumblers and eliminate the need for MRI imaging, quasi-2D tumblers are used. A quasi-2D tumbler has a depth which is small relative to the tumbler diameter. Quasi-2D tumblers can be used to study the end effects of 3D tumblers as well [3]. In this study, the experimental set up was designed for quasi-2D tumblers.

Segregation is an important characteristic in granular flow. It can occur when there are particles of different shape, size, or density. Segregation works against mixing which is why it is important and highly studied [2]. It is desirable to understand the granular flow dynamics of segregation. There are two main approaches which have been taken when studying segregation in quasi-2D tumblers.

The first was to study systems with constant free surface length and periodically varying rotation rates. Radial segregation with lobe formation was seen. The number of lobes corresponded to the acceleration cycles per revolution. In figure 1.1 a result is shown for a half filled tumbler and eight acceleration cycles per revolution. Four lobes formed in the half filled circular tumbler corresponding to the eight acceleration cycles per revolution [1].

Figure 1.1: Segregation of particles in half filled circular tumbler with eight acceleration cycles per revolution [1]

The second explored a changing free surface length, L, via polygon tumblers. As a polygon tumbler rotates at a constant speed the free surface changes as a function of tumbler orientation. Stable and time independent segregation patterns were observed. Prediction of segregation patters was possible with simple models of linear velocity profiles [9] and observed patterns were dependent on fill level and tumbler shape which directly affect the repeatability frequency of the flowing layer [2]. Figure 1.2 shows the development of segregation lobes formed in two different shaped tumblers.

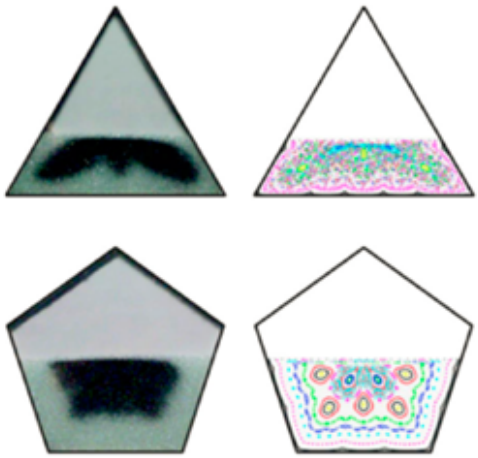

Figure 1.2: Experiments (left image) of segregation of small black particles compared to simulations (right images) of repeated Poincare sections of the simple velocity profile [2]

All of the prior research in granular tumblers relied on steady-state operations or variations of one of the driving factors of free surface flow: length dimension and rotation rate.

The repetition of either one of those variables induced the lobed segregation that makes granular flows unique. The goal of this research is to build an experimental foundation on which will begin exploration of the combined effects of both changing shape and varying rotation rate within granular flow. The goal is to create a repeatable system for flow observation under a combination of factors. Described in this thesis will be configurable for varying polygon shapes and rotation rates as a function of all input parameters of a generalized periodic function. The system will have a user interface which will allow for the set-up of user desired specifications making an easier transition for undergraduate students. The interface will be combined with a signal controller which will vary the rotation rate as a function of orientation.

Chapter 2 describes the mathematical functions of the periodic system as related to tumbler shape. Chapter 3 then explores hardware and signal controller set up. Chapter 4 goes on to describe the verification process. Results comparing the input performance to a example shape are reported in Chapter 5. The opportunities for future directions is suggested in the conclusions of Chapter 6.

CHAPTER 2

THEORY

2.1 Angular Velocity as a Function of Edge Features

Rotation rate or angular velocity, ω, is specified by the user as a function of orientation of the tumbler, θ. Angular velocity takes the following form:

$$\omega(\theta) = \omega_{ss} + \frac{\omega_{max} + \omega_{min}}{2} \sin(f\theta - \phi) \tag{2.1}$$

ω_{ss} is the average angular velocity which will be experienced by the tumbler. The maximum and minimum values for angular velocity are determined by the amplitude of the function. Amplitude is user specified value and will determine the extrema for the configuration. The frequency, f, and phase shift, ϕ, of the sine function are determined by a combination of user inputs including the number of polygon sides, N, number of cycles per side length, F, and cycle offset, P, that is specified as a percent of the period of the wave form. Frequency is calculated by equation 2.2.

$$f = 2\pi \frac{F}{L} \tag{2.2}$$

Here side length, L, was calculated as a function of vertex locations and was dependent on N. The actual dimension of L was not important, rather the degrees of rotation per side that was key. The first vertex for any N-sided polygon was placed at the point $(1, 0)$. The subsequent vertices were also at a radial length of 1, but rotated the number of degrees per N side.

Phase shift is calculated by equation 2.3.

$$\phi = 2\pi \frac{P}{F} \tag{2.3}$$

While phase of argument in equation 2.1 is in radians, there is an intuitive nature of P given in equation 2.3. When P is zero, there is no offset meaning the rotation rate, as the initial vertex passes through the point $(1, 0)$, will be ω_{ss}. When P is 0.25, angular velocity will start at its maximum rotation rate (at the peak of the sine curve) and experience negative acceleration. In the case that P is 0.75, the system begins at its minimum rotation rate and accelerates out of the gate. Selecting any P value can determine whether the flowing layer dimension would be affected by acceleration or deceleration of ω when at longest lengths of the free surface.

Combining equations 2.1, 2.2, and 2.3 into one, the following equation culminates.

$$\omega(\theta) = \omega_{ss} + A \sin\left(\frac{2\pi F(\theta - \frac{LP}{F})}{L}\right) \tag{2.4}$$

where all the coefficients beside θ serve as user inputs. It is used as a basis for visualization graphing within the GUI, figure 2.1, as well as the driving signal control function.

2.2 Angular Velocity as Function of Time

The system developed was a feed-forward system which is controlled based on timers available in the computing and digital data acquisition hardware. While it is desirable to be able to run tests and think about angular velocity as a function of orientation and edge features, controlling the system by θ would require integration of a rotary encoder, which

was beyond the purpose of the present research. Because the system is controlled based on time, equation 2.4 needed to be converted to an equation dependent on time.

Equation 2.5 shows the time dependent form of ω.

$$\omega(t) = \omega_{ss} + A\sin(2\pi(f_{time}t - P)) \tag{2.5}$$

In equation 2.5 ω_{ss}, A and P are direct user inputs. F, specified by the user in cycles per side, needed to be converted to units of Hz. Frequency in Hz is described in equation 2.6.

$$f_{time} = \left(\frac{F\ cycles}{side}\right)\left(\frac{N\ sides}{rev}\right)\left(\frac{\omega_{ss}\ rev}{min}\right)\left(\frac{1min}{60s}\right) \tag{2.6}$$

Equation 2.6 was formulated by unit analysis. The time it takes for rotate through one side of any give polygon at an average rotation rate of ω_{ss} was calculated in seconds and then multiplied by F which is in units of $\frac{cycles}{side}$ yielding equation 2.6.

Equation 2.5 describes ω as a function of time and was a driving equation within the signal controller. It allowed for tests to be run for a certain specified runtime, t_{run}.

2.3 GUI Operation and Formulation

The complexities of angular velocity control need to be encapsulated into a clean and simple package for future researchers, especially undergraduate students. A graphical user interface was developed to take user the specified values which make up equation 2.4 and drive a rotating motor based on the specified configuration. The GUI provides visualization for the driven signal, which will allow future researchers easy access understating the experimental domain.

2.3.1 Operation

The user interface was developed using MATLAB's *uifigure* functions. The *uifigure* is a dynamic environment which updates according to user inputs. It can be seen in figure 2.1. The GUI contains six user edit fields and four push buttons. It takes inputs for ω_{ss}, N, F, P, A, and t_{run}.

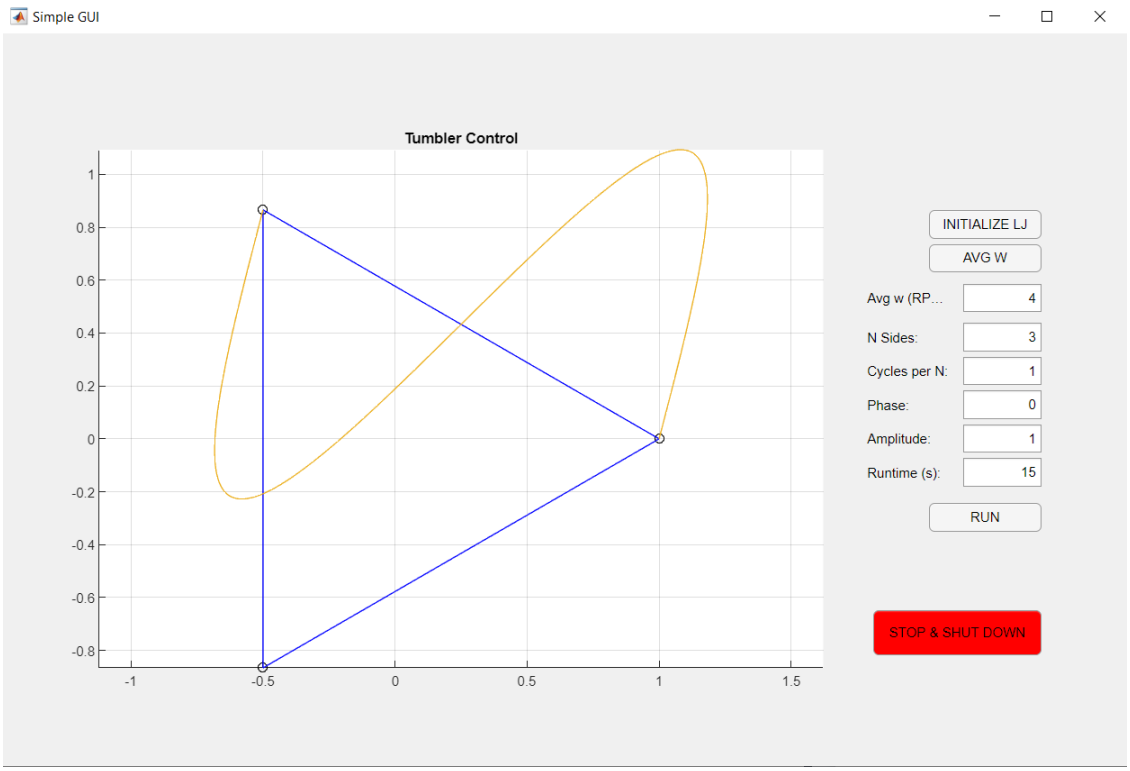

Figure 2.1: Graphical User Interface after initialization in MATLAB, blue lines show the shape of N-sided polygon, orange sinasoidal wave represents ω as a particular point passes through a reference

The axis area shows a visualization of the specified set up. The user enters values by typing the desired number and then either pressing enter or clicking out of the edit field. Once values are entered, the graph updates to reflect changes. Inputs for N, F, P, and A affect the graph. It is of note that, although the starting vertex always remains at graph

location (1,0), the user may chose to start the physical tumbler at any starting orientation. The rest of the visualization is displayed based on the fixed vertex. Inputs for ω_{ss} and t_{run} do not change the graph but they are specified like the others, by typing the desired value and pressing enter or clicking away from the edit field. Values for A and ω_{ss} are specified in RPM and values for t_{run} are specified in seconds.

The four push buttons were located on the GUI. The *INITIALIZE LJ* button finds and sets up the LabJack signal generator (described in greater detail in Chapter 3). The user must initialize the LabJack before running any control function. The *AVG* ω button sends a signal which turns the tumbler at the specified average value. The signal continues until it is manually stopped via the *STOP AND SHUTDOWN* button. The *STOP AND SHUTDOWN* button will shut off the signal and disable DAQ hardware. The *RUN* button runs the user specified varying velocity configuration for the duration of the user specified runtime. It is important to note that the *RUN* setting runs until the runtime is reached and will not be stopped by the *STOP AND SHUTDOWN*. In the event of emergency, the system can be shutdown via the power strip source.

2.3.2 Code Formulation

The GUI pop up was created in MATLAB as a dynamic $uifigure$. Upon initial launch, initial values for all inputs are set and an initial graph is populated. The process flow diagram is as shown in figure 2.2.

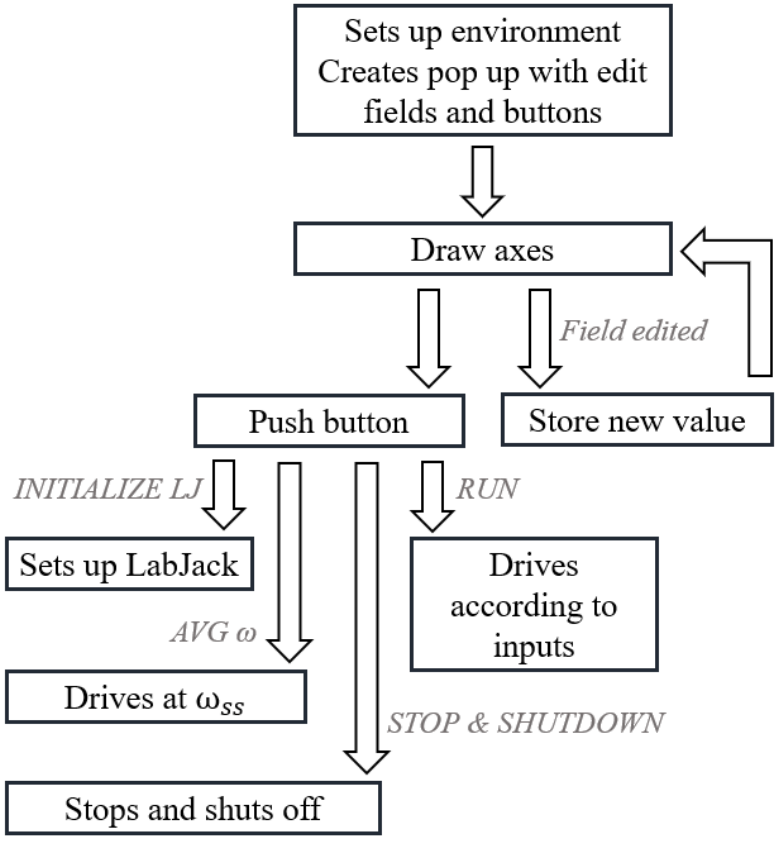

Figure 2.2: Process Flow of Developed GUI code in MATLAB

2.3.2.1 The Graphing Function

A dominating feature of the GUI is the graph which provides a visualization of any given set up. First the polygon needed to be created for any given N value. Coordinates for vertex locations were determined by fixing the first point at $(0, 1)$ then rotating around, placing a new point every $360/N$ degrees at a unit length 1. Since orientation does not scale with tumbler dimension, this unit value is sufficient. The vertices were then plotted to create the polygons. Side length, L, of any given polygon was calculated based on the distance formula between a pair of vertex coordinates.

The orange sinusoidal wave plotted across the edge of was formulated as the visualization of angular velocity as a function of tumbler orientation (e.g. which portion of the polygon is passing through the horizontal x-axis). First equation 2.7 was created based on user inputs. It was then rotated to be graphed along the length of a polygon side.

$$\omega = -A \sin\left(\frac{2\pi F(\theta - \frac{PL}{F})}{L}\right) \tag{2.7}$$

The rotated equation for ω was plotted starting at the fixed vertex and was plotted back across the side length. It starts at the vertex and then changes along the side length as if the polygon were rotating through the angle θ.

2.3.2.2 Edit Field Callback Functions

Each user input edit field is accompanied with a corresponding callback function within the MATLAB code. When a value is changed and entered, the callback rewrites the value for that particular quantity. The callback functions for number of sides, cycles per side, phase, and amplitude also call the *draw* function after the value is reset. Recalling the *draw* function updates the graph.

Callback functions for amplitude and average angular velocity have an additional process. Negative ω values are not allowed in the system meaning the tumbler must always be rotating in the clockwise direction. Because of this condition, the average angular velocity must always be larger than the amplitude. When the amplitude is reset, the callback checks to see if the value is greater or smaller than the current value for average angular velocity. If the value for amplitude is smaller the average angular velocity is updated to a value of $A + 0.3$. Similarly, when average angular velocity is resent, its callback checks to see if it is smaller or

larger than the current amplitude value. If the value for average angular velocity is smaller, the callback rewrites it to a value of $A + 0.3$.

2.3.2.3 Push-button Action Functions

The *INITLIZE LJ* must be the first button action function run. The *INITIALIZE LJ* function serves to find and configure the LabJack signal controller within the MATLAB script. The LabJack .NET is made visible in MATLAB and then necessary constants are set for MATLAB–LabJack communication.

After the LabJack is initilized, the *AVG* ω function can be run. The *AVG* ω function sends a signal with drives the system at the user specified value, ω_{ss}. The *STOP AND SHUT DOWN* button runs a shutdown sequence which first turns off the micro-stepper and then shuts off the signal from the LabJack.

The *RUN* button pushes the varying angular velocity configuration specified by the user. It does so by converting angular velocity from a function of θ to a function of time as previously described in equation 2.5. Figure 2.3 shows an overview of the function process flow.

MATLAB outputs the varying square wave signal at discrete time steps which are some δt apart. The value of δt was set to a value of $0.1s$. In order to track time, MATLAB's *tic toc* functions were used. Two timers were established within the code. One timer was used to measure the global time and keep track of runtime. The other timer was used as a local lap timer to keep track of δt. MATLAB continues to loop, outputting a new ω value every 0.1s, until the runtime is reached. Once the runtime is reached, the loop is exited. The micro stepper is disabled and the LabJack signal is shut off.

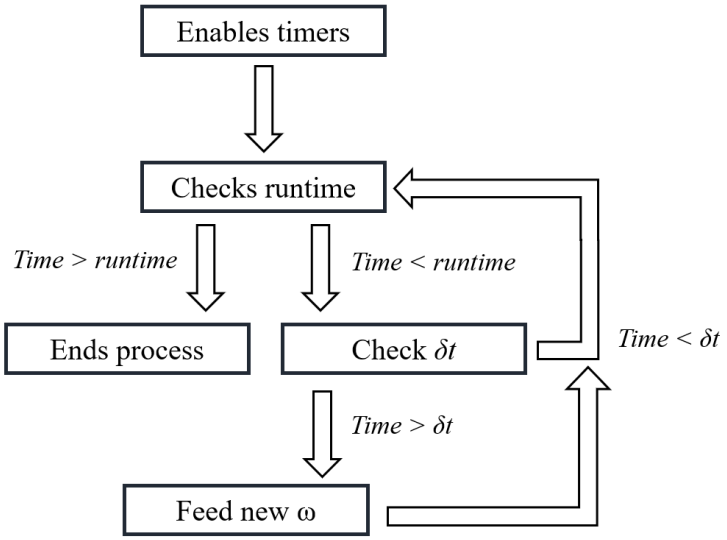

Figure 2.3: Process flow for *RUN* function

2.4 Summary of Described Theory

The calculations necessary for adjusting variables have been described. Future users will have a multidimensional combination of variables that would apply to any set of polygon-shaped quasi-2D tumblers. With the math and software configured, the information must interface with the experimental hardware which is described in the next chapter.

CHAPTER 3

HARDWARE

The physical set up for the lab can be seen in figure 3.1. Indicated are the LabJack, micro-stepper, motor, and driven shaft. Figure 3.2 depicts how the physical devices are connected in series with the developed GUI.

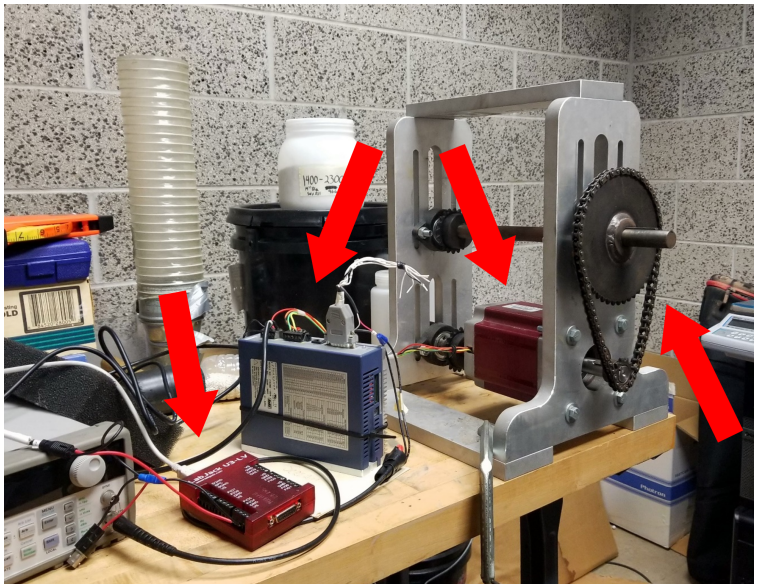

Figure 3.1: Lab Set-Up, from left to right showing LabJack, Micro-stepper, Motor and Driven Shaft

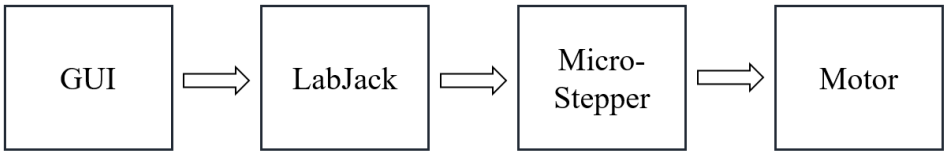

Figure 3.2: Schematic depicting the attachment flow of hardware

3.1 Tumbler

A typical tumbler is constructed of three panels, the middle panel having a cavity which can be filled with particles. The cavities form the shape of the granular flow tumbler environment. In evaluating the performance of the developed system a triangle attachment was mounted on the end of the drive shaft in place of a tumbler. The triangle is chosen over a single pointer because it best emulates the lowest N polygon shape.

The triangle attachment was made from black acrylic material. The front face was cut to be an equilateral triangle. Drawing of triangle front face is shown in figure 3.3.

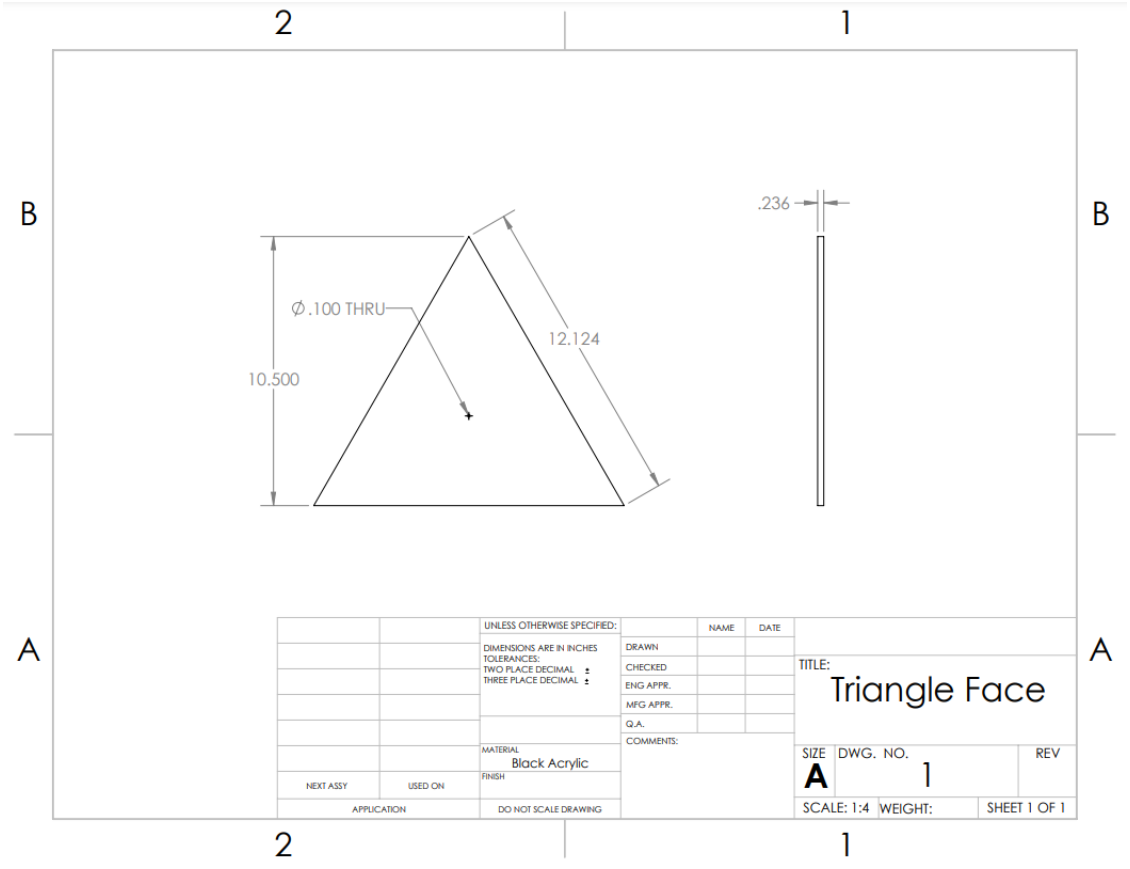

Figure 3.3: Front Face CAD Drawing

A small hole was placed in the center of the triangle so the back mount could be aligned during attachment. The center hole also served as a visual marker with which the center of the triangle was located within frame during the system verification process (described in full detail in Chapter 4).

A back mount was cut also from acrylic material to allow for attachment into the keyed motor shaft in the lab set-up. The back mount was glued to the triangular front face. Drawings for back mount is shown in figure 3.4.

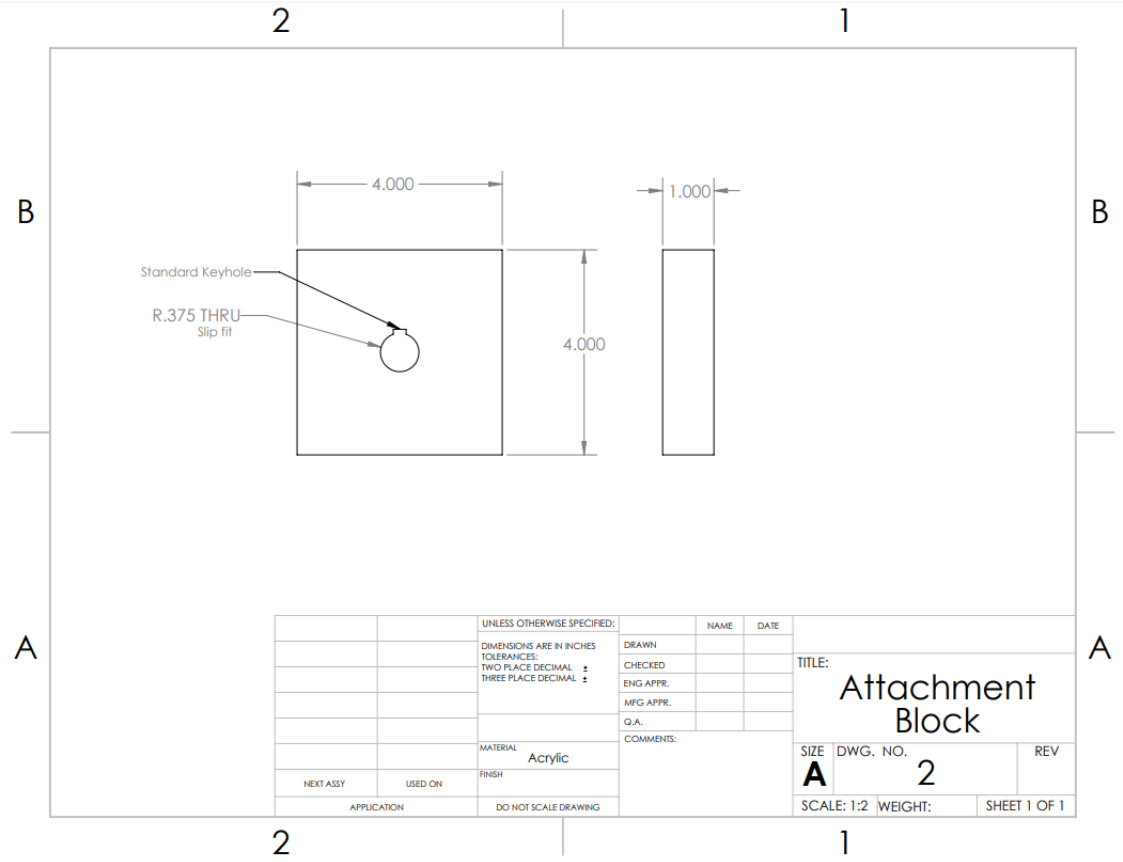

Figure 3.4: Back Mount CAD Drawing

The face of the acrylic triangle was coated with several layers of matt finish paint to mitigate light reflection on the surface.

3.2 Motor Set up

The tumbler panels are driven by a high torque Danaher K42HRLG-LNK-NS-00 stepper motor and a Danaher P70360-SDN motor controller. The motor has a keyed axial drive shaft which tumblers are attached to. The shaft is driven by a chain and sprocket configuration containing two gears. The gear ratio is shown in equation 3.1

$$GR = \frac{11\ driving\ teeth}{48\ driven\ teeth} = 0.229 \tag{3.1}$$

The micro-stepper ensures smooth driving of the motor even when the tumbler is rotating a low rotation rates. It was set to 18,000 steps. During prior experiments at steady-state rotation, a square wave generator was used to input a stepping frequency to the micro controller. For the work conducted as part of this thesis, it takes in the square wave signal from the LabJack. Micro-stepper pins 5 and 6 are used to enable or disable the micro-stepper. If pin 5 is set to 0V and pin 6 is set to 5V, the stepper is enabled. If pin 5 is set to 5V and pin 6 is set to 0V, the stepper is disabled. In the disabled state the micro-stepper the motor is not driven even if the LabJack is still sending a signal. The micro-stepper will remain in the disabled state until it is enabled again.

Completely disabling the micro-stepper adds a layer of safety. If any undesired signal is run by the user, the motor will remain still until the micro-stepper is enabled.

3.3 Signal Controller: LabJack U3-LV

A vital component of the assembly is the signal controller which was available as part of a timer signal integrated within the LabJack U3 LV digital data acquisition hardware.

MATLAB directly communicates with the LabJack to send desired signals to motor set-up. The LabJack U3-LV has multiple modes and can accommodate inputs and outputs for many functions.

One mode is the Frequency Output Mode, which outputs a square wave signal with any specified frequency value. In equation 3.2, $f_{square\ wave}$ is what drives the value of ω but it must be fed to the LabJack as an integer *value* which ranges from 0 to 255. The LabJack has two onboard clock options, a 1MHz clock and at 48MHz clock. The two clocks give two ranges for frequency values. The 1MHz clock was selected. The output square wave signal is defined by several quantities which can be set based on application needs. Equation 3.2 describes the relationship frequency of the square wave, $f_{square\ wave}$, and LabJack values.

$$f_{square\ wave} = \frac{base}{2 * value * divisor} \tag{3.2}$$

where *base* is the clock selected to it takes a value of 1MHz. The *divisor* divides the clock base to narrow frequency range for better application. The *divisor* was set to a value of 2.

In order to determine what *value* to feed to the LabJack, the conversion between user input rotation rate, in units of RPM, and square wave frequency must first be performed. Equation 3.3 shows the conversion between rotation rate and square wave frequency.

$$f_{square\ wave} = \omega_{RPM} \left(\frac{18,000 \frac{steps}{revolution}}{GR} \right) \left(\frac{1\ minute}{60\ seconds} \right) \tag{3.3}$$

Once the angular velocity value is converted into a square wave frequency, equation 3.2 is rearranged and a *value* can be found for any desired ω_{RPM}. Because the LabJack must receive an integer for *value*, the MATLAB *round* function is used to *value* to an integer. The timer value is sent to the LabJack using LabJack integrated MATLAB function.

For the signal output the LabJack is connected to the micro-stepper at FIO4 and ground pins. DAC0 and DAC1 are connected to micro-stepper pins 5 and 6 respectively to execute the enable and disable operations.

3.4 Summary of Hardware Set-Up

A triangle attachment was built to know orientation of the drive shaft. Mechanical connections made to convert rotation rates from signal-motor-shaft. Signal is produced and output to generate steady-state and periodic rotation rates. With all systems operational, the performance of the set-up needs to be evaluated.

$$f_{square\ wave} = \omega_{RPM} \left(18,000 \frac{steps}{revolution}\right) \left(\frac{11\ driving\ teeth}{48\ driven\ teeth}\right) \left(\frac{1\ minute}{60\ seconds}\right) \tag{3.4}$$

CHAPTER 4

VERIFICATION PROCESS

After the system was developed verification needed to be performed to test the accuracy of the control system. Verification was performed by taking videos of eight different testing configurations. The videos were processed as a sequence of images. Observed motion was analyzed and compared to expected behavior.

There are two major components to the verification process: the physical set-up and the digital set-up.

4.1 Physical Set-Up

The testing set up can be seen in figure 4.1.

A triangle attachment was fabricated to fit the motor shaft in the same way the tumbler attachments fit. The reason for creating the attachment was to provide a rotating object which will have high contrast in video data. The triangle attachment was made from black acrylic. The acrylic face was coated in a matt finish to decrease glare for the lighting. To create clean edged contrast with the triangle attachment, a white backdrop was placed just behind the triangle.

During the collection of video data, the triangle was illuminated by a singular 150 Watt halogen work light. The single light allowed for uniform lighting and created minimal shadows around the edge of the rotating triangle. Video was taken using a camera operating at 30fps.

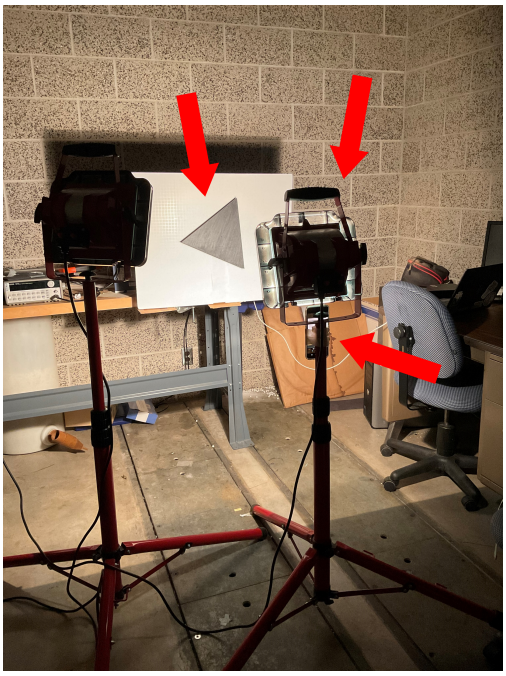

Figure 4.1: Lab Set Up for Testing, from left to right, Triangle Attachment, Halogen Light and Camera

Several different rotation rate configurations were tested with varying N, F, P, and t_{run} See figure 4.1 below for testing combinations.

For all tests, ω_{ss} was set to 4 RPM and amplitude was set to 2 RPM. Average angular velocity of 4 RPM meant that the triangle should rotate on full revolution in 15 seconds. Selecting 2 RPM fpr the amplitude meant that the system ω extrema were half and twice ω_{ss}. Runtime was set to 30 s for the test 1 through test 7. In 30 s it was expected that the triangle would make two full revolutions if the system remained in phase. Additionally, 30 s yielded 900 frames for tests 1 through 7. Since there no feedback loop, test 8 was performed to observe how long the system would remain in phase. Test 8 was performed with a t_{run} on 180 s to test system after 12 revolutions to observe lag over time. Test 8 yielded 5400 frames.

Table Showing Test Configurations						
	N	F	P	Runtime (s)	AVG (RPM)	A (RPM)
Test 1	3	1	0	30	4	2
Test 2	3	1.5	0	30	4	2
Test 3	4	1.5	0	30	4	2
Test 4	4	1.5	0.25	30	4	2
Test 5	4	1.5	0.75	30	4	2
Test 6	5	1	0.25	30	4	2
Test 7	5	1	0.75	30	4	2
Test 8	3	1	0	180	4	2

Table 4.1: Testing Configurations Each Taken as Video Data

4.2 Digital Set-Up

In order to process the test video, a verification code was developed. Videos were exported into individual frames using IrfanView graphic viewer and loaded in developed MATLAB code as a series of frames. The code tracked vertex location across the frames. Vertex data was processed to yield orientation of the triangle in each frame as well as ω across the videos. The verification process is show in figure 4.2.

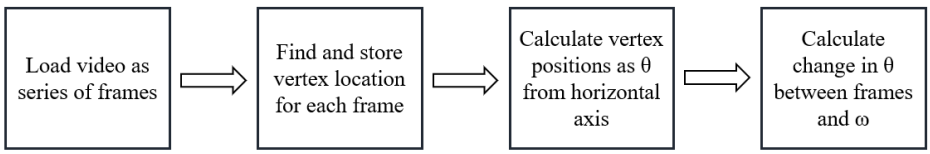

Figure 4.2: Verification Digital Process Flow

While simple in terms of process and information flow, detailed features are necessary to have reliable performance evaluation.

4.2.1 Main Script

Before running the verification script, two quantities were observed from the first frame of each test video. Figure 4.3 shows an example. Size dimension and center location were specified as global pixel locations. Note that in images, the coordinate system is flipped in the y-direction meaning the y pixel location value increases down the image. The location of

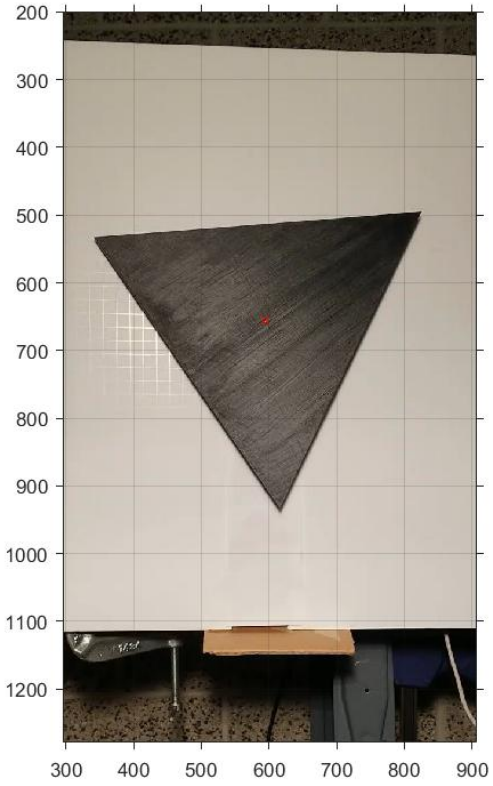

Figure 4.3: Frame loaded for visual inspection of size dimension and center location in pixels, center location is marked by red x

the center of the triangle in pixels were observed and the distance from the center point to the vertex was calculated. Both center location and size were entered into the verification script. The measurements needed to be close to actual values but some error was acceptable. The

main script passed each frame along with the size and center location data to a developed function which outputs the location of a single triangle vertex within the frame. The vertex location was stored in a matrix. As the script looped over each frame, the vertex matrix was filled with the vertex locations across all frames in the test video.

4.2.2 Vertex Calculator

The vertex calculator is at the heart of the digital verification processes. Figure 4.4 shows the process flow of the function.

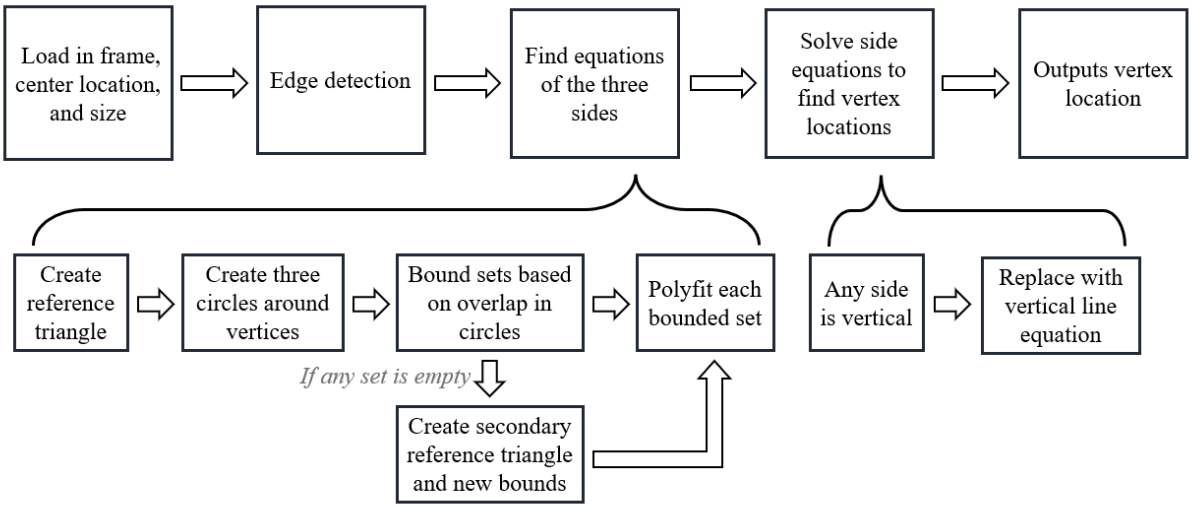

Figure 4.4: Vertex Calculator Digital Process Flow

The vertex calculator function took in a single frame along with the location of the center of the triangle in the frame and the distance in pixels from the center to a vertex. The first step was to clean up the image. Canny edge detection was performed on the frame [10]. Next, the sides of the triangle needed to be located in the frame for the equations of each side to be calculated. Figure 4.5 shows the internal process of the vertex calculator.

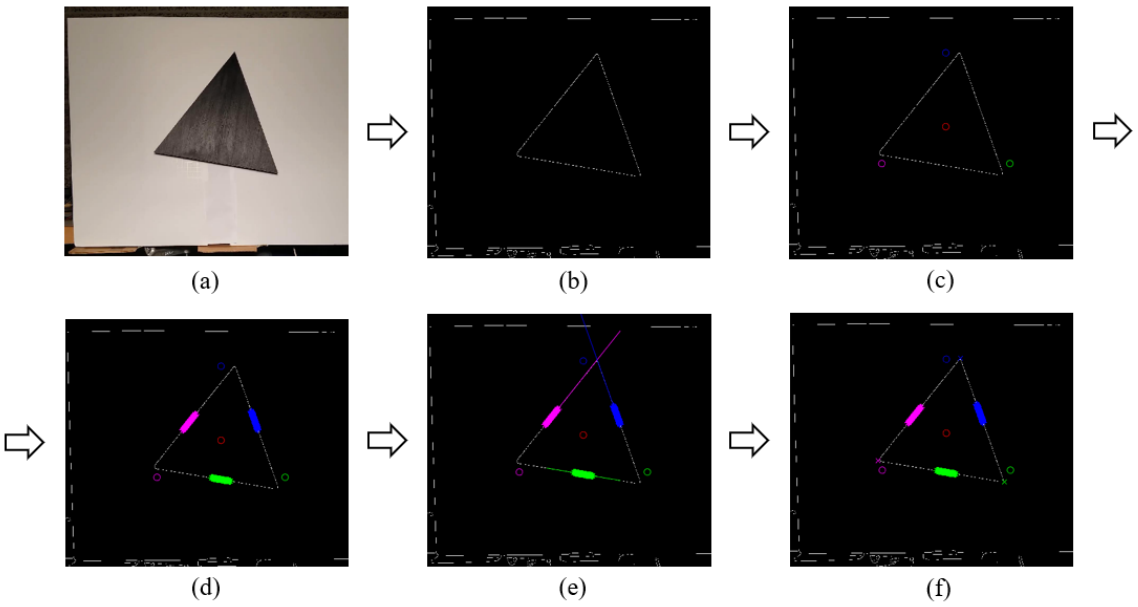

Figure 4.5: Plot showing internal process of vertex calculation script, (a) Frame as it is loaded to function, (b) Frame after edge detection is performed, (c) Reference triangle vertices are shown, (d) Highlighted sections show bounded regions of each side, (e) Polyfit lines plotted, (f) Calculated vertices plotted

To uniquely isolate each side of the triangle, a method was developed which used the overlapping regions of three strategically placed circles to bound portions of triangle sides. To begin, a digital reference triangle was created with the same size and center location as the actual triangle in the frame. Around each of the three reference vertices, a digital circle was created with a radius equal to the size of the triangle. Figure 4.6 gives a depiction of the reference triangle and three circles. The overlapping regions of the three circles, highlighted in figure 4.6 as the blue, green, and pink regions, uniquely bounded a portion of each of the three sides of the triangle.

Using the reference triangle and circle formation shown in figure 4.6 was effective in bounding regions of the triangle sides for almost all triangle orientations. In the case where the triangle in the frame was 180° rotated compared to the reference triangle, the overlapping

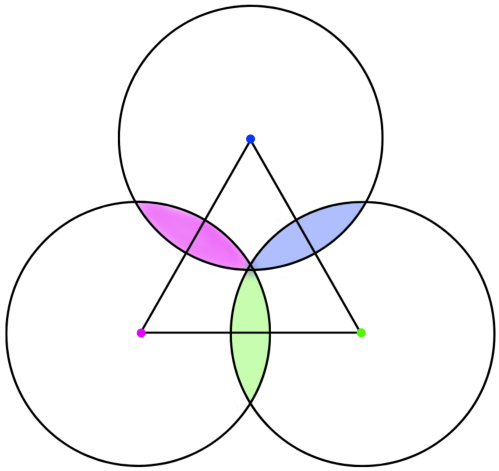

Figure 4.6: Reference triangle with Three Circle Bounding Region Diagram

sections would be empty. Thus no sides would be located. To accommodate all triangle orientations, a condition was set so if any of the three overlaps were found to be empty a secondary reference triangle would be created. The secondary reference triangle was simply a horizontal reflection of the primary reference triangle. Three new circles were then created based on the secondary reference triangle and the new overlaps successfully bounded sets for each side of the triangle in frame. Figure 4.7 gives an example frame which required used the secondary reference triangle to locate the sides and vertices.

The edge pixels, which fall into each overlap region, were each fit with a first order polynomial. The MATLAB function *polyfit* was used to find the polynomial fits. The *polyfit* function does not work for the case of a vertical line because it uses the least-squares method [11]. Issues arise when the slope of the line nears infinity. A condition was implemented into the code which checked the slope generated by the *polyfit* function. If the slope became too large, indicating the presence of a vertical or nearly vertical line, the polynomial generated by *polyfit* was replaced with a horizontal line. The value for the horizontal line

was taken to be the x value of the first pixel in the bounded set for that particular side. The range for large slopes was set to $90° \pm 0.2°$. Figure 4.8 gives an example frame which required the alternative equation formulation and vertex calculation for vertical sides.

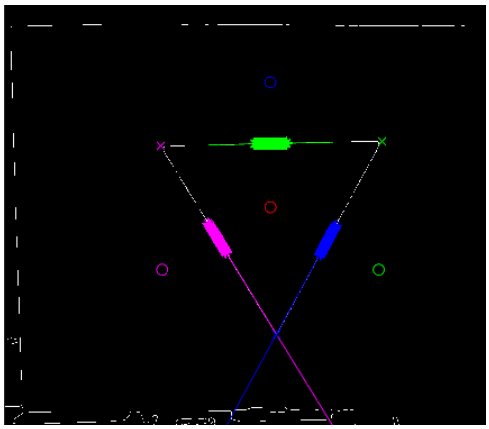

Figure 4.7: Used 2nd reference triangle

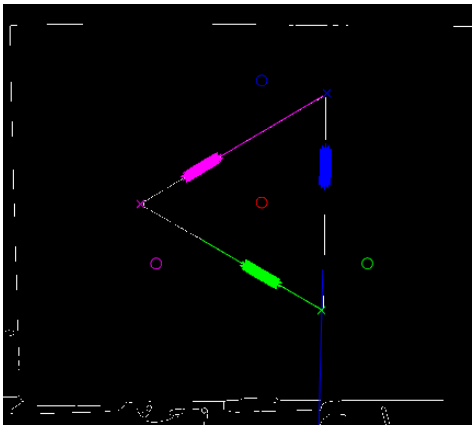

Figure 4.8: Frame which has a vertical side

The intercepts of side-describing polynomials occurred at the vertex locations. Once the equations for each side were found, their intercepts were calculated. Only one vertex location was returned as an output of the function. Because the triangle was an equilateral triangle, the full motion of the triangle is known based on the motion of one vertex and simply repeats periodically every 120°. The vertex which was reported was chosen to avoid error due to consistent shadowing in the frames. A slight shadow occurred under the bottom right edge of the triangle in the frame. The shadow caused some inconsistency in the polynomial edge fits. The vertex which was unaffected by the inconsistency was the vertex between the pink and green bounded regions, (see figure 4.5).

4.3 Summary of Described Verification Process

Video data was collected for eight different test combinations. Each video was exported into frames and analyzed as a series of frames. Developed MATLAB code looped across

each frame and calculated vertex locations as pixel positions. A function was developed to identify all three sides of the triangle in each frame, calculate equations for the sides, and solve the side equations for the vertex locations. Vertex location was stored across the series of frames. Figure 4.9 shows an example of the tracked vertex locations. Shown is the vertex data for Test 6.

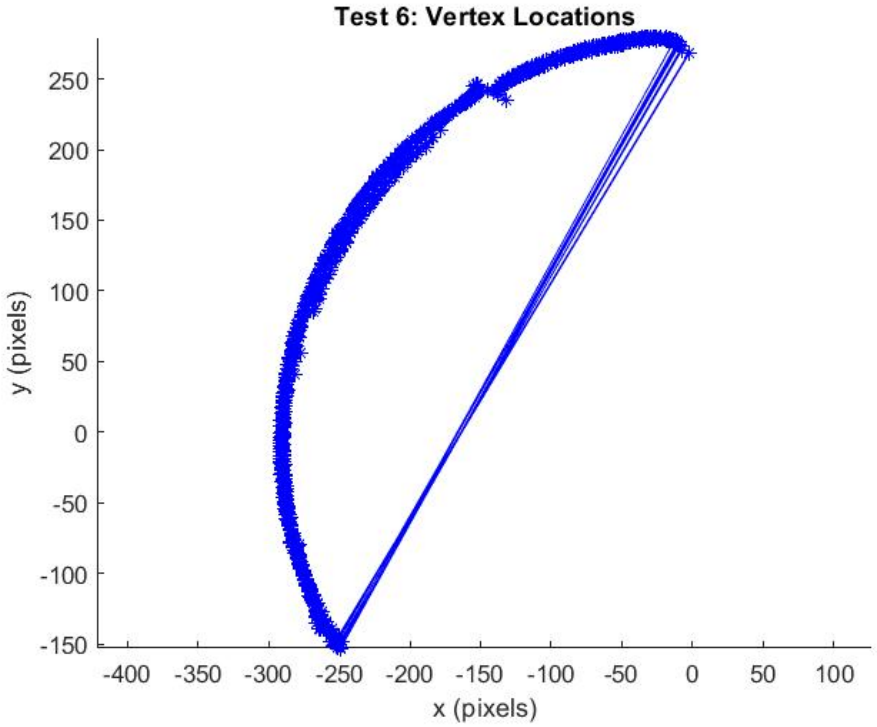

Figure 4.9: Vertex location plot for Test 6

With vertex locations stored, quantities for orientation and angular velocity needed be calculated and analyzed to assess the system performance.

CHAPTER 5

RESULTS

5.1 The Raw Theta Information

From the vertex location data output by the verification script, theθ values needed to be calculated. An inverse tangent function was used to convert for vertex position in x and y pixels to angles in degrees. When the inverse tangent function was directly applied to the vertex location data, the θ values produced were not smooth. Figure 5.1 shows the raw θ data after applying the inverse tangent.

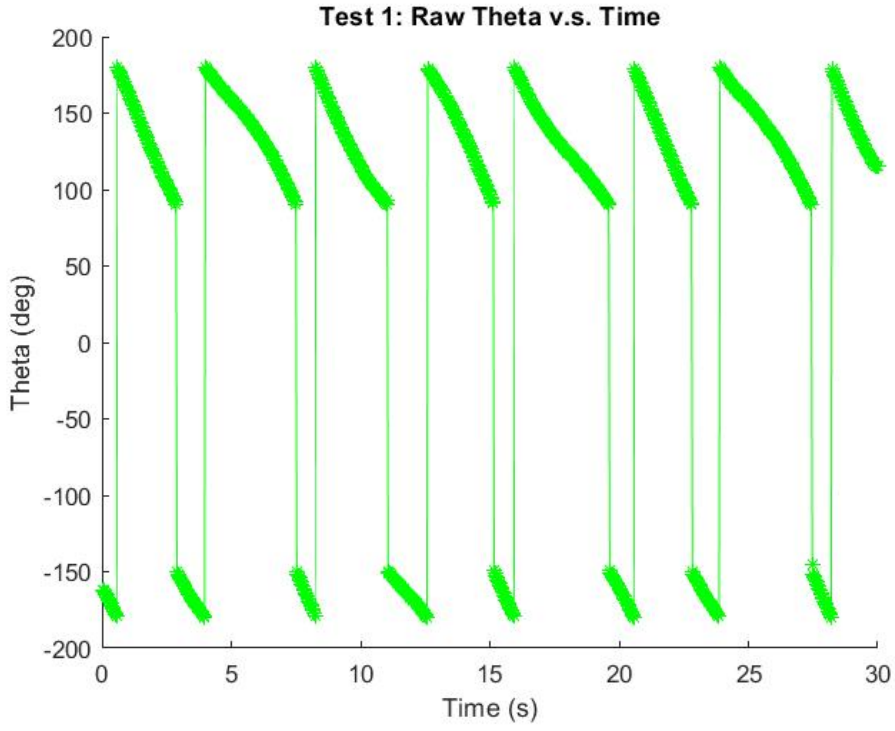

Figure 5.1: Test 1 theta values before unwrapping

Two layers of unwrapping were required to produce smooth angle data. First, the MATLAB *unwrap* function was used to eliminate the jumps from 180° to -180°. The next layer of unwrapping addressed the 120° jumps present because of the periodic vertex tracking. After both unwraps were performed, smooth theta graphs were produced. Figure 5.2 shows the orientation plots for Test 1 after the unwrapping process.

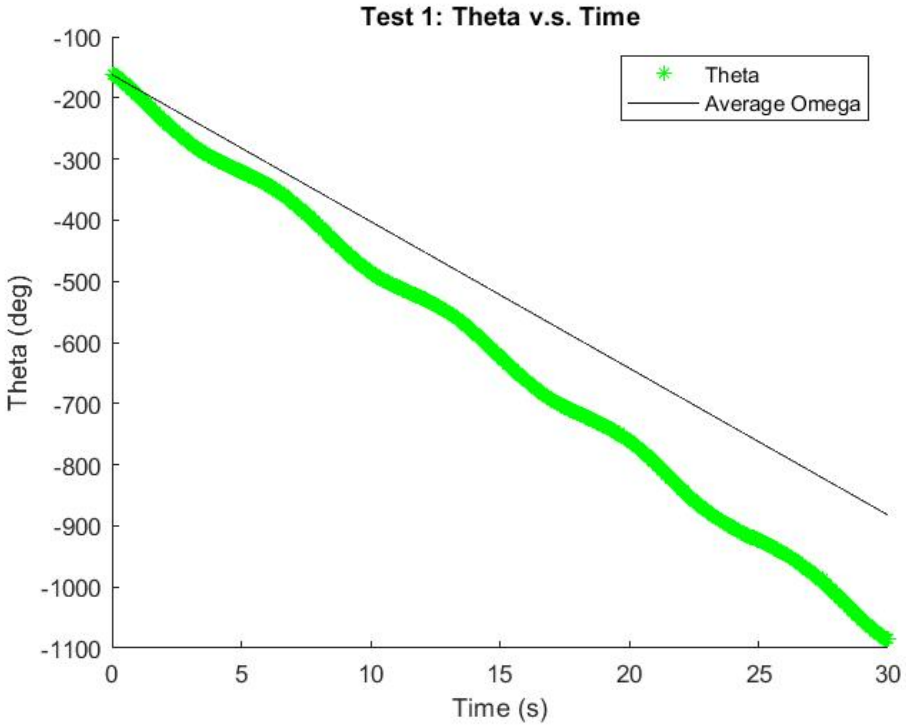

Figure 5.2: Test 1 theta values after unwrapping

The overall slope of the θ graph should have a value which corresponded to ω_{ss} specified for the system. 4 RPM was converted to $24°/s$. The solid black line has a slope of $-24°/s$. The deviation of the θ trend and average ω line indicates some discrepancies in the system.

5.2 Omega Information and Calculation

Angular velocities were calculated by numerical differentiation, $\frac{\Delta\theta}{\Delta t}$. The *diff* function in MATLAB was used to calculate $\Delta\theta$ across the series of frames. Change in time was known between frames to be $1/30$ s due to the frame rate. ω data was noisy due to the numerical derivative. The noise was also related to the sensitivity of vertex tracking.

One main issue of sensitivity in the vertex tracking transpired when the left side of the triangle was at or near vertical position. The vertex calculator should switch to the alternative equation side formulation. To decrease the hick-up frequency, the range of lines considered vertical could be fine-tuned. Figure 5.3 shows the vertex tracking plot with the triangle orientation corresponding to the hick-up area.

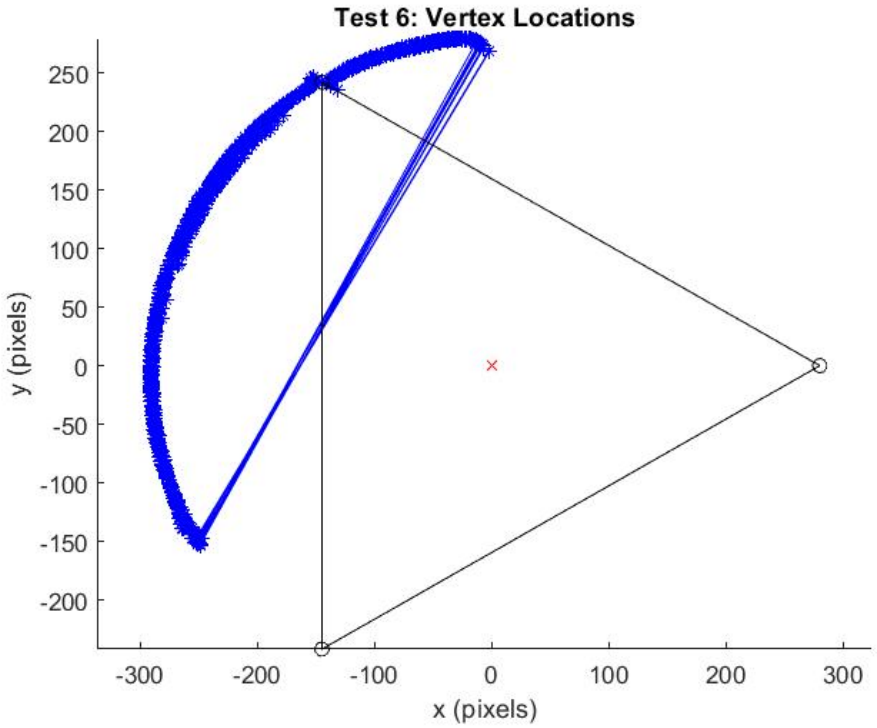

Figure 5.3: Test 6 vertex tracking plot and triangle at issue orientation

Angular velocities were calculated for all trials and comparative trails are discussed in the following sections. For additional comparison, expected average velocity was shown on all ω plots at a value of $-24°/s$. Over all the observed average angular velocity was faster than the expected ω_{ss}. While the system ran faster than expected overall, it was noted that the range of ω was similar to what was expected. With an amplitude it of ± 2 RPM it was expected that ω would have a range of $\pm 12°/s$. ω ranges were observed to be close to the expected range value, indicating that the system was able to convert discrete RPM values to signal frequency properly.

5.2.1 Test 1 and Test 2: Changing F

Figure 5.4 shows the angular velocity plots for test 1 and test 2.

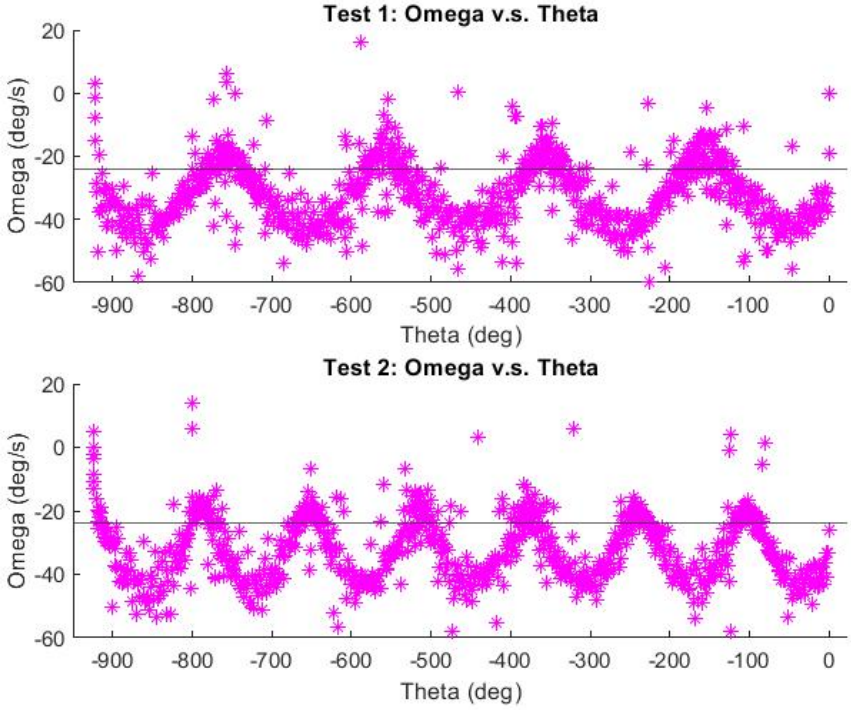

Figure 5.4: Angular Velocity for Test 1 vs Test 2

Between test 1 and test 2, the value of F changes from 1 $\frac{cycles}{side}$ to 1.5 $\frac{cycles}{side}$. Increasing F should increase the number of peaks seen in the velocity plot. In the test 1 plot there are 4 fully developed peaks. In the test 2 plot there are 6 peaks. The observed increase is consistent with the expected effect of increasing F.

5.2.2 Test 2 and Test 3: Changing N

Test 2 and test 3 velocity plots are shown in figure 5.5. Test 2 had an N value of 3, while test 3 had an N value of 4. All other inputs were held constant. It was expected that test 3 would have three more velocity peaks compared to test 2. Test 3 has two additional side lengths it should be passing through. With 1.5 $\frac{cycles}{side}$ and two additional sides, an increase of 3 peaks was anticipated.

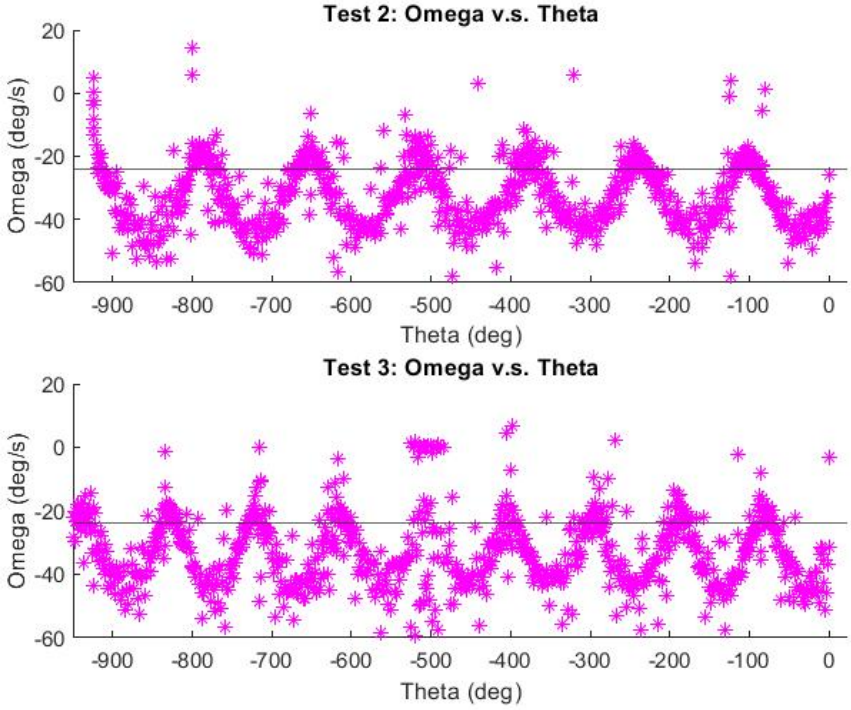

Figure 5.5: Angular Velocity for Test 2 vs Test 3

The number of full peaks in test 2 was 7 and the number of peaks in test 3 was just over 9. While the number of peaks did increase, the increase was less than expected. The discrepancy between the expected increase and the observed increase shows that the time controller was not fully accurate.

5.2.3 Test 4, Test 5, Test 6 and Test 7: Changing N, F, and P

Figure 5.6 shows velocity plots for test 4 through test 7.

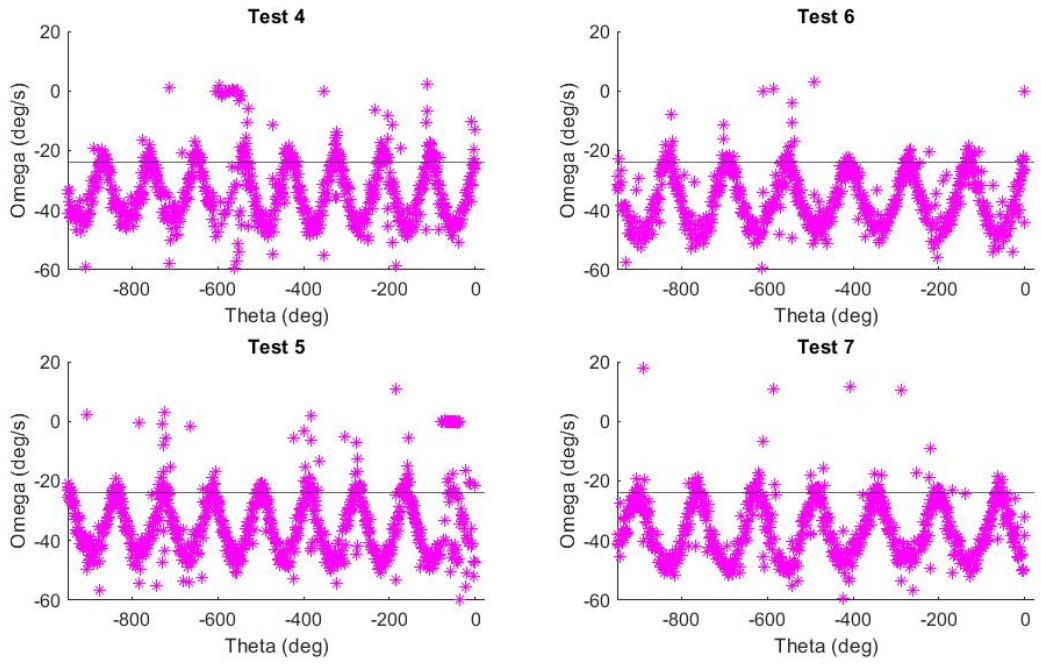

Figure 5.6: Plots of Angular Velocity for Tests 4 through 7

The first column of plots, test 4 and test 5, had the same number of sides, 4, and the same number of cycles per side, 1.5. Plots in the second column had an N value of 5 and an F value of 1. The first column was expected to have more velocity peaks because the

number of sides times the number of cycles per side was greater for the first column. More velocity peaks were seen in test 4 and test 5 when compared to test 6 and test 7.

The top row of plots, test 4 and test 6 had a phase offset of 0.25 while the second plots had a P value of 0.75. Due to the change in P it was expected that the ω plots of test 5 and test 7 should be horizontal reflections ω plots shown for test 4 and test 6. That expectation was confirmed in figure 5.6. This indicates that phase offset was accurately controlled.

5.3 Omega as Calculated in Equation 2.5

While anticipated trends for increasing or decreasing number of peaks were seen, the actual number of peaks were lower that what should have been present. This result indicates that there is an issue with how $\omega(t)$ was formulated in equation 2.5. In particular, f_{time} was formulated in equation 2.6 based on the average angular velocity when in reality it should have been based on instantaneous angular velocity. ω is changing with orientation of the triangle and assuming it to be constant caused error in the conversion from $\omega(\theta)$ to $\omega(t)$. The error was seen in the high average angular velocities observed and it was highly present in the results from the long runtime test. θ values for test 8 are shown in figure 5.7.

An ω_{ss} line was plotted for comparison. It was seen that over time the error in the system becomes increasingly large. In order for the system to be able to update the control equation, equation 2.5, during runtime a feedback system is needed. Chapter 6 further discusses possibilities for future work and incorporation of an encoder.

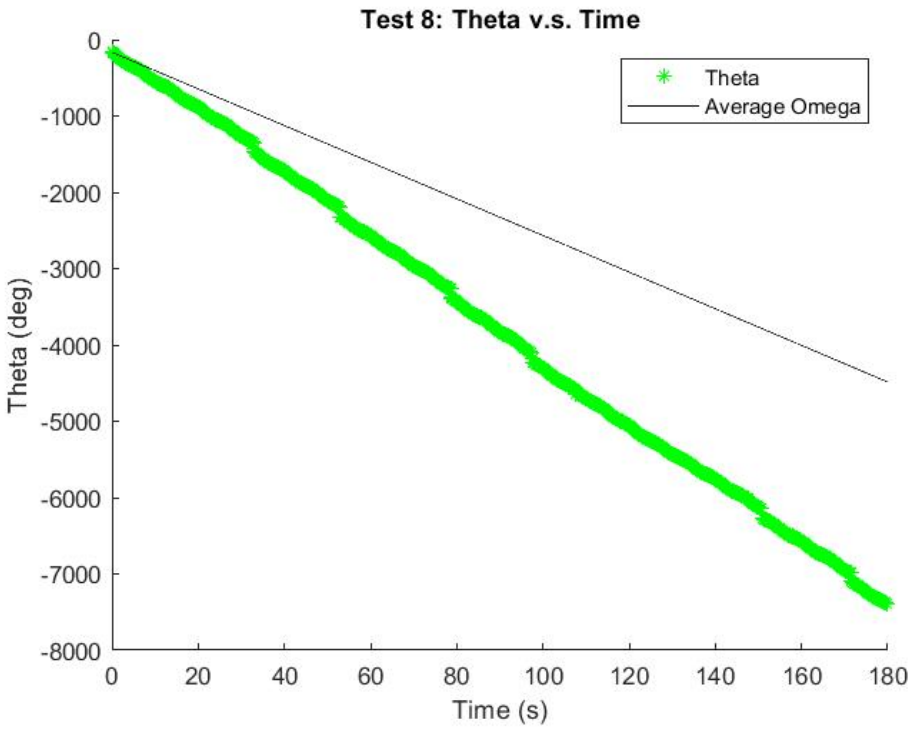

Figure 5.7: Test 8 theta plot and expected average angular velocity

CHAPTER 6

CONCLUSION AND FUTURE WORK

A simple user interface with visualization tool was successfully developed in MATLAB. The GUI can take inputs, update visualization, and send varying signal to the LabJack. Conversion from ω values specified in RPM to frequency output form the LabJack was successful. The system was able to change the driven ω as a function of time based on user inputs. Video data was successfully analyzed via the developed verification processes.

While the system was driven in time and the trend of changes was as anticipated, it was seen that the time based control was not fully accurate. The system can not be controlled by time alone. Overall, the system moved more quickly than anticipated and the correct number of velocity peak were not perfectly produced. To improve control in the future, an encoder can be used to create a feedback loop by providing the instantaneous orientation data needed to accurately update ω as a function of time. Incorporating a feedback loop could also help with lag and help prevent the system getting out of phase over time. Visual feedback could also be incorporated. The same camera used to observe segregation and granular flow properties could be used to observe live rotation rate and adjust speeds.

Other future steps include improving the lighting set-up during testing for better analysis conditions and attaching actual tumblers to observe segregation patterns under various changing flow effects.

REFERENCES

[1] S. J. FIEDOR and J. M. OTTINO. Mixing and segregation of granular matter: multi-lobe formation in time-periodic flows. *Journal of Fluid Mechanics*, 533:223–236, 2005.

[2] Stephen E. Cisar, Paul B. Umbanhowar, and Julio M. Ottino. Radial granular segregation under chaotic flow in two-dimensional tumblers. *Phys. Rev. E*, 74:051305, Nov 2006.

[3] Jason Ostenburg. Experimental analysis of flowing layer characteristicsin equilateral triangular tumblers for quasi-2dmacro-particle flow. Master's thesis, Northern Illinois University, 2022.

[4] Front matter. In A. A Boateng, editor, *Rotary Kilns (Second Edition)*, page iii. Butterworth-Heinemann, Boston, second edition edition, 2016.

[5] Gdr Midi. On dense granular flows. *European Physical Journal E*, 14:341–365, 08 2004.

[6] H. Henein, J.K. Brimacombe, and A.P. Watkinson. Experimental study of transverse bed motion in rotary kilns. *Metall Mater Trans B 14, 191–205*, 1983.

[7] Bruno Laurent and John Bridgwater. Continuous mixing of solids. *Chemical Engineering Technology*, 22:16–18, 2000.

[8] Masamichi Nakagawa, Stephen A. Altobelli, Arvind Caprihan, E. Fukushima, and E. K. Jeong. Non-invasive measurements of granular flows by magnetic resonance imaging. *Experiments in Fluids*, 16:54–60, 1993.

[9] D. V. Khakhar, J. J. McCarthy, Troy Shinbrot, and J. M. Ottino. Transverse flow and mixing of granular materials in a rotating cylinder. *Physics of Fluids*, 9(1):31–43, 1997.

[10] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[11] MathWorks. Polynomial curve fitting.

APPENDIX A

USER INTERFACE CONTROL CODE

A.1 *simple_gui.m*

```
function simple_gui
close all force;
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

%create and then hide the UI as it is being consturcted
f = uifigure('Visible', 'off', 'Position', [360 350 1000 650]);
%figure properties
f.MenuBar = 'none';     %hides the figure menu bar
f.Name = 'Simple GUI';      %names figure title
f.NumberTitle = 'off';      %turns off the figure number display
% f.Resize = 'off';       %makes it so you can't resize the figure
movegui(f,'center')     %brings figure to center of screen


% Initialize Variables:
% A is the amplitude,
% F is the frequency;
% P is the phase shift;
% N is the numer of sides for the polygon
global A F P N L avg runtime ...
```

```
23      ljhandle ljudObj timerClockDivisor aEnableTimers
            aEnableCounters tcPinOffset ...
24      aTimerModes aTimerValues timerClockBaseIndex;
25  A = 1; F = 1; P = 0; N = 3; avg = 4; runtime = 15;
26
27  %Construct the components
28  %creating NumericEditFields
29  nLabel = uilabel(f);
30  nLabel.Text = 'N Sides:';
31  nLabel.Position = [770,370,70,25];
32  nSides = uieditfield(f,'numeric',...
33                  'Position',[855,370,70,25],...
34                  'ValueChangedFcn', @(nSides, event) nChanged(nSides
                        ));
35  nSides.Value = N;
36
37  ampLabel = uilabel(f);
38  ampLabel.Text = 'Amplitude:';
39  ampLabel.Position = [770,280,70,25];
40  amplitude = uieditfield(f,'numeric',...
41                  'Position',[855,280,70,25],...
42                  'ValueChangedFcn', @(amplitude, event) vChanged(
                        amplitude));
43  amplitude.Value = A;
44
```

```
freqLabel = uilabel(f);
freqLabel.Text = 'Cycles per N:';
freqLabel.Position = [770,340,75,25];
frequency = uieditfield(f,'numeric',...
                'Position',[855,340,70,25],...
                'ValueChangedFcn', @(frequency, event) aChanged(
                    frequency));
frequency.Value = F;

phaseLabel = uilabel(f);
phaseLabel.Text = 'Phase:';
phaseLabel.Position = [770,310,70,25];
phase = uieditfield(f,'numeric',...
                'Position',[855,310,70,25],...
                'ValueChangedFcn', @(phase, event) fChanged(phase))
                    ;
phase.Value = P;

avgLabel = uilabel(f);
avgLabel.Text = 'Avg w (RPM):';
avgLabel.Position = [770,405,70,25];
average = uieditfield(f,'numeric',...
                'Position',[855,405,70,25],...
                'ValueChangedFcn', @(average, event) avgChanged(
                    average));
```

```
67 average.Value = avg;
68
69 runtimeLabel = uilabel(f);
70 runtimeLabel.Text = 'Runtime (s):';
71 runtimeLabel.Position = [770,250,70,25];
72 time = uieditfield(f,'numeric',...
73                 'Position',[855,250,70,25],...
74                 'ValueChangedFcn', @(time, event) timeChanged(time)
                      );
75 time.Value = runtime;
76
77 % Create buttons
78 initialize = uibutton(f,'push',...
79                 'Text', 'INITIALIZE LJ',...
80                 'Position',[825, 470, 100, 25],...
81                 'ButtonPushedFcn', @(initialize,event) findLJ(
                      initialize));
82
83 avgW = uibutton(f,'push',...
84                 'Text', 'AVG W',...
85                 'Position',[825, 440, 100, 25],...
86                 'ButtonPushedFcn', @(avgW,event) avgOmega(avgW));
87
88 feed = uibutton(f,'push',...
89                 'Text', 'RUN',...
```

```
90                  'Position',[825, 210, 100, 25],...
91                  'ButtonPushedFcn', @(feed,event) feedConfig(feed));
92
93  stop = uibutton(f,'push',...
94                  'Text', 'STOP & SHUT DOWN',...
95                  'Position',[775, 100, 150, 40],...
96                  'ButtonPushedFcn', @(stop,event) shutdownLJ(stop));
97  set(stop,'Backgroundcolor','r')
98
99  %create axes
100 axes1 = uiaxes(f,'Units','pixels','Position',[50,70,700,500]);
101 %ploting on the axes and specfing graph properties
102
103
104 draw(axes1);
105
106 % Draw Graph
107 function draw(axes1) %This function creats the axes with in the UI
108     cla(axes1, 'reset')
109     %Ploting Polygon
110     %calcualting vertices
111     x = zeros(N,1);    %initializing vectors to hold vert
            coordinates
112     y = zeros(N,1);
```

```
113     i = 1;              %calculating x and y for each vertex based
            on num of sides N
114     for i = 0:1:N
115         x(i+1) = cosd(360*i/N);
116         y(i+1) = sind(360*i/N);
117     end
118
119     hold(axes1,'on');
120     plot(axes1,x,y,'ko')   %plotting vertices
121     plot(axes1,x,y,'b')    %plotting sides of polygon
122     grid(axes1,'on')
123     axis(axes1, 'equal')
124     title(axes1, 'Tumbler Configuration with Signal Plot')
125     title(axes1,'Tumbler Control')
126     %L = (2 +2*cosd(360/N));    %length of side based on N
127     L = norm([x(2) - x(1), y(2) - y(1)]);
128     phi = - 90*(N - 2)/N;  %angle between x and x'(side of
            polygon)
129     theta = linspace(0, -L, 250); %theta vector evenly spaced
            from -L/2 to L/2 with 250 points
130     %Omega which is plotted over one side of the polygon
131     omega = -A*sin(2*pi*F*(theta- L*(P)/F)/L );
132
133     %plotting the rotation
```

```
134         R = [cosd(phi), -sind(phi); sind(phi), cosd(phi)]; %rotation
                mtx
135         rotCos = R*[theta; omega] + [(x(end) + x(1))/2;(y(end) + y(1)
                )/2]; %rotated x and ys
136         plot(axes1,rotCos(1,:), rotCos(2,:));     %plotting the
                rotated coordinates
137         hold(axes1,'off');
138 end
139
140 % Callbacks
141 function nChanged(ieditfield)
142     N = ieditfield.Value;
143     draw(axes1);
144 end
145 function vChanged(ieditfield)
146     A = ieditfield.Value;
147     draw(axes1);
148     if A > avg %forcing no negitive speeds.
149         avg = A + 0.003;
150         disp('Amp can not be greater than average. Avg reset to =
                A + 0.003')
151         ieditfield.Value = avg; %changes wrong thing....
152     end
153 end
154 function aChanged(ieditfield)
```

```
155     F = uieditfield.Value;
156     draw(axes1);
157 end
158 function fChanged(uieditfield)
159     P = uieditfield.Value;
160     draw(axes1);
161 end
162 function avgChanged(uieditfield)
163     avg = uieditfield.Value;
164     if avg < A %forcing no negitive speeds.
165         avg = A + 0.003;
166         disp('Amp can not be greater than average. Avg reset to =
              A + 0.003')
167         uieditfield.Value = avg;
168     end
169 end
170 function timeChanged(uieditfield)
171     runtime = uieditfield.Value;
172 end
173
174 % initialize function/finding LJ
175 function findLJ(uibutton)
176
177     disp('Initilizing LJ');
178     %\\\ Finding and setting up the LJ
```

```
179     % Make the UD .NET assembly visible in MATLAB.
180     ljasm = NET.addAssembly('LJUDDotNet');
181     ljudObj = LabJack.LabJackUD.LJUD;
182
183     % Read and display the UD version.
184     disp(['UD Driver Version = ' num2str(ljudObj.GetDriverVersion
            ())]) % Probably unecessary -mjem
185
186     % Open the first found LabJack U3.
187     [ljerror, ljhandle] = ljudObj.OpenLabJackS('LJ_dtU3', '
            LJ_ctUSB', '0', true, 0);
188     disp('LabJack found') % Display for check point
189
190     % Start by using the pin_configuration_reset IOType so that
            all pin
191     % assignments are in the factory default condition.
192     ljudObj.ePutS(ljhandle, 'LJ_ioPIN_CONFIGURATION_RESET', 0, 0,
            0); %sets all pin configs to defalut - mjem
193     disp('Defaults set') %check point
194
195     % innitiantion/set up section.
196     % Create arrays and get constant values.
197     aEnableTimers = NET.createArray('System.Int32', 2);
198     aEnableCounters = NET.createArray('System.Int32', 2);
199     aTimerModes = NET.createArray('System.Int32', 2);
```

```
200     aEnableCounters = NET.createArray('System.Int32', 2);
201     aReadTimers = NET.createArray('System.Int32', 2);
202     aUpdateResetTimers = NET.createArray('System.Int32', 2);
203     aReadCounters = NET.createArray('System.Int32', 2);
204     aResetCounters = NET.createArray('System.Int32', 2);
205     aTimerValues = NET.createArray('System.Double', 2);
206     aCounterValues = NET.createArray('System.Double', 2);
207     disp('Arrays created') %check point
208
209     %Timer mode freq output set up
210     LJ_tc1MHZ_DIV = ljudObj.StringToConstant('LJ_tc1MHZ_DIV'); %
            Initianting clock base mode to 1MHz
211     aEnableTimers(1) = 1;  % Enable Timer0 (uses FIO4).
212     aEnableTimers(2) = 1;  % Enable Timer1 (uses FIO5).
213     aEnableCounters(1) = 0;  % enable Counter0.
214     aEnableCounters(2) = 1;  % Enable Counter1 (uses FIO6).
215     disp('Timers and Counters enabled.')
216     tcPinOffset = 4;  % Offset is 4, so timers/counters start at
            FIO4.
217     timerClockBaseIndex = LJ_tc1MHZ_DIV;  % Base clock is 1MHz
            with divisor support
218     timerClockDivisor = 2;  % setting clock divisor
219     LJ_tmFREQOUT = ljudObj.StringToConstant('LJ_tmFREQOUT');
220     aTimerModes(1) = LJ_tmFREQOUT;
221
```

```
222 end
223 function avgOmega(uibutton)
224     disp('Enabling stepper')
225     % Set DAC0 to 0 volts.
226     voltage = 0;
227     binary = 0;
228     ljudObj.eDAC(ljhandle, 0, voltage, binary, 0, 0);
229     disp(['DAC0 set to ' num2str(voltage) ' V'])
230     % Set DAC1 to 5.0 volts.
231     voltage = 5.0;
232     ljudObj.eDAC(ljhandle, 1, voltage, binary, 0, 0);
233     disp(['DAC1 set to ' num2str(voltage) ' V'])
234
235
236     %get it going at avg rpm
237     disp('Pushing avg omgea');
238     avg = 3;
239     base = 1e6;
240     current_rpm = avg;
241     rpm2freq = (48/11)*(18e3/60);
242     MM = round(base/(2*timerClockDivisor*current_rpm*rpm2freq)); %
          rounded to be an integer
243     aTimerValues(1) = MM;
244     %push configuration and values
```

```
245     ljudObj.eTCConfig(ljhandle, aEnableTimers, aEnableCounters,
            tcPinOffset, timerClockBaseIndex, timerClockDivisor,
            aTimerModes, aTimerValues, 0, 0);
246     disp(['Timer value = ' num2str(aTimerValues(1))])
247
248 end
249 function feedConfig(uibutton)
250     disp('Enabling stepper')
251     % Set DAC0 to 0 volts.
252     voltage = 0;
253     binary = 0;
254     ljudObj.eDAC(ljhandle, 0, voltage, binary, 0, 0);
255     disp(['DAC0 set to ' num2str(voltage) ' V'])
256     % Set DAC1 to 5.0 volts.
257     voltage = 5.0;
258     ljudObj.eDAC(ljhandle, 1, voltage, binary, 0, 0);
259     disp(['DAC1 set to ' num2str(voltage) ' V'])
260
261     dt = 0.1;
262     total = tic; %starting global timer
263     lap = total; %starting lap timer
264     flag = 1;
265     while(flag)
266         tottime = toc(total);
267         laptime = toc(lap);
```

```
268         if laptime > dt
269             %SOMEHOW CALC w(time) value to push
270             %t = toc(lap);
271             freq = avg*N*F/60; %cycles/sec
272             omega = A*sin(2*pi*(freq*tottime - P)) + avg; %w(t)
273             disp('feeding');
274             base = 1e6;
275             current_rpm = omega;
276             rpm2freq = (48/11)*(18e3/60);
277             MM = round(base/(2*timerClockDivisor*current_rpm*
                    rpm2freq)); % converted RPM to timer value (rounded
                    )
278             aTimerValues(1) = MM;
279             %push configuration and values
280             ljudObj.eTCConfig(ljhandle, aEnableTimers,
                    aEnableCounters, tcPinOffset, timerClockBaseIndex,
                    timerClockDivisor, aTimerModes, aTimerValues, 0, 0)
                    ;
281             lap = tic; %reset lap timer
282         end
283         laptime = toc(lap);
284         if toc(total) > runtime
285             flag = 0; %exit while loop when max time is reached
286             disp('break loop')
287         end
```

```
288     end
289     %disables stepper
290     disp('Runtime complete, stepper off');
291     voltage = 5.0;
292     binary = 0;
293     ljudObj.eDAC(ljhandle, 0, voltage, binary, 0, 0);
294     disp(['DAC0 set to ' num2str(voltage) ' V'])
295     % Set DAC1 to 0.0 volts.
296     voltage = 0.0;
297     ljudObj.eDAC(ljhandle, 1, voltage, binary, 0, 0);
298     disp(['DAC1 set to ' num2str(voltage) ' V'])
299
300     %dissabling timmers // shuts off signal
301     aEnableTimers(1) = 0;  % disable Timer0 (uses FIO4).
302     aEnableTimers(2) = 0;  % disable Timer1 (uses FIO5).
303     aEnableCounters(1) = 0;  % disable Counter0.
304     aEnableCounters(2) = 0;  % disable Counter1 (uses FIO6).
305     ljudObj.eTCConfig(ljhandle, aEnableTimers, aEnableCounters,
            tcPinOffset, timerClockBaseIndex, timerClockDivisor,
            aTimerModes, aTimerValues, 0, 0);
306     disp('Timers and Counters disabled. Shutting off signal.')
307 end
308 function shutdownLJ(uibutton)
309     voltage = 5.0;
310     binary = 0;
```

```
311     ljudObj.eDAC(ljhandle, 0, voltage, binary, 0, 0);
312     disp(['DAC0 set to ' num2str(voltage) ' V'])
313     % Set DAC1 to 0.0 volts.
314     voltage = 0.0;
315     ljudObj.eDAC(ljhandle, 1, voltage, binary, 0, 0);
316     disp(['DAC1 set to ' num2str(voltage) ' V'])
317
318     %dissabling timmers
319     aEnableTimers(1) = 0;  % disable Timer0 (uses FIO4).
320     aEnableTimers(2) = 0;  % disable Timer1 (uses FIO5).
321     aEnableCounters(1) = 0;  % disable Counter0.
322     aEnableCounters(2) = 0;  % disable Counter1 (uses FIO6).
323     ljudObj.eTCConfig(ljhandle, aEnableTimers, aEnableCounters,
            tcPinOffset, timerClockBaseIndex, timerClockDivisor,
            aTimerModes, aTimerValues, 0, 0);
324     disp('Timers and Counters disabled. Shutting off signal.')
325
326 end
327
328
329 %Make figure visible!
330 f.Visible = 'on';
331
332
333 end
```

APPENDIX B

VERIFICATION CODE

B.1 Main Script: *verScript.m*

```
% Verification Script
% Takes series of frames, calcuates and stores vertex locations
    across
% frames of video
close all;

%read in frames
vid ='./Hold Frame1/';
frames=dir([vid,filesep,'*.bmp']);

%Test video 1
siz = 280;
cent = [605, 660];

%Test video 2
% siz = 291; % "radius" of tranlge in frame
% cent = [565,662]; %pixel location ssof center of triangle

%Test videos 3-8
% siz = 280;
% cent = [595, 655];

dt = 1/30;
```

```
23 for  ii = 1:size(frames,1)
24     if (rem(ii,30) == 0)
25         ii
26     end
27     img_pass=(imread([vid,filesep, frames(ii).name]));
28     vert1(ii,:) = edgeFind3Circles_vert1Out(img_pass, siz, cent);
29     t(ii) = ii*dt; %store global time
30 end
```

B.2 Vertex Calculator: *edgeFind3Circles_vert1Out.m*

```
function [bLocal1] = edgeFind3Circles_vert1Out(img,triSize,center)
% edgeFind3Circles takes image, center location, and triangle size and gives
% vertex location for a particular frame
%   Images need to already be loaded into memory before passed into this function (already do imread)
frame = img;
D = triSize;
trueCent = center;

% convert image to grayscale and do edge detection
img_gray = rgb2gray(frame);
img_edge = edge(img_gray, 'canny', 0.25, 4);

% Adding graph to help with error checking
figure(7)
subplot(1,2,1); imshow(frame); hold on; axis on;
subplot(1,2,2); imshow(img_edge);
hold on
axis on

% find all the true pixels in frame (should be all the pixels on the edge
```

```
% of the triangle, may need to add exclusive condition to exclude
    center
% point pixels.
[I,J] = find(img_edge);
%subplot(1,2,1); plot(J,I,'ro')

% Reference vertice locations
topVert = [trueCent(1); trueCent(2)-D];
leftVert = [trueCent(1) - sqrt(3)*D/2, trueCent(2) + D/2];
rightVert = [trueCent(1) + sqrt(3)*D/2, trueCent(2) + D/2];
%plotting ref vertices
plot(topVert(1),topVert(2),'bo')
plot(leftVert(1),leftVert(2),'mo')
plot(rightVert(1),rightVert(2),'go')
x = [topVert(1), leftVert(1), rightVert(1)];
xMid = mean(x);
y = [topVert(2), leftVert(2), rightVert(2)];
yMid = mean(y);
plot(xMid, yMid, 'ro')
% Conditions for the three sides
cond1 = find((J-topVert(1)).^2 + (I-topVert(2)).^2<D^2 & (J-
    leftVert(1)).^2 + (I-leftVert(2)).^2<D^2);
cond2 = find((J-leftVert(1)).^2 + (I-leftVert(2)).^2<D^2 & (J-
    rightVert(1)).^2 + (I-rightVert(2)).^2<D^2);
```

```
42 cond3 = find((J-topVert(1)).^2 + (I-topVert(2)).^2<D^2 & (J-
       rightVert(1)).^2 + (I-rightVert(2)).^2<D^2);
43 if(isempty(cond1) | isempty(cond2) | isempty(cond3))
44     % Change reference triangle
45     % new ref vertices, simply rewrite
46     topVert = [trueCent(1); trueCent(2)+D];
47     leftVert = [trueCent(1) - sqrt(3)*D/2, trueCent(2) - D/2];
48     rightVert = [trueCent(1) + sqrt(3)*D/2, trueCent(2) - D/2];
49     % new conditions based on new ref traingle, simply rewrite
50     cond1 = find((J-topVert(1)).^2 + (I-topVert(2)).^2<D^2 & (J-
           leftVert(1)).^2 + (I-leftVert(2)).^2<D^2);
51     cond2 = find((J-leftVert(1)).^2 + (I-leftVert(2)).^2<D^2 & (J-
           rightVert(1)).^2 + (I-rightVert(2)).^2<D^2);
52     cond3 = find((J-topVert(1)).^2 + (I-topVert(2)).^2<D^2 & (J-
           rightVert(1)).^2 + (I-rightVert(2)).^2<D^2);
53 end
54 subplot(1,2,2)
55 plot(J(cond1),I(cond1),'m*');
56 plot(J(cond2),I(cond2),'g*');
57 plot(J(cond3),I(cond3),'b*');
58 %[temp] = [J(cond3), I(cond3)]
59
60 % Fits for three sides
61 side1 = polyfit(J(cond1),I(cond1),1);
62 side2 = polyfit(J(cond2),I(cond2),1);
```

```
63 side3 = polyfit(J(cond3),I(cond3),1);
64
65 %conditional to see if slope is greater than 286.5
66 if (abs(side1(1)) > 286.5 | abs(side2(1)) > 286.5 | abs(side3(1))
   > 286.5)
67     %one of the sides is vertical then and we must do a different
          way to
68     %find vertices of triangle
69     if (abs(side1(1)) > 286.5)
70         %side1 is vertical
71         x1 = J(cond1); %the x values for this side
72         line1 = x1(1); %equation for the line of side1
73
74         %plotting lines on frame
75         subplot(1,2,1);
76         xloc =  trueCent(1)-D/2*linspace(-1,1,25);
77         %xline(line1,'m-')
78         plot(xloc,polyval(side2,xloc),'g-')
79         plot(xloc,polyval(side3,xloc),'b-')
80
81         %calculating vertices
82         bGlobal1 = [line1, polyval(side2, line1)]; %two vertices
              shared with line1
83         bGlobal3 = [line1, polyval(side3, line1)];
84         % uneffected vertex
```

```
85          A2 = [-side3(1) 1;-side2(1) 1];
86          C2 = [side3(2);side2(2)];
87          bGlobal2 = inv(A2)*C2;
88
89      elseif (abs(side2(1)) > 286.5)
90          %side2 is vertical
91          x2 = J(cond2);
92          line2 = x2(1);
93
94          %plotting lines on frame
95          subplot(1,2,1);
96          xloc =  trueCent(1)-D/2*linspace(-1,1,25);
97          plot(xloc,polyval(side1,xloc),'m-')
98          %xline(line2,'g-')
99          plot(xloc,polyval(side3,xloc),'b-')
100
101         %calculating vertices
102         bGlobal1 = [line2, polyval(side1, line2)];
103         bGlobal2 = [line2, polyval(side3, line2)];
104         A3 = [-side1(1) 1;-side3(1) 1];
105         C3 = [side1(2);side3(2)];
106         bGlobal3 = inv(A3)*C3;
107
108     else %(abs(side3(1)) > 286.5)
109         %side3 is vertical
```

```
110         x3 = J(cond3);
111         line3 = x3(1);
112
113         %plotting lines
114         subplot(1,2,1);
115         xloc =  trueCent(1)-D/2*linspace(-1,1,25);
116         plot(xloc,polyval(side1,xloc),'m-')
117         plot(xloc,polyval(side2,xloc),'g-')
118         %xline(line3,'b-')
119
120         %calculating vertices
121         bGlobal2 = [line3, polyval(side2, line3)];
122         bGlobal3 = [line3, polyval(side1, line3)];
123         A1 = [-side1(1) 1;-side2(1) 1];
124         C1 = [side1(2);side2(2)];
125         bGlobal1 = inv(A1)*C1;
126
127     end
128
129 else
130     %case of no vertcal sides
131     %ploting the side lines on frame to check
132     subplot(1,2,1);
133     xloc =  trueCent(1)-D/2*linspace(-1,1,25);
134     plot(xloc,polyval(side1,xloc),'m-')
```

```
135     plot(xloc,polyval(side2,xloc),'g-')
136     plot(xloc,polyval(side3,xloc),'b-')
137
138     % Vertices in global coordiantes
139     A1 = [-side1(1) 1;-side2(1) 1];
140     C1 = [side1(2);side2(2)];
141     bGlobal1 = inv(A1)*C1;
142     A2 = [-side3(1) 1;-side2(1) 1];
143     C2 = [side3(2);side2(2)];
144     bGlobal2 = inv(A2)*C2;
145     A3 = [-side1(1) 1;-side3(1) 1];
146     C3 = [side1(2);side3(2)];
147     bGlobal3 = inv(A3)*C3;
148
149 end
150
151 % checking by plotting vertices
152 subplot(1,2,2)
153 plot(bGlobal1(1), bGlobal1(2), 'mx')
154 plot(bGlobal2(1), bGlobal2(2), 'gx')
155 plot(bGlobal3(1), bGlobal3(2), 'bx')
156
157 % Convert vertices to local coordandate (center of triangle is
        (0,0))
158 bLocal1(1) = bGlobal1(1) - trueCent(1);
```

```
159 bLocal1(2) = bGlobal1(2) - trueCent(2);
160 bLocal2(1) = bGlobal2(1) - trueCent(1);
161 bLocal2(2) = bGlobal2(2) - trueCent(2);
162 bLocal3(1) = bGlobal3(1) - trueCent(1);
163 bLocal3(2) = bGlobal3(2) - trueCent(2);
164
165 end
```

B.3 Data Processing: *dataPostPros.m*

```
function [thetM, w] = dataPostPros(vertexLocations, timeMtx)
%dataPostPros takes vertex locaion mtx and retruns theta and omega
%   Takes vertex data and time mtx, finds theta for each frame and
    unwraps, finds omega.
%   Works for a camera frame rate of 30fps

t = timeMtx;
vert1 = vertexLocations;
%dt based on 30fps
dt = 1/30;
%finding thetas and smooth
thetM = unwrap(atan2(vert1(:,2), vert1(:,1)))*180/pi;
j = find(diff(thetM)>60);
for I = 1:length(j)
    thetM(j(I)+1:end) = thetM(j(I)+1:end)-120;
end
%omega
w = diff(thetM)/dt; %-1
% w = (thetM(3:end) - 2*thetM(2:end-1) + thetM(1:end-2))/(2*dt);
    %-2
% w = (thetM(5:end) - 2*thetM(4:end-1)...
%     + 2*thetM(3:end-2) - 2*thetM(2:end-3)...
%     + thetM(1:end-4))/(4*dt); %-4
```

```
% size(w);
%plotting resutls
figure(4)
hold on
title('Test 1: Raw Theta v.s. Time')
xlabel('Time (s)')
ylabel('Theta (deg)')
plot(t,thetM,'g*')
hold off
figure(5)
hold on
title('Test 1: Omega v.s. Theta')
ylabel('Omega (deg/s)')
xlabel('Theta (deg)')
plot(thetM(1:end-1),w,'m*')
yline(-24); %this is expected average vleocity based on agerate of
    4RPM
ylim([-60,20]);
hold off
figure(6)
hold on
title('Test 1: Vertex Locations')
xlabel('x (pixels)')
ylabel('y (pixels)')
plot(vert1(:,1),vert1(:,2), 'b-*')
```

```
46 axis equal
47 hold off
48
49 end
```