Force-rate cues reduce object deformation necessary to discriminate compliances harder than the skin

---

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Master of Science

by

Steven Conrad Hauser

August

2016

APPROVAL SHEET

The thesis

is submitted in partial fulfillment of the requirements

for the degree of

Master of Science

Steven Hauser

AUTHOR

The thesis has been read and approved by the examining committee:

Gregory Gerling

Advisor

Shayn Peirce-Cottler

Silvia Blemker

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

August

2016

ABSTRACT

Grasping and manipulating an object requires us to perceive its material compliance. Compliance is thought to be encoded by relationships of force, displacement and contact area at the finger pad. Prior work suggests that objects must be sufficiently deformed to become discriminable, but the utility of time-dependent cues has not been fully explored. The studies herein find that the availability of force-rate cues improve compliance discriminability so as to require less deformation of stimulus and finger pad. In particular, we tested the impact of controlling force-rate and displacement-velocity cues in passive touch psychophysical experiments. Additionally, a novel ink-based method to mark the finger pad was used to measure contact area per stimulus, simultaneously with displacement and force. Stimulus compliances were chosen to span a range both harder and softer than the finger pad. There were three major findings. First, compliances harder than the finger pad are more readily discriminable by force cues – i.e., when displacement is controlled between stimuli. Second, the harder stimuli are discriminable at lower forces when force-rate cues are available – i.e., when displacement-velocity is controlled between stimuli – than vice versa. In contrast, compliances softer than the finger pad are equally discriminable at low forces, regardless of control mode. Third, selecting particular force-rates can confuse observers. For example, the harder compliances are less discriminable if the more compliant of the two is indented at a greater force-rate than if the more compliant of the two is indented at a lesser force-rate. These findings are important for the next generation of tactile rendering displays and robotic exploration strategies.

TABLE OF CONTENTS

ABSTRACT i

TABLE OF CONTENTS ii

TABLE OF FIGURES iii

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: METHODS 3

A. Experimental Apparatus: Stimuli and Indenter 3

B. Participants 6

C. Measurement of Contact Area 7

D. Experimental Procedures 10

Initial psychophysical discriminability 10

Experiment 1: Biomechanical, ink-based experiment with discrete displacements 10

Experiment 2: Biomechanical, force-displacement experiment with continuous displacement 11

Experiment 3: Psychophysical experiment with controlled indentation velocity or force-rate 11

Experiment 4: Psychophysical experiment with varied force-rates 12

CHAPTER 3: RESULTS 13

Experiment 1: Ink-based experiment with discrete displacements 14

Experiment 2: Force-displacement experiment with continuous displacement 15

Experiment 3: Psychophysical experiment with controlled indentation velocity or force-rate 17

Experiment 4: Psychophysical Experiment with varied force-rates 20

CHAPTER 4: DISCUSSION 21

CHAPTER 5: CONCLUSIONS AND FUTURE DIRECTION 23

ACKNOWLEDGEMENTS 25

REFERENCES 26

APPENDIX 1: FORCE-RATE CONTROL CIRCUITRY 28

APPENDIX 2: CODE 30

XPS motion control and psychophysics software 30

Contact area imaging software 107

TABLE OF FIGURES

Figure 1. Stiffness of the stimuli and comparison to data of Srinivasan and LaMotte 1995 5
Figure 2. Basic discriminability of each set of objects 5
Figure 3. Indenter setup 6
Table 1. Summary of the dimensions of each participant's distal phalange 6
Figure 4. Fingerprints and shape/area analysis 7
Figure 5. Biomechanical relationships of force, displacement, and contact area for 5 subjects 14
Figure 6. Force-displacement relationships and cue differences between stimuli 15
Figure 7. Psychophysical results in which force-rate or indentation velocities were controlled between stimuli 17
Figure 8. Effect of control mode on discriminability with hard and soft compliances 19
Figure 9. Psychophysical results in which force-rates were varied between stimuli 20
Equation 1. Force-rate control equation 28
Figure S1. Generalized Schematic of Force-Control circuit 29

CHAPTER 1: INTRODUCTION

Compliance perception is necessary to properly manipulate or interact with naturalistic objects. Understanding the cues underlying compliance perception is vital for the design of tactile displays to render virtual environments [1-5]. Psychophysical experiments with the bare finger—as opposed to probe or stick-based interactions—have investigated the biomechanical cues that underlie compliance perception [6-8], which is significantly less understood than the perception of rigid objects. While proprioceptive and cutaneous cues together convey compliance information, the cutaneous cues related to surface deformation of both stimulus and finger pad are the most vital [6, 9]. However, exactly which cutaneous cues are most important remains unknown.

Relationships at the finger pad surface between force, indentation depth, and contact area are thought to be key encoding mechanisms. The generally accepted paradigm considers compliance to be determined by contact area as a function of force; the same force applied to two different compliances will result in two different contact areas, which can be used to distinguish them [2, 3, 10]. However work to replicate these cues with tactile rendering displays suggests that other cues, including distributions of stresses and strains within the skin's layers and time-dependent cues such as force-rate, may also be relevant. Additionally, there is evidence to suggest that we rely on different cues based on the compliance of the material [1, 8, 9, 11]. In particular there is some significant perceptual distinction between objects less compliant than the finger pad versus those more compliant—which are categorized by subjects as "hard" and "soft," respectively [8]. Despite this distinction, the majority of work has only considered compliances much harder than the finger pad.

Time-dependent cues, though much less studied, may comprise vital information available during the behavioral timescales of real interaction with a stimulus. Force-rate in particular is of interest, along with indentation velocity and contact area-rate [2, 12]. Externally applied forces during

exploration of a stimulus alter compliance estimates [13]. Additionally, work involving kinesthetic haptic displays (i.e., through a stick- or probe-based object, not directly with the bare finger) has suggested that compliance judgements are based upon force-rate and velocity information, as opposed to steady-state force-displacement relationships [12]. However only one study has investigated the effects of force-rate in discrimination tasks with the bare finger [6], and force-rate was not commanded directly; instead, stimuli were presented at randomized indentation velocities to control for its effects.

New techniques must be developed to measure skin-object cues available during interaction with compliant stimuli. Such measurements have been performed in recent studies involving grip and slip of the finger pad on transparent glass-plates, in which changing contact area is measured over the time course of finger slip [14, 15]. However, these studies focus upon rigid body interactions. Our group has begun work recently along a similar vein for compliant stimuli [11]. Quantifying biomechanical interactions at the same time as psychophysical experiments is especially important to elucidate which cues might have been used by subjects.

To attempt to show that time-dependent cues may be key to compliance perception and better tie to behavioral timescales, we employed both force-rate and indentation-velocity stimulus control in a series of biomechanical and psychophysical experiments. Compliances harder and softer than the finger pad were utilized. We measured biomechanical cues between silicone stimuli and the finger pad including contact area, which required devising a novel technique. The results suggest that force-rate cues are critically important in discriminating objects harder than the finger pad, by reducing the amount that each stimulus must be deformed into the skin surface.

CHAPTER 2: METHODS

Our overall approach was to predict indentation control modes which might affect how easily subjects can discriminate two compliances, based on measured biomechanical relationships between the human finger and stimuli both harder and softer than the skin. Using a novel, ink-based method to measure contact areas between stimuli and the finger pad, force and contact area relationships for five individuals were measured at discrete levels of displacement. To alter the availability of temporal cues, we ran passive touch psychophysical experiments utilizing two control modes. In one control mode, stimuli were presented with identical 2-second triangle waves of force, leaving indentation velocity (and possibly contact area-rate) information available to distinguish the objects. In the other control mode, stimuli were presented with identical 2-second triangle waves of displacement, leaving force-rate information available. In a final psychophysical experiment with the hard set of stimuli, each trial one stimulus was indented at twice the force-rate of the other to the same peak force, such that in half of the trials the more compliant object was indented at a higher force-rate than the less compliant.

A. Experimental Apparatus: Stimuli and Indenter

We constructed two sets of stimuli, one less compliant or the "hard" set than the other more compliant or the "soft" set. We refer later to the hard stimuli as "Hard 1" and "Hard 2," and the soft stimuli as "Soft 1" and "Soft 2." Stimulus modulus was estimated by the mixing ratio of silicone elastomer determined from previous experiments [16-18]. The sets of compliance values were chosen to be greater or lesser than that of the fingertip. The stiffness of each stimulus and a human finger were determined using a 6 mm flat-plate indenter indenting at 0.5 mm/s to a force of 1 N. Each stimulus was cylindrical with a diameter of 3.8 cm and height of 1.0 cm, so that its diameter was larger than the fingertip contact. Stimuli were constructed with a silicone elastomer (BJB Enterprises, Tustin, CA; TC-5005), the compliance of which was controlled through a ratio of crosslinker component [16-18]. A small

indentation at the center of the surface of each stimulus, approximately 1.0 mm in diameter with a depth of 0.3 mm and imperceptible to the participant, was introduced in the casting process so as to be usable later as a consistent point of comparison between stimuli and across stimulus replications and indentation levels. As well, it enabled a means to decipher the directional spread in contact area (proximal-distal versus lateral-medial). Basic passive discriminability of stimuli within each set was determined through a brief psychophysical experiment.

To indent stimuli into the finger (Fig. 3), we controlled a Newport ILS-100 MVTP Linear Motion Stage with a Newport XPS Motion Controller. A Windows 7 PC running software written in Python 2.7 commanded indentations through an ethernet connection with the motion controller, which directly interfaced with the motion stage. Force was measured by a load cell (0 – 22.4 N range; Omegadyne LCFD-5, Stamford, CO, USA) mounted to a cantilever attached to the motion stage. Force was sampled through an analog-to-digital converter on the motion controller. A 3D-printed housing for stimuli was constructed with an embedded servo motor (pictured in Figure 3). This device allowed stimuli to quickly be switched during passive psychophysics experiments. A padded armrest was bolted onto the base of the motion stage to secure the forearm and index finger. Custom circuitry was built to allow the indenter to profile force in time, which was utilized to control stimulus force-rates in some experiments.

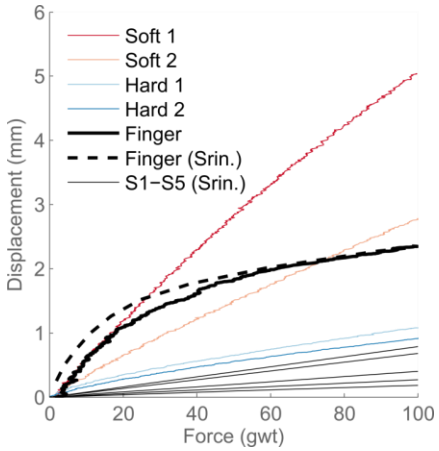

Figure 1. Stiffness of the stimuli and comparison to data of Srinivasan and LaMotte 1995. Stimulus stiffness was measured using a 6 mm cylindrical flat-plate indenter and confirmed that the soft set was more compliant than the finger pad and the hard set less compliant.

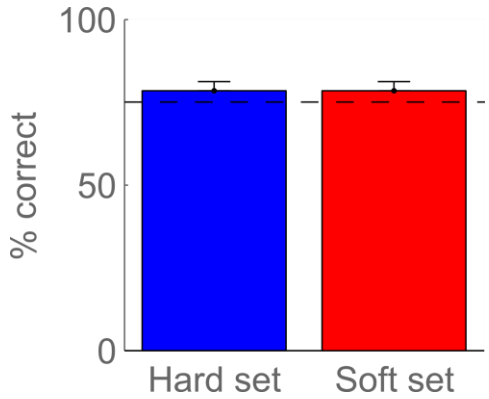

Figure 2. Basic discriminability of each set of objects. Stimuli were indented at 2 mm/s to a force of 3 N. Each set of stimuli demonstrated discriminability with >75% correct responses for 3 subjects.

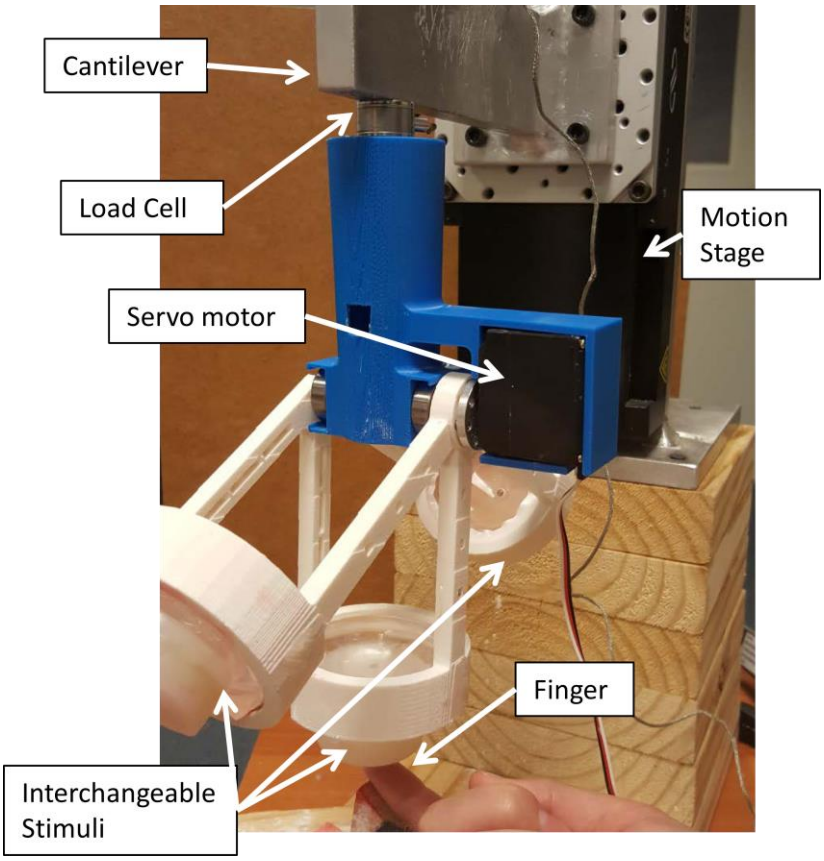

Figure 3. Indenter setup. The main components of the indenter and device used to quickly interchange stimuli in psychophysical experiments.

B. Participants

	Lateral-medial	Thickness	Distal-Proximal	Experiment #
Subject 1	16.4	10.7	25.9	3
Subject 2	14.3	10.6	24.0	1,3
Subject 3	14.0	10.5	22.3	3
Subject 4	15.0	11.6	24.0	3,4
Subject 5	14.0	10.6	28.3	3
Subject 6	15.3	9.7	23.9	1
Subject 7	16.3	12.3	25.2	1,4
Subject 8	14.4	9.1	24.6	1,2
Subject 9	16.0	10.6	28.8	1

Table 1. Summary of the dimensions of each participant's distal phalange. Tabulated are finger pad measurements for each of the 9 subjects. All units in millimeters.

Nine subjects in total participated in the experiments (mean age = 22.7, SD = 1.5, 5 male, 4 female). Finger pad measurements were taken for each subject (Table 1). All enrollees granted their consent to participate. All participants continued to completion and no data were disregarded.

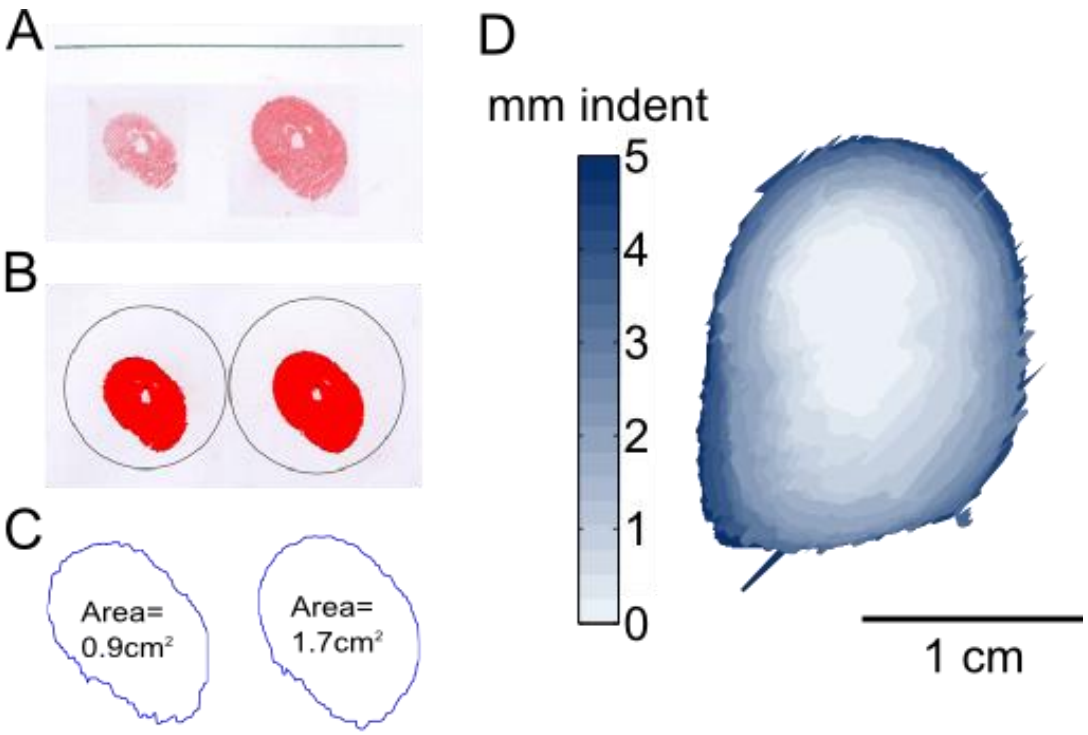

Figure 4. Fingerprints and shape/area analysis. A) Fingerprints stamped onto a sheet of paper after successive indents with the Soft 2 stimulus, scanned into .jpg format. The uncolored area in the middle of the print represents the marker created the small indentation in the stimulus which allowed the experimenter to identify a consistent point between indents. B) A portion of the software tool where the fingerprint is identified, color thresholded, and an outline is determined around the fingerprint. C) The output of the tool: a set of vertices representing the shape of the fingerprint and an area determined by Gauss's area formula. D) The contact areas for a series of 20 displacements overlaid in sequence.

C. Measurement of Contact Area

We developed a novel method to measure the area of finger pad-stimulus contact upon indentation by compliant stimuli. This method complements one used previously for rigid body stimuli [15]. We directly measured the finger pad-stimulus contact areas at different indentation levels. An example overview of the process is given in Fig. 4 and detailed below.

In specific, washable ink (Studio G Red) was applied to the stimulus before each indent with a stamp pad. After each indent, a sheet of plain white paper was carefully rolled onto the finger pad in order to transfer the ink to the paper. Between indents the finger was gently wiped with a moist paper towel to remove ink. This process was repeated several times with one sheet of paper at various indentation levels. Afterwards, the sheets of paper were marked with a 5.0 cm line to scale the data, scanned into .jpg format, and processed by custom software (written in Python 2.7). Within the software were identified the points in each fingerprint created by the small indentation on the bottom of each stimulus. The color threshold was adjusted to distinguish the area of the red ink from the background.

We tested two different software methods to analyze the contact areas. The first method was almost entirely automated, and required the use of sheets of paper with a series of printed 5.0 cm by 5.0 cm black boxes to record fingerprints. Each fingerprint recorded during an experiment was placed into a separate box. After scanning an image into .jpg format, a blob-finding algorithm identified the printed black squares on the page. Next, the fingerprint color was thresholded, and the number of pixels above threshold within the bounds of each box was counted. This number was then scaled by the area of the printed box (in pixels) in order to determine the area of the fingerprint. There were a few problems with this method. The number of boxes printed on the page seemed like an unnecessary hindrance on gathering the data. The grooves on the finger pad created many lightened lines within the fingerprint, which were often missed during thresholding and subsequently not counted in the area calculation. Lastly, there was no way to extract the shape of the contact areas or overlay them as only an area value was calculated. We did not end up using this method.

The second method, which was ultimately used, required user input during the calculation but generated more reliable results and included shape and position information along with area. We used blank sheets of paper with a drawn-on 5.0 cm reference line to record fingerprints with this method.

Additionally, we began constructing stimuli with small indentations in their surface, so that a marker would be left on each fingerprint at a consistent reference point. After a series of indentations the sheet of paper was scanned into .jpg format and loaded into software. The software displayed the image and requested that the user identify the marker within each fingerprint. After clicking on each marker, the analyst designated a radius from the marker in which to search for each fingerprint. Fingerprints were stored as a marker location, radius pair.

After selecting each fingerprint on the page, the analyst used a slider to select a threshold value for the fingerprint ink color. This updated the on-screen image with bright red indicating thresholded pixels. The threshold value was modified until the edges of all fingerprints were thresholded. Next the experimenter used another tool in the software to identify the 5.0 cm line in the image. Each end of the line was selected and the software calculated the length of each pixel in centimeters, which was later used to calculate the area of each fingerprint.

For each marker location and radius identified per fingerprint, from earlier, a serial search was conducted to determine the bounds of the fingerprint. Edges of the fingerprint were determined by searching from the top-to-bottom of each circle for transition to thresholded color, then bottom-to-top. This resulted in a series of points, which outlined the fingerprint. This set of points was subtracted from the marker point such that coordinates were consistent between fingerprints from the same stimuli. The final set of points was used to determine an area in pixels using Gauss's area formula, which was scaled to a physical area in squared centimeters through calculations from the reference line. The final output consisted of a set of coordinates and an area per fingerprint.

D. Experimental Procedures

Initial psychophysical discriminability

The first psychophysical experiment utilized forced-choice discrimination to evaluate the pairs of less and more compliant stimuli. Using a servomotor affixed to the load cell, we were able to quickly alternate a set of two stimuli during the experiment (8 sec between successive stimulus presentations). A total of 80 trials were run per subject: 40 trials between the two hard stimuli and 40 trials between the two softer stimuli. The 40 trials consisted of a randomized set of 20 trials where the same stimulus was presented twice and 20 trials where different stimuli were presented. Within the hard or soft set, each stimulus was presented first and second in the trial the same number of times. In each trial, the first stimulus was indented into the finger pad at 2 mm/s and force was measured on the load cell. The indentation speed of 2 mm/s was similar to the range of velocities used in [6], which ranged from 2.4 mm/s to 3.6 mm/s. A limitation of a constant indentation velocity was that the subjects could possibly distinguish the objects based on the total time of indentation; however, informally subjects made no mention of indentation time as a factor in their estimates. The indenter stopped moving when the force reached 3 N and remained still for 1 second. Then the indenter retracted and the next stimulus was presented in the same manner approximately 8 seconds later. After each trial subjects were asked to choose which of the two stimuli was harder.

Experiment 1: Biomechanical, ink-based experiment with discrete displacements

The first biomechanical experiment utilized our method of measuring the contact area between the stimulus and finger pad. This was done to generate a series of discrete force to contact area and displacement to contact area relationships. Using this method four stimuli were indented into the fingertips of the human-subjects, to different levels of displacement. Velocity was controlled at 2 mm/s. A point of contact for each stimulus was determined where the stimulus first made visible contact with

the index finger and the subject detected contact. Then, a subsequent set of 15 indentations (5 sets of 3 replications) up to 5 mm was made for each stimulus.

Experiment 2: Biomechanical, force-displacement experiment with continuous displacement

As the first experiment could only take static measurements at peak displacements, a second biomechanical experiment was run with continuous displacement into the finger pad to examine force and displacement as they changed dynamically throughout indentation. Only force was recorded, as the contact area measurement method could not be applied to a continuous indent. Force was sampled at approximately 50 Hz as each stimulus was indented from contact to 4 mm at 1 mm/s. Measurements were taken with all 4 stimuli.

Experiment 3: Psychophysical experiment with controlled indentation velocity or force-rate

In a forced-choice psychophysical experiment, either indentation velocity (displacement-rate) or force-rate were controlled between stimuli. Using a 3D-printed device with a servo-motor, stimuli could be switched out very quickly between indents with approximately 3 seconds between. For each subject, 80 trials in total were performed: 40 with the hard set of stimuli and 40 with the soft set. Within each set, 20 trials controlled force-rate between stimuli and 20 controlled indentation velocity. For one single subject, an additional 20 trials were performed with controlled force-rate using greater forces. Each stimulus within a set was presented first and second in the trial an equal number of times. In every trial, time was kept constant at 2 seconds per indent such that it could not be used in subjects' judgements. In a force-rate controlled trial, each stimulus was indented into the finger pad with a triangle-wave of force peaking at a desired force level at t=1 second. In an indentation velocity controlled trial, each stimulus was presented with a triangle-wave of displacement peaking at a desired displacement at t=1 second. After each trial subjects were asked to choose which of the two stimuli was harder. Force and displacement were sampled during each trial at approximately 300 Hz for further analysis.

For the hard set of objects, brief experiments were run to the presented trials to determine a force-controlled condition in which subjects could not distinguish the objects. The first trials began with 5-6 trials of 4 N/s to 4 N, and discriminability was estimated form these few responses. If not, these parameters were used in the final set of trials. Otherwise, force-rate was halved to 2 N/s to 2 N and another brief set of trials were ran. This process was repeated until the objects were no longer discriminable. Afterwards displacements were selected to produce lower forces to test in displacement-controlled experiments (typically 1 mm/s to 1 mm). For the soft set of objects, a similar procedure was employed. However, every subject could discriminate at 0.5 N/s to 0.5 N, which was the smallest force condition we could reliably deliver with our setup.

Experiment 4: Psychophysical experiment with varied force-rates

A final forced-choice psychophysical experiment varied force-rates between the hard set of stimuli. The experiment was performed on two subjects; 40 trials with one subject and 48 trials with the other. Within each trial, one stimulus was indented with a triangle wave of force at 0.5 N/s to 0.5 N and the other with a triangle wave of force at 1 N/s to 0.5 N. Time was not controlled within trials, such that the object with the higher force rate was presented for half as long as the other. Both objects in the set were presented an equal number of times with the higher or lower force rate, and also as first or second in the trials. Subjects were asked to choose which stimulus was harder after each trial.

CHAPTER 3: RESULTS

We found that time-dependent cues such as force-rate and indentation velocity impact the requisite deformation to discriminate compliances. Biomechanical relationships at the skin surface were measured between finger pad and stimulus for 5 subjects and all stimuli, which indicated that displacement differences at a given force were much smaller for the hard stimuli than for the soft. For the hard set of stimuli, subjects required more force to discriminate when each stimulus was presented with an identical triangle-wave of force than an identical triangle-wave of displacement. For the soft set of stimuli, subjects could discriminate in either case. A final experiment found that indenting the more compliant hard object at a higher force-rate than the less compliant one decreased their discriminability.

Experiment 1: Ink-based experiment with discrete displacements

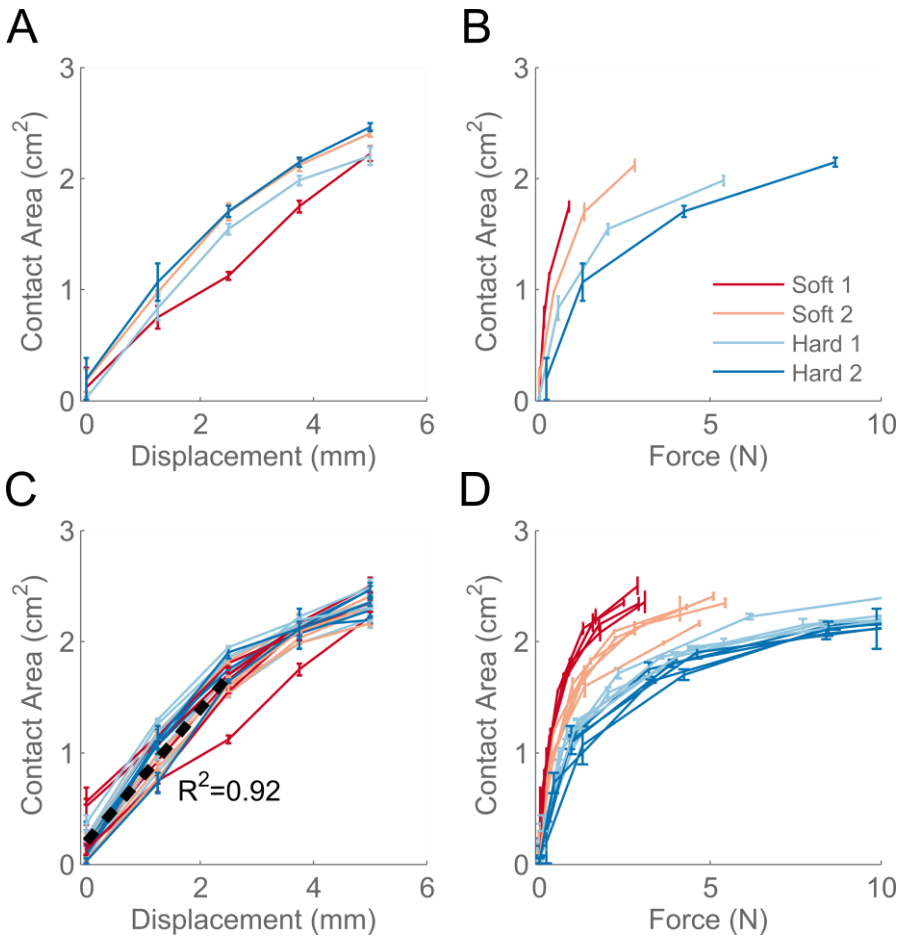

Figure 5. Biomechanical relationships of force, displacement, and contact area for 5 subjects. The ink-based method was used to measure the contact area along with force and displacement in biomechanics experiments with 5 subjects. A) Displacement-contact area relationships for all four stimuli per one example subject. B) Force-contact area relationships for the same subject, for displacements up to 3.75 mm. C) and D) plot the same as above but for all 5 subjects. In C) a line fits displacement to contact area from 0 to 2.5 mm.

Contact area, force, and displacement relationships were measured for 5 subjects at several discrete displacement levels (Figure 5). Force-contact area relationships appeared to be well separated for the two soft stimuli, but less so for the set of hard stimuli (Figure 5B, D). There was a consistent relationship between displacement and contact area across all individuals and stimulus compliances (Figure 5C). A linear relationship fit contact area to displacement up to 2.5 mm with an R^2 value of 0.92.

Experiment 2: Force-displacement experiment with continuous displacement

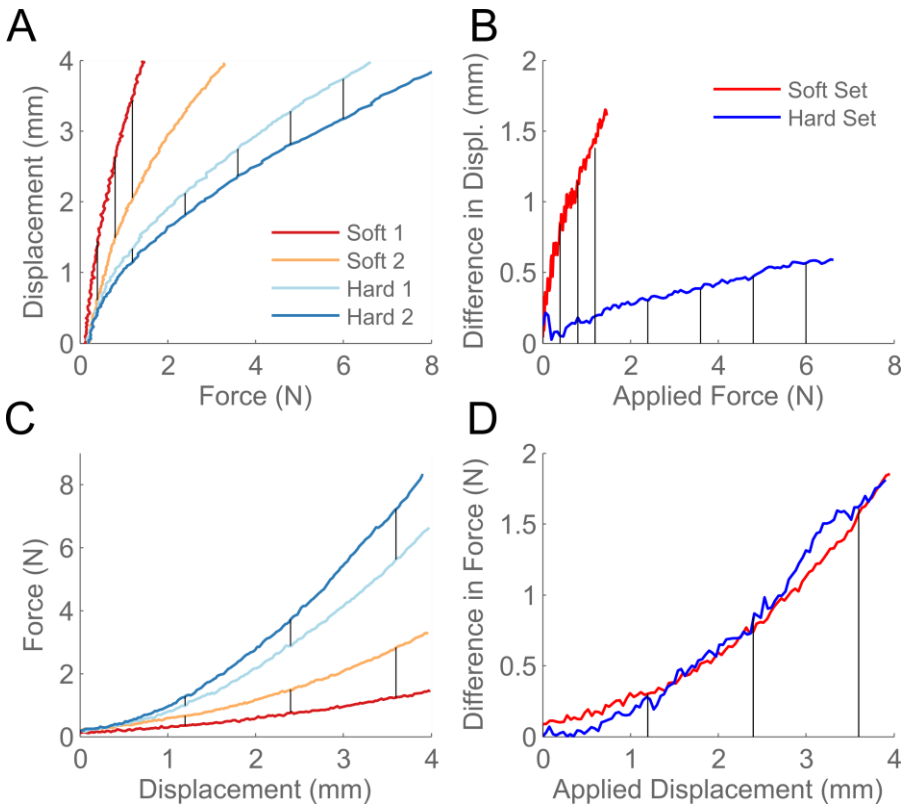

Figure 6. Force-displacement relationships and cue differences between stimuli. Each stimulus was indented into the finger pad of one subject at a rate of 1 mm/s from contact to 4 mm and force was measured at approx. 50Hz. A) Force-displacement relationships for each stimulus with force on the x-axis. Vertical lines between stimuli represent differences in displacement at a given force. B) Differences in displacement at the same applied force between the objects in each set. Vertical lines match those in A. C) The same force-displacement relationships in A., but plotted with displacement on the x-axis. Vertical lines represent differences in force at a given displacement. D) Force differences within each set at a given displacement.

We performed a separate biomechanics experiment with a single subject to examine the force-displacement relationships between finger pad and stimuli at greater resolution (Figure 6). Force was sampled at a high rate as each stimulus was pressed into the finger pad at a constant velocity. We found that inter-set differences in these relationships were not consistent between the hard and soft sets (Figure 6A, C). An equal force applied to both soft stimuli resulted in large displacement differences between them; however, an equal force applied to both hard stimuli resulted in much smaller displacement differences (Figure 6B). With an equal displacement applied to both stimuli, force differences were very similar within both the hard and soft sets (Figure 6D). The results suggest that the

soft stimuli may be more differentiable by indentation depth, when force is controlled, than hard stimuli.

Experiment 3: Psychophysical experiment with controlled indentation velocity or force-rate

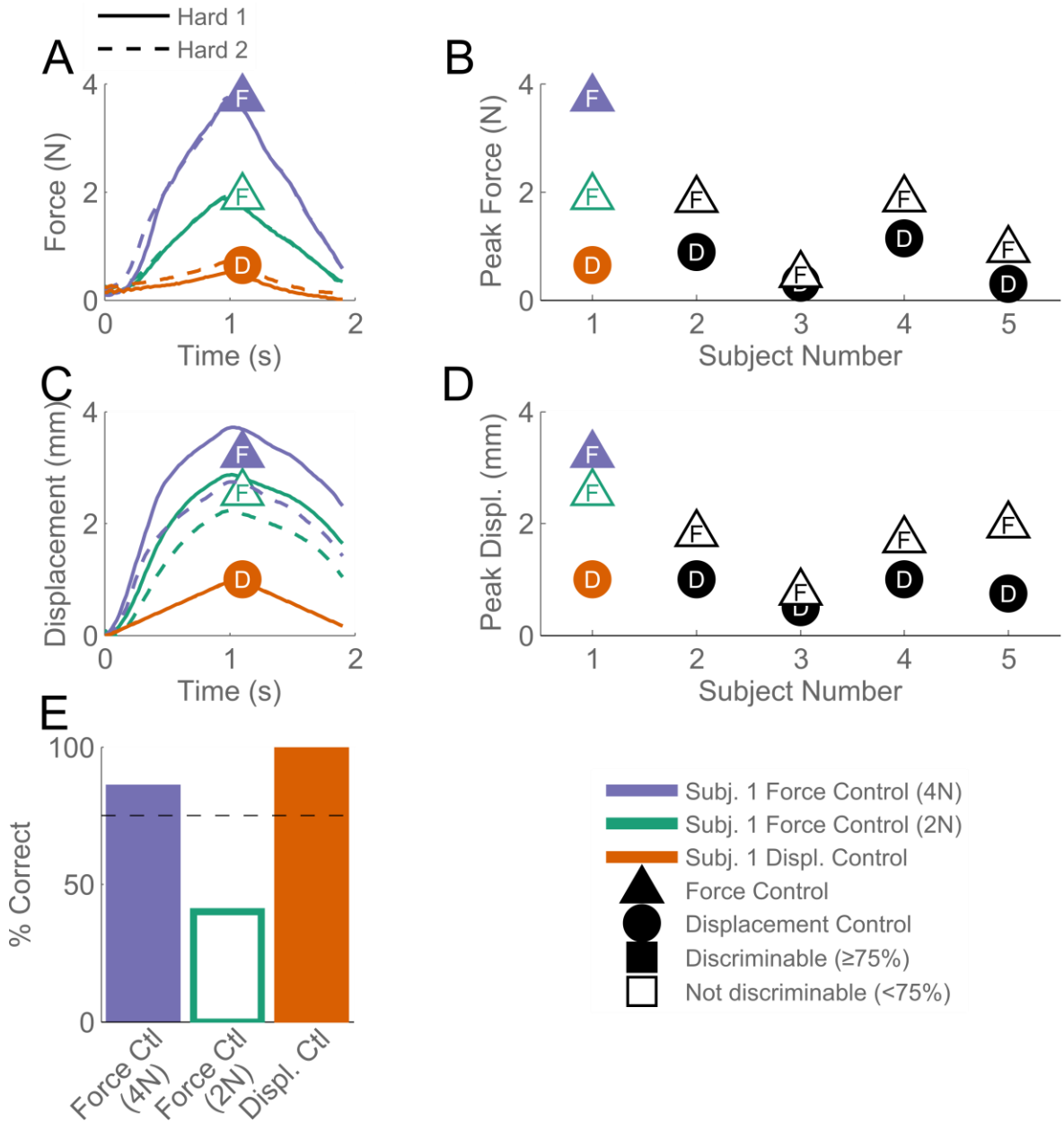

Figure 7. Psychophysical results in which force-rate or indentation velocities were controlled between stimuli. We systematically controlled force-rate and indentation velocity to determine their importance to softness discrimination. An example subject was run in 3 control modes: 1) force-rate controlled per stimulus to 4 N, 2) force-rate controlled per stimulus to 2 N, and 3) displacement-rate (indentation velocity) controlled per stimulus to 1 mm. A) Mean force values throughout the time-course of all trials for the example subject. The mean peak values for each control mode are marked with a triangle for force-controlled modes or a circle for displacement-controlled modes. B) Mean displacement values throughout the time-course of all trials for the example subject. C) Psychophysical responses given for each control mode with the example subjects. Only the force-controlled trials up to 2 N were not discriminable (<75% correct responses). D) Aggregate data in the force domain across all subjects. Each symbol marks the mean peak force from control modes as in A. E) Aggregate displacement data across all subjects.

Next we performed a psychophysical experiment with the hard set of stimuli in which either force-rate or indentation velocity (displacement-rate) was controlled between stimuli (Figure 7). These control modes attempted to replicate conditions along the independent axes of Figure 6B and D, in which force or contact area was controlled between stimuli through time. The total time of each indent was held constant within trials to exclude it from judgements. The psychophysical results for one subject are depicted in Figure 7E, in which objects were discriminable with force-rate controlled to 4 N and with displacement-rate controlled, but not with force-rate controlled to 2 N. Although the subject could not discriminate in this control mode, peak mean forces and displacements were much greater than those presented in the displacement-controlled trials (Figure 7A, C). With 4 additional subjects, we consistently found that stimuli were discriminable in velocity-controlled trials at much lower forces and displacements than force-rate controlled trails, despite some individual variability in discrimination ability (Figure 7B, D). Force-rate cues may therefore better discriminate hard objects than displacement-rate cues.

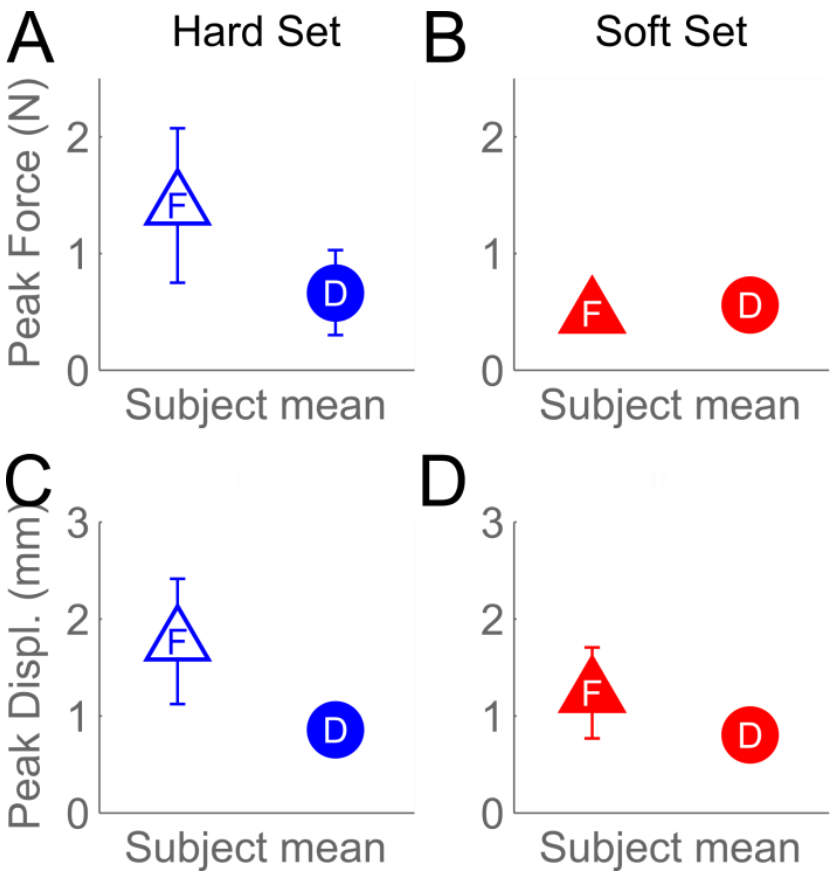

Figure 8. Effect of control mode on discriminability with hard and soft compliances. Aggregate results from all five subjects are plotted for all five subjects. A) Mean peak force used in force-controlled and displacement-controlled control modes for all subjects with the hard set of objects. B) Same as A. but with the soft set of objects. C) and D) plot data from A and B with mean peak displacement instead of force.

A similar psychophysical experiment was run with the soft stimuli (Figure 8). Trials employing both velocity and force-rate control modes were run at low forces with the soft stimuli and the same 5 subjects. In contrast to trials with the hard stimuli, the soft stimuli were consistently discriminable in both control modes at the smallest forces we could reliably deliver. These data suggest that the soft set may be equally discriminable by force-rate or displacement-rate; alternatively, subjects may not have been utilizing these cues at all in their judgements.

Experiment 4: Psychophysical Experiment with varied force-rates

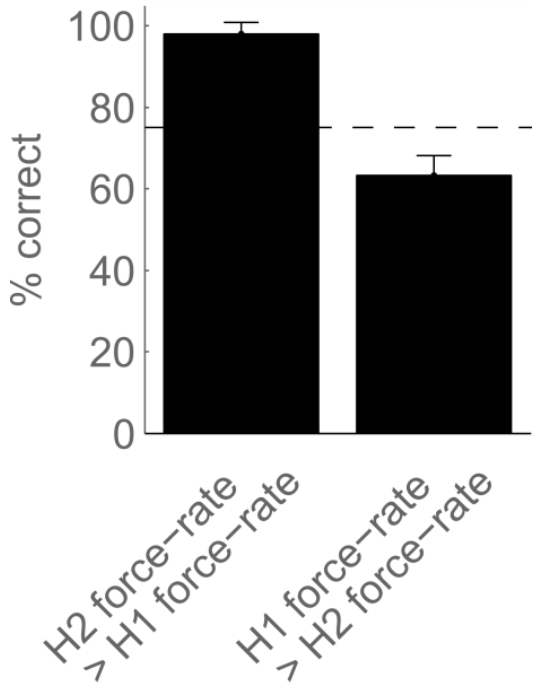

Figure 9. Psychophysical results in which force-rates were varied between stimuli. A comparison of correct responses when the Hard 2 stimulus was indented at a greater force rate than Hard 1 ($FR_{H2} > FR_{H1}$), and vice-versa ($FR_{H1} > FR_{H2}$),. Each stimulus was indented with triangle waves at either 1 N/s to 0.5 N (greater force rate) or 0.5 N/s to 0.5 N (lower force rate) within each trial.

We ran a final experiment with 2 subjects in which we indented the hard stimuli at different force-rates within a trial. In half of the trials, the softer stimulus was presented "correctly" at a lower force rate than the harder stimulus; in the other half, the softer stimulus was presented "incorrectly" at a higher force rate. Unlike in previous experiments, time was varied such that peak force was consistent within trials. Subjects were able to easily discriminate the objects (98% correct) when force-rate was lesser for the soft object. However, the subjects could no longer discriminate the objects (63% correct) when force-rate was greater.

CHAPTER 4: DISCUSSION

In this work we determined that temporal cues can be utilized by subjects to reduce the deformation necessary to discriminate two compliances harder than the finger pad. This is in contrast to prior work, which has suggested that the discriminability of two objects is based principally on the total deformation applied to them, and not how this deformation is applied [6, 19]. Specifically, indentation strategies which provide subjects with "correct" force-rate information—that is, a less compliant object presented at a higher force-rate—appear to improve discriminability. For compliances softer than the finger pad, the data is somewhat inconclusive. It is possible that the temporal cues do not impact judgements of these objects, or that we were not able to deliver forces low enough to observe their effect. Our findings are somewhat in accord with another study, in which increased stimulus deformation did not impact the Weber fraction as greatly for the more compliant stimuli used [19]. It does seem clear, however, that displacement-rate better differentiates softer objects than harder objects when force is controlled between stimuli.

Indentation velocity and contact area-rate may be related cues. It appears that across the full range of stimulus compliance used in our experiments, displacement and contact area had a consistent and linear relationship. If applied to our psychophysical results, this might suggest that contact area-rate is a better discriminator for soft objects than for hard objects. However, our biomechanical measurements were taken statically at discrete displacements, and further work is needed to confirm if contact area might encode for displacement throughout dynamic interaction with a stimulus.

We found force-rate to be an important cue for compliance discrimination with the bare finger. The results suggest that subjects are worse at detecting small differences in velocity or contact area-rate than they are at discerning differences in force-rate. This might explain why, in one study, subjects held the finger more perpendicular when exploring harder stimuli to magnify contact-area cues [19].

Integrated displays, which are able to render force-displacement relationships along with force-contact area relationships, may therefore be more effective in rendering harder stimuli [1].

The observed importance of force-rate cues is in contrast to previous experiments bare-finger experiments, which found that force-rate did not greatly impact discriminability by randomizing velocities [6]. However, unlike in previous experiments, we profiled force directly to ensure that force-rate was greater for the softer stimulus in half of the trials.

Our findings demonstrate that different presentation strategies may enhance the discriminability of two objects, and not solely the amount a subject deforms them [6, 19]. In passive-touch haptic displays, for example, presenting a rendered hard object at constant velocity might be a more efficient means of conveying softness information than presenting the virtual object at a force-rate. Further study is required to apply our results to active touch; however, it is possible that subjects apply certain exploration strategies to maximize time-dependent cues and reduce the deformation required to discriminate two compliances. Overall, it is vital to examine time-dependent cues as they relate to compliance discrimination, alongside steady-state biomechanical relationships.

CHAPTER 5: CONCLUSIONS AND FUTURE DIRECTION

In this work, we determined that the availability of force-rate cues, obtained by controlling displacement-rate between stimuli, influences discriminability in passive touch for compliances harder than the fingerpad. This is in direct contrast to prior results, which stated that force-rate does not impact the bare-finger discriminability of objects with deformable surfaces [6]. We observed that, in the absence of force-rate information, subjects required greater deformation of stimulus and fingerpad to discriminate two compliances. We also observed a phenomenon in which subjects were confused when a softer stimulus was indented at a higher force-rate than a harder one, which possibly resulted in a cue mismatch.Therefore further study should examine the role of force-rate in compliance discrimination, and how subjects may amplify this cue in active touch.

Overall, our results show that not only does the greater deformation of two stimuli enhance their discriminability, but also how this deformation is reached. Certain methods of exploring or presenting stimuli, such as with indentation velocity controlled between them, may reduce the deformation required to discriminate them. This finding may be important to the design of future haptic displays to increase their efficiency. In particular rendering force-rate and other time-dependent cues to reduce the necessary deformation of virtual objects may facilitate the use of novel electroactive polymers in compliance displays, which are currently capable of rendering only very small forces.

In accord with prior work, we concluded that higher terminal deformation of an object into the skin enhances discriminability, via either control mode. Regardless of whether force-rate information was available, it was still possible for subjects to discriminate compliances at very large deformation. It is possible that at these deformation levels another set of cues became sufficient for judgement, such as a stress distribution within the fingerpad skin or contact area-rate. While this study identified the

phenomenon, further study is required to determine which specific magnitudes of time-dependent and static relationship cues are sufficient for discriminability.

The novel method of measuring contact area with ink introduced in this work can be applied to other areas of haptics, such as contact with textural elements on rigid surfaces. The method is particularly useful as it does not interfere with finger-stimulus contact, and can be applied to interactions with a wide variety of objects. This is in contrast to other methods of measuring contact area, such as with a camera, which is limited to use with clear or translucent stimuli. A limitation of the method is that it only works at static displacements, and cannot be used to measure contact area continuously through an interaction.

The psychophysical and biomechanical experiments performed in this work were partially motivated by attempts to model populations of neural afferents encoding compliance information. Some preliminary finite-element simulations were performed with an axisymmetric model of the fingerpad in order to examine stress distributions within the skin. However, further modeling of compliance interactions which might lead to neural encoding was problematic. One main issue was that the importance of biomechanical cues is still unknown. Stress distributions, force-rate, and contact area might all need to be sampled from the model, among other cues. Another problem is placing these mechanical responses in a psychophysical context and time-scale. In order to study neural encoding which might relate to perception, it is important first to know which kinds of differences in compliance or biomechanical cues are perceptible, as well as what kind of timescales or presentation strategies are necessary to discriminate them. A final issue lies in modeling neural populations. It is unknown which types of neural afferents are most important with regards to compliance information, although prior work suggests that SAI and RA afferents are the most vital.

The results from this work can be used to further this effort to model afferent populations involved in compliance perception. Force-rate information, for example, is most likely carried by SAI afferents, although RA afferents may also be involved. Minute differences in contact area might be carried by the RA afferents, which respond very quickly to light touch. Therefore a neural population model should include at least SAI and RA afferents. Furthermore, data from the time-course of our psychophysical experiments can be used to reconstruct neural responses available during interaction with a stimulus. Along with the psychophysical responses recorded from these experiments, modeling population responses from our reconstructed experiments can be used in conjunction with signal detection theory order to predict perceptible differences in neuronal firing. It is crucial to combine biomechanical, psychophysical, and computational modeling experiments in order to examine the neural codes underlying perception.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Gregory Gerling for all his support and encouragement. I would also like to acknowledge Dr. Yuxiang Wang, who provided me with advice concerning both research and graduate school in general. Finally I would like to thank my committee members Dr. Shayn Peirce-Cottler and Dr. Silvia Blemker for their time, questions and input.

REFERENCES

[1] E. P. Scilingo, M. Bianchi, G. Grioli, and A. Bicchi, "Rendering softness: Integration of kinesthetic and cutaneous information in a haptic device," *Haptics, IEEE Transactions on,* vol. 3, pp. 109-118, 2010.

[2] A. Bicchi, E. P. Scilingo, and D. De Rossi, "Haptic discrimination of softness in teleoperation: the role of the contact area spread rate," *Robotics and Automation, IEEE Transactions on,* vol. 16, pp. 496-504, 2000.

[3] M. Bianchi and A. Serio, "Design and characterization of a fabric-based softness display," *IEEE transactions on haptics,* vol. 8, pp. 152-163, 2015.

[4] S. Yazdian, A. J. Doxon, D. E. Johnson, H. Z. Tan, and W. R. Provancher, "Compliance display using a tilting-plate tactile feedback device," in *2014 IEEE Haptics Symposium (HAPTICS)*, 2014, pp. 13-18.

[5] M. Bianchi, M. Poggiani, A. Serio, and A. Bicchi, "A novel tactile display for softness and texture rendering in tele-operation tasks," in *World Haptics Conference (WHC), 2015 IEEE*, 2015, pp. 49-56.

[6] M. A. Srinivasan and R. H. LaMotte, "Tactual discrimination of softness," *Journal of Neurophysiology,* vol. 73, pp. 88-101, 1995.

[7] R. Harper and S. Stevens, "Subjective hardness of compliant materials," *Quarterly Journal of Experimental Psychology,* vol. 16, pp. 204-215, 1964.

[8] R. M. Friedman, K. D. Hester, B. G. Green, and R. H. LaMotte, "Magnitude estimation of softness," *Experimental brain research,* vol. 191, pp. 133-142, 2008.

[9] W. M. B. Tiest and A. M. Kappers, "Cues for haptic perception of compliance," *Haptics, IEEE Transactions on,* vol. 2, pp. 189-199, 2009.

[10] M. Bianchi, E. Battaglia, M. Poggiani, S. Ciotti, and A. Bicchi, "A Wearable Fabric-based display for haptic multi-cue delivery," in *2016 IEEE Haptics Symposium (HAPTICS)*, 2016, pp. 277-283.

[11] S. C. Hauser and G. J. Gerling, "Measuring tactile cues at the fingerpad for object compliances harder and softer than the skin," in *2016 IEEE Haptics Symposium (HAPTICS)*, 2016, pp. 247-252.

[12] D. Lawrence, L. Y. Pao, A. M. Dougherty, M. Salada, and Y. Pavlou, "Rate-hardness: A new performance metric for haptic interfaces," *Robotics and Automation, IEEE Transactions on,* vol. 16, pp. 357-371, 2000.

[13] A. Metzger and K. Drewing, "Haptically perceived softness of deformable stimuli can be manipulated by applying external forces during the exploration," in *World Haptics Conference (WHC), 2015 IEEE*, 2015, pp. 75-81.

[14] V. Levesque and V. Hayward, "Experimental evidence of lateral skin strain during tactile exploration," in *Proceedings of EUROHAPTICS*, 2003.

[15] B. Delhaye, P. Lefèvre, and J.-L. Thonnard, "Dynamics of fingertip contact during the onset of tangential slip," *Journal of The Royal Society Interface,* vol. 11, p. 20140698, 2014.

[16] L. A. Baumgart, G. J. Gerling, and E. J. Bass, "Characterizing the range of simulated prostate abnormalities palpable by digital rectal examination," *Cancer epidemiology,* vol. 34, pp. 79-84, 2010.

[17] W. C. Carson, G. J. Gerling, T. L. Krupski, C. G. Kowalik, J. C. Harper, and C. A. Moskaluk, "Material characterization of ex vivo prostate tissue via spherical indentation in the clinic," *Medical engineering & physics,* vol. 33, pp. 302-309, 2011.

[18] C. Kowalik, G. Gerling, A. Lee, W. Carson, J. Harper, C. Moskaluk*, et al.*, "Construct validity in a high-fidelity prostate exam simulator," *Prostate cancer and prostatic diseases,* vol. 15, pp. 63-69, 2012.

[19] L. Kaim and K. Drewing, "Exploratory strategies in haptic softness discrimination are tuned to achieve high levels of task performance," *Haptics, IEEE Transactions on,* vol. 4, pp. 242-252, 2011.

APPENDIX 1: FORCE-RATE CONTROL CIRCUITRY

Custom circuitry was built to allow the XPS motion controller to profile force in time (Figure S1). The central mechanism to control force was to approximate dx/dF on-the-fly; that is, how much a small change in position caused a change in force. This value was then used to scale the velocity used for the analog tracking mode on the indenter. The overall circuit therefore set the velocity of the motion stage proportional to:

Equation 1. Force-rate control equation

$$velocity \sim (F_{current} - F_{desired}) * \frac{\Delta x}{\Delta F}$$

First, force output from the load cell was differentiated through a differentiator op-amp circuit to obtain dF/dt. Next, velocity was continually output from the XPS motion controller, giving dx/dt. The absolute values of these were taken using additional circuitry, as the Arduino does not accept negative input voltages; additionally, we can assume that in all cases, dx/dF > 0, as pushing into further into the stimulus will not decrease the force. The Arduino continually performs division on these inputs $\left|\frac{dx}{dt}\right| / \left|\frac{dF}{dt}\right|$ to approximate dx/dF.

A digital-to-analog converter on the motion controller was used to profile the desired forces, updated at a rate of approximately 300 Hz. An instrumentation amp subtracted this value from the current load cell value to obtain an error signal. The resistance determining the gain on the instrumentation amp was set by an AD5290 digital potentiometer. The AD5290 was set by the Arduino such that the gain of the amplifier was approximately equal to the calculated dx/dF.

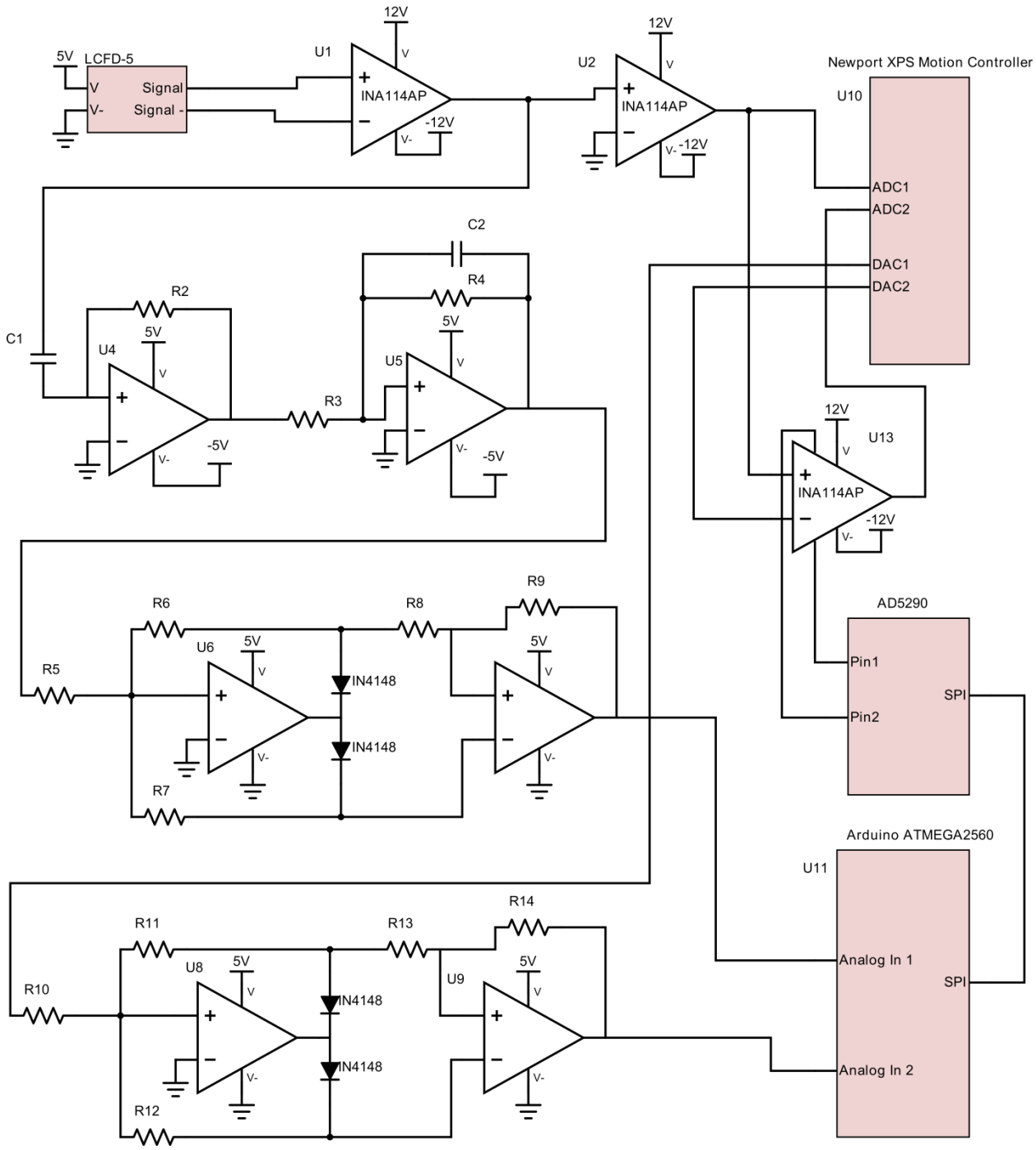

Figure S1. Generalized Schematic of Force-Control circuit. A basic schematic showing the design of the circuit used for profiling force on the load cell. Resistor values: R_{U1}= 56 Ω. R_{U2}= 33 kΩ. R_{U13}= 5.6 kΩ. R2=200 kΩ. R3=1 kΩ. R4=10 kΩ. R5=R6=R7=R8=R9=R10=R11=R12=R13=R14=10 kΩ. Capacitor values: C1=1 µF. C2=14.7 µF. Not pictured: an extra op-amp further amplifies the difference from U13 using the AD5290 on the input and 3 kΩ feedback.

APPENDIX 2: CODE

XPS motion control and psychophysics software

There were 12 files used with the XPS Motion controller and control of psychophysics experiments.

1. **MotionController.py**—main wrapper of all commands related to motion or data collection, also schedules and manages indenter tasks asynchronously
2. **Psychophysics_Window.py**—main GUI for running psychophysics experiments
3. **Load_Cell.py**—all load cell functions, such as sampling force and calibrating
4. **Indenter.py**—movement of the linear motion stage
5. **Stimulus_Selector.py**—switching between stimuli using the 3D printed device
6. **Stimulus.py**—data model for stimuli, stores data associated with the stimuli
7. **Procedure.py**—class to store a schedule of tasks for a MotionController object to follow
8. **Procedure_Window.py**—GUI to display list of current and upcoming tasks scheduled for a MotionController instance
9. **Psychophysics_Experiment.py**—data model for a psychophysical experiment, storing number of runs of each type and shuffling order
10. **Results_Window.py**—GUI to show psychophysical results
11. **Calibration_Window.py**—GUI that allows more unrestricted control over the indenter for testing or other purposes
12. **CONSTANTS.py**—all constants for indenter control, allows flexibility between configurations or indenters

MotionController.py

```
__author__ = 'Steven'
from Load_Cell import *
from Indenter import *
from XPS_GPIO import *
from Stimulus_Selector import *
import XPS_Q8_drivers
import threading
from CONSTANTS import *
import multiprocessing
import random
import copy
from Procedure import Procedure


class Motion_Controller:

    def __init__(self):
        self.get_socket_lock = threading.Lock()
        #these need to be done first
        self.xps = XPS_Q8_drivers.XPS()

        self.GROUP_NAME = CONSTANTS.INDENTER_GROUP_NAME

        self.sockets = threading.local()
        self.get_socket_lock = threading.Lock()

        #Main Thread Socket
        self.socket = self.get_new_socket()
        self.abort_socket = self.get_new_socket()
        self.current_position_socket = self.get_new_socket()

        self.procedure_thread = threading.Thread()
        self.lock = threading.Lock()
        self.running_procedure = False
        self._should_stop_running_procedure = False
        self.current_command = None
        self.command_list = []
        self.run_id_lock = threading.Lock()
        self.run_id = 0

        #create objects
        self.load_cell = Load_Cell(self.xps,self.get_new_socket())
        self.indenter = Indenter(self.xps,self.get_new_socket())
        self.stimulus_selector = Stimulus_Selector()
        self.allowing_joystick = False
```

```
        self._is_gathering = False
        self._should_stop_gathering = False
        self.gathering_lock = threading.Lock()
        self.load_cell_recording = []
        self.indenter_position_recording = []
        self.time_recording = []
        self.gathering_sock = self.get_new_socket()

        self.funcdict = {
            'MOVE_INDENTER_HOME':self.move_indenter_home,
            'MOVE_INDENTER_DOWN':self.move_indenter_down,
            'MOVE_INDENTER_UP':self.move_indenter_up,
            'MOVE_INDENTER_TO_TARGET':self.indenter.move_to_target,
            'ABORT_AT_FORCE':self.abort_at_force,
            'WAIT_FOR_TIME':self.wait_for_time,
            'ABORT_INDENTER':self.abort_indenter,
            'RESET':self.reset,
            'CLEAR_EVENTS':self.clear_events,
            'SWITCH_STIMULI':self.switch_stimuli,
            'SET_INDENTER_MODE':self.set_indenter_mode,
            'SET_MOTION_PARAMS':self.indenter.set_motion_params,
            'SET_JOGGING_PARAMS':self.indenter.set_jogging_params,
            'CALIBRATE_LOAD_CELL_ZERO':self.calibrate_load_cell_zero,
            'SET_INDENTER_HOME_AT_CURRENT_POS':self.set_indenter_home_at_current_pos,
            'TRACE_FORCE':self.trace_force,
            'INDENTER_STOP_JOGGING':self.indenter.end_jogging,
            'INDENTER_BEGIN_JOGGING':self.indenter.begin_jogging,
            'INDENTER_BEGIN_MOTION':self.indenter.begin_motion,
            'INDENTER_END_MOTION':self.indenter.end_motion,
            'INDENTER_ENABLE_JOYSTICK':self.enable_joystick,
            'INDENTER_DISABLE_JOYSTICK':self.disable_joystick,
            'SELECT_STIMULUS':self.select_stimulus,
            'SET_INDENTER_LIMIT':self.set_indenter_limit,
            'BEGIN_GATHERING':self.begin_gathering,
            'STOP_GATHERING':self.stop_gathering,
        }
        self._output_velocity()
        self.reset()

        #TODO calibration?


#####################################SOCKETS#############################################
##############################
    def get_new_socket(self):
        with self.get_socket_lock:
            socket_num = self.xps.TCP_ConnectToServer('192.168.0.254',5001,30)
```

```
            if socket_num == 0: self.xps.CloseAllOtherSockets(socket_num)
            return socket_num


    #TODO
    ############################ACTIONS YOU CAN ALWAYS
CALL!!#################################################################
    def get_load_cell_force(self):
        return self.load_cell.get_force()

    def enable_joystick(self):
        with self.lock: self.allowing_joystick = True

    def disable_joystick(self):
        with self.lock: self.allowing_joystick = False

    def input_joystick(self,val):
        with self.lock: flag = self.allowing_joystick
        if flag:
            self.indenter.set_jogging_params(val)

    def get_indenter_position(self):
        return self.indenter.get_current_position(self.current_position_socket)

    #BIG ISSUE HERE IT CAN RETURN STUPID THINGS
    def get_current_command(self):
        with self.lock:
            command = copy.deepcopy(self.current_command)
            command_list = copy.deepcopy(self.command_list)
        if command is None or command > len(command_list)-1:
            command = 0
            command_list = ['Halted']
        return [command,command_list]

    def get_selected_stimulus(self):
        return self.stimulus_selector.get_selected_stimulus()

    def stop_procedure(self):
        self.xps.GroupMoveAbort(self.abort_socket, self.GROUP_NAME)
        self.xps.GroupAnalogTrackingModeDisable(self.abort_socket,self.GROUP_NAME)
        self.xps.GroupJogModeDisable(self.abort_socket,self.GROUP_NAME)
        self.reset()
        with self.lock:
            self._should_stop_running_procedure = True
        while self.is_running_procedure():
            self.xps.GroupMoveAbort(self.abort_socket, self.GROUP_NAME)
        with self.lock: self._should_stop_running_procedure = False
        self.xps.GroupMoveAbort(self.abort_socket, self.GROUP_NAME)
```

```
    def is_running_procedure(self):
        with self.lock: return self.running_procedure

    def get_run_id(self):
        ret = 0
        with self.run_id_lock: ret = self.run_id
        return ret

    def run(self,procedure):
        if isinstance(procedure,Procedure):
            procedure = procedure.command_list
        if self.is_running_procedure():
            self.stop_procedure()
        self.process = threading.Thread(target=self._run_procedure,args=(procedure,))
        self.process.start()

    ####################################PROCEDURE
MANAGEMENT#########################################################################
    def should_stop_running_procedure(self):
        ret = self._should_stop_running_procedure
        with self.lock: return ret

    def _run_procedure(self,command_list):
        with self.lock:
            self.command_list = command_list
            self.running_procedure = True
        # print(command_list)
        ind = 0
        for command in command_list:
            with self.lock: self.current_command = ind
            if self.should_stop_running_procedure():
                break
            command_name = command[0]
            if len(command) > 1:
                command_params = command[1:]
    #           print(command_name,command_params)
                self.funcdict[command_name](*command_params)
            else:
    #           print(command_name)
                self.funcdict[command_name]()
            ind = ind + 1
        with self.lock:
            self.running_procedure = False
        with self.run_id_lock:
            self.run_id = self.run_id+1
            self.current_command = None
```

```
        self.command_list = []


    ######################################IN-PROCEDURE
COMMANDS#############################################################
    def reset(self):
        if not self.is_running_procedure():
            self.run([('RESET',)])
        else:
            self.abort_indenter()
            self.indenter.restore_defaults()

            #TODO TAKE THIS OUT
            #TODO DO OUTPUT VELOCITYYY
            self.clear_events()
        #    self._output_velocity()


    def set_indenter_home_at_current_pos(self):
         if not self.is_running_procedure():
            self.run([('SET_INDENTER_HOME_AT_CURRENT_POS',),])
         else:
            self.indenter.set_home_position(self.indenter.get_current_position())

    def set_indenter_limit(self,max):
        if not self.is_running_procedure():
            self.run([('SET_INDENTER_LIMIT',max)])
        else:
            min = self.indenter.get_home_position()
            [a,b] = self.xps.PositionerUserTravelLimitsSet(self.socket,
CONSTANTS.INDENTER_POSITIONER_NAME, min, max)
#           print(a,b)

    def should_stop_gathering(self):
        with self.gathering_lock: ret = self._should_stop_gathering
        return ret

    def _gathering_loop(self,delay):
        start_t = time.time()
        while not self.should_stop_running_procedure():
            self.indenter_position_recording.append(self.indenter.get_current_position(self.gathering_sock))
            self.load_cell_recording.append(self.load_cell.get_force())
            self.time_recording.append(time.time()-start_t)
            if self.should_stop_gathering():
                with self.gathering_lock:
                    self._is_gathering = False
                    self._should_stop_gathering = False
                return
```

```
            time.sleep(float(delay)/1000.)

    def is_gathering(self):
        with self.gathering_lock: ret = self._is_gathering
        return ret

    #10s max
    def begin_gathering(self,delay=1):
        if self.is_gathering(): return
        else:
            self.load_cell_recording = []
            self.indenter_position_recording = []
            self.time_recording = []
            with self.gathering_lock: self._is_gathering = True
            targ = threading.Thread(target=self._gathering_loop,args=(delay,))
            targ.start()

     #    print(self.xps.EventExtendedAllGet(self.socket))
     # #   return
     #    if not self.is_running_procedure():
     #        self.run([('BEGIN_GATHERING',)])
     #    else:
     #        #setpointposition?
     #        #self.xps.TimerSet(self.socket,'Timer1',8)
     #        [a,b] = self.xps.GatheringReset(self.socket)
     #        print('resetting?',a,b)
     #        [a,b] =
self.xps.GatheringConfigurationSet(self.socket,[CONSTANTS.INDENTER_POSITIONER_NAME+'.CurrentPo
sition',GPIO.PINS[GPIO.PIN_LOADCELL_IN]])
     #        print(a,b)
     #        [a,b] = self.xps.EventExtendedConfigurationTriggerSet(self.socket,['Immediate',], ['0',], ['0',],
['0',], ['0',])
     #        print('IMPORTANTSTUFF',a,b)
     #        [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,['GatheringRun',],['10000',],['8',],['0',],['0',])
     #
     #        print('IMPORTANTSTUFF',a,b)
     #        [a,b] = self.xps.EventExtendedStart(self.socket)
     #        print('import3',a,b)

    def stop_gathering(self,time_,position,force):
        if self.is_gathering():
            with self.gathering_lock: self._should_stop_gathering = True
            while self.is_gathering(): pass
        position[:] = self.indenter_position_recording
        force[:] = self.load_cell_recording
        time_[:] = self.time_recording
```

```
    # if not self.is_running_procedure():
    #     self.run([('STOP_GATHERING',)])
    # else:
    #   #  self.xps.GatheringStop(self.socket)
    #     [a,b] = self.xps.GatheringStopAndSave(self.socket)
    #     print(a,b)
    #     [a,disp] = self.xps.GatheringDataGet(self.socket,[0,])
    #     [b,force] = self.xps.GatheringDataGet(self.socket,[1,])
    #    # self.xps.GatheringReset(self.socket)
    #     print(a,disp)
    #     print(b,force)

def trace_force(self,t,f):
    #TODO modify params below
    #TODO should stop waiting etc
    if not self.is_running_procedure():
        self.run([('TRACE_FORCE',t,f)])
    else:
        self.stimulus_selector.ser.write(str(-1)+'\n')
        '''self._output_DAC(GPIO.PIN_ANALOG_OUT,1)
    return'''
        # t = [0, 3, 4, 7,10,]
        # v = [-9.2, -7.5, -7.5, -9.2,-9.2]
        # f =[0,1,1,0,0]

     #  t = [0, 2, 5, 7,]
     #  v = [-9.2, -4.5, -4.5, -9.2]

     #   t = np.linspace(0,10,60000)
     #   v = (((np.sin((t*(2*np.pi))))+1)*.5)-4.5

     #   t = np.linspace(0,10,60000)
     #   v = (((np.sin((t*(1*2*np.pi))))+1)*.5)-9


        t2f = scipy.interpolate.interp1d(t,f,'linear')
        self._output_velocity()
        self.indenter.set_tracking_params(0,-.75,1,15,8)
    #    self.indenter.set_tracking_params(0,-.55,1,15,8)
        self.indenter.begin_tracking()
        tstart = time.time()
        while True:
            if self.should_stop_running_procedure():
                break
            tcurr = time.time()-tstart
            if tcurr > max(t):
                break
```

```
            else: self._output_DAC(GPIO.PIN_ANALOG_OUT,self.load_cell._F2D(t2f(tcurr)))
        self.indenter.end_tracking()
        self.stimulus_selector.ser.write(str(-1)+'\n')



    def _output_DAC(self,pin,voltage):
        [a,b] = self.xps.GPIOAnalogSet(self.socket,[GPIO.PINS[pin],],[voltage,])

    def select_stimulus(self,stim_num):
        if not self.is_running_procedure():
            self.run([('SELECT_STIMULUS',stim_num)])
        else:
            self.stimulus_selector.select(stim_num)


    #TODO this should be in procedure
    def indent_to_force(self,force,velocity=None,acceleration=None):
        if not self.is_running_procedure():
            self.run([('INDENT_TO_FORCE',force,velocity,acceleration)])
        else:
            old_velocity = self.indenter.get_set_velocity()
            old_acceleration = self.indenter.get_set_acceleration()
            ##TODO
            old_velocity = 2
            old_acceleration = 3
            if velocity is None: velocity=old_velocity
            if acceleration is None: acceleration = old_acceleration

            self.indenter.set_motion_params(velocity,acceleration)
            self.indenter.move_to_target(self.indenter.get_max_indentation(),blocking=False)
            self.wait_for_force(force)
            self.abort_indenter()
            self.wait_for_time(1)

self.indenter.set_motion_params(self.indenter.get_default_velocity(),self.indenter.get_default_accelera
tion())
            self.indenter.move_to_home()

    def move_indenter_home(self,blocking=True):
        if not self.is_running_procedure():
            self.run([('MOVE_INDENTER_HOME',blocking)])
        else:
 #          print('MOVING HOME?')
            self.indenter.move_to_home(blocking)
 #          print('AAAAAND DONE')

    def move_indenter_down(self,amt,blocking=True):
```

```
    if not self.is_running_procedure():
        self.run([('MOVE_INDENTER_DOWN',amt,blocking)])
    else:
        self.indenter.move_down(amt,blocking)

def move_indenter_up(self,amt,blocking=True):
    if not self.is_running_procedure():
        self.run([('MOVE_INDENTER_UP',amt,blocking)])
    else:
        self.indenter.move_up(amt,blocking)

def set_indenter_mode(self,mode):
    if not self.is_running_procedure():
        self.run([('CHANGE_INDENTER_MODE',)])
    else:
        pass
#       print(self.indenter.get_status())

#TODO check for true and/or false!!
def abort_indenter(self):
    if not self.is_running_procedure():
        self.run([('ABORT_INDENTER',)])
    else:
        self.xps.GroupMoveAbort(self.abort_socket, self.GROUP_NAME)

#TODO
def abort_at_force(self,force,lim='high'):
    if not self.is_running_procedure():
        self.run(('ABORT_AT_FORCE',force,lim))
    else:
        print('lim??',lim)
        id = self._set_force_limit(force,lim)
        [error_str,ret_str] = self.xps.EventExtendedAllGet(self.socket)
        print('important',error_str,ret_str)
        while True:
      #     print('in hurr')
            if self.should_stop_running_procedure():
                break
      #     print('past hurr?')
            if self.load_cell.get_force() >= force:
                pass
                '''self._remove_id(id)
                break'''
       #    print('even hrur')
            if not self._check_id_active(id):
                print('im in here!!')
                break
      #     print('but not here')
```

```
    def _check_id_active(self,id):
        [error_str,ret_str] = self.xps.EventExtendedAllGet(self.socket)
        elist = ret_str.split(',')
        if error_str != 0: return False
        if id in elist: return True
        else: return False

    def _remove_id(self,id):
        self.xps.EventExtendedRemove(self.socket,id)

    #VELOCITY IS DIVIDED BY 2!
    #TODO current velocity? is listed as one option
    def _output_velocity(self):
        [a,b] = self.xps.EventExtendedConfigurationTriggerSet(self.socket,['Always',], ['0',], ['0',], ['0',], ['0',])
        [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,[str(GPIO.PINS[GPIO.PIN_VELOCITY_OUT])+'.D
ACSet.SetpointVelocity',],['GROUP1.POSITIONER',],['1',],['.013',],['0',])
    #    print('IMPORTANTSTUFF',a,b)
    #    print('IMPORTANTSTUFF',a,b)
        [a,b] = self.xps.EventExtendedStart(self.socket)
    #    print('OUTP_VELOCITY',a,b)
    #     [a,b] = self.xps.EventExtendedConfigurationTriggerSet(self.socket,['Always',], ['0',], ['0',], ['0',],
['0',])
    #     [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,[str(GPIO.PINS[GPIO.PIN_POSITION_OUT])+'.
DACSet.SetpointPosition',],['GROUP1.POSITIONER',],['.1',],['.013',],['0',])
    # #    print('IMPORTANTSTUFF',a,b)
    #     [a,b] = self.xps.EventExtendedStart(self.socket)
    # #    print('OUTP_VELOCITY',a,b)

    def wait_for_time(self,duration):
        if not self.is_running_procedure():
            self.run([('WAIT_FOR_TIME',duration)])
        else:
            start_time = time.time()
            while time.time()-start_time < duration:
                if self.should_stop_running_procedure():
                    break

    def clear_events(self):
        if not self.is_running_procedure():
            self.run([('CLEAR_EVENTS',)])
        else:
     #      print('list..??',self.xps.EventListGet(self.socket))
     #      print('list2..?',self.xps.ActionExtendedListGet(self.socket))
            [error_str,ret_str] = self.xps.EventExtendedAllGet(self.socket)
     #      print('CLEARING!!',error_str,ret_str)
```

```
        elist = ret_str.split(',')
        if error_str == 0:
            for event in elist:
    #             print('event',event)
                self.xps.EventExtendedRemove(self.socket,int(event))
        id = 0
        while True:
            [a,b] = self.xps.EventExtendedRemove(self.socket,id)
            if a != 0:
                break
            id = id + 1

    def switch_stimuli(self,stimulus):
        if not self.is_running_procedure():
            self.run([('SWITCH_STIMULI',stimulus)])
        else:
            self.stimulus_selector.select(stimulus)

self.indenter.set_max_indentation(self.stimulus_selector.get_selected_stimulus().get_max_indentation(
))

    def calibrate_load_cell_zero(self,blocking=False):
        if not self.is_running_procedure():
            self.run([('CALIBRATE_LOAD_CELL_ZERO',blocking)])
        else:
            self.load_cell.calibrate_zero()
            if blocking:
                while self.load_cell.is_calibrating_zero():
                    pass

    #TODO these are reversed
    #TODO they should also return something... like the id or something
    def _set_force_limit(self,force,lim='high'):
        print("AIM I BEING CALLED!?")
        D = self.load_cell._F2D(force)
        print('digitallim',D)
        print(self.load_cell.get_output_voltage())
        if lim == 'high':
            print(str(GPIO.PINS[GPIO.PIN_LOADCELL_IN]))
            [a,b] =
self.xps.EventExtendedConfigurationTriggerSet(self.socket,[str(GPIO.PINS[GPIO.PIN_LOADCELL_IN])+'.A
DCHighLimit','GROUP1.POSITIONER.SGamma.MotionState'], ['-7.','0'], ['0','0'], ['0','0'], ['0','0'])
            [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,['GROUP1.POSITIONER1.MoveAbort'],['0',],['0'
,],['0',],['0',])
            [a,b] = self.xps.EventExtendedStart(self.socket)
            print('quick',self.xps.EventExtendedAllGet(self.socket))
            print('HERESTUFF',a,b)
```

```
            return b
        else:
            [a,b] =
self.xps.EventExtendedConfigurationTriggerSet(self.socket,[str(GPIO.PINS[GPIO.PIN_LOADCELL_IN])+'.A
DCLowLimit','GROUP1.POSITIONER.SGamma.MotionState'], [str(D),'0'], ['0','0'], ['0','0'], ['0','0'])
            [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,['GROUP1.POSITIONER1.MoveAbort'],['0',],['0'
,],['0',],['0',])
            [a,b] = self.xps.EventExtendedStart(self.socket)
            return b
```

Psychophysics_Window.py

```
__author__ = 'Steven'
import Tkinter as tk
from Indenter import *
from Load_Cell import *
from XPS_GPIO import *
from Results_Window import *
import time
import _tkinter
import PIL.Image
from PIL import ImageTk
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure
import cPickle
from Table import *
import XboxController
from Procedure import Procedure
import tkMessageBox as messagebox
import threading
from Psychophysics_Experiment import *
from Motion_Controller import *
import tkFileDialog
from Procedure_Window import Procedure_Window

class Psychophysics_Window:

   def __init__(self):
      #TODO DATA REFRESH MESSES WITH FORCE CONTROL?
      self.DATA_REFRESH_TIME = 200 #ms
      self.JOYSTICK_REFRESH_TIME = 10 #ms
      self.BETWEEN_TRIAL_TIME = 100 #ms

      self.motion_controller = Motion_Controller()
      self.experiment = Experiment()

      self.xboxCont = XboxController.XboxController(controllerCallBack = None,joystickNo = 0,deadzone =
0.1,scale = 1,invertYAxis = False)
      self.xboxCont.start()
      self.xboxCont.setup_vibration()

      self.is_using_joystick = False
      self.waiting_for_input = False
      self.experiment_running = False
      self.current_run = None
      self.running_trials = False
```

```
        self.info_disabled = True
        self.repeat_pressed = False
        self.first_trial = True

        self.root = tk.Tk()
        self.root.title("University of Virginia Psychophysics Controller")
        self.results_win = Results_Window(self.root,self.experiment)

        self.PADY = 10
        self.BAR_WIDTH = 300
        self.BAR_HEIGHT = 20
        self.RED = '#FF7F50'
        self.WHITE = '#ffffff'
        self.BLUE = '#87CEFA'

        self.lcf_high = CONSTANTS.LOADCELL_MAX_FORCE
        self.lcf_low = 0 #N

        self.STARTUP_FRAME = 0
        self.STIMULUS_INFO_FRAME = 1
        self.CALIBRATION_FRAME = 2
        self.INFO_FRAME = 3
        self.INPUT_FRAME = 4
        self.PROGRESS_FRAME = 5
        self.EXPERIMENT_INFO_FRAME = 6

        self.A = 0
        self.B = 1
        self.X = 2
        self.Y = 3
        self.button_binds = {}
        self.toggles = {self.A:0,self.B:0,self.X:0,self.Y:0}

        self.BANNER_WIDTH=420
        self.BANNER_HEIGHT=145
        image = PIL.Image.open('UVA.gif')
        image = image.resize((self.BANNER_WIDTH, self.BANNER_HEIGHT), PIL.Image.ANTIALIAS) #The
(250, 250) is (height, width)
        self.uva_im = ImageTk.PhotoImage(image)
        self.uva_banner_can =
tk.Canvas(self.root,width=self.BANNER_WIDTH,height=self.BANNER_HEIGHT)
        self.uva_banner_can.grid()
        self.uva_banner =
self.uva_banner_can.create_image(self.BANNER_WIDTH/2,self.BANNER_HEIGHT/2,image=self.uva_im)

        self.frames = {}
        self.frames[self.STARTUP_FRAME] = self.construct_startup_frame(1)
        self.frames[self.STIMULUS_INFO_FRAME] = self.construct_stimulus_info_frame(2)
```

```
        self.frames[self.CALIBRATION_FRAME] = self.construct_calibration_frame(4)
        self.frames[self.INFO_FRAME] = self.construct_info_frame(6)
        self.frames[self.INPUT_FRAME] = self.construct_input_frame(7)
        self.frames[self.PROGRESS_FRAME] = self.construct_progress_frame(5)
        self.frames[self.EXPERIMENT_INFO_FRAME] = self.construct_experiment_info_frame(3)

        self.hide(self.INFO_FRAME)
        self.hide(self.INPUT_FRAME)
        self.hide(self.CALIBRATION_FRAME)
        self.hide(self.STIMULUS_INFO_FRAME)
        self.hide(self.PROGRESS_FRAME)
        self.hide(self.EXPERIMENT_INFO_FRAME)

        #bindings
        self.root.bind('<Up>', self.upKey)
        self.root.bind('<Down>', self.downKey)
        self.root.bind('<Right>', self.rightKey)

        self.root.focus_set()
        self.file_opt = options = {}
        options['defaultextension'] = '.xml'
        options['filetypes'] = [('Experiment XML files', '.xml'), ('text files', '.txt'), ('all files', '.*')]
        #options['initialdir'] = 'C:\\'
        options['initialfile'] = '.xml'
        options['parent'] = self.root
        options['title'] = 'Save Simulation As...'

        self.procedure_window = Procedure_Window(self.root,self.motion_controller)

        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
        self.root.after(self.BETWEEN_TRIAL_TIME,self.run_trial)
        self.root.after(self.DATA_REFRESH_TIME,self.data_refresh)
        self.root.after(self.JOYSTICK_REFRESH_TIME,self.joystick_refresh)
        self.root.mainloop()

    def on_closing(self):
        #if messagebox.askokcancel("Quit", "Do you want to quit?"):
            self.stop_experiment_button_pressed()
            self.root.destroy()

    def enable_joystick(self):
        self.is_using_joystick = True
      #  self.motion_controller.indenter.begin_jogging()

    def disable_joystick(self):
        self.is_using_joystick = False
        self.xboxCont.set_vibration(0,0,0)
      #  self.motion_controller.indenter.end_jogging()
```

```
def joystick_refresh(self):
    if self.xboxCont.A == 1 and self.toggles[self.A]==0:
        self.toggles[self.A] = 1
    #    print('yo')
        self.button_binds[self.A]() #self.stop_experiment_button_pressed()
    elif self.xboxCont.A == 0 and self.toggles[self.A]==1:
    #    print('yo2')
        self.toggles[self.A] = 0
    if self.xboxCont.B == 1 and self.toggles[self.B]==0:
        self.toggles[self.B] = 1
        self.button_binds[self.B]() #self.stop_experiment_button_pressed()
    elif self.xboxCont.B == 0 and self.toggles[self.B]==1:
        self.toggles[self.B] = 0
    if self.xboxCont.X == 1 and self.toggles[self.X]==0:
        self.toggles[self.X] = 1
        self.button_binds[self.X]() #self.stop_experiment_button_pressed()
    elif self.xboxCont.X == 0 and self.toggles[self.X]==1:
        self.toggles[self.X] = 0
    if self.xboxCont.Y == 1 and self.toggles[self.Y]==0:
        self.toggles[self.Y] = 1
        self.button_binds[self.Y]() #self.stop_experiment_button_pressed()
    elif self.xboxCont.Y == 0 and self.toggles[self.Y]==1:
        self.toggles[self.Y] = 0
    if self.is_using_joystick:
        modifier = 1
        if self.xboxCont.RB == 1: modifier = 5
        val = self.xboxCont.RTHUMBY*modifier
        self.motion_controller.input_joystick(val)
       # self.motion_controller.indenter.set_jogging_params(val)
    self.root.after(self.JOYSTICK_REFRESH_TIME,self.joystick_refresh)

def hide(self,frameID):
    for child in self.frames[frameID].winfo_children():
        child.grid_remove()
    self.frames[frameID].grid_remove()

def hide_frame(self,frame):
    for child in frame.winfo_children():
        child.grid_remove()
    frame.grid_remove()

def unhide_frame(self,frame):
    for child in frame.winfo_children():
        child.grid()
    frame.grid()

def enable(self,frameID):
```

```
        for child in self.frames[frameID].winfo_children():
            child.configure(state=tk.NORMAL)

    def enable_frame(self,frame):
        for child in frame.winfo_children():
            child.configure(state=tk.NORMAL)

    def disable_frame(self,frame):
        for child in frame.winfo_children():
            child.configure(state=tk.DISABLED)

    def unhide(self,frameID):
        for child in self.frames[frameID].winfo_children():
            child.grid()
        self.frames[frameID].grid()

    def disable(self,frameID):
        for child in self.frames[frameID].winfo_children():
            child.configure(state=tk.DISABLED)

    def construct_experiment_info_frame(self,row):
        self.experiment_info_frame = tk.Frame(self.root)
        self.experiment_info_frame.grid(row=row,pady=self.PADY,sticky=tk.N+tk.S+tk.E+tk.W)
        self.experiment_info_label= tk.Label(self.experiment_info_frame,text="Experiment
Information",font='bold')
        self.experiment_info_label.grid()
        self.begin_calibration_button = tk.Button(self.experiment_info_frame,text="Begin
Calibration",command=self.begin_calibration_button_pressed)
        self.begin_calibration_button.grid(row=2,pady=(20,0),sticky=tk.S)
        tk.Grid.rowconfigure(self.root,row,weight=1)
        tk.Grid.rowconfigure(self.experiment_info_frame,1,weight=1)
        tk.Grid.columnconfigure(self.experiment_info_frame,0,weight=1)

        return self.experiment_info_frame

    def construct_progress_frame(self,row):
        self.progress_frame = tk.Frame(self.root)
        self.progress_frame.grid(row=row,pady=self.PADY)
        self.progress_label = tk.Label(self.progress_frame,text="Experiment Progress",font='bold')
        self.progress_label.grid(columnspan=3)
        self.progress_can = tk.Canvas(self.progress_frame,bg=self.WHITE,height = self.BAR_HEIGHT,width =
self.BAR_WIDTH)
        self.progress_can.grid(columnspan=3)
        self.progress_fill = self.progress_can.create_rectangle(0,0,0,0,fill=self.BLUE,outline=self.BLUE)
        self.progress_text = self.progress_can.create_text(self.BAR_WIDTH/2,self.BAR_HEIGHT/2)
        self.stop_experiment_button = tk.Button(self.progress_frame,text="Stop
Experiment",command=self.stop_experiment_button_pressed)
        self.stop_experiment_button.grid(row=2,column=0)
```

```
        self.resume_button = tk.Button(self.progress_frame,text="Resume
Experiment",command=self.resume_button_pressed)
        self.resume_button.grid(row=2,column=1)
        self.resume_button.config(state=tk.DISABLED)
        self.save_progress_button = tk.Button(self.progress_frame,text="Save
Progress",command=self.save_progress_button_pressed)
        self.save_progress_button.grid(row=2,column=2)
        self.save_progress_button.config(state=tk.DISABLED)
        return self.progress_frame

    def stop_experiment_button_pressed(self):
        self.motion_controller.stop_executing_run()
        self.motion_controller.halt_indenter_movement()
        self.motion_controller.indenter.set_default_movement_params()
        self.motion_controller.move_indenter_home()
        self.stop_experiment_button.config(state=tk.DISABLED)
        self.resume_button.config(state=tk.NORMAL)
        self.save_progress_button.config(state=tk.NORMAL)
        self.disable(self.INPUT_FRAME)

    def resume_button_pressed(self):
        pass

    def save_progress_button_pressed(self):
        self.motion_controller.stop_executing_run()
        f = tkFileDialog.asksaveasfile(mode='w',**self.file_opt)
        if f is None: return
        cPickle.dump(self.experiment.get_results(),f)


    def construct_startup_frame(self,row):
        self.startup_frame = tk.Frame(self.root)
        self.startup_frame.grid(row=row,pady=self.PADY)
        self.startup_label = tk.Label(self.startup_frame,text="Select Experiment",font='bold')
        self.startup_label.grid(columnspan=2)
        self.load_button = tk.Button(self.startup_frame,text="Load From
File",command=self.load_button_pressed)
        self.load_button.grid(row=1,sticky=tk.E+tk.W+tk.S+tk.N,column=0)
        self.new_experiment_button = tk.Button(self.startup_frame,text="New
Experiment",command=self.new_experiment_button_pressed)
        self.new_experiment_button.grid(sticky=tk.E+tk.W+tk.S+tk.N,row=1,column=1)
        return self.startup_frame

    def construct_info_frame(self,row):
        self.info_frame = tk.Frame(self.root)
        self.info_frame.grid(row=row,pady=self.PADY)
        tk.Label(self.info_frame,text="Run Information",font="bold").grid(row=0,columnspan=3)
        tk.Label(self.info_frame,text="Load Cell Force:").grid(row=1,sticky=tk.E)
```

```
        tk.Label(self.info_frame,text="Indenter Position:").grid(row=2,sticky=tk.E)
        tk.Label(self.info_frame,text="Indenter Status:").grid(row=3,sticky=tk.E)
        tk.Label(self.info_frame,text="Selected Stimulus:").grid(row=4,sticky=tk.E)
        tk.Label(self.info_frame,text="Currently Running:").grid(row=5,sticky=tk.E)
        self.ss_text = tk.StringVar()
        self.ss_label = tk.Label(self.info_frame,textvariable=self.ss_text).grid(row=4,column=1)
        self.is_text = tk.StringVar()
        self.is_label = tk.Label(self.info_frame,textvariable=self.is_text).grid(row=3,column=1)
        self.cr_text = tk.StringVar()
        self.cr_label = tk.Label(self.info_frame,textvariable=self.cr_text).grid(row=5,column=1)
        self.lcf_can =
tk.Canvas(self.info_frame,width=self.BAR_WIDTH,height=self.BAR_HEIGHT,bg="#ffffff")
        self.lcf_can.grid(row=1,column=1)
        self.ip_can =
tk.Canvas(self.info_frame,width=self.BAR_WIDTH,height=self.BAR_HEIGHT,bg="#ffffff")
        self.ip_can.grid(row=2,column=1)
        self.ip_high=self.motion_controller.indenter.get_max_indentation()
        self.ip_low=self.motion_controller.indenter.get_home_position()
        self.ip_fill = self.ip_can.create_rectangle(0,0,100,self.BAR_HEIGHT,outline='#87CEFA',fill='#87CEFA')
        self.ip_text = self.ip_can.create_text(self.BAR_WIDTH/2,self.BAR_HEIGHT/2)
        self.lcf_target = self.lcf_can.create_rectangle(0,0,0,0)
        self.lcf_fill =
self.lcf_can.create_rectangle(0,0,100,self.BAR_HEIGHT,outline='#87CEFA',fill='#87CEFA')
        self.lcf_text = self.lcf_can.create_text(self.BAR_WIDTH/2,self.BAR_HEIGHT/2)
        return self.info_frame

    def construct_calibration_frame(self,row):
        self.calibration_frame = tk.Frame(self.root)
        self.calibration_frame.grid(row=row,pady=self.PADY,sticky=tk.N+tk.S+tk.E+tk.W)
        self.calibration_label = tk.Label(self.calibration_frame,text="Calibration",font='bold')
        self.calibration_label.grid(row=0,columnspan=2,sticky=tk.E+tk.W+tk.S+tk.N)
        self.calibration_begin_button = tk.Button(self.calibration_frame,text="Begin
Experiment",command=self.begin_button_pressed)
        self.calibration_begin_button.grid(row=3,columnspan=2,pady=(20,0))
        CAN_WIDTH = 180
        CAN_HEIGHT = 150
        CAN_BG = "#ffffff"
        self.CALIB_LABEL_WRAPLEN = 200
        self.calib_canvas =
tk.Canvas(self.calibration_frame,width=CAN_WIDTH,height=CAN_HEIGHT,bg=CAN_BG)
        CAN_PAD = 40
        self.calib_canvas.grid(row=1,column=1,padx=(CAN_PAD,0))
        self.can_frame = tk.Frame(self.calibration_frame)
        self.can_frame.grid(row=1,column=0,sticky=tk.N)
        self.home_label = tk.Label(self.can_frame,text='1. Select Home Position',justify=tk.LEFT)

        self.STAGE_WIDTH = CAN_WIDTH/10.
        self.STAGE_HEIGHT = CAN_HEIGHT*(2./3.)
```

```
        self.STAGE_X = CAN_WIDTH-self.STAGE_WIDTH-self.STAGE_WIDTH
        self.STAGE_Y = CAN_HEIGHT*(1./3.)*(1./2.)
        self.STAGE_COL = "#000000"

        self.IND_WIDTH = self.STAGE_WIDTH/2.
        self.IND_HEIGHT = self.STAGE_HEIGHT/3.
        self.IND_X = self.STAGE_X-self.IND_WIDTH
        self.IND_COL = "#dddddd"

        self.CANTIL_WIDTH = self.STAGE_WIDTH*5.
        self.CANTIL_HEIGHT = self.IND_HEIGHT/3.
        self.CANTIL_X = self.IND_X-self.CANTIL_WIDTH

        self.LC_WIDTH = self.CANTIL_WIDTH/8.
        self.LC_HEIGHT = self.LC_WIDTH/2.
        self.LC_X = self.CANTIL_X+self.CANTIL_WIDTH/10.
        self.LC_COL = "#bbbbbb"
        self.LC_FORCE_X = self.LC_X+self.LC_WIDTH/2.-self.LC_WIDTH*2.5
        self.LC_FORCE_COL = "#0000ff"
        self.LC_FORCE_WIDTH = self.LC_WIDTH*5
        self.LC_FORCE_HEIGHT = self.LC_HEIGHT

        self.can_stage =
self.calib_canvas.create_rectangle(self.STAGE_X,self.STAGE_Y,self.STAGE_WIDTH+self.STAGE_X,self.STA
GE_HEIGHT+self.STAGE_Y,fill=self.STAGE_COL,outline=self.STAGE_COL)
        self.can_indenter =
self.calib_canvas.create_rectangle(self.IND_X,0,self.IND_WIDTH+self.IND_X,self.IND_HEIGHT+0,fill=self.I
ND_COL,outline=self.STAGE_COL)
        self.can_cantil = self.calib_canvas.create_rectangle(self.CANTIL_X,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2,self.CANTIL_X+self.CANTIL_WIDTH,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT,fill=self.IND_COL,outline=self.STAGE_COL)
        self.can_lc = self.calib_canvas.create_rectangle(self.LC_X,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT,self.LC_X+self.LC_WIDTH,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT,fill=self.LC_COL,outline=self.STAGE_COL)
        self.can_lc_fb = self.calib_canvas.create_rectangle(self.LC_FORCE_X,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT,self.LC_FORCE_X+self.LC_FORCE_WIDTH,0+self.IND_HEIGH
T/2-self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_FORCE_HEIGHT,outline=self.LC_FORCE_COL)
        self.can_lc_f = self.calib_canvas.create_rectangle(self.LC_FORCE_X,0+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT,self.LC_FORCE_X+self.LC_FORCE_WIDTH,0+self.IND_HEIGH
T/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_FORCE_HEIGHT,outline=self.LC_FORCE_COL,fill=self
.LC_FORCE_COL)
        self.can_lc_text = self.calib_canvas.create_text(0,0,text='hi?')

        self.home_label_info = tk.Label(self.can_frame,text="Select a position for the indenter to return to
between indents using the connected
joystick.",justify=tk.LEFT,wraplength=self.CALIB_LABEL_WRAPLEN)
        self.home_label.grid(sticky=tk.W)
```

```
        self.home_label_info.grid(sticky=tk.N)
        self.home_position_var = tk.StringVar()
        self.home_position_var.set('Home Position: <UNSET>')
        self.home_position_label = tk.Label(self.can_frame,textvariable=self.home_position_var)
        self.home_label_info.grid()
        self.home_position_label.grid(pady=(3))

        self.radio_id = -1


        self.set_home_button = tk.Button(self.calibration_frame,text="Select Position as
Home",command=self.set_home_button_pressed)
        self.set_home_button.grid(row=2,column=1,sticky=tk.W+tk.E+tk.N,padx=(CAN_PAD,0))

        tk.Grid.columnconfigure(self.can_frame,0,weight=1)

        return self.calibration_frame

    def set_home_button_pressed(self):
        self.disable_joystick()
        self.motion_controller.set_indenter_home_at_current_pos()
        self.ip_low = self.motion_controller.indenter.get_home_position()
        self.set_home_button.config(state=tk.DISABLED)
        self.home_position_var.set('Home Position:
'+str(self.motion_controller.indenter.get_home_position()))

        self.max_label = tk.Label(self.can_frame,text="2. Select Max Indent for each Stimulus")
        self.max_label.grid(sticky=tk.W)
        self.max_label_info = tk.Label(self.can_frame,text="Select the maximum indentation the participant
is comfortable with for each stimulus.",justify=tk.LEFT,wraplength=self.CALIB_LABEL_WRAPLEN)
        self.max_label_info.grid(sticky=tk.N)
        self.max_frames_frame = tk.Frame(self.can_frame,bg="#000000",height=200)
        self.max_frames_frame.grid(sticky=tk.N+tk.E+tk.W+tk.S)
        self.max_stim_frames = []

        self.max_stim_set_variables = []
        tk.Grid.rowconfigure(self.max_frames_frame,0,weight=1)
        self.max_stim_radio_var = tk.IntVar()
        self.max_stim_radio_var.set(0)
        self.max_stim_last_val = 0
        self.max_stim_radio_buttons = []

        for i in range(self.experiment.num_stimuli):
            frame = tk.Frame(self.max_frames_frame)
            frame.grid(row=0,column=i,sticky=tk.N+tk.E+tk.W+tk.S)
            tk.Grid.columnconfigure(self.max_frames_frame,i,weight=1)
            label = tk.Label(frame,text=str(self.experiment.get_stimuli()[i].name))
            label.grid()
```

```
        radiob =
tk.Radiobutton(frame,variable=self.max_stim_radio_var,value=i,command=self.max_indent_radio_butt
on_pressed)
        radiob.grid(row=0,column=1)
        self.max_stim_radio_buttons.append(radiob)
        svar = tk.StringVar()
        self.max_stim_set_variables.append(svar)
        svar.set("MAX <UNSET>")
        label2 = tk.Label(frame,textvariable=svar)
        label2.grid(columnspan=2)
        button = tk.Button(frame,text="Set",command=lambda index = i:
self.max_indent_set_button_pressed(index))
        button.grid(sticky=tk.N+tk.E+tk.W+tk.S,columnspan=2)
        self.max_stim_frames.append(frame)
      self.max_indent_radio_button_pressed()
      self.button_binds[self.A] = lambda index = 0: self.max_indent_set_button_pressed(index)
      self.enable_joystick()

   def max_indent_set_button_pressed(self,index):
      self.experiment.stimuli[index].set_max_indent(self.motion_controller.get_indenter_position())
      self.max_stim_set_variables[index].set(self.experiment.stimuli[index].get_max_indent())
      if len(self.experiment.stimuli) > index+1:
         self.button_binds[self.A] = lambda i = index+1: self.max_indent_set_button_pressed(i)
         self.max_indent_radio_button_pressed(index + 1)
      else:
         self.button_binds[self.A] = self.begin_button_pressed
       #  self.motion_controller.move_indenter_home()

   #TODO home and swap command..? or soemthing like that threaded in motion controller.
   def max_indent_radio_button_pressed(self,index=None):
      val = self.max_stim_radio_var.get()
      if not index is None:
         val = index
         self.max_stim_radio_var.set(val)
      [flag,self.radio_id] = self.home_and_select(self.experiment.stimuli[val],self.radio_id)
      if not flag: self.max_stim_radio_var.set(self.max_stim_last_val)
      else:
         self.max_stim_last_val = val
         self.enable_frame(self.max_stim_frames[val])
         for i in range(self.experiment.num_stimuli):
            if i != val: self.disable_frame(self.max_stim_frames[i])
            self.max_stim_radio_buttons[i].config(state=tk.NORMAL)

   def home_and_select(self,stimulus,id):
      if id == self.motion_controller.get_run_id():
         return [False,id]
      else:
         p = Procedure()
```

```
            p.add(('INDENTER_DISABLE_JOYSTICK',))
            p.add(('INDENTER_STOP_JOGGING',))
            p.add(('SET_MOTION_PARAMS',4))
            p.add(('MOVE_INDENTER_HOME',))
            p.add(('SELECT_STIMULUS',stimulus))
            p.add(('INDENTER_BEGIN_JOGGING',))
            p.add(('INDENTER_ENABLE_JOYSTICK',))
            self.motion_controller.run(p)
            new_id = self.motion_controller.get_run_id()
            return [True,new_id]


    def construct_input_frame(self,row):
        self.input_frame = tk.Frame(self.root)
        self.input_frame.grid(row=row,columnspan=3,pady=self.PADY,sticky=tk.N+tk.S+tk.E+tk.W)
        tk.Grid.rowconfigure(self.input_frame, 0, weight=1)
        tk.Grid.rowconfigure(self.input_frame, 1, weight=1)
        for i in range(3):
            tk.Grid.columnconfigure(self.input_frame,i,weight=1)
        self.same_button = tk.Button(self.input_frame,text='Same',command=self.same_button_pressed)
        self.same_button.grid(row=1,column=2,sticky=tk.N+tk.S+tk.E+tk.W)
        self.different_button =
tk.Button(self.input_frame,text='Different',command=self.different_button_pressed)
        self.different_button.grid(row=1,column=1,sticky=tk.N+tk.S+tk.E+tk.W)
        self.repeat_button = tk.Button(self.input_frame,text='Repeat
Run',command=self.repeat_run_button_pressed)
        self.repeat_button.grid(row=1,column=0,sticky=tk.N+tk.S+tk.E+tk.W)
        self.input_label = tk.Label(self.input_frame,text='User Input',font='bold')
        self.input_label.grid(row=0,columnspan=3)
        return self.input_frame

    def construct_stimulus_info_frame(self,row):
        self.stimulus_frame = tk.Frame(self.root)
        #self.stimulus_frame.grid(row=1,pady=5,sticky=tk.E+tk.W+tk.S+tk.N)
        tk.Grid.rowconfigure(self.stimulus_frame, 0, weight=1)
        tk.Grid.rowconfigure(self.stimulus_frame, 1, weight=1)
        tk.Grid.columnconfigure(self.stimulus_frame, 0, weight=1)
        self.stimulus_frame.grid(row=row,pady=self.PADY,sticky=tk.N+tk.S+tk.E+tk.W)

        self.stimulus_label = tk.Label(self.stimulus_frame,text="Stimulus Information",font='bold')
        #Something is wrong with this...
        self.stimulus_label.grid(row=0)
        return self.stimulus_frame

    def begin_calibration_button_pressed(self):
        self.hide(self.EXPERIMENT_INFO_FRAME)
        self.hide(self.STIMULUS_INFO_FRAME)
        self.hide(self.STARTUP_FRAME)
```

```
    self.unhide(self.CALIBRATION_FRAME)
    self.motion_controller.move_indenter_home()
    self.motion_controller.stimulus_selector.select(self.experiment.stimuli[0])
    self.enable_joystick()
    self.motion_controller.indenter.begin_jogging()
 #   self.motion_controller.move_indenter_up(100)
    self.motion_controller.calibrate_load_cell_zero()
    self.button_binds[self.A] = self.set_home_button_pressed


#TODO STOP SIMULATION?
def begin_button_pressed(self):
#    print('ya')
    if not self.running_trials:
        self.running_trials = True
        self.unhide(self.INFO_FRAME)
        self.unhide(self.INPUT_FRAME)
        self.unhide(self.PROGRESS_FRAME)
        self.hide(self.STIMULUS_INFO_FRAME)
        self.hide(self.CALIBRATION_FRAME)
        self.hide(self.STARTUP_FRAME)
        self.hide(self.EXPERIMENT_INFO_FRAME)
        self.disable_joystick()
        self.button_binds[self.X] = self.same_button_pressed
        self.button_binds[self.B] = self.different_button_pressed
        self.button_binds[self.Y] = self.repeat_run_button_pressed
        self.button_binds[self.A] = self.stop_experiment_button_pressed
        self.enable_info()

def load_button_pressed(self):
    #TODO File selection stuff.
    f = tkFileDialog.askopenfile(mode='r', **self.file_opt)
    if f is None: return

    self.unhide(self.STIMULUS_INFO_FRAME)
    self.unhide(self.EXPERIMENT_INFO_FRAME)
    self.experiment.load_from_file('stimuli.xml')
    self.table = Table(self.stimulus_frame,nrows=self.experiment.num_stimuli,ncols=3)
    self.table.frame.grid(row=1,columnspan=2,sticky=tk.E+tk.W+tk.S+tk.N)

    for i in range(self.experiment.num_stimuli):
        self.table.label(row=i+1,label='Position ' +str(i+1))
        self.table.set(i+1,1,self.experiment.stimuli[i].name)
        self.table.set(i+1,3,self.experiment.stimuli[i].shape)
        self.table.set(i+1,2,str(self.experiment.stimuli[i].kPa))

    self.table.label(col=2,label="kPa")
    self.table.label(col=1,label="Name")
```

```
        self.table.label(col=3,label="Shape")


        self.experiment_info_table =
Table(self.experiment_info_frame,nrows=len(self.experiment.pairs),ncols=7)
        self.experiment_info_table.frame.grid(row=1,sticky=tk.E+tk.W+tk.S+tk.N)

        self.experiment_info_table.label(col=1,label="Fmin")
        self.experiment_info_table.label(col=2,label="Fmax")
        self.experiment_info_table.label(col=3,label="n")
        self.experiment_info_table.label(col=4,label="Vmin")
        self.experiment_info_table.label(col=5,label="Vmax")
        self.experiment_info_table.label(col=6,label="n")
        self.experiment_info_table.label(col=7,label="n trials")

        for i in range(len(self.experiment.pairs)):
            p = self.experiment.pairs[i]
     #      print(i+1)
            self.experiment_info_table.label(row=i+1,label=str(p.stim1.name)+ " vs. " + str(p.stim2.name))
            self.experiment_info_table.set(i+1,1,min(p.forces))
            self.experiment_info_table.set(i+1,2,max(p.forces))
            self.experiment_info_table.set(i+1,3,len(p.forces))
            self.experiment_info_table.set(i+1,4,min(p.velocities))
            self.experiment_info_table.set(i+1,5,max(p.velocities))
            self.experiment_info_table.set(i+1,6,len(p.velocities))
            self.experiment_info_table.set(i+1,7,p.num_trials)

    def new_experiment_button_pressed(self):
        pass

    def enable_info(self):
        for child in self.info_frame.winfo_children():
            child.configure(state=tk.NORMAL)
            self.info_disabled = False

    def input_received(self):
        self.waiting_for_input = False
        self.enable_info()
        self.disable_input()

    def different_button_pressed(self):
        self.input_received()
        self.current_run.different()

    def same_button_pressed(self):
        self.input_received()
        self.current_run.same()
```

```
    def repeat_run_button_pressed(self):
        self.repeat_pressed = True
        self.input_received()

    def disable_info(self):
        for child in self.info_frame.winfo_children():
            child.configure(state=tk.DISABLED)
        self.info_disabled = True

    def is_waiting_for_input(self):
        return self.waiting_for_input

    def disable_input(self):
        for child in self.input_frame.winfo_children():
            child.configure(state=tk.DISABLED)
    #    self.different_button.config(state=tk.DISABLED)
    #    self.repeat_button.config(state=tk.DISABLED)
     #   self.same_button.config(state=tk.DISABLED)

    def enable_input(self):
        for child in self.input_frame.winfo_children():
            child.configure(state=tk.NORMAL)

    #TODO SHOULD RETURN THE SAME RUN IF REPEATED
    def get_run(self):
        if self.repeat_pressed:
            self.repeat_pressed = False
            return self.current_run
        return self.experiment.next_run()

    #TODO LAST RUN WILL KEEP DISPLAYING I THINK...
    def run_trial(self):
        if self.running_trials:

            if not self.motion_controller.is_running_procedure() and self.experiment_running:
                self.enable_input()
                self.disable_info()
                self.waiting_for_input = True
                self.experiment_running = False

            elif(self.experiment.has_next() and (not self.motion_controller.is_running_procedure()) and not
self.is_waiting_for_input()):
                self.disable_input()
                self.experiment_running = True
                self.current_run = self.get_run()
                place,total = self.experiment.get_progress()
                place_pos = int((float(place)/float(total))*float(self.BAR_WIDTH))
                self.progress_can.coords(self.progress_fill,0,0,place_pos,self.BAR_HEIGHT)
```

```
            self.progress_can.itemconfig(self.progress_text,text="Run "+str(place)+" of " + str(total))
            if self.current_run.rate_type == 'F':
               self.cr_text.set(str(self.current_run.stim1.name) + " vs. " + str(self.current_run.stim2.name) +
" at " + str(self.current_run.rate) + " N/s to " + str(self.current_run.peak) + " N.")
            else:
               self.cr_text.set(str(self.current_run.stim1.name) + " vs. " + str(self.current_run.stim2.name) +
" at " + str(self.current_run.rate) + " mm/s to " + str(self.current_run.peak) + " mm.")
           #  print(run.pos1,run.pos2,run.force,run.velocity)
            target_pos = int((float(self.current_run.peak-self.lcf_low)/float(self.lcf_high-
self.lcf_low))*self.BAR_WIDTH)
            self.lcf_can.coords(self.lcf_target,target_pos,0,target_pos,self.BAR_HEIGHT)
            self.lcf_can.itemconfig(self.lcf_target,fill='#FF7F50',outline='#FF7F50')


            p = Procedure()
            p.add_calibrate_loadcell()
            cr = self.current_run.get_list()
            if self.current_run.has_2_indents:
               if self.current_run.rate_type == 'poly2':

p.add_poly_control(self.current_run.rate,self.current_run.peak,self.current_run.stim1,self.current_run.t
ime_r1,self.current_run.pos_r1,self.current_run.force_r1)
               else:

p.add_force_control(self.current_run.rate,self.current_run.peak,self.current_run.stim1,self.current_run
.time_r1,self.current_run.pos_r1,self.current_run.force_r1)
               if self.current_run.rate_type2 == 'poly2':

p.add_poly_control(self.current_run.rate2,self.current_run.peak2,self.current_run.stim2,self.current_ru
n.time_r1,self.current_run.pos_r1,self.current_run.force_r1)
               else:

p.add_force_control(self.current_run.rate2,self.current_run.peak2,self.current_run.stim2,self.current_r
un.time_r2,self.current_run.pos_r2,self.current_run.force_r2)

            else:
               if cr[-1] == 'D':

p.add_displacement_control(cr[2],cr[3],self.current_run.stim1,self.current_run.time_r1,self.current_run
.pos_r1,self.current_run.force_r1)

p.add_displacement_control(cr[2],cr[3],self.current_run.stim2,self.current_run.time_r2,self.current_run
.pos_r2,self.current_run.force_r2)
               else:

p.add_force_control(cr[2],cr[3],self.current_run.stim1,self.current_run.time_r1,self.current_run.pos_r1,
self.current_run.force_r1)
```

```
p.add_force_control(cr[2],cr[3],self.current_run.stim2,self.current_run.time_r2,self.current_run.pos_r2,
self.current_run.force_r2)
            self.motion_controller.run(p)

        self.root.after(self.BETWEEN_TRIAL_TIME,self.run_trial)

    def data_refresh(self):
     #   if not self.info_disabled:
            #CHECK FOR CALIBRATING!!!
        calibrating = False
        if not self.motion_controller.load_cell.is_calibrating_zero():
            lcf = self.motion_controller.get_load_cell_force()
            col = '#87CEFA'
            lc_perc = float(lcf-self.lcf_low)/float(self.lcf_high-self.lcf_low)
            if self.is_using_joystick and lc_perc >0.01:
                self.xboxCont.set_vibration(0,0,1 if lc_perc*2 > 1 else lc_perc*2)
            else: self.xboxCont.set_vibration(0,0,0)
            lcf_w = int(float(lcf-self.lcf_low)/float(self.lcf_high-self.lcf_low)*self.BAR_WIDTH)
            if lcf_w > self.BAR_WIDTH:
                lcf_w == self.BAR_WIDTH
                col = '#FF7F50'
            self.lcf_can.coords(self.lcf_fill,0,0,lcf_w,self.BAR_HEIGHT)
            self.lcf_can.itemconfig(self.lcf_fill,outline=col,fill=col)
            if not self.info_disabled: self.lcf_can.itemconfig(self.lcf_text,text="{0:.1f}".format(lcf)+"N")
            self.lcf_can.itemconfig(self.lcf_target,state=tk.NORMAL)
        else:
            calibrating = True
            self.lcf_can.itemconfig(self.lcf_target,state=tk.HIDDEN)
            lc_perc = self.motion_controller.load_cell.get_calibrate_zero_progress()
            pos = int(lc_perc*self.BAR_WIDTH)
            col = "#6AFB92"
            self.lcf_can.coords(self.lcf_fill,0,0,pos,self.BAR_HEIGHT)
            self.lcf_can.itemconfig(self.lcf_fill,outline=col,fill=col)
            self.lcf_can.itemconfig(self.lcf_text,text="Calibrating...")

        ip = self.motion_controller.get_indenter_position()
        self.ip_high = self.motion_controller.indenter.get_max_indentation()
        ip_w = int(float(ip-self.ip_low)/float(self.ip_high-self.ip_low)*self.BAR_WIDTH)
        self.ip_can.coords(self.ip_fill,0,0,ip_w,self.BAR_HEIGHT)
        self.ip_can.itemconfig(self.ip_text,text="{0:.1f}".format(ip)+" units")
        [cn, clist] = self.motion_controller.get_current_command()
        #print('CLIST',clist)
        if (len(clist)==0):
            self.is_text.set('huh?')
        else:
             self.is_text.set(clist[cn][0])
```

```
        ss = self.motion_controller.get_selected_stimulus()
        if ss is None: self.ss_text.set('No stimulus selected.')
        else: self.ss_text.set(ss.name + ", " + str(ss.kPa) + "kPa " + ss.shape)

        i_min = self.STAGE_Y
        i_max = self.STAGE_Y+self.STAGE_HEIGHT-self.IND_HEIGHT
        i_perc = (float(ip)-(-50.))/100.
        i_val = int(float(i_max-i_min)*i_perc+ i_min)

self.calib_canvas.coords(self.can_indenter,self.IND_X,i_val,self.IND_WIDTH+self.IND_X,self.IND_HEIGHT
+i_val)
        self.calib_canvas.coords(self.can_cantil,self.CANTIL_X,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2,self.CANTIL_X+self.CANTIL_WIDTH,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT)
        self.calib_canvas.coords(self.can_lc,self.LC_X,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT,self.LC_X+self.LC_WIDTH,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT)
        self.calib_canvas.coords(self.can_lc_fb,self.LC_X-self.LC_WIDTH*2.,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT*2,self.LC_X-
self.LC_WIDTH*2.+self.LC_FORCE_WIDTH,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT*2+self.LC_FORCE_HEIGHT)
        self.calib_canvas.coords(self.can_lc_text,self.LC_X-
self.LC_WIDTH*2.+self.LC_FORCE_WIDTH/2,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT*3.5+self.LC_FORCE_HEIGHT)
        if lc_perc > 1: lc_perc = 1
        pos = lc_perc*self.LC_FORCE_WIDTH

        self.calib_canvas.coords(self.can_lc_f,self.LC_X-self.LC_WIDTH*2.,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT*2,self.LC_X-
self.LC_WIDTH*2.+pos,i_val+self.IND_HEIGHT/2-
self.CANTIL_HEIGHT/2+self.CANTIL_HEIGHT+self.LC_HEIGHT*2+self.LC_FORCE_HEIGHT)
        #TODO HACKY
        if col == '#87CEFA': col = '#0000ff'
        elif col == "#6AFB92": col = '#00ff00'
        elif col ==  '#FF7F50': col = '#ff0000'
        self.calib_canvas.itemconfig(self.can_lc_fb,outline=col)
        self.calib_canvas.itemconfig(self.can_lc_f,outline=col,fill=col)
        if not calibrating: self.calib_canvas.itemconfig(self.can_lc_text,text="{0:.1f}".format(lcf)+"N",fill =
"#000000")
        else: self.calib_canvas.itemconfig(self.can_lc_text,text="Calibrating...",fill = "#000000")


        self.root.after(self.DATA_REFRESH_TIME,self.data_refresh)

    #keypresses
    def upKey(self,event):
        val = float(2)
        self.motion_controller.move_indenter_up(val)
```

```
    def downKey(self,event):
        val = float(2)
        self.motion_controller.move_indenter_down(val)

    def rightKey(self,event):
        self.motion_controller.xps.GroupMoveAbort(self.motion_controller.socket,
CONSTANTS.INDENTER_GROUP_NAME)
        self.motion_controller.stop_procedure()

pw = Psychophysics_Window()
```

Load_Cell.py

```
__author__ = 'Steven'
import threading
import collections
import scipy.interpolate
import cPickle
import scipy.stats
import numpy as np
import time
import random
from CONSTANTS import *

#TODO MAKE SURE THAT THE OFFSET ALLOWS FOR THE DETECTABLE RANGE!! SAFETY ISSUE!!
class Load_Cell:

    #not sure I like calibrating_mode
    def __init__(self, xps,socket, calibrating_mode=False):
        self.socket = socket
        self.calibrating_zero = False
        self.xps = xps

        infile = open('fdcurvestim1.dat')
        self.dummyx,self.dummyy = cPickle.load(infile)
        self.dummyx = np.array(self.dummyx)-min(self.dummyx)
        print(self.dummyx)
        print(self.dummyy)
        infile.close()

        self._orig_zero = None

        self.m = 0  #None?
        self.b = 0  #None?

        self.lock = threading.Lock()

        self.calibration_table = {} #should eventually be None
        #TODO Full calibration, maybe
        if not calibrating_mode: self._load_calibration()
        #self.calibrate_zero()

    def is_calibrating_zero(self):
        with self.lock: return self.calibrating_zero

    def get_force(self,num=1):
        if self.is_calibrating_zero():
            return None
        if num == 1:
```

```
        ret = self._D2F(self.get_output_voltage())
        return ret
    else:
        ret = []
        for i in range(num):
            ret.append(self._D2F(self.get_output_voltage()))
        return np.mean(ret)

#digital out to Force
def _D2F(self,D):
    with self.lock: gwt = self.m*D+self.b
    return gwt*0.00980665002864 #to Newtons

def _F2D(self,F):
    gwt = F/0.00980665002864
    with self.lock: D = (gwt-self.b)/self.m
    return D

def calibrate_zero(self):
    with self.lock: self.calibrating_zero = True
    vals = []
    npts = CONSTANTS.LOADCELL_CALIB_ZERO_NPOINTS
    with self.lock: self.calibrate_zero_progress_pts = 0
    for i in range(npts):
        vals.append(self.get_output_voltage())
        with self.lock: self.calibrate_zero_progress_pts += 1
    zero_v = np.mean(vals)
    old_0 = self._orig_zero
    delta = old_0 - zero_v
    x = []
    y = []
    for key in sorted(self.calibration_table.keys()):
        x.append(key-delta)
        y.append(self.calibration_table[key])
 #TODO WTF
    slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x,y)
   # print('IMPORTANT',r_value)
    with self.lock:
        self.m = slope
        self.b = intercept
    #print(self.m,self.b)
    self.last_zero_calibration = time.time()
    with self.lock:
        self.calibrating_zero = False
        self.calibrate_zero_progress_pts = 0

def get_calibrate_zero_progress(self):
    return float(self.calibrate_zero_progress_pts)/float(CONSTANTS.LOADCELL_CALIB_ZERO_NPOINTS)
```

```
    #TODO this shouldn't always break
    def _load_calibration(self):
        read_file = open(CONSTANTS.LOADCELL_CALIB_FILE, 'rb')
       # self.calibration_table = cPickle.load(read_file)
        try:
            self.calibration_table = cPickle.load(read_file)
        except:
            raise ValueError('No Calibration Data in <' + CONSTANTS.LOADCELL_CALIB_FILE + '>!')
        x = []
        y = []
        #TODO WTF HAVE TO NEGATIVE
        for key in sorted(self.calibration_table.keys()):
            x.append(key)
            y.append(self.calibration_table[key])
        #TODO FLIP
        '''y = np.flipud(y)
        index = 0
        for key in sorted(self.calibration_table.keys()):
            self.calibration_table[key] = y[index]
            index += 1'''
        slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x,y)
        self.m = slope
        self.b = intercept
        print('load_calib')
        print(self.m,self.b)
        self._orig_zero = self._F2D(0)
        read_file.close()

    def get_dummy_force(self,x_val):
        time.sleep(.003)
        if x_val < 0: x_val = 0
        print(x_val)
        if x_val > 2.802: x_val = 2.802
        noise = .1-random.random()*.2
        x2f = scipy.interpolate.interp1d(self.dummyx,self.dummyy)
        return x2f(x_val)+noise

    #will not make concurrent requests
    def get_output_voltage(self):
        with self.lock: [error_ret, str_ret] =
self.xps.GPIOAnalogGet(self.socket,[GPIO.PINS[GPIO.PIN_LOADCELL_IN],])
        return float(str_ret)
```

Indenter.py

```
__author__ = 'Steven'
import XPS_Q8_drivers
from Tkinter import *
import scipy.interpolate
import numpy as np
import cPickle
from collections import deque
import matplotlib.pyplot as plt
from CONSTANTS import *
import threading
import Queue
import time


class Indenter:

    #WHEN A MOTION IS HAPPENING, YOU CAN ONLY ABORT

    def __init__(self,xps,socket):

        self.POSITIONER_NAME = CONSTANTS.INDENTER_POSITIONER_NAME
        self.GROUP_NAME = CONSTANTS.INDENTER_GROUP_NAME
        self.home_position = CONSTANTS.INDENTER_DEFAULT_HOME_POSITION

        self.xps = xps
        self.socket = socket

        self.xps.GroupInitializeWithEncoderCalibration(self.socket, self.GROUP_NAME)
        self.xps.GroupInitializeWithEncoderCalibration(self.socket, self.POSITIONER_NAME+'1')

        self.xps.GroupHomeSearchAndRelativeMove(self.socket, self.POSITIONER_NAME+'1', [0,])

        self.default_velocity = CONSTANTS.INDENTER_DEFAULT_VELOCITY
        self.default_acceleration = CONSTANTS.INDENTER_DEFAULT_ACCELERATION

        self.velocity = CONSTANTS.INDENTER_DEFAULT_VELOCITY
        self.acceleration = CONSTANTS.INDENTER_DEFAULT_ACCELERATION
        self.jerk = CONSTANTS.INDENTER_DEFAULT_JERK

        self.set_motion_params(self.get_default_velocity(),self.get_default_acceleration())

        #Locked actions--getting current position
        self.lock = threading.Lock()

        self.max_indentation = CONSTANTS.INDENTER_DEFAULT_MAX_INDENTATION
        #TODO what happens when this event is triggered?
```

```
        self.max_indentation_event_id = None
        self.set_max_indentation(self.max_indentation)
        self.min_indentation = CONSTANTS.INDENTER_DEFAULT_MIN_INDENTATION
        [a,b] = self.xps.PositionerUserTravelLimitsSet(self.socket,
CONSTANTS.INDENTER_POSITIONER_NAME, self.min_indentation, self.max_indentation)

    #TODO
    def restore_defaults(self):
        pass

    #TODO
    def get_set_velocity(self):
        return 'TODO'
    #TODO
    def get_set_acceleration(self):
        return 'TODO'

    def get_home_position(self):
        return self.home_position

    #TODO
    #WAS THE ABORT SUCCESSFUL? return true or false!
    def abort(self):
        [a,b] = self.xps.GroupMoveAbort(self.socket, self.GROUP_NAME)
        if a == 0: return True
        else: return False

    def get_current_position(self,socket=None):
        if socket is None: socket = self.socket
        with self.lock: [a,b] = self.xps.GroupPositionCurrentGet(socket,self.GROUP_NAME, 1)
        return float(b)

    def move_to_home(self,blocking=True):
        self.move_to_target(self.get_home_position(),blocking)

    def move_down(self,amt,blocking=True):
        self.move_to_target(self.get_current_position()+amt,blocking)

    def move_up(self,amt,blocking=True):
        self.move_to_target(self.get_current_position()-amt,blocking)

    def set_home_position(self,pos):
        self.home_position = pos

    def move_to_target(self,target,blocking=True):
        if target == 'MAX':
            target = self.get_max_indentation()
        if blocking:
```

```
            self._absolute_move(target)
        else:
            self._initialize_movement(target)

    def get_status(self):
        [a,b] =
self.xps.GroupStatusStringGet(self.socket,self.xps.GroupStatusGet(self.socket,self.GROUP_NAME)[1])
        return b

    def set_motion_params(self,velocity=None,acceleration=None):
        if velocity is None: velocity = self.get_default_velocity()
        if acceleration is None: acceleration = self.get_default_acceleration()
        [a,b]=self.xps.PositionerSGammaParametersSet(self.socket, self.POSITIONER_NAME, float(velocity),
float(acceleration), float(self.jerk), float(self.jerk))

    def set_jogging_params(self,velocity=None,acceleration= None):
    #    print('is one of these not finishing?')
        if velocity is None: velocity = self.get_default_velocity()
        if acceleration is None: acceleration = self.get_default_acceleration()
        self.xps.GroupJogParametersSet(self.socket,self.GROUP_NAME,(velocity,),(acceleration,))
     #   print('guess not')

    def set_tracking_params(self,voltage,scale,order,max_velocity,max_acceleration):
        [a,b] =
self.xps.PositionerAnalogTrackingVelocityParametersSet(self.socket,self.POSITIONER_NAME,GPIO.PINS[
GPIO.PIN_ANALOG_IN],voltage,getdouble(scale),0,order,max_velocity,str(max_acceleration))
     #   print('setting!!',a,b)

    def end_tracking(self):
        self.xps.GroupAnalogTrackingModeDisable(self.socket,self.GROUP_NAME)

    #TODO check this if it's position!
    def begin_tracking(self,type='velocity'):
        if type == 'velocity': type = 'Velocity'
        else: type = 'Position'
        [a,b] =
self.xps.GroupStatusStringGet(self.socket,self.xps.GroupStatusGet(self.socket,self.GROUP_NAME)[1])
     #   print(a,b)
        [a,b] = self.xps.GroupMoveAbort(self.socket,self.POSITIONER_NAME+'1')
        [a,b] = self.xps.GroupJogModeDisable(self.socket,self.POSITIONER_NAME+'1')
     #   print(a,b)
        [a,b] = self.xps.GroupJogModeDisable(self.socket,self.POSITIONER_NAME)
      #  print(a,b)
        [a,b]=self.xps.GroupJogModeDisable(self.socket,self.GROUP_NAME)
      #  print(a,b)
        [a,b] = self.xps.GroupJogModeDisable(self.socket,self.POSITIONER_NAME+'1')
      #  print(a,b)
        [a,b] = self.xps.GroupJogModeDisable(self.socket,self.POSITIONER_NAME)
```

```
 #  print(a,b)
   [a,b]=self.xps.GroupMotionEnable(self.socket,self.GROUP_NAME)
 #   print(a,b)
   self.xps.GroupAnalogTrackingModeDisable(self.socket,self.GROUP_NAME)
   [a,b] = self.xps.GroupAnalogTrackingModeEnable(self.socket,self.GROUP_NAME,type)
  # print('trackingggg',a,b)

def end_jogging(self):
 #   print('jogging over!')
   [a,b] = self.xps.GroupJogModeDisable(self.socket,self.GROUP_NAME)
 #   print(a,b)

def begin_jogging(self):
   self.xps.GroupJogModeEnable(self.socket,self.GROUP_NAME)

def end_motion(self):
   self.xps.GroupMotionDisable(self.socket,self.GROUP_NAME)

def begin_motion(self):
 #   print('motion beginning?')
   [a,b] = self.xps.GroupMotionEnable(self.socket,self.GROUP_NAME)
 #   print(a,b)

def get_default_velocity(self):
   return self.default_velocity

def get_default_acceleration(self):
   return self.default_acceleration

def _absolute_move(self,target):
   if target < self.min_indentation: target = self.min_indentation
   elif target > self.max_indentation: target = self.max_indentation
   [a,b] = self.xps.GroupMoveAbsolute(self.socket, self.POSITIONER_NAME, [target,])
 #  print('DID THIS FAIL OR SOMETHING?')
 #  print(a,b,'wut')

def _initialize_movement(self,target):
   thread = threading.Thread(target=self._absolute_move, args=(target,))
   thread.start()

def get_max_indentation(self):
   with self.lock: return self.max_indentation

def set_max_indentation(self,max_indentation):
   with self.lock: self.max_indentation = max_indentation
   self.xps.EventExtendedRemove(self.socket,self.max_indentation_event_id)
```

```
      '''[a,b] = self.xps.EventExtendedConfigurationTriggerSet(self.socket,['Always',], ['0',], ['0',], ['0',],
['0',])
      [a,b] =
self.xps.EventExtendedConfigurationActionSet(self.socket,[str(GPIO.PINS[GPIO.PIN_VELOCITY_OUT])+'.D
ACSet.SetpointVelocity',],['GROUP1.POSITIONER',],['1',],['0',],['0',])
      [a,b] = self.xps.EventExtendedStart(self.socket)
      self.max_indentation_event_id = b'''
      return 0

   '''def _safety_loop(self):
      #TODO This is just okay
      while True:
         if self.get_current_pos() > self.get_max_indentation():
            self._clear_commands()
            self._command(self.END_JOGGING)
            self._command(self.ABORT)
            self._command(self.MOVE_TO_TARGET,self.get_max_indentation()+.001)
            self._set_should_stop_waiting(True)
            #TODO THIS MIGHT FREEZE THE SAFETY LOOP...
            while abs(self.get_current_pos() - self.get_max_indentation())>.0001: pass
         #TODO IT'S GONNA CLAMP AT HIGH FORCE
         if self.motion_controller.load_cell.get_force() > CONSTANTS.LOADCELL_MAX_FORCE:
            self._clear_commands()
            self._command(self.END_JOGGING)
            self._command(self.ABORT)
            self._set_should_stop_waiting(True)

   def _main_loop(self):
      while True:
         command_tuple = self._get_command()
         command = command_tuple[0]
         args = command_tuple[1]
         arg1 = args[0]
         arg2 = args[1]
         if command == self.NO_COMMAND:
            pass
         #TODO SOME PROBLEM IN ABORTING
         elif command == self.ABORT:
            self._set_status(self.ABORTING)
            self._abort()
         elif command == self.MOVE_TO_TARGET:
            self._initialize_movement(arg1)
            print('going to:',arg1)
            self._set_status(self.MOVING_TO_TARGET)
            self._await_movement()
         elif command == self.MOVE_UP:
            self._initialize_movement(self.get_current_pos()-arg1)
            print('going to ' + str(self.get_current_pos()-arg1))
```

```
            self._set_status(self.MOVING_TO_TARGET)
            self._await_movement()
        elif command == self.MOVE_TO_HOME:
            self._initialize_movement(self.get_home_position())
            self._set_status(self.MOVING_TO_HOME)
            self._await_movement()
        elif command == self.CHANGE_MOVEMENT_PARAMETERS:
            self._set_movement_parameters(arg1,arg2)
        #ACTUALLY WAITS FOR THE END OF MOVEMENT
        #TODO will this work?
        elif command == self.WAIT_FOR_READY:
        #    time.sleep(.01)
        #    while self._is_moving():
            while self._is_moving() or self._is_awaiting_movement():
                if self._should_stop_waiting():
                    self._set_should_stop_waiting(False)
                    break
        elif command == self.TRACE_FORCE:
            print('here?')
            self._set_status(self.INDENTING_TO_FORCE)
            self._set_mode(self.TRACKING_MODE)
            t2f = scipy.interpolate.interp1d(arg1,arg2,'linear')
            tstart = time.time()
            FIN_T = []
            FIN_F = []
            count = 0
            #self._set_tracking_params(0,200,2,20,120) GOOD FOR LOW FORCE RATES
            self._set_tracking_params(0,20,2,5,5)
            while True:
                if self._should_stop_waiting():
                    self._set_should_stop_waiting(False)

                    break
                tcurr = time.time()-tstart

                if tcurr > max(arg1): break
                else:
                    fset = t2f(tcurr)
                    print(fset)
                    voltage = self.motion_controller.load_cell._F2D(fset)
                    voltage = voltage*0.999602726549304-0.004143488684578
                    #print('set to',voltage)
                    with self.tracking_lock:
self.motion_controller.gpio.set(self.tracking_sock,self.motion_controller.gpio.PIN_ANALOG_OUT,voltag
e)
                    count = count + 1
                    #FIN_T.append(tcurr)
                    #with self.tracking_lock: FIN_F.append(self.motion_controller.load_cell.get_force())
```

```
        file_out = open('teststuff.dat','w')
        cPickle.dump((FIN_T,FIN_F),file_out)
        file_out.close()
        print(count)
        self._end_tracking()
    elif command == self.END_TRACKING:
        self._end_tracking()
    elif command == self.PRINT_FORCE:
        print(self.motion_controller.load_cell.get_force())
    elif command == self.WAIT_FOR_TIME:
        self._set_status(self.WAITING)
        start_time = time.time()
        while True:
            curr_t = time.time()
            if curr_t-start_time >= arg1: break
            if self._should_stop_waiting():
                self._set_should_stop_waiting(False)
                break
    elif command == self.WAIT_FOR_INPUT:
        self._set_status(self.WAITING_FOR_INPUT)
        with self.input_lock:
            self.input = False
        while True:
            with self.input_lock:
                if self.input: break
            if self._should_stop_waiting():
                self._set_should_stop_waiting(False)
                break
    #TODO PROPERLY SET/UNSET SHOULD_STOP_WAITING
    elif command == self.WAIT_FOR_FORCE:
        self._set_status(self.INDENTING_TO_FORCE)
        print('FORCE',arg1)
        self.motion_controller.set_max_force_limit(self.force_limit_sock,arg1)
        x = []
        y = []
        while True:
            x.append(self.get_current_pos())
            y.append(self.motion_controller.load_cell.get_force())
            if self._should_stop_waiting():
                self._set_should_stop_waiting(False)
                break
            elif self.motion_controller.load_cell.get_force() >= arg1:
                print('this way')
                self.motion_controller.remove_force_limit(self.force_limit_sock)
                self._set_should_stop_waiting(False)
                break
            elif not self._is_moving():
                print('that way')
```

```
                self._set_should_stop_waiting(False)
                break
            print(x)
            print(y)
          #  outf = open('fdcurve.dat','w')
          #  cPickle.dump((x,y),outf)
          #  outf.close()
        elif command == self.WAIT_FOR_FORCE_LOW:
            self._set_status(self.INDENTING_TO_FORCE)
            self.motion_controller.set_min_force_limit(self.force_limit_sock,arg1)
            while True:
                if self._should_stop_waiting():
                    self._set_should_stop_waiting(False)
                    break
                elif self.motion_controller.load_cell.get_force() <= arg1:
                    print('this way')
                    self.motion_controller.remove_force_limit(self.force_limit_sock)
                    self._set_should_stop_waiting(False)
                    break
                elif not self._is_moving():
                    print('that way')
                    self._set_should_stop_waiting(False)
                    break
        elif command == self.BEGIN_JOGGING:
            print('here2.1')
            self._set_status(self.JOGGING)
            print('here2.2')
            self._set_mode(self.JOG_MODE)
        elif command == self.CHANGE_JOGGING_PARAMETERS:
            self._set_jogging_params(arg1,arg2)
        elif command == self.END_JOGGING:
            self._set_mode(self.MOVEMENT_MODE)
            self._set_status(self.READY)


        if self._is_awaiting_movement():
            pass
        elif not self._is_moving() and self._get_status() != self.JOGGING:
            self._set_status(self.READY)

    def send_input(self):
        with self.input_lock: self.input = True

    #TODO THIS METHOD
    def _set_movement_parameters(self,velocity,acceleration):
        with self.params_lock:
            if velocity is None: velocity = self.velocity
            if acceleration is None: acceleration = self.acceleration
```

```
        self.xps.PositionerSGammaParametersSet(self.params_sock, self.POSITIONER_NAME, velocity,
acceleration, self.jerk, self.jerk)
        #thread = threading.Thread(target=self.targ,args=(velocity,acceleration))
        #thread.start()

    def targ(self,velocity,acceleration):
        time.sleep(1)
        self.xps.PositionerSGammaParametersSet(self.params_sock, self.POSITIONER_NAME, velocity,
acceleration, self.jerk, self.jerk)


    #TODO THIS SHOULD ACTUALLY CHECK THE INDENTER'S MODE
    def _get_mode(self):
        pass
    #    with self.lock: return self.mode

    def set_default_movement_params(self):

self._command(self.CHANGE_MOVEMENT_PARAMETERS,self.get_default_velocity(),self.get_default_ac
celeration())


    def _set_tracking_params(self,voltage,scale,order,max_velocity,max_acceleration):
        with self.tracking_lock:
            #print(force)
            print(self._get_ind_status())
            # print(self.motion_controller.gpio.pins[self.motion_controller.gpio.PIN_LOADCELL_IN])
            #print(self.motion_controller.load_cell._F2D(force))
            #print(self.motion_controller.load_cell.get_output_voltage())
            self._end_tracking()
            #print(self._get_ind_status())

[a,b]=self.xps.PositionerAnalogTrackingVelocityParametersSet(self.tracking_sock,self.POSITIONER_NAM
E,self.motion_controller.gpio.pins[self.motion_controller.gpio.PIN_ANALOG_IN],voltage,getdouble(scale
),0.01,order,max_velocity,max_acceleration)
        self._begin_tracking()
        #print(a,b)

    def is_jogging(self):
        if self._get_mode() == self.JOG_MODE: return True
        else: return False

    #TODO MAKE SURE WORKING
    def _set_mode(self,mode):
        with self.set_mode_lock:
            #  self.mode = mode
            if mode == self.JOG_MODE:
                #TODO FIX THIS
```

```
            [a,b] = self.xps.GroupJogModeEnable(self.set_mode_sock,self.GROUP_NAME)
            print("WHO IS ASKING TO JOG")
        elif mode == self.MOVEMENT_MODE: [a,b] =
self.xps.GroupMotionEnable(self.set_mode_sock,self.GROUP_NAME)
        elif mode == self.TRACKING_MODE:
            self._abort()
            print('stat1',self._get_ind_status())
            [a,b] = self.xps.GroupJogModeDisable(self.set_mode_sock,self.POSITIONER_NAME+'1')
            print(a,b)
            [a,b] = self.xps.GroupJogModeDisable(self.set_mode_sock,self.POSITIONER_NAME)
            print(a,b)
            [a,b]=self.xps.GroupJogModeDisable(self.set_mode_sock,self.GROUP_NAME)
            print(a,b)
            [a,b] = self.xps.GroupJogModeDisable(self.set_mode_sock,self.POSITIONER_NAME+'1')
            print(a,b)
            [a,b] = self.xps.GroupJogModeDisable(self.set_mode_sock,self.POSITIONER_NAME)
            print(a,b)
            print('stat2',self._get_ind_status())
            [a,b]=self.xps.GroupMotionEnable(self.set_mode_sock,self.GROUP_NAME)
            print(a,b)
            print('stat',self._get_ind_status())
            self.xps.GroupAnalogTrackingModeDisable(self.set_mode_sock,self.GROUP_NAME)
            [a,b] =
self.xps.GroupAnalogTrackingModeEnable(self.set_mode_sock,self.GROUP_NAME,'Velocity')
            #[a,b] =
self.xps.GroupAnalogTrackingModeEnable(self.set_mode_sock,self.POSITIONER_NAME,'Velocity')
            print('set?',(a,b))
        #print(a,b)


    def begin_jogging(self):
        self._command(self.ABORT)
        self._command(self.BEGIN_JOGGING)

    def end_jogging(self):
        self._command(self.END_JOGGING)

self._command(self.CHANGE_MOVEMENT_PARAMETERS,self.get_default_velocity(),self.get_default_ac
celeration())

    def _set_jogging_params(self,velocity,acceleration):
        with self.jog_parameters_lock:
            [a,b] =
self.xps.GroupJogParametersSet(self.jog_parameters_sock,self.GROUP_NAME,(velocity,),(acceleration,))
        #print('jogo',a,b)

    def set_jogging_params(self,velocity,acceleration=None):
        if acceleration == None:
```

```
            with self.params_lock: acceleration = self.acceleration
        self._command(self.CHANGE_JOGGING_PARAMETERS,velocity,acceleration)

    #call this method
    def _await_movement(self):
        self._set_awaiting_movement(True)

    def _set_awaiting_movement(self,value):
        with self.awaiting_movement_lock: self.awaiting_movement = value
        if value:
            with self.awaiting_movement_lock: self.awaiting_movement_start = time.time()

    def _is_awaiting_movement(self):
       # print('awaiting!!')
        with self.awaiting_movement_lock:
            val = self.awaiting_movement
            last_time = self.awaiting_movement_start
            timeout = self.AWAITING_MOVEMENT_TIMEOUT
        if not val: return False
        elif time.time()-last_time >= timeout:
            self._set_awaiting_movement(False)
            return False
        elif self._is_moving():
            self._set_awaiting_movement(False)
            return False
        else: return True

    def finished_commands(self):
        return self.command_queue.empty() and self._get_status() == self.READY

    def _should_stop_waiting(self):
        with self.lock:
        #    print(self.should_stop_waiting)
            return self.should_stop_waiting

    def _set_should_stop_waiting(self,stop_waiting):
        with self.lock: self.should_stop_waiting = stop_waiting

    def get_current_pos(self):
        with self.current_pos_lock: [a,b] =
self.xps.GroupPositionCurrentGet(self.current_pos_sock,self.GROUP_NAME, 1)
        return float(b)

    def get_status_string(self):
        return self.status_strings[self._get_status()]


    #OLD VERSION: while self._is_moving, pass
```

```
def abort_all_movement(self):
   self._clear_commands()
   self._command(self.END_JOGGING)
   self._command(self.ABORT)
   self._command(self.END_TRACKING)
   self._set_should_stop_waiting(True)
   while self._is_moving():
      self._command(self.END_JOGGING)
      self._command(self.ABORT)
      self._command(self.END_TRACKING)
      self._set_should_stop_waiting(True)

def _command(self,command,arg1=None,arg2=None):
   print('commanding',command)
   self.command_queue.put((command,(arg1,arg2)))

def get_max_indentation(self):
   with self.lock: return self.max_indentation

def _get_min_indentation(self):
   with self.lock: return self.min_indentation

def set_max_indentation(self,max_indentation):
   with self.lock: self.max_indentation = max_indentation

def _set_min_indentation(self,min_indentation):
   with self.lock: self.min_indentation = min_indentation

def set_home_position(self,home_position):
   with self.lock: self.home_position = home_position

def get_home_position(self):
   with self.lock: return self.home_position

#TODO MOVEMENT PARAMETERS
def initialize_indent_to_force(self,force,max_indent,velocity=None,acceleration=None):
   if velocity is None: velocity = self.get_default_velocity()
   if acceleration is None: acceleration = self.get_default_acceleration()
   self._command(self.ABORT)
   self._command(self.MOVE_TO_HOME)
   self._command(self.WAIT_FOR_READY)
   self._command(self.CHANGE_MOVEMENT_PARAMETERS,velocity,acceleration)
   self._command(self.MOVE_TO_TARGET,max_indent)
   self._command(self.WAIT_FOR_FORCE,force)
   self._command(self.ABORT)
   self._command(self.WAIT_FOR_TIME,1)
```

```
self._command(self.CHANGE_MOVEMENT_PARAMETERS,self.get_default_velocity(),self.get_default_ac
celeration())
      self._command(self.MOVE_TO_HOME)
      self._command(self.WAIT_FOR_READY)



   def _clear_commands(self):
      while not self.command_queue.empty():
         self.command_queue.get()




   def _enable_analog_tracking(self):
      with self.tracking_lock:
         [a,b] = self.xps.GroupAnalogTrackingModeEnable(self.tracking_sock,self.GROUP_NAME,'Velocity')

   #TODO SOMETHING IS WRONG WITH THIS
   def _get_command(self):
      if self.command_queue.empty(): return (self.NO_COMMAND,(None,None))
      else:
         command = self.command_queue.get()
         return command
      #return self.command_queue.get()

   def _get_status(self):
      with self.lock: return self._status

   def get_status(self):
      return self._get_status()

   def _abort(self):
       #  print('abort called')
         with self.abort_lock: [a,b] = self.xps.GroupMoveAbort(self.abort_sock, self.GROUP_NAME)
       #  print(a,b)
   #    self.stop_i2f()

   def move_to(self,target):
      self._command(self.ABORT)
      self._command(self.MOVE_TO_TARGET,target)

   #TODO MAYBE IF THE NEXT COMMAND IS ABORT, YOU SHOULD STOP WAITING??
   #TODO MAYBE ABORT SHOULDN'T BE A COMMAND LIKE THE OTHERS??
   def move_down(self,amt):
      self._command(self.ABORT)
```

```
        self._command(self.MOVE_TO_TARGET,self.get_current_pos()+amt)

    def move_up(self,amt):
        self._command(self.ABORT)
        self._command(self.MOVE_TO_TARGET,self.get_current_pos()-amt)

    def change_target(self,target):
        self._clear_commands()
        self._command(self.ABORT)
        self._command(self.MOVE_TO_TARGET,target)

    def _get_ind_status(self):
        with self.templock:
            [a,b] =
self.xps.GroupStatusStringGet(self.tempsock,self.xps.GroupStatusGet(self.is_moving_sock,self.GROUP_
NAME)[1])
            return b

    def _is_moving(self):
        with self.is_moving_lock: [a,b] =
self.xps.GroupStatusStringGet(self.is_moving_sock,self.xps.GroupStatusGet(self.is_moving_sock,self.GR
OUP_NAME)[1])
        retVal = (b == 'Moving state')
        return retVal

    def _set_status(self,status):
        with self.lock: self._status = status

     #this does not change during the run
    def set_velocity(self,velocity):
        self.velocity = velocity
        self.set_movement_parameters()

    #this does not change during the run
    def set_acceleration(self,acceleration):
        self.acceleration = acceleration
        self.set_movement_parameters()

    #this does not change during the run
    def set(self,velocity,acceleration):
        self.velocity = velocity
        self.acceleration = acceleration
        self.set_movement_parameters()'''
```

Stimulus_Selector.py

```
__author__ = 'Steven'
from CONSTANTS import *
import threading
import time
import random
from Stimulus import *
import serial
import copy

#TODO IT'S TURNED OFF
class Stimulus_Selector:

    def __init__(self):
    #    self.motion_controller = motion_controller
#        self.motor = self.motion_controller.motor

        self.ANGLES = {1:116,2:47}
        #TODO THESE SHOULD BE COMMENTED BACK ON
        self.ser = serial.Serial('COM8')
        self.ser.baudrate = 9600
        self.ser.close()
        self.ser.open()
        self.ser.flush()
        self.motormap = {1:37,2:99,3:154}

    #    self.NSTIMULI = CONSTANTS.NUM_STIMULI

  #      self.motor_map = {}
  #      for i in range(self.NSTIMULI):
  #          self.motor_map[i+1] = (CONSTANTS.MOTOR_MAX_ROTATION/(self.NSTIMULI-1))*i

        self.lock = threading.Lock()
        self.selected_stimulus = Stimulus()

  #      self.sock = self.motion_controller.get_new_socket()

    #TODO AT SOME POINT IT WILL BE SELECTED WITHOUT MAX INDENT
    def select(self,stimulus):
        #TODO: change this!!
        self.ser.flush()
        self.random_movements()
     #   print('did i finish though..')
        self.ser.write(str(self.motormap[stimulus.position])+'\n')
        with self.lock: self.selected_stimulus = stimulus
        return
        with self.lock:
```

```
        if not (stimulus.position in self.motor_map): return
        else:
            #TODO BELOW LINE MAYBE?
           # self.motor.command_angle(self.motor_map[stimulus.position])
            #print(chr(int('1')))
           # self.ser.write(str(stimulus.position)[0])
            self.random_movements()
            a = self.ser.write(str(self.ANGLES[stimulus.position])+'\n')
            print(a)
            print(str(self.ANGLES[stimulus.position])+'\n')
            if stimulus.get_max_indent() is None: print(stimulus.name + ' does not have a max indent set.')
            else: self.motion_controller.indenter.set_max_indentation(stimulus.get_max_indent())
            time.sleep(0.2)
            self._selected_stimuli = stimulus


def random_movements(self):
    for i in range(3):
        angle = random.randint(0,178)
        self.ser.write(str(angle) + '\n')
        time.sleep(0.333)


#for read_only
def get_selected_stimuli_info(self):
    with self.lock:
        if self._selected_stimuli is None: return None
        return None

#TODO is it thread safe?
def get_selected_stimulus(self):
    ret = None
    with self.lock:
        ret = copy.deepcopy(self.selected_stimulus)

    return ret
```

Stimulus.py

```
__author__ = 'Steven'
import collections
import Image_Parser
from CONSTANTS import *

class Stimulus:

    def __init__(self,name='',kPa='',shape='',position=None,notes = ''):
        self.name = name
        self.kPa = kPa
        self.shape = shape
        self.notes = notes
        self.position = position
        self.contact_point = CONSTANTS.INDENTER_DEFAULT_HOME_POSITION
        self.max_indent = CONSTANTS.INDENTER_DEFAULT_MAX_INDENTATION
        self.calib_displacements = []
        self.calib_contact_areas = []


    def set_max_indent(self,pos):
        self.max_indent = pos

    def get_max_indent(self):
        return self.max_indent

    def set_calib_displacements(self,calib_displacements):
        self.calib_displacements = calib_displacements

    def load_contact_areas(self,filename):
        contact_areas = Image_Parser.Image_Parser.parse_image(filename)
        self.calib_contact_areas = contact_areas

    def get_max_indentation(self):
        return self.max_indent
```

Procedure.py

```
__author__ = 'Steven'
import numpy as np


'''self.funcdict = {
        'MOVE_INDENTER_HOME':self.move_indenter_home,
        'WAIT_FOR_FORCE':self.wait_for_force,
        'WAIT_FOR_TIME':self.wait_for_time,
        'ABORT_INDENTER':self.abort_indenter,
        'RESET':self.reset,
        'CLEAR_EVENTS':self.clear_events,
        'SWITCH_STIMULI':self.switch_stimuli,
        'CALIBRATE_LOAD_CELL_ZERO':self.calibrate_load_cell_zero
    }
'''

#('INDENT_TO_FORCE',force,wait_time,lim_type,velocity,acceleration)

class Procedure:
    def __init__(self,clist = None):
        if clist is None: self.command_list = []
        else: self.command_list = clist

    def load_procedure(self,filename):
        pass

    def add_indent_to_force(self,force,velocity,acceleration):
        l = [('SET_MOTION_PARAMS',velocity,acceleration),
            ('MOVE_INDENTER_TO_TARGET','MAX',False),
            ('ABORT_AT_FORCE',1),
            ]
        self.command_list.extend(l)

    def add(self,command):
        self.command_list.append(command)

    def add_calibrate_loadcell(self):
        l = [('CALIBRATE_LOAD_CELL_ZERO',),]
        self.command_list.extend(l)


    #TODO TRIANGLE WAVES!
    def add_displacement_control(self,rate,peak,stimulus,t_r=[],pos=[],force=[]):
        l = [('SET_INDENTER_LIMIT',stimulus.max_indent),
            ('SET_MOTION_PARAMS',15),
            ('MOVE_INDENTER_HOME',),
```

```
      ('SELECT_STIMULUS',stimulus),
      ('WAIT_FOR_TIME',0.5),
      ('CALIBRATE_LOAD_CELL_ZERO',),
      ('SET_MOTION_PARAMS',15),
      ('MOVE_INDENTER_TO_TARGET',stimulus.contact_point),
      ('SET_MOTION_PARAMS',rate),
      ('BEGIN_GATHERING',),
      ('MOVE_INDENTER_DOWN',peak),
     # ('WAIT_FOR_TIME',1),
      ('MOVE_INDENTER_UP',peak),
      ('STOP_GATHERING',t_r,pos,force),
      ('SET_MOTION_PARAMS',15),
      ('MOVE_INDENTER_HOME',)]
   self.command_list.extend(l)

def add_force_control(self,rate,peak,stimulus,t_r = [],pos=[],force=[]):

   rate = float(rate)
   peak = float(peak)
   t = [0, peak/rate, peak/rate, (peak/rate)*2]
   f = [0, peak, peak, 0]
   l = [('SET_INDENTER_LIMIT',stimulus.max_indent),
      ('MOVE_INDENTER_HOME',),
      ('SELECT_STIMULUS',stimulus),
      ('WAIT_FOR_TIME',0.5),
      ('CALIBRATE_LOAD_CELL_ZERO',),
      ('SET_MOTION_PARAMS',15),
      ('MOVE_INDENTER_TO_TARGET',stimulus.contact_point),
      ('BEGIN_GATHERING',),
      ('TRACE_FORCE',t,f),
      ('STOP_GATHERING',t_r,pos,force),
      ('SET_MOTION_PARAMS',15),
      ('MOVE_INDENTER_TO_TARGET',stimulus.contact_point),
      ('MOVE_INDENTER_HOME',)]
   self.command_list.extend(l)

def add_poly_control(self,time,peak,stimulus,t_r = [],pos=[],force=[],order=2):
   time = float(time)
   peak = float(peak)

   t1 = np.linspace(0,time,100)
   t1 = t1
   t = []
   t.extend(t1)
   t.extend([time,time*2])
   f = []
   f.extend(2.*(t1**2.))
   f.extend([peak,0])
```

```
    # print(t)
    # print(f)
    # return

    l = [('SET_INDENTER_LIMIT',stimulus.max_indent),
        ('MOVE_INDENTER_HOME',),
        ('SELECT_STIMULUS',stimulus),
        ('WAIT_FOR_TIME',0.5),
        ('CALIBRATE_LOAD_CELL_ZERO',),
        ('SET_MOTION_PARAMS',15),
        ('MOVE_INDENTER_TO_TARGET',stimulus.contact_point),
        ('BEGIN_GATHERING',),
        ('TRACE_FORCE',t,f),
        ('STOP_GATHERING',t_r,pos,force),
        ('SET_MOTION_PARAMS',15),
        ('MOVE_INDENTER_TO_TARGET',stimulus.contact_point),
        ('MOVE_INDENTER_HOME',)]
    self.command_list.extend(l)

def add_dealio(self):
    l = [('SET_MOTION_PARAMS',1,80),
        ('MOVE_INDENTER_DOWN',1),
        ('MOVE_INDENTER_UP',1)
        ]
    self.command_list.extend(l)
```

Procedure_Window.py

```
__author__ = 'Steven'
__author__ = 'Steven'
import Tkinter as tk
from Indenter import *
from Load_Cell import *
from XPS_GPIO import *
import time
import _tkinter
import PIL.Image
from PIL import ImageTk
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure
import cPickle
from Table import *
import XboxController
import threading
from Psychophysics_Experiment import *
from Motion_Controller import *

class Procedure_Window:

    def __init__(self,root,mc):
        self.REFRESH_TIME = 100 #ms
        self.root = root
        self.mc = mc

        self.win = tk.Toplevel(self.root)
        self.win.title("Procedure Window")
        self.text = tk.Text(self.win)
        self.text.grid(sticky=tk.S+tk.N+tk.E+tk.W)


        self.root.after(self.REFRESH_TIME,self.refresh)

    def refresh(self):
        self.text.delete('1.0', END)
        [c_num, command_list] = self.mc.get_current_command()
        i = 0
        for command in command_list:
            cstr = command[0]
            self.text.insert(tk.INSERT,cstr+'\n')
            if i == c_num:
                self.text.tag_add("here", str(i+1)+'.0', str(i+1)+'.'+str(len(cstr)+1))
                self.text.tag_config("here", background="yellow", foreground="blue")
```

```
    i = i+ 1
self.root.after(self.REFRESH_TIME,self.refresh)
```

Psychophysics_Experiment.py

```
__author__ = 'Steven'
from Motion_Controller import *
from Stimulus import *
import cPickle
import collections
from CONSTANTS import *
import numpy


#TODO to save, just pickle the experiment! doy
#TODO RANDOM ORDER OF PAIRED STIMULI
class Experiment:

    def __init__(self):

        #N_STIMULI SHOULD NOT BE IN CONSTANTS!!
        self.pairs = []
        self.num_stimuli = 2
        self.velocities = []
        self.forces = []
        self.stimuli = []

        self.runs = None
        self.place = 0

        self.n_trials = 0

    def get_progress(self):
        return (self.place,len(self.runs))

    def create_runs(self):
        self.n_trials = 10
        self.runs = []
        self.stimuli = []
        stim1 = (Stimulus('Stimulus 1',18,'cylinder',1))
        stim2 = (Stimulus('Stimulus 2',22,'cylinder',2))
        stim1.contact_point = 32.66  #hard contacts__quinn
        stim2.contact_point = 31.6

        stim1.contact_point = 32.66  #hard contacts__loi
        stim2.contact_point = 31.6

        # stim1.contact_point = 32.85  #hard contacts__alan
        # stim2.contact_point = 30.9

        stim1.contact_point = 31.65  #hard contacts__alan
```

```
stim2.contact_point = 29.2

stim1.contact_point = 31.65
stim2.contact_point = 29.6

stim1.contact_point = 32.65
stim2.contact_point = 30.6

stim1.contact_point = 31.65
stim2.contact_point = 29.6

stim1.contact_point = 27.8
stim2.contact_point = 26.7

# stim1.contact_point = 32.66
# stim2.contact_point = 31.0


#hard contact_1 should be about 2mm below
# stim1.contact_point = 32.347
# stim2.contact_point = 35.0745

# stim1.contact_point = 37.1
# stim2.contact_point = 35.3
#
# stim1.contact_point = 36.7
# stim2.contact_point = 34.9

# stim1.contact_point = 36.3
# stim2.contact_point = 34.0
#
stim1.contact_point = 33.1005
stim2.contact_point = 31.028

stim1.contact_point = 32.1405
stim2.contact_point = 29.9


# stim1.contact_point = 52.8
# stim2.contact_point = 52.8
self.stimuli.append(stim1)
self.stimuli.append(stim2)
for i in range(self.n_trials):
  self.runs.append(Run(stim1,stim2,0.5,'F',0.5,'F'))
  self.runs.append(Run(stim1,stim2,0.8,'D',0.8,'D'))
  self.runs.append(Run(stim2,stim1,0.5,'F',0.5,'F'))
  self.runs.append(Run(stim2,stim1,0.8,'D',0.8,'D'))
```

```
        self.place = 0

        random.shuffle(self.runs)
        self.runs.insert(0,Run(stim1,stim2,.01,'F',.01,'F'))

    #TODO THIS METHOD
    def has_next(self):
        if self.place >= len(self.runs): return False
        else: return True

    #TODO needs checks
    def next_run(self):
        ret_run = self.runs[self.place]
        self.place += 1
        return ret_run

    def get_stimuli(self):
        return self.stimuli

    def add_stimulus(self,stimulus):
        self.stimuli.append(stimulus)
        stimulus.set_max_indent(CONSTANTS.INDENTER_DEFAULT_MAX_INDENTATION)
        self.num_stimuli += 1

    #TODO Each stimuli needs a max indent..?
    #eventually this will load from a file
    def load_from_file(self,filename):
        self.create_runs()
        return
       # self.velocities = [0.2]
        self.velocities = [2]
        self.forces = [3] #Newtons
        self.n_trials = 10
        self.add_stimulus(Stimulus('Stimulus 1',18,'cylinder',1))
        self.add_stimulus(Stimulus('Stimulus 2',22,'cylinder',2))
        self.create_runs()

    def get_results(self):
        ret = {}
        for pair in self.pairs:
            ret[pair.stim1.name+','+pair.stim2.name] = pair.get_results()
        return ret


class Run:
    RESULT_SAME = CONSTANTS.RESULT_SAME
    RESULT_DIFFERENT = CONSTANTS.RESULT_DIFFERENT
    NUM = 1
```

```
    OUTPUT = []

    def __init__(self,stim1,stim2,rate,rate_type,peak,peak_type,has_2_indents =
False,rate2=None,rate_type2=None,peak2=None,peak_type2=None):
        self.stim1 = stim1
        self.stim2 = stim2

        self.rate=rate
        self.rate_type = rate_type
        self.peak = peak
        self.peak_type = peak_type

        self.has_2_indents = has_2_indents
        if has_2_indents:
            #second rate and peak
            self.rate2 = rate2
            self.rate_type2 = rate_type2
            self.peak2 = peak2
            self.peak_type2 = peak_type2

        self.time_r1 = []
        self.pos_r1 = []
        self.force_r1 = []
        self.time_r2 = []
        self.pos_r2 = []
        self.force_r2 = []

        self._result = None

    def get_list(self):
        return [self.stim1,self.stim2,self.rate,self.peak,self.rate_type]

    def clear_recordings(self):
        self.time_r1 = []
        self.pos_r1 = []
        self.force_r1 = []
        self.time_r2 = []
        self.pos_r2 = []
        self.force_r2 = []

    def same(self):
        self._result = Run.RESULT_SAME
        str_ = 'RUN' + str(Run.NUM)+ ': ' + self.rate_type + ', ' + str(self.stim1.name) + '>' +
str(self.stim2.name)
        if self.has_2_indents:
            str_ = str_ + '                    ' +self.stim1.name +   ': '  +  self.rate_type + '      RATE: ' + str(self.rate)
        Run.NUM = Run.NUM + 1
```

```
Run.OUTPUT.append((str_,self.stim1.name,self.time_r1,self.pos_r1,self.force_r1,self.time_r2,self.pos_r
2,self.force_r2))
      write_file = open('psychophysics.out', 'wb')
      cPickle.dump(Run.OUTPUT ,write_file)
      write_file.close()
      print(str_)

   def get_result(self):
      return self._result

   def different(self):
      self._result = Run.RESULT_DIFFERENT
      str_ = 'RUN' + str(Run.NUM)+ ': ' + self.rate_type + ', ' + str(self.stim2.name) + '>' +
str(self.stim1.name)
      if self.has_2_indents:
         str_ = str_ + '                    ' +self.stim1.name +   ': '  +  self.rate_type + '      RATE: ' + str(self.rate)
      Run.NUM = Run.NUM + 1

Run.OUTPUT.append((str_,self.stim1.name,self.time_r1,self.pos_r1,self.force_r1,self.time_r2,self.pos_r
2,self.force_r2))
      write_file = open('psychophysics.out', 'wb')
      cPickle.dump(Run.OUTPUT ,write_file)
      write_file.close()
      print(str_)

#TODO MIGHT NOT NEED PAIR CLASS
class Pair:

   def __init__(self,stim1,stim2,forces,velocities,num_trials):
      self.stim1 = stim1
      self.stim2 = stim2
      self.num_trials = num_trials
      self.forces = sorted(forces)
      self.velocities = sorted(velocities)
      self.f_v = []      #tuple of force, velocity for each run

      for force in forces:
         for velocity in velocities:
            self.f_v.append((force,velocity))

      self.runs = []
      for f_v in self.f_v:
         force = f_v[0]
         velocity = f_v[1]
         for i in range(self.num_trials):
            self.runs.append(Run(stim1,stim2,force,velocity))
```

```
#Looks like [velocity][force][%different]
def get_results(self):
    temp_struct = {}
    for run in self.runs:
        if not run.velocity in temp_struct.keys(): temp_struct[run.velocity] = {}
        if not run.force in temp_struct[run.velocity].keys(): temp_struct[run.velocity][run.force] = []
        if not run.get_result() is None: temp_struct[run.velocity][run.force].append(run.get_result())
    ret_struct = collections.OrderedDict()
    for velocity in sorted(temp_struct.keys()):
        ret_struct[velocity] = collections.OrderedDict()
        for force in sorted(temp_struct[velocity].keys()):
            num_dif = 0
            num_same = 0
            if len(temp_struct[velocity][force]) == 0:
                perc_dif = None
            else:
                for result in temp_struct[velocity][force]:
                    if result == Run.RESULT_DIFFERENT: num_dif += 1
                    elif result == Run.RESULT_SAME: num_same += 1
                perc_dif = float(num_dif)/float(len(temp_struct[velocity][force]))

            ret_struct[velocity][force] = (num_dif,num_same,perc_dif)
    return ret_struct


def get_runs(self):
    return self.runs
```

Results_Window.py

```
__author__ = 'Steven'
import Tkinter as tk
from Indenter import *
from Load_Cell import *
from XPS_GPIO import *
import time
import _tkinter
import PIL.Image
from PIL import ImageTk
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure
import cPickle
from Table import *
import XboxController
import threading
from Psychophysics_Experiment import *
from Motion_Controller import *

class Results_Window:

    def __init__(self,root,experiment):
        self.REFRESH_TIME = 1000 #ms
        self.root = root
        self.experiment = experiment

        self.win = tk.Toplevel(self.root)
        self.win.title("University of Virginia Psychophysics Results")
        self.table_frame = tk.Frame(self.win)
        self.table_frame.grid(sticky=tk.S+tk.N+tk.E+tk.W)
      #  self.table = Table(self.table_frame,nrows=3,ncols=20)
        tk.Grid.rowconfigure(self.win,0,weight=1)
        tk.Grid.columnconfigure(self.win,0,weight=1)
        tk.Grid.rowconfigure(self.win,1,weight=1)
        tk.Grid.columnconfigure(self.table_frame,0,weight=1)
       # tk.Grid.rowconfigure(self.table.frame,0,weight=1)
     #  self.table.frame.grid(sticky=tk.S+tk.N+tk.E+tk.W)

        self.figure = Figure()
        self.temp_plot = self.figure.add_subplot(111)

        self.tk_figure = FigureCanvasTkAgg(self.figure, master = self.win)
        self.tk_figure.get_tk_widget().grid()
        self.tk_figure.show()
        self.lines = {}
```

```
    self.initialized = False
    self.tables = {}

    self.root.after(self.REFRESH_TIME,self.refresh)

#MAKE NEW TABLE FOR NEW FORCES??
#just update a data structure..?
def refresh(self):
  #  self.temp_plot.plot([2,3,4],[5,6,7])
    if not self.initialized and len(self.experiment.pairs) == 0: pass
    elif not self.initialized and len(self.experiment.pairs) != 0:
        print('dude')
        #tk.Grid.columnconfigure(self.table.frame,0,weight=1)
        self.initialized = True
        for i in range(len(self.experiment.pairs)):
            pair = self.experiment.pairs[i]
            tk.Grid.rowconfigure(self.table_frame,i,weight=1)
            table = Table(self.table_frame,nrows=len(pair.velocities),ncols=len(pair.forces)+1)
            table.label(row=1,label=pair.stim1.name + " vs. " + pair.stim2.name)
            table.label(col=1,label='Velocity')
            for i2 in range(len(pair.velocities)):
                table.set(i2+1,1,sorted(pair.velocities)[i2])
            for i2 in range(len(pair.forces)):
                table.label(col=i2+2,label=str(sorted(pair.forces)[i2]) + 'N')
            self.tables[pair] = table
            table.frame.grid(sticky=tk.N+tk.E+tk.S+tk.W,pady=(0,10))
            tk.Grid.rowconfigure(self.table_frame,i,weight=1)

    max_f = 0
    for i in range(len(self.experiment.pairs)):
        pair = self.experiment.pairs[i]
        if not pair in self.lines.keys():
            self.lines[pair] = {}
        results = pair.get_results()
        i1 = 0
        for velocity in results.keys():
            if not velocity in self.lines[pair].keys(): self.lines[pair][velocity] = self.temp_plot.plot([],[])
            #self.table.label(row=i+1,label=pair.stim1.name + " vs. " + pair.stim2.name)
            #self.table.set(i+1,1,velocity)
            table = self.tables[pair]
            i2 = 0
            f = []
            perc = []
            for force in results[velocity].keys():
                if not results[velocity][force][2] is None:
                    f.append(force)
                    perc.append(results[velocity][force][2]*100)
                    table.set(i1+1,i2+2,"{0:.1f}".format(results[velocity][force][2]*100)+"%")
```

```
            i2 += 1
        self.lines[pair][velocity][0].set_data(f,perc)
        i1 += 1
       # print('do')
       # f = [1,2,3,4,5]
       # perc = [50,60,70,80,90]
       # self.lines[pair][velocity][0].set_data(f,perc)
        if len(f) != 0:
            if max(f) > max_f: max_f = max(f)
self.temp_plot.set_xlim([0-.00000001,max_f])
self.temp_plot.set_ylim([-5,105])
self.tk_figure.draw()
self.root.after(self.REFRESH_TIME,self.refresh)
```

Calibration_Window.py

```
__author__ = 'Steven'
import Tkinter as tk
from Indenter import *
from Load_Cell import *
from XPS_GPIO import *
import time
import _tkinter
from PIL import Image
from PIL import ImageTk
import matplotlib
import XboxController
matplotlib.use('TkAgg')
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure
from Procedure import Procedure
from Psychophysics_Experiment import Experiment
import cPickle
#import camera_3_23 as cam
from Motion_Controller import *
from Procedure_Window import Procedure_Window

class Calibration_Window:


    #TODO MOVEABORT ETC ON WINDOW CLOSE
    def __init__(self):

        self.REFRESH_WAIT = 500 #time between screen refreshes, in ms
        self.DATA_COLLECT_RATE = 10 #time between data collections, in ms
        self.CALIB_NPOINTS = 100 #number of points collected to average for each calibration

        self.calibration_data = {}

        self.pos_r = []
        self.force_r = []
        self.time_r = []

        self.motion_controller = Motion_Controller()
        self.indenter = self.motion_controller.indenter
        self.loadcell = self.motion_controller.load_cell
        self.experiment = Experiment()
        self.experiment.load_from_file('stimulus.txt')
        #self.indenter.set_max_indentation(50.8)

        #self.xboxCont = XboxController.XboxController(controllerCallBack = None,joystickNo = 0,deadzone
= 0.1,scale = 1,invertYAxis = False)
```

```
        #self.xboxCont.start()
        #self.xboxCont.setup_vibration()

        self.root = tk.Tk()
        self.root.title("Calibration Window")
        self.canvas = tk.Canvas(self.root, width=1080, height=900)
        self.canvas.pack()
        self.procedure_window = Procedure_Window(self.root,self.motion_controller)

        self.figure = Figure()
        self.temp_plot = self.figure.add_subplot(111)
        self.plot_bounds = None
        self.loadcell_axs = None

        self.tk_figure = FigureCanvasTkAgg(self.figure, master = self.root)
        self.tk_figure.get_tk_widget().place(x=0, y=0)
        self.tk_figure.show()

        self.dist_entry = tk.Entry(self.root)
        self.dist_entry.delete(0, END)
        self.dist_entry.insert(0, "1")
        self.dist_entry.place(x=800,y=50)
        self.dist_label = tk.Label(self.root, text="units")
        self.dist_label.place(x=900,y=50)
        self.indenter_position_text = tk.StringVar()
        self.indenter_status_text = tk.StringVar()
        self.target_pos_text = tk.StringVar()
        self.INDENTS_MIN = None
        self.INDENTS_MAX = None
        self.INDENTS_N = 5


        self.indents_min_text = tk.StringVar()
        self.indents_max_text = tk.StringVar()
        self.indents_min_text.set('Select Min (none)')
        self.indents_max_text.set('Select Max (none)')
        self.indents_select_max_button =
tk.Button(self.root,textvariable=self.indents_max_text,command=self.indents_select_max_button_pres
sed)
        self.indents_select_max_button.place(x=550,y=800)
        self.indents_select_min_button =
tk.Button(self.root,textvariable=self.indents_min_text,command=self.indents_select_min_button_press
ed)
        self.indents_select_min_button.place(x=550,y=830)
        self.begin_indents_button = tk.Button(self.root,text='Begin
Indents',command=self.begin_indents_button_pressed)
        self.begin_indents_button.place(x=550,y=860)
```

```
        self.TARGET_POS = None
        self.target_pos_text.set('Select Target (none)')
        self.set_target_pos_button =
tk.Button(self.root,textvariable=self.target_pos_text,command=self.set_target_pos_button_pressed)
        self.set_target_pos_button.place(x=550,y=600)
        self.indent_to_target_button = tk.Button(self.root,text='Indent to
Target',command=self.indent_to_target_button_pressed)
        self.indent_to_target_button.place(x=550,y=680)
        self.indenter_position_label = tk.Label(self.root,textvariable=self.indenter_position_text)
        self.indenter_position_label.place(x = 20,y=500)
        self.indenter_status_label = tk.Label(self.root,textvariable=self.indenter_status_text)
        self.indenter_status_label.place(x = 20,y=525)

        self.force_entry = tk.Entry(self.root)
        self.force_entry.place(x=800,y=250)
        self.achieve_force_entry = tk.Entry(self.root)
        self.achieve_force_entry.place(x=800,y=500)
        self.dist_label = tk.Label(self.root,text="gwt")
        self.dist_label.place(x=900,y=250)

        self.calibrate_button = tk.Button(self.root,text="Calibrate
force",command=self.calibrate_button_pressed)
        self.calibrate_button.place(x=800,y=300)
        self.save_button = tk.Button(self.root,text="Save Calibration Data",command=self.save_calibration)
        self.save_button.place(x=800,y=350)
        self.calibrate_zero_button = tk.Button(self.root,text="Calibrate Load Cell
Zero",command=self.calibrate_zero_button_pressed)
        self.calibrate_zero_button.place(x=800,y=400)
        self.indenter_home_button = tk.Button(self.root,text="Indenter
Home",command=self.indenter_home_button_pressed)
        self.indenter_home_button.place(x=800,y=450)
        self.achieve_force_button = tk.Button(self.root,text="Achieve
Force",command=self.achieve_force_button_pressed)
        self.achieve_force_button.place(x=800,y=550)
        self.indenter_max_indent_text = tk.StringVar()
        self.set_max_indent_button =
tk.Button(self.root,textvariable=self.indenter_max_indent_text,command=self.set_max_indent_button
_pressed)
        self.set_max_indent_button.place(x = 250,y = 500)
        self.indenter_home_text = tk.StringVar()
        self.set_home_button =
tk.Button(self.root,textvariable=self.indenter_home_text,command=self.set_home_button_pressed)
        self.set_home_button.place(x = 450,y = 500)
        self.abort_all_button = tk.Button(self.root,text="ABORT ALL
MOVEMENT",command=self.abort_all_button_pressed)
```

```
        self.abort_all_button.place(x=800, y=620)
        self.trace_force_button = tk.Button(self.root,text="TRACE
FORCE",command=self.trace_force_button_pressed)
        self.trace_force_button.place(x=800,y=700)
        self.trace_force_button2 = tk.Button(self.root,text="TRACE
FORCE2",command=self.trace_force_button_pressed2)
        self.trace_force_button2.place(x=800,y=750)
        self.make_contact_button = tk.Button(self.root,text="MAKE
CONTACT",command=self.make_contact_button_pressed)
        self.make_contact_button.place(x=600,y=700)

        self.stimuli = [Stimulus(position=1),Stimulus(position=2),Stimulus(position=3)]

        self.selected_stimuli = tk.IntVar()
        self.selected_stimuli_button1 = Radiobutton(self.root, text="Stim1", variable=self.selected_stimuli,
value=1, command=self.selected_stimuli_button_pressed)
        self.selected_stimuli_button1.place(x = 20, y = 600)
        self.selected_stimuli_button1 = Radiobutton(self.root, text="Stim2", variable=self.selected_stimuli,
value=2, command=self.selected_stimuli_button_pressed)
        self.selected_stimuli_button1.place(x = 100, y = 600)
        self.selected_stimuli_button1 = Radiobutton(self.root, text="Stim3", variable=self.selected_stimuli,
value=3, command=self.selected_stimuli_button_pressed)
        self.selected_stimuli_button1.place(x = 180, y = 600)

        self.force_indents_min = tk.Entry(self.root)
        self.force_indents_min.place(x=400,y=700)
        self.force_indents_min_label = tk.Label(self.root,text='Select min Force')
        self.force_indents_min_label.place(x=300,y=700)
        self.force_indents_max = tk.Entry(self.root)
        self.force_indents_max.place(x=400,y=750)
        self.force_indents_max_label = tk.Label(self.root,text='Select max Force')
        self.force_indents_max_label.place(x=300,y=750)
        self.force_n_steps = tk.Entry(self.root)
        self.force_n_steps.place(x=400,y=800)
        self.force_n_steps_label = tk.Label(self.root,text="N_Steps")
        self.force_n_steps_label.place(x=300,y=800)
        self.force_slow_point = tk.Entry(self.root)
        self.force_slow_point.place(x=400,y=835)
        self.force_slow_point_label = tk.Label(self.root,text="Slow Point")
        self.force_slow_point_label.place(x=300,y=835)
        self.begin_force_indents_button = tk.Button(self.root,text='Begin Force
indents',command=self.begin_force_indents_button_pressed)
        self.begin_force_indents_button.place(x=350,y=850)


        self.stimuli_info_text = tk.StringVar()
        self.stimuli_label = tk.Label(self.root,textvariable=self.stimuli_info_text)
        self.stimuli_label.place(x = 20,y=625)
```

```
        #histories
        self.t0 = time.time()
        self.loadcell_hist = ([],[],[])
        self.indenter_hist = ([],[])

        #bindings
        self.root.bind('<Up>', self.upKey)
        self.root.bind('<Down>', self.downKey)
        self.root.bind('<Right>', self.rightKey)
        self.root.bind('<Left>', self.leftKey)
        self.root.bind('<space>',self.spaceKey)


        self.root.focus_set()
        self.root.after(0, self.refreshScreen)
        self.root.after(0, self.collect_data)
       # self.camera = cam.Camera(False,self.root,self.motion_controller)
        self.root.mainloop()

    def begin_force_indents_button_pressed(self):
        print(self.force_indents_min.get())
        print(self.force_indents_max.get())
        print(self.force_n_steps.get())
        print(self.force_slow_point.get())
        slow_p = float(self.force_slow_point.get())
        forces =
np.linspace(float(self.force_indents_min.get()),float(self.force_indents_max.get()),float(self.force_n_ste
ps.get()))
        print(forces)
        for f in forces:
            ntrials = 1
            for i in range(ntrials):
                print('going')

self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,5)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_HOME)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_TARGET,slow_p)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)

self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,1)

self.motion_controller.indenter._command(self.indenter.MOVE_TO_TARGET,self.indenter.get_max_ind
entation())
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_FORCE,f)
                self.motion_controller.indenter._command(self.indenter.ABORT)
                self.motion_controller.indenter._command(self.indenter.PRINT_FORCE)
```

```
self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,5)
            self.motion_controller.indenter._command(self.indenter.MOVE_TO_HOME)
            self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)
            self.motion_controller.indenter._command(self.indenter.WAIT_FOR_INPUT)

    def spaceKey(self,event):
        #self.indenter.send_input()
        write_file = open('LOADCELL.out', 'wb')
        cPickle.dump(self.loadcell_hist, write_file )
        write_file.close()
        # print(self.time_r)
        # print(self.pos_r)
        # print(self.force_r)

    def leftKey(self,event):
        # p = Procedure()
        # p.add_displacement_control(1,1,self.stimuli[0])
        # p.add_displacement_control(3,3,self.stimuli[1])
        # p.add_force_control(1,1,self.stimuli[0])
        # self.motion_controller.run(p)
        self.time_r = []
        self.pos_r = []
        self.force_r = []

        p = Procedure()
        p.add(('BEGIN_GATHERING',))
        p.add(('MOVE_INDENTER_DOWN',1))
        p.add(('STOP_GATHERING',self.time_r,self.pos_r,self.force_r))
        p.add(('MOVE_INDENTER_UP',1))
        self.motion_controller.run(p)


    def indents_select_max_button_pressed(self):
        self.INDENTS_MAX = self.motion_controller.get_indenter_position()
        self.indents_max_text.set('Select Max ('+str(self.INDENTS_MAX) + ')')

    def indents_select_min_button_pressed(self):
        self.INDENTS_MIN = self.motion_controller.get_indenter_position()
        self.indents_min_text.set('Select Min ('+str(self.INDENTS_MIN) + ')')

    def indent_to_target_button_pressed(self):
        self.indenter._command(self.indenter.MOVE_TO_HOME)
        self.indenter._command(self.indenter.MOVE_TO_TARGET,self.TARGET_POS)
        self.indenter._command(self.indenter.WAIT_FOR_TIME,1)
        self.indenter._command(self.indenter.MOVE_TO_HOME)

    def set_target_pos_button_pressed(self):
```

```
        self.TARGET_POS = self.motion_controller.get_indenter_position()
        self.target_pos_text.set('Select Target '+str(self.TARGET_POS))

    def calibrate_zero_button_pressed(self):
        self.motion_controller.calibrate_load_cell_zero()

    def begin_indents_button_pressed(self):
        #self.motion_controller.indenter._command(self.indenter.WAIT_FOR_TIME,20)
        max = float(self.INDENTS_MAX)
        min = float(self.INDENTS_MIN)
        slow_p = float(self.force_slow_point.get())
        disps = np.linspace(min,max,self.INDENTS_N)
        for val in disps:
            if val > max:
                val = max
                print('uh oh')
            ntrials = 3
            for i in range(ntrials):

self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,10)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_HOME)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_TARGET,slow_p)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)

self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,5)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_TARGET,val)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)
                self.motion_controller.indenter._command(self.indenter.PRINT_FORCE)

self.motion_controller.indenter._command(self.indenter.CHANGE_MOVEMENT_PARAMETERS,10)
                self.motion_controller.indenter._command(self.indenter.MOVE_TO_HOME)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_READY)
               # self.motion_controller.indenter._command(self.indenter.WAIT_FOR_TIME,25)
                self.motion_controller.indenter._command(self.indenter.WAIT_FOR_INPUT)


    def abort_all_button_pressed(self):
        self.motion_controller.abort_indenter()

    def trace_force_button_pressed(self):
        t = [0, 1, 2, 3,5,]
        f =[0,3,3,0,0]
        self.motion_controller.trace_force(t,f)

    def trace_force_button_pressed2(self):
        p = Procedure()
        t = [0, 3, 4, 7,10,]
```

```
        f =[0,3,3,0,0]
        self.time_r = []
        self.pos_r = []
        self.force_r = []
        p.add(('BEGIN_GATHERING',))
        p.add(('TRACE_FORCE',t,f))
        p.add(('STOP_GATHERING',self.time_r,self.pos_r,self.force_r))
        self.motion_controller.run(p)

    def make_contact_button_pressed(self):
        self.motion_controller.make_contact()

    def selected_stimuli_button_pressed(self):
        print(self.selected_stimuli.get())

        self.motion_controller.select_stimulus(self.stimuli[self.selected_stimuli.get()-1])

    def indenter_home_button_pressed(self):
        self.motion_controller.move_indenter_home()

    def set_home_button_pressed(self):
        self.motion_controller.set_indenter_home_at_current_pos()

    def achieve_force_button_pressed(self):
        val = float(self.achieve_force_entry.get())
        force_indent = Procedure()
        force_indent.add_indent_to_force(None,None,None)
        self.motion_controller.run(force_indent)

    def set_max_indent_button_pressed(self):
        self.motion_controller.set_indenter_max_at_current_pos()

    def collect_data(self):
        if not self.loadcell.is_calibrating_zero():
            elapsed_time = time.time()-self.t0
            loadcell_val = self.motion_controller.load_cell.get_force()
           # loadcell_val = self.loadcell.get_output_voltage()
            self.loadcell_hist[0].append(elapsed_time)
            self.loadcell_hist[1].append(loadcell_val)
            self.loadcell_hist[2].append(self.motion_controller.get_indenter_position())
        else: pass #print (str(self.loadcell.calibrate_zero_progress()*100) + '%')
        indent_pos = self.motion_controller.get_indenter_position()
        indenter_pos_str = 'Indenter Position at: ' + str(indent_pos) + ' units'
        if indent_pos == self.motion_controller.indenter.get_max_indentation(): indenter_pos_str += '
(MAX)'
        elif indent_pos == self.motion_controller.indenter.get_home_position(): indenter_pos_str += '
(HOME)'
        self.indenter_position_text.set(indenter_pos_str)
```

```
        self.indenter_max_indent_text.set('Set Max Indent (Cur: ' +
str(self.motion_controller.indenter.get_max_indentation()) + ')')
        self.indenter_home_text.set('Set Home (Cur: ' +
str(self.motion_controller.indenter.get_home_position()) + ')')
#       self.indenter_status_text.set('Indenter Status: ' +
self.motion_controller.indenter.get_status_string())
        '''stimulus_info = self.motion_controller.get_current_stimulus_info()
        if stimulus_info is None: self.stimuli_info_text.set('No Stimulus Selected.')
        else:
          self.stimuli_info_text.set('\''+stimulus_info[0]+'\' '+', ' + str(stimulus_info[1]) + ' kPa ' +
stimulus_info[2])'''
        self.root.after(self.DATA_COLLECT_RATE,self.collect_data)

    def refreshScreen(self):
        self.plot_bounds = [0, 0, 0, 0]
        if(self.loadcell_axs == None): self.loadcell_axs = self.temp_plot.plot(self.loadcell_hist[0],
self.loadcell_hist[1])
        self.loadcell_axs[0].set_data(self.loadcell_hist[0],self.loadcell_hist[1])
        loval = -0.0000000001
        if(len(self.loadcell_hist[0]) == 0 or len(self.loadcell_hist[1])==0): pass
        else:
          if max(self.loadcell_hist[0]) > 0.5*60:
            loval = max(self.loadcell_hist[0])-0.5*60
          self.temp_plot.set_xlim([loval , max(self.loadcell_hist[0])])
          self.temp_plot.set_ylim([min(self.loadcell_hist[1])-.00000001, max(self.loadcell_hist[1])])

        self.tk_figure.draw()
        self.root.after(self.REFRESH_WAIT, self.refreshScreen)

    #keypresses
    def upKey(self,event):
        val = float(self.dist_entry.get())
        self.motion_controller.move_indenter_up(val)

    def downKey(self,event):
        val = float(self.dist_entry.get())
        self.motion_controller.move_indenter_down(val)

    def rightKey(self,event):
        self.motion_controller.stop_procedure()

    def calibrate_button_pressed(self):
        force = float(self.force_entry.get())
        voltage = self.average_loadcell(self.CALIB_NPOINTS)
        self.calibration_data[voltage] = force

    def save_calibration(self):
        write_file = open('calibration.out', 'wb')
```

```
        cPickle.dump(self.calibration_data, write_file )
        write_file.close()

    def average_loadcell(self,npoints):
        sum = 0.
        for i in range(npoints):
            sum = sum + self.loadcell.get_output_voltage()
        return sum/npoints


calib = Calibration_Window()
```

CONSTANTS.py

```
__author__ = 'Steven'
class CONSTANTS:
    # ALL CONSTANTS STORED HERE                                                 #
    # SOME OF THESE CONSTANTS MAY BE STORED INDIVIDUALLY IN CLASSES AS WELL, FROM THIS FILE #
    # **READ-ONLY DATA, THREAD-SAFE**                                           #

    # **IN CODE, ALL THREAD-UNSAFE METHODS BEGIN WITH AN UNDERSCORE (_)
    # OTHER METHODS ARE THREAD-SAFE

    #THREAD IDs
    THREAD_ID_MAIN = 0
    THREAD_ID_LOADCELL_RECORD = 1
    THREAD_ID_MOTOR_HOLD_POSITION = 2

    #TODO RETIRE TO GPIO CLASS
    #GPIO PINS
    PIN_MOTOR_CLOCK = None
    PIN_MOTOR_SIGNAL = None
    PIN_MOTOR_ACK = None
    PIN_LOADCELL_IN = 14
    PIN_FUNGEN_RED = None
    PIN_FUNGEN_BLACK = None
    PIN_ANALOG_OUT = 19
    PIN_ANALOG_IN = 16
    PIN_VELOCITY_OUT = 20

    #TODO I DONT THINK THIS SHOULD BE IN HERE!
    #STIMULUS SELECTOR
    NUM_STIMULI = 3

    #MOTOR
    ARDUINO_VOLTAGE_HIGH = 5 #V
    ARDUINO_VOLTAGE_LOW = 0 #V
    MOTOR_SIG_NBITS = 9 #bits
    MOTOR_MAX_ROTATION = 140 #degrees
```

```
MOTOR_ACK_TIMEOUT = 10 #ms

#LOAD CELL
LOADCELL_CALIB_ZERO_NPOINTS = 2000
LOADCELL_CALIB_FILE = 'calibration.out'
LOADCELL_MAX_FORCE = 22.2 #N

#INDENTER
INDENTER_GROUP_NAME = 'GROUP1'
#INDENTER_POSITIONER_NAME = 'GROUP1.POSITIONER1'
INDENTER_POSITIONER_NAME = 'GROUP1.POSITIONER'
#INDENTER_POSITIONER_NAME = 'G1.P1'
INDENTER_DEFAULT_VELOCITY = 10
INDENTER_DEFAULT_VELOCITY = 1
INDENTER_DEFAULT_ACCELERATION = 80
INDENTER_DEFAULT_JERK = 0.02

INDENTER_DEFAULT_HOME_POSITION = 20
INDENTER_DEFAULT_MAX_INDENTATION = 100
#INDENTER_DEFAULT_MAX_INDENTATION = 35 ORIGINAL!!!
INDENTER_DEFAULT_MAX_INDENTATION = 100

#INDENTER_DEFAULT_HOME_POSITION = 0
#INDENTER_DEFAULT_MAX_INDENTATION = 50

INDENTER_AWAITING_MOVEMENT_TIMEOUT = 0.1     #seconds

INDENTER_DEFAULT_MIN_INDENTATION = 0

INDENTER_STATUS_ABORTING = 0
INDENTER_STATUS_MOVING_TO_TARGET = 1
INDENTER_STATUS_READY = 2
INDENTER_STATUS_MOVING_TO_HOME = 3
INDENTER_STATUS_INDENTING_TO_FORCE = 4
INDENTER_STATUS_JOGGING = 5
INDENTER_STATUS_WAITING = 6
INDENTER_STATUS_WAITING_FOR_INPUT = 7

INDENTER_COMMAND_ABORT = 0
INDENTER_COMMAND_MOVE_TO_TARGET = 1
INDENTER_COMMAND_WAIT_FOR_READY = 2
INDENTER_COMMAND_NO_COMMAND = 3
INDENTER_COMMAND_WAIT_FOR_FORCE = 4
INDENTER_COMMAND_CHANGE_MOVEMENT_PARAMETERS = 5
INDENTER_COMMAND_MOVE_TO_HOME = 6
INDENTER_COMMAND_BEGIN_JOGGING = 7
INDENTER_COMMAND_CHANGE_JOGGING_PARAMETERS = 8
INDENTER_COMMAND_END_JOGGING = 9
```

```
    INDENTER_COMMAND_WAIT_FOR_TIME = 10
    INDENTER_COMMAND_TRACE_FORCE = 11
    INDENTER_COMMAND_MOVE_UP = 12
    INDENTER_COMMAND_WAIT_FOR_FORCE_LOW = 13
    INDENTER_COMMAND_END_TRACKING = 14
    INDENTER_COMMAND_PRINT_FORCE = 15
    INDENTER_COMMAND_WAIT_FOR_INPUT = 16

    INDENTER_MODE_JOG = 0
    INDENTER_MODE_MOVEMENT = 1
    INDENTER_MODE_TRACKING = 2

    #RUNS
    RESULT_SAME = 0
    RESULT_DIFFERENT = 1

class GPIO:
    PINS =
{14:'GPIO2.ADC1',15:'GPIO2.ADC2',16:'GPIO2.ADC3',17:'GPIO2.ADC4',19:'GPIO2.DAC1',20:'GPIO2.DAC2',
21:'GPIO2.DAC3',22:'GPIO2.DAC4'}
    PIN_MOTOR_CLOCK = None
    PIN_MOTOR_SIGNAL = None
    PIN_MOTOR_ACK = None
    PIN_LOADCELL_IN = 14
    PIN_FUNGEN_RED = None
    PIN_FUNGEN_BLACK = None
    PIN_ANALOG_OUT = 19
    PIN_ANALOG_IN = 16
    PIN_VELOCITY_OUT = 20
    PIN_POSITION_OUT = 21
```

Imaging2.py

```
__author__ = 'Steven'
from PIL import Image
import Tkinter as tk
import matplotlib
import copy
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
import threading
import scipy.spatial
from scipy import misc
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
from matplotlib.figure import Figure

class Image_Window:

   def center_select_pressed(self):
       self.mode = self.SELECTING_POINT

   def onmove(self,event):
      if self.mode == self.TRACKING:
         print('mvoed')
         x = event.xdata
         y = event.ydata
         if x is None or y is None: return
         dist = (float(x-self.tracking_point[0])**2+float(y-self.tracking_point[1])**2)**(0.5)
         print(dist)
         #self.tk_figure.get_tk_widget().coords(self.tracking_circle,(self.tracking_point),dist)
         self.tracking_circle.radius = dist
      elif self.mode == self.SECOND_POINT:
         x = event.xdata
         y = event.ydata
         dist = (float(x-self.tracking_point[0])**2+float(y-self.tracking_point[1])**2)**(0.5)
         ang = np.arctan2((self.tracking_point[0]-x),(self.tracking_point[1]-y))
         print(x,y)
         print(np.cos(ang))
         self.tracking_line.set_xdata([x,x+2*dist*np.sin(ang)])
         self.tracking_line.set_ydata([y,y+2*dist*np.cos(ang)])
         #self.tk_figure._tkcanvas.coords(self.tracking_circle,(self.tracking_point),dist)
         #self.tk_figure._tkcanvas.draw()

   #"SHOELACE" ALGORITHM
   def find_area(self,arr):
      a = 0.
```

```
    ox,oy = arr[0]
    ox = float(ox)
    oy = float(oy)
    for x,y in arr[1:]:
        x = float(x)
        y = float(y)
        a += (x*oy-y*ox)
        ox,oy = x,y
    return abs(a/2.)

def onclick(self,event):
    print ('label',self.toolbar._message_label.cget('text'))
    print 'zoom rect' in self.toolbar._message_label.cget('text')
    if 'zoom rect' in self.toolbar._message_label.cget('text'):
        print 'yeppers'
        return
    if event.button == 1:
        if self.mode == self.SELECTING_POINT:
            print 'button=%d, x=%d, y=%d, xdata=%f, ydata=%f'%(
            event.button, event.x, event.y, event.xdata, event.ydata)
            self.ax.plot(event.xdata,event.ydata,'.k')
            self.centers.append((event.ydata,event.xdata))
            self.ax.set_xlim([0,self.pic.shape[1]])
            self.ax.set_ylim([0,self.pic.shape[0]])
            self.tk_figure.draw()
            self.tracking_point = (event.xdata,event.ydata)

            self.tracking_circle = plt.Circle(self.tracking_point,0,color='k',fill=False)
            self.ax.add_artist(self.tracking_circle)
            self.mode = self.TRACKING

            #self.mode = self.SECOND_POINT
            #self.tracking_line = plt.Line2D([0,0],[0,0],color='k')
            #self.ax.add_artist(self.tracking_line)
        elif self.mode == self.SECOND_POINT:
            x = event.xdata
            y = event.ydata
            self.ax.plot(event.xdata,event.ydata,'.k')
            ang = np.arctan2((self.tracking_point[0]-x),(self.tracking_point[1]-y))
            print('ang',ang)
            if ang < 0: ang = ang+2*np.pi
            self.angles[(self.tracking_point[1],self.tracking_point[0])] = ang
            self.tracking_circle = plt.Circle(self.tracking_point,0,color='k',fill=False)
            self.ax.add_artist(self.tracking_circle)
            self.mode = self.TRACKING
        elif self.mode == self.TRACKING:
            x = event.xdata
            y = event.ydata
```

```
            if x is None or y is None: return
            dist = (float(x-self.tracking_point[0])**2+float(y-self.tracking_point[1])**2)**(0.5)
            self.circles[self.tracking_point] = dist
            self.mode = self.SELECTING_POINT
            boolar =
scipy.spatial.distance.cdist(np.array([[self.tracking_point[1],self.tracking_point[0]]]),self.pic_coords) <=
dist
            self.points[(self.tracking_point[1],self.tracking_point[0])] = self.pic_coords[boolar[0],]
         elif self.mode == self.SELECTING_DISTANCE:
            print 'here'
            if len(self.distance_points) == 0:
               self.distance_points.append((float(event.xdata),float(event.ydata)))
            elif len(self.distance_points) == 1:
               self.distance_points.append((float(event.xdata),float(event.ydata)))
               x1,y1,x2,y2 =
self.distance_points[0][0],self.distance_points[0][1],self.distance_points[1][0],self.distance_points[1][1]
               self.distance_ratio = 5/(((x1-x2)**2.+(y1-y2)**2.)**(0.5))
               self.area_ratio = self.distance_ratio**2.
               print(self.area_ratio)
               self.mode = self.NONE

   def __init__(self):
      self.root = tk.Tk()
      self.centers = []
      self.circles = {}
      self.distance_points = []
      self.distance_ratio = None #in cm/pix_len
      self.area_ratio = None #in cm^2/pix
      self.points = {}
      self.angles = {}

      self.NONE = 0
      self.TRACKING = 1
      self.SELECTING_POINT = 2
      self.SELECTING_DISTANCE = 3
      self.SECOND_POINT = 4
      self.mode = self.NONE
      #pic = misc.imread('example_spots.png')
      #pic = misc.imread('IMAG0070.png')
      #pic = misc.imread('rsz_imag0070.png')
      self.pic = misc.imread('Image.jpg')
      #self.pic = misc.imread('unnamed (1).jpg')
      #self.pic = misc.imread('IMAG0086.jpg')
      #self.pic = misc.imread('IMAG0087.jpg')
      #self.pic = misc.imread('IMAG0088.jpg')
      #self.pic = misc.imread('hard1.jpg')
      #self.pic = misc.imread('IMAG0091_BURST002.jpg')
```

```
#IN FIGURE 3:
#self.pic = misc.imread('hard_1_10-12.jpg')

self.file_str = '10_19/hard_2_AB.jpg'

self.pic = misc.imread(self.file_str)
self.pic_width = self.pic.shape[0]
self.pic_height = self.pic.shape[1]
self.pic_coords = []
for i in range(self.pic_width):
   for j in range(self.pic_height):
      self.pic_coords.append([i,j])
self.pic_coords = np.array(self.pic_coords)
self.display_pic = self.pic.copy()
self.fig = plt.figure()
self.ax = self.fig.add_subplot(111)
self.imageax = self.ax.imshow(self.display_pic)
self.ax.set_xlim([0,self.pic.shape[1]])
self.ax.set_ylim([0,self.pic.shape[0]])

self.rendering_thread = threading.Thread(target=self.render_red)
self.is_rendering = False
self.first_render = True
self.should_stop_rendering = False
self.display_pic_lock = threading.Lock()
self.pic_lock = threading.Lock()
self.lock = threading.Lock()

#self.tracking_on = False
self.tracking_point = None


self.mode_var = tk.StringVar()
self.mode_label = tk.Label(self.root,textvariable=self.mode_var)
self.mode_label.grid()

#fig = plt.imshow(pic)


self.tk_figure = FigureCanvasTkAgg(self.fig, master = self.root)
#NavigationToolbar2TkAgg(self.tk_figure,self.root)
toolbar_frame = tk.Frame(self.root)
toolbar_frame.grid(columnspan=2,sticky=tk.W)
self.toolbar = NavigationToolbar2TkAgg(self.tk_figure, toolbar_frame )

self.tk_figure.get_tk_widget().grid(sticky=tk.W)

self.rendering_text_var = tk.StringVar()
```

```
        self.rendering_progress = -1
        self.rendering_text = tk.Label(self.root,textvariable=self.rendering_text_var)
        self.rendering_text.grid()

        self.center_select_button = tk.Button(self.root,text='Select
Centers',command=self.center_select_pressed)
        self.center_select_button.grid()
        self.red_slider = tk.Scale(self.root,from_=1, to=1.4,resolution=.005,command=self.render_red)
        self.current_val = None
        self.red_slider.grid()
        self.set_distance_button = tk.Button(self.root,text='Set
Distance',command=self.set_distance_button_pressed)
        self.set_distance_button.grid()
        self.find_prints_button = tk.Button(self.root,text='Find Prints',command=self.find_prints)
        self.find_prints_button.grid()
        self.calculate_areas_button = tk.Button(self.root,text='Calculate
Areas',command=self.calculate_areas)
        self.calculate_areas_button.grid()


        cid = self.tk_figure.mpl_connect('button_press_event', self.onclick)
        self.tk_figure.mpl_connect('motion_notify_event', self.onmove)

        self.root.after(0,self.refresh)
        self.root.mainloop()

        self.tk_figure.show()

    def set_distance_button_pressed(self):
        #window = tk.Toplevel()
        self.mode = self.SELECTING_DISTANCE

    def calculate_areas(self):
        pass

    def is_red(self,pixel,val):
        #val = float(self.red_slider.get())
        avg = (float(pixel[1])+float(pixel[2]))/2.
        #print(avg)
        if float(pixel[0]) > avg*val:
            return True
        else: return False

    def find_prints(self):
        fingerprints = []
        fingerprint_outlines = []
        for tracking_point in self.centers:
            outline = []
```

```
        search_points = self.points[tracking_point].tolist()
        search_points.sort(key=lambda lis: (lis[1],lis[0]))
        #print(search_points)
        red_found = False
        for i in range(len(search_points)-1):
            point = search_points[i]
            next_point = search_points[i+1]
            if next_point[1]>point[1]: #new line
                red_found = False
            elif red_found:
                pass
            #print(self.pic[next_point[0],next_point[1]])
            if self.is_red(self.pic[next_point[0],next_point[1]],self.current_val) and not red_found:
                with self.display_pic_lock: self.display_pic[point[0],point[1]] = [0,0,0]
                outline.append([-tracking_point[1]+point[1],-tracking_point[0]+point[0]])
                red_found = True
        search_points.sort(key=lambda lis: (lis[1],lis[0]),reverse=True)
        red_found = False
        for i in range(len(search_points)-1):
            point = search_points[i]
            next_point = search_points[i+1]
            if next_point[1]<point[1]: #new line
                red_found = False
            elif red_found:
                pass
            #print(self.pic[next_point[0],next_point[1]])
            if self.is_red(self.pic[next_point[0],next_point[1]],self.current_val) and not red_found:
                with self.display_pic_lock: self.display_pic[point[0],point[1]] = [0,0,0]
                #p = [-tracking_point[1] + point[1],-tracking_point[0]+point[0]]
                #theta = self.angles[tracking_point]
                #theta = (np.pi)/2.
                #new_p_prime = p
                #new_p_prime = [p[0]*np.cos(theta)-
p[1]*np.sin(theta),p[1]*np.cos(theta)+p[0]*np.sin(theta)]
                outline.append([-tracking_point[1] + point[1],-tracking_point[0]+point[0]])
                #outline.append(new_p_prime)
                red_found = True
        outline.append(outline[0])
#         theta = self.angles[tracking_point]
        theta = 0
        outline = np.dot(np.array([[np.cos(theta),-
np.sin(theta)],[np.sin(theta),np.cos(theta)]]),np.transpose(outline))
        fingerprint_outlines.append(np.transpose(outline))

    new_window = tk.Toplevel()
    fig = plt.figure()
    ax = fig.add_subplot(111)
    tk_figure = FigureCanvasTkAgg(fig, master = new_window)
```

```
        tk_figure.get_tk_widget().grid(sticky=tk.W)
        num = 0
        final_dat = []
        for outline in fingerprint_outlines:
            num = num +1
            x = []
            y = []
            for point in outline:
                x.append(point[0]*self.distance_ratio)
                y.append(point[1]*self.distance_ratio)
            #"SHOELACE" ALGORITHM
            #print ('area?',str(self.find_area(outline)*self.area_ratio)+'cm^2')
            ax.plot(x,y)
            final_dat.append(x)
            final_dat.append(y)
            final_dat.append([self.find_area(outline)*self.area_ratio])
            #print('xh'+str(num)+'A='+str(x)+';')
            #print('yh'+str(num)+'A='+str(y)+';')
            #print('ah'+str(num)+'A='+str(str(self.find_area(outline)*self.area_ratio))+';')
        longest = 0
        for dat in final_dat:
            if len(dat) > longest:
                longest = len(dat)

        final_mat = np.NaN*np.ones([len(final_dat),longest])
        i1 = 0
        for dat in final_dat:
            i2 = 0
            for dat_in in dat:
                final_mat[i1,i2] = dat_in
                i2 = i2 + 1
            i1 = i1 + 1


        #np.savetxt(, X, fmt='%.18e', delimiter=',', newline='\n')
        np.savetxt(str(self.file_str.split('.')[0])+'_processed.txt',final_mat, fmt='%.18e', delimiter=',',
newline='\n')
        plt.gca().set_aspect('equal', adjustable='box')
        lis = []
        lis.extend(plt.xlim())
        lis.extend(plt.ylim())
        lis2 = []
        for num in lis:
            lis2.append(abs(num))
        print('max:',max(lis))
        plt.xlim(-max(lis2), max(lis2))
        plt.ylim(-max(lis2), max(lis2))
        plt.gca().set_aspect('equal', adjustable='box')
```

```
        plt.draw()


        #plt.show()

    def render_red(self,val):
        val = float(val)
        self.current_val = val
        if self.first_render:
            self.first_render = False
            return
        print(val)
        with self.lock: can_proceed = not self.is_rendering
        if not can_proceed:
            with self.lock:
                self.should_stop_rendering = True
            while True:
                with self.lock:
                    if not self.is_rendering: break
        with self.lock:
            self.is_rendering = True
            self.should_stop_rendering = False
        currcenters = copy.deepcopy(self.centers)
        currpoints = copy.deepcopy(self.points)
        self.rendering_thread =
threading.Thread(target=self._render_image,args=(val,currcenters,currpoints))
        self.rendering_thread.start()

    def _render_image(self,val,currcenters,currpoints):
        print('rendering')
        with self.pic_lock: new_pic = self.pic.copy()
        width = self.pic.shape[0]
        height = self.pic.shape[1]
        total = float(width)*float(height)
        # for i in range(width):
        #     with self.lock:
        #         if self.should_stop_rendering: break
        #     for j in range(height):
        #         with self.lock:
        #             if self.should_stop_rendering: break
        #         pixel = self.pic[i,j]
        #         if self.is_red(pixel,val):
        #             new_pic[i,j] = [255,0,0]
        #         position = float(i*width+j)
        #         with self.lock: self.rendering_progress = position/total
        for center in currcenters:
            for point in currpoints[center]:
                i,j = point
```

```
            with self.lock:
                if self.should_stop_rendering: break
            pixel = self.pic[i,j]
            if self.is_red(pixel,val):
                new_pic[i,j] = [255,0,0]
            position = float(i*width+j)
            with self.lock: self.rendering_progress = position/total
        if not self.should_stop_rendering:
            with self.display_pic_lock:
                self.display_pic = new_pic.copy()
        print('done rendering')
        self.rendering_progress = -1
        self.is_rendering = False


    def refresh(self):
        self.mode_var.set(self.mode)
        with self.display_pic_lock:
            self.imageax.set_data(self.display_pic)
            self.tk_figure.draw()
            #print(self.display_pic)
        with self.lock:
            if self.rendering_progress == -1: self.rendering_text_var.set('Rendering complete.')
            else: self.rendering_text_var.set('Rendering... (' + "{0:.1f}".format(self.rendering_progress*100.) +
'%)')
        self.root.after(1000/60,self.refresh)

Image_Window()
```