An Extensible Software Library for Developing Tactile Perception Experiments

by

Samir V. Meghani

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology May 20, 2004

© 2004 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author ..........................................................................................................................
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....................................................................................................................
Dr. Mandayam A. Srinivasan
Senior Research Scientist, Department of Mechanical Engineering
Thesis Supervisor

Accepted by ....................................................................................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
An Extensible Software Library for Developing Tactile Perception Experiments

by

Samir V. Meghani

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 2004, in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science

Abstract

A system was designed to reduce the time undergraduate student researchers spend programming while developing tactile perception experiments. The system allows students to program in Matlab, a high-level, interpreted language, and captures functionality common to most experiments in an easily extensible library. To further aid the student, an experiment template that illustrates the structure of a typical experiment is provided. The effectiveness of the system was evaluated by implementing a real-world experiment for measuring the 2-point error of localization of touch on the hand. The experience indicates a decrease in the total programming time and suggests the potential usefulness of interpreters and converters in general software development.

Thesis Supervisor: Dr. Mandayam A. Srinivasan
Title: Senior Research Scientist, Department of Mechanical Engineering
Acknowledgments

Writing this thesis has been a challenging and rewarding experience. I would like to express my gratitude to Dr. Mandayam A. Srinivasan for giving me the opportunity to work in the Touch Lab and encouraging me to experience research that was not purely computer science. My deepest thanks also go to Dr. David Schloerb for his guidance throughout the project. He has kept me focused and pushed me in the right direction when I was veering off course.

I would also like to thank all of the members of the Touch Lab for their help in various aspects of the project. Dr. James Biggs has assisted me with many technical issues that would have taken me hours to solve otherwise. Angela Glass and Monica Pegis have provided administrative support and promptly answered my many questions. Several others have assisted me in resolving day-to-day problems.

This work was supported by Defense Advanced Research Projects Agency(DARPA) Contract N66001-02-C-8022 and National Institute of Health(NIH) Grant 5R01-NS33778.
# Contents

1 Introduction ........................................ 9
   1.1 Overview ..................................... 9
   1.2 Organization ................................. 11

2 Background ......................................... 12
   2.1 Tactile Perception Test Apparatus .......... 12
   2.2 Psychophysics Toolbox ....................... 14
   2.3 Gentle Slope Systems ........................ 14
   2.4 Learning by Example ......................... 15
   2.5 Compiled and Interpreted Languages ........ 15

3 Design and Implementation ....................... 17
   3.1 Problem Analysis ............................ 17
      3.1.1 Student Background ..................... 17
      3.1.2 C Programming .......................... 18
      3.1.3 Code Reuse ............................. 19
   3.2 Design ....................................... 19
      3.2.1 Preliminary Design ..................... 19
      3.2.2 Improvements ........................... 20
   3.3 Implementation ............................... 22
      3.3.1 Implementation Considerations .......... 22
      3.3.2 Implementation Decisions ............... 23
A.4 Controlling the Indentors ........................................... 48
A.5 Communicating with Subjects ........................................ 50
  A.5.1 Getting Input .................................................. 50
  A.5.2 Messaging Subjects ............................................ 50
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Tactile Perception Test Apparatus Block Diagram</td>
<td>13</td>
</tr>
<tr>
<td>2-2</td>
<td>Typical TPTA Stimulus</td>
<td>13</td>
</tr>
<tr>
<td>3-1</td>
<td>Test Procedure Psuedo-Code</td>
<td>21</td>
</tr>
<tr>
<td>4-1</td>
<td>Stimulation Site</td>
<td>28</td>
</tr>
<tr>
<td>4-2</td>
<td>Experimental Apparatus</td>
<td>29</td>
</tr>
<tr>
<td>4-3</td>
<td>Stimulus Signal</td>
<td>30</td>
</tr>
<tr>
<td>4-4</td>
<td>Results for Seven Subjects</td>
<td>31</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

In recent years, computers and microcontrollers have become widely used in the control of electrical and mechanical systems. Programming has, as a result, become a crucial step in the development of such systems. While there are several benefits to this shift, it also causes a notable problem. Engineers, who had previously focused on solving electrical and mechanical problems, must instead develop software. Many of these engineers have little programming experience, which causes an increase in the total development time. The value of a method which would allow inexperienced programmers to develop software more quickly is clear.

The Tactile Perception Test Apparatus (TPTA) is one system which illustrates this point particularly well. The MIT Laboratory for Human and Machine Haptics (Touch Lab) developed the TPTA to measure biomechanical properties of skin and determine various perceptual limits of touch. The TPTA is a significant technological improvement over previous experimental apparatus, as it will allow researchers to present precise tactile stimuli to a subject, while also recording the resulting force. A significant advantage of the apparatus is that a wide range of experiments can be performed simply by implementing new software.

Unfortunately, the expected benefit of the TPTA has not been fully realized to date because of the effort required to develop the software to run experiments. Post-
doctoral scientists and graduate students in tactile perception have spent months performing the tedious tasks of writing and testing code for experiments on similar apparatus in the past. Furthermore, the Touch Lab would prefer to assign this task to undergraduate students. These students are often inexperienced programmers and do not commit to projects for long periods. The goal of this thesis is to reduce the time students spend programming in the development of an experiment, and allow them to focus on experimental design. In the process, it aims to provide insights into the broader problem of reducing programming time when inexperienced programmers are used.

This thesis shortens the software development time by implementing an extensible library of functions called from the Matlab programming language, similar to the work of Brainard and Pelli[1, 13]. The library provides powerful functionality, including a hardware programming interface, user interface components, and file input/output capability. Because Matlab is a gentle slope, high-level programming language, it is easy to learn and enables quicker development. In order to ensure students understand how to use the library, tutorials and practical examples are provided which clearly illustrate the library’s functions and their interactions. A template for experiments which can be completed by following a step-by-step process is also provided, so students understand how to structure their experiments.

To measure the effectiveness of the library, an actual experiment to determine the 2-point error of localization of human touch on the hand was developed and conducted. The experience proves that the system exposes enough functionality to build an actual experiment and provides evidence that the environment created is conducive to rapid experiment development. Our experience indicates that the majority of the time spent on developing the experiment involved iteratively modifying the experimental design, rather than programming. Use by future students will reveal whether the tutorials, examples, and templates do in fact encourage students to use the library.
1.2 Organization

The remainder of the thesis is organized as follows. Chapter 2 provides some of the background material upon which this work relies. Chapter 3 presents the design of the library, and Chapter 4 presents the experiment which was used to measure the effectiveness of the library. Chapter 5 discusses the strengths and weaknesses uncovered in developing and running the experiment and the implications of this thesis on the broader problem of developing software with inexperienced programmers. Chapter 6 concludes and suggests future work. A tutorial for using the library appears in Appendix A.
Chapter 2

Background

2.1 Tactile Perception Test Apparatus

Though the Tactile Perception Test Apparatus (TPTA) is not the focus of this thesis, understanding its design will help clarify the design of the system built. The structure of the TPTA is shown in Figure 2-1. At the heart of the system is a pair of independently controlled indentors which stimulate a subject's skin, while measuring the resulting force. The positioning of the indentors is adjustable via a 4-axis micropositioner. A computer equipped with a digital acquisition card outputs signals to control the indentation depth of the indentors as a function of time and makes digital records of the measured forces. The position and force signals are highly precise - 1 micron position resolution and .3 mN force resolution.

The TPTA enables researchers to perform a wide range of experiments because it can present complex stimuli such as that shown in Figure 2-2 with great precision. Using earlier versions of the TPTA, researchers have performed experiments to measure tactile properties when stimulated at a single point. These measurements include the frequency response of human skin[5] and human perception thresholds for vibratory stimuli[7]. Both experiments used the measured reaction forces to calculate power dissipated, which had not been previously measured. Such experiments are important in understanding human touch and may help determine specifications for input/output devices which function using touch.
Figure 2-1: Block Diagram of the Tactile Perception Test Apparatus

Figure 2-2: A Typical TPTA Stimulus. Labeled values may vary by experiment.
2.2 Psychophysics Toolbox

The Psychophysics Toolbox[1, 13] was built by visual perception researchers to address a problem in their field similar to that of this thesis. Students needed to develop experiments involving the presentation of visual stimuli. This required writing software to interface directly with display hardware, using a low-level language like C. The developers found that students often re-implemented this component of the experiment, rather than reusing previous work. Because students did not generally know C very well and had to deal with numerous hardware details, they wasted a significant amount of time. To avoid this time drain, Brainard and Pelli created the Psychophysics Toolbox. The Toolbox is essentially a set of functions called from Matlab which provide an interface to display hardware. The project has been very successful, as the toolbox has become a widely cited tool in visual perception research papers.

2.3 Gentle Slope Systems

A gentle slope system is one where the usefulness of the results provided by the system grows linearly with the effort required to get the results. Dertouzos’ example of an individual trying to do accounting using his computer illustrates the problem addressed by gentle slope systems[4]. In the beginning, the individual uses a spreadsheet program to add and multiply numbers, and he can do everything he needs. Eventually, he needs to do something more complicated, and has to write macros to implement this functionality. This requires learning the complicated syntax for macros. After some more time passes, the individual decides he would like a better user interface than what his spreadsheet program provides. He must now learn to program in C++ to implement the user interface. As the individual needed more functionality, he had to learn an ever-increasing amount. In a gentle slope system, an incremental increase in effort results in an incremental increase in usefulness.

Gentle slope programming systems apply this concept to software development.
Such systems are intended for beginning programmers who generally start out writing simple programs, but can still be used by the programmer when he wants to develop more complicated programs. For example, CURL is considered a gentle slope language[8], whereas normal Windows programming is not[12]. The key property of a gentle slope development system is consistency in the linguistic constructs and syntax. Writing a simple Windows program using Visual Basic requires using only the BASIC programming language. However, as more complicated functionality is required, the developer must begin to use C. Alternatively, CURL is linguistically and syntactically consistent no matter what level of complexity is desired. This thesis aims to create a gentle slope development system.

2.4 Learning by Example

The most widely used methods of learning to use a software library are reading books or taking training courses. Effective books and courses make extensive use of example programs to demonstrate the function of the library. However, because examples tend to be toy programs, books and courses often do not explain the more obscure details of the library which are necessary for building real-world programs. For instance, a programmer who needs to build a word processor may need to figure out how to add custom functionality to a text editor, which he is unlikely to get from an overview of the user interface libraries that implement the text editor. This leaves the programmer with two options: figure it out for himself which can be time-consuming, or find example programs that do something similar and modify them to suit his needs. The second option is more efficient [11, 14]. It is therefore useful for the programmer to have access to real-world programs.

2.5 Compiled and Interpreted Languages

Programming languages generally fall in one of two categories: compiled or interpreted. In a compiled language, a compiler converts programs to machine-specific,
native code prior to execution. Interpreted languages instead convert programs into machine code during program execution. Both types of languages have relative advantages over the other. Compiled programs tends to be faster because there is no overhead of code conversion and the compiler can make optimizations. However, programming in an interpreted language is generally more productive because programmers do not need to learn how to use the compiler or wait for long compilations to finish. Additionally, the programmer does not have to recompile every time he makes a minor modification, which can reduce debugging time.
Chapter 3

Design and Implementation

3.1 Problem Analysis

The Tactile Perception Test Apparatus is a potentially powerful tool for researchers in haptics, as it can be used in a wide range of perception experiments. The Touch Lab would like to perform many of these experiments, with undergraduate students responsible for designing and running the experiments. These students may come from a variety of backgrounds and commit to the project for only short periods of time. Experience indicates that the typical student has little, if any, programming experience and works on a project part-time for less than an academic year (nine months). Given that graduate students with programming experience have previously spent several months writing and debugging programs for similar apparatus, the typical undergraduate is not likely to complete designing, implementing, and running an experiment on the TPTA in the time available. The designed system must reduce the amount of time the student spends programming and enable students to focus on experimental design. This section examines the reasons programming takes so long.

3.1.1 Student Background

Though the Touch Lab would prefer students with a strong programming background, it accepts students without this background if they are otherwise suited to perform-
ing an experiment. As a result, students could spend several weeks learning a programming language before they even begin to implement an experiment. When the programming begins, the inexperience of the students causes them to produce code which is poorly structured and hard to read[9]. The programs are therefore difficult to modify and update when minor changes need to be made to the experimental design[3]. Additionally, the students do not know how to properly test software, so they have difficulty finding and correcting bugs[2].

3.1.2 C Programming

The computer controls the TPTA utilizing a motion control card and a data acquisition card. Both of the cards have a C programming interface. Past students have therefore used the C programming language to implement the code for their experiments. However, the choice is not ideal. Programming in C tends to be a slow process for inexperienced programmers, for the following reasons.

- C is a low-level language. Students have to perform error-prone tasks such as managing memory. This can lead to memory leaks and out of memory errors which are difficult to isolate.

- C has very little error checking built in. For example, there is no array bounds checking, so programmers can write off of the end of an array. Such errors will cause the program to crash at unpredictable times. Locating such errors is time consuming.

- C is a compiled language. As described in Chapter 2, developing using a compiled language is typically slower than programming in an interpreted language.

- Developing in C is not gentle slope. Gentle slope languages are more conducive to writing complex programs quickly, as explained in Chapter 2.
3.1.3 Code Reuse

Most tactile perception experiments have a common structure and perform similar sub-functions. For instance, most experiments consist of a series of trials in which a subject is presented a stimulus and must provide a response. The user’s response must be recorded and stored in a file. Each student re-implements such functionality, even though previous students have written and tested similar code. There are two reasons for the lack of reuse. First, code written by other students is poorly documented and the current student thinks trying to read and understand it will take longer than re-implementing it. Second, previous students’ code is sometimes poorly structured, making it impossible to extract a module from it. Students could gain significant productivity by reusing code, but they opt not to do so.

3.2 Design

3.2.1 Preliminary Design

A preliminary solution to the problems described above is to build a library which captures functionality common to most experiments. The student would interface with the library through a high-level, interpreted language that is easy to learn and does not have the negative characteristics of C. The library would be easily extensible, so students could add functions to it that may have been overlooked originally. In practice, however, this solution is insufficient for two reasons. First, students tend to use only small parts of the library, because they do not want to read detailed software specifications for many of the functions. Second, the student is unable to put the pieces of the library together to implement complex experiments. In many cases, the student will not make an effort to use the library and will re-implement many of the functions. Though the student has the benefit of now working in an interpreted language which will allow him to program more quickly, the overall situation is only slightly improved.
3.2.2 Improvements

The preliminary solution is a good starting point, but needs modification. Students need to understand the library and how to use it to build a real-world experiment. The following improvements are provided to address these issues.

- Tutorials which illustrate the library’s function. The tutorials will serve to explain what each library module does in ordinary English. Rather than reading software specifications, students will be able to quickly understand each function.

- A template for experiments. Most tactile perception experiments follow the same general structure. Templates organize this structure into clear code, so the student does not have to think about it.

- A real-world example. Though tutorials explain each function and templates illustrate the structure of the experiment, the student may still have problems. The implementation of a real-world experiment will serve as a reference when these problems occur. In many cases, the student can simply modify a few lines of the sample experiment to make his own experiment.

Library

Writing a program for a TPTA experiment usually involves writing code to do the following:

- Storing data and reading it back. Students often come up with their own file formats.

- Generating signals for controlling the indentor positions. Most of the time, the signal involves ramping in, vibrating sinusoidally, and then ramping back out, as shown in Figure 2-2.

- Provide instructions or feedback to subjects. The subject receives such information through a computer display, so the student must implement a simple user interface.
• Manipulate data and experimental parameters. Experiments usually involve randomizing certain parameters and permuting or lengthening arrays.

• Control the indentor positions.

• Control the micropositioner.

The library consists of six modules which address each of these areas. A complete list and description of each function is presented in Appendix A.

**Experiment Template**

The structure of most tactile perception experiments follows a similar pattern. An experiment consists of a set of tests, with a test usually corresponding to one subject. Each test consists of several runs, or groups of trials. The number of trials and their parameters may be specified in advance by reading from a file, or set dynamically based on the results of previous trials. Each trial involves presenting a stimulus of some sort while recording the position and force of the indentors over time. The trial may also involve asking the subject a question, and saving his response. The results of each trial and its parameters are logged in files. The structure of a typical test is shown by the psuedo-code in Figure 3-1. The actual template contains a concrete implementation of the algorithm described by the psuedo-code. To run an experiment, the student would run such code for each subject.

```java
  do {
    run_parameters = load_parameters(test_parameters.file)
    trial_parameters = initialize_trial_parameters(run_parameters)
    do {
      trial_results = execute_trial(trial_parameters)
      save_to_file(trial_results)
      trial_parameters = update_trial_parameters(trial_parameters)
      run_results = update_run_results(trial_results, run_results)
    } while (continue_run(trial_results, run_results))
  } while (continue_test(run_results, run_parameters))
```

*Figure 3-1: Psuedo-Code Describing Test Procedure Template*
Real-World Example

The example provided is an experiment for measuring the 2-point error of localization of touch at a particular body site. The code is presented in two forms. A stand-alone, runnable version of the experiment is available for the student so he can step through the execution of the program and see the physical action of the TPTA. The template also contains the example code in the form of comments. The student is able to easily reference this code as he develops his own programs.

The benefit of 2-point error of localization experiment is that it is conceptually simple, so the code is not obfuscated with complicated routines. However, it has enough complexity to demonstrate use of the library modules and some of the subtler points of using the system. For instance, the experiment shows the student that signals for controlling the stimulators must be of the same length, so the shorter signal must be lengthened. Such points are not obvious, and students could spend significant time figuring them out.

3.3 Implementation

Several possible implementations of the design described are possible. The library could have been written in object-oriented LISP, for instance. We chose to implement the library as a set of functions, called from Matlab. This section describes the important implementation issues and explains why a functional, Matlab implementation was selected.

3.3.1 Implementation Considerations

In implementing the system, we took into account the following items:

- Speed. The system should be fast enough to generate very long stimulus signals at the same rate as they are required by the digital acquisition card. Though speed is rarely a problem with compiled languages, interpreted languages have little optimization and speed could therefore be an issue.
• Memory. When very long stimuli need to be presented, the computer may have very limited physical memory available. If the memory system needs to swap data to disk, the system performance will be unacceptably slow.

• Portability. The computer running the TPTA may be upgraded in the future. Changing the computer should not cause the software to misbehave. The upgrade should be fairly simple.

• Maintenance. The system should be based on third party libraries whenever possible. This reduces the burden of bug-fixing placed on lab members. Additionally, the system will be used at remote sites, so the software should be easily updatable.

• Extensibility. When lab members wish to make additions to the library, they should be able to do so without difficulty.

• Flexibility. It should be easy to call libraries written in other languages from the system, as some experiments may required interfacing with other hardware or software.

3.3.2 Implementation Decisions

Two key decisions were made regarding implementation, keeping the considerations described above in mind. Justification for these decisions is provided below.

Matlab

The decision to implement the system as a set of functions called through Matlab was inspired largely by the Psychophysics Toolbox [1, 13]. Matlab is a gentle slope, interpreted language which does not suffer from the problems described in Section 3.1.2. Matlab also makes it easy to make graphs and plot data, which is useful in monitoring the execution of an experiment. Many students have written simple Matlab programs in their coursework. Therefore, students are usually somewhat familiar with Matlab, making it easy for them to learn. Also, the data analysis of experimental results is
usually performed in Matlab, so the student would likely have to learn it at some point anyway. Both of these factors contribute to decreased total development time.

The decision to use Matlab as a base for the system trades speed and memory for other factors. Matlab programs tend to be slower and more memory intensive than programs written in C, or even other interpreted languages like LISP. However, Matlab programs are completely portable and a Matlab library can be extended simply by placing files in the appropriate places. In other languages, libraries must be packaged correctly, which can be a burden to inexperienced programmers. Matlab also provides toolboxes such as the Data Acquisition Toolbox, which makes it unnecessary to write and maintain code to interface with the data acquisition card. Finally, Matlab provides flexibility – the user can easily call programs written in C, C++, or Java.

**Functional versus Object-Oriented**

The structure of a typical tactile perception experiment can be captured by an object oriented model, but a functional approach was chosen. Matlab enables programmers to write object-oriented code, but for inexperienced programmers, it would likely only serve as another obstacle. The student would have to learn the fundamentals of object-oriented programming to implement their experiment. The functional implementation is more intuitive for the student, as they can view the program as a step by step procedure, rather than as the interaction of several objects. Additionally, the functional approach yields programs that are faster and use less memory. However, they are slightly more difficulty to maintain.
Chapter 4

Experiment

Implementing a real-world experiment is a key part of this thesis. The experiment serves three purposes. First, it proves that the library exposes enough functionality to build experiments and uncovers the holes in the library. Second, implementing the experiment provides both qualitative and quantitative information about the effectiveness of the system in reducing total development time. Finally, the experiment serves as documentation for future students. This chapter explains the design and results of an experiment for measuring the 2-point localization error of touch.

4.1 Introduction

4.1.1 Purpose

The 2-point localization error of touch is a measure of how accurately a subject can identify the location of a tactile stimulus. Precisely defined, it is the radius of the largest circle centered at a point of tactile stimulation inside which a subject stimulated at any point perceives it as identical to being stimulated at the center point. The purpose of this experiment is to measure the localization error on an area of the hand.
4.1.2 Prior Work

The classical method of measuring the 2-point localization error of touch uses the method of limits. The subject is presented with a tactile stimulus (by an indentor) at a reference point. He is then presented an identical stimulus, at a set distance away from the reference. The subject must identify whether he perceived the two stimuli as occurring at the same point or different points. This step is repeated several times, decreasing the distance between the two points at each iteration. The localization error is reported as the distance where the subject first responds that he felt he was stimulated at a single point. The progression may be presented several times, with the final measure for the localization error being the average of each run. Weinstein made measurements using the classical method at several body sites for many subjects[15]. The measured values vary greatly at different body sites and are significantly different for males and females.

Two other methods have been used to measure localization error. In the first method, an experimenter stimulates the skin with an indentor at a reference point, whose location is recorded. The subject is then asked to identify the stimulation point using an indentor. The localization error is the distance between the reference point and the point identified by the subject, possibly averaged over several trials. Results using this method have not been published to date because the method is typically only used to teach students about psychophysics. A second method of measuring localization error uses a signal detection approach. In each trial, a reference stimulus is presented, and followed shortly thereafter by a stimulus either to left or right of the reference by a specified distance. The subject must respond “left” or “right”. The distance between the two points is altered randomly in each run. The localization error is the distance at which the subject is able to respond correctly in 75% of the trials. Such an experiment was performed by Loomis[10], with the measured localization errors being a fraction of those reported by Weinstein at four body sites.

The experiments which used these methods, however, are crude and imprecise. In the experiments that involved presenting stimuli consisting of indenting the skin,
the depth of indentation was difficult to control. Slight differences in indentation depth may have caused a systematic difference in the subject’s perception of stimuli at different points and his perception of stimuli at the same point. The time between the two stimuli was also difficult to control precisely. The measured localization error is likely to increase as more time passes between stimuli. Each of these problems is likely to affect the results of the overall experiment.

4.2 Methods

The experiment takes a signal detection approach to measuring the localization error of touch on the hand. Two indentors are initially positioned so that they come in contact with the skin on a line from the center of the base of the palm to the center of the base of the thumb, as illustrated in Figure 4-1. The subject participates in six runs of 40 trials each. The separation distance between the indentors during each run is held constant, but selected randomly from the set \{1.59mm, 5.59mm, 9.59mm, 13.59mm, 17.59mm, 21.59mm\}. On each trial, one of the two indentors is chosen randomly to present a reference stimulus. After 0.1 seconds, a second stimulus is presented by a randomly selected indentor. The subject must determine whether the two stimuli occurred at the same or different points. He is not able to see the indentors during the test. The entire test takes approximately forty minutes.

4.2.1 Subjects

Seven subjects participated in the experiment: four males and three females. The subjects ranged in age from 19 to 43 years, and were taken from the general MIT community. Subjects were asked if they had any problems with their sense of touch or major injuries to their hands. No subject reported any such problems.
4.2.2 Apparatus

The apparatus used was a modified version of the TPTA described in Section 2.1. The micropositioner was not used because parts for mounting the stimulators had not been completed at the time. In place of the micropositioner, a manually controlled boom stand was used to control the position of the stimulators. The separation between the points of stimulation was controlled by a manual positioning stage with a digital micrometer. Cylindrical, Delrin tips with 0.5 mm diameter were attached to the end of the indentors using wax. The tips are the only part that come in contact with the skin. Figure 4-2 shows a schematic of the apparatus.

4.2.3 Subject Preparation

When a subject arrived, he was first asked to wash and dry his hands. The subject then filled out consent forms and provided biographical data and information on any injuries or problems related to his sense of touch. Next, he participated in a few trials where he could watch the stimulators in action to make sure he understood the experiment. The experimenter made sure the subject responded correctly in five consecutive trials before beginning the test. At least 15 minutes passed before the
Figure 4-2: Experimental Apparatus

actual experiment began, so the subject’s body could come into equilibrium with the room’s temperature and relative humidity. Room temperature was observed to be between 70 and 74 degrees Fahrenheit. Relative humidity was generally less than 20% and could not be measured accurately, but reached as high as 43% for Subject 4.

4.2.4 Stimuli

On each trial, the indentors were initially positioned so that they would present stimuli in a direction normal to the skin. To begin, the indentors were moved towards the skin until an opposing force of 100.0 mN was measured. This position is considered the "zero position" for each indentor and all the position measurements are in reference to it. The indentor that will be presenting the first stimulus (reference indentor) holds at this position for 0.1 seconds, while the other indentor moves 2.0 mm away from the zero position. Then, the reference stimulator ramps into a depth randomly selected from the range 0.64-0.80 mm at a rate of 1 mm/second. It holds at this position for a 0.15 seconds and then ramps out to the position 2.0 mm away from the skin at a 1.0 mm/second rate. After 0.1 seconds, one of the indentors selected at random ramps in and out in similar fashion. The stimulus is plotted in Figure 4-3.
Figure 4-3: The stimulus consists of two separate stimuli, each of which may be presented by either of the two indentors, selected at random.

![Graph showing the Two Stimuli with time and indentation position.](image)

4.3 Results

For each of the seven subjects, the percentage of correct responses in a run was calculated. The probability of a correct response on each trial was assumed to follow a Bernoulli distribution, and trials were independent. This assumption was used to calculate the 95% confidence intervals for each data point. The results of these calculations are plotted with the corresponding error bars for each point in Figure 4-4.

Localization errors were then extracted for each subject by interpolating lines between neighboring points, as plotted in Figure 4-4. The localization error was reported as the largest x-value of the points where the interpolated lines crossed a horizontal line indicating 75% correct. The extracted values are presented in Table 4.1. For subjects 1 and 4, the localization errors appear to be too large to be measured using this experimental design.
Figure 4-4: The results for each of the seven subjects. The plots show the percentage of correct responses at each stimulator separation distance. Error bars represent the 95% confidence intervals.
Table 4.1: Extracted localization errors with a lower bound on their average, in millimeters, for the seven subjects. Because no data could be extracted for Subjects 1 and 4, the localization error was assumed to be larger than the maximum separation distance used in our test.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Extracted Localization Error (mm)</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject 1</td>
<td>&gt; 21.59</td>
<td>F</td>
</tr>
<tr>
<td>Subject 2</td>
<td>9.6</td>
<td>M</td>
</tr>
<tr>
<td>Subject 3</td>
<td>14.8</td>
<td>F</td>
</tr>
<tr>
<td>Subject 4</td>
<td>&gt; 21.59</td>
<td>M</td>
</tr>
<tr>
<td>Subject 5</td>
<td>16.7</td>
<td>M</td>
</tr>
<tr>
<td>Subject 6</td>
<td>18.6</td>
<td>F</td>
</tr>
<tr>
<td>Subject 7</td>
<td>11.6</td>
<td>M</td>
</tr>
<tr>
<td>Average</td>
<td>&gt; 16.3</td>
<td>-</td>
</tr>
<tr>
<td>Average (males)</td>
<td>&gt; 14.9</td>
<td>-</td>
</tr>
<tr>
<td>Average (females)</td>
<td>&gt; 18.3</td>
<td>-</td>
</tr>
</tbody>
</table>

4.4 Discussion

The results of the experiment are somewhat inconclusive. The lower bound on the averages that were calculated, 14.9 mm and 18.3 mm, is significantly larger than the 5 mm and 7 mm averages reported by Weinstein for males and females respectively[15]. The source of the disparity is not clear. One possible explanation is that the two subjects for which a localization error could not be extracted actually did not understand the experiment correctly. Ignoring these two subjects, the average localization errors are 12.6 mm and 16.7 mm for males and females, which is still significantly larger than Weinstein’s results, but could be the result of random variation. Other possible reasons the results do not match previous results include low stimulus intensity, high variability in relative humidity, and the uncertainties in Weinstein’s procedure described in Section 4.1.2
Chapter 5

Discussion

5.1 Evaluation

The design of the system was evaluated primarily through the implementation of the real-world experiment. The experience provided information about the overall effectiveness of the system. Based on this information, some strengths and weaknesses of the design were identified.

5.1.1 Experiment Implementation Experience

Though the results of the 2-point localization error experiment were inconclusive, implementing the experiment provided valuable feedback about the system. The large scale goal of the system was to reduce the total time an undergraduate student with limited programming experience spends in writing code. As a result, the student would be able to concentrate on designing an experiment. The experience indicates these goals were met.

The design and implementation of the experiment was completed in under three months of part-time work. This time period included the entire experiment development process: the experiment was conceived, designed, implemented, and iteratively improved. The programming work was performed by the author, a fairly experienced programmer. The time required compares favorably with the several months re-
searchers with similar programming experience have spent implementing comparable experiments on similar apparatus in the past. The time savings for experienced programmers is expected to translate to a time savings for inexperienced programmers as well.

The amount of code that was written in the experiment was small. Excluding comments and blank space, the entire experiment program required writing approximately 175 lines of code. Approximately 45 of these lines were provided by the experiment template. The library was called roughly 30 times. If the library functions that were called had to be re-implemented, an additional 300 lines of code would be required. In comparison, a slightly more complicated experiment developed in C++ for a similar apparatus required writing 1300 lines of code. The object-oriented nature of C++ and the manual memory management was the main source of this disparity. Matlab also has a great deal of functionality built in, such as user interface components and matrix operations, that had to be re-implemented in the C++ code. Programming style could have affected these numbers, as well. In any case, the numbers suggest that the Matlab code is more concise than C++ code, and therefore probably more clear.

The fraction of total time spent programming while developing the experiment did not seem to be substantial. Though amount of time spent on each task was not recorded, programming certainly was not a burden. The bulk of the programming work was finished quickly, with minor additions made to it as the experiment design was iteratively improved. The tasks which seemed to take the most time were tweaking the experimental parameters which determine the tactile stimuli and finding ways to make human subjects comfortable during the experiment. In this sense, software development was not the controlling factor in the total time required to develop experiments.

5.1.2 Strengths

Chapter 3 described the potential strengths of the system, which were realized for the most part. The high-level, interpreted nature of Matlab enabled rapid develop-
ment of the experiment program of Chapter 4. The library, implemented as a set of functions called directly from Matlab, includes much of the functionality required for the experiment and is easily extensible. The experiment template provides a good starting point for the experiment. Overall, most of the expected advantages of the systems were observed.

Additional strengths were uncovered which were not expected. The first strength is the simplicity of the code. In the entire program for the experiment of Chapter 4, the most complicated programming construct used is a simple, conditional loop, and even this is provided by the experiment template (see Figure 3-1). The new code contained, for the most part, only “if-else” statements and a few function declarations. This simplicity may indicate that students need only learn the most basic elements of Matlab programming to implement their experiments, which would further reduce the student’s ramp-up time. Though some experiments will require writing more complicated programs, many experiments are expected to be at this same level of complexity.

The debugging process using the system is surprisingly effective. One of the key benefits of Matlab is that one can modify a program while it is running. This feature was particularly useful in developing the experiment of Chapter 4. The development required tweaking many experimental parameters, to ensure subjects would be able to both feel the stimulus presented and not be hurt by it. These parameters are normally read from a file at the beginning of the experiment, and changing them requires restarting the experiment. Restarting each time would have been extremely inefficient, because the experiment involved significant set up at the beginning. In Matlab, the programmer is able to enter in new values for the experimental parameters just before they are used, enabling him to skip any initial set up without writing additional code. This feature was also useful when improving the appearance of user interface items.

A significant benefit to the system which had not been considered was the number of people around campus that are able to help resolve programming errors when working with Matlab. Most students at MIT have worked with Matlab for coursework,
and are able to help find bugs in code. Such students can also point out built-in features of Matlab that may have been overlooked. Even students that may never have worked with Matlab find the syntax simple enough to help find algorithmic errors. Using other students as resources made the experiment programs more concise and quicker to write. Similar reports are expected from future students.

5.1.3 Weaknesses

Two major weaknesses were discovered in the design of the system. The first weakness in the design is the large amount of memory used by the system. Memory was expected to be only a minor concern. Though the computer did not run out of memory during the experiments, Matlab often crashed with out of memory errors immediately following an experiment. The problems were eliminated by making explicit calls to the Matlab memory compression functions. The system was designed with the goal of completely automatic memory management, but this goal was not achieved.

The lack of compiler optimizations in Matlab also posed an unexpected problem. Generally, compiled programs are optimized automatically such that slow operations such as painting to the screen, are executed in parallel with other operations. However, such optimizations are not possible in Matlab because of its interpreted nature. In Matlab, multiple operations cannot execute at once; an operation must be completed before the next operation can begin execution. At times, operations such as writing to the screen were noticeably slow. However, in the experiment developed, the subjects did not notice these delays because they were focusing on other items at the time. In future experiments, it may not be possible to hide the delay from the subjects.
5.2 Implications

5.2.1 Tactile Research

This thesis will have its greatest impact on the field of tactile research. The TPTA can perform a range of experiments, many of which have never been attempted. While the work of this thesis is not an absolute requirement to performing such experiments, it should enable such experiments to be completed faster. Performing experiments that may have been ignored previously because the effort required was too large or the experiments were too difficult to be developed by students may now be feasible. Already, two experiments which utilize the software developed in this thesis are being planned. The results of these experiments could contribute significantly to our understanding of the mechanical properties of touch and the way the brain processes tactile signals. The experiments may also aid in the development of novel tactile devices.

5.2.2 Software Development

Though the system designed in this thesis is aimed at reducing programming time for students developing tactile perception experiments, it also has implications on the broader problem of developing any type of software with inexperienced programmers. Though the system needs to be used by more students, the work thus far has provided insights into this problem. The following two recommendations are made for addressing the general problem.

Interpreters

The work of this thesis illustrates the benefit of using interpreted languages to develop software. For many applications, however, an interpreted language does not provide the necessary performance. Programmers then have to revert to using compiled languages like C. However, this requirement means only that the end product of his work needs to be a program compiled into machine code. In the development
process, though, he can still use an interpreter for his language and gain many of the benefits of using an interpreted language. Interpreters for most popular languages exist, but they are used largely to gain portability. Instead, these interpreters should be used during development to increase productivity.

Converters

The experience using the system underscores the need for programming language converters. A key strength of the system was that the developer was able to get help from others in debugging code, because other students were familiar with the language used. When colleagues are not familiar with a language, this advantage can still be attained using a converter. A programmer could convert his code to a language which his colleagues do know, and use this to get help debugging his algorithms. The colleague may even be able to fix the converted code, which could then be translated back to its original language. Converters for many languages already exist, but do not always produce clear code. More attention should be given to developing quality converters because programmers, especially inexperienced ones, could achieve great benefits by using them.
Chapter 6

Conclusion

6.1 Summary

This thesis presented the design and development of a system to reduce the amount of time students spend programming in creating tactile perception experiments. The following five steps were taken:

- Instead of forcing students to program in C, a system was developed so they could write program using Matlab, a gentle slope, interpreted language.

- A library was built to capture functionality common to most experiments. Students can reuse this code rather than re-implement it.

- Tutorials and simple examples were developed. These make students more likely to reuse code, because they do not have to read complicated software specifications to figure out how to use the library.

- A real-world experiment was implemented. This example demonstrates many of the nuances of programming for the system, that are not covered in tutorials.

- An experiment template was created. The template illustrates how to structure an experiment, and break the development into smaller, discrete tasks.

The effectiveness of the system was evaluated using the real-world example as a case-study. The experiment was developed in a relatively short amount of time, and
the code was a fraction of the size of a previous experiment program for a similar apparatus written in C. Use by future students will determine if the library is truly as effective as our experience suggests. The work suggests that interpreters and converters should be more widely used when writing software.

6.2 Future Work

6.2.1 Verification

A substantial problem remaining with the system built is that students may still produce programs which have fundamental flaws. Though the programs seem to behave correctly in demonstrations, there may be subtle errors in the code. The results of experiments would then be flawed. While it is impossible to automatically verify that a program is correct, it is useful in this case to verify that the program behaved correctly during the experiment.

Dynamic invariant detection[6], a technique that analyzes program traces to determine properties that held throughout execution, may be useful in the verification task. The type of properties extracted may be temporal constraints or relationships between variables. For instance, a dynamic invariant detector could state that "Event A" always occurs before "Event B" or that a variable is always within a specific range. The properties extracted are those that held for one particular execution; they are likely, but not guaranteed, to hold for later executions.

The nature of tactile perception experiments make the properties extracted through dynamic invariant detection especially useful. In the real-world experiment described in Chapter 4, for example, it would have been useful to verify that the indentation depth was always between 0.64 and 0.80 mm, and that the indentors were moved into contact with the skin before a stimulus was presented. If either of these properties did not hold, the invariant detector would report that they did not hold and the experimenter would know that he needs to reinterpret his data. While such properties should be checked explicitly in the code, students may forget to do so or implement
the checks incorrectly. An invariant detector provides many of these checks automatically, and could be of great value.

6.2.2 Simulator

While the focus of this thesis was on reducing the total time a particular student takes to complete an experiment, it would also be useful to increase the number of students who can simultaneously develop programs for the TPTA. Currently, only a single student can be developing a program at any one time, because there is only one physical device. Code cannot be tested out without access to the physical device. Students may be able to share the device, but this can be difficult to coordinate.

A software-based simulator of the TPTA would allow multiple students to develop programs at once. The simulator would allow students to control every aspect of the TPTA. Additionally, they could perform experiments on “virtual subjects” that have pre-specified mechanical models. Using such a system, the students would be able to work out the algorithms for their experiments without physically using the TPTA. The actual TPTA would only need to be used by students to fine-tune experimental parameters and execute actual experiments. The number of students running an experiment at any given time would still be limited, but an unlimited number of students would be able to develop experiments.
Appendix A

Appendix A: Library Overview

This appendix provides a brief overview of how to begin building an experiment using Matlab and the library. The first section presents the functions in the library, and a brief description of what they do. The following sections illustrate how to read and write files, control the micropositioner and indentors, and communicate with a subject. These are the most fundamental tasks in an experiment, and can be performed using the library and built in Matlab commands. After reading this appendix, you should know enough to begin implementing an experiment.

A.1 Function List

The library functions are organized into six directories, which are described below. A brief description of each function is provided, but details are left for the help notes. To view the help notes for a function, type “help functionname” in Matlab.

- daq: Changing indentor positions.
  - findskin: Move an indentor into contact with the skin.
  - findskin_simultaneous: Move both indentors into contact with the skin, at the same time.
  - getvalue: Read a single value from a single channel of the data acquisition card.
- run_daq: Begin inputting and outputting signals that have previously been queued.

- setup_daq: Initialize the data acquisition card to record force and position, while controlling position.

- writevalue: Write a single value to a single output channel of the data acquisition card.

- fileio: Reading and writing files.
  - load_conditions: Load a matrix of numbers stored as text from a file into a struct.
  - save_struct: Save the values of a struct into a Matlab script file (.m)

- positioning: Controlling the micropositioner.
  - move_axis: Move a single axis of the micropositioner.
  - read_axis: Read the current position of a single axis of the micropositioner.

- signals: Generating typical signals for the position waves.
  - holdwave: Generate a wave that holds at a constant value for a specified number of samples.
  - ramp: Generate a signal that linearly ramps from one value to another, over a specified number of samples.
  - sinewave: Generate a sinusoidal wave.
  - standard: Generate the typical signal in TPTA experiments (ramp in, hold, vibrate sinusoidally, hold, ramp out).

- ui: User interface components for communicating with the subject or aiding the experimenter.
  - msg_subject: Send a message to the subject, on his screen.
  - close_msg: Close a message previously written to the subject’s screen.
setup: Graphically control the position of the indentors.

diowait: Get a subject’s response to a question from a keypad connected to the computer.

setup_stimulator_position: Iteratively helps the experimenter adjust the positioning of the system so that the indentors don’t have to move too far to find the skin.

timed_question: Get a subject’s response to a question from keypad, but only give him a limited amount of time to answer.

• util: General utility functions.

  interpolation_intersection: Linearly interpolate lines between neighboring x-y points in a plot, and find the points where the lines intersect a given value.

  pad: Given two arrays, lengthen the shorter one. The last value in the shorter array is filled into the additional length.

  randomize: Shuffle the rows of a matrix.

A.2 Reading and Writing Files

Any experiment you build will certainly require reading or writing files. You should try to write your experiment so that as many experimental parameters as possible are read from a file. This makes it easier to vary these parameters later on. You should also try to store enough information to disk during your experiment so that you can later go back and look at this data and have a pretty good idea of what happened. For instance, in an experiment where you present a sine wave to a subject using an indentor selected at random, you should read all the parameters that specify the sine wave (amplitude, frequency, etc) from a file. The parameters as well as some indicator of which indentor presented the stimulus should also be stored in a file for later viewing.
There are several possible formats you can store files in. Which format you select will depend on what type of data you want to store, whether or not you want the data to be readable by a human, and how fast you need to read and write it. In the course of writing an experiment, you will likely have to make different tradeoffs at several points. For example, you probably want to specify experimental parameters that are read at the beginning of an experiment in human readable form, so you can modify them quickly later on. However, if you need to read new parameters from disk on every trial, you will need the reads to be fast.

If all of the data you are storing is numerical, the best method of storing it is to place it into a matrix or array, and write this structure directly to disk. The data will be stored as a matrix in a text file, with each line corresponding to a row, and columns separated by spaces. Matlab has built in commands to read data specified this way into a matrix. The following code demonstrates how to do this. You can view the file created by double clicking on it in the current directory pane of Matlab.

```matlab
a = [1 2 3; 4 5 6] % initialize a matrix
save -ascii 'mmyfile.txt' a % write the data to 'mmyfile.txt'
b = load('mmyfile.txt') % read back the data and write it into variable b
```

An alternative to reading matrix data stored in a text file into a matrix is to read one row at a time into a struct. Structs are easier to work with than matrices and arrays, because you can refer to values by their names, rather than indices. Here's an example which reads the second row of the matrix “a” from the file created in the previous example. At the end, variables b.f, b.g, and b.h will have values 4, 5, and 6 respectively.

```matlab
b = load_conditions('mmyfile.txt', 2, 'f', 'g', 'h')
```

Another format you can save files in is a binary .mat format, which Matlab uses to store data. The .mat files are not human readable, but you can store any type of data you want. The names of the variables are also saved, and reloaded when you read the file. This code stores data in a .mat file, and reads it back.

```matlab
a = [1 2 3; 4 5 6] % initialize a matrix
b = 'mymystring' % initialize a string
```
c.x = 7;  % initialize a struct, with a field called x
save 'myfile.mat' a b c;  % save all the variables
clear;  % erase all variables from memory
load('myfile.mat');  % variables a, b, and c are restored exactly as before

In your programs, you will likely come to a point where you have a struct filled with experimental parameters that you would like to store to disk. You can do this using the .mat file described above, but if you want the data to be human readable, you can instead save the fields of the struct in a Matlab script file (.m). Each variable will have one line in the file, similar to “variable.name = variable.value”. This function will let you store strings if they are fields in the struct, but won’t let you store array fields. Type the following code to try this out to store a struct to a file, and read back the fields.

a.x = 3;  % initialize a struct, with one field called x
a.y = 5;  % add field y to the struct
save struct(a, 'myfile.m');  % save the struct to file “myfile.m”

In your programs, you will likely come to a point where you have a struct filled with experimental parameters that you would like to store to disk. You can do this using the .mat file described above, but if you want the data to be human readable, you can instead save the fields of the struct in a Matlab script file (.m). Each variable will have one line in the file, similar to “variable.name = variable.value”. This function will let you store strings if they are fields in the struct, but won’t let you store array fields. Type the following code to try this out to store a struct to a file, and read back the fields.

a.x = 3;  % initialize a struct, with one field called x
a.y = 5;  % add field y to the struct
save struct(a, 'myfile.m');  % save the struct to file “myfile.m”

clear;  % erase all the data
run(myfile);  % reload the fields of the struct

A.3 Controlling the Micropositioner

The micropositioner lets you control automatically and precisely the positioning of the stimulators using the four stepper motors. The nuDrive system which is on the TPTA cart drives the micropositioner based on signals from the motion card. Each motor adjusts one of the four axes (x, y, θ, z). You should connect the the cable from the nuDrive system marked ‘1’ to the x motor, ‘2’ to the y motor, ‘3’ to the θ motor and ‘4’ to the z motor. If everyone keeps this convention, you will not have to adjust the cables every time a different experiment is run.

In order to move the micropositioner along an axis, you will use the move_axis function. To read the current position of an axis, you will use the read_axis function. The format of your calls will be:
move_axis(axis_number, steps_per_revolution, timeout, velocity, acceleration, position)
position = read_axis(axis_number)

The meaning of each parameters is as follows:

- axis_number. An integer between 1 and 4, inclusive, which indicates the axis to move the stimulators along.

- steps_per_revolution. The number of steps you want the motors to take to complete one revolution. The more precision you want in the control of the position, the larger the value you should enter here. A value of 100 is appropriate most of the time.

- timeout. The maximum time you want to wait for the position to be reached. Otherwise, give up.

- velocity. The maximum speed you want the motors to move at, in revolutions per minute. A typical value is 200.

- acceleration. The acceleration of the motors until it reaches the maximum velocity specified. Acceleration is specified in steps per second squared. A typical value is 100.

- Position. The absolute position you want the motor to move to. The position of the motor at power-up is assumed to be the zero position.

Here’s an example that moves the micropositioner along the first axis, and reads out the position.

move_axis(1, 300, 60, 10, 10000) % move 10000 steps on axis 1
position = read_axis(1) % read axis 1 position
move_axis(1, 300, 60, 10, 0) % move back to original position
position = read_axis(1) % read axis 1 position
A.4 Controlling the Indentors

The system uses the Data Acquisition Toolbox to control the indentors, though this is invisible to the user. Most of the time, the library provided has all the functionality you will need, but if you need to do something out of the ordinary, you can use the toolbox directly. A tutorial on the toolbox is available through Matlab help. This section explains how to read and write the indenter positions and forces at a single instant of time, move indentors into contact with the skin, and output a signal controlling the indenter position over time while recording its force and position.

The simplest thing you will want to do with the indentors is read or write a single value indicating the force or position of the indentors. To do so, you will first need to identify which channels you will be reading and writing from. Looking at the BNC connector block, suppose you have the connector labeled ACH0 connected to “Force Out” on one of the Aurora’s and the connected labeled DAC0 connected to “Length In.” You can read and write to these channels as follows:

\[
\text{writevalue}(0, 5) \text{ \% write 5 volts to the position channel}
\]

\[
\text{force} = \text{getvalue}(0) \text{ \% read the voltage of the force channel}
\]

The numbers you read and write are voltages. You need to convert them to correct values using the appropriate scale factors determined in calibration (approximately .5 mm/V for position, .05 N/V for force).

In most experiments, you will need to move the indentors into contact with the subject’s skin. The findskin function does this for you. The function slowly changes the position of the indenter until a specific force is measured. A call to findskin takes the following format:

\[
\text{findskin(output\_position\_channel, input\_force\_channel, force\_threshold\_voltage, up\_position\_voltage, max\_position\_voltage)}
\]

Assume the channels are connected as before, and the scale factors are .5 mm/V and .05 N/V. Using the following command, the indentors will begin at a position 2 millimeters above the zero position, and start moving down slowly until a force of .1 N is measured. This procedure will be repeated two times and the average over the
three iterations of the position where the required force was measured is returned. If the required force is not measured by the time the indentor reaches a position 3 millimeters below the zero position in any of the iterations, the function stops and returns a value that is larger than 1000.

\[
\text{position} = \text{findskin}(0,0,.1/.05,-2/.5,3/.5)
\]

A large component of your experiment will likely involve presenting a tactile stimulus over time. The above functions are not sufficient, because you need to specify the position of the indentors as a function of time, and measure the position and force over time as well. To do so, you will need to take the following steps.

1. Specify the channels you need to read and write from, as well as the name of the file you want to store data in. So, you would enter a command like this:

   \[
   \text{[ao, ai] = setup\_daq([0 1], [0 1 2 3], 'myfile\_daq')}
   \]

   This function will set up the system to write to channels 0 and 1, read from channels 0, 1, 2, and 3 and store the values read into a file called myfile\_daq. The values returned are objects which represent conceptually the analog output and analog input components. The analog output and input objects are all you need the rest of the way. You use the setup\_daq function only once in an experiment.

2. Specify the signal to output. Assume you want to present two stimuli stored in arrays signal1 and signal2, at a rate of 1000 samples per second while recording all other values at a rate of 10000 samples per second. You would execute the following commands:

   \[
   \text{set(ao, 'SampleRate', 1000); \% set output sample rate}
   \text{set(ai, 'SampleRate', 10000); \% set input sample rate}
   \text{putdata(ao, [signal1' signal2']) \% specify the signals in the columns of a matrix}
   \]

   An important thing to notice here is that signal1 and signal2 must be the same length, because the two columns in the matrix must be the same length. If for some reason they are not, you can use the function “pad” to lengthen them. Type “help pad” in Matlab to see how to use it.
3. Start the output. This requires executing only this command: `run_daq(ao, ai, 60) % do it, but if it takes longer than 60 seconds, stop.

4. View the position and force data. You will use the following commands.
   
   \[
   d = \text{daqread('myfile.daq')}; \quad \text{% read data into a matrix}
   \]
   
   \[
   \text{channel0} = d(:,1); \quad \text{% extract the channel 0 data (1st column in d)}
   \]
   
   \[
   \text{plot(channel0);} \quad \text{% plot the data}
   \]

A.5 Communicating with Subjects

If your experiment is a perception experiment, you will need to ask subjects questions and record his responses. This section explains how to do so.

A.5.1 Getting Input

The keyboard and mouse will be used by the experimenter during the experiments, so the subject will have to use a keypad like device to provide input. This device must be connected to the digital input and output ports on the data acquisition card. A two-button keypad is currently connected. You can wait for a subject to push a button by using the command

\[
\text{button\_number} = \text{diowait}
\]

This command will return the number of the button pushed. The function only works for a 2-input keypad though. If you need more buttons, you will have to modify it slightly.

A.5.2 Messaging Subjects

The subject will be looking at the second monitor attached to the computer, so you need to write messages to that screen. You will also have to close messages once they are no longer needed. This code is an example of sending a message to a subject, waiting for a response, and then closing the window.

\[
\text{handle} = \text{msg\_subject('my message')};
\]
d = diowait;

close_msg(handle);

The above code gives the subject an unlimited amount of time to answer a question. If you need to give the subject only a short time to answer for some reason, you can write code like this, which gives the subject five seconds to respond.

response = timed_question('string', 'my message', 'timeout', '5')
Bibliography


