REAL-TIME CONTROL OF A MULTI-FINGERED ROBOT HAND

USING EMG SIGNALS

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Luenin Adriel Barrios

Summer 2010
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the
Thesis of Luenin Adriel Barrios:

Real-Time Control of a Multi-Fingered Robot Hand using EMG Signals

Marko Vuskovic, Chair
Department of Computer Science

Roman Swiniarski
Department of Computer Science

Christopher Paolini
Computational Science Research Center

6-29-10
Approval Date
Copyright © 2010
by
Luenin Adriel Barrios
All Rights Reserved
DEDICATION

This thesis is dedicated to my father and mother. I did this for you.
ABSTRACT OF THE THESIS

Real-Time Control of a Multi-Fingered Robot Hand using EMG Signals
by
Luenin Adriel Barrios
Master of Science in Computer Science
San Diego State University, 2010

The design and development of a real-time computer program for controlling the anthropomorphic six degrees of freedom robotic hand developed in the Robotics Laboratory at San Diego State University is presented. The program allows the hand to be controlled either manually or via EMG signals received from electrodes attached to the user’s forearm. The robot hand’s joints can be operated independently or ordered to perform synergetic grasp motions. In particular, this research continues the previous work conducted at the lab of using feature extraction and classification of EMG signals in real-time to determine the user’s hand position and command the robot hand to achieve the same motion. The grasp modes studied include: cylindrical, spherical, point, and lateral hand positions, and a first-order rational approximation function is used to achieve the synergetic motions of the robot hand. The results demonstrate that feature extraction and classification of EMG signals serve as an adequate method for controlling the motions of a robot hand and that such control systems may be utilized in areas such as prostheses.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td></td>
<td>xiii</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>MULTI-FINGERED ROBOT HANDS AND PROSTHESIS</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>REAL-TIME MEASUREMENT OF PREHENSILE EMG SIGNALS</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>EMG Electrodes</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>EMG Amplifier Issues</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>EMG Amplifier Design</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>EMG Capture Program</td>
<td>14</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Main Parameter</td>
<td>15</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Play Mode</td>
<td>15</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Voltage Scale</td>
<td>15</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Time and Readout</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>REAL-TIME FEATURE EXTRACTION AND CLASSIFICATION OF PREHENSILE EMG SIGNALS</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>EMG Signal Collection</td>
<td>16</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Properties of the Electrodes</td>
<td>16</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Human Forearm Anatomy</td>
<td>17</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Electrode Placement</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>EMG Signal Processing</td>
<td>20</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Onset of Movement</td>
<td>21</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Bonato Method</td>
<td>21</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Feature Extraction</td>
<td>22</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Waveform Length</td>
<td>22</td>
</tr>
</tbody>
</table>
4.2.5 Spectral Moments ................................................................. 22
4.2.6 Spectral Moment Parameter Optimization ......................... 24
4.2.7 Feature Vector Transformations ....................................... 26
4.2.8 Parameter Results ............................................................. 27
4.3 Real-Time EMG Classifier Program ..................................... 28
  4.3.1 Program Files ................................................................. 28
  4.3.2 Program Features ........................................................... 29
5  ROBOT HAND MOTION CONTROL SYSTEM .......................... 32
  5.1 Overall System Description ............................................... 32
    5.1.1 SDSU Robot Hand ....................................................... 32
    5.1.2 Joint Controllers ....................................................... 34
    5.1.3 Hand Actuator Model .................................................. 34
      5.1.3.1 PD Controller ...................................................... 39
      5.1.3.2 PID Controller .................................................... 40
    5.1.4 Servo To Go Board ..................................................... 44
      5.1.4.1 STG Hardware ..................................................... 44
      5.1.4.2 STG Software Driver ............................................ 49
    5.1.5 Signal Transition Box .................................................. 52
6  SYNERGETIC MAPPING .......................................................... 60
7  SYSTEM INTEGRATION AND CALIBRATION ............................. 65
8  EMG ROBOT HAND PROGRAM ............................................... 85
  8.1 Program Components ....................................................... 85
    8.1.1 Calibrate ................................................................. 87
    8.1.2 Absolute Motion ....................................................... 87
    8.1.3 Relative Motion ....................................................... 89
    8.1.4 Object Space ........................................................... 89
    8.1.5 Display ................................................................. 90
    8.1.6 Reset ................................................................. 90
    8.1.7 Show Menu ............................................................ 91
    8.1.8 Wait for EMG .......................................................... 91
    8.1.9 Exit ................................................................. 91
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>NI 6220 DAQ Specifications</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Tunable Parameters in Spectral Moment Calculation</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Optimized feature extraction parameters</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Specifications of the Maxon RE 016-042-05-EAA100A DC Motor</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Specifications of the 8-axis ISA Bus Servo I/O Card</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>STG board connector pin-outs</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Description of the STG connector pins</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Elements of the IoCtlBuf command structure</td>
<td>51</td>
</tr>
<tr>
<td>5.6</td>
<td>DB50 connector cable pin-out specifications</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>DB25 connector cable pin-out specifications</td>
<td>55</td>
</tr>
<tr>
<td>5.8</td>
<td>Analog ground connections for Transition box</td>
<td>56</td>
</tr>
<tr>
<td>5.9</td>
<td>Digital ground connections for Transition box</td>
<td>56</td>
</tr>
<tr>
<td>5.10</td>
<td>Complete interconnection table for transition box pins</td>
<td>57</td>
</tr>
<tr>
<td>8.1</td>
<td>Maximum and minimum joint angles of each finger</td>
<td>88</td>
</tr>
<tr>
<td>8.2</td>
<td>Source files for EMG Robot Hand and modified EMG Classifier Programs</td>
<td>94</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Otto Bock 2000 grasp pincher hand</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>The TAP Version 3 Prototype Hand</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>The SDSU robot hand shown with various grasp positions</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>EMG Amplifier Schematic</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Complete Schematic of Saksit's Amplifier and EMG Capture Program</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>The PCB of Saksit's EMG amplifier device</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Snapshot image of the EMG Capture Program GUI</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Bipolar electrodes designed by Saksit for use with EMG amplifier</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Human Forearm extensor muscles used for EMG signal capture</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Human forearm properly attached to EMG amplifier device with correct electrode placement shown</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Sample main screen display of the Real-Time EMG Classifier Program</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Overall System diagram showing the various components</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Approximated simplified diagram of a DC motor</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Reduced DC Motor Diagram</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>Final DC Motor diagram after transformations</td>
<td>36</td>
</tr>
<tr>
<td>5.5</td>
<td>Block diagram of DC motor with load applied to system</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>Reduced block diagram of Figure 5.5</td>
<td>37</td>
</tr>
<tr>
<td>5.7</td>
<td>Final simplified Motor block diagram</td>
<td>38</td>
</tr>
<tr>
<td>5.8</td>
<td>SIMULINK model of motor showing all components</td>
<td>39</td>
</tr>
<tr>
<td>5.9</td>
<td>Block diagram of PD controller</td>
<td>39</td>
</tr>
<tr>
<td>5.10</td>
<td>Block diagram of a PID Controller</td>
<td>41</td>
</tr>
<tr>
<td>5.11</td>
<td>Top Level diagram of Controller and Actuator</td>
<td>41</td>
</tr>
<tr>
<td>5.12</td>
<td>SIMULINK model of the Actuator subsystem</td>
<td>42</td>
</tr>
<tr>
<td>5.13</td>
<td>SIMULINK model of the PID Controller subsystem</td>
<td>42</td>
</tr>
<tr>
<td>5.14</td>
<td>Impact of torque disturbance on error with $t_d = 0.5$Nm</td>
<td>43</td>
</tr>
<tr>
<td>5.15</td>
<td>The 8-axis ISA Bus Servo I/O Card</td>
<td>44</td>
</tr>
<tr>
<td>5.16</td>
<td>STG board P1 and P2 pin-out specifications</td>
<td>46</td>
</tr>
</tbody>
</table>
Figure 5.17. STG board P3 and P4 pin-out specifications ........................................... 47
Figure 5.18. Transition box connection to robot hand, amplifier and STG board ................. 52
Figure 5.19. Close up image of pin connections of transition box .................................. 53
Figure 6.1. Measurement of joint angle $\theta_1$ .................................................................. 61
Figure 6.2. Aperture diameter relationship with joint angle for the approximation function. ....................................................................................................................... 62
Figure 7.1. Flow chart diagram showing processes and actions performed on SKIPPER ................................................................................................................................. 66
Figure 7.2. Flow chart diagram for processes on BRONCO .............................................. 66
Figure 7.3. Spherical grasp objects and sizes .................................................................... 67
Figure 7.4. Precision grasp objects and sizes .................................................................... 67
Figure 7.5. Cylindrical grasp objects and sizes ................................................................. 68
Figure 7.6. Lateral grasp objects and sizes ........................................................................ 68
Figure 7.7. Robot hand grasping small precision: Garbanzo bean diameter 9mm .............. 69
Figure 7.8. Precision grasp of Garbanzo bean. Object diameter 9mm ............................... 69
Figure 7.9. Robot hand grasping medium precision: Cap diameter 16mm ....................... 70
Figure 7.10. Precision grasp of Cap. Object diameter 16mm ............................................. 70
Figure 7.11. Robot hand grasping large precision: Nut diameter 28mm ................................ 71
Figure 7.12. Precision grasp of Nut. Object diameter 28mm ............................................. 71
Figure 7.13. Robot hand grasping small cylinder: Tool case diameter 28mm .................... 72
Figure 7.14. Cylindrical grasp of Tool case. Object diameter 28mm ............................... 72
Figure 7.15. Robot hand grasping medium Cylinder: Pepper shaker diameter 42mm ....... 73
Figure 7.16. Cylindrical grasp of Pepper shaker. Object diameter 42mm ......................... 73
Figure 7.17. Robot hand grasping large cylinder: Plastic bottle diameter 56mm ............... 74
Figure 7.18. Cylindrical grasp of Plastic bottle. Object diameter 56mm ............................ 74
Figure 7.19. Robot hand grasping small spherical: Plastic ball diameter 39mm ................. 75
Figure 7.20. Spherical grasp of Plastic ball. Object diameter 39mm .................................... 75
Figure 7.21. Robot hand grasping medium spherical: Racquetball diameter 60mm .......... 76
Figure 7.22. Spherical grasp of Racquetball. Object diameter 60mm ............................... 76
Figure 7.23. Robot hand grasping large spherical: Softball diameter 81mm ..................... 77
Figure 7.24. Spherical grasp of Softball. Object diameter 81mm ....................................... 77
Figure 7.25. Robot hand grasping small lateral: Block diameter 6mm .............................. 78
Figure 7.26. Lateral grasp of small block. Object diameter 6mm ....................................... 78
Figure 7.27. Robot hand grasping medium lateral: Block diameter 19mm. .......................... 79
Figure 7.28. Lateral grasp of medium block. Object diameter 19mm. .......................... 79
Figure 7.29. Robot hand grasping large lateral: Block diameter 25mm. .......................... 80
Figure 7.30. Lateral grasp of large block. Object diameter 25mm. .......................... 80
Figure 7.31. EMG Classifier and real-time test objects: a key, a cup, a tennis ball and a small cap. .......................................................... 81
Figure 7.32. Real-time Point grasp of robot hand using the EMG Classifier Program
Note the correspondence of classification with robot grasp. .............................. 83
Figure 7.33. Real-time Spherical grasp of robot hand using EMG Classifier Program
Note the correspondence of classification with robot grasp. .............................. 83
Figure 7.34. Real-time Cylindrical grasp of robot hand using EMG Classifier Program. Note the correspondence of classification with the robot grasp. .............................. 84
Figure 7.35. Real-time Lateral grasp of robot hand using EMG Classifier Program
Note the correspondence of classification with robot grasp. .............................. 84
Figure 8.1. Status message shown upon successful connection of EMG Robot client program with server program .......................... 86
Figure 8.2. EMG Robot Hand user interface with the list of available commands. .......... 86
Figure 8.3. Sample absolute motion command for joint=0 and angle = 45°. ................. 88
Figure 8.4. Sample object space command of point grasp diameter 9mm. .................. 89
Figure 8.5. Sample Display command showing all joint angles for the hand in a Point grasp mode ............................................................. 90
Figure 8.6. The Wait for EMG signal command is shown waiting for real-time data to be received from the EMG Classifier Program. .......................... 91
ACKNOWLEDGEMENTS

I would like to thank Dr. Marko Vuskovic for allowing me to conduct research in his lab and helping me to fulfill one of my dreams. In addition, his guidance and counsel in my academic career proved invaluable and I am deeply appreciative and grateful for his support.

I would also like to thank Dr. Christopher Paolini for his help and advice regarding communication between two computers. It proved extremely beneficial and allowed me to continue my work and progress. Lastly, my gratitude to Saksit Siriprayoonsak and Christopher Miller. Without the foundations they laid, this work would not be possible. It was a privilege to continue the research they began.
CHAPTER 1

INTRODUCTION

The design and development of multi-fingered robot hands and their application to prosthetics has been an area of continuous research at the Robotics Laboratory at San Diego State University (SDSU). In particular, the complexities arising out of human-machine interfacing and the responsiveness and maneuverability of the robot hand has been the primary focus of the research conducted. Previous work in biomedical engineering has demonstrated that correct acquisition and processing of EMG signals for pattern classification can be used to identify motion commands for controlling robot hands [1,2,3]. In particular, the importance of noise reduction and the successful extraction of features from the data [3,4,5,6] is a precondition for the performance and effectiveness of pattern classification.

EMG signals offer a noninvasive and effective method for obtaining the signals emitted from human subjects. These signals contain vital information regarding the motions of the arm and hand. If an efficient and precise method of extracting the features of a signal can be obtained, and the classification of the resultant data can be successfully performed, then EMG signals offer a simple but powerful way of determining the motions of the muscles of human subjects.

The goal of this project is to continue the research conducted at the Robotics Laboratory at San Diego State University regarding the EMG controlled synergetic grasp motions of the SDSU robot hand proposed by Vuskovic and Marjanski [7]. Specifically, this project aims to utilize Saksit’s EMG Amplifier device [8] together with Miller’s EMG Classifier program [9] as the method of acquiring, analyzing and classifying forearm EMG signals. The software developed in this project will then use this data for real-time control of the SDSU robot hand. Furthermore, the robot hand will be equipped with synergetic grasp motions of four modes: cylindrical, spherical, point and lateral and be able to grasp objects of various shapes and sizes belonging to those classes. The software created will allow a human user to control in real-time the grasp motions of the robot hand thus demonstrating the
capacity and effectiveness of EMG signals to drive robotic hands and their potential use in the area of prosthetics.

The material in this thesis is organized as follows. Chapter 2 provides a brief historical overview of previous and current work conducted in the area of multi-fingered robot hands and prosthetic. The chapter identifies the limitations of earlier work and presents the advantages of the six degrees of freedom robot hand used in this project.

Chapter 3 summarizes the previous work conducted by Saksit at the Robotics Laboratory regarding the effective acquisition of EMG signals from the human arm. The chapter is a compendium of the EMG Amplifier device created by Saksit which is capable of reading and storing EMG data.

Chapter 4 progresses into the actual analysis of the EMG data. The previous work by Chris Miller into feature extraction of the relevant information and the classification strategies used are discussed. The various algorithms used are briefly summarized and the final EMG Classifier program is presented.

Chapter 5 provides a detailed discussion of the PID control system used in this project for commanding the multi-fingered SDSU robot hand. It also discusses a few of the mechanics of the robot hand before delving into a detailed overview of the hardware used in this project. Specifically, the transition box created in this project for interconnection between the various devices is explained in detail.

Chapter 6 is a discussion of the mathematical concepts and formulas used in this project for achieving synergetic motion. The approximation function used for determining joint angles given grasp mode and object diameter is thoroughly expounded.

Chapter 7 presents the methodology utilized by the software created in this project to train the synergetic motions of the robot hand. Various examples of cylindrical, spherical, point and lateral grasp motions are also given. More importantly, this chapter showcases the final overall software in action and successful demonstrates the real-time user controlled motions of the SDSU robot hand.

Chapter 8 describes the specifics of the EMG Robot Hand program developed in this project. An overview of the many commands is given along with examples. The chapter concludes with a discussion of the properties and functions of the many program files used.
and created for this project and includes an in depth tutorial on how further modifications may be performed to the existing projects.

Lastly, Chapter 9 concludes and summarizes the results of this project and provides a few suggestions into future improvements and work that may refine and perfect the work previously performed.
CHAPTER 2

MULTI-FINGERED ROBOT HANDS AND PROSTHESIS

The design of multi-fingered robot hands for use as prosthesis has seen great advancements and developments over the years. The creation of such devices has been hampered by the difficulties arising in simulating the highly complex human hand. The articulation of the human hand and its corresponding anatomy is undoubtedly one of the pinnacles of the natural world. It’s prehensile and joint motion capabilities give it an element of versatility that is very difficult to replicate or model [10,11,12,13]. Furthermore, once an adequate model has been developed, the interfacing of a human subject with a robot hand has posed even greater problems that are currently the subject of intense research. These problems include: the generation of forearm muscle and tendon signals for proper simulation, their feature extraction for correct pattern classification, and lastly, their interfacing with a human-robot system for adequate real-time control. The solution to these problems has been the basis and focus of multi-fingered robot hand research and its application to the area of prosthesis.

The original prosthetic devices developed in the early years of research were essentially single DOF grippers capable of only one grasp type, namely pinch grasps [14,15]. For example, early single DOF robot hand designs include the VV 5-9 hand developed by Varity Ability Systems Inc. (VASI) and the Otto Bock 2000 hand manufactured by Otto Bock shown in Figure 2.1. As can be seen, these first class of multi-fingered robot hands were very limited in their abilities. The single DOF offered little grasping functionalities beyond the simple pinch grasp [16,17]. This coupled with their excessive weight and bulkiness rendered their use in real life environments almost totally useless. The inadequacy and primitive nature of these early robot hands quickly gave way to more sophisticated and versatile models. Specifically, the need for the greater DOF needed for the manipulation and grasping of different objects saw the rise of highly articulated robot hands [14]. This, in conjunction with biomedical research regarding acquisition of myoelectric or EMG signals from the human forearm led to the rapid development of extremely dexterous robot hands.
While efforts were underway to extract more independent channels from EMG signals for controlling specific robot finger joints [18,19,20], these early investigations were still in their nascent stages. Other investigations pursued more physical means of manipulation. These involved the attachment of wires to the natural motor pathways for directly measuring muscle and tendon movement. These approaches, although promising, were hindered by the surgical nature of the method. For example, in the Tendon Activated Pneumatic (TAP) control approach, the subjects had volition tendon sliders inserted in their forearms that measured the natural tendon movement. The displacement of the tendons then resulted in displacements of air in foam sensors placed on the skin in proximity to the selected target area. These displacements were later processed and used to control the joints of a multi-fingered robot hand [21]. An initial prototype of such a device is shown in Figure 2.2. Clearly, the effectiveness of such methods was offset by their invasiveness. The subjects required surgery for the insertion of tendon sliders and other sensory equipment. Furthermore, the pain and discomfort caused by repeated surgeries to replace malfunctioning devices made these approaches, for the time being, obsolete.

Further experimentation and studies have now demonstrated that EMG signals are sufficient and adequate enough for usage in control of dexterous articulated multi-fingered
robot hands [1,2]. In particular, the noninvasive approach offered by using surface electrodes placed on the skin to read EMG signals allowed myoelectric approaches to gain greater favor in the robotics community. Whereas the invasive treatments had relied on the direct connection of tendon sliders or wires to actuate robot joint motors, the myoelectric method necessitated only the correct acquisition and processing of EMG data for joint control. This introduced a new series of problems, of which noise reduction of the signals and the appropriate feature extraction of the data are only a few.

Figure 2.2. The TAP Version 3 Prototype Hand. Source: W. Craelius, R. Abboudi, N.A. Newby. “Control of a Multi-Finger Prosthetic Hand.” *International Conference on Rehabilitation Robotics*, Stanford, CA 1999.

As refinements in EMG signal feature extraction have improved to include nonparametric approaches such as temporal and spectral methods [3], as well as other approaches tolerating a degree of noise in the signals using Modified Mean Frequency (MMNF) and Modified Median Frequency (MMDF) [22], the ability for EMG signal based control of multi-fingered robot hands has become the primary focus of research. Although a branch of research had centered on the processing of EMG data to produce channel specific signals that can actuate their corresponding joints, the prosthetics community had already agreed on a set of fundamental classes into which most human hand positions can be classified [14]. These include: lateral or key grasps, spherical grasps, cylindrical grasps,
precision or point grasps, and lastly, hook grasps. These classes define the most basic or elementary hand positions and nearly all grasps are variations of these. Because daily human hand movements operate spontaneously and without active conscience control of every single joint, using the EMG signals for pattern classification into the basic hand positions seemed like the most natural choice for multi-fingered robot hand control. This synergetic ability allows through the collapse of large quantities of variables and joint controls into only two: the grasp aperture (a continuous variable) and the grasp type specifier (a five element categorical variable) a system that is both relatively simple and yet highly dexterous and versatile. The robotic hand designed at the Robotics Laboratory at San Diego State University (SDSU) is a prime example of a device being used to investigate the various grasp classes.

The SDSU multi-fingered robot hand shown in Figure 2.3 is a six degree of freedom articulated hand consisting of four fingers each actuated by a single motor, and a thumb actuated by two motors, one for flexion and the other for rotational movement. The hand is driven by the amplifier-motor unit connected via the ServoToGo board which operates as an input/output interface that receives and sends signals to the hand.

![Figure 2.3. The SDSU robot hand shown with various grasp positions.](image)

Each finger consists of three joints, one which serves as an attachment to the base of the hand and is powered by a 6 Watt DC motor. The other two joints are mechanically
coupled and linked through a parallel cascade driven by the amplifier-motor unit connected via the ServoToGo board which operates as an input/output interface that receives and sends signals to the hand. Each finger consists of three joints, one which serves as an attachment to the base of the hand and is powered by a 6 Watt DC motor. The other two joints are mechanically coupled and linked through a parallel cascade [23]. Each joint is independently controlled via encoder signals for proper positioning and this allows the hand to achieve a large range of natural human-like finger motions. Due to this capability, the SDSU robot hand is suited for studying the various fundamental grasps.

The SDSU robot hand in conjunction with the EMG signal research also conducted at the lab, namely, algorithms for real-time feature extraction and classification developed by Vuskovic and Du [3,24,25,26,27], the measurement of prehensile EMG signals by Saksit Siriprayoonsak [8] in 2005, and more recently, the real-time implementation of software for feature extraction and classification of those signals in 2008 by Christopher Miller [9], laid the foundations for developing and creating a system that interfaces human subjects with the SDSU robot hand. The design and implementation of such a system is described in this thesis, but first, a detailed explanation and description of the real-time measurement and feature extraction of EMG signals by Saksit [8] and Miller [9] is presented.
CHAPTER 3

REAL-TIME MEASUREMENT OF PREHENSILE EMG SIGNALS

In order to create an accurate system for reading EMG signals, both the physiology and nature of the targeted human body area and signal itself must be clearly understood. The human body emits and transmits a large quantity of electrical signals associated with muscle and tendon movements. These signals are usually propagated through nerves that carry the signal to the specified location. Electromyography (EMG) is the study and measurement of myoelectric activity, which is the change in the electrical impulses that occur when muscles are at rest or contracted [8]. The kind of body tissue associated with the myoelectric activity of interest in this research is called excitable tissue.

Excitable tissue is present in the human nervous system and is the type most commonly involved during voluntary muscle movements. This tissue, which is composed of neurons, carries the nerve stimuli throughout the body in response to active muscle movement through sensory receptors and axons [28]. The axons conduct electrical impulses to the central nervous system as well as to the spinal cord and brain [8]. The electrical impulses themselves are nothing more than action potentials (APs) or potential differences in the electrical membranes of a cell. These action potentials are transmitted in domino-like fashion down interneurons or motoneurons until reaching a type of axon called efferent, which is responsible for carrying information and signals down to the body parts. At that point, the muscle moves accordingly.

The total voltage generated by all the action potentials falls in the range of 0 to 10 millivolts (peak-to-peak) or 0 to 1.5 millivolts (rms) [8]. This comprises the amplitude of the signal which together with the usable frequency (50-150 Hz) [29] are the most important aspects of the data. These signals are clearly very small and thus an appropriate receiver to gather the data, as well as a method of amplification once it has been obtained, are needed in order to make the information more concrete and useful. The acquisition of surface skin voltages is easily accomplished through the use of surface electrodes. Their usefulness depends on both the type of electrode used and it’s precise placement.
3.1 EMG Electrodes

There are various kinds of electrodes capable of receiving EMG signals. As mentioned before, the small voltages or action potentials created before and during muscle movement travel along nerves and axons to and from the brain. Upon reaching the target area, the voltages are suffused around the surrounding tissues, spreading and conducting through cell liquids and muscles until ultimately reaching the skin. Thus, if the electrode is placed accurately enough, it can record and read the bioelectrical activity of particular joints and tissues.

For the purposes of his research, Saksit has determined that the best electrode to use for gathering EMG signals is the bipolar electrode [8]. The bipolar electrode is both simple to use and inexpensive. Furthermore, it combines accuracy without the invasiveness that other types of larger electrodes have such as tripolar and multipolar electrodes. With the bipolar electrode, one node is used for positive input while the other is used for negative input. These are precisely the ideal inputs required for the next stage, namely, the creation of a differential amplifier to amplify the EMG signals. By using only a single node electrode as a body reference, the overall complexity of the system is reduced. With a working method for obtaining EMG signals set, the processing of the data must be handled very careful so as to avoid and eliminate any external or residual errors or noise. These requirements can be fulfilled concurrently while designing the EMG amplifier.

3.2 EMG Amplifier Issues

The amplification of any electronic signal serves a twofold effect. First, by increasing the strength of the signal it is protected against corruption during transmission, and secondly, it is increased to a level suitable for storage [30]. In particular, when dealing with bioelectric signals read from the skin tissue, other factors such as ambient noise and interference must be taken into account. These arise from sundry sources, such as nearby electronic devices or other instruments producing electromagnetic fields and have detrimental effects on the data. There can also be residual noise interference from the instability of the electrodes themselves. All of these are important considerations that must be taken into account when designing an amplifier. While it is certainly true that no signal can ever be completely free of noise interference, a suitable compromise must be found that
maintains as much of the integrity of the data while reducing and eliminating the greatest noise. In designing his amplifier, Saksit has developed a differential amplifier incorporating an implicit filter that serves well to achieve such standards.

### 3.3 EMG Amplifier Design

The EMG amplifier used in this project was designed and created by Saksit Siriprayoonsak in 2005 [8] at the Robotics Laboratory at San Diego State University. The device is a multi-purpose amplifier that increases signal strength while filtering out noise interference. The general schematic of the amplifier is shown in Figure 3.1.

![Figure 3.1. EMG Amplifier Schematic.](image)

The amplifier consists of several stages. During the preamplification stage, the signal is increased to a level resistant to interference while simultaneously filtering out the noise. Because the raw EMG signals collected by the electrodes are small, a very large gain will be required. Saksit has designed an adequate preamplifier using a differential amplifier with a gain of about 1137 dB [8]. Each channel of EMG data is similarly processed. The circuit has the capacity of receiving four input channels from four electrodes as well as one body reference electrode. The monopolar electrode which serves as the body reference node gives an added measure of stability while correcting the signal level. Each of the four electrodes shares the single body reference node to avoid extraneous superfluous electrodes.

The filter used is an RC high pass filter with a cutoff frequency of about 12 Hz. As mentioned earlier, the usable frequency range of the signals falls between 50-150 Hz. There-
The high pass filter used by Saksit is more than suitable for the data. The last stage of the amplifier serves as a bias or gain adjustment. Because the amplifier must increase the level of the signal to a strength sufficient for appropriate processing afterwards by the A/D converter (to be discussed next), it is sometimes necessary for further corrections to take place. Furthermore, it may be also possible for the signal to have an offset or bias. To correct these, Saksit has incorporated into his device an individual gain and bias adjustment unit for each channel. Thus, each channel can be independently modified without affecting the others. Saksit has determined that a suitable amplification gain for each channel is 21 dB. The bias portion of the last stage is a common reference level adjustment circuit that allows corrections to the signal to be made such that it’s reference or ground line is zero volts or any other quantity as desired by the user [8].

Processing and storage of the EMG signals can only be done once it has been converted from its analog to its digital form. The EMG signal is sent as an analog wave form that must first be digitized [31]. The specific A/D interface card used in Saksit’s project is the NI 6220 M-Series Multipurpose DAQ from National Instrument Company [21]. The relevant specifications are shown in Table 3.1.

Table 3.1. NI 6220 DAQ Specifications

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Channels</td>
<td>8 differential or 16 single ended</td>
</tr>
<tr>
<td>ADC Resolution</td>
<td>16 bits</td>
</tr>
<tr>
<td>Sampling Rate</td>
<td>Maximum 250 kS/s single channel</td>
</tr>
<tr>
<td>Input Range</td>
<td>± 10 V, ± 5 V, ± 1 V, ± 0.2 V</td>
</tr>
<tr>
<td>CMRR (DC to 60 Hz)</td>
<td>92 dB</td>
</tr>
</tbody>
</table>

The device has 16 single-ended channels although only 4 are used in Saksit’s device. It also has 16 bits of analog input resolution as well as possessing a sampling rate more than above the 1000 S/s needed for the project. The A/D interface card allows the data to be processed and manipulated. It’s output is then interpreted by the software program also developed by Saksit called ‘EMG Capture’ or ‘EMGC’ [8]. The program uses the NI 6220’s driver to send and receive data from the A/D card, which itself is receiving signals from the
amplifier device taking input from the electrodes. Thus, a complete system for reading, recording and viewing EMG signals has now been developed. The overall schematic is shown in Figure 3.2 while an actual image of the final manufactured EMG amplifier device is shown in Figure 3.3.

![Figure 3.2. Complete Schematic of Saksit’s Amplifier and EMG Capture Program](image)

Saksit’s amplifier together with the program ‘EMGC’ comprise the total device. The EMG signals are read, processed and amplified by the amplifier, converted to their digital format by the A/D converter, and lastly, displayed and shown to the user with the GUI.
program ‘EMGC’. The whole device is powered by two 9 volt batteries connected in series providing positive and negative voltages, as well as ground for the entire circuit [8].

Saksit has also developed a graphical user interface (GUI) for displaying the amplified EMG signals as well as providing other capabilities for manipulating and displaying the data. These functionalities are part of the EMGC program which is described next.

3.4 EMG CAPTURE PROGRAM

The EMG Capture program is shown in Figure 3.4. The program was developed using Microsoft Visual C++ 6.0 and the Microsoft Foundation Class (MFC) for the Graphical user interface (GUI). The program in only able to run on a machine running Windows 2000 that also has the M-6220 A/D interface card.

Figure 3.4. Snapshot image of the EMG Capture Program GUI.
In order to function properly, this driver file must be included in the EMG Capture Program project. The driver for the card is written in C++ and the program contains the following properties.

### 3.4.1 Main Parameter

The Main Parameter subpanel allows the user to set the sampling rate and maximum buffer size when operating in Real or Record mode. The maximum number of samples that can be stored by the program is contained in the Max Buffer variable and if the program exceeds this value the Max Buffer field will be automatically increased to cover the actual samples of the data.

### 3.4.2 Play Mode

The play mode allows the user to enter either Real-Time, Record or Playback mode. The Real-Time mode initiates continuous EMG data capture from the user. The Record mode allows a segment of interest to the user to be recorded and saved. This same segment can then be replayed using Playback mode. The results of these modes is displayed in the Output Display screen.

### 3.4.3 Voltage Scale

This subpanel allows the user to independently assign different voltage scales to each of the output channels. These scales allow voltage channel displays to be modified so that the data may be appear more graphically readable.

### 3.4.4 Time and Readout

The Time Base subpanel allows the user to specify how many milliseconds of data are shown per division. It resets all channels to the same amount. The Readout subpanel allows the user to display the exact data values corresponding with a specific time. The channel chosen will update to display a line indicating the specified position and the value of the data at that time. The timestamp value outputs a voltage value which is also updated on the Readout subpanel. These panels facilitate data analysis and viewing.
CHAPTER 4

REAL-TIME FEATURE EXTRACTION AND CLASSIFICATION OF PREHENSILE EMG SIGNALS

With the real-time measurement of prehensile EMG signals having been accomplished by Saksit Siriprayoonsak [8], the next stage in the development of a complete system for real-time human driven robot hand grasping movement was the creation of a standardized process and program for obtaining the EMG signals and performing feature classification. The latter subject will be discussed in later sections, but first, the electrode placement steps are outlined.

4.1 EMG SIGNAL COLLECTION

The quality of the EMG signal is affected by a slew of factors, each of which must be understood for optimal placement of the electrodes. A list of the most important factors as formulated by the biomedical community [1,4,32,33] are given below:

1. The type and quality of the electrodes
2. The stability of contact between the electrodes and the skin
3. The distance between the electrode and the target muscle area
4. The timing and intensity of muscle contraction
5. The properties of the tissue surround the target area including thickness and type

The first three factors can be overcome through proper selection and placement of the electrodes relative to the skin. This can only be accomplished by an in depth study of the human forearm, the area of interest in this project.

4.1.1 Properties of the Electrodes

As previously discussed, several methods exist for obtaining EMG signals from the human subject. For this project, wire and needle methods attached directly to target muscle fibers were dismissed due to their invasiveness. Instead, surface electrodes placed on the skin at locations in propinquity to the associated target muscle area provide an adequate
alternative. The electrodes used are four bipolar electrodes placed directly on the skin and one monopolar electrode serving as a common body reference also placed on the skin. Bipolar electrodes were chosen because the differential amplifier inherent in their design provides filtering of unwanted outside noises surrounding the target muscle [9,31]. The bipolar electrodes used were custom made by Saksit Siriprayoonsak [8] for use in his thesis and are shown in Figure 4.1.


The standard recommendations for electrode placement as established in 1999 by the Surface EMG for Non-Invasive Assessment of Muscles (SENIAM) specified that the size of the electrodes should not exceed 10 mm or about 0.4 inches when placed in the direction of the target muscles and that the inter-electrode distance should also not go beyond 20 mm or about 0.787 inches [9,34]. From the image above it is clear that the size is twice the recommended value while the inter-electrode distance is greater by about 0.22 inches. These are elements that affect the quality of the EMG signal collected, making the data susceptible to errors and prone to noise artifacts. These are factors that will have to be addressed in the future to improve the quality of the EMG signals collection.

### 4.1.2 Human Forearm Anatomy

The area of signal interest in this project is the human forearm. It is from the muscle fibers controlling hand and finger movements that the EMG data will originate from. The human forearm is an anatomically complex system composed of nerve, tendon and various muscle fibers. However, for this project, only the areas in Figure 4.2 are considered.
The muscles shown comprise the posterior compartment of the forearm, and as the name suggests, are located on the backside of the human arm. There are four major extensor muscles involved [9]:

1. The *extensor digitorum* which is responsible for the extension of the wrist and finger joints, controlling primarily the middle and ring fingers, and to a lesser extent the index and little fingers.

2. The *extensor pollicis longus* and *extensor pollicis brevis* both responsible for the extension of the thumb and abduction of the wrist.

3. The *extensor indicis* responsible for the extension and adduction of the index finger.

4. The *extensor digiti minimi* responsible for the extension of the little finger.

The muscle groups located on the anterior side of the forearm are responsible for the flexion of the finger joints. They are ideally suited for studying grasp tension and power of the fingers. In future research, these muscle groups could monitored with further electrodes thus providing more control over grasp functions. The ideal electrode placement for the purposes of this project was investigated by Chris Miller [9] in 2008, and together with the diagram in Figure 4.2 allowed a formulation of the steps required.
4.1.3 Electrode Placement

The following procedures outline the steps necessary for optimal electrode placement:

1. The electrode skin contact should be as clean as possible. Shaving of forearm hair is preferable, but not necessary. It is suggested that the skin areas be cleaned with alcohol and allowed to dry before proceeding.

2. Properly orient the bipolar electrode. The imaginary line connecting the two electrodes as shown in Figure 4.1 constitutes the axis of orientation. This imaginary line should be in parallel with the muscle group.

3. Position one bipolar electrode correctly oriented on each one of the extensor muscles shown in Figure 4.2. The letter markers in Figure 4.2 identify the optimum location. Only markers A, B, and D are used for this project.

4. Position one monopolar electrode on the inside of the wrist. The monopolar electrode serves as reference and has the longest cable length.

5. Ensure each electrode is tightly secured. Flexion of the arm should not result in sliding or movement of the electrodes. Use the Velcro strap to tighten. Attempt as much as possible consistent electrode placement for the duration of the runs. Note that the electrodes are covered in a mild adhesive which provides further stability.

6. Use the two additional Velcro straps to secure the forearm to Saksit’s EMG amplifier. The arm should be secure and flush with the device.

7. Lastly, connect bipolar electrode on A to Channel 2. Connect bipolar electrode on D to Channel 1 and connect bipolar electrode on B to Channel 3. The monopolar reference electrode connects to the Ref. channel.

As previously mentioned, only four channels are used in this project: three for the extensor muscles and one as a common body reference. Channel 1 is used to obtain EMG signals from the index finger, channel 2 for the middle and ring fingers, and finally channel 3 is used for the thumb. This project does not utilize readings from channel 4 although in the future this functionality could be added to provide further precision of finger movements and grasp tension. A picture of the final setup is shown in Figure 4.3. The arm is now ready for EMG signal processing. Chris Miller has extended Saksit’s ‘EMGC’ program to incorporate a feature classification of grasps. The program contains several features that incorporate EMG signal reading capabilities as well as their analysis using various algorithms to extract features and perform classification. The combined projects allow real-time experimentation of grasp modes that can later be utilized and integrated into a unified system for control of the six degrees of freedom anthropomorphic robot hand at SDSU. A discussion of his program now follows.
The process developed up to now can be summarized as follows: the forearm has been correctly equipped and accoutered with the electrodes; it has then been attached to Saksit’s amplifier device which is itself connected to the computer; the A/D interface card converts the signals from their analog to digital forms and finally, the software program ‘EMGC’ allows the user to view the incoming data. Because the ultimate purpose of this project is to control in real-time the SDSU multi-fingered robot hand using the EMG signals, the next stage is the processing of the data to extract the correct grasp features. The identification of grasps will then provide a means to send appropriate commands to the SDSU robot hand for real-time control. Feature extraction is without a doubt the most important step in signal processing since it will determine whether the classification method operates effectively. The feature extraction and classification methods used in this research were first researched and developed by Vuskovic and Du [3,24,25,26,27] while Miller [9] provided their software implementation. Before proceeding with a description of the feature extraction method employed by Chris Miller in his thesis, it is appropriate to first characterize what segment of
the data requires analysis. It’s clear that only a window surrounding the initial arm and hand movement is of interest since this segment is responsible for the grasp.

### 4.2.1 Onset of Movement

The portion of the data following the initial movement of the arm to perform a grasp is called the “onset of movement.” The success of EMG signal classification depends critically on whether the correct part of the data is extracted and analyzed. Failure to extract the relevant data permits the intrusion of noise and inconsequential rest position signals to permeate the data thus resulting in later erroneous classifications [9].

### 4.2.2 Bonato Method

The method employed by Chris Miller in his thesis to detect the onset of movement is called the “Bonato method” [9]. The Bonato method uses a baseline and the calculation of its variance to determine when the onset of movement has occurred [25,26]. Prior to any calculation, the Bonato method pre-processes the data using a whitening filter [35]. Whitening of the raw EMG signals is used to make the samples uncorrelated with each other [35]. Afterwards, the squared values of two successive samples of whitened data are summed. This is the test function $g_k$. Using the test function $g_k$ and a threshold value $h$, the method returns an estimate for the minimum time $t_a$ which begins the onset of movement. The value of $t_a$ is simply the first time that $n$ out of $m$ successive samples exceed the chosen threshold $h$, where $m$ is the width of the detection window and exceeds a separate threshold value $T_1$ designated the minimum time period [25,26,35]. The Bonato method is given below:

$$g_k = \frac{1}{\sigma_0^2} (y_{k-1}^2 + y_k^2)$$

$$t_a = \min \{k = 1, 3, 5, \ldots : g_k \geq h \}$$

Note that only odd-numbered samples for $k$ are used so as to avoid overlap between successive values of $g_k$. The formula provides a quick method for determining the minimum time period required for the detection window which can then be used to find the onset of movement.
4.2.3 Feature Extraction

The Bonato method provides an accurate and efficient way of finding the onset of movement. Given the starting time $t_a$, the following 400 ms comprise the EMG signal. This time window was determined manually [9]. For his project, Chris Miller used only four channels, thus there are 1600 samples of raw EMG data. It is from this data that extraction of features must be performed. Extracting the relevant information is a difficult and costly process; and many methods have been devised including temporal and spectral approaches. In his thesis Chris Miller has utilized one of each as a compromise between efficiency and accuracy.

4.2.4 Waveform Length

The waveform length is a temporal approach known for its easy implementation. It provides an overall analysis of the amplitude, frequency and duration of the EMG signal [25, 26]. The Waveform Length method provides one simple formula that yields a precise measurement and analysis of the characteristics of the signal across its total time period. The Waveform Length formula is given below [25,26,35]:

$$WL = \sum_{k=2}^{N} |x_k - x_{k-1}|$$

Simply put, the Waveform Length is the same as stretching the EMG signal into a straight line and measuring its length. This single value measurement results in a value that incorporates changes in frequency and amplitude and as can be seen, requires only one formula to calculate.

4.2.5 Spectral Moments

A spectral method was also used using spectral moments. Although in general, spectral approaches are more computationally intense, they have been shown to yield far greater discriminatory abilities [25,26]. The spectral moment approach used by Chris Miller in his thesis was developed by Vuskovic and Du [3,24,25,26,27] and is an improvement over the classical approach. Namely, the original spectral moment calculation required the use of the power spectral density (PSD) [26]. The continuous frequency of the PSD was first made discrete and then Fourier transformed. The discrete components are then summed and
averaged for each moment [9,24]. Vuskovic and Du have developed an algorithm for computation of spectral moments directly from temporal signals without the need to compute the power spectral density. The algorithm is described as follows: the PSD can be computed by the Wiener-Khintchine representation and is defined as:

\[
P(f) = \sum_{k=-N+1}^{N-1} C_{ss}[k] e^{-j2\pi fk}
\]

where \(C_{ss}[k]\) is given as:

\[
C_{ss}[k] = \frac{1}{N} \sum_{i=0}^{N-k-1} s[i]s[i+k], k \pm 1, \pm 2, \ldots \pm (N-1)
\]

and is the autocorrelation function of the sequence \(s[i]\) [26]. Next, the spectral moments are calculated as follows:

\[
M_m = C_{ss}[0]I_m(0) + 2 \sum_{k=1}^{K} C_{ss}[k]I_m(k)
\]

The initial \(I_m(k)\) have been given by Wiener-Khintchine and further \(I\)-coefficients can be computed recursively. The \(I\)-coefficient equations are given below:

\[
I_m(0) = \frac{1}{2^{m+1}(m+1)}
\]

\[
I_0(k) = 0, \quad I_1(k) = \frac{-1^k - 1}{(2\pi k)^2}, \quad I_2(k) = \frac{(-1)^k}{(2\pi k)^3}, \quad I_3(k) = \frac{3(-1)^k((\pi k)^k - 2) + 2}{(2\pi k)^4}
\]

\[
I_k(k) = 2 \frac{(-1)^k((\pi k)^k - 6}{(2\pi k)^4}, \quad k = 1, 2, 3, \ldots
\]

\[
I_m(k) = \frac{m}{2^{m-1}(2\pi k)^2} ((-1)^k - 2^{m-1}(m-1)I_{m-2}(k))
\]

In his work Miller used the “reduced moments” which are an improvement over the aforementioned moment calculation. They reduce considerably the moment’s susceptibility to noise while delivering feature vectors more yielding to classification. The improved “reduced moments” were first introduced by Vuskovic and Du [25] and apply a further linear
transformation to spectral moment calculation [25]. The reduced moments are given by the following equation [24,25]:

\[ R_m = \frac{M_m - 2I_m(0)M_0}{2} = \frac{1}{2} \sum_{k=1}^{K} C_{ks}[k]I_m(k) \]

Together with Waveform Length, performance tests [9] identified both reduced and straight moments as the best feature extraction methods for use in real-time operation.

### 4.2.6 Spectral Moment Parameter Optimization

Spectral moments, both reduced and straight have several tunable parameters that may be changed to increase feature vector classification success. The accuracy of classification depends on extraction of the salient features within the EMG data. This aids the classifier in distinguishing a set of data from others and ordering it within related data sets. For optimum characterization, the spectral moment parameters must each be optimized. Table 4.1 outlines the set of tunable parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k)</td>
<td>0..N</td>
<td>Time-lag parameter in I-coefficient and Autocorrelation function (N is the sample size)</td>
</tr>
<tr>
<td>(m)</td>
<td>0..4</td>
<td>Highest moment index to use in feature vector (i.e. (m = 2) means that (M_0, M_1, \text{and } M_2) are included)</td>
</tr>
<tr>
<td>(Ts)</td>
<td>0..N</td>
<td>Size of time sample to use for moment calculation</td>
</tr>
<tr>
<td>Flag</td>
<td>0..2</td>
<td>0 – use straight moments 1 – use (M_0) and reduced moments for higher values of (m) 2 – use strictly reduced moments</td>
</tr>
</tbody>
</table>


In his thesis, Chris conducted experiments to obtain the optimized values for the parameters. A parameter was chosen and altered while the remaining three were held constant. The feature vectors obtained were then classified using leave one out validation using 180 samples of EMG recordings. The classifiers used were both the Maximum Likelihood Estimation (MLE) classifier and the Mahalanobis Distance classifier. The classifiers use the feature vectors produced by the four channels of EMG data to create
divisions of sets from which new feature vectors can be classified. These will first be described before proceeding with parameter optimization.

It is important to first understand the classification method suggested by Vuskovic and implemented by Miller in order to gain insight into the selection of values for the tunable parameters. The two classifiers chosen are the Maximum Likelihood Estimation (MLE) and the Mahalanobis--distance classifiers. Training for both classifiers is performed by using the set of feature vectors and calculating a covariance matrix and a mean vector for each class in the set. The equation below defines the covariance equation:

\[ \text{cov}(X) = E[(X - \mu)(X - \mu)^T] \]

where \( E \) is the expected value and \( \mu \) is the average or mean of matrix \( X \). Each class in the set of training vectors has a mean vector. New data is then classified by first obtaining its feature vector and then comparing it against the mean vectors for propinquity of characteristics. In the case of the Mahalanobis-distance classifier, a feature vector’s classification is determined by finding the Mahalanobis-distance between it all and the other set of classes. Proximity of distance determines the vector’s classification within a class.

The equation for Mahalanobis-distance is as follows:

\[ MD(x) = (\mu - x)^T S^{-1}_c (\mu - x) \]

where \( S_c \) is the covariance matrix for class \( C \) and \( \mu_c \) is the mean of class \( C \).

Similar to the Mahalanobis-distance classifier, the MLE also uses the covariance matrix and mean vector for each class but incorporates a probability density function (pdf) with Gaussian distribution to calculate the probability of generating the data given a specific set of parameters \([36]\). The probability density function equation is given below:

\[ p(x \mid \Phi) = \frac{1}{(2\pi)^{\frac{N}{2}}} \left| S_c \right|^{\frac{1}{2}} \exp \left[ -\frac{1}{2} (x - \mu)^T S^{-1}_c (x - \mu) \right] \]

where \( \Phi \) is the maximum likelihood estimator \([36,37]\). The highest probability returned by the pdf in the MLE over all the classes is the new feature vectors class. The simplicity and power of both the MLE and the Mahalanobis-distance classifiers allowed Chris Miller to implement both classification methods in his real-time prehensile EMG signal program\([9]\).
4.2.7 Feature Vector Transformations

The extensive experimental tests conducted by Chris Miller in his thesis demonstrated that optimization of the time-lag, moment, time sample, flag and $\lambda$ parameters had a severe impact on the success of the classifier [9]. Specifically, the time sample parameter appeared to have a linear consequence on successful classification. For example, the Mahalanobis-distance classifier had its highest classification rate (92.8%) at 400 ms with an early peak performance of 91.1% at a time of exactly 270 ms. Similarly, the MLE classifier also had its highest classification rate (94.4%) at 400 ms with an early peak performance of 90.6% at a time of 270 ms. See [9] for more information regarding classifier performance. The important observation is that the highest classification rate appears to occur at higher time sample windows. Clearly, 400 ms is too large a time sample to use in a real-time application while 91.1% success rate with the Mahalanobis-distance at 270 ms is a poor classification rate. In order to improve performance, Miller has applied yet another transformation to the feature vectors in order to reduce the impact of noise and other outliers and normalize errors [9].

The logarithmic transformation proposed by Kajitani et al. [38] is a simple method of improving a classifiers performance by producing feature vectors whose class members are more closely located while simultaneously generating larger distances between different classes. A further improvement was introduced by Box and Cox [39] in which each value of $y$ is instead replaced by $y^{\lambda}$. The Box-Cox transformation used by Miller is given below:

$$y^{(\lambda)} = \begin{cases} \frac{y^{\lambda} - 1}{\lambda}, & \lambda \neq 0 \\ \log y, & \lambda = 0 \end{cases}$$

Visual tests based on three dimensional feature vectors using only three channels and incorporating the application of the Box-Cox transformation revealed that overlap between class grasps was decidedly decreased after the transformations. In particular, from the work conducted by Miller [9], it is clear that there is a significantly reduced variance between the grasp classes. Note that only three channels were used in the transformation tests so as to
provide a direct mapping onto three dimensional space. The experiments indicate that such a transformation greatly increases class differences. Because of this, both the MLE and Mahalanobis-distance classifiers can be expected to display improved discriminatory powers in grasp classification.

4.2.8 Parameter Results

The results in Table 4.2 summarize the optimized parameter values found by Chris Miller [9].

<table>
<thead>
<tr>
<th>Feature Extraction Method</th>
<th>Parameters</th>
<th>Time Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral Moments</td>
<td>Flag = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m = 2 (highest moment index)</td>
<td>400 ms</td>
</tr>
<tr>
<td></td>
<td>k = 11(time lag in I-coefficient)</td>
<td></td>
</tr>
<tr>
<td>Waveform Length</td>
<td>N/A</td>
<td>400 ms</td>
</tr>
</tbody>
</table>

Miller found that a comparison of the performance of the MLE and the Mahalanobis-distance (MAH) classifier clearly demonstrated that although the MLE tended to outperform the MAH in general by small percentage amounts (i.e. 0.2%), the speed and simplicity of the MAH gave it greater versatility and efficiency for use in a real-time application [9]. Furthermore, the performance of the MAH depended intrinsically on the number of grasp samples provided to it. These results indicated that a hybrid approach utilizing both the spectral and waveform length methods offered the greatest flexibility for use with the MAH. The Waveform Length method demonstrated consistently high classification rates with the MAH, scoring above 95% consistently. However, even this impressive performance was surpassed by the spectral method when it was applied to larger data sets. After extensive tests, Miller found that the ideal peak performance occurred when the MAH used the Waveform Length method given few recordings for each grasp (less than 15) while higher classification rates were achieved when it switched to using Spectral methods if larger training sets were provided [9]. These higher MAH rates were accomplished when the Box-Cox transformation was applied with $\lambda = 0.06$.

The preceding discussions have outlined the methodology employed by Chris Miller to perform the feature extraction and classification of prehensile EMG signals. Namely, the
Bonato method is used to first find the onset of movement. Next, using the MAH classifier, the system is trained using either the Waveform Length or Spectral method for feature extraction with the Box-Cox transformation applied. The number of grasps determines which one is used. Lastly, upon performing a new grasp, its feature vector can then be classified using the MAH classifier. The process given above was developed into a software program called “EMGGUI” which contains a graphical user interface for feature extraction and grasp classification. The program allows the user to perform the necessary training and testing of various grasp positions in real-time. The program is an extension of the previous work done by Saksit and its description is the final step in understanding this project’s creation of a real-time program for control of the multi-fingered robot hand at SDSU.

4.3 REAL-TIME EMG CLASSIFIER PROGRAM

The EMG Classifier Program is the software implementation of the feature extraction and classification methods discussed previously. The program has various features facilitating the testing of grasp motions. Several modes endow the user with a wide range of capabilities. The user is allowed to train the classifier using real-time EMG recordings from the user’s forearm or by uploading previously recorded training data. Once training has concluded, the user may classify grasps individually one at a time, or by entering real-time mode in which the program begins continuous real-time classification of the user’s hand grasps. The program was developed by Chris Miller in 2008 [9] and the functions relevant to this project are described next. For a detailed discussion on the program’s features, see [9].

4.3.1 Program Files

The EMG Classifier Program was implemented with MATLAB and C programming languages. The main file is the MATLAB file \textit{EMGProgGUIFC.m} which is the program’s project source file. The secondary file utilized is the dynamic link library file (DLL) \textit{EMGDLL.dll} which contains the functions necessary for data reading from the NI-DAQ card [9]. The project source file automatically loads the remaining necessary MATLAB functions such as \textit{BonatoOnset.m} and \textit{BoxCox.m} and others (see [9] for a thorough description) thus readying the program for feature extraction and classification. The program comes with a default classifier called \textit{dflt_train.mat} enabling the user to begin program usage immediately, although it is highly recommended that the classifier be trained by each user.
due to differences in skin thickness, fatty tissue quantities and other variables. The executable is called `EMGProgGUIFC.exe` and in order to run the program, the running machine must have the Windows 2000 OS, MATLAB or it’s component runtime libraries installed, and the NI-DAQ card installed.

### 4.3.2 Program Features

A sample display from the EMG Classifier Program is shown in Figure 4.4.

![Sample main screen display of the Real-Time EMG Classifier Program](image)

**Figure 4.4. Sample main screen display of the Real-Time EMG Classifier Program.**


The first step that must be taken is training of the Mahalanobis-Classifier. The user may either load the default classifier or tailor the classifier to the user’s own arm. Individual training is done by pressing `Train` under the `Common Tasks` subpanel. The training procedure lasts about 20-30 minutes. During this time, the user is prompted to perform hand grasps at random until 18 recordings for each of the four different hand grasps are completed.
Thus, there will be a total of 72 grasp recordings entered into the classifier. Note that before starting training, the user is asked how many recordings per grasp he wishes to enter. In his thesis experiments Miller has found that 18 recordings for each grasp yields a classifier with a success rate in the 98th percentile [9]. For quicker training, the number of grasps may be reduced, but this will come at the expense of classification rate performance.

The EMG display subpanel shows the EMG signal outputs for each of the three channels. The x-axis is measured in milliseconds while the y-axis shows output in millivolts. It is important that this section be monitored during training. Clean sharp signals are desired or else random noise will infiltrate and have a negative impact on classification performance. A clean signal can be defined as one in which the onset of movement occurs at least 200 ms after the beginning of data collection. The onset of motion is shown by a vertical green line indicating the start of grasp signal recording. As previously mentioned, the highest classification rate occurs using a 400 ms time sample window. Thus, each channel has 400 ms of EMG data entered per grasp. The three combined channels comprise that particular grasps three dimensional vector. After further analysis through either Waveform Length or Spectral moment transformations, the final feature vector is produced. The first 200 ms are used to establish rest position. A signal that has its onset of movement (green vertical line) at any time after the initial 200 ms is considered an adequate signal and should be stored into the classifier.

Once training has concluded, the user may enter either Single Grasp mode or Real-Time mode. Single Grasp mode is entered by selecting Classify Single Grasp in the Options for 3 EMG Signals subpanel. As it’s name suggests, this mode allows the user to perform a single grasp after which the program will classify the data and display/read back the grasp the user has just performed. The grasp’s classification is shown in the Classification subpanel in which a pictorial representation of the recently performed grasp is displayed. Similarly, the Real-Time mode allows classifications of user grasps, however, the program continuously reads, classifies, and updates the Classification display.

The Real-Time mode requires that the user first establish Rest Position under the EMGDevice drop down menu. Establishing Rest Position allows the classifier to determine precisely the onset of movement by determining the variance of the EMG signal emitted by the arm for use with the Bonato method. After this step, the user may continuously perform
grasps. The data from these grasps is interpreted and classified in real-time by the classifier and a graphical image of the grasp updated on the Classification display. It is important to note that for the purposes of this project, the Grasp Control Channel and Grasp Tension subpanels will not be used. These functionalities will serve future use once pressure sensors are attached to the finger tips of the SDSU robot hand in order to test finger force and pressure abilities.

The preceding sections have described and outlined the previous work performed at the Robotics Laboratory at SDSU. Specifically, the foundations for the design and creation of a real-time EMG signal controlled multi-fingered robot hand program have been laid. The work began with the creation of an EMG amplifier device as developed by Saksit [8] in 2005 and progressed with the creation of the EMG Classifier Program by Chris Miller [9] in 2008. This program allowed the extraction and classification of the data acquired through the EMG amplifier and provided real-time grasp analysis. Both these projects will serve as crucial components in the focus and purpose of this project.
CHAPTER 5

ROBOT HAND MOTION CONTROL SYSTEM

This chapter provides a detailed description of the overall hardware and software used for real-time robot hand control. The initial section outlines the SDSU robot hand and discusses the theory behind joint controllers, specifically the design of PID controllers for controlling the fingers of the robotic hand. This is followed by a description of the individual components, describing each in detail and their use in the final integrated system. Finally, the chapter concludes with an overview of the interconnection table and discusses how communication between the various devices is achieved via the transition box.

5.1 OVERALL SYSTEM DESCRIPTION

The overall system consists of the multi-fingered robot hand manufactured by Deconel Instruments, the EMG amplifier device created by Saksit [8], the EMG Classifier Program developed by Chris Miller [9], and includes in addition several other hardware and software components developed in this thesis. The overall system schematic diagram is shown in Figure 5.1. The two aforementioned components have already been described in detail. The EMG Classifier Program runs on the computer called “SKIPPER”, which is the same computer that the EMG amplifier device is attached to and sends data to. The previous work concentrated upon usage of that computer for computation and analysis of grasp motions. The remainder of the components including: the SDSU robot hand and its motion control system, the ServoToGo board, and the Transition board will now be described.

5.1.1 SDSU Robot Hand

As discussed in Chapter 2, the SDSU multi-fingered robot hand is a 6 DOF articulated hand. The hand is capable of individual joint movements, however each of the finger segments cannot be independently operated. The hand uses six Maxon motors, five attached to the base of the fingers providing flexion for each joint. The last motor is used for thumb rotation. Each of the finger segments is coupled through linkage connections [40]
Figure 5.1. Overall System Diagram showing the various components.
providing the hand with realistic imitation of the grasp motions of the human hand. This type of movement is termed “Synergetic” motion. Synergetic motion is the type of coordinated biological movement readily seen in nature. It involves the contraction of degrees of freedom into a type of synergetic mapping that creates the coordinated functional behavior. Synergy itself can be defined as specific ordered behavior or coordination that “a) identifies collective variables (order parameters) that characterize movement patterns; b) mapping the observed stable patterns onto attractors of the collective variable [40,41].” For the purposes of this project, the collective variables that require mapping are the individual joint values for each of the finger joints. These values comprise the joint space of the multi-fingered robot hand. In total there are six joint values (joints $\theta_i$, $i = 1,2,\ldots,6$). These values will be contracted into “one” value corresponding to the specific grasp performed by the user.

The specific methodology employed by this project for contraction of degrees of freedom will be discussed at a later section wherein the robot grasp mode training will also be described. For now it suffices to say that the parallel link connections of the SDSU robot hand make it ideally suited for testing synergetic hand motions and grasps.

### 5.1.2 Joint Controllers

The joint controllers of the SDSU robot hand are responsible for the motions and movements of the fingers. They receive desired position values from the motors and then respond appropriately to ensure those values are reached. Signals are generated and communicated back and forth between the joint controllers and the Servo To Go hardware board. The signals sent to the joints then go through a PID controller to ensure smooth and quick responses to the commands received.

### 5.1.3 Hand Actuator Model

Controlling the end effector is an integral part of the joint control device. These joint controllers receive the desired joint displacements and output control signals in digital form that after processing through a D/A converter are interpreted by the actuators [42]. Each joint has a separate joint controller, actuator and speed reducing gear. The simplified block diagram of a DC hand actuator is shown in Figure 5.2 [42]. A description of all variables with some values is also given in Table 5.1.

Table 5.1. Specifications of the Maxon RE 016-042-05-EAA100A DC Motor.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_a$</td>
<td>Terminal or Armature Resistance</td>
<td>3.38 Ohm</td>
</tr>
<tr>
<td>$K_a$</td>
<td>Torque Constant</td>
<td>8.11 mNm/A</td>
</tr>
<tr>
<td>$J_m$</td>
<td>Rotor Inertia</td>
<td>1.27 gcm²</td>
</tr>
<tr>
<td>$K_g$</td>
<td>Gear Transmission Ratio - Thumb</td>
<td>1:26</td>
</tr>
<tr>
<td></td>
<td>Gear Transmission Ratio - Finger</td>
<td>1:19</td>
</tr>
<tr>
<td>$G_a$</td>
<td>Driver Gain</td>
<td>1</td>
</tr>
<tr>
<td>$K_b$</td>
<td>Speed or Proportionality Constant</td>
<td>1180 rpm/V</td>
</tr>
<tr>
<td>$V_0$</td>
<td>Nominal Voltage</td>
<td>12 Volt</td>
</tr>
<tr>
<td>$\omega_0$</td>
<td>No Load Speed</td>
<td>13900 rpm</td>
</tr>
</tbody>
</table>

The values for the variables were obtained from the Maxon Precision Motors data sheet [43]. It is important to note that the $L_a$ variable, termed the “armature inductance”, is usually of small value and can be neglected. Thus, the above block diagram can be further reduced. This reduction will allow us to calculate the remaining variables specifically $B_m$, the motor viscous friction. After eliminating $L_a$ and combining the remaining blocks, we arrive at simplified block diagram in Figure 5.3. Removal of the feedback loop through block diagram transformations gives the final DC motor diagram in Figure 5.4.
From Figure 5.4, $K_m$ and $T_m$ are defined as follows:

$$K_m = \frac{K_a}{R_a B_m + K_a K_b} \quad (5.1)$$

$$T_m = \frac{J_m R_a}{R_a B_m + K_a K_b} \quad (5.2)$$

$K_m$ and $T_m$ cannot yet be solved. First the value of $B_m$ must be found. To find this value, an input step function of amplitude $V_o$ is applied to the no load speed $\omega_o$. The final settled value after application of this nominal input voltage gives:

$$\omega_0 = \omega_m (t = \infty) \lim_{s \to 0} s \frac{K_m V_0}{(T_m s + 1)s} = K_m V_0 \quad (5.3)$$

Substitution of equation (5.1) into equation (5.3) now gives:

$$\omega_0 = \frac{K_a V_0}{R_a B_m + K_a K_b} \quad (5.4)$$

which can be rearranged and solved for $B_m$ yielding:
After substituting in the values given from Table 5.1 into equation (5.5), the final calculation for $B_m$ results in $B_m = 3.623 \times 10^{-7}$. Using the value for $B_m$ and plugging it into equations (5.1) and (5.2) also gives:

\[
K_m = 7.65 \quad \text{and} \quad T_m = 0.000405
\]

The above are the values for the manipulator gain and time constraint respectively.

More generally, expressions for the variables that aid in the calculation of intermediate values when load is taken into account give $J_{eff}$ and $B_{eff}$. These are the effective moment of inertia and viscous friction, respectively, when a load ($J_L$ and $B_L$) is also applied to the system. This new system can be represented with the block diagram in Figure 5.5 which can be reduced to Figure 5.6.

![Figure 5.5. Block diagram of DC motor with load applied to system.](image)

![Figure 5.6. Reduced block diagram of Figure 5.5.](image)

\[
B_m = \left(\frac{K_m}{R_a}\right)\left(\frac{V_o}{\omega_o} - K_b\right)
\]  

(5.5)
The final result for $J_{eff}$ and $B_{eff}$ below can now be found using Figure 5.7.

\[ J_{eff} = J_m + K_g^2 J_L \]
\[ B_{eff} = B_m + K_g^2 B_L \]

![Figure 5.7. Final simplified Motor block diagram.](image)

The overall general equations for manipulator gain and manipulator time constant are now computed. The below equations incorporate the link load and thus allow for its inclusion or exclusion when calculating values.

\[ K = \frac{(K_a K_g G_a)}{(R_a B_{eff} + K_a K_b)} \]
\[ T = \frac{R_a J_{eff}}{(R_a B_{eff} + K_a K_b)} \]
\[ K_p = \frac{K_g R_a}{G_a K_a} \]

Lastly, a detailed SIMULINK simulation diagram of the hand actuator model is presented in Figure 5.8. The diagram displays all the major components with a sample input step function as discussed previously in equation (5.3). The model allows analysis of the behavior of the angular displacement and angular velocity of the motor shaft to various voltage inputs.
5.1.3.1 PD CONTROLLER

The above joint controller discussion described the simple P controller. The P controller is a simplified joint controller with only positional feedback \[42\]. However, these type of controllers are usually insufficient since they lack stability when even small disturbances are introduced into the system. The PD controller addresses this issue by incorporating an additional velocity feedback into the system that provides a damping effect that gives more stability to the controller. The new block diagram of the PD controller is shown in Figure 5.9:

![Block diagram of PD controller](image)

**Figure 5.8. SIMULINK model of motor showing all components.**

**Figure 5.9. Block diagram of PD controller.**
The standard form of the transfer function is given in the equation below:

\[ W(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \]  

(5.6)

Calculating the transfer function for the block diagram in Figure 5.9 and comparing with equation (5.6) gives:

\[
W_{pd}(s) = \frac{\theta(s)}{\theta_d(s)} = \frac{KK_p}{(Ts^2 + s(1 + KK_v) + KK_p)}
\]

\[
\zeta = \frac{1 + KK_v}{2T\omega_n}
\]

\[
\omega_n^2 = \frac{KK_p}{T}
\]

Now \(K_p\) and \(K_v\) must satisfy the following conditions if the maximal value for \(K_p\) is chosen [42]:

\[
\omega_n^2 \leq \frac{\omega_r^2}{4} \Rightarrow K_p \leq \frac{\omega_r^2 T}{4K}
\]

\[K_v \geq \frac{\omega_r T - 1}{K}\]

Experimentally altering and modifying the values for \(K_p\) and \(K_v\) within the allowable constraints ensures that the controller is able to control both the overshoot and the resonance of the vibrations caused by introducing disturbances to the system. This is what makes PD controllers particular attractive. The extra positional and velocity parameters provide greater stability and control with minimal added complexity.

5.1.3.2 PID CONTROLLER

The last type of controller is the PID controller. It is the type of controller used in this project to control the fingers of the SDSU robot hand. The PID controller offers better tracking capability and greater ability to compensate for the random torque disturbances than
the previously discussed PD controller. It includes an additional parameter called the integral feedback constant. The block diagram for a PID controller is shown in Figure 5.10.

![Block diagram of a PID Controller.](image)

**Figure 5.10. Block diagram of a PID Controller.**

The SIMULINK model for a complete system that includes the actuator and the controller is shown in Figure 5.11. The actuator component sub-diagram is also shown in Figure 5.12 and the controller sub-diagram in Figure 5.13.

![Top Level diagram of Controller and Actuator.](image)

**Figure 5.11. Top Level diagram of Controller and Actuator.**
As previously mentioned, the simplicity and stability of the PID controller makes it an ideal controller to use especially in situations where disturbances may be introduced. This is precisely why it is heavily used in industry and why it was chosen as the controller type for the joints of the robot hand in this project. It is easy to show that there are no static errors with a PID controller. In a non-disturbed system, the unit step input error is:

$$\varepsilon_1 = \lim_{s \to 0} s(1 - W_{PD}(s)) \frac{1}{s} = 0$$
since,

\[ W_{PID} = \frac{sKK_p + KK_i}{(Ts + 1 + KK_r)s^2 + KK_p s + KK_i} \]

and for a unit ramp input is also 0:

\[ \varepsilon_2 = \lim_{s \to 0} s(1 - W_{PID}) \frac{1}{s^2} = \lim_{s \to 0} \frac{(Ts + 1 + KK_r)s}{(Ts + 1 + KK_r)s^2 + KK_p s + KK_i} = 0 \]

The error if a torque disturbance is included is also 0 since,

\[ E(s) = \frac{K_pK}{(Ts + 1 + K_rK)s + K_pK + \frac{K_rK}{s}} \]

\[ \varepsilon_d = \lim_{s \to 0} sE(s) = \frac{K_pK_5T_D}{(Ts + 1 + K_rK)s + K_pK + K_rK} = 0 \]

The above demonstrate that there is no tracking error with a PID controller and that furthermore, the static errors for the controller have been eliminated [42]. As graphical confirmation, the graph in Figure 5.14 shows how a small disturbance torque applied to the system is stabilized by the PID controller model of Figure 5.11.

Figure 5.14. Impact of torque disturbance on error with \( t_d = 0.5\text{Nm} \).
5.1.4 Servo To Go Board

As shown in Figure 5.1, the Servo To Go board installed on the Bronco Computer is the hardware interface between the SDSU robot hand, the transition board, and the software program developed in this project. The design of the board and the versatility it offers in providing a relatively low cost motion control board makes it ideally suited for applications including machine control, automation and robotics [44]. The following discussion describes a few of the most important and relevant properties of the Servo To Go (STG) hardware. This is then followed by a description of the STG software driver and some of its command highlights.

5.1.4.1 STG HARDWARE

The Servo To Go board is an ISA bus card capable of controlling up to eight servo motors. The model type used in this project is the 8-axis ISA Bus Servo I/O Card. A close up image of the STG board is shown in Figure 5.15:

![STG Board Image](image_url)

**Figure 5.15. The 8-axis ISA Bus Servo I/O Card.**

The board has many features including 8 channels of quadrature encoder input, 8 channels of analog output, 32 bits of digital I/O, 8 channels of analog input, and an interval timer capable of interrupting the PC [44]. Table 5.2 gives a summary of the board.
Table 5.2. Specifications of the 8-axis ISA Bus Servo I/O Card.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoder Input</td>
<td>8 channels of encoder input A,B, and Index signal inputs</td>
</tr>
<tr>
<td></td>
<td>24 Bit counters</td>
</tr>
<tr>
<td>Analog Output</td>
<td>8 channels of analog output +10 Volt to -10 Volt span</td>
</tr>
<tr>
<td></td>
<td>13 bit resolution</td>
</tr>
<tr>
<td>Digital Input and Output</td>
<td>32 bits configurable in various input and output combinations Opto-22 compatible</td>
</tr>
<tr>
<td>Analog Input</td>
<td>8 channels of analog input 13 bit resolution Configurable +/-10V or +/- 5V spans</td>
</tr>
<tr>
<td>Interval Timers</td>
<td>Capable of interrupting the PC Programmable to 10 minutes in 25 microsecond increments</td>
</tr>
</tbody>
</table>


As shown in Figure 5.15, the STG board contains four connectors: P1, P2, P3 and P4. Each of these is a 50 pin connector. For the purposes of this project only pin connectors P1, P3 and P4 are used. Connectors P3 and P4 are responsible for the encoder input/outputs that control the motor’s position. Specifically, from Table 5.2 there are only 8 channels of encoder input as well as analog input and output. For this project, the SDSU robot hand consists of 6 DOF and thus the connector responsibilities have been distributed as follows: P3 controls and monitors encoder input and analog output for joint axis 0, 1, 2, and 3. These are the pinky, ring, middle, and index fingers respectively; P4 controls and monitors encoder input and analog output for joint axis 4 and 5, the thumb’s flexion and rotation. Figure 5.16 and Figure 5.17 show the pin-out connections for P1, P2, P3, and P4, while Table 5.3 and Table 5.4 provide detailed descriptions of the symbols used in the figures [45].

Pin-out connectors P3 and P4 are used to control robot joints, while P1 is used to supply the 5V needed to power all encoders. The encoders are quadrature encoders with inputs A and B capable of generating a 24 bit position count. Each axis has one set of A+, A-, B+, B-, I+ and I- signals. The RS-422 receives differential inputs and translates this into a positive or negative encoder count. The I+/I- is used to monitor the Index or Home position and emits a pulse every time the motor passes through that position. However, for the purposes of this project, only the A+ and B+ encoder signals are used. With only
<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Pin</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Opto-23, C7</td>
<td>2</td>
<td>Gnd</td>
</tr>
<tr>
<td>3</td>
<td>Opto-22, C6</td>
<td>4</td>
<td>Gnd</td>
</tr>
<tr>
<td>5</td>
<td>Opto-21, C5</td>
<td>6</td>
<td>Gnd</td>
</tr>
<tr>
<td>7</td>
<td>Opto-20, C4</td>
<td>8</td>
<td>Gnd</td>
</tr>
<tr>
<td>9</td>
<td>Opto-19, C3</td>
<td>10</td>
<td>Gnd</td>
</tr>
<tr>
<td>11</td>
<td>Opto-18, C2</td>
<td>12</td>
<td>Gnd</td>
</tr>
<tr>
<td>13</td>
<td>Opto-17, C1</td>
<td>14</td>
<td>Gnd</td>
</tr>
<tr>
<td>15</td>
<td>Opto-16, C0</td>
<td>16</td>
<td>Gnd</td>
</tr>
<tr>
<td>17</td>
<td>Opto-15, B7</td>
<td>18</td>
<td>Gnd</td>
</tr>
<tr>
<td>19</td>
<td>Opto-14, B6</td>
<td>20</td>
<td>Gnd</td>
</tr>
<tr>
<td>21</td>
<td>Opto-13, B5</td>
<td>22</td>
<td>Gnd</td>
</tr>
<tr>
<td>23</td>
<td>Opto-12, B4</td>
<td>24</td>
<td>Gnd</td>
</tr>
<tr>
<td>25</td>
<td>Opto-11, B3</td>
<td>26</td>
<td>Gnd</td>
</tr>
<tr>
<td>27</td>
<td>Opto-10, B2</td>
<td>28</td>
<td>Gnd</td>
</tr>
<tr>
<td>29</td>
<td>Opto-9, B1</td>
<td>30</td>
<td>Gnd</td>
</tr>
<tr>
<td>31</td>
<td>Opto-8, B0</td>
<td>32</td>
<td>Gnd</td>
</tr>
<tr>
<td>33</td>
<td>Opto-7, A7</td>
<td>34</td>
<td>Gnd</td>
</tr>
<tr>
<td>35</td>
<td>Opto-6, A6</td>
<td>36</td>
<td>Gnd</td>
</tr>
<tr>
<td>37</td>
<td>Opto-5, A5</td>
<td>38</td>
<td>Gnd</td>
</tr>
<tr>
<td>39</td>
<td>Opto-4, A4</td>
<td>40</td>
<td>Gnd</td>
</tr>
<tr>
<td>41</td>
<td>Opto-3, A3</td>
<td>42</td>
<td>Gnd</td>
</tr>
<tr>
<td>43</td>
<td>Opto-2, A2</td>
<td>44</td>
<td>Gnd</td>
</tr>
<tr>
<td>45</td>
<td>Opto-1, A1</td>
<td>46</td>
<td>Gnd</td>
</tr>
<tr>
<td>47</td>
<td>Opto-0, A0</td>
<td>48</td>
<td>Gnd</td>
</tr>
<tr>
<td>49</td>
<td>+5V</td>
<td>50</td>
<td>Gnd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Pin</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADC Chan 0</td>
<td>2</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>3</td>
<td>ADC Chan 1</td>
<td>4</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>5</td>
<td>ADC Chan 2</td>
<td>6</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>7</td>
<td>ADC Chan 3</td>
<td>8</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>9</td>
<td>ADC Chan 4</td>
<td>10</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>11</td>
<td>ADC Chan 5</td>
<td>12</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>13</td>
<td>ADC Chan 6</td>
<td>14</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>15</td>
<td>ADC Chan 7</td>
<td>16</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>17</td>
<td>Opto-15, D7</td>
<td>18</td>
<td>Gnd</td>
</tr>
<tr>
<td>19</td>
<td>Opto-14, D6</td>
<td>20</td>
<td>Gnd</td>
</tr>
<tr>
<td>21</td>
<td>Opto-13, D5</td>
<td>22</td>
<td>Gnd</td>
</tr>
<tr>
<td>23</td>
<td>Opto-12, D4</td>
<td>24</td>
<td>Gnd</td>
</tr>
<tr>
<td>25</td>
<td>Opto-11, D3</td>
<td>26</td>
<td>Gnd</td>
</tr>
<tr>
<td>27</td>
<td>Opto-10, D2</td>
<td>28</td>
<td>Gnd</td>
</tr>
<tr>
<td>29</td>
<td>Opto-9, D1</td>
<td>30</td>
<td>Gnd</td>
</tr>
<tr>
<td>31</td>
<td>Opto-8, D0</td>
<td>32</td>
<td>Gnd</td>
</tr>
<tr>
<td>33</td>
<td>IN2</td>
<td>34</td>
<td>Gnd</td>
</tr>
<tr>
<td>35</td>
<td>IN1</td>
<td>36</td>
<td>Gnd</td>
</tr>
<tr>
<td>37</td>
<td>IN0</td>
<td>38</td>
<td>Gnd</td>
</tr>
<tr>
<td>39</td>
<td>EXLATCH</td>
<td>40</td>
<td>Gnd</td>
</tr>
<tr>
<td>41</td>
<td>T2GATE</td>
<td>42</td>
<td>Gnd</td>
</tr>
<tr>
<td>43</td>
<td>TA2</td>
<td>44</td>
<td>Gnd</td>
</tr>
<tr>
<td>45</td>
<td>/WATCHDOG</td>
<td>46</td>
<td>Gnd</td>
</tr>
<tr>
<td>47</td>
<td>NC</td>
<td>48</td>
<td>Gnd</td>
</tr>
<tr>
<td>49</td>
<td>+5V</td>
<td>50</td>
<td>Gnd</td>
</tr>
</tbody>
</table>

Figure 5.16. STG board P1 and P2 pin-out specifications.
<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Pin</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analog Gnd</td>
<td>2</td>
<td>DAC 0</td>
</tr>
<tr>
<td>3</td>
<td>Analog Gnd</td>
<td>4</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>5</td>
<td>DAC 2</td>
<td>6</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>7</td>
<td>Analog Gnd</td>
<td>8</td>
<td>DAC 1</td>
</tr>
<tr>
<td>9</td>
<td>Analog Gnd</td>
<td>10</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>11</td>
<td>DAC 3</td>
<td>12</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>13</td>
<td>Gnd</td>
<td>14</td>
<td>A 0 +</td>
</tr>
<tr>
<td>15</td>
<td>A 0 -</td>
<td>16</td>
<td>Gnd</td>
</tr>
<tr>
<td>17</td>
<td>B 0 +</td>
<td>18</td>
<td>B 0 -</td>
</tr>
<tr>
<td>19</td>
<td>Gnd</td>
<td>20</td>
<td>I 0 +</td>
</tr>
<tr>
<td>21</td>
<td>I 0 -</td>
<td>22</td>
<td>Gnd</td>
</tr>
<tr>
<td>23</td>
<td>A 1 +</td>
<td>24</td>
<td>A 1 -</td>
</tr>
<tr>
<td>25</td>
<td>Gnd</td>
<td>26</td>
<td>B 1 +</td>
</tr>
<tr>
<td>27</td>
<td>B 1 -</td>
<td>28</td>
<td>Gnd</td>
</tr>
<tr>
<td>29</td>
<td>I 1 +</td>
<td>30</td>
<td>I 1 -</td>
</tr>
<tr>
<td>31</td>
<td>Gnd</td>
<td>32</td>
<td>A 2 +</td>
</tr>
<tr>
<td>33</td>
<td>A 2 -</td>
<td>34</td>
<td>Gnd</td>
</tr>
<tr>
<td>35</td>
<td>B 2 +</td>
<td>36</td>
<td>B 2 -</td>
</tr>
<tr>
<td>37</td>
<td>Gnd</td>
<td>38</td>
<td>I 2 +</td>
</tr>
<tr>
<td>39</td>
<td>I 2 -</td>
<td>40</td>
<td>Gnd</td>
</tr>
<tr>
<td>41</td>
<td>A 3 +</td>
<td>42</td>
<td>A 3 -</td>
</tr>
<tr>
<td>43</td>
<td>Gnd</td>
<td>44</td>
<td>B 3 +</td>
</tr>
<tr>
<td>45</td>
<td>B 3 -</td>
<td>46</td>
<td>Gnd</td>
</tr>
<tr>
<td>47</td>
<td>I 3 +</td>
<td>48</td>
<td>I 3 -</td>
</tr>
<tr>
<td>49</td>
<td>+5</td>
<td>50</td>
<td>+5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Pin</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analog Gnd</td>
<td>2</td>
<td>DAC 4</td>
</tr>
<tr>
<td>3</td>
<td>Analog Gnd</td>
<td>4</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>5</td>
<td>DAC 6</td>
<td>6</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>7</td>
<td>Analog Gnd</td>
<td>8</td>
<td>DAC 5</td>
</tr>
<tr>
<td>9</td>
<td>Analog Gnd</td>
<td>10</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>11</td>
<td>DAC 7</td>
<td>12</td>
<td>Analog Gnd</td>
</tr>
<tr>
<td>13</td>
<td>Gnd</td>
<td>14</td>
<td>A 4 +</td>
</tr>
<tr>
<td>15</td>
<td>A 4 -</td>
<td>16</td>
<td>Gnd</td>
</tr>
<tr>
<td>17</td>
<td>B 4 +</td>
<td>18</td>
<td>B 4 -</td>
</tr>
<tr>
<td>19</td>
<td>Gnd</td>
<td>20</td>
<td>I 4 +</td>
</tr>
<tr>
<td>21</td>
<td>I 4 -</td>
<td>22</td>
<td>Gnd</td>
</tr>
<tr>
<td>23</td>
<td>A 5 +</td>
<td>24</td>
<td>A 5 -</td>
</tr>
<tr>
<td>25</td>
<td>Gnd</td>
<td>26</td>
<td>B 5 +</td>
</tr>
<tr>
<td>27</td>
<td>B 5 -</td>
<td>28</td>
<td>Gnd</td>
</tr>
<tr>
<td>29</td>
<td>I 5 +</td>
<td>30</td>
<td>I 5 -</td>
</tr>
<tr>
<td>31</td>
<td>Gnd</td>
<td>32</td>
<td>A 6 +</td>
</tr>
<tr>
<td>33</td>
<td>A 6 -</td>
<td>34</td>
<td>Gnd</td>
</tr>
<tr>
<td>35</td>
<td>B 6 +</td>
<td>36</td>
<td>B 6 -</td>
</tr>
<tr>
<td>37</td>
<td>Gnd</td>
<td>38</td>
<td>I 6 +</td>
</tr>
<tr>
<td>39</td>
<td>I 6 -</td>
<td>40</td>
<td>Gnd</td>
</tr>
<tr>
<td>41</td>
<td>A 7 +</td>
<td>42</td>
<td>A 7 -</td>
</tr>
<tr>
<td>43</td>
<td>Gnd</td>
<td>44</td>
<td>B 7 +</td>
</tr>
<tr>
<td>45</td>
<td>B 7 -</td>
<td>46</td>
<td>Gnd</td>
</tr>
<tr>
<td>47</td>
<td>I 7 +</td>
<td>48</td>
<td>I 7 -</td>
</tr>
<tr>
<td>49</td>
<td>+5</td>
<td>50</td>
<td>+5</td>
</tr>
</tbody>
</table>

Figure 5.17. STG board P3 and P4 pin-out specifications.
Table 5.3. STG board connector pin-outs.

<table>
<thead>
<tr>
<th>Connector Name</th>
<th>Location*</th>
<th>Pin Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Furthest right, has a bracket.</td>
<td>50</td>
<td>24 bits of digital I/O (Ports A, B, and C) which are Opto-22 compatible.</td>
</tr>
<tr>
<td>P2</td>
<td>2nd from right</td>
<td>50</td>
<td>8 bits of user I/O (Port D), 8 motor direction bits, for a total of 16 Opto-22 compatible bits, and 8 channels of analog input.</td>
</tr>
<tr>
<td>P3</td>
<td>3rd from right</td>
<td>50</td>
<td>Encoder input and analog output for Axis 0-3</td>
</tr>
<tr>
<td>P4</td>
<td>Left-most</td>
<td>50</td>
<td>Encoder input and analog output for Axis 4-7</td>
</tr>
<tr>
<td>P5</td>
<td>near bottom edge</td>
<td>2</td>
<td>Battery input, right pin is ground, left pin is +V.</td>
</tr>
</tbody>
</table>

Table 5.4. Description of the STG connector pins.

| Opto-23, C7 | is | the I/O Port C Bit 7, and pin 23 of an Opto-22 card. |
| DAC 1       | is | the digital-to-analog converter channel 1. |
| A 1 +       | is | the encoder “A” signal for channel 1. For differential input it is the more positive “A” signal, and for single-ended input it is the only “A” signal. |
| A 1 -       | is | the more negative differential input for encoder channel “A”. For single-ended mode signals, this pin must be left unconnected (NOT grounded). |
| ADC Chan 7  | is | the analog-to-digital converter channel 7 |
| Analog Gnd  | is | analog ground |
| /WATCHDOG   | is | watch dog timer output |
| TA2          | is | input to timer 2 (if selected by jumper J7) |
| T2GATE       | is | gate for timer 2 |
| EXLATCH      | is | external latch for encoder counters (enabled via software) |
| IN2          | is | general purpose input 2 (see register BRDTST) |
| Gnd          | is | digital ground. |
| NC           | is | not connected. |
| +5           | is | 5 volt power. |
these two signals, abduction and adduction motions of the finger joints can be properly monitored.

The encoder signals are used to obtain position values for all the joints. These signals are then used by the PID controller for each joint to reach the desired joint position. As described previously in section (5.1.3.2), the control law for a PID controller uses the error in desired motor position, positional and velocity feedback constants, and the derivative of the current motor angle to produce corrections to the voltage outputs given to each of the joints. In this manner, proper motions of the joints can be achieved.

The signals the STG board can send and receive are part of the software driver program provided by the manufacturer. This driver program will now be described.

### 5.1.4.2 STG SOFTWARE DRIVER

The STG device driver allows communication between application programs and the Servo To Go board. For this project, the driver is a kernel mode device driver running in Windows 2000 [45]. Kernel mode drivers give the calling program complete control over CPU instructions and memory. This greater access translates into a much higher degree of performance. The other type of driver commonly used for hardware devices is a user mode driver. User mode drivers do not typically provide the same kind of performance kernel mode drivers deliver. Whereas kernel mode drivers have unrestricted and direct access to the underlying hardware, user mode drivers must act through intermediaries. The intermediary is usually the system API which can directly access the hardware and memory. However, despite their reduced performance, user mode drivers offer greater protection and recoverability. Because they do not operate at the lowest and most trusted level of the operating system, errors and crashes do not domino effect into the entire system. Thus, crashes are usually isolated and not system wide. However, because the ISA Bus Sevo I/O card usually operates in single hardware device environments such as assembly automation or specialty machine control, the device driver included with the STG board is a kernel mode driver.

The device driver acts as a translator between the application programs or operating system and the actual hardware device itself. They facilitate interaction with the hardware device by creating a uniform standardized way of calling or invoking routines in the driver.
The application program developed by the programmer uses the functions provided by the device driver’s Application Programming Interface (API). These routines are then interpreted by the driver to issue the appropriate commands to the device.

The software driver and the application program reside in different layers on the operating system. At the lowest level with the operating system is the device driver. At a much higher level lies the application program coded in C++/C or other languages. Like many other hardware devices, the driver is installed directly into the operating system. The Servo To Go Windows driver consists of two files: stgdrvr.sys and servodll.dll. In order to install the driver, both these files must be placed in the C:\WINDOWS\system32\drivers\ directory. The driver also comes with a .ini file that makes the needed changes to entries in the Windows registry. Although the STG software includes a DLL file called stglib.dll for use mainly with Visual Basic programs, it may also be used with C++ programs since it simplifies calls to the driver. The DLL must be placed in a location where the application program can find it. For use with multiple application programs, it is suggested that the DLL be placed in a central location such as the “Windows\System\” directory. For more information regarding system specific driver installation please see [45].

The STG driver allows control of up to eight servo motors and can act on over 40 different commands from the user. These commands enable the driver to perform a PID servo control algorithm on up to eight axis of motion in less than 1 ms on an Intel 486, 66 MHz CPU [45]. Furthermore, the driver allows simulation of hardware capabilities if it determines that the STG board is not installed.

The STG driver may be accessed in one of two ways, directly through the driver commands or through the STG API. The driver commands are the standard method of invoking the STG’s routines. They are available directly once the STG driver and stgdrvr.sys file has been installed correctly. In order to use the driver commands, a path to the driver must first be opened. In Windows this is usually done using the CreateFile() function from the Windows API. Once this has been accomplished, a call to the actual device can be made and the device is available for use. Direct usage of the driver commands involves first declaring a variable of type IoCtlBuf. The IoCtlBuf is a structure that contains several command elements. Table 5.5 describes the structure of the IoCtlBuf variable [45].
Table 5.5. Elements of the *IoCtlBuf* command structure.

<table>
<thead>
<tr>
<th>Structure Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCommand</td>
<td>The specific command to the servo driver. Different commands require different combinations of the structure members.</td>
</tr>
<tr>
<td>nAxis</td>
<td>Some commands are for a single axis only. This value specifies the axis number.</td>
</tr>
<tr>
<td>lErrorCode</td>
<td>The value of the error, or 0 if none has occurred. Error codes are defined in header files.</td>
</tr>
<tr>
<td>acErrorMessage</td>
<td>Depending on the error, a short description of it is written here.</td>
</tr>
<tr>
<td>lTimeStamp</td>
<td>The time stamp, incremented each time an interrupt is serviced. The value overflows and repeats ≈ 50 days if sampling at 1KHz.</td>
</tr>
<tr>
<td>lDataCount</td>
<td>The number of data points in Params.</td>
</tr>
<tr>
<td>alParams</td>
<td>An array for both input data and returned data.</td>
</tr>
<tr>
<td>alBuffer</td>
<td>Another array for returned data that has a one-to-one correspondence with the alParams data array.</td>
</tr>
</tbody>
</table>

Depending on the command being issued, only the required command structure elements need be declared. For example, the *ZERO_ENC_ONE* driver command requires that only the *nAxis* element be specified, while a command such as *SET_I_GAIN_ONE* needs both the *nAxis* and *lParams* elements defined. Once the specific structure elements for a command have been setup, the command can be sent to the driver via the *SendStg*( ) call which performs the desired action.

The STG driver commands include a bevy of useful commands for setting and obtaining values of encoder positions and analog voltages. For a complete list of commands please see [45].

The driver commands may also be invoked using the library file provided by Servo To Go Inc. The interface to the library can be found in the *Stg_io.h* file located in the *Stgconap* example provided by the manufacturer. The library implemented only a few of the most important and useful driver commands and its purpose was to eliminate the need for declaring *IoCtlBuf* structures every time a command was issued. The driver commands were simply called very much like standard C++ functions with the appropriate arguments. However, during software tests it was discovered that the C interface in *Stg_io.h* does not support multi-thread operations. For this reason, only the stgdrvr commands are used when threading is required.
5.1.5 Signal Transition Box

The last project component designed and implemented in this signal transition box. Referring back to Figure 5.1 at the beginning of this chapter, in order to work properly, the SDSU robot hand and the STG board must communicate with each other. This communication is accomplished through the transition box. The transition box is the central hub where all signals/cables from the STG board and the SDSU robot hand are relayed and sent to their appropriate channels. The transition box performs the following duties:

1. Receives analog voltages for each axis from the STG board.
2. Relays voltage values to amplifier for amplification.
3. Receives encoder counts for each axis from the SDSU robot hand which it sends to the STG board for software motor position monitoring.

The transition box communicates with the other components as follows: a DB25 cable is used to send analog voltages to the amplifier, a DB50 cable is used to send and receive encoder counts from the SDSU robot hand, and lastly, flat cables are used to connect P1, P3, and P4 on the transition box to P1, P3, and P4 on the STG board respectively. These connections are shown in Figure 5.18.

Figure 5.18. Transition box connection to robot hand, amplifier and STG board.
As mentioned previously, P3 originates from the STG board and is responsible for joint axis 0, 1, 2, and 3. It must receive encoder counts from the robot hand (via the DB50 cable), and apply the proper motor position corrects via analog voltage outputs (the DB25 cable). In a similar fashion, P4 must receive the encoder signals from the DB50 cable connected to the robot hand and send voltage values to the amplifier via the DB25 cable. P4 controls joint axis 4 and 5 of the thumb. Lastly, the purpose of P1 is to supply the 5V necessary to power all the encoders. Thus, the pins from the STG connector, the DB50 and DB25 cables must all be correctly connected to ensure the right signals are delivered and received. The image in Figure 5.19 shows the inside wiring of the transition box. Pin specifications for P1, P3, and P4 have already been given in Section 5.1.4.1. The diagrams in Table 5.6 and Table 5.7 show the pin specifications for the DB50 and DB25 connector cables respectively, while Table 5.8 and Table 5.9 show the ground connections for all the components. Lastly, Table 5.10 provides the important overall connection diagram for all joints.

Figure 5.19. Close up image of pin connections of transition box.
Table 5.6. DB50 connector cable pin-out specifications.

<table>
<thead>
<tr>
<th>pin</th>
<th>Signal</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AGND</td>
<td>common ground for A/D converter</td>
</tr>
<tr>
<td>2</td>
<td>FSR0</td>
<td>touch sensors</td>
</tr>
<tr>
<td>3</td>
<td>FSR1</td>
<td>touch sensors</td>
</tr>
<tr>
<td>4</td>
<td>FSR2</td>
<td>touch sensors</td>
</tr>
<tr>
<td>5</td>
<td>FSR3</td>
<td>touch sensors</td>
</tr>
<tr>
<td>6</td>
<td>FSR4</td>
<td>touch sensors</td>
</tr>
<tr>
<td>7</td>
<td>FSR5</td>
<td>touch sensors</td>
</tr>
<tr>
<td>8</td>
<td>FSR6</td>
<td>touch sensors</td>
</tr>
<tr>
<td>9</td>
<td>FSR7</td>
<td>touch sensors</td>
</tr>
<tr>
<td>10</td>
<td>FSR8</td>
<td>touch sensors</td>
</tr>
<tr>
<td>11</td>
<td>FSR9</td>
<td>touch sensors</td>
</tr>
<tr>
<td>12</td>
<td>FSR10</td>
<td>touch sensors</td>
</tr>
<tr>
<td>13</td>
<td>FSR11</td>
<td>touch sensors</td>
</tr>
<tr>
<td>14</td>
<td>FSR12</td>
<td>touch sensors</td>
</tr>
<tr>
<td>15</td>
<td>FSR13</td>
<td>touch sensors</td>
</tr>
<tr>
<td>16</td>
<td>FSR14</td>
<td>touch sensors</td>
</tr>
<tr>
<td>17</td>
<td>FSR15</td>
<td>touch sensors</td>
</tr>
<tr>
<td>18</td>
<td>+12V</td>
<td>power supply for conditioning circuits on hand</td>
</tr>
<tr>
<td>19</td>
<td>-12V</td>
<td>power supply for conditioning circuits on hand</td>
</tr>
<tr>
<td>20</td>
<td>GND</td>
<td>power supply for conditioning circuits on hand</td>
</tr>
<tr>
<td>21-27</td>
<td>reserved for force sensors</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>LS1</td>
<td>limit switch - small finger</td>
</tr>
<tr>
<td>29</td>
<td>LS2</td>
<td>limit switch - ring finger</td>
</tr>
<tr>
<td>30</td>
<td>LS3</td>
<td>limit switch - middle finger</td>
</tr>
<tr>
<td>31</td>
<td>LS4</td>
<td>limit switch - index finger</td>
</tr>
<tr>
<td>32</td>
<td>LS5</td>
<td>limit switch – thumb</td>
</tr>
<tr>
<td>33</td>
<td>LSCOM</td>
<td>limit switch – common</td>
</tr>
<tr>
<td>34</td>
<td>M1B</td>
<td>motor 1 encoder signal B</td>
</tr>
<tr>
<td>35</td>
<td>M1A</td>
<td>motor 1 encoder signal A</td>
</tr>
<tr>
<td>36</td>
<td>M2B</td>
<td>motor 2 encoder signal B</td>
</tr>
<tr>
<td>37</td>
<td>M2A</td>
<td>motor 2 encoder signal A</td>
</tr>
<tr>
<td>38</td>
<td>M3B</td>
<td>motor 3 encoder signal B</td>
</tr>
<tr>
<td>39</td>
<td>M3A</td>
<td>motor 3 encoder signal A</td>
</tr>
<tr>
<td>40</td>
<td>M4B</td>
<td>motor 4 encoder signal B</td>
</tr>
<tr>
<td>41</td>
<td>M4A</td>
<td>motor 4 encoder signal A</td>
</tr>
<tr>
<td>42</td>
<td>M5B</td>
<td>motor 5 encoder signal B</td>
</tr>
<tr>
<td>43</td>
<td>M5A</td>
<td>motor 5 encoder signal A</td>
</tr>
<tr>
<td>44</td>
<td>M6B</td>
<td>motor 6 encoder signal B</td>
</tr>
<tr>
<td>45</td>
<td>M6A</td>
<td>motor 6 encoder signal A</td>
</tr>
<tr>
<td>46</td>
<td>GND</td>
<td>encoder ground</td>
</tr>
<tr>
<td>47</td>
<td>5V</td>
<td>encoder power</td>
</tr>
<tr>
<td>48-50</td>
<td>unused</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.7. DB25 connector cable pin-out specifications.

<table>
<thead>
<tr>
<th>Pin #</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GND</td>
</tr>
<tr>
<td>2</td>
<td>Axis 0</td>
</tr>
<tr>
<td>3</td>
<td>Axis 1</td>
</tr>
<tr>
<td>4</td>
<td>Axis 2</td>
</tr>
<tr>
<td>5</td>
<td>Axis 3</td>
</tr>
<tr>
<td>6</td>
<td>Axis 4</td>
</tr>
<tr>
<td>7</td>
<td>Axis 5</td>
</tr>
<tr>
<td>8</td>
<td>Axis 6</td>
</tr>
<tr>
<td>9</td>
<td>Axis 7</td>
</tr>
<tr>
<td>10</td>
<td>Manual On</td>
</tr>
<tr>
<td>11</td>
<td>n/c</td>
</tr>
<tr>
<td>12</td>
<td>+12 V</td>
</tr>
<tr>
<td>13</td>
<td>n/c</td>
</tr>
<tr>
<td>14</td>
<td>n/c</td>
</tr>
<tr>
<td>15</td>
<td>n/c</td>
</tr>
<tr>
<td>16</td>
<td>n/c</td>
</tr>
<tr>
<td>17</td>
<td>n/c</td>
</tr>
<tr>
<td>18</td>
<td>n/c</td>
</tr>
<tr>
<td>19</td>
<td>n/c</td>
</tr>
<tr>
<td>20</td>
<td>n/c</td>
</tr>
<tr>
<td>21</td>
<td>n/c</td>
</tr>
<tr>
<td>22</td>
<td>n/c</td>
</tr>
<tr>
<td>23</td>
<td>n/c</td>
</tr>
<tr>
<td>24</td>
<td>n/c</td>
</tr>
<tr>
<td>25</td>
<td>n/c</td>
</tr>
<tr>
<td>26</td>
<td>n/c</td>
</tr>
</tbody>
</table>
### Table 5.8. Analog ground connections for Transition box.

<table>
<thead>
<tr>
<th>Device</th>
<th>Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB25 Cable</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>Axis 0: 1</td>
</tr>
<tr>
<td></td>
<td>Axis 1: 7</td>
</tr>
<tr>
<td></td>
<td>Axis 2: 6</td>
</tr>
<tr>
<td></td>
<td>Axis 3: 12</td>
</tr>
<tr>
<td>P4</td>
<td>Axis 4: 1</td>
</tr>
<tr>
<td></td>
<td>Axis 5: 7</td>
</tr>
</tbody>
</table>

### Table 5.9. Digital ground connections for Transition box.

<table>
<thead>
<tr>
<th>Device</th>
<th>Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB50 Cable</td>
<td>46</td>
</tr>
<tr>
<td>P1</td>
<td>50</td>
</tr>
<tr>
<td>P3</td>
<td>Axis 0: 13, 16</td>
</tr>
<tr>
<td></td>
<td>Axis 1: 22, 25</td>
</tr>
<tr>
<td></td>
<td>Axis 2: 31, 34</td>
</tr>
<tr>
<td></td>
<td>Axis 3: 40, 43</td>
</tr>
<tr>
<td>P4</td>
<td>Axis 4: 13, 16</td>
</tr>
<tr>
<td></td>
<td>Axis 5: 22, 25</td>
</tr>
</tbody>
</table>
### Table 5.10. Complete interconnection table for transition box.

<table>
<thead>
<tr>
<th>Component Description</th>
<th>Joint Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>DAC</td>
<td></td>
</tr>
<tr>
<td>EncoderInput – A</td>
<td>14</td>
</tr>
<tr>
<td>EncoderInput – B</td>
<td>17</td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
<tr>
<td>DAC</td>
<td></td>
</tr>
<tr>
<td>EncoderInput – A</td>
<td></td>
</tr>
<tr>
<td>EncoderInput – B</td>
<td></td>
</tr>
<tr>
<td>DB50</td>
<td></td>
</tr>
<tr>
<td>EncoderOutput – A</td>
<td>35</td>
</tr>
<tr>
<td>EncoderOutput – B</td>
<td>34</td>
</tr>
<tr>
<td>DB25</td>
<td></td>
</tr>
<tr>
<td>AnalogInput</td>
<td>2</td>
</tr>
</tbody>
</table>
The analog ground connections from Table 5.8 consist of the DB25 cable, pins on P3 and P4. Because the BRONCO computer is incapable of supplying the power necessary to operate the hand, the final voltage values it outputs must be amplified. These voltages are transmitted to the DB25 cable, which itself then sends them to the amplifier for amplification. P3 sends analog output voltages for axis 0, 1, 2 and 3. The analog grounds for each axis from Table 5.8 are pins: 1, 7, 6 and 12 respectively. Similarly, P4 outputs analog voltages for axis 4 and 5 with analog grounds for the axis on pins 1 and 7 respectively. Lastly, the DB25 cable transmits analog voltage values for amplification for all axis: 0, 1, 2, 3, 4 and 5. The DB25 pin 1 is the common ground for all these analog voltages.

The digital grounds are associated with the encoder signals. P1 supplies the 5V to power all encoders. From Table 5.9, pin 50 on P1 serves as its digital ground. The 5V from P1 must be transmitted along the DB50 which is directly connected to the robot hand and receives encoder count signals from it. Pin 46 on the DB50 cable serves as the encoder ground. Lastly, as discussed in section 5.1.4.1, each axis has two encoder channels A+ and B+. These must also be grounded. For example from Table 5.9, axis 0 channel A+ has digital ground on pin 13 and B+ channel ground on pin 16. The remaining digital ground pins for the remaining axis on P3 and P4 are also shown in Table 5.9.

The image in Figure 5.19 shows both the digital and analog ground connections. The common digital ground for all components is labeled DGRND and the common analog ground is labeled AGR.

The final interconnection diagram from Table 5.10 outlines the connections among the various components as signals are transmitted to their appropriate destinations. An example will demonstrate how the pins are connected. The application program using the driver commands of the STG board invokes a routine in which an analog voltage is sent to a specific joint commanding the axis motor into motion. For this example, let the axis motor be joint 0 on P3. This analog voltage, or DAC, is sent along the flat P3 cable into the transition box. The DAC must be amplified naturally. The DAC is first relayed through the DB25 cable. Table 5.10 shows that for joint 0, the DB25 cable accepts analog input on its pin 2. This voltage value enters the amplifier, is increased in strength and sent to the robot hand causing finger motion for joint 0. Once finger motion has started, the encoder counts for that joint will also change in value. The DB50 cable which is used to transmit these
encoder counts back to the STG board and application program begins to receive encoder values. For joint 0, the encoder output of channel A+ is DB50 pin 35 and for channel B+ it is pin 34. Naturally, these encoder values must be retransmitted to their final destination, the STG board. P3 is responsible for joint 0 and accepts encoder input for channel A+ on pin 14 and B+ on pin 17. The rest of the axes work in a similar fashion. Thus, the complete cycle has now been demonstrated where output values (DAC) from the STG board reached their appropriate destinations and final input values (encoder counts) were sent back to it. As a final note, Table 5.10 and Table 5.6 show an inconsistency in pin-out connections of encoder outputs A+ and B+ for axis 4 and 5, the thumb’s flexion and rotation. It was discovered during hand testing that circuitry on the SDSU robot hand for joints 4 and 5 was cross-wired. A simple correction of re-crossing the pins for these encoder outputs on the transition box resolved the issue. Therefore, Table 5.10 should be taken as the final correct interconnection table for all the components.
CHAPTER 6

SYNERGETIC MAPPING

The software program developed in this project “EMGRobotHand” incorporates a type of motion termed “synergetic motion” to control the 6 DOF SDSU robot hand. Synergetic motion is an effective way of controlling dexterous hands with many control inputs. In effect, the motion of the joints is represented in terms of only two control inputs: opening the hand and closing the hand [7]. A limpid example would be a baseball player’s glove. The glove compresses the many degrees of freedom of the hand to produce only two motions that enable the player to open and close the glove.

The idea for synergetic motion in robotics was borrowed from that naturally observed in biological systems. In nature, synergies are commonly used to cause several degrees of freedom to act with a single purpose [46,47]. Functional dynamic behaviors such as flying and swimming all display synergistic phenomena wherein the complexity of a large number of muscles and nerves cooperate to produce a general desired action [48,49,50].

In robotics, the types of synergy seen are either inherent to the design of the system, or externally produced through programming [7]. In the first, the hardware devices themselves are implemented such that the motions of the device move in accordance to synergetic laws. These types of devices are generally very limited and very much like the initial robotic hands discussed in Chapter 2. Their design precludes them from having many degrees of freedom and thus restricts the range of motions they are capable of performing.

The second kind of synergetic motion is achieved through software methods. In these types, the hand is generally equipped with a large number of degrees of freedom, however, the degrees are usually compressed into a fewer number of inputs to produce hand grasping motions. This is accomplished through programming wherein parallel driver commands are sent to the joints commanding it to move to the desired position. Multi-threading is naturally involved and such systems are generally much more versatile since synergetic motions, as well as independent joint motions may be performed.
The type of synergy used in this project is implemented through computer software. Several algorithms have been proposed in the past to accomplish the compression of DOF. The algorithm used here is a synergetic mapping that establishes a relationship between the angle of the finger joints and the aperture of the robot hand [7]. Such a relationship will allow the coordinated movement of all finger joints for tests on several grasp modes. The mapping is a simple rational function that easily permits transformations from one space to another.

The mapping function used in this project creates a relationship between the joint angle $\theta_1$ of the finger joints and the aperture diameter $D$. The joint angle $\theta_1$ is taken to be the angle between the horizontal plane at the base of the finger connecting it with the palm of the hand and the proximal phalanx segment of the finger. The image in Figure 6.1 shows the measurement of the joint angle. The diameter $D$ is simply taken to be the diameter of the object, e.g. a ball or block. Since the hand will be tested on four different classes of objects each with three sub-cases: small, medium and large, what is desired is a mapping function of the form $\theta_j = f_j (m, D)$ where $j = 0, 1...5$ and $m = 1...4$ that maps the size and type of object to the correct joint angle.

\[ \text{Figure 6.1. Measurement of joint angle } \theta_1. \]

Derivation of such a mapping function can be quite a difficult task. However, Vuskovic and Marjanski have demonstrated that a first-order rational function is a good Approximation [7]. The function they have proposed is given below:
\[ D = \alpha_{m,j} \frac{\beta_{m,j} - \theta_j}{\gamma_{m,j} + \theta_j} \]

where \( m \) is the particular grasp mode, \( j \) is the joint number and \( D \) is the object diameter. The inverse is given as:

\[ \theta_j = a_{m,j} \frac{b_{m,j} - D}{c_{m,j} + D} \]

where,

\[ a_{m,j} = \gamma_{m,j} \]
\[ b_{m,j} = \alpha_{m,j} \frac{\beta_{m,j}}{\gamma_{m,j}} \]
\[ c_{m,j} = \alpha_{m,j} \]

are the approximation coefficients. Thus, in order to map a type and size of grasp to the joint angles, only the approximations coefficients need be calculated. These can be experimentally found by noting that the approximation function has a hyperbolic character [7]. If three points are chosen such that the joint angles \( \theta_j \) for all fingers are known for that grasp mode, and the diameter \( D_m \) of that grasp mode is also known, then the system of equations may be solved to yield the approximation coefficients. The image in Figure 6.2 shows hyperbolic nature of the approximation function. For this project, the three different points correspond to small, medium and large sizes of a particular grasp mode.

Figure 6.2. Aperture diameter relationship with joint angle for the approximation function.
Using the approximation function for the three points we have the following equations:

\[
\theta_1 (c_{m,1} + D_1) = a_{m,1} b_{m,1} - a_{m,1} D_1 \tag{6.1}
\]

\[
\theta_2 (c_{m,2} + D_2) = a_{m,2} b_{m,2} - a_{m,2} D_2 \tag{6.2}
\]

\[
\theta_3 (c_{m,3} + D_3) = a_{m,3} b_{m,3} - a_{m,3} D_3 \tag{6.3}
\]

Taking equation (6.1) and solving for \( \theta_1 \) gives:

\[
\theta_1 = a_{m,1} D_1 + \beta_{m,1} \theta_1 D_1 + \gamma_{m,1}
\]

where,

\[
\alpha_{m,1} = \frac{-a_{m,1}}{c_{m,1}}
\]

\[
\beta_{m,1} = \frac{-1}{c_{m,1}}
\]

\[
\gamma_{m,1} = \frac{a_{m,1} b_{m,1}}{c_{m,1}}
\]

In a similar fashion, equation (6.2) and (6.3) may also be solved for the remaining two object sizes. The system of three equations can be expressed in matrix form as:

\[
\begin{bmatrix}
\theta_1 \\
\theta_2 \\
\theta_3
\end{bmatrix} =
\begin{bmatrix}
D_1 & \theta_1 D_1 & 1 \\
D_2 & \theta_2 D_2 & 1 \\
D_3 & \theta_3 D_3 & 1
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\beta \\
\gamma
\end{bmatrix}
\]

which can be solved for coefficients \( \alpha, \beta, \) and \( \gamma \):

\[
\begin{bmatrix}
\alpha \\
\beta \\
\gamma
\end{bmatrix} =
\begin{bmatrix}
D_1 & \theta_1 D_1 & 1 \\
D_2 & \theta_2 D_2 & 1 \\
D_3 & \theta_3 D_3 & 1
\end{bmatrix}^{-1}
\begin{bmatrix}
\theta_1 \\
\theta_2 \\
\theta_3
\end{bmatrix}
\]

and finally, the original coefficients \( a_{m,j}, b_{m,j}, \) and \( c_{m,j} \) become:
Thus, a framework has been developed for training the synergetic motions of the robot hand. First, a grasp mode is chosen, e.g. spherical grasp. Then three different objects of that class are chosen to represent the small, medium and large cases. The diameters of the three objects are measured, and these correspond to the parameters $D_3$, $D_2$, and $D_1$ in equations (6.1), (6.2) and (6.3) respectively. For each object, the joint angles $\theta_j$ for all fingers is measured such that the hand forms a snug fit around the object. Now, each separate joint has three different angles corresponding with the three size cases. Lastly, for each joint the same process is used to find the approximation coefficients; each finger joint requires calculation of its individual approximation coefficients. These can then be used with the mapping function to obtain the joint angle for a specific finger given an object diameter.

Once this process has been repeated for all degrees of freedom, the hand can be coordinated to move into a particular grasp once the grasp mode and object diameter have been specified. Thus, the robot hand can act with synergetic motion wherein the 6 DOF have been compressed into only two inputs: grasp mode and object diameter.

\[
c_{m,j} = \frac{-1}{\beta_{m,j}}
\]

\[
a_{m,j} = \frac{\alpha_{m,j}}{\beta_{m,j}}
\]

\[
b_{m,j} = \frac{\gamma_{m,j}c_{m,j}}{a_{m,j}}
\]
CHAPTER 7

SYSTEM INTEGRATION AND CALIBRATION

The EMG Robot Hand program developed for this project was used in conjunction with the EMG Classifier Program developed by Chris Miller. The EMG Robot Hand program consists of two files, both identically called *EMGRobotHand.exe*. One portion of the program (the server) runs on the computer named SKIPPER and serves as the user interface, while the other (the client) runs on the BRONCO computer and accesses directly the driver commands of the STG board. As discussed previously, the Classifier program operates on an entirely separate computer (i.e., SKIPPER) from the one installed with the STG board and drivers (i.e., BRONCO). Thus, in order to provide communication between Miller’s program and the software developed here, a workaround had to be devised. The method chosen was to connect both computers using a crossover cable. A review of Figure 5.1 demonstrates this clearly. With this link, communication was easily achieved with Windows Sockets using TCP/IP packets.

The EMG Robot Hand server program running on SKIPPER operates as a user interface that obtains the desired commands from the user or from the Classifier Program. These commands are then communicated through the crossover cable via Windows sockets to the BRONCO computer. The EMG Robot Hand client program running on the BRONCO computer receives these commands and calls the appropriate driver functions to carry out the specified actions. Furthermore, because the Classifier program and the EMG Robot Hand user interface execute on the same machine, they are also able to communicate with each other in real-time. This communication is accomplished through an output text file which can be accessed by both programs. In this manner, the classifier program may write specific information about a certain grasp to the file which the robot hand user interface program may afterwards read. After this step, the information can be relayed to the EMG Robot Hand client program on the BRONCO machine to have the hand move in real-time in response to actions performed using the Classifier Program. Figure 7.1 and Figure 7.2 are the runtime diagrams that outline the process described above.
Figure 7.1. Flow chart diagram showing processes and actions performed on SKIPPER.

Figure 7.2. Flow chart diagram for processes on BRONCO.
The EMG Robot Hand program commands allow the hand to move into a myriad of grasp positions. For this project, only spherical, cylindrical, precision, and lateral grasps were employed for real-time movement. This follows directly from the four grasp classifications tested in Miller’s program. The hand was trained for each of the grasp modes using the approximation function technique outlined in Chapter 6. That method required three object diameters, $D_3$, $D_2$, and $D_1$ from which the joint angles for all fingers can be measured. Three different sized objects are chosen for each grasp mode with the same process repeated for the four categories. With this, the approximation coefficients can be calculated and using the approximation function the angle for the proximal phalanx finger segment can be found. The hand can then be ordered to move into a specified position given a specific object and object diameter. The images in Figure 7.3 through Figure 7.6 show the objects used to train the robot hand for synergetic grasping.

**Figure 7.3. Spherical grasp objects and sizes.**

**Figure 7.4. Precision grasp objects and sizes.**
The exact specifics of the EMGRobotHand user interface program will be outlined shortly, but for now it is important to understand that prior to any robot grasping, the hand is first sent into home position. Home position corresponds to all fingers pointing skyward at their near vertical position limits (90°). This rest position will later be used for real-time movement using the Classifier program. The images in Figure 7.7 through Figure 7.30 show the hand performing each of the grasp modes: cylindrical, spherical, lateral, and precision for the three object sizes used for training.
Figure 7.7. Robot hand grasping small precision: Garbanzo bean diameter 9mm.

Figure 7.8. Precision grasp of Garbanzo bean. Object diameter 9mm.
Figure 7.9. Robot hand grasping medium precision: Cap diameter 16mm.

Figure 7.10. Precision grasp of Cap. Object diameter 16mm.
Figure 7.11. Robot hand grasping large precision: Nut diameter 28mm.

Figure 7.12. Precision grasp of Nut. Object diameter 28mm.
Figure 7.13. Robot hand grasping small cylinder: Tool case diameter 28mm.

Figure 7.14. Cylindrical grasp of Tool case. Object diameter 28mm.
Figure 7.15. Robot hand grasping medium Cylinder: Pepper shaker diameter 42mm.

Figure 7.16. Cylindrical grasp of Pepper shaker. Object diameter 42mm.
Figure 7.17. Robot hand grasping large cylinder: Plastic bottle diameter 56mm.

Figure 7.18. Cylindrical grasp of Plastic bottle. Object diameter 56mm.
Figure 7.19. Robot hand grasping small spherical: Plastic ball diameter 39mm.

Figure 7.20. Spherical grasp of Plastic ball. Object diameter 39mm.
Figure 7.21. Robot hand grasping medium spherical: Racquetball diameter 60mm.

Figure 7.22. Spherical grasp of Racquetball. Object diameter 60mm.
Figure 7.23. Robot hand grasping large spherical: Softball diameter 81mm.

Figure 7.24. Spherical grasp of Softball. Object diameter 81mm.
Figure 7.25. Robot hand grasping small lateral: Block diameter 6mm.

Figure 7.26. Lateral grasp of small block. Object diameter 6mm.
Figure 7.27. Robot hand grasping medium lateral: Block diameter 19mm.

Figure 7.28. Lateral grasp of medium block. Object diameter 19mm.
Figure 7.29. Robot hand grasping large lateral: Block diameter 25mm.

Figure 7.30. Lateral grasp of large block. Object diameter 25mm.
Once training had concluded, the approximation coefficients for each grasp mode were hard coded into the EMG Robot Hand client program. As mentioned previously, the client program on the BRONCO machine directly invokes the STG driver commands. The hand is then ordered into its home rest position. It is now ready to accept real-time EMG signals from the Classifier program.

Next, the steps in Section 4.1.3 of Chapter 4 were used for electrode placement and initialization of the Classifier program. The Classifier program was then executed using Miller’s recommendation of 18 repetitions for each grasp for training. Additional grasps may be performed for increased classifier accuracy. The classifier was trained using the same set of objects utilized in [9], namely: a small cap, a tennis ball, a cup, and a key inserted in a box. These objects correspond to the four grasp modes tested in this project: point, spherical, cylindrical and lateral. Figure 7.31 shows these training and test objects.

![Figure 7.31. EMG Classifier and real-time test objects: a key, a cup, a tennis Ball and a small cap.](image)

The training data is saved and the Classifier determines the optimal time parameter and feature to use for classification. This may be either the Waveform Length or Spectral Moment feature depending on which offers the optimal classification rate with the
lowest associated time period. For an in depth analysis of feature optimization and selection
during classification, please see [9]. The ‘w’ command on the EMG Robot Hand server
program was then executed. This command readies the robot hand to receive EMG signal
data as soon as it is available by the Classifier Program. The ‘w’ corresponds to *Wait for
EMG Signal*. Next, the *Classify Single Grasp* procedure was selected on the Classifier. The
program now prompts the user to perform any of the four grasps. A grasp consists of the
user grasping any of the four objects in Figure 7.29. As soon as enough data is received, the
Classifier will perform real-time classification of the grasp and update its *Classification
window* to indicate the just performed grasp. Furthermore, the EMG Classifier was modified
to write to a text file, *output1.txt*, the results of its classification. The information is an
integer value corresponding to the four grasp modes: 1 for cylindrical, 2 for spherical, 3 for
point and 4 for lateral. Upon updating *output1.txt*, the information is immediately retrieved
by the EMG Robot Hand program whose ‘w’ command continuously checks for updates to
the file. The hand then executes the corresponding grasp just performed by the user
achieving real-time user controlled grasp movement. Note that for the purposes of this
project, the robot hand movement corresponding to the user’s performed grasp is the grasp
position most clearly identifiable as being in the same object space as that of the user. For
example, for spherical grasp the robot hand is ordered to move into a grasp position
 corresponding to the small plastic ball of object diameter 39 mm as shown in Figure 7.17.
This was chosen to distinguish it more clearly from cylindrical grasp positions which share
many similarities of finger positions.

Lastly, it is important to note that the success of the robot hand’s real-time movement
depends critically on the classification rate of the Classifier. The hand is simply a messenger
relaying the information it receives in real-time from the Classifier. Thus, for this project,
only classification rates in the 95 percentile or above were chosen for use. If the initial
classification rate falls below this threshold value, it is suggested that additional grasp
repetitions be performed by appending data to the saved trained data file.

The images in Figure 7.32 through Figure 7.35 show a sample run of the real-time
user controlled robot hand using the Classifier’s *Classify Single Grasp* procedure. The
results of the Classifier along with the robot hand’s grasp position are shown to demonstrate
the correspondence of their positions.
Figure 7.32. Real-time Point grasp of robot hand using the EMG Classifier Program. Note the correspondence of classification with robot grasp.

Figure 7.33. Real-time Spherical grasp of robot hand using EMG Classifier Program. Note the correspondence of classification with robot grasp.
Figure 7.34. Real-time Cylindrical grasp of robot hand using EMG Classifier Program. Note the correspondence of classification with the robot grasp.

Figure 7.35. Real-time Lateral grasp of robot hand using EMG Classifier Program. Note the correspondence of classification with robot grasp.
CHAPTER 8

EMG ROBOT HAND PROGRAM

The EMG Robot Hand program essentially integrates all system components into an experimental real-time system and provides a user command interface. The program works in conjunction with the Classifier program to create a framework that allows a user to perform a specific grasp and have the robot hand mimic the grasp mode in real-time. Aside from real-time grasp motions, the program also allows the user to position the hand into a myriad of grasps using various functions.

The following sections describe the components of the EMG Robot Hand program. Specifically, all of the available commands are described in detail and samples of their output given. The chapter concludes with a short tutorial on how to execute the program for testing of real-time grasp movements and discusses the implementation of the program and the various files associated with it.

8.1 PROGRAM COMPONENTS

As previously mentioned, the EMG Robot Hand program developed in this project actually consists of two files both identically named. The primary file resides on the SKIPPER machine and is the main point of interaction between the hand and the user. The secondary file resides on the BRONCO machine and serves solely to relay the commands issued by the primary file. It does so by invoking the STG driver commands available only to it (the STG board and driver are only available to BRONCO). The primary file may also be referred to as the server, and the secondary file as the client. The server and client communicate with each other via a red crossover cable that connects both computers. The actual sending of information was achieved with Windows Sockets using TCP/IP packets. To use this method, the server computer’s IP address is manually set to a specific value and a port chosen as the channel of communication. The client simply connects to the same address and port and now both computers are open to send and receive information from each other. The values for the TCP/IP address and port number are given below:
IP address: 192.168.1.10
Subnet mask: 255.255.255.0
Default gateway: 192.168.1.1
Preferred DNS Sever: 130.191.1.1
Alternate DNS Server: 130.191.200.1

and the port number used is 9974. Clearly, in order for a connection to be made, the server must be running first before the client program is executed. The image in Figure 8.1 shows a successful connection of the EMG Robot Hand client program to the server program.

![Server: The connection socket is successfully created. Waiting for client... Server: A connection has been accepted from a client.](image)

**Figure 8.1. Status message shown upon successful connection of EMG Robot client program with server program.**

Once both programs are successfully connected, the EMG Robot Hand user interface display comes up. The display gives a brief overview of the commands available to the hand. The EMG Robot Hand user interface is shown in Figure 8.2.

![PID Controller for Robot Hand
Enter the letter of the corresponding command
n - Relative motion n <joint> <value>
g - Absolute motion g <joint> <value>
h - Reset all axes to home position
w - Wait for EMG
O - Object Space <grasp mode> <aperture>
Grasp Mode
1 - Cylindrical
2 - Spherical
3 - Point
4 - Lateral
d - Display all position values
c - Calibrate
u - Show Menu
x - Exit
Command:](image)

**Figure 8.2. EMG Robot Hand user interface with the list of available commands.**

There are a total of nine commands that may be chosen. The first command that must always be selected prior to execution of all others is the calibration command ‘c’. Each of the commands listed in Figure 8.2 and their functionality will now be described in detail.
8.1.1 Calibrate

The command ‘c’ is the calibrate command. This is the first command that should be selected once the menu interface of Figure 8.2 has appeared. The calibrate command ensures that the joint angle movements of the robot hand correspond as closely as possible to the encoder counts of each joint. It does this by measuring the total encoder counts of a joint from its near vertical limit to a position approaching the palm of the hand. Thus, each finger sweeps out an area from a completely open position to a completely closed position. Once the encoder counts have been found, the counts per angle can be calculated by measuring the same angle swept by the finger’s movement from open to close. The angles for each finger correspond to the angle swept by a joint’s proximal phalanx finger segment as described in section 6.1. The angles for each joint were found manually by measuring them using a protractor and thus are approximations.

Once calibration is complete, the hand is sent to its home position. The home position of the hand corresponds to one in which all joints are completely open. Because the encoder counts per angle have been saved, the hand is now able to receive the other joint angle commands available to the hand.

As a final note, it is important during prolonged operation of the EMG Robot Hand program that the hand be calibrated occasionally. During prolonged use, the joint motors and encoders tend to “loosen up” and calibration maintains the accuracy between the encoder counts and angles. Once calibration has concluded, any of the remaining commands may now be selected.

8.1.2 Absolute Motion

The absolute motion command is selected by entering the keyword ‘g’ at the command prompt. The exact syntax of the command is:

\[ g \ <\text{joint}> \ <\text{value}> \]

where \text{joint} is a number from 0 to 5 and \text{value} is the desired position of the joint in degrees. The command orders a finger to a specific angle position given by the \text{value} variable. The angle entered cannot be greater than the maximum upper limit or lower than minimum angle position. If a value outside these bounds is entered, the program displays an error message.
prompting the user to reenter the command. The upper and lower angle limits as described in section 8.1.1 were found manually and are approximations. Furthermore, they differ for each joint due to the position of the finger on the hand and the mechanics of the linkage segments connecting the finger with its motor. Table 8.1 shows the measured minimum and maximum joint angles for each finger.

Table 8.1. Maximum and minimum joint angles of each finger.

<table>
<thead>
<tr>
<th>Joint Number</th>
<th>Minimum Angle°</th>
<th>Maximum Angle°</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>81</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>87</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>88</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>42</td>
</tr>
</tbody>
</table>

The angle value must lie in the range shown in Table 8.1 for the associated joint chosen. The image in Figure 8.3 provides an example of the absolute motion command and the resultant motion of the robot hand. As discussed in Chapter 6, the angle is defined as the angle created by the line along the proximal phalanx finger segment and the horizontal plane with the vertex being the junction of the two lines at the base of the finger.

Figure 8.3. Sample absolute motion command for joint=0 and angle = 45°.
8.1.3 Relative Motion

The relative motion command operates in a similar fashion to that of the absolute motion command. As the name implies, the relative motion command moves the chosen joint to a position given by the sum of its current position and the angle value entered during execution of the command. The syntax for the command is

\[ m \ <\text{joint}> \ <\text{value}> \]

where \textit{joint} is a number from 0 to 5 and \textit{value} is the new displacement angle. A positive value causes motion of the finger to the open position while a negative value causes motion toward the palm of the hand. The final joint angle position of the joints must also lie within the bounds outlined in Table 8.1. Thus if the angle \textit{value} entered causes the absolute angle position of the joint to exceed the allowable limits, the program returns an error message alerting the user of the mistake and displaying the permissible absolute angle for that joint.

8.1.4 Object Space

The object space command causes synergetic motion of the hand to one of the four grasp modes: cylindrical, spherical, point or lateral. The exact syntax of the command is

\[ o \ <\text{grasp mode}> \ <\text{aperture}> \]

where \textit{grasp mode} is the grasp type and \textit{aperture} is the object diameter in millimeters. The image in Figure 8.4 provides an example of the use of the object space command.

![Figure 8.4. Sample object space command of point grasp diameter 9mm.](image)
The example in Figure 8.4 shows that the object space command was issued with the Point grasp mode selected (this corresponds with the integer value 3) and finally, the diameter of the point object to be grasped is 9mm. The *grasp mode* can be an integer value from one to four. The menu from Figure 8.2 above specifies which integer values correspond to which grasp modes. Lastly, the *aperture* value varies depending on the type of grasp mode chosen.

### 8.1.5 Display

The command corresponding to entering only the keyword ‘*d*’ is the display command. The display command shows the user the current position values for all of the joints. The values shown are in degrees. The command is useful for determining the exact finger locations required to grasp different objects. This can be done by first issuing absolute motion commands to the joints until the tested object is firmly grasped. The angle values can then be displayed and saved for future. Figure 8.5 provides an example of the display command. The ‘*d*’ command was entered immediately after issuing the object space command of Figure 8.4 above. Thus, the position of the joints for the Point grasp with an object of diameter 9mm are shown.

![Command: o 3 9
Command: d
Axis: 0 Current Angle 86
Axis: 1 Current Angle 81
Axis: 2 Current Angle 47
Axis: 3 Current Angle 44
Axis: 4 Current Angle 88
Axis: 5 Current Angle 9](image)

**Figure 8.5.** Sample Display command showing all joint angles for the hand in a Point grasp mode.

### 8.1.6 Reset

The reset to home position command is issued by the keyword ‘*h*’ at the prompt. The reset command moves all of the fingers back to their home positions, namely, the open hand position in which all finger joints are pointing at the near vertical limits as defined by their maximum angles as given in Table 8.1. The reset position should be issued prior to any joint movements since it provides consistency of encoder and angle measurements.
8.1.7 Show Menu

Entering the keyword ‘u’ at the prompt corresponds to the show menu command. This command brings up the overview of all available commands as shown previously in Figure 8.2. The command is useful for redisplaying the list of all commands since continuous usage of the EMG Robot Hand program tends to clutter the screen. The EMG Robot Hand program itself has an inherent show menu function that clears the screen and redisplay the general menu again after five consecutive commands have been issued. However, if the user wishes to bring up the overall menu prior to that, this command is included so as to provide for that capability.

8.1.8. Wait for Emg

The Wait for EMG command is perhaps the most important command in the EMG Robot Hand Program. This is the only command that allows the user to control the grasp motions of the robot hand in real-time. The command is selected by entering ‘w’ at the prompt. Once selected, the program enters a continuous loop that waits for EMG signal data to be sent to it from the Classifier program. Therefore, it is important that this command only be selected after the Classifier has been trained and the user has begun real-time grasp classification by choosing the Classify Single Grasp option of the Classifier program. Failure to follow the above steps leaves the program in an infinite loop that can only be stopped by manually terminating the EMG Robot Hand server and client programs. A screenshot of the Wait for EMG command is shown in Figure 8.6. The specifics on how to utilize this command will be discussed shortly in the tutorial section.

Figure 8.6. The Wait for EMG signal command is shown waiting for real-time data to be received from the EMG Classifier Program.

8.1.9 Exit

The final command available is the Exit command. It is selected by entering ‘x’ at the prompt. This command should always be used to end the EMG Robot Hand Program. It
is important because the command successfully terminates both the EMG Robot Hand server and client programs on the SKIPPER and BRONCO computers. Failure to use this command to close the programs may result in remnant and stray voltage signals being sent to the hand even after the programs have been closed. This could potentially lock the joint motors and fingers requiring a manual fix using pliers. For these reasons, only this command should be used to close sessions of the EMG Robot Hand.

**8.2 EMG ROBOT HAND TUTORIAL**

In order to successfully perform EMG driven control of the robot hand in real-time, a specific sequence of events must be executed. The following steps detail the steps required to successfully run and terminate the EMG Robot Hand Program for real-time use with the Classifier program.

1. Attach all electrodes to the forearm and activate the EMG amplifier by connecting it to the SKIPPER machine. These steps were described in Chapter 4.
2. Execute the `EMGProgGUIFC.exe` on the SKIPPER computer. This is the EMG Classifier Program.
3. Select *Train* and begin training the Classifier. This process takes about 20-30 minutes.
4. Save the trained data. The trained Classifier will be used for real-time control.
5. Now execute the `EMGRobotHand.exe` program on the SKIPPER machine. This is the server program and must be run first.
6. Next, run the `EMGRobotHand.exe` program on the BRONCO machine. This is the client program. The two programs should now be connected to each other.
7. Enter ‘c’ at the command prompt. The hand should now be ready and calibrated.
8. Select *Load Classifier* on the EMG Classifier and load the trained data.
9. Enter ‘w’ at the command prompt of the EMG Robot Hand. The hand is now ready to receive information from the Classifier.
10. Select *Classify Single Grasp* on the EMG Classifier. A prompt will appear asking the user to perform a grasp. Once a grasp has been successfully performed and analyzed, the EMG Classifier will send the information to the hand resulting in its movement to the same grasp.
11. Repeat Steps 9-10 above as desired for continued real-time control of the robot hand. Otherwise, select other commands on the EMG Robot Hand program to test other grasp positions.
12. To exit, enter ‘x’ at the command prompt. This successfully ends both server and client EMG Robot Hand programs.
13. Lastly, exit the EMG Classifier program to end the session.

8.3 PROGRAM IMPLEMENTATION

The EMG Robot Hand program was created using Microsoft Visual C++ 6.0. The program files were written in C++ and the executable program created using the Microsoft Visual C++ 6.0 compiler. The project also consists of the modified EMG Classifier program. That program was altered so as to communicate in real-time with the EMG Robot Hand program developed for this project. The modified EMG Classifier program was written in MATLAB using the MATLAB 7.0 GUI layout editor and the final executable was also created using the MATLAB compiler. All programs run using the Windows 2000 operating system (OS) and require that both the NI-DAQ card and the STG board be installed along with their associated drivers. Lastly, the target computers must also have MATLAB or the MATLAB component runtime (MCR) libraries installed.

Table 8.2 details all of the files used in this project. They consist of the files used for the modified EMG Classifier as well as the EMG Robot Hand files created in this project. The user is directed to [9] for a comprehensive summary and description of the use and purpose of the many MATLAB source files comprising the EMG Classifier. The MATLAB source code is included in Appendix A, while the C++ source files are shown in Appendix B and Appendix C.

8.3.1 EmgRobotHand.cpp Server

This file implements the user interface front-end for the EMG Robot Hand program. The program begins by opening a port for communication between it and the BRONCO computer. Windows Sockets are used for TCP/IP packet communication. The program receives the entered commands from the user and relays them to the client program whose direct access to the STG driver commands allows only it to issue motion commands to the hand. The program ensures that the user’s requested commands are performed by receiving signals from the client program that notify it of the successful completion of the designated command. This file also implements communication between the robot hand and the EMG Classifier. When the ‘w’ command is entered, the file enters a continuous loop that constantly checks for updated information being written to output1.txt.
Table 8.2. Source files for EMG Robot Hand and modified EMG Classifier Programs.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMGRobotHand.cpp</td>
<td>C++</td>
<td>GUI support functions for program</td>
</tr>
<tr>
<td>(SKIPPER)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMGRobotHand.cpp</td>
<td></td>
<td>Functions that invoke STG driver</td>
</tr>
<tr>
<td>(BRONCO)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>output1.txt</td>
<td></td>
<td>Contains integer values for real-time use</td>
</tr>
<tr>
<td>EMGProgGUIFC.m (modified)</td>
<td></td>
<td>Modified GUI for real-time control</td>
</tr>
<tr>
<td>EMGProgGUIFC.fig</td>
<td></td>
<td>GUI front end</td>
</tr>
<tr>
<td>Train.m</td>
<td></td>
<td>Generates S,C, and N parameters</td>
</tr>
<tr>
<td>LOOValidation.m</td>
<td></td>
<td>Classifies data for current classifier</td>
</tr>
<tr>
<td>BonatoOnset.m</td>
<td></td>
<td>Determines onset of movement of data</td>
</tr>
<tr>
<td>BonantoRealTime.m</td>
<td></td>
<td>Determines onset of movement real time</td>
</tr>
<tr>
<td>Features.m</td>
<td></td>
<td>Creates feature vector (spectral moments)</td>
</tr>
<tr>
<td>Moments.m</td>
<td></td>
<td>Computers spectral moments (time signal)</td>
</tr>
<tr>
<td>GenerateI.m</td>
<td>MATLAB</td>
<td>Generates I coefficients for moments</td>
</tr>
<tr>
<td>FeaturesWL.m</td>
<td></td>
<td>Creates feature vector (Waveform length)</td>
</tr>
<tr>
<td>FeaturesMAV.m</td>
<td></td>
<td>Creates feature vector from MAV</td>
</tr>
<tr>
<td>BoxCox.m</td>
<td></td>
<td>Performs Box-Cox transformation</td>
</tr>
<tr>
<td>Dflt_train.mat</td>
<td></td>
<td>Contains default train data</td>
</tr>
<tr>
<td>EMGDLL.mfile.m</td>
<td></td>
<td>DAQ functions MATLAB prototype file</td>
</tr>
<tr>
<td>EMGDLL.dll</td>
<td></td>
<td>DLL for DAQ card</td>
</tr>
<tr>
<td>DAQ_Functions.h</td>
<td>C</td>
<td>National Instruments DAQ header</td>
</tr>
<tr>
<td>DAQ_Functions.c</td>
<td></td>
<td>DAQ functions implementation</td>
</tr>
</tbody>
</table>

8.3.2 EmgRobotHand.cpp Client

This file implements all of the commands sent to it by the user interface server program just discussed. The file contains functions that perform the absolute, relative and object space motion for the robot hand. The approximation coefficients for all four grasp modes are implemented within as separate functions for each. The file also contains code for first connecting it to the server program to establish open communication between both programs. This file resides on the BRONCO computer and performs no output of any kind. It serves only to invoke the STG driver commands to accomplish the desired behavior as received from the server program and the user. The next section describes file used for communication between the various components of the program to achieve real-time control of the robot hand.
8.3.3 Output1.txt

This file serves as the communication link between the EMG Classifier and the EMG Robot Hand programs. The file is written to by the EMG Classifier with an integer value corresponding to the grasp mode just performed. The values are 1 for cylindrical, 2 for spherical, 3 for point and 4 for lateral. The file is created by the EMGProgGUIFC.m which is the main driver file for the EMG Classifier. The file is created upon entering the Classify Single Grasp mode in the Classifier program and receives its information as soon as the Classifier has classified the user’s grasp. The file is then read by the EMG Robot Hand program in Wait for EMG mode to achieve real-time motion control.

8.3.4 EmgProgGuifc.m

The main driver file for the EMG Classifier. This file contains calls to all the associated MATLAB support functions of Table 8.2. The file also contains the skeleton to create the EMG Classifier GUI and loads the dynamic link library file EMGDLL.dll which allows the Classifier to read data from the EMG device through the NI-DAQ card. This file calls the various support functions of Table 8.2 to train the classifier, read EMG data from the NI-DAQ card, classify EMG data using the classifier, and update the GUI classifier class displays. Furthermore, the file has been modified to communicate with the EMG Robot Hand program by writing information regarding the user’s grasp to the output1.txt. The file writes integer values to the text file when the Classify Single Grasp features has been selected and the program has finished classifying the EMG data of the grasp.

8.4 FILE EDITING AND UPDATES

The EMG Robot Hand programs were created using Microsoft Visual C++ 6.0. Both programs contain a project workspace file (.dsw) that facilitates and simplifies their editing for future improvements. The EMGRobotHand.dsw project workspace file contains all the needed libraries and driver files already installed to call the STG board driver commands. These libraries can be found by selecting the Project -> Settings -> Link -> Object/Library Modules option in the toolbar. The path to the library files must match their location within the EMG Robot Hand debug folder. The EMG Robot Hand server and client programs were implemented as Win32 Console Applications and their executables can be created using the
Microsoft Visual C++ compiler by first selecting the *Build -> Compile Project* option in the toolbar menu, then the *Build -> Build* executable option, and lastly *Build -> Run executable* to run the program. The location of the final executable may also be modified within the same project settings option. Upon opening the project workspace file, the main driver file *EMGRobotHand.cpp* is opened. The user may then edit the file and run the modified version using the build steps just described. These same steps apply to both the server and client versions of the *EMGRobotHand.cpp* file.

The *EMGRobotHand.cpp* server file on the SKIPPER machine also contains file reading code for receiving information from the Classifier program. It is important to remember that during editing this file path must match the location of *output1.txt*. The exact location of *output1.txt* is determined by the *EMGProgGUIFC.m* MATLAB file also on the same machine. These two file paths must match otherwise errors may occur when entering *Wait for EMG* mode.

The EMG Classifier program may also be similar modified. The program comes with its own MATLAB project file (.prj) that packages the program for ease of use. To edit the Classifier program, open the *EMGProgGUIFC.prj* file. Then, in the *Deployment Tool* window select Main Function and open the *EMGProgGUIFC.m* file. Editing this file allows modification of the GUI skeleton program of the Classifier and the final executable for the program. To build the executable, select *Tools -> Build* from the main menu toolbar. Other settings for the project such as the output directory can be edited by selecting the *.prj (Standalone Application)* folder in the *Deployment Tool* menu and modifying the location of the final executable. The accompanying support functions of Table 8.2 are simple m-files that may be also edited by opening them with MATLAB.

Because the *EMGProgGUIFC.m* file uses the accompanying support functions of Table 8.2, it is important that the m-files for all the functions be located in the same folder as the *EMGProgGUIFC.m* file. Furthermore, if the user wishes to alter the location of the *EMGProgGUIFC.exe*, then the following files must also be moved along with it to ensure they lie in the same folder: *key.gif, cylinder.gif, disk.gif, sphere.gif, rest.gif*. These are the Classification window images for the grasps and are required by the program to update its class display. The *EMGDLL.dll, EMGDLLmfile.m, EMG* exp file, *EMGClassifier* exp file and *EMGClassifier.dll* files must also be included. These files allow the executable to
properly read EMG data from the EMG amplifier device. The *dfl_train* m-file must also be included. This is the default trained data for the classifier.

Lastly, include the *EMGClassifier.ctf* and *EMGProgGUIFC.ctf* in the same directory as the executable. These are MATLAB archive files created during compilation of the project whose extraction may be needed to run the executable.
CHAPTER 9

RESULTS AND FUTURE WORK

The goal of this research was to create software that allows the SDSU robot hand to be controlled in real-time by a user either by direct motion commands or by using EMG signals received from a human subject. This project has successfully accomplished this task by developing the EMG Robot Hand server and client programs. These programs in conjunction with the modified EMG Classifier allow a user to control the robot hand in real-time to perform grasp motions of four modes: cylindrical, spherical, lateral and point. The real-time EMG Classifier program serves as the vehicle by which the user’s forearm and hand motions can be analyzed and classified. The data received from the user are surface EMG signals amplified with a specialized EMG amplifier device which contains the electrode attachments that are placed directly on the user’s forearm. The information contained in these EMG signals is first classified and then sent to the robot hand in real-time through file sharing. The software developed here can then retrieve the grasp information and command the robot hand to act accordingly.

The software developed here successfully implemented the real-time user controlled synergetic motions of the SDSU robot hand for the four grasp types: cylindrical, spherical, precision, and lateral. The program consists of two client-server process arrangements residing on separate computers. One serves as a user interface obtaining commands from the user while the other issues direct commands to the robot hand by invoking the motion controller interface board driver commands. The user interface EMG Robot Hand program can obtain grasp commands either directly from a user via the command prompt, or it may enter real-time mode in which a single grasp may be classified and performed by the robot hand. The four grasp modes used by this project were successfully implemented using the approximation function which performs the synergetic movements of the hand.

The real-time user controlled motions of the robot hand and the overall research of this project has demonstrated the feasibility and practicality of using EMG signals of the forearm to control a robot hand. Furthermore, this project has shown the great utility such
systems may have for disabled or crippled human subjects. Whereas earlier versions had limited versatility and range of motions, this project has shown that a highly dexterous six degrees of freedom hand can be adequately used for synergetic hand motions without restricting the number of degrees of freedom. Such versatility is precisely the type needed and required in the area of prosthetics. The overall goal being to develop adequate replacements for the human hand, it is critical that robot hand models maintain the same kind of functionality and adaptability seen in the natural world.

The research of this project has also revealed a few of the drawbacks and impediments that must be resolved if such an artificial model is ever to replace the natural one. Namely, the contact between the bipolar electrodes and the skin must be absolutely secure and the location of the electrodes themselves must be precise to ensure the correct EMG signals are being read. The Velcro system used here also revealed the fatiguing nature of having such a cumbersome device wrapped around the human forearm. Because training the classifier requires around 20 minutes and testing in real-time further prolongs the overall time, it is only natural for the forearm and blood vessels to feel strain and fatigue from excessive use of the EMG amplifier device. This is further exacerbated by the amount of pressure required to properly secure the electrodes, a problem made worse for thin and skinny subjects whose forearm muscles are not as clearly defined.

Furthermore, systems that hope to allow a human subject to control the motions of a robot hand must have some method of classifying the real-time EMG signals. This project used the EMG Classifier program. However, due to the nature of the classifier program, the first 200 ms are reserved and treated as rest position data and ignored during classification. In addition to this, the time sample used for feature extraction and classification is the 250 ms after the beginning of the data collection. Clearly, these quantities of time sum to create a significant delay between the user’s arm movements and the robot hand’s response. In addition to this, the type of information sharing used in this project consists of reading/writing to a shared file. This process can be refined by having the interprocess communication performed directly through TCP/IP packets, greatly improving speed.

The following are recommendations for future work: (a) Improve the human-electrode system used. Suggestions have already been made to develop wireless electrodes
integrated in an Ad-Hoc wireless sensory network. This would greatly minimize the strain and awkwardness of the current EMG amplifier device; (b) Improve the efficiency of the classification methods and procedures used. The time delays incurred during analysis and classification must be reduced as much as possible to achieve faster response times; (c) Improve the efficiency of communication between the EMG Classifier program and the EMG Robot Hand program. This project developed server and client programs because the NI-DAQ card and STG board were installed on separate computers and a method for having them communicate with each other was sought using a cross-over cable. In the future, it is suggested that all hardware and software reside on the same machine. This would reduce the delay of communication and allow both programs to communicate using other methods such as TCP/CP instead of the slower file sharing used in this project. If these improvements can be made, the real-time motions of the robot hand would be much more efficient and responsive to a user’s movements.
BIBLIOGRAPHY


APPENDIX A

MATLAB CODE
function varargout = EMGProgGUIFC(varargin)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Delete the exiting output1.txt
% remember that notepad looks for \\n for newline
if (exist('output1.txt','file'))
    delete('output1.txt');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
EMGPROGGUIFC M-file for EMGProgGUIFC.fig
EMGPROGGUIFC, by itself, creates a new EMGPROGGUIFC or raises the
% existing singleton*.
%
H = EMGPROGGUIFC returns the handle to a new EMGPROGGUIFC or the
% handle to the existing singleton*.
%
EMGPROGGUIFC('CALLBACK',hObject,eventData,handles,...) calls the
local function named CALLBACK in EMGPROGGUIFC.M with the given
input arguments.
%
EMGPROGGUIFC('Property','Value',...) creates a new EMGPROGGUIFC or
raises the existing singleton*. Starting from the left, property
value pairs are applied to the GUI before
EMGProgGUIFC_OpeningFunction gets called. An unrecognized property
name or invalid value makes property application stop. All inputs
are passed to EMGProgGUIFC_OpeningFcn via varargin.
%
*See GUI Options on GUIDE's Tools menu. Choose "GUI allows only
one instance to run (singleton)".
%
See also: GUIDE, GUIDATA, GUIHANDLES
%
Edit the above text to modify the response to help EMGProgGUIFC
%
% Last Modified by Luenin Adriel Barrios Monday June 14 2010 for real-time
% communication with EMG Robot Hand server program.
%
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name', mfilename, ...
    'gui_Singleton', gui_Singleton, ...
    'gui_OpeningFcn', @EMGProgGUIFC_OpeningFcnc, ...
    'gui_OutputFcn', @EMGProgGUIFC_OutputFcnc, ...
    'gui_LayoutFcnc', [], ...
    'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else

```
gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before EMGProgGUIFC is made visible.
function EMGProgGUIFC_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to EMGProgGUIFC (see VARARGIN)

% Load the shared library for interfacing with the EMG device
if exist('EMGDLLmfile.m','file') && exist('EMGDLL.dll','file')
    loadlibrary('EMGDLL', @EMGDLLmf);
end
% Enable/disable buttons based on status of shared library
if libisloaded('EMGDLL')
    set(findobj('Tag','pushbuttoncapture'),'Enable','on');
    % Enable single grasp capture
    set(findobj('Tag','pushbuttonrealtime'),'Enable','on');
    % Enable real-time mode
    set(findobj('Tag','establishMaxTension'),'Enable','on');
    % Enable grasp tension button
    set(findobj('Tag','maxtension'),'Enable','on');
    % Enable grasp tension menu
    set(findobj('Tag','establishRest'),'Enable','on');
    % Enable rest position function
else
    set(findobj('Tag','pushbuttoncapture'),'Enable','off');
    % Disable single grasp capture
    set(findobj('Tag','pushbuttonrealtime'),'Enable','off');
    % Disable real-time mode
    set(findobj('Tag','establishMaxTension'),'Enable','off');
    % Disable grasp tension
    set(findobj('Tag','maxtension'),'Enable','off');
    % Disable grasp tension menu
    set(findobj('Tag','establishRest'),'Enable','off');
    % Disable rest position function
end
set(findobj('Tag','pushbuttontrain'),'Enable','on');
% Enable train mode
set(findobj('Tag','stopRealTime'),'Enable','off');
% Disable real-time stop button

% Load the default classifier
load('dflt_train.mat','matrices');
% Default training data is loaded
set(findobj('Tag','statusbar'),'String','Default Classifier Loaded');
% Update status bar
matrices.EMG = zeros(400,3);
% Create a default EMG matrix
% Add supporting fields, if necessary
if ~isfield(matrices,'threshold')
    matrices.threshold = 15;
%end
if ~isfield(matrices,'restVar')
    matrices.restVar = [10 10 10 10];
end
if ~isfield(matrices,'maxpower')
    matrices.maxpower = [100 100 100 100];
end
if ~isfield(matrices,'tension')
    matrices.tension = 0;
end

% The following key structure elements are contained in the default file
% matrices:
%  S, C, N - Training variables
%  I - I coefficients for spectral moment calculation
%  optTime - optimal time for classification
%  bestRate - classification performance
%  fvFlag - indicates feature vector to use: 0 for wavelength, 1 for
%           spectral moments
%  restVar - variance of rest position recording
%  threshold - Bonato method threshold
%  maxpower - maximum power for each grasp type: index 1 for ball,
%             2 for cylinder, 3 for disk, 4 for key
%  training - previously recorded raw training data
%  classSet - classification of training data
%  EMG - temp variable for storing current EMG data
%  tension - temp variable for storing current tension level

setappdata(hObject,'mydata',matrices);
% Make the classifier available to the GUI

% Load the images for depicting grasps
[A, map, alpha] = imread('key.gif', 1);
grasps.key = A;
grasps.map = map;
A = imread('disk.gif', 1);
grasps.disk = A;
A = imread('sphere.gif', 1);
grasps.sphere = A;
A = imread('cylinder.gif', 1);
grasps.cylinder = A;
A = imread('rest.gif', 1);
grasps.rest = A;
grasps.current = 'rest';
setappdata(hObject,'myimages',grasps);

% Set a flag for determining if stop real-time button has been pressed
buttonFlag.stop = false;
setappdata(hObject,'buttonFlag',buttonFlag);

% Choose default command line output for EMGProgGUIFC
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes EMGProgGUIFC wait for user response (see UIRESUME)
% uiwait(handles.figure1);
% --- Outputs from this function are returned to the command line.
function varargout = EMGProgGUIFC_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% ---Classifies current EMG data and updates class display
function update_class(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Pull classifier and EMG data from GUI memory
matrices = getappdata(handles.figure1,'mydata');

% Classify the EMG signal
class = GetClass(matrices.EMG(:,1:3), matrices.optTime, matrices.S,...
matrices.C, matrices.N, matrices.I, 'MAH', matrices.fvFlag, 0.06);

% Update the Classification display with the designated class image
% Also write corresponding grasp integer value to output1.txt so that
% the EMG Robot Hand may read the values and achieve the same grasp
% position
axes(handles.axesclass);
cia;
grasps = getappdata(handles.figure1,'myimages');
colormap(grasps.map);
switch class
    case 1
        image(grasps.sphere);
displayString='Spherical Grasp';
grasps.current = 'sphere';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%for real time with the robotic hand Spherical
fid = fopen('output1.txt','w');
fprintf(fid, '
%d',2);
fclose(fid);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 2
        image(grasps.cylinder);
displayString='Cylindrical Grasp';
grasps.current = 'cylinder';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%for real time with the robotic hand Cylindrical
fid = fopen('output1.txt','w');
fprintf(fid, '
%d',1);
fclose(fid);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    case 3
        image(grasps.disk);
displayString='Disk Grasp';
grasps.current = 'disk';

...
for real time with the robotic hand Precision
fid = fopen('output1.txt','w');
fprintf(fid, '\r\n%d', 3);
fclose(fid);

case 4
image(grasps.key);
displayString='Key Grasp';
grasps.current = 'key';

% for real time with the robotic hand Lateral
fid = fopen('output1.txt','w');
fprintf(fid, '\r\n%d', 4);
fclose(fid);

otherwise
image(grasps.rest);
displayString='Rest';
grasps.current = 'rest';

% for real time with the robotic hand Rest
fid = fopen('output1.txt','w');
fprintf(fid, '\r\n%d', 0);
fclose(fid);
end
axis off; % Deactivate any axis displays
setappdata(handles.figure1, 'myimages', grasps);
set(findobj('Tag','statusbar'),'String',displayString);
% Update status bar

% --- Updates the 1st 3 channel displays
function update_3EMGDisplay(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Pull EMG data from GUI memory
matrices = getappdata(handles.figure1, 'mydata');

% Clear each EMG display and plot the latest EMG data
axes(handles.axes1);
cla;
plot(matrices.EMG(:,1));
axes(handles.axes2);
cla;
plot(matrices.EMG(:,2));
axes(handles.axes3);
cla;
plot(matrices.EMG(:,3));

% --- Updates all 4 channel displays
function update_4EMGDisplay(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% handles structure with handles and user data (see GUIDATA)

% Pull classifier and EMG data from GUI memory
matrices = getappdata(handles.figure1,'mydata');

% Clear each EMG display and plot the latest EMG data
axes(handles.axes1);
cla;
plot(matrices.EMG(:,1));
axes(handles.axes2);
cla;
plot(matrices.EMG(:,2));
axes(handles.axes3);
cla;
plot(matrices.EMG(:,3));
axes(handles.axes4);
cla;
plot(matrices.EMG(:,4));

% --- Clear all 4 channel displays and the class display
function clear_EMGDisplay(hObject, eventdata, handles)
% hObject  handle to pushbutton1 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Clear each EMG display and plot the latest EMG data
axes(handles.axes1);
cla;
axes(handles.axes2);
cla;
axes(handles.axes3);
cla;
axes(handles.axes4);
cla;
axes(handles.axesclass);
cla;

% --- Update the Tension Bar display with grasp tension levels
function update_TensionBar(hObject, eventdata, handles)
% hObject  handle to pushbutton1 (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% Determine current grasp
grasps = getappdata(handles.figure1,'myimages');
% Pull classifier and EMG data from GUI memory
matrices = getappdata(handles.figure1,'mydata');
% maxpower is 1x4 array containing power of each grasp type
maxpower = matrices.maxpower;
% Get the MAV for channels 3 and 4
mav4 = FeaturesMAV(matrices.EMG(:,4));
mav3 = FeaturesMAV(matrices.EMG(:,3));

% Calculate grasp power using channels 3 and/or 4 as proportion of max
% power
switch grasps.current
    case 'sphere'
        power = (mav4+mav3)/maxpower(1);
    case 'cylinder'
        power = (mav4+mav3)/maxpower(2);
    case 'disk'
        power = (mav4+mav3)/maxpower(3);
    case 'key'
        power = mav3/maxpower(4);
    otherwise
        power = 0;
end

% Convert to percentage
if power>1
    power = 100;
else
    power = power*100;
end
matrices.tension = power;
setappdata(handles.figure1, 'mydata', matrices);
axes(handles.tensionbar);
cla;
barh(power);
set(gca, 'xLim', [0 100]);
set(gca, 'XTick', 0:25:100);
set(gca, 'YTick', []);

% --- Update the EMG channel displays with additional training indicators
function update_EMGTrainDisplay(hObject, eventdata, handles)
    % hObject    handle to pushbutton1 (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

    % Pull classifier and EMG data from GUI memory
    matrices = getappdata(handles.figure1, 'mydata');
    Data1 = matrices.EMG(:,1);
    Data2 = matrices.EMG(:,2);
    Data3 = matrices.EMG(:,3);
    startPt = matrices.startPoint;
    endPt = matrices.endPoint;

    % Clear each EMG display and plot the latest EMG data
    axes(handles.axes1);
cia;
    plot(Data1);
    min1 = min(Data1);
    max1 = max(Data1);
    line([startPt startPt], [min1 max1], 'Color', 'g');
    line([endPt endPt], [min1 max1], 'Color', 'r');

    axes(handles.axes2);
cia;
    plot(Data2);
    min2 = min(Data2);
    max2 = max(Data2);
    line([startPt startPt], [min2 max2], 'Color', 'g');
    line([endPt endPt], [min2 max2], 'Color', 'r');
axes(handles.axes3);
cla;
plot(Data3);
min3 = min(Data3);
max3 = max(Data3);
line([startPt startPt],[min3 max3],'Color','g');
line([endPt endPt],[min3 max3],'Color','r');

% --- Update class display only
function update_classDisplay(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Update the Classification display with the designated class image
axes(handles.axesclass);
cla;
grasps = getappdata(handles.figure1,'myimages');
colormap(grasps.map);
switch grasps.current
  case 'sphere'
    image(grasps.sphere);
    %displayString = 'Spherical Grasp';
  case 'cylinder'
    image(grasps.cylinder);
    %displayString = 'Cylindrical Grasp';
  case 'disk'
    image(grasps.disk);
    %displayString = 'Disk Grasp';
  case 'key'
    image(grasps.key);
    %displayString = 'Key Grasp';
  otherwise
    image(grasps.rest);
    %displayString = 'Rest';
end
axis off;
%set(findobj('Tag','statusbar'),'String',displayString);
% Update status bar

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

function FileMenu_Callback(hObject, eventdata, handles)
% hObject    handle to FileMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --- Function to open EMG data from files and classify the data
function OpenEMGFile_Callback(hObject, eventdata, handles)
% hObject    handle to OpenEMGFile (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Use Open File dialog to select EMG file
[FileName,PathName] = uigetfile('*txt','Select an EMG Data File');
if ~isequal(FileName,0) && ~isequal(PathName,0)
set(findobj('Tag','statusbar'),'String',['Loading ' FileName]);

% Update status bar
X = load(fullfile(PathName,FileName));
EMGData = X(:,1:3);

% Update the EMG data in memory
matrices = getappdata(handles.figure1,'mydata');
matrices.EMG = EMGData; % Store the uploaded file in EMG
setappdata(handles.figure1,'mydata',matrices); % Save the changes
update_class(hObject, eventdata, handles); % Classify the EMG data
update_3EMGDisplay(hObject, eventdata, handles); % Show the EMG data
end

% -------------------------------
% function CloseMenuItem_Callback(hObject, eventdata, handles)
% hObject    handle to CloseMenuItem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
selection = questdlg(['Close ' get(handles.figure1,'Name') '?'],...
    ['Close ' get(handles.figure1,'Name') '...'],...
    'Yes','No','Yes');
if strcmp(selection,'No')
    return;
end
if libisloaded('EMGDLL')
    unloadlibrary EMGDLL; % Clear the EMGDLL from memory
end
delete(handles.figure1)

% --- Executes on button press of Classify Single Grasp.
% function pushbuttoncapture_Callback(hObject, eventdata, handles)
% hObject    handle to pushbuttoncapture (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
if (exist('output1.txt','file'))
    delete('output1.txt');
end

% Get classifier from GUI memory
matrices = getappdata(handles.figure1,'mydata');
optTime = matrices.optTime;
%restVar = matrices.restVar;
threshold = matrices.threshold;

% Create pointers for EMG device control
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);
read = int32(0);
readPtr = libpointer('int32Ptr', read);
samplesPerChan = 100;
X = zeros(samplesPerChan,3); % Preallocate matrix for 2 seconds of data
graspData = [];

% Create flags for detecting grasp
graspFound = false;
onsetFound = false;
onsetStarted = false;

clear_EMGDisplay(hObject, eventdata, handles);

set(findobj('Tag','statusbar'),'String','Configuring device for single grasp mode'); % Update status bar
% Configure DAQ to collect data from 3 channels
ConfigFlag = calllib('EMGDLL','Configure_EMG_RealTimeMode',3,samplesPerChan*30,taskPtr);
if ConfigFlag == 0
    set(findobj('Tag','statusbar'),'String','Device ready'); % Update status bar
    selection = questdlg('Press Enter to capture grasp.','Start Capture','Start','Cancel','Start');
    if strcmp(selection,'Cancel')
        return;
    end
    StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
    if StartFlag == 0
        % Establish rest variance
        Z = zeros(samplesPerChan*2,3);
        % read first 200 ms and get rest variance
        [ReadFlag, Y, numread] = calllib('EMGDLL','Read_EMG',taskPtr.value,Z',samplesPerChan*2,readPtr);
        restVar = var(Y'*1000);
        % Scan samplesPerChan at a time until the onset of movement is % detected
        while ~onsetFound
            [ReadFlag, Y, numread] = calllib('EMGDLL','Read_EMG',taskPtr.value,X',samplesPerChan,readPtr);
            if ReadFlag == 0
                if onsetStarted
                    graspData = [graspData; Y'*1000]; %Need to multiply by % 1000 to convert to mV
                    [onset, activeCount, firstk, lastIndex] = BonatoRealTime(graspData,lastIndex,activeCount,firstk,restVar,threshold);
                else
                    % Determine first index of EMG signal for onset of % movement
                    graspData = Y'*1000;
                    [onset, activeCount, firstk, lastIndex] = BonatoRealTime(graspData,1,[],[],restVar,threshold);
                end
                if activeCount == zeros(1,size(graspData,2))
                    onsetStarted = false;
                else
                    onsetStarted = true;
                    if onset<=size(graspData,1)
                        onsetFound = true;
                    end
                end
            else
                break;
            end
        end
    end
end
h = errordlg('Error reading EMG data','Data Collection Error','modal');
uiwait(h);
calllib('EMGDLL','Stop_EMG',taskPtr.value);
return;
else
    h = errordlg('Device failed to start','Data Collection Error','modal');
    uiwait(h);
calllib('EMGDLL','Stop_EMG',taskPtr.value);
    return;
end
end

graspFound = true;
end

% sufficient data to classify a grasp has been collected at this time
matrices.EMG = graspData(onset:onset+optTime-1,:);
% Store the new data in EMG
setappdata(handles.figure1,'mydata',matrices);
% Save the changes
update_class(hObject, eventdata, handles);
update_3EMGDisplay(hObject, eventdata, handles);
calllib('EMGDLL','Stop_EMG',taskPtr.value);
else
    h = errordlg('Device failed to start','Data Collection Error','modal');
    uiwait(h);
else
    set(findobj('Tag','statusbar'),'String','Device configuration failed');
    % Update status bar
    h = errordlg('Error configuring EMG device','Device Configuration Error','modal');
    uiwait(h);
end

% --- Executes on button press of Train.
function pushbuttontrain_Callback(hObject, eventdata, handles)
% hObject    handle to pushbuttontrain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% hObject    handle to pushbuttontrain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

set(findobj('Tag','statusbar'),'String','Training Mode'); % Update status bar
displayString = {'You are about to train the system. Training requires you to perform multiple grasps repeatedly and will take 20-30 minutes.',
    'Press Start to continue.'};
selection = questdlg(displayString,'Start Training?','Start','Cancel','Start');
if strcmp(selection,'Cancel')
    return;
end

% If the EMGRecordings folder doesn't exist, create it
if ~exist('EMGRecordings','dir')
    mkdir('EMGRecordings');
end

clear_EMGDisplay(hObject, eventdata, handles);

repeatTraining = true;
while repeatTraining
    % continue training until user satisfied with performance
    repeatTraining = false;
    numTraining = 0;
    prevtables = 0;

    % Load rest data and previously recorded data
    matrices = getappdata(handles.figure1,'mydata');
    threshold = matrices.threshold;

    % Load previously stored training data
    PreviousTrainData3D = [];
    if isfield(matrices,'training')
        PreviousTrainData3D = matrices.training;
    end
    PreviousClassSet = [];
    if isfield(matrices,'classSet')
        PreviousClassSet = matrices.classSet;
    end

    % Determine whether to append or discard previous data
    appendPrevious = false;
    if ~isempty(PreviousTrainData3D)
        [rows cols prevtables] = size(PreviousTrainData3D);
        displayString = [int2str(prevtables),' training sets are already loaded.',' Do you want to discard this training or append to it?'];
        selection = questdlg(displayString,'Disposition of Existing Training','Append','Discard','Append');
        if strcmp(selection,'Append')
            appendPrevious = true;
        else
            matrices.training = [];
            matrices.classSet = [];
            setappdata(handles.figure1,'mydata',matrices);
            prevtables = 0;
        end
    end
end

% Perform training only if shared library is loaded
if libisloaded('EMGDLL')

% Create pointers for EMG device control
read = int32(0);
readPtr = libpointer('int32Ptr', read);
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);

% Number of training grasps to collect.
% Recommend 72 or 18 per grasp in order to achieve classification
% rates in
% high 90th percentile with smaller time periods
trainingDetermined = false;
while ~trainingDetermined
  if appendPrevious
    displayString = 'Enter the desired number of additional training samples for each grasp: '
    userEntry = inputdlg(displayString, 'Determine Training Sample Size', 1, {'3'});
  else
    displayString = [['Enter the desired number of training samples for each grasp: ',...
      '(18 Recommended, 3 Minimum)']
    userEntry = inputdlg(displayString, 'Determine Training Sample Size', 1, ['18']);
  end
  if size(userEntry) > 0
    userNum = str2num(userEntry{1,1});
    trainingDetermined = true;
    if ~appendPrevious && userNum<3
      userNum = 3;
    elseif appendPrevious && (userNum+prevtables/4)<3
      userNum = 3-(round(prevtables/4)+userNum);
    end
    numTraining = userNum*4;
  else
    selection2 = questdlg('Are you sure you want to cancel training?','Cancel Training?','Yes','No','No');
    if strcmp(selection2,'Yes')
      return;
    end
  end
end

if numTraining>0
  % Now ready to start collecting training grasps
  set(findobj('Tag','statusbar'),'String','Training Mode: Collect grasp readings'); % Update status bar
displayString = {{'You will now perform 4 grasps','int2str(userNum),' times each in random order.'},
  {'You will have 2 seconds to complete each grasp. After each grasp, you may accept or discard the reading.'}};
  selection = questdlg(displayString,'Start Grasp
Readings?','Start','Cancel','Start');
if strcmp(selection,'Cancel')
    selection2 = questdlg('Are you sure you want to cancel training?','Cancel Training?','Yes','No','No');
    if strcmp(selection2,'Yes')
        return;
    end
end
grasps = getappdata(handles.figure1,'myimages');
samplesPerChan = 2000;
X = zeros(samplesPerChan,3); % Preallocate space for each grasp reading
TrainData3D(:,:,numTraining) = zeros(400,3); % Preallocate space to store all grasp readings
trainingID = 1:numTraining; % Create an array with a training ID for each grasp
while ~isempty(trainingID)
    % Get index from X to determine grasp type
    index = ceil(rand()*length(trainingID));
    tID = trainingID(index); % Remove the selected index from the list of training grasps to collect
    trainingID = cat(2,trainingID(1:index-1),trainingID(index+1:length(trainingID)));
    if tID <= (numTraining*0.25)
        graspType = 1; % Sphere
        grasps.current = 'sphere';
    elseif tID <= (numTraining*0.5)
        graspType = 2; % Cylinder
        grasps.current = 'cylinder';
    elseif tID <= (numTraining*0.75)
        graspType = 3; % Disk
        grasps.current = 'disk';
    else
        graspType = 4; % Key
        grasps.current = 'key';
    end
    % Now update the class display to prompt the user
    setappdata(handles.figure1,'myimages',grasps);
    update_classDisplay(hObject, eventdata, handles);
    doneGrasp = false;
    while ~doneGrasp
        ConfigFlag = calllib('EMGDLL','Configure_EMG_RecordMode',3,samplesPerChan*3,taskPtr);
        %taskPtr.value;
        if ConfigFlag == 0
            doneStart = false; % Use doneStart to give user option to cancel training
            while ~doneStart
                switch graspType
                case 1
                    selection = questdlg('Perform a Spherical Grasp','Read Grasp','Start','Cancel','Start');
                case 2
                    selection = questdlg('Perform a Cylinder Grasp','Read Grasp','Start','Cancel','Start');
            end
case 3
    selection = questdlg('Perform a Disk Grasp', 'Read Grasp', 'Start', 'Cancel', 'Start');
    otherwise
        selection = questdlg('Perform a Key Grasp', 'Read Grasp', 'Start', 'Cancel', 'Start');
    end
    if strcmp(selection, 'Cancel')
        selection2 = questdlg('Are you sure you want to cancel training?', 'Cancel Training?', 'Yes', 'No', 'No');
        if strcmp(selection2, 'Yes')
            return;
        end
    end
doneStart = true;
end
StartFlag = calllib('EMGDLL', 'Start_EMG', taskPtr.value)
    if StartFlag == 0
        [ReadFlag, Y, numread] = calllib('EMGDLL', 'Read_EMG', taskPtr.value, X', samplesPerChan, readPtr);
        if ReadFlag == 0
            graspData = Y'*1000; % Need to multiply by 1000 to convert to mV
            matrices.EMG = graspData; % Store the new data in EMG
            % Determine first index of EMG signal crossing rest threshold
            onset = BonatoOnset(graspData, threshold);
            if onset>size(graspData,1)
                h = errordlg('Rest position threshold not exceeded', 'Data Collection Error', 'modal');
                uiwait(h);
            else
                % verify that startPoint + 400 does not exceed size of data
                [rows cols] = size(graspData);
                if rows<onset+399
                    h = errordlg('Insufficient grasp data collected', 'Data Collection Error', 'modal');
                    uiwait(h);
                else
                    matrices.startPoint = onset;
                    matrices.endPoint = onset+399;
                    setappdata(handles.figure1, 'mydata', matrices); % Save the changes
                    update_EMGTrainDisplay(hObject, eventdata, handles); % Update display
                    selection2 = questdlg('Do you want to keep this reading?', 'Keep Grasp Reading?', 'Yes', 'No', 'Yes');
                    if strcmp(selection2, 'Yes')
                        % Store the grasp reading in the training set
                    end
                end
            end
        end
    end
TrainData3D(1:400,1:3,tID) = graspData(onset:onset+399,1:3);

switch graspType
    case 1
        filename = strcat('.\EMGRecordings\Sphere_',int2str(tID),'\','.txt');
    case 2
        filename = strcat('.\EMGRecordings\Cylinder_',int2str(tID),'\','.txt');
    case 3
        filename = strcat('.\EMGRecordings\Disk_',int2str(tID),'\','.txt');
    otherwise
        filename = strcat('.\EMGRecordings\Key_',int2str(tID),'\','.txt');
end

% Now save the 400ms grasp window to permanent file for backup and potential study
dataToSave = graspData(onset:onset+399,1:3);
save(filename,'dataToSave','-ascii','-double','-tabs');
doneGrasp = true;
numCompleted = numTraining - length(trainingID);
displayString = ['Training Mode: ' int2str(numCompleted) '/ ' int2str(numTraining) ' grasps completed'];

set(findobj('Tag','statusbar'),'String',displayString);  % Update status bar

else
    h = errordlg('Error reading EMG data','Data Collection Error','modal');
    uwait(h);
end

StopFlag = calllib('EMGDLL','Stop_EMG',taskPtr.value);

end  %end of ConfigFlag check
end  %end of while loop for collecting single grasp data
end  %end of collecting all grasps
end  % end check to determine if additional training was necessary

if numTraining+prevtables>0
h = msgbox('Please wait while the optimal time for classification is determined.', 'Optimizing Time Parameter');
displayString = 'Optimizing Time for Classification: Please wait...';
set(findobj('Tag','statusbar'), 'String', displayString);  % Update status bar
  %uiwait(h, 0.1);

% Now determine optimal time for classification using LOO Validation

% Preallocate TrainSetSM, TrainSetWL, and ClassSet for speed
if appendPrevious
    TrainSetSM = zeros(numTraining+prevtables, 9);
    TrainSetWL = zeros(numTraining+prevtables, 3);
    ClassSet = zeros(numTraining+prevtables, 1);
else
    TrainSetSM = zeros(numTraining, 9);
    TrainSetWL = zeros(numTraining, 3);
    ClassSet = zeros(numTraining, 1);
end

% Assign corresponding position in ClassSet with appropriate class
ClassSet(1:numTraining*0.25)=1;  % Sphere
ClassSet(numTraining*0.25+1:numTraining*0.5)=2;  % Cylinder
ClassSet(numTraining*0.5+1:numTraining*0.75)=3;  % Disk
ClassSet(numTraining*0.75+1:numTraining)=4;  % Key

% If the user is appending previously recorded raw training data, it is done here
if appendPrevious
    for n = 1:prevtables
        % Append tables to end
        TrainData3D(:,:,numTraining+n) = PreviousTrainData3D(:,:,n);
        % Append classes to class set
        ClassSet(numTraining+n,1) = PreviousClassSet(n,1);
    end
end

% Get the I-coefficients for moment calculation
I = GenerateI(2,11);

% Update numTraining based on actual number in the training set
numTraining = size(TrainData3D,3);

% Test performance of classifier for training set at each 10 ms time
% step from 200-400 msec
scoreWL = [];
scoreSM = [];
for Ts = 200:10:400
    % Create training set using wavelength feature no matter how much training is on hand
    for n = 1:numTraining
tabledata = TrainData3D(:,:,n); % Extract specific training grasp
F = FeaturesWL(tabledata(1:Ts,:)); % Perform feature extraction
TrainSetWL(n,:) = F; % Store feature vector in training set
end
% Perform BoxCox transformation on training set
TrainSetWL = BoxCox(TrainSetWL, 0.06);
% Perform Leave-One-Out Validation to determine optimal time
H = LOOValidation(TrainSetWL, ClassSet, 'MAH');
scoreWL = [scoreWL; Ts, H]; % maintain score for each time step

if numTraining > 60
% Sufficient training exists to compare both feature vectors
% Loop through each table of training data and create spectral moment feature vector using current Ts level
for n = 1:numTraining
    tabledata = TrainData3D(:,:,n); % Extract specific training grasp
    F = Features(tabledata, Ts, 11, 2, 0, I); % Perform feature extraction
    TrainSetSM(n,:) = F; % Store feature vector in training set
end
% Perform BoxCox transformation on training set
TrainSetSM = BoxCox(TrainSetSM, 0.06);
% Perform Leave-One-Out Validation to determine best time period for classification
H = LOOValidation(TrainSetSM, ClassSet, 'MAH');
scoreSM = [scoreSM; Ts, H]; % maintain score for each time step
end
% Determine best performing time index
WLbestIndex = find(scoreWL(:,2) > 0.98, 1);
if isempty(WLbestIndex)
    [WLbestRate WLbestIndex] = max(scoreWL);
    WLbestTime = scoreWL(WLbestIndex(2), 1);
else
    WLbestTime = scoreWL(WLbestIndex, 1);
    WLbestRate = scoreWL(WLbestIndex, 2);
end
% Now check the immediate time samples surrounding the optimal times
scoreWL = [];
if WLbestTime == 400
    startTime = 391;
    endTime = 400;
elseif WLbestTime == 200
    startTime = 200;
    endTime = 209;
else
    startTime = WLbestTime - 9;
    endTime = WLbestTime + 9;
end
for Ts = startTime:endTime
    for n = 1:numTraining
        tabledata = TrainData3D(:,:,n); % Extract specific
        % training grasp
        F = FeaturesWL(tabledata(1:Ts,:)); % Perform feature
        % extraction
        TrainSetWL(n,:) = F; % Store feature vector in training set
    end
    % Perform BoxCox transformation on training set
    TrainSetWL = BoxCox(TrainSetWL, 0.06);
    % Perform Leave-One-Out Validation to determine optimal time
    H = LOOValidation(TrainSetWL,ClassSet,'MAH');
    scoreWL = [scoreWL; Ts, H]; % maintain score for each time
    % step
end
if numTraining > 60
    SMbestIndex = find(scoreSM(:,2)>0.98,1);
    if isempty(SMbestIndex)
        [SMbestRate SMbestIndex] = max(scoreSM);
        SMbestTime = scoreSM(SMbestIndex(2), 1);
    else
        SMbestTime = scoreSM(SMbestIndex,1);
        SMbestRate = scoreSM(SMbestIndex,2);
    end
    scoreSM = []; % SM best time
    if SMbestTime == 400
        startTime = 391;
        endTime = 400;
    elseif SMbestTime == 200
        startTime = 200;
        endTime = 209;
    else
        startTime = SMbestTime - 9;
        endTime = SMbestTime + 9;
    end
    for Ts = startTime:endTime
        for n = 1:numTraining
            tabledata = TrainData3D(:,:,n); % Extract specific
            % training grasp
            F = Features(tabledata,Ts,11,2,0,I); % Perform feature
            % extraction
            TrainSetSM(n,:) = F; % Store feature vector in training set
        end
        % Perform BoxCox transformation on training set
        TrainSetSM = BoxCox(TrainSetSM, 0.06);
        % Perform Leave-One-Out Validation to determine optimal time
        H = LOOValidation(TrainSetSM,ClassSet,'MAH');
        scoreSM = [scoreSM; Ts, H]; % maintain score for each time
        % step
    end
end
% Determine best performing time index
[WLbestRate WLbestIndex] = max(scoreWL);
WLbestTime = scoreWL(WLbestIndex(2), 1);
% Default setting is to use the wavelength feature
fvFlag = 0;
bestTime = WLbestTime;
bestRate = WLbestRate(2);
if numTraining > 60
    % With sufficient training, consider spectral moment
    % performance
    [SMbestRate SMbestIndex] = max(scoreSM);
    SMbestTime = scoreSM(SMbestIndex(2), 1);
    % Need to score both feature vectors
    % Every 1ms gain in optimal time is rated as equivalent to a
    % 0.1%
    % gain in classification rate. So, if spectral moments is
    % faster
    % than wavelength by 10ms, but classifies within 1% of
    % wavelength,
    % then it will be selected as the optimal feature vector
    SMscore = SMbestRate(2) + (WLbestTime-SMbestTime)*0.001;
    WLscore = WLbestRate(2);
    if SMscore > WLscore
        % Designate spectral moments as the feature vector
        fvFlag = 1;
        bestTime = SMbestTime;
        bestRate = SMbestRate(2);
    end
end

close(h); % Destroy message box referencing this test
matrices.fvFlag = fvFlag;
matrices.optTime = bestTime; % Store the bestTime for later
% reference
matrices.bestRate = bestRate*100; % Store the best rate as a
% percentage for reference
if fvFlag
    displayString = {{'Highest classification rate: '},
                        num2str(bestRate*100,'%10.1f'),','},
                        {'Optimal Time: '},
                        int2str(bestTime),', msec'},
                        {'Feature: Spectral Moments'},
                        'Would you like to conduct further training?'});
else
    displayString = {{'Highest classification rate: '},
                        num2str(bestRate*100,'%10.1f'),','},
                        {'Optimal Time: '},
                        int2str(bestTime),', msec'},
                        {'Feature: Waveform Length'},
                        'Would you like to conduct further training?'});
end

% Finally, set the reference training data to the optimal training
% set parameters
for n = 1:numTraining
    tabledata = TrainData3D(:,:,n); % Extract specific training
    % grasp
    if fvFlag
F = Features(tabledata,bestTime,11,2,0,I); % Perform
% feature extraction
else F = FeaturesWL(tabledata(1:bestTime,:));
end
TrainSet(n,:) = F;
end
TrainSet = BoxCox(TrainSet, 0.06); % Perform BoxCox transformation
% on training set

% Train the classifier
[S, C, N] = Train(TrainSet, ClassSet, 'MAH');
matrices.S = S;
matrices.C = C;
matrices.N = N;
matrices.I = I;
% Save raw training data with classifier for easy appending of new
% training
matrices.training = TrainData3D;
matrices.classSet = ClassSet;
setappdata(handles.figure1,'mydata',matrices); % Save the
% classifier data

if bestRate > 0.9 && bestTime < 300
    selection = questdlg(displayString,'Conduct Additional
Training?','Yes','No','No');
else
    selection = questdlg(displayString,'Conduct Additional
Training?','Yes','No','Yes');
end
if strcmp(selection,'Yes')
    repeatTraining = true;
end
set(findobj('Tag','statusbar'),'String','Classifier Trained'); %
Update status bar
end % end of ensuring training data exists
end % end training loop

% Now save the classifier variables to file
SaveTrainingData_Callback(hObject, eventdata, handles)

% Hints: get(hObject,'String') returns contents of statusbar as text
% str2double(get(hObject,'String')) returns contents of statusbar
as a double

% --- Executes during object creation, after setting all properties.
function statusbar_CreateFcn(hObject, eventdata, handles)
% hObject    handle to statusbar (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% function statusbar_Callback(hObject, eventdata, handles)
% hObject    handle to statusbar (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function ClassifierMenu_Callback(hObject, eventdata, handles)
% hObject handle to ClassifierMenu (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

function LoadTrainingData_Callback(hObject, eventdata, handles)
% hObject handle to LoadTrainingData (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

function pushbuttonrealtime_Callback(hObject, eventdata, handles)
% hObject handle to pushbuttonrealtime (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

set(findobj('Tag','stopRealTime'),'Enable','on'); % Enable real-time stop
% button
matrices = getappdata(handles.figure1,'mydata');
restVar = matrices.restVar;
optTime = matrices.optTime;
threshold = matrices.threshold;

% Create pointers for DAQ management
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);
read = int32(0);
readPtr = libpointer('int32Ptr', read);
sampsPerChan = 100;
X = zeros(sampsPerChan,4);
realtimeDone = false;

set(findobj('Tag','statusbar'),'String','Configuring device for real-time
mode'); % Update status bar
% Configure DAQ to collect 4 channels of EMG data
ConfigFlag =
calllib('EMGDLL','Configure_EMG_RealTimeMode',4,sampsPerChan*30,taskPtr);
if ConfigFlag == 0
    set(findobj('Tag','statusbar'),'String','Device ready'); % Update
% status bar
    selection = questdlg('Press Enter to Start','Real-Time
Mode','Start','Cancel','Start');
    if strcmp(selection,'Cancel')
        return;
    end
StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
if StartFlag == 0
    % Begin continuous loop
    % Now in rest position
    % Update the classification display to a rest position
    grasps.current = 'rest';
    setappdata(handles.figure1,'myimages',grasps);
    update_classDisplay(hObject, eventdata, handles);
    set(findobj('Tag','statusbar'),'String','Rest'); % Update
% status bar

    graspData = [];
graspFound = false;
onsetFound = false;
onsetStarted = false;
while ~onsetFound
    [ReadFlag, Y, numread] =
calllib('EMGDLL','Read_EMG',taskPtr.value,X',sampsPerChan,readPtr);
    if ReadFlag == 0
        if onsetStarted
            graspData = [graspData; Y'*1000]; % Need to
% multiply by 1000 to convert to mV
            [onset, activeCount, firstk, lastIndex] =
BonatoRealTime(graspData(:,1:3),lastIndex,activeCount,firstk,restVar,threshold);
else
    % Determine first index of EMG signal for onset of
    % movement
    graspData = Y'*1000;
    [onset, activeCount, firstk, lastIndex] =
    BonatoRealTime(graspData(:,1:3),1,[],[],restVar,threshold);
end
if activeCount == zeros(1,3)
    onsetStarted = false;
else
    onsetStarted = true;
    if onset<=size(graspData,1)
        onsetFound = true;
    end
end
else
    h = errordlg('Error reading EMG data', 'Data Collection
    Error', 'modal');
    uiwait(h);
    calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
    return;
end
end
% onset has been detected, now need to ensure enough data is
% collected to classify grasp
while ~graspFound
    samples = size(graspData,1);
    if samples<onset+optTime-1
        [ReadFlag, Y, numread] =
        calllib('EMGDLL', 'Read_EMG', taskPtr.value, X', sampsPerChan, readPtr);
        if ReadFlag == 0
            graspData = [graspData; Y'*1000];  % Need to
            % multiply by 1000 to convert to mV
        else
            h = errordlg('Error reading EMG data', 'Data
            Collection Error', 'modal');
            uiwait(h);
            calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
            return;
        end
    else
        graspFound = true;
    end
end
% sufficient data to classify a grasp has been collected at
% this time
matrices.EMG = graspData(onset:onset+optTime-1,:);  % Store the
% new data in EMG
setappdata(handles.figure1, 'mydata', matrices);  % Save the
% changes
update_class(hObject, eventdata, handles);  % Get
% classification
update_4EMGDisplay(hObject, eventdata, handles);

% Maintain grasp until rest position detected
holdUntilRest = true;
while holdUntilRest
In holdUntilRest state

```matlab
[ReadFlag, Y, numread] = calllib('EMGDLL', 'Read_EMG', taskPtr.value, X', sampsPerChan, readPtr);
if ReadFlag == 0
    graspData = Y' * 1000; % Need to multiply by 1000 to convert to mV
    matrices.EMG = graspData; % Update EMG variable
    setappdata(handles.figure1, 'mydata', matrices); % Update the tension bar display with the currently applied grasp tension
    update_TensionBar(hObject, eventdata, handles);
    matrices = getappdata(handles.figure1, 'mydata'); % if tension falls below 10 percent then in rest state
    if matrices.tension < 10
        holdUntilRest = false;
    end
    % update_4EMGDisplay(hObject, eventdata, handles); % Show the EMG data
    else
        h = errordlg('Error reading EMG data', 'Data Collection Error', 'modal');
        uiwait(h);
        calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
        return;
    end
end
% buttonFlag is used to end real-time mode, when the stop button is pressed
buttonFlag = getappdata(handles.figure1, 'buttonFlag');
if buttonFlag.stop
    realtimeDone = true;
end
% End of real-time loop
calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
set(findobj('Tag', 'statusbar'), 'String', 'Real-time mode ended');
else
    h = errordlg('Device failed to start', 'Data Collection Error', 'modal');
    uiwait(h);
else
    h = errordlg('Error configuring EMG device', 'Device Configuration Error', 'modal');
    uiwait(h);
end
```

% ---------------------------------------------------------------
% function DeviceMenu_Callback(hObject, eventdata, handles)
% hObject    handle to DeviceMenu (see GCBO)
% eventdata  reserved to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% ---------------------------------------------------------------

```matlab
function DeviceMenu_Callback(hObject, eventdata, handles)
    % hObject    handle to DeviceMenu (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)
```

% ---------------------------------------------------------------
% function SaveTrainingData_Callback(hObject, eventdata, handles)
% hObject    handle to SaveTrainingData (see GCBO)
% ---------------------------------------------------------------

```matlab
function SaveTrainingData_Callback(hObject, eventdata, handles)
    % hObject    handle to SaveTrainingData (see GCBO)
```
matrices = getappdata(handles.figure1,'mydata');
% Now save the classifier variables to file  
[FileName,PathName] = uiputfile('.mat','Save  
Classifier','traindata.mat');
if ~isequal(FileName,0) && ~isequal(PathName,0)  
save(fullfile(PathName,FileName),'matrices');
end

% Get the classifier data from GUI memory
matrices = getappdata(handles.figure1,'mydata');
bestRate = matrices.bestRate;
bestTime = matrices.optTime;
if matrices.fvFlag  
displayString = {{'Highest classification rate:  
',num2str(bestRate,'%10.1f'),'%',...  
  ['Optimal Time: ',int2str(bestTime),' msec'],'Feature: Spectral  
Moments'};}  
else  
displayString = {{'Highest classification rate:  
',num2str(bestRate,'%10.1f'),'%',...  
  ['Optimal Time: ',int2str(bestTime),' msec'],'Feature: Waveform  
Length'};}  
end
h = msgbox(displayString,'Classifier Trained','modal');
uiwait(h);

% --------------------------------------------------------------------
function HelpMenu_Callback(hObject, eventdata, handles)
% hObject    handle to HelpMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --------------------------------------------------------------------
function AboutMenu_Callback(hObject, eventdata, handles)
% hObject    handle to AboutMenu (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

displayString = {'Copyright 2008, Christopher J. Miller, All Rights  
Reserved','Masters Thesis: Real-Time Feature Extraction and Classification  
of Prehensile EMG Signals, Spring 2008','San Diego State University (SDSU)'};

h = helpdlg(displayString,'About');
uiwait(h);

% --------------------------------------------------------------------
function viewPerformance_Callback(hObject, eventdata, handles)
% hObject    handle to viewPerformance (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get the classifier data from GUI memory
matrices = getappdata(handles.figure1,'mydata');
bestRate = matrices.bestRate;
bestTime = matrices.optTime;
if matrices.fvFlag  
displayString = {{'Highest classification rate:  
',num2str(bestRate,'%10.1f'),'%',...  
  ['Optimal Time: ',int2str(bestTime),' msec'],'Feature: Spectral  
Moments'};}  
else  
displayString = {{'Highest classification rate:  
',num2str(bestRate,'%10.1f'),'%',...  
  ['Optimal Time: ',int2str(bestTime),' msec'],'Feature: Waveform  
Length'};}  
end
h = msgbox(displayString,'Classifier Trained','modal');
uiwait(h);
if libisloaded('EMGDLL')
    % First set a baseline for the resting position
    samplesPerChan = 5000;
    X = zeros(samplesPerChan,4);
    % Preallocate matrix for 5 seconds of data
    % Create pointers for EMG device control
    read = int32(0);
    readPtr = libpointer('int32Ptr', read);
    taskHandle = uint32(0);
    taskPtr = libpointer('uint32Ptr', taskHandle);
    % Update the display to a rest position
    grasps = getappdata(handles.figure1,'myimages');
    grasps.current = 'rest';
    setappdata(handles.figure1,'myimages',grasps);
    update_classDisplay(hObject, eventdata, handles);
    doneRest = false;
    while ~doneRest
        ConfigFlag = calllib('EMGDLL','Configure_EMG_RecordMode',4,samplesPerChan,taskPtr);
        taskPtr.value
        if ConfigFlag == 0
            doneStart = false;
            while ~doneStart
                set(findobj('Tag','statusbar'),'String','Training Mode: Establish rest position'); % Update status bar
                selection = questdlg('Place your hand in the rest position for 5 seconds.','Establish Rest Position','Start','Cancel','Start');
                if strcmp(selection,'Cancel')
                    selection2 = questdlg('Are you sure you want to cancel rest position training?','Cancel Rest Position Training?','Yes','No','No');
                    if strcmp(selection2,'Yes')
                        return;
                    end
                end
                doneStart = true;
            end
            StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
            if StartFlag == 0
                [ReadFlag, Y, numread] = calllib('EMGDLL','Read_EMG',taskPtr.value,X',samplesPerChan,readPtr);
                % ReadFlag = 0;
                if ReadFlag == 0
                    % restData = zeros(5000,3);
                    % Test case
                    restData = Y' *1000;
                    % Need to multiply by 1000 to convert to mV
                    matrices = getappdata(handles.figure1,'mydata');
                    % Get classifier data
                    matrices.EMG = restData; % Store the new data in EMG
setappdata(handles.figure1, 'mydata', matrices);
    % Save the changes
update_3EMGDisplay(hObject, eventdata, handles);
selection2 = questdlg('Do you want to keep this reading?', 'Keep Rest Data?', 'Yes', 'No', 'Yes');
if strcmp(selection2, 'Yes')
    doneRest = true;
end
% Determine max and min values for each channel to determine threshold for starting grasp reading
matrices.restVar = var(restData(:,1:3));
%matrices.closeVar = var(restData(:,3:4));
setappdata(handles.figure1, 'mydata', matrices);
% Save the changes
else
    h = errordlg('Error reading EMG data', 'Data Collection Error', 'modal');
    uiwait(h);
end
callib('EMGDLL', 'Stop_EMG', taskPtr.value);
else
    h = errordlg('Device failed to start', 'Data Collection Error', 'modal');
    uiwait(h);
end
%end of ConfigFlag check

end % end of while loop for collecting rest data
end % end of checking EMG library

% --- Function to establish max power for spherical grasp
function establishSpherePower_Callback(hObject, eventdata, handles)
% hObject    handle to establishPower (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
samplesPerChan = 2000;
X = zeros(samplesPerChan,4);  % Preallocate matrix for 5 seconds of data

% Create pointers for EMG device control
read = int32(0);
readPtr = libpointer('int32Ptr', read);
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);

% Update the display to a spherical grasp position
grasps = getappdata(handles.figure1, 'myimages');
grasps.current = 'sphere';
setappdata(handles.figure1, 'myimages', grasps);
update_classDisplay(hObject, eventdata, handles);

doneGrasp = false;
while ~doneGrasp
    ConfigFlag = callib('EMGDLL', 'Configure_EMG_RecordMode', 4, samplesPerChan, taskPtr);
    taskPtr.value
if ConfigFlag == 0
    doneStart = false;
    while ~doneStart
        set(findobj('Tag','statusbar'),'String','Training Mode: Establish maximum power for spherical grasp'); % Update status bar
        selection = questdlg('Apply maximum grasp tension to a ball for 2 seconds.','Establish Max Power','Start','Cancel','Start');
        if strcmp(selection,'Cancel')
            selection2 = questdlg('Are you sure you want to cancel grasp power training?','Cancel Max Power Training?','Yes','No','No');
            if strcmp(selection2,'Yes')
                return;
            end
        end
        doneStart = true;
    end
end

StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
if StartFlag == 0
    [ReadFlag, Y, numread] = calllib('EMGDLL','Read_EMG',taskPtr.value,X',samplesPerChan,readPtr);
    if ReadFlag == 0
        readData = Y'*1000;
        % Need to multiply by 1000 to convert to mV
        matrices = getappdata(handles.figure1,'mydata');
        % Get classifier data
        matrices.EMG = readData; % Store the new data in EMG
        setappdata(handles.figure1,'mydata',matrices);
        % Save the changes
        update_4EMGDisplay(hObject, eventdata, handles);
        selection2 = questdlg('Do you want to keep this reading?','Keep Sphere Power Data?','Yes','No','Yes');
        if strcmp(selection2,'Yes')
            doneGrasp = true;
            % Determine max power using channels 3 and 4
            mav4 = FeaturesMAV(matrices.EMG(:,4));
            mav3 = FeaturesMAV(matrices.EMG(:,3));
            if isfield(matrices,'maxpower')
                maxpower = matrices.maxpower;
            end
            maxpower(1) = mav4+mav3;
            matrices.maxpower = maxpower;
            setappdata(handles.figure1,'mydata',matrices);
            % Save the changes
        end
    else
        h = errordlg('Error reading EMG data','Data Collection Error','modal');
        uiswait(h);
    end
    calllib('EMGDLL','Stop_EMG',taskPtr.value);
else
    h = errordlg('Device failed to start','Data Collection Error','modal');
    uiswait(h);
end
end %end of ConfigFlag check
end %end of while loop for collecting sphere grasp data

% --- Function to establish max power for cylindrical grasp
function establishCylinderPower_Callback(hObject, eventdata, handles)
% hObject    handle to establishPower (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

samplesPerChan = 2000;
X = zeros(samplesPerChan,4); % Preallocate matrix for 5 seconds of data

% Create pointers for EMG device control
read = int32(0);
readPtr = libpointer('int32Ptr', read);
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);

% Update the display to a grasp position
grasps = getappdata(handles.figure1,'myimages');
grasps.current = 'cylinder';
setappdata(handles.figure1,'myimages',grasps);
update_classDisplay(hObject, eventdata, handles);

doneGrasp = false;
while ~doneGrasp
    ConfigFlag = calllib('EMGDLL','Configure_EMG_RecordMode',4,samplesPerChan,taskPtr);
taskPtr.value
    if ConfigFlag == 0
        doneStart = false;
        while ~doneStart
            set(findobj('Tag','statusbar','String','Training Mode:
Establish maximum power for cylindrical grasp'); % Update status bar
            selection = questdlg('Apply maximum grasp tension to a
cylinder for 2 seconds.','Establish Max Power','Start','Cancel','Start');
            if strcmp(selection,'Cancel')
                selection2 = questdlg('Are you sure you want to cancel
grp power training?','Cancel Max Power Training?','Yes','No','No');
                if strcmp(selection2,'Yes')
                    return;
                end
            end
        end
        doneStart = true;
    end
    StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
    if StartFlag == 0
        [ReadFlag, Y, numread] = calllib('EMGDLL','Read_EMG',taskPtr.value,X',samplesPerChan,readPtr);
        if ReadFlag == 0
            readData = Y'*1000;
            % Need to multiply by 1000 to convert to mV
            matrices = getappdata(handles.figure1,'mydata');
            % Get classifier data
            matrices.EMG = readData; % Store the new data in EMG
            setappdata(handles.figure1,'mydata',matrices);
% Save the changes
update_4EMGDisplay(hObject, eventdata, handles);
selection2 = questdlg('Do you want to keep this reading?', 'Keep Cylinder Power Data?', 'Yes', 'No', 'Yes');
if strcmp(selection2, 'Yes')
doneGrasp = true;

% Determine max power using channels 3 and 4
mav4 = FeaturesMAV(matrices.EMG(:,4));
mav3 = FeaturesMAV(matrices.EMG(:,3));
if isfield(matrices, 'maxpower')
    maxpower = matrices.maxpower;
end
maxpower(2) = mav4 + mav3;
matrices.maxpower = maxpower;
setappdata(handles.figure1, 'mydata', matrices);
% Save the changes
end
else
h = errordlg('Error reading EMG data', 'Data Collection Error', 'modal');
uiwait(h);
calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
else
h = errordlg('Device failed to start', 'Data Collection Error', 'modal');
uiwait(h);
end
%end of while loop for collecting cylinder grasp data

% --- Function to establish max power for disk or precision grasp
function establishDiskPower_Callback(hObject, eventdata, handles)
% hObject    handle to establishPower (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

samplesPerChan = 2000;
X = zeros(samplesPerChan, 4); % Preallocate matrix for 5 seconds of data

% Create pointers for EMG device control
read = int32(0);
readPtr = libpointer('int32Ptr', read);
taskHandle = uint32(0);
taskPtr = libpointer('uint32Ptr', taskHandle);

% Update the display to a grasp position
grasps = getappdata(handles.figure1, 'myimages');
grasps.current = 'disk';
setappdata(handles.figure1, 'myimages', grasps);
update_classDisplay(hObject, eventdata, handles);

doneGrasp = false;
while ~doneGrasp
    ConfigFlag = calllib('EMGDLL', 'Configure_EMG_RecordMode', 4, samplesPerChan, taskPtr.value);
    taskPtr.value
    if ConfigFlag == 0
        doneStart = false;
        while ~doneStart
            set(findobj('Tag', 'statusbar'), 'String', 'Training Mode:
            Establish maximum power for precision grasp'); % Update status bar
            selection = questdlg('Apply maximum grasp tension to a small
            item for 2 seconds.', 'Establish Max Power', 'Start', 'Cancel', 'Start');
            if strcmp(selection, 'Cancel')
                selection2 = questdlg('Are you sure you want to cancel
                grasp power training?', 'Cancel Max Power Training?', 'Yes', 'No', 'No');
                if strcmp(selection2, 'Yes')
                    return;
                end
            end
            doneStart = true;
        end
    end
    StartFlag = calllib('EMGDLL', 'Start_EMG', taskPtr.value);
    if StartFlag == 0
        [ReadFlag, Y, numread] = calllib('EMGDLL', 'Read_EMG', taskPtr.value, X', samplesPerChan, readPtr);
        if ReadFlag == 0
            readData = Y'*1000;
            % Need to multiply by 1000 to convert to mV
            matrices = getappdata(handles.figure1, 'mydata');
            % Get classifier data
            matrices.EMG = readData; % Store the new data in EMG
            setappdata(handles.figure1, 'mydata', matrices);
            % Save the changes
            update_4EMGDisplay(hObject, eventdata, handles);
        selection2 = questdlg('Do you want to keep this
        reading?', 'Keep Disk Power Data?', 'Yes', 'No', 'Yes');
        if strcmp(selection2, 'Yes')
            doneGrasp = true;
        end
        % Determine max power using channels 3 and 4
        mav4 = FeaturesMAV(matrices.EMG(:, 4));
        mav3 = FeaturesMAV(matrices.EMG(:, 3));
        if isfield(matrices, 'maxpower')
            maxpower = matrices.maxpower;
        end
        maxpower(3) = mav4 + mav3;
        matrices.maxpower = maxpower;
        setappdata(handles.figure1, 'mydata', matrices);
        % Save the changes
    end
else
    h = errordlg('Error reading EMG data', 'Data Collection
    Error', 'modal');
    uiwait(h);
end
    calllib('EMGDLL', 'Stop_EMG', taskPtr.value);
else
    h = errordlg('Device failed to start', 'Data Collection
    Error', 'modal');
end
uiwait(h);
end
end % end of ConfigFlag check

end %end of while loop for collecting disk grasp data

% --- Function to establish max power for key or lateral grasp
function establishKeyPower_Callback(hObject, eventdata, handles)
    % hObject    handle to establishPower (see GCBO)
    % eventdata  reserved - to be defined in a future version of MATLAB
    % handles    structure with handles and user data (see GUIDATA)

    samplesPerChan = 2000;
    X = zeros(samplesPerChan,4); % Preallocate matrix for 5 seconds of data

    % Create pointers for EMG device control
    read = int32(0);
    readPtr = libpointer('int32Ptr', read);
    taskHandle = uint32(0);
    taskPtr = libpointer('uint32Ptr', taskHandle);

    % Update the display to a grasp position
    grasps = getappdata(handles.figure1,'myimages');
    grasps.current = 'key';
    setappdata(handles.figure1,'myimages',grasps);
    update_classDisplay(hObject, eventdata, handles);

    doneGrasp = false;
    while ~doneGrasp
        ConfigFlag = calllib('EMGDLL','Configure_EMG_RecordMode',4,samplesPerChan,taskPtr);
        taskPtr.value
        if ConfigFlag == 0
            doneStart = false;
            while ~doneStart
                set(findobj('Tag','statusbar'),'String','Training Mode:
                Establish maximum power for lateral grasp'); % Update status bar
                selection = questdlg('Apply maximum grasp tension to a key for
                2 seconds.','Establish Max Power','Start','Cancel','Start');
                if strcmp(selection,'Cancel')
                    selection2 = questdlg('Are you sure you want to cancel
                    grasp power training?','Cancel Max Power Training?','Yes','No','No');
                    if strcmp(selection2,'Yes')
                        return;
                    end
                end
            doneStart = true;
        end
        StartFlag = calllib('EMGDLL','Start_EMG',taskPtr.value);
        if StartFlag == 0
            [ReadFlag, Y, numread] =
            calllib('EMGDLL','Read_EMG',taskPtr.value,X,samplesPerChan,readPtr);
            if ReadFlag == 0
                readData = Y'*1000;
                % Need to multiply by 1000 to convert to mV
matrices = getappdata(handles.figure1,'mydata');
% Get classifier data
matrices.EMG = readData; % Store the new data in EMG
setappdata(handles.figure1,'mydata',matrices);
% Save the changes
update_4EMGDisplay(hObject, eventdata, handles);
selection2 = questdlg('Do you want to keep this reading?','Keep Key Power Data?','Yes','No','Yes');
if strcmp(selection2,'Yes')
    doneGrasp = true;

    % Determine max power using channel 3 only
    mav3 = FeaturesMAV(matrices.EMG(:,3));
    if isfield(matrices,'maxpower')
        maxpower = matrices.maxpower;
    end
    maxpower(4) = mav3;
    matrices.maxpower = maxpower;
    setappdata(handles.figure1,'mydata',matrices);
    % Save the changes
end
else
    h = errordlg('Error reading EMG data','Data Collection Error','modal');
    uipwait(h);
end
calllib('EMGDLL','Stop_EMG',taskPtr.value);
else
    h = errordlg('Device failed to start','Data Collection Error','modal');
    uipwait(h);
end
end %end of ConfigFlag check

end %end of while loop for collecting key grasp data

% ________________________________________________________________
function LoadTrainingFiles_Callback(hObject, eventdata, handles)
% hObject    handle to LoadTrainingFiles (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Use Open File dialog to select EMG file

matrices = getappdata(handles.figure1,'mydata');
TrainData3D = [];
ClassSet = [];
tables = 0;
if isfield(matrices,'training') && isfield(matrices,'classSet')
    TrainData3D = matrices.training;
    ClassSet = matrices.classSet;
    [rows cols tables] = size(TrainData3D);
    if isempty(TrainData3D)
        tables = 0;
    end
end
[FileName, PathName] = uigetfile('*.txt', 'Select Multiple EMG Data Files of the Same Class', 'MultiSelect', 'on');

if ~isequal(PathName, 0)
    %set(findobj('Tag','statusbar'),'String',['Loading ' FileName]); % Update status bar
    entryComplete = false;
    while ~entryComplete
        displayString = 'Enter the class for these grasps: 1-Sphere, 2-Cylinder, 3-Disk, 4-Key';
        userEntry = inputdlg(displayString, 'Enter Grasp Class Number', 1, {'1'});
        if size(userEntry) > 0
            userNum = str2num(userEntry{1,1});
            if userNum >= 1 && userNum <= 4
                entryComplete = true;
            else
                errordlg('Invalid entry, try again.', 'Invalid Class Designator', 'modal');
            end
        end
    end
end
assignedClass = userNum;

if iscell(FileName) % cell array indicates multiple file selection
    [rows cols] = size(FileName);
    for n = 1:cols
        X = load(fullfile(PathName, FileName{:,n}));
        TrainData3D(:,1:3,tables+n) = X(:,1:3);
        ClassSet(tables+n, 1) = assignedClass;
    end
    %disp(['Added ',int2str(cols), ' tables.']);
else
    X = load(fullfile(PathName, FileName));
    TrainData3D(:,1:3,tables+1) = X(:,1:3);
    ClassSet(tables+1, 1) = assignedClass;
end
matrices.training = TrainData3D;
matrices.classSet = ClassSet;
setappdata(handles.figure1, 'mydata', matrices); % Save the changes
end

% --- Menu item for selecting to establish max power for each grasp
function maxtension_Callback(hObject, eventdata, handles)
% hObject handle to maxtension (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

% --- Executes on button pressed for Establish Grasp Power
function establishMaxTension_Callback(hObject, eventdata, handles)

% establish max power for each grasp
establishSpherePower_Callback(hObject, eventdata, handles);
establishCylinderPower_Callback(hObject, eventdata, handles);
establishDiskPower_Callback(hObject, eventdata, handles);
establishKeyPower_Callback(hObject, eventdata, handles);
% --- Executes on button press to stop Real-Time Mode
function stopRealTime_Callback(hObject, eventdata, handles)
% hObject    handle to stopRealTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

buttonFlag.stop = true;
setappdata(handles.figure1,'buttonFlag',buttonFlag);
set(findobj('Tag','pushbuttonrealtime'),'Enable','on');  % Enable real-time mode
set(findobj('Tag','stopRealTime'),'Enable','off');    % Disable real-time stop button
APPENDIX B

C++ SERVER CODE
EMG Robot Hand program for real-time user controlled synergetic grasp motions of the SDSU robot hand

Server Program

This program receives a connection from the EMG Robot Hand client program. This is the user interface front-end of the project. It has the ability to receive orders either through entered commands at the prompt, or by entering real-time mode with communication with Miller's EMG Classifier Program.

Communication is achieved by reading from a shared file named output1.txt.

Last modified Monday June 14, 2010.

Created by Luenin Barrios

SDSU Robotics Laboratory

*/

#include <winsock2.h>
#include <Windows.h>
#include <string.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <fstream>
#include "include\Stg_comn.h"
#include "include\stg_io.h"
#include "include\funcgen.h"
#include "basetsd.h"

void showMenu();

//Define the port used for socket communication
#define PORT (u_short)9974

//The character buffers used for sending and receiving messages
//from the client program
char messageBuffer[] = "THIS IS A TEST";
char robotBuffer[20];

//char repeatCommand[26];

main()
{
  using namespace std;

  //initialize winsock Dll
  WSADATA data;
  int status = -1;

  status = WSASStartup(MAKEWORD(2,2),&data);
  if(status != 0)
  {
    std::cout << "Server Error: WSASStartup Unsuccessful" << std::endl;
    exit(-1);
  }

  //Define the IP address and Port values used and create socket
  SOCKADDR_IN sa;
  sa.sin_port = htons(PORT);
  sa.sin_family = AF_INET;
  sa.sin_addr.s_addr = htonl(INADDR_ANY);

  SOCKET ss = socket(AF_INET, SOCK_STREAM, 0);

  if (ss == INVALID_SOCKET)
  {
    std::cout << "Client: Can't create socket. Error code = " << 
             WSAGetLastError() << std::endl;

    if(!WSACleanup())
      std::cout << "Client: WSACleanup failed. Error code = " << 
                 WSAGetLastError() << std::endl;

    exit(-1);
  }

  // Now we need to bind the socket to the port
  status = bind(ss,(LPSOCKADDR)&sa, sizeof(sa));

  //If socket creation failed, close the socket
  if(status == SOCKET_ERROR)
  {

// Now the second binding to listen for a connection
status = listen(ss,SOMAXCONN);

if(status == SOCKET_ERROR)
{
    std::cout << "Server: Can't bind the socket. Error code = " <<
    WSAGetLastError() << std::endl;
    closesocket(ss);
    exit(-1);
}

std::cout << "Server: The connection socket is successfully created" << std::endl;

// Accept a connection from the client

SOCKET css;
int addrlen = sizeof(SOCKADDR_IN);

SOCKADDR_IN ca;

// remember, after accepting, the new socket css will be used for communication
// in send and recieve etc

// Accept a client connection
std::cout << "Waiting for client..." << std::endl;
css = accept(ss,(LPSOCKADDR)&ca,&addrlen);

if(css == INVALID_SOCKET)
{
    std::cout << "Server: Can't accept a connection. Error code = " <<
    WSAGetLastError() << std::endl;
    closesocket(ss);
    exit(-1);
}

std::cout << "Server: A connection has been accepted from a client" << std::endl;

int n = send(css,messageBuffer,strlen(messageBuffer)+1,0);
if(n == SOCKET_ERROR)
{
    std::cout << "Client: send unsuccessful. Error = " << WSAGetLastError() << std::endl;
    exit(-1);
}

cout << "\n\n\n\n";

//Integer value to clear screen after more than 5 commands have been entered
int screenClear = 0;

    //Display the user interface menu
    char c;
    cout << "\nPID Controller for Robot Hand" << std::endl;
    cout << "Enter the letter of the corresponding command\n" << std::endl;
    cout << " m - Relative motion m <joint> <value>" << endl;
    cout << " g - Absolute motion g <joint> <value>" << endl;
    cout << " h - Reset all axes to home position" << endl;
    cout << " w - Wait for EMG" << endl;
    cout << " o - Object Space <grasp mode> <aperture>" << endl;
    cout << " 1 - Cylindrical" << endl;
    cout << " 2 - Spherical" << endl;
    cout << " 3 - Point" << endl;
    cout << " 4 - Lateral" << endl;
        cout << " d - Display all position values" << endl;
    cout << " c - Calibrate" << endl;
    cout << " u - Show Menu" << endl;
    cout << " x - Exit" << endl;
    do
    {
        //Clear screen and redisplay menu after 5 commands have been entered
        if(screenClear > 5)
        {
            system("cls");
            screenClear = 0;
            showMenu();
        }
    }
//Store the command
cout << "\nCommand: ";
std::cin >> c;

switch(c)
{
    //If u was entered, show menu
    case 'u': showMenu();
        break;
    //If Wait for EMG was entered
    case 'w': {
        //Notify the client program the command is w
        int n = send(css, "h", strlen(messageBuffer)+1, 0);

        string line;

        //If the output1.txt file exists, delete it. We don't want
        //old data stored there.
        remove("E:\Barrios\T400\blahtest\output1.txt");

        WIN32_FIND_DATA FindFileData;
        HANDLE hfind;

        //As soon as the output1.txt has been created by the EMG Classifier,
        //it will be found and read
        hfind = FindFirstFile("E:\Barrios\T400\blahtest\output1.txt", &FindFileData);

        cout << "Waiting..." << endl;

        //Search for output1.txt file
        while(hfind == INVALID_HANDLE_VALUE)
        {
            hfind = FindFirstFile("E:\Barrios\T400\blahtest\output1.txt", &FindFileData);
            //cout << "nothing yet" << endl;
        }

        //The file has been created with the real-time grasp integer value
        //The information can now be read and sent to the client
        cout << "We have the signal!" << endl;

        cout << "The file found is " << FindFileData.cFileName << endl;

        ifstream myfile("E:\Barrios\T400\blahtest\output1.txt");

        //Read the file for the integer grasp value
if (myfile.is_open())
{
    while (! myfile.eof())
    {
        getline(myfile,line);
        cout "What is the hand movement? "
<< line << endl;
    }

    myfile.close();
    }
else
    cout "UNABLE TO OPEN FILE" << endl;
    //Convert string to integer
    int EMGSignal = atoi(line.c_str());

    //Notify the client program which grasp was just entered.
    //For the purposes of this project, it was decided that if the user performs
    //a cylindrical grasp, the hand should go to a medium cylinder position(salt shaker)
    //For Spherical: small spherical(small yellow ping pong ball)
    //For Point: medium lateral(small black cap)
    //For Lateral: small block
    int s = send(css,"o",strlen(messageBuffer)+1,0);
    }

    //Cylindrical
    if (EMGSignal == 1)
    {

        //Act like the user just entered the equivalent object space command
        //and send the hand to the above described position
        int handMovement = 1*1000 + 42;

        itoa(handMovement,robotBuffer,10);
        cout "REAL TIME EMG COMMAND CYLINDRICAL " <<
        handMovement << endl;
        int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);

    }
    //Spherical
    else if (EMGSignal == 2)
    {
        //Act like the user just entered the equivalent object space command
        //and send the hand to the above described position
        int handMovement = 2*1000 + 39;

        itoa(handMovement,robotBuffer,10);
        cout "REAL TIME EMG COMMAND SPHERICAL " << handMovement << endl;

int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);

} //Point
else if(EMGSignal == 3)
{
    //Act like the user just entered the equivalent object space command
    //and send the hand to the above described position
    int handMovement = 3*1000 + 16;

    itoa(handMovement,robotBuffer,10);
    cout << "REAL TIME EMG COMMAND PRECISION " << handMovement << endl;
    int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);
}
//Lateral
else if(EMGSignal == 4)
{
    //Act like the user just entered the equivalent object space command
    //and send the hand to the above described position
    int handMovement = 4*1000 + 6;

    itoa(handMovement,robotBuffer,10);
    cout << "REAL TIME EMG COMMAND LATERAL " << handMovement << endl;
    int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);
}

break;

//For object space command
case 'o':{
    screenClear++;
    //Notify client object space command was entered
    int n = send(css,"o",strlen(messageBuffer)+1,0);

    //Save the grasp mode and object diameter
    int grasp;
    int aperture;

    cin >> grasp;
    //Check to ensure values are legitimate
    if(grasp < 0 || grasp >4)
    {
        cout << "Object space value must be 1-4" << endl;
        int b = send(css,"b",strlen(messageBuffer)+1,0);
    }
//cin >> aperture;
    cin.clear();
    cin.ignore(100,\n');
    break;
}
cin >> aperture;

//The aperture or object diameter values cannot exceed certain
//values. The limits vary for each grasp mode.
if((aperture < 6 || aperture > 25) && grasp == 4)
{
cout << "Lateral aperture size must be between 6 and 25 mm" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if((aperture < 28 || aperture > 56) && grasp == 1)
{
cout << "Cylindrical aperture size must be between 28 and 56 mm" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if((aperture < 39 || aperture > 81) && grasp == 2)
{
cout << "Spherical aperture size must be between 39 and 81 mm" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if((aperture < 9 || aperture > 28) && grasp == 3)
{
cout << "Precision aperture size must be between 9 and 28 mm" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}

//Send the grasp mode and object diameter to the client so it may
//perform the desired motion
int handMovement = grasp*1000 + aperture;

itoa(handMovement,robotBuffer,10);
//cout << "SENDING THE RIGHT OBJECT SPACE? " << handMovement << endl;
    int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);
    
cin.clear();
cin.ignore(100,'\n');
}
break;

/////////////////////////////////////////////////////////////////////////////////

//Relative motion command
case 'm':{
    screenClear++;
    //relativeMotion();
    //Notify the client relative motion was entered
    int n = send(css,"m",strlen(messageBuffer)+1,0);
    int joint;
    int angle;
    //Ensure joint entered is legitimate
    cin >> joint;
    if(joint < 0 || joint > 5)
    {
        cout << "Incorrect value for joint!" << endl;
        int b = send(css,"b",strlen(messageBuffer)+1,0);
        cin >> angle;
        cin.clear();
cin.ignore(100,'\n');
        break;
    }
    //Send the joint number to the client
    itoa(joint,robotBuffer,10);
    //cout << "THE REL joint " << robotBuffer << endl;
    int j = send(css,robotBuffer,strlen(robotBuffer)+1,0);
    Sleep(500);
    cin >> angle;
    //Retrieve the current angle value from the client
    int ja = recv(css,robotBuffer,sizeof(rob
    int curAngle = atoi(robotBuffer);
    //Check the final angle position is within bounds
    if(curAngle + angle > 85 || curAngle + angle < 17)
cout << "Axis absolute angle must be: 17 <= q <= 88" << endl;
int b = send(css, "b", strlen(messageBuffer)+1, 0);
cin.clear();
cin.ignore(100, 'n');
break;
}

// If it is, notify client to move to that position
Sleep(500);
itoa(angle, robotBuffer, 10);
// cout << "THE rel angle " << robotBuffer << endl;
int ang = send(css, robotBuffer, strlen(robotBuffer)+1, 0);
cin.clear();
cin.ignore(100, 'n');

break;
}// Absolute motion command
case 'g': {
    screenClear++;
    // Notify client absolute motion command was entered
    int n = send(css, "g", strlen(messageBuffer)+1, 0);

    int joint;
    int angle;
    // Check for valid joint number
    std::cin >> joint;
    if(joint < 0 || joint > 5)
    {
        cout << "Incorrect value for joint!" << endl;
        int b = send(css, "b", strlen(messageBuffer)+1, 0);
        cin >> angle;

        cin.clear();
        cin.ignore(10);
        break;
    }

    itoa(joint, robotBuffer, 10);

    // cout << "The JOINT is " << robotBuffer << endl;
    // Now send the joint number to client
    int j = send(css, robotBuffer, strlen(robotBuffer)+1, 0);
Sleep(500);
cin >> angle;

// Each joint has lower and upper limits. Ensure that the final
// Absolute angle value for it falls within the allowable limits
// If it doesn't, notify client of error

if(joint == 0 && (angle < 16 || angle > 89))
{
    cout << "Axis: 0 angle must be: 16 <= q <= 89" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if(joint == 1 && (angle < 18 || angle > 81))
{
    cout << "Axis: 1 angle must be: 18 <= q <= 81" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if(joint == 2 && (angle < 16 || angle > 87))
{
    cout << "Axis: 2 angle must be: 16 <= q <= 87" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if(joint == 3 && (angle < 16 || angle > 84))
{
    cout << "Axis: 3 angle must be: 16 <= q <= 84" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    cin.clear();
    cin.ignore(100,\n');
    break;
}
else if(joint == 4 && (angle < 30 || angle > 88))
{
    cout << "Axis: 4 angle must be: 30 <= q <= 88" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
    break;
}

// Limit thumb flexion to 30 degrees to avoid rotation collision
// against palm of hand
else if(joint == 4 && (angle < 30 || angle > 88))
{
    cout << "Axis: 4 angle must be: 30 <= q <= 88" << endl;
    int b = send(css,"b",strlen(messageBuffer)+1,0);
}
cin.clear();
cin.ignore(100,'\n');
break;
}
else if (joint == 5 && (angle < 2 || angle > 42))
{
cout << "Axis:5 angle must be: 2 <= q <= 42" << endl;
int b = send(css, "b", strlen(messageBuffer)+1,0);
cin.clear();
cin.ignore(100,'\n');
break;
}

/////////////////////////////////////////////////////////////////////////
//Now send the correct angle value
itoa(angle,robotBuffer,10);
//cout << "THE ANGLE IS " << robotBuffer << endl;
int ang = send(css,robotBuffer,strlen(robotBuffer)+1,0);

cin.clear();
cin.ignore(100,'\n');

}
break;
//Display all current angle values
case 'd':
{
//Retrieve the current angles from the client and display them
int d = send(css, "d", strlen(messageBuffer)+1,0);
for(int x = 0; x < 6; x++)
{
int ja = recv(css,robotBuffer,sizeof(robotBuffer)+1,0);
int angle = atoi(robotBuffer);
cout << "Axis: " << x << " Current Angle " << angle << endl;
}

cin.clear();
cin.ignore(100,'\n');
}

break;
//Calibrate command
case 'c':
{
//Notify client calibrate command was entered. It will
//perform the calibration
int n = send(css,"c",strlen(robotBuffer)+1,0);
    
cin.clear();
    cin.ignore(100,'n');
}
break;
//Reset all joints to home position
case 'h': {
    //Notify client home command was entered
    screenClear++;
    int n = send(css,"h",strlen(messageBuffer)+1,0);
    //goHome();
    cin.clear();
    cin.ignore(100,'n');
}
break;
//Exit command
case 'x': break;
default: {
    //cin.clear();
    //cin.ignore(100,'n');
    //cin.ignore(100,'n');
        
    //If an incorrect command was entered, display warning
    cout << "n\nYou have entered an illegal Command!" << endl;
    cout << "Try Again" << endl;
    cin.clear();
    cin.ignore(100,'n');
    //showMenu();
    //int n = send(css,"x",strlen(robotBuffer)+1,0);
}
break;
}
while(c != 'x');
int quitprogram = send(css,"x",strlen(robotBuffer)+1,0);

//Close socket upon program exit
closesocket(ss);
exito(0);

    return 0;
//Function that redisplays menu when user selects 'u' command

void showMenu()
{
    using namespace std;
    cout << "\n\n m - Relative motion m <joint> <value>" << endl;
    cout << " g - Absolute motion g <joint> <value>" << endl;
    cout << " h - Reset all axes to home position" << endl;
    cout << " w - Wait for EMG" << endl;
    cout << " o - Object Space <grasp mode> <aperture>" << endl;
    cout << " Grasp Mode " << endl;
    cout << " 1 - Cylindrical" << endl;
    cout << " 2 - Spherical" << endl;
    cout << " 3 - Point" << endl;
    cout << " 4 - Lateral" << endl;
    cout << " d - Display all position values" << endl;
    cout << " c - Calibrate" << endl;
    cout << " u - Show Menu" << endl;
    cout << " x - Exit" << endl;
}
APPENDIX C

C++ CLIENT CODE
/*
 EMG Robot Hand program for real-time user controlled
 synergetic grasp motions of the SDSU robot hand.

Client Program

This program connects to the EMG Robot Hand server
program to receive user issued commands for motion
control. This program invokes the STG driver to
perform the grasp motions. The file includes functions
for synergetic movements for Cylindrical, Spherical,
Lateral and Point grasp. See EMG Robot Hand GUI
for list of other commands this program may perform.

Last modified Monday, June 14 2010.

Created by Luenin Barrios

SDSU Robotics Laboratory

*/

#include <winsock2.h>
#include <Windows.h>
#include <string.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include <fstream>
#include "include\Stg_comn.h"
#include "include\stg_io.h"
#include "include\funcgen.h"
#include "basetsd.h"

//The IO control structure used for threaded driver commands
IOCTLVxdData IoCtlBuf2 = {0};

//Functions to calibrate hand, for absolute motion of the hand
//and for multi-threaded real-time movements for synergetic grasping
int CalibrateAll(int joint);
int CalibrateThumb(int joint);
void goAbsolute(int joint, int angle);
void AbsoluteForThread(int joint, int angle);
void whichAxis();
double jointAngle(int joint);
void tfun(DWORD n);
void tMove(DWORD n);
int CylinderGrasp(double diameter, int joint);
int SphericalGrasp(double diameter, int joint);
int PrecisionGrasp(double diameter, int joint);

//Matrix to store the
int jointValues [6][3];

//Matrix to store the minimum angle each joint can reach
int minJoint [6][1];

//Matrix to store the current angle for all joints
int currentAngle[6][1];

ePort_typ a2port(int c);

//Connect to port 9974 on the server(SKIPPER) computer
#define PORT (u_short)9974

//The server(SKIPPER) IP address
#define DEST_IP "192.168.1.10"

//Character array to hold the information sent and received
//by the server and client programs
char message[10000];

#if defined DRIVER_TEST_PROGRAM

#include <process.h>
#include ".\include\stgdefs.h"
#include ".\include\stgcmd.h"

//Enum needed to call the STG driver commands specifying joint to move
typedef enum {axis0, axis1, axis2, axis3, axis4, axis5, axis6, axis7, axisEnd} AxisType;

ClosedLoopController C;

unsigned int WINAPI Sampler(LPVOID p)
{
    for (;;)
{  
    Sleep(1);
    C.EncoderLatch();
    C.SamplePeriodTick();
}
return 0;
}
#endif  // DRIVER_TEST_PROGRAM

using namespace std;
main()
{
    //Open a path to the STG driver
    #ifdef DRIVER_TEST_PROGRAM

        unsigned ThreadId;
        C.Controller_Init(0, 0x240);

        _beginthreadex(NULL,
          0,
          Sampler,
          0,
          0,
          &ThreadId);

    #else

        if (OpenStgDriver() == STG_FAILURE)
        {
            printf("Unable to open path to driver.\n");
            exit(0);
        }
    #endif

    //The difference in degrees achieved by each joint
    //For example, joint 4 (the thumb flexion) can vary from
    //2 to 92 degrees => 92 - 2 = 90 degrees.
    jointValues[0][2] = 77;
    jointValues[1][2] = 65;
    jointValues[2][2] = 75;
    jointValues[3][2] = 70;
    jointValues[4][2] = 90;
jointValues[5][2] = 70;

//The minimum joint angle reachable by all joints
minJoint[0][0] = 13;
minJoint[1][0] = 17;
minJoint[2][0] = 13;
minJoint[3][0] = 15;
minJoint[4][0] = 0;
minJoint[5][0] = 0;

//Set the analog voltages for all joints to 0. This ensures
//that remnant voltage values do not interfere with each run
//of the program.
SetDAC(axis0, 0);
SetDAC(axis1, 0);
SetDAC(axis2, 0);
SetDAC(axis3, 0);
SetDAC(axis4, 0);
SetDAC(axis5, 0);

//Create a windows socket for communication via crossover cable.
WSADATA data;
int status = -1;

status = WSAStartup(MAKEWORD(2,2),&data);
if(status != 0)
{
    std::cout <<"Client: WSAStartup unsuccessful" << std::endl;
    exit(-1);
}
//Set the IP address and Port Number to be used.
SOCKADDR_IN da;
da.sin_family = AF_INET;
da.sin_port = htons(PORT);
da.sin_addr.s_addr = inet_addr(DEST_IP);

//Create the socket that will communicate
SOCKET cs = socket(AF_INET,SOCK_STREAM,0);
if(cs == INVALID_SOCKET)
{

std::cout << "Client: Can't create socket. Error code = " <<
WSAGetLastError() << std::endl;

if(!WSACleanup())
    std::cout << "Client: WSACleanup failed. Error code = " <<
    WSAGetLastError() << std::endl;
exit(-1);

//Now connect!

status = connect(cs,(LPSOCKADDR)&da,sizeof(da));

if(status == SOCKET_ERROR)
{
    std::cout << "Client: Can't connect with server. Error code = " <<
    WSAGetLastError() << std::endl;
    closesocket(cs);
    exit(-1);
}

std::cout << "Now connected to EMG Robot Hand server!" << std::endl;
int n = recv(cs,message,sizeof(message)+1,0);
//std::cout << "Did we get the message? " << std::endl << message << std::endl;
cout << "\n\n\n\n\n";

//Clear the character array that will store the messages
//sent and received
memset(message,0,sizeof(message));

//While the user has not exited, receive a message and parse it
//Depending on the command, perform the required action
do
{
    //Receive the message
    memset(message,0,sizeof(message));
    int cali = recv(cs,message,sizeof(message)+1,0);
    //cout << "What message do we have in the buffer? " << message << endl;

    //If Calibrate command has been selected, then call the calibrate
    //function.
    if(strcmp(message, "c") == 0)
    {
        //Store the encoder count corresponding to the finger being
// in the half open position and also the completely open position
// Do this for the thumb and the other fingers.
jointValues[5][0] = -CalibrateThumb(5);
jointValues[5][1] = 2*jointValues[5][0];
goAbsolute(5,42);
currentAngle[5][0] = 42;

jointValues[4][0] = -CalibrateAll(4);
jointValues[4][1] = 2*jointValues[4][0];
goAbsolute(4,88);
currentAngle[4][0] = 88;

for(int i = 0; i < 4; i++)
{
    jointValues[i][0] = -CalibrateAll(i);
    jointValues[i][1] = 2*jointValues[i][0];
}

// Now send the fingers to their home position. Home position is
// the hand completely open.
goAbsolute(3,84);
currentAngle[3][0] = 84;
goAbsolute(1,81);
currentAngle[1][0] = 81;
goAbsolute(2,87);
currentAngle[2][0] = 87;
goAbsolute(0,87);
currentAngle[0][0] = 87;

// If the command is Absolute Motion, call the absolute motion function
else if(strcmp(message, "g") == 0)
{
    int joint;
    int b = recv(cs,message,sizeof(message)+1,0);
    if(strcmp(message, "b") == 0)
    {
        continue;
    }
    else
    {
        //
    }
//Retrieve joint number from the command
joint = atoi(message);
}
//Retrieve the desired absolute angle
int angle;
int ang = recv(cs,message,sizeof(message)+1,0);
if(strcmp(message,"b") == 0)
{
    continue;
}

//Convert to integer for use by the Absolute motion function
angle = atoi(message);
currentAngle[joint][0] = angle;
goAbsolute(joint,angle);

//Call the absolute motion function with the desired joint and angle
currentAngle[joint][0] = angle;
goAbsolute(joint,angle);

//If the command is relative motion
else if(strcmp(message, "m") == 0)
{
    int joint;
    int j = recv(cs,message,sizeof(message)+1,0);
    if(strcmp(message,"b") == 0)
    {
        continue;
    }
    else
    {
        //Retrieve the joint number from the command
        joint = atoi(message);
    }

    //Obtain the current angle of the joint. Used to ensure the overal
    //movement does not result in a value beyond the allowable limits.
    int curAng = jointAngle(joint);
    itoa(curAng, message,10);
    int ja = send(cs,message,strlen(message)+1,0);
    Sleep(500);

    int ang = recv(cs,message,sizeof(message)+1,0);
    if(strcmp(message,"b") == 0)
{ continue;
}

// If the relative angle is permissible, then simply add
// it to the current joint angle and issue the Absolute motion
// function to that final value
int angle = atoi(message);

goAbsolute(joint, currentAngle[joint][0] + angle);
currentAngle[joint][0] = currentAngle[joint][0] + angle;

} // Send all fingers to their home position
else if(strcmp(message, "h") == 0)
{

currentAngle[0][0] = 87;
goAbsolute(0,87);

currentAngle[1][0] = 81;
goAbsolute(1,81);

currentAngle[2][0] = 87;
goAbsolute(2,87);

currentAngle[3][0] = 84;
goAbsolute(3,84);

currentAngle[4][0] = 88;
goAbsolute(4,88);

currentAngle[5][0] = 42;
goAbsolute(5,42);

} // For the display all joint angles command
else if(strcmp(message, "d") == 0)
{
// For each joint, find its current angle and then send
// it back to the server program so that it may display it
for(int x = 0; x < 6; x++)
{
    int angle = jointAngle(x);
    itoa (angle,message,10);
    int ja = send(cs,message,strlen(message)+1,0);
    Sleep(185);
} //For Object space command
else if(strcmp(message, "o") == 0)
{

//Create 6 handles and thread ids
HANDLE k[6];
DWORD threadID[6];

//Each time an object space command is entered, first send all joints
//to their home position
k[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
384, 0,&threadID[3]);
k[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
86, 0,&threadID[0]);
k[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
287, 0,&threadID[2]);
k[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
181, 0,&threadID[1]);

k[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
488, 0,&threadID[4]);
k[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID*)
542, 0,&threadID[5]);

//Wait until the hand has reached home position before continuing
while(WaitForMultipleObjects(6,k,TRUE,0) ==WAIT_TIMEOUT)
{
    //cout << "MAKING object go home first! " << endl;
    Sleep(200);
}

//Receive the exact object space command from the server
int handPosition;
int b = recv(cs,message,sizeof(message)+1,0);
if(strcmp(message, "b") == 0)
    {
        continue;
    }
else
{
    //Parse the command to obtain the joint and object diameter.
    //The message was sent from the server as a 4 digit number.
    //For example, a small lateral would have been sent as 3009.
    //So the handPosition would be 3 and diameter would be 9
    handPosition = atoi(message);
    int objectSpace = handPosition/1000;
        //cout << "we do have the right objectspace right? " <<
        objectSpace << endl;

    int diameter = handPosition%1000;
        //cout << "What is our diameter " << diameter << endl;

    //Create 6 handles and threadIds for actually moving the hand
    HANDLE h[6];
    DWORD threadID[6];

    //If the command was Lateral movement
    if(objectSpace == 4)
    {
        //The approximation coefficients for lateral movement. These values
        //were calculated manually. See the section on Synergetic training
        //in Chapter 6 of thesis.
        double a = 16.9032;
        double b = -46.7939;
        double c = -61.7742;

        //Calculate the thumb rotation angle
        int qthumb5 = 501 + (int)(a*((b - diameter)/(c + diameter)));

        //Move the hand to the desired Lateral Position. Note that for this project
        //It was decided that joints 0, 1 and 2 should always go to the same angle
        //namely: 20, 21, and 30 degrees respectively. Joint 3 should remain at home
        //since opposition with the thumb is not possible if it moves down
        h[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *)
                        230, 0,&threadID[2]);
        h[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *)
                        121, 0,&threadID[1]);
h[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 20, 0,&threadID[0]);
h[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 486, 0,&threadID[4]);

//h[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 121, 0,&threadID[1]);
//h[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 230, 0,&threadID[2]);
h[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 384, 0,&threadID[3]);
//h[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 486, 0,&threadID[4]);

h[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) qthumb5, 0,&threadID[5]);

}

//For Cylindrical movement
else if (objectSpace == 1)
{
    //Perform the hand movement by calling the CylinderGrasp function with threads
    h[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (200 + CylinderGrasp(diameter, 2)), 0, &threadID[2]);
h[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (CylinderGrasp(diameter, 0)), 0, &threadID[0]);
h[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (300 + CylinderGrasp(diameter, 3)), 0, &threadID[3]);
h[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (100 + CylinderGrasp(diameter, 1)), 0, &threadID[1]);
h[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) 488, 0, &threadID[4]);
h[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) 542, 0, &threadID[5]);

//For Spherical movement
else if(objectSpace == 2)
{
    //Perform the hand movement by calling the SphericalGrasp function with threads
    h[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (300 + SphericalGrasp(diameter, 3)), 0, &threadID[3]);
h[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) 488, 0, &threadID[4]);
h[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (200 + SphericalGrasp(diameter, 2)), 0, &threadID[2]);
h[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *)
(SphericalGrasp(diameter, 0)), 0, &threadID[0]);
h[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (102 +
SphericalGrasp(diameter, 1)), 0, &threadID[1]);
h[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) 542, 0,
&threadID[5]);
}
//For Point movement
else if(objectSpace == 3)
{
    //Perform the hand movement by calling the PrecisionGrasp function with threads
    h[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 86,
    0,&threadID[0]);
h[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 181,
    0,&threadID[1]);
h[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 509,
    0,&threadID[5]);
h[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (200 +
    PrecisionGrasp(diameter, 2)), 0, &threadID[2]);
h[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) (300 +
    PrecisionGrasp(diameter, 3)), 0, &threadID[3]);
h[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove, (VOID *) 488, 0,
    &threadID[4]);
}
}
}

while(strcmp(message, "x") != 0);

//If the user has exited, set all joint voltage values to 0.
//
SetDAC(axis0, 0);
SetDAC(axis1, 0);
SetDAC(axis2, 0);
SetDAC(axis3, 0);
SetDAC(axis4, 0);
SetDAC(axis5, 0);

HANDLE t[6];
DWORD threadID[6];
//Now move the hand to home position. Used so that next time the program is run
//The first command issued (Calibrate) will not encounter any interference by the joints

\[ t[3] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 384, 0,&threadID[3]); \]

\[ t[0] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 86, 0,&threadID[0]); \]

\[ t[2] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 287, 0,&threadID[2]); \]

\[ t[1] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 181, 0,&threadID[1]); \]

\[ t[4] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 488, 0,&threadID[4]); \]

\[ t[5] = CreateThread(0,0,(LPTHREAD_START_ROUTINE)tMove,(VOID *) 542, 0,&threadID[5]); \]

//Wait for threads to finish
while(WaitForMultipleObjects(6,t,TRUE,0) == WAIT_TIMEOUT)
{
    //cout << "MAKING ALL THE JOINTZZAZZ GO HOME! " << endl;
    Sleep(200);
}

//Wait 2 secs after the server program has closed before closing sockets and ending
Sleep(2000);
closesocket(cs);
exit(0);

return 0;

}

ePort_typ a2port(int c)
{
    switch (c)
    {
    case 'a': return A_port;
    case 'b': return B_port;
    case 'c': return C_port;
    case 'd': return D_port;
    default: return A_port; // an error, input out of range
}
}

//Calibrate function for joints 0,1,2,3
int CalibrateAll(int joint)
int x = joint;
long qcurrent = 0;
long qcali = 0;
AxisType axisTest;
axisTest = AxisType(x);
// First send the hand to its open position
for(int y = 0; y < 160; y++)
{
    // std::cout << "pos dac" << std::endl;
    SetDAC(axisTest, -0x0500);
}

// Now zero the joint's encoder and startin from open position
// go all the way to closed position
Sleep(2000);
SetDAC(axisTest, 0);
ZeroEncoderOne(axisTest);
for(int z = 0; z < 160; z++)
{
    // std::cout << "testing" << std::endl;
    SetDAC(axisTest, 0x0500);
}
Sleep(3500);
// Set voltage to 0 to stop joint and now obtain total encoder count
SetDAC(axisTest, 0);
qcali = GetEncoderOne(axisTest);

// Get encoder count corresponding to finger in half open position
int mid = qcali/2;
ZeroEncoderOne(axisTest);

// Use the encoder count to send hand to half open position
while(abs(qcurrent + mid) >= 34)
{
    SetDAC(axisTest, -0x0490);
    qcurrent = GetEncoderOne(axisTest);
}

SetDAC(axisTest, 0);
// Set the halfway joint angle for the specified finger
currentAngle[0][0] = 51;
currentAngle[1][0] = 46;
currentAngle[2][0] = 50;
currentAngle[3][0] = 47;
return mid;

// Function that sends a joint to an absolute joint angle
void goAbsolute(int joint, int angle)
{
    AxisType axisMoving;
    axisMoving = AxisType(joint);

    long qcurrent = GetEncoderOne(axisMoving);

    //long qdesired = 1698 + angle*jointValues[joint][1]/jointValues[joint][2];
    //minJoint*(max/(counts/degree)) + angle*max(counts/degree)

    // Translate the desired angle into a corresponding encoder count so the hand
    // can be stopped when it reaches that value
    long qdesired = -minJoint[joint][0]*(jointValues[joint][1]/jointValues[joint][2]) +
                   angle*jointValues[joint][1]/jointValues[joint][2];

    long qdot = 0;

    // While hand hasn't reached encoder count, keep moving it
    while(abs(qcurrent - qdesired) >= 34)
    {
        qcurrent = GetEncoderOne(axisMoving);

        // PID controller for joint motion
        long u = .3*(qdesired - qcurrent) - .8*qdot;
        // std::cout << "UUU " << u << std::endl;

        // Unfortunately, it was found that using the PID controller reaches a voltage
        // value to low to make the fingers move. Therefore, below a threshold, just /
        // slowly move the joint
        if (u > 0.0 && u < 465 && (joint == 0 || joint == 4 || joint == 5))
            u = 0x0260;
        else if (u < 0.0 && u > -465 && (joint == 0 || joint == 4 || joint == 5))
            u = -0x0260;

        if (u > 0.0 && u < 900 && (joint == 1 || joint == 2 || joint == 3))
            u = 0x0460;
        else if (u < 0.0 && u > -900 && (joint == 1 || joint == 2 || joint == 3))
            u = -0x0510;
SetDAC(axisMoving, u);
long qnew = GetEncoderOne(axisMoving);

qdot = qnew - qcurrent;

SetDAC(axisMoving, 0);

//Function that returns the current joint angle
double jointAngle(int joint)
{
    return currentAngle[joint][0];
}

//Function for threading to move fingers in parallel
void tMove(DWORD n)
{
    int joint;
    int angle;

    //Find the desired joint and angle values
    //These values are received from the corresponding
    //grasp functions
    joint = n/100;
    angle = n%100;

    currentAngle[joint][0] = angle;
    Sleep(450);

    //Now move finger to desired angle
    AbsoluteForThread(joint, angle);
}

//Function to calibrate thumb
int CalibrateThumb(int joint)
{
    int x = joint;
    long qcurrent = 0;
    long qcali = 0;
    AxisType axisTest;
    axisTest = AxisType(x);
    //Send thumb to open position
    for(int y = 0; y < 35; y++)
SetDAC(axisTest, 0x0500);
}

Sleep(2000);
// Zero the encoder and begin moving it to closed position
SetDAC(axisTest, 0);
ZeroEncoderOne(axisTest);
for(int z = 0; z < 35; z++)
{
    SetDAC(axisTest, -0x0500);
}
Sleep(3500);
SetDAC(axisTest, 0);
// Find the total encoder counts from open to close
qcali = GetEncoderOne(axisTest);

int mid = qcali/2;
ZeroEncoderOne(axisTest);

// Now thumb to midpoint of open and closed position
while(abs(qcurrent + mid) >= 35)
{
    SetDAC(axisTest, 0x0400);
    qcurrent = GetEncoderOne(axisTest);
}

SetDAC(axisTest, 0);

currentAngle[5][0] = 42;
return mid;

} // Function for absolute motion used in threading
// The STG API software layer does not support multi-threading
// so STG driver does. So the IOCTLBuf structure must be used.
void AbsoluteForThread(int joint, int angle)
{
    AxisType axisMoving;
    axisMoving = AxisType(joint);
IoCtlBuf2.nCommand = GET_ENCODER_ALL;
SendStg(&IoCtlBuf2);
long qcurrent = IoCtlBuf2.alParams[(unsigned short)axisMoving];

//Translate desired joint angle into an encoder count
long qdesired = -minJoint[joint][0]*(jointValues[joint][1]/jointValues[joint][2]) +
angle*jointValues[joint][1]/jointValues[joint][2];

long qdot = 0;

//While the finger has not reached the desired count, keep moving it
while(abs(qcurrent - qdesired) >= 34)
{
    IoCtlBuf2.nCommand = GET_ENCODER_ALL;
    SendStg(&IoCtlBuf2);
    qcurrent = IoCtlBuf2.alParams[(unsigned short)axisMoving];

    long u = .3*(qdesired - qcurrent) - .8*qdot;

    //Below a certain voltage value, fingers no longer move. So supply enough
    //voltage to ensure they reach their final destinations
    if ( u > 0.0 && u < 465 && (joint == 0 || joint == 4 || joint == 5))
        u = 0x0260;
    else if ( u < 0.0 && u > -465 && (joint == 0 || joint == 4 || joint == 5))
        u = -0x0260;

    if ( u > 0.0 && u < 900 && (joint == 1 || joint == 2 || joint == 3 ))
        u = 0x0460;
    else if ( u < 0.0 && u > -900 && (joint == 1 || joint == 2 || joint == 3 ))
        u = -0x0510;

    IoCtlBuf2.nCommand = SET_DAC_ONE;
    IoCtlBuf2.nAxis = (unsigned short)axisMoving;
    IoCtlBuf2.alParams[0] = u;
    SendStg(&IoCtlBuf2);

    IoCtlBuf2.nCommand = GET_ENCODER_ALL;
    SendStg(&IoCtlBuf2);
    long qnew = IoCtlBuf2.alParams[(unsigned short)axisMoving];

    qdot = qnew - qcurrent;
}
//SetDAC(axisMoving, 0);
// Set finger voltage to 0 and stop moving joint
IoCtlBuf2.nCommand = SET_DAC_ONE;
IoCtlBuf2.nAxis = (unsigned short)axisMoving;
IoCtlBuf2.alParams[0] = 0;
SendStg(&IoCtlBuf2);

} // Function that uses Cylindrical approx. coefficients to calculate joint angles
int CylinderGrasp(double diameter, int joint)
{
    // Approx coefficients are different for each joint
    // and were calculate manually
    double a, b, c;
    if(joint == 0)
    {
        a = -41.2;
        b = 25.75556;
        c = -24.9;
    }
    else if(joint == 1)
    {
        a = -37.6;
        b = 20.25319;
        c = -16.5;
    }
    else if(joint == 2)
    {
        a = -34.2857;
        b = 24.9;
        c = -23.7;
    }
    else if(joint == 3)
    {
        a = -44;
        b = 14.97273;
        c = 0.3;
    }
    // return the final joint angle
    return 2 + (int)(a*(b - diameter)/(c + diameter));
    // again, add 2 so grasp won't be too tight
}

} // Function that uses Spherical approx. coefficients to calculate joint angles
int SphericalGrasp(double diameter, int joint)
double a, b, c;
if(joint == 0) {
    a = -162.0792;
    b = 22.4851;
    c = 108.3465;
}
else if(joint == 1) {
    a = -343.9072;
    b = 10.9865;
    c = 465.213;
}
else if(joint == 2) {
    a = -18.7998;
    b = 120.1875;
    c = -97.658;
}
else if (joint == 3) {
    a = -24;
    b = 76.3488;
    c = -77.6186;
}
//return the angle
return (int)(a*((b - diameter)/(c + diameter)));

int PrecisionGrasp(double diameter, int joint) {
    double a, b, c;
    if(joint == 2) {
        a = -61.4;
        b = -11.3616;
        c = 17.6;
    }
    else if(joint == 3) {
        a = -58.4;
    }
    //Approx coefficients are different for each joint
    //and were calculate manually
b = -11.0411;
c = 17.6;
}
return (int)(a*((b - diameter)/(c + diameter)));
}