ABSTRACT

KELLY, JOHN W. An Investigation and Expansion of Musculoskeletal Modeling and Analysis Techniques. (Under the direction of Dr. Edward Grant).

A better understanding of human movement and its underlying dynamics is essential in developing more effective methods of rehabilitation and in assisting the diagnosis of physical ailments. Currently, the methods for analyzing and modeling human motion lag behind the methods available for capturing motion data. Many software packages have been developed to close this gap; one of the most recent and promising packages is OpenSim. However, a major problem exists with OpenSim: it can only handle a small range of formats used for capturing motion. Here, a new software package was developed that allowed motion capture files in the C3D format to be used in OpenSim. There were still minor problems associated with OpenSim's ability to analyze this data, however, ultimately, the results obtained from this new software package proved to be as accurate as that obtained from an analysis conducted using the same data and a proprietary software package for musculoskeletal analysis. Also, like other modeling software, OpenSim can only calculate the parameters associated with motion; it does not reduce the data to relevant statistics or determine patterns related to motion. To address these limitations two large sets of motion data were analyzed: first a motion data set from recovering stroke patients, and second, a data set from healthy subjects. Features of interest were extracted from the data sets and used to create a pattern classifier that recognized the distinct motion patterns exhibited by recovering stroke victims. Experiments with this new proof-of-concept system proved that C3D motion capture data could be successfully imported into OpenSim and analyzed, and then important motion patterns could be extracted and classified to show abnormalities in human movement.

An Investigation and Expansion of Musculoskeletal Modeling and Analysis Techniques

by
John Wade Kelly

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Electrical Engineering

Raleigh, North Carolina

2008

APPROVED BY:

_____	_____
Dr. Carol Giuliani	Dr. H. Troy Nagle

_____
Dr. Edward Grant
Chair of Advisory Committee

DEDICATION

To Jessi

BIOGRAPHY

John Kelly was born November 19, 1984 in Oak Ridge, Tennessee. He grew up in Oak Ridge and in nearby Morgan County before coming to North Carolina State University (NCSU), where he would graduate in 2007 as a valedictorian with BS degrees in Electrical and Computer Engineering. As an undergraduate he was a Park Scholar, Department of Homeland Security Undergraduate Scholar, a member of Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu, and spent a large amount of time volunteering with Young Life. He continued his graduate work at NCSU as a Tau Beta Pi Fellow under the direction of Dr. Edward Grant in the Center for Robotics and Intelligent Machines (CRIM).

ACKNOWLEDGMENTS

For the past five years Dr. Edward Grant, my advisor, has been invaluable in giving me guidance and support. I owe him my gratitude for this and for helping me select this thesis topic. I would also like to thank my other committee members, Dr. Carol Giuliani and Dr. H. Troy Nagle.

Much of this work would not have been possible without the assistance of Steve Leigh, from the Center for Human Movement Science, Division of Physical Therapy at the University of North Carolina at Chapel Hill. His clinical experience and biomedical knowledge were vital in helping me to understand many aspects of the data that I was unfamiliar with due to my background in electrical engineering. The experience and knowledge of the other students in the CRIM were also extremely helpful and I consider myself lucky to have been able to work with such a group.

Last I would like to my family and my Creator, who have given me the ability and the support necessary to complete this project. My parents have given me the encouragement I needed and have helped me reach this accomplishment, and my brother has always provided an outstanding example to follow. My fiancée has shown tremendous patience as I've done this project on a tight time frame, and she will no doubt help me reach any accomplishments I might have in the future.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

- csv – Comma Separated Value
- CMC – Computed Muscle Control
- DOF – Degrees of Freedom
- GRFs – Ground Reaction Forces
- GUI – Graphical User Interface
- kNN – k-Nearest Neighbor
- ID – Inverse Dynamics
- IK – Inverse Kinematics
- MMSE – Minimum Mean Squared Error
- mocap – Motion Capture
- NIH – National Institute of Health
- PCA – Principal Component Analysis
- PDF – Probability Distribution Function
- RRA – the Residual Reduction Algorithm
- SIMM – Software for Interactive Musculoskeletal Modeling
- SVM – Support Vector Machine
- wrt – With Respect To

Chapter 1

Introduction

1.1 Motivation

In the United States about 700,000 people suffer from stroke each year, and this number is increasing as life expectancy increases [1]. Many of the stroke victims who survive are left with severe physical impairments. While these individuals make up one of the largest and most visible groups with physical ailments, they are only a fraction of those in society who live with injuries or disorders that inhibit the proper functioning of their body's musculoskeletal system. Not only do these ailments sharply decrease the quality of life of individuals, they also significantly increase the cost of patient care and place a tremendous burden on the health care system [2].

Problems with the human musculoskeletal system can be improved upon, or prevented, through the effective use of physical therapy. In order to develop more effective rehabilitation methods there needs to be a better understanding of the human musculoskeletal system. This is necessary to be able to effectively model, analyze, and classify human body movement. So by improving upon musculoskeletal modeling and analysis techniques, it will

simultaneously increase the quality of life for many patients and reduce the cost of long-term patient care.

The common process for analyzing human movement follows the basic structure given in the flowchart in Figure 1.1. One major problem with this process is that nothing is standardized. Two different labs performing the exact same experiment and analysis might place the markers in different locations, use different frames of reference, store the results in different file formats, and have different joint definitions for kinematics. Some of these differences are unavoidable and even necessary, but it would be good if parts of this process, such as the mocap data format, could be standardized to the point of allowing collaboration between research centers and easier interpretation of data. The tools for the process of analyzing motion, such as software for IK and ID calculations, also need to be made more widely available. Finally, it would be useful to demonstrate an effective method of analyzing large amounts of motion data. If all of this is done then problems with each part of Figure 1.1 from "Mocap Data" on will be alleviated.

A good starting point for addressing some of the needs just mentioned is OpenSim, a software system that is currently being developed for musculoskeletal modeling and analysis. OpenSim can be used for IK and ID calculations and is an attractive solution because: (1) it offers a broad range of capabilities, and (2) it is open source, and therefore free to download. However, the software is still in its development stage, and it is only compatible with a limited number of mocap systems. One of the biggest shortcomings of OpenSim is that it cannot handle C3D files, one of the most commonly used mocap file formats. Therefore, the

effectiveness of OpenSim will be increased when it can handle C3D files and other mocap systems that it cannot handle currently.

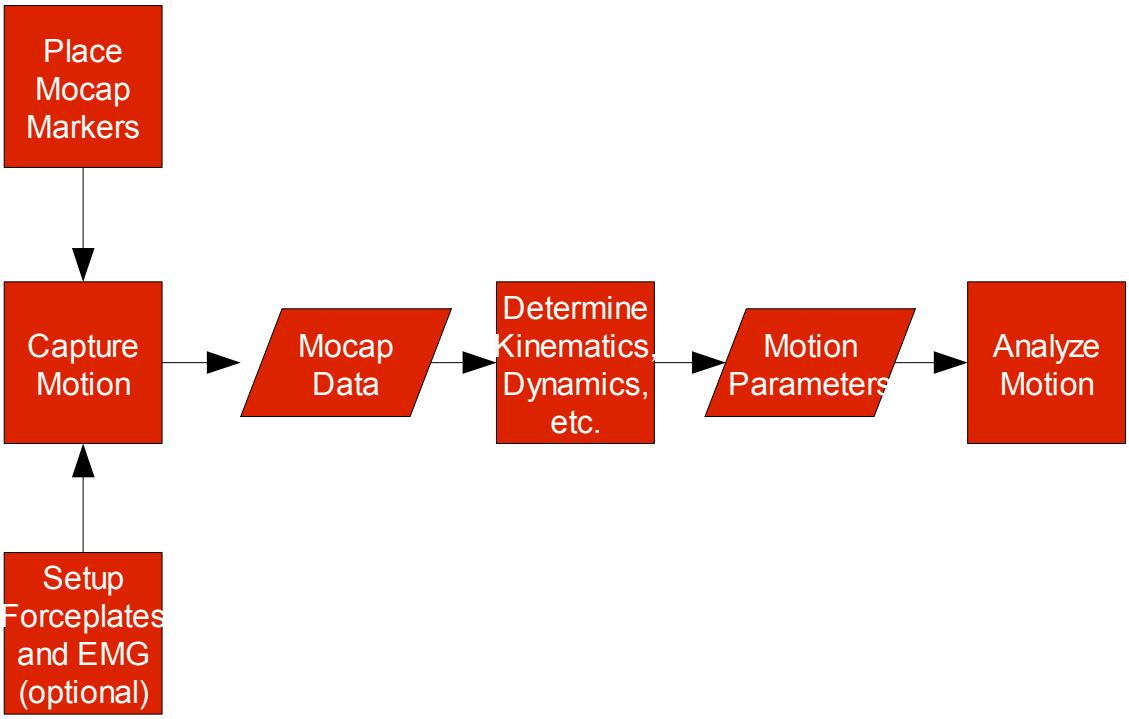


Figure 1.1: Common Process for Analysis of Motion

Problems associated with analyzing musculoskeletal movement do not end when motion is modeled, or after the kinematics and dynamics are computed. The data must be reduced to relevant features, features that can be used for comparison and classification purposes (the final step in Figure 1.1). This can help reveal the distinct patterns that classify a particular movement. For example, stroke victims exhibit movements that are distinct from healthy people, and it would be useful to be able to identify those features in their motion that most distinguish a stroke victim from a healthy person. With effective classifiers there is quicker diagnoses and more effective rehabilitation techniques will be developed that specifically address the needs of the individual patient.

1.2 Objectives

To address some of the needs identified above, the objectives of this research are to:

- Develop a method for converting mocap data stored in the C3D file format into formats that are usable by OpenSim.
- Validate the converted data by fitting it to an OpenSim model and playing the recorded motion by using the marker data to compute IK
- Evaluate the effectiveness of OpenSim by determining the accuracy of its IK calculations.
- Efficiently process and reduce a large set of motion data taken from stroke patients and healthy subjects in order to extract useful and interesting features from the data.
- Use features found in the processed data set to investigate possible patterns in motion that are unique to stroke victims.
- Demonstrate a proof-of-concept for pattern recognition in musculoskeletal data by creating a classifier that can identify the motion of arms that are affected by stroke.

1.3 Outline

Chapter 2 gives a basic overview of the features of OpenSim that are relevant to later chapters. This includes OpenSim's operation, capabilities, and some of its shortcomings. It also provides a basic overview of a few other software solutions for musculoskeletal modeling.

Chapter 3 covers the conversion of C3D mocap data to formats that can be applied to

OpenSim models.

Chapter 4 shows the results of importing data converted from C3D files into OpenSim. The resulting IK calculations are also compared to two other software systems.

Chapter 5 introduces a large set of motion data and describes how it was processed in order to glean important features and information from the data. The data set contains movement data recorded on both stroke patients and healthy subjects.

Chapter 6 discusses the patterns present in the large motion data set and demonstrates how these patterns can be used to create classifiers that automatically recognize physical disorders or injuries.

Chapter 7 gives possible directions for future work that can build upon this research.

The appendices at the end provide sample videos, additional figures, instructions for using the computer code created for this project, further information on some of the data used, and finally the computer code itself.

Chapter 2

An Overview of Musculoskeletal Software

2.1 OpenSim

2.1.1 OpenSim Overview

This section will familiarize readers with OpenSim and as such facilitate an understanding of work presented in later chapters. Good documentation is available at the OpenSim website (https://simtk.org/home/opensim) and with installations of OpenSim.

OpenSim is being developed as part of Simbios, an NIH center at Stanford University. The software is an open-source package for analyzing and visualizing musculoskeletal motion and dynamics [3]. Although OpenSim is not fully developed, it is already proving to be a valuable tool in the biomechanics community. It gives users the ability to view mocap data on a musculoskeletal model and to analyze motion through observing the IK, ID, and the muscle activations that create motion. It also allows users to simulate forward dynamics after altering the kinematics, muscle activation patterns, and other parameters. Any data calculated in OpenSim can easily be visualized using OpenSim's plotter.

The base software for OpenSim is written in C++, with a GUI written in Java [4]. The GUI (see Figure 2.1) provides tools for loading models and motions, and for scaling models to a static pose. It also gives users the ability to perform all the calculations mentioned in the paragraph above. There is a full muscle editor that allows users to configure muscle parameters, attachment points, and force length curves. A tool is also available for editing muscle excitations. Models can easily be moved to new poses, but currently not much else is provided in OpenSim for creating or editing models in the GUI. Markers, segments, joints, DOF, etc., must be manually configured in an XML-style file.

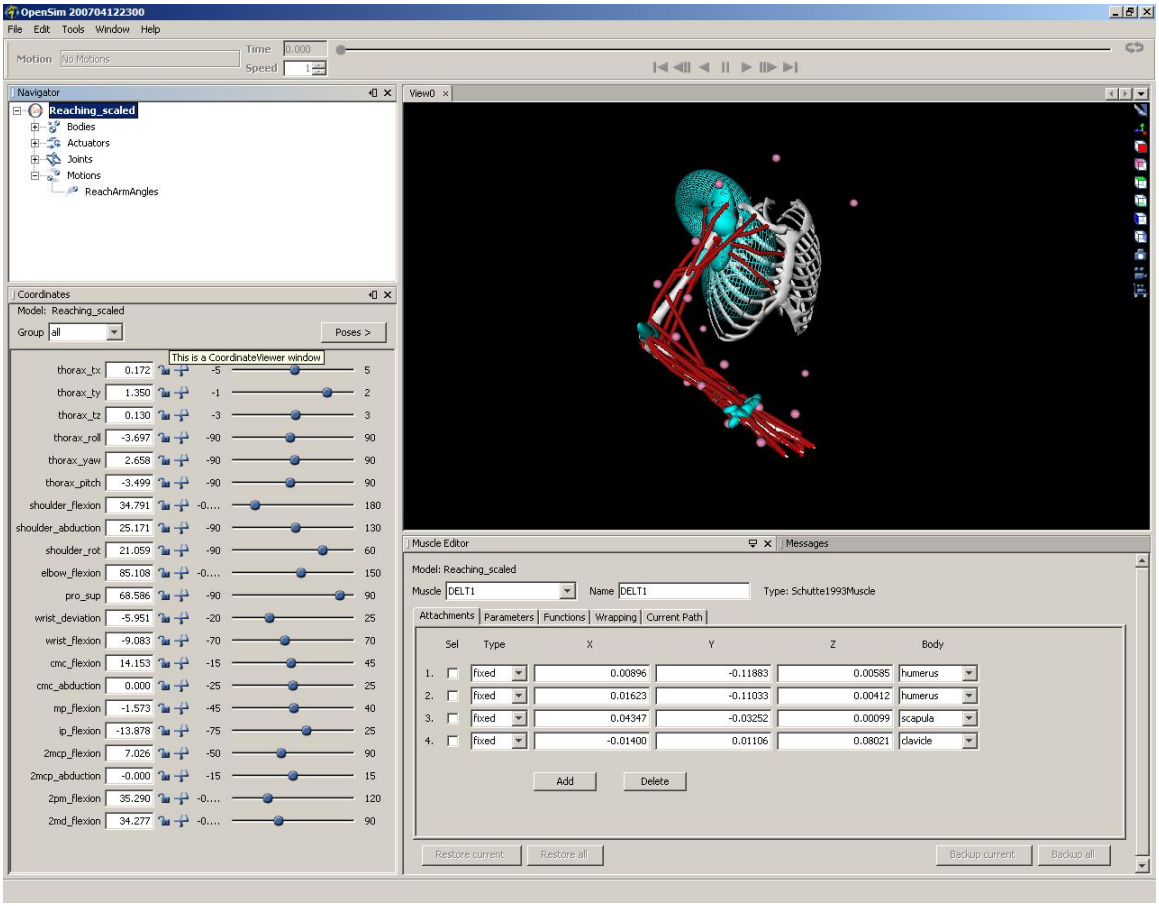


Figure 2.1: OpenSim GUI

2.1.2 Inverse Kinematics and Dynamics

2.1.2.1 Scaling the Model

Before IK and ID calculations can be performed, the model must be properly scaled to fit the subject from which the mocap data was recorded. This is a fairly simple process that merely requires the user to load a static pose file, one that gives all the appropriate kinematic coordinate markers for the model. Additional information on the subject, such as body mass, manual scale factors for each body segment, and custom measurement sets that record distance between markers, can also be loaded into the model. Each marker can also have a weight factor applied to it, to allow scaling within the model. It is recommended that markers attached to rigid points of the body, i.e., bones, be given larger weights. When scaling is complete, the dimensions and masses of each model segment are adjusted to match the mocap data. At this time the model markers are adjusted to match the calculated positions of the experimental markers.

2.1.2.2 Inverse Kinematics

Once scaling is complete, mocap data can be loaded to simulate the motion and calculate IK. This determines parameters and properties of the motion without regard to the masses or forces associated with the motion. For musculoskeletal modeling the most useful values calculated during IK are the joint angles. In OpenSim, IK calculation is typically done by loading a trc format mocap file that contains coordinates for those constructing the OpenSim model. If GRF data is present, it is also necessary to load a mot data file. Then the GRF data can be synchronized with the motion. As with scaling, each marker can be

assigned a weight for use during calculation.

Each frame of IK calculation in OpenSim is a weighted least squares optimization computation. During calculation, each marker has a corresponding marker error assigned to it. This marker error is defined as the difference between the experimental marker position from the mocap data and the virtual marker that is affixed to the model. The weight that is input for each marker is the coefficient used for the squared marker error when all of the squared marker errors are summed. The general coordinates that give the minimum sum of weighted squared errors are given as the result of the calculation. The equation for this calculation is given below (Equation 2.1) where q is the set of general coordinates, $x_i(q)$ is the position of the virtual marker, x_i^{exp} is the position of the experimental marker, and w_i is the marker weight [5].

$$\min_{q}\left[\sum_{i\varepsilon markers} w_i\left(x_i^{\exp}-x_i(q)\right)^2\right] \tag{2.1}$$

2.1.2.3 Inverse Dynamics

Dynamics, as opposed to kinematics, deals with the forces and moments involved in motion. To perform ID calculations OpenSim needs the following information: (1) the subject's mass and inertias, (2) force data (such as GRF) that was collected with the mocap data, and (3) the motion calculated during IK. It should be noted that OpenSim does not directly calculate GRFs and other forces from the force-plate data; this must be done prior to loading the data into OpenSim via a mot file. Also, only models based on the SimBody engine can be directly loaded into OpenSim and use the ID tool. Models based on the SIMM Kinematics engine must first be built with SD/Fast, from Symbolic Dynamics, Mountain

View, CA, before ID can be run. The trade-off is that the SIMM Kinematics engine allows for much more complicated joints to be defined and allows for the customization of muscle force curves. Soon these capabilities will be added to SimBody, meaning any SIMM model can be converted into a SimBody model [6].

The ID tool in OpenSim calculates all of the forces and moments that produce the motion calculated by the IK tool. This is a valuable tool for evaluating the stresses that are being placed on the various segments and joints in a musculoskeletal body. The information is also useful in determining the contributions of individual muscles to the calculated motion, or the external forces necessary to reproduce the motion.

2.1.3 Other OpenSim Capabilities

2.1.3.1 Computed Muscle Control

The CMC tool in OpenSim adjusts the model to be dynamically consistent and to calculate the excitations of each muscle during motion [7-9]. Before the muscle activations can be computed the inconsistencies that appear during calculations of the system dynamics should be minimized. These residual errors normally occur because of measurement errors, model errors, and/or errors in previous calculations. In OpenSim these errors are mitigated using RRA, which adjusts the center of mass of a model segment and allows for adjustments to the kinematics of motion [10]. The RRA should be used after the model has been scaled, and only after it has gone through both the IK and ID calculations.

After the model has been adjusted using RRA, CMC can be run. From this, the muscle activations that cause the dynamics of motion are determined. The result of the CMC

calculation is displayed visually when the motion sequence is played back, by virtue of the colors of the muscles changing as a function of their activations.

2.1.3.2 Forward Dynamics

The forward dynamics tool in OpenSim allows dynamics to be recalculated if the muscle excitations from CMC are edited. A separate tool is available for editing the muscle excitations. This is useful in viewing the effects that muscle excitations have on motion.

2.1.3.3 Plotting Data

OpenSim's plotting tool allows nearly any parameter of models, motions, or other calculated values to be plotted with respect to any other value. This is obviously extremely useful in visualizing data. The plotting tool is fairly intuitive and easy to use, and data from multiple models or calculations can be plotted simultaneously.

2.1.4 OpenSim Limitations

One of the biggest drawbacks of OpenSim is the limited number of mocap file formats that it can handle. Marker data can only be in either the trc or the mot format, and analog data can only be contained in mot files. Also, OpenSim cannot process GRFs, so analog data from a force-plate must be preprocessed before being imported into OpenSim. The file formats that can be used are consistent with Motion Analysis (http://www.motionanalysis.com) mocap systems, but most other mocap systems, such as Vicon (http://www.vicon.com) systems, are incapable of generating an appropriate mocap file format. This problem will be addressed in Chapter 3.

OpenSim also does not have any tools for editing or creating models. Models must

be created and/or modified using a separate program. Or, they must be manually generated by creating and editing an OpenSim model file. This is not as much of a problem as it first appears, because many models are freely available in the OpenSim community and more are always being created.

Certain advanced tools in OpenSim are complicated to use. For example, RRA and CMC can perform incorrectly. Any incorrect setting with these tools can cause OpenSim to crash or cause grossly inaccurate results. Forward dynamics requires RRA and CMC to be executed first, so it is important for the user to understand the CMC tool. These calculations need to be easier to perform, or better documentation needs to be available on the parameters that impact these calculations. Also, the iterative nature of many of these calculations (along with OpenSim's use of Java) causes high computational demands. Last, OpenSim's calculation methods show little regard for constraints placed on joints. At times the IK tool can get stuck attempting to move a joint to an illegal position, and the CMC tool will move joints to illegal positions without any constraint.

2.2 SIMM

2.2.1 SIMM Overview

SIMM is musculoskeletal modeling software created by MusculoGraphics, Inc., a division of Motion Analysis. SIMM is also the basis for OpenSim and many of the same people work on both software projects, so SIMM contains many of the same functions of OpenSim. The intention of SIMM is to provide the ability to do a full analysis on the body

of nearly any living creature [11]. As a stand-alone piece of software, though, SIMM has very limited capabilities. Most of the useful functions require the purchase of extra modules in addition to the base SIMM software. Also, since SIMM is a product of Motion Analysis, the software works best with Motion Analysis mocap systems.

2.2.2 SIMM Models

Models in SIMM are defined by a joint file and a muscle file. When a model is loaded in SIMM, many parameters of the model can be altered or analyzed via GUIs. This is probably the biggest advantage that SIMM currently has over OpenSim. The GUI tools allow models to be created or modified without having to manually edit the large XML-style files that make up the models. The graphical tools that SIMM provides for altering and/or analyzing models are discussed briefly below. These discussions will mostly focus on differences with regard to OpenSim, but each SIMM tool will at least be presented. Figure 2.2 shows a model that has been loaded into SIMM.

2.2.2.1 Model Viewer

The Model Viewer allows the user to perform simple operations such as changing the appearance of the model or model window. Any normal operation can be applied to the model (translation, rotation, and scaling) and any aspect relating to the position of joints can be changed as well. The one capability here worth noting is that different parts of the model can be aesthetically highlighted by using different shadings.

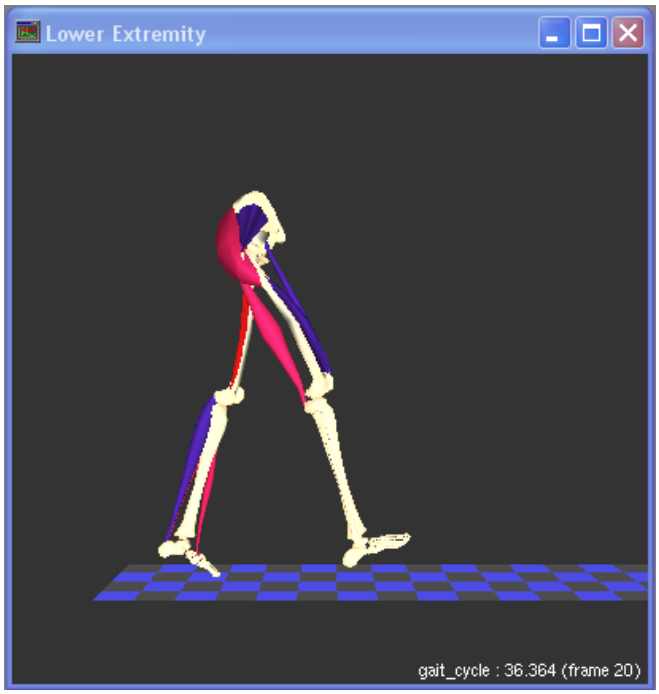


Figure 2.2: Lower Body Model in SIMM

2.2.2.2 Plot Maker and Plot Viewer

The Plot Maker can create plots of various computations performed on the musculoskeletal model. Additionally, external data can be imported and plotted. The Plot Maker works alongside the Plot Viewer, which allows the user to change the configuration or view of the plot window. These tools have the same function as OpenSim's plotter, but they are a little less intuitive to use. Figure 2.3 is a sample plot produced in SIMM.

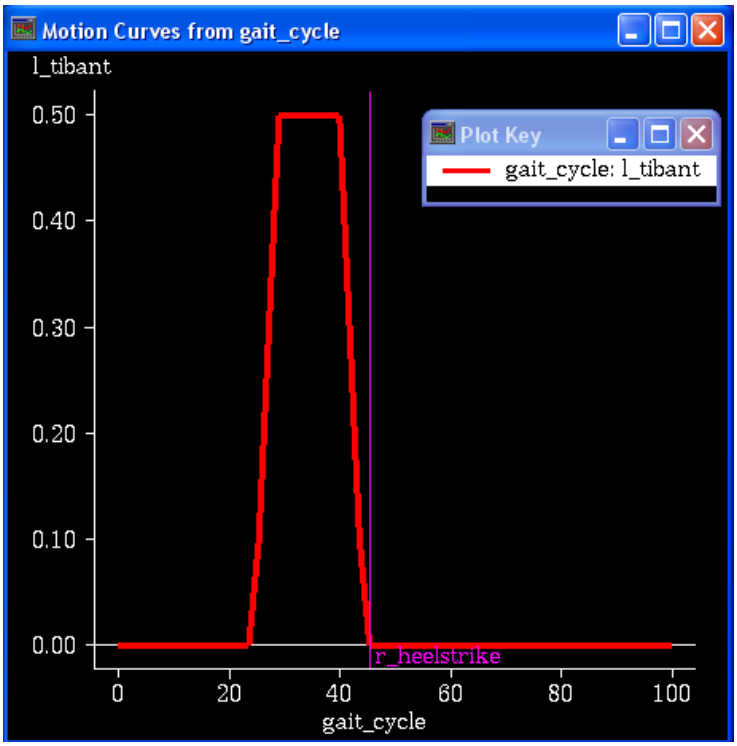


Figure 2.3: Sample Plot from SIMM

2.2.2.3 Muscle Editor and Wrap Editor

With the Muscle Editor nearly all of the parameters for each muscle can be altered. This provides capabilities very similar to those given by the muscle editor in OpenSim. The main advantage over OpenSim here is that wrap objects can easily be added and modified. This is a capability that is planned for OpenSim, but not yet implemented. SIMM used to have the advantage of allowing editing of muscle excitations, but a comparable tool was just recently added to OpenSim.

2.2.2.4 Segment Editor

The Segment Editor at first appears to be a major addition to the arsenal of tools that OpenSim does not offer, but it is fairly useless unless the Dynamics Pipeline module for SIMM is also purchased and installed. The Segment Editor can still make aesthetic changes, such as material and shading, but most of the other parameters that can be changed do not mean anything, unless dynamics calculations, i.e., those that require Dynamics Pipeline, are being made. These other parameters include mass, center of mass, inertia, force matte, and collision detection. If the model is imported into OpenSim these parameters will be present, but for SIMM models OpenSim requires SD/Fast for dynamics calculations.

2.2.2.5 Joint Editor

In the Joint Editor the kinematics of the joints can be manually changed. Each joint is defined as having six DOF, but some of these are locked out for certain joints by defining the DOF as a constant. Any DOF can be changed between a constant and a function, and of course the function or constant that defines the DOF can be changed as well. Any function for a DOF can either be a function of body segments or a function of the generalized coordinates. Being able to make these changes is important, and this is not something that OpenSim allows, but changing joint limits and DOF is one of the easiest things to do manually by editing the model file.

2.2.2.6 Gencoord Editor and Constraint Editor

This editor is a simple tool for modifying or viewing the generalized coordinates. The Gencoord Editor can also be used to add or change restraint functions for the generalized

coordinates. Some of these restraints are useless without Dynamics Pipeline. The Constraint Editor allows constraints to be added, which differ from restraints in that restraints limit the generalized coordinates or joint angles, and constraints provide methods of ensuring that certain points on one segment remain in contact with a certain area on another segment.

2.2.2.7 Bone Editor and Deform Editor

The Bone Editor allows for the modification of individual bones. This includes translations, rotations, and scaling, which is useful for creating bones for new models. The Deform Editor allows any segment in the model to be warped, bent, or twisted. All related bone vertices, wrap objects, muscle attachment points, etc. are updated accordingly.

2.2.2.8 Marker Editor and Motion Editor

The Marker Editor is fairly useless for SIMM unless the Real Time Motion Module or C3D Module is purchased and installed along with SIMM. Otherwise it's impossible to import marker data into SIMM and so adding markers to the model serves no purpose. With OpenSim, though, having access to SIMM's Marker Editor is a huge advantage. It allows markers to be graphically added or modified before the model is imported into OpenSim. Modifying markers in OpenSim requires a lot of guesswork and manual text editing. SIMM's Motion Editor allows editing of SIMM motion files, by either cropping the motion wrt time or inserting events into the motion (such as a heelstrike) that can be displayed on data plots.

2.2.3 Additional Modules

2.2.3.1 Real Time Motion Module

The Real Time Motion Module has the ability to import motion tracking files. When motion data is imported, a full-body model with 344 muscles is created in SIMM. The muscle attachment points, joint kinematics, and body segments are adjusted to fit the size of the subject's body. Additionally, analog signals such as EMG data can be imported with the motion, as well as kinetic data from OrthoTrak. The Real Time Motion Module can import recorded data, but as the name suggests it can also import data in real time. This allows the motion and streaming data plots to be displayed in SIMM while the motion is actually occurring [12].

It should be noted that the Real Time Motion Module is only capable of real time data input with Motion Analysis mocap systems. Post-processed files can be input in trc, trb, anc, anb, xls, and C3D. With the C3D_2_OSim package discussed in Chapter 3, though, OpenSim has the same capabilities with post-processed files. So without a Motion Analysis mocap system, the Real Time Motion Module has minimal benefit beyond what the OpenSim IK tool offers.

2.2.3.2 C3D Module

The C3D Module is essentially a subset of the Real Time Motion Module [Peter Loan, personal communication, February 2008]. It does not have the ability to do real time mocap data processing, it does not come with the full body model, and it can only input post-processed files in the C3D format. So basically, this module does for SIMM what the

C3D_2_OSim package from Chapter 3 does for OpenSim. The difference is that the C3D_2_OSim package is now free and open-source.

2.2.3.3 Dynamics Pipeline

The Dynamics Pipeline greatly expands the abilities of SIMM by allowing forward and inverse dynamics analyses [12]. The module also requires SD/Fast and Microsoft Visual Studio. Inverse dynamics requires an input of joint angles for the model, which usually requires that a motion has already been calculated. This means that the Real Time Motion Module and C3D Module are not required to use the Dynamics Pipeline, but it is highly useful to have one of them. Forward dynamics are calculated by specifying muscle excitations and optionally adding joint torques and external forces.

One important observation is that currently the Dynamics Pipeline cannot calculate the individual muscle forces that produce joint forces and moments, but that capability will be added soon. This capability is already available in OpenSim, though, with the CMC tool. OpenSim is also capable of performing forward and inverse dynamics on SimBody models, and on SIMM models with SD/Fast.

2.2.3.4 FIT Module

The FIT (Forward Inverse Dynamics Tool) is a subset of the Dynamics Pipeline. The difference is that each FIT Module is customized to only be able to perform dynamics calculations on one model, as opposed to a limitless number of different models [P. Loan, personal communication, February 2008]. SD/Fast and Visual Studio are not required with this module, though.

2.2.4 Interaction with OpenSim

The main benefit of using SIMM along with OpenSim is the ease with which new musculoskeletal models can be created and modified. SIMM contains GUIs that allow for interactive modifications of the model's segments and markers. Without SIMM these alterations must be done manually in an XML-style file. SIMM also has tools available for importing mocap data, but this normally requires purchasing additional modules.

2.3 The AnyBody Modeling System

2.3.1 AnyBody Overview

The AnyBody Modeling System is a product of AnyBody Technology, which is based in Denmark, and was originally developed at Aalborg University. It is capable of doing full body ID calculations with computation of joint kinematics, forces, torques, muscle excitations, etc. [13]. It is based on the AnyScript modeling language, but has many GUI tools available. It can also be run as a subroutine in MATLAB or with any C++ program. AnyBody is still being actively developed, and webcasts are available that contain valuable demonstrations of new capabilities and uses for the system. Newsletters and thorough documentation can also be found on the AnyBody website. Soon open forums will be available where users can exchange information, models, and ideas regarding the system.

2.3.2 Defining a Model

The AnyBody Modeling System is capable of performing a full analysis on the body of nearly any living creature. In general, AnyBody models (Figure 2.4 and Figure 2.5) are more detailed than SIMM and OpenSim models, but this is mostly aesthetic. Each body is defined using AnyScript, an object-oriented language developed specifically for modeling the structure and behavior of musculoskeletal systems. It allows the system to be defined as a class hierarchy, which facilitates the development of new systems using existing components. The syntax of AnyScript is similar to other object-oriented languages (Java, C++, etc.).

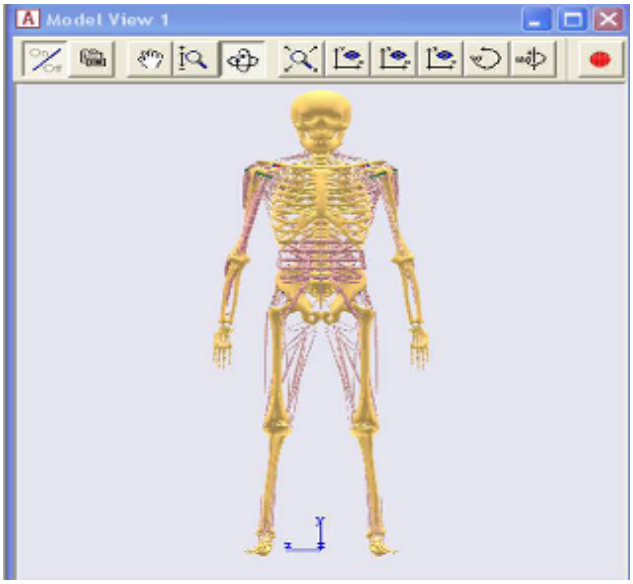


Figure 2.4: Full AnyBody model [14]

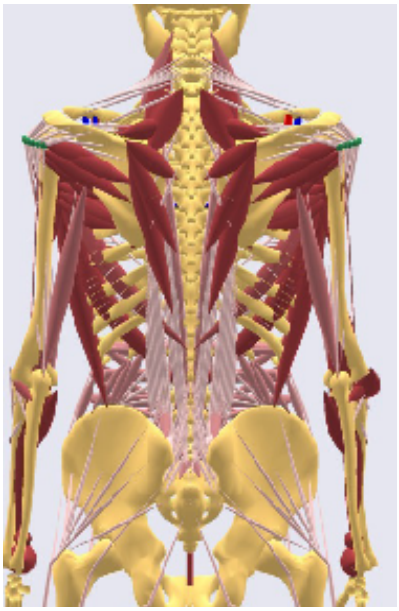


Figure 2.5:AnyBody back [14]

AnyScript is capable of defining bones, joints, muscles, movements, constraints, and external loads. The AnyBody System comes pre-loaded with a model for a human body, which can easily be manipulated to change sizes, joint angles, muscle strengths, external

forces, etc. It is also important to note that the system is capable of defining multi-body systems, such as a musculoskeletal system along with an exoskeletal robotic system.

2.3.3 Model Analysis

AnyBody relies on ID for analysis. So for a static model, running an analysis will use the position of the body and all known external forces (gravity and external loads) to determine the muscle activations, model dynamics, and model kinematics. Once this analysis is complete the muscles in the model will bulge according to their force and change colors depending on the muscle tone. The properties that the bulging and coloration represent can be customized. This feature is a major improvement over the CMC tool in OpenSim, which color codes muscle excitations. As with all other musculoskeletal modeling and analysis software discussed, charts and graphs can easily be created from any data available.

2.3.4 Movement Analysis

Once a model has been defined, motion is added using AnyScript. Code is also freely available for importing C3D mocap files into AnyBody. During motion the muscles will visibly bulge and change color. This can be seen in a video provided in Appendix A. From the motion a complete kinematic and dynamic analysis of the system can be done. As stated earlier, the analysis relies on ID. If given the system model and the motion, nearly any desired parameter of that motion can be calculated. At this time, however, the AnyBody system is incapable of calculating forward dynamics. This capability will most likely be added in the future, but right now the system cannot calculate motion if given a set of muscle

excitations or other parameters.

It is possible to perform "inverse-inverse dynamics." In this analysis, some parameters are designated as design parameters, and then a user-defined objective function is placed inside a loop that computes the ID. The objective function is then optimized in a computationally efficient way [Arne Klis, personal communication, January 2008]. For example, the minimal effort path of motion or the optimal muscle attachment points for an exoskeletal robotic system could be determined. Below is a chart demonstrating inverse-inverse dynamics parameter optimization for maximizing the metabolic efficiency of a bicycle as a function of seat height and seat horizontal position.

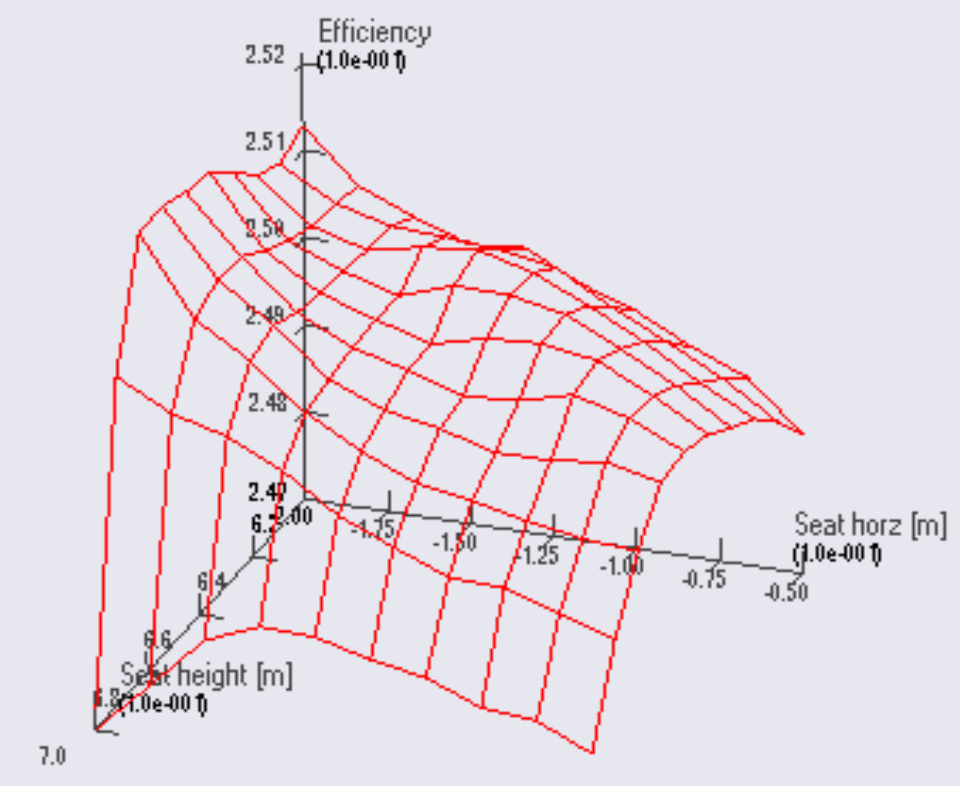


Figure 2.6: AnyBody Parameter Optimization Example [13]

2.4 Other Available Modeling Options

2.4.1 Lumo MotionVIZ

Lumo MotionVIZ is a product of RE-lioN, a company based in The Netherlands [15]. The software is capable of rendering a complete human skeleton consisting of 50 segments. Motion can be added to the model and various data can be calculated for each segment, such as force, center-of-mass, velocity, acceleration, etc. Muscles are not a part of the system and thus the software is more useful for rendering motion data than it is for analysis of motion data.

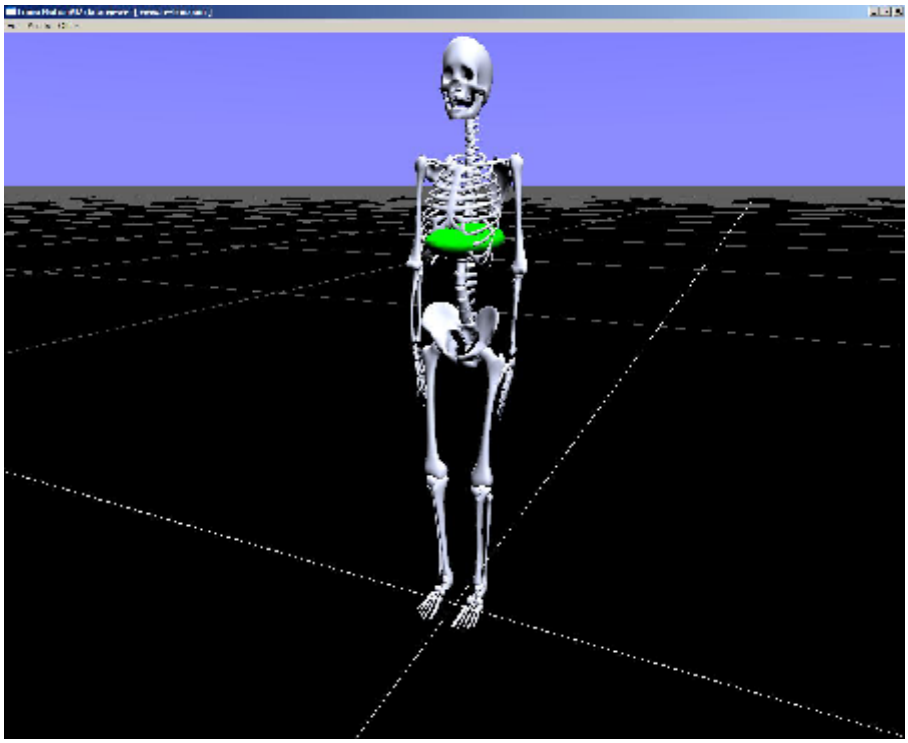


Figure 2.7: Lumo MotionVIZ Rendered Skeleton

2.4.2 Generic Wireframe Simulation Softwares

Many generic libraries are available that can simulate and analyze multi-segmented systems. These libraries were developed mostly for the purpose of robotic simulations, but they could also be applied to musculoskeletal systems. One of these libraries is DynaMechs, a software project that started at Ohio State University [16]. The advantages of using a system such as DynaMechs is that all the software is free and open-source. With the release of open-source musculoskeletal modeling software such as OpenSim, though, attempting to add musculoskeletal analysis to generic dynamic simulation software would be unnecessarily reinventing the wheel. A tremendous effort would need to be put forth to add the capabilities that OpenSim already has, so software packages such as DynaMechs are no longer practical solutions for musculoskeletal modeling and analysis.

2.5 Comparison of Musculoskeletal Software

The AnyBody Modeling System seems to produce better simulations and better ID calculations, but it is hindered by its current inability to do forward dynamics. It seems to be much better at handling multibody systems and external forces than SIMM or OpenSim, and the fact that the user has access to a programming language that can control AnyBody models and simulations gives an extra degree of control that makes custom or unique projects easier to handle. For example, the parameter optimization problem seems to work fairly well in AnyBody, but there might be difficulties with such a problem in SIMM or OpenSim. Of course the simulations in AnyBody look much nicer, with the bulging muscles and better

graphics detail, but this is mostly aesthetic due to the fact that a thorough analysis can never be done merely by looking at an animation of the motion. AnyBody has a final advantage over SIMM of having an extremely active community and a website with outstanding resources (webcasts, tutorials, samples, etc.).

SIMM's GUI is really nice for performing quick analysis and customizations. In AnyBody any modifications require code changes, and OpenSim's GUI tools don't have quite as many capabilities as SIMM. The GUI does take away some custom controls, though, which forces the user to buy additional modules to perform most useful tasks. With the addition of a few modules, very useful calculations can easily be done on recorded data. SIMM's main advantage over AnyBody is its ability to perform forward dynamics through the FIT Module or Dynamics Pipeline.

OpenSim has nearly all the advantages of SIMM, plus the benefit of being open-source and having an active development community. SIMM has a few abilities that OpenSim doesn't have yet, but these abilities are mostly just for convenience and do not warrant the price tag that comes with SIMM. OpenSim's advantages over AnyBody are forward dynamics and being open-source. AnyBody's inverse-inverse dynamics for parameter optimization appears to be extremely efficient and useful, though, and the fact that AnyBody runs its own programming language means that more custom and complex simulations can be run than in OpenSim. So if those two aspects of AnyBody are worth the cost, and direct forward dynamic calculation is not important, AnyBody might be the best choice for musculoskeletal modeling and analysis. Otherwise, OpenSim is the best solution.

Chapter 3

Conversion of C3D Motion Capture Files

3.1 Overview and Objectives

As stated in Chapter 2, one of the greatest limitations of OpenSim is the limited number of mocap file formats that it can handle. For OpenSim to become a truly useful tool in the musculoskeletal modeling and analysis community, the software must be made accessible to the largest possible number of researchers and clinicians. The quickest and best way to do this is to add OpenSim support for the C3D file format.

The C3D file format was first used by NIH, and is a public domain, binary file format for collecting synchronized 3D mocap and analog data [17]. It is supported by nearly all mocap system manufacturers and has the ability to store almost all useful data and information derived from mocap-based experiments [18]. Due to the format's versatility, standardization, and widespread use, a tool for importing C3D files into OpenSim would be quite valuable. It would also be desirable to use a public domain file format such as C3D for open-source software such as OpenSim. The main problem with C3D files is that they are in a binary format. Due to the availability of the binary structure, it was still possible to create a

program for converting C3D data to OpenSim format. This program is described here.

3.2 Preparing Necessary Data

The goal was not just to convert file formats, but to make the data directly applicable to OpenSim models. In order to do this, some extra information must be known about the mocap system and about the OpenSim model itself. Additionally, a rotation matrix must be created to transform the mocap data from its original world axes to the X-forward (sagittal axis), Y-up (longitudinal axis), Z-right (frontal axis) system used by OpenSim.

3.2.1 Loading OpenSim Model Information

OpenSim models have two important sets of parameters that are vital for loading mocap data. The first set is the gencoords. Each coordinate is a single DOF on the model, so collectively they completely define the position of the model. In mot files (Figure 3.4), which is one of the end file formats, there is a column of data for each gencoord. So in order to create the mot file, the gencoords for the OpenSim model must be defined.

The second set of parameters that must be defined in an OpenSim model are the marker labels. The marker labels will either be contained in the OpenSim model file itself, or in a separate XML file that defines a marker set for the model. It is highly possible, if not probable, that the names given to the actual markers in a mocap system during data collection will differ from the names used for the markers placed on an OpenSim model. So, in order for the marker data from the mocap system to be applied to the OpenSim markers, the marker labels must be renamed to match the labels used by the OpenSim model. Alternatively, the

markers in the OpenSim model can be renamed, but as will be shown later, it is easier and more efficient to rename the marker labels from the mocap system.

To extract these two sets of important parameters from the OpenSim model, a program was created that quickly gathers the needed information when the appropriate files are opened, i.e., the OpenSim model file, and if needed, the marker set XML file. The program only needs to be used once for each OpenSim model.

3.2.2 Loading Forceplate Information

This currently is the only step that needs to be performed manually, but it only needs to be performed if force-plates are used, and it only needs to be done once for each mocap experiment setup. This step has to be carried out manually because C3D files, while capable of storing force-plate information, do not always do so. In order to process GRF, the calibration information for each forceplate must be known. This information needs to be loaded into a MATLAB structure named FPinfo with the fields in Table 3.1.

Table 3.1: Forceplate Information Data Structure

Field Name	Description
calMatrix	6x6 calibration matrix for the forceplate
orientationMatrix	matrix for rotating from forceplate axes to mocap system axes
originTranslation	vector for translating to mocap system axes (after rotation)
gain	gain of the forceplate

3.3 Reading the C3D File

The binary C3D format consists of three main sections: (1) data, (2) standard parameters, and (3) custom parameters. The data structure for the absolute minimum information possible in C3D files is shown in Table 3.2. The standard parameters and the format of the data should remain consistent between mocap systems, labs, and testing conditions, but the custom parameters might vary [18]. For the reasons given, the data and the standard parameters are the vital data embedded within the file. To assist with the task of extracting this data readC3D.m, a program created by Alan Morris and modified by Jaap Harlaar, will be further modified to allow all necessary data to be efficiently read.

Table 3.2: The Basic C3D File Structure [18]

A single, 512 byte header section
A parameter section consisting of one or more 512-byte blocks
3D point/analog data section consisting of one or more 512-byte blocks

The most important change made to the readC3D.m program involved allowing it to read analog data as floating point values rather than integers. The format that the analog data is stored in varies between mocap systems, but it is easy to determine the correct format for any system through simple trial and error with the modified program. Once the data and parameters are read from the C3D file, they can be processed for use in OpenSim.

3.4 Extracting and Processing Data

3.4.1 Marker Data

3.4.1.1 Extracting Parameters

Although marker data can be read from the C3D file as a data array, marker labels and mocap parameters must be carefully extracted. All necessary parameters used for the 3D marker data are stored in the 'POINT' group found in the parameter section. The only parameter that the user really needs to be aware of is the location of the marker labels. By default, marker labels are stored in the 'LABELS' field of the 'POINT' group. In some systems this field places a limitation on the name of the label. For this reason, many people choose to store the actual marker label names in the 'DESCRIPTIONS' field instead. The user must tell the program which of these two fields will be used for the marker labels. Table 3.3 shows the important fields in the 'POINT' group, and how to use them. If any fields are missing a warning will be given and their values will be filled in with generic, assumed values. Even so, it is best if all values are present. The only field that is absolutely required is the 'RATE' field. Duplicate fields will also cause the parameter extraction to fail.

Table 3.3: Marker Parameter Fields from the C3D 'POINT' Group

Field Name	Contents
LABELS or DESCRIPTIONS	marker label names
USED	number of markers used
FRAMES	number of recorded 3D marker frames
SCALE	amount marker coordinates are scaled by
UNITS	unit of measurement for marker coordinates
RATE	sampling rate for marker data

3.4.1.2 Renaming Labels and Upsampling

In order for the marker data to be applied to an OpenSim model, the labels for the markers in the mocap data must match the labels of the markers on the model. Once the set of marker labels to be used by the OpenSim model has been loaded, a quick comparison with the mocap marker labels will reveal any inconsistencies. Any labels in the mocap data that are not recognized by the OpenSim model will be flagged. After seeing a list of labels used by the model, the user then inputs the equivalent OpenSim name for each unrecognized mocap marker. Markers can also be skipped if they are not a part of the model. Once this process is complete the marker label conversion information can be stored for later use. Then if the same mocap marker set is used again the lookup table can be utilized for automatic marker label conversion. Figure 3.1 contains a screenshot of this process.

Most of the time analog data is sampled faster than marker data. Once both sets of data are loaded into OpenSim the analog data is downsampled to match the marker data. To prevent data loss, and to maintain a high resolution in analog data, it is sometimes desirable to upsample the marker data to match the analog data. The user has the option of doing this.

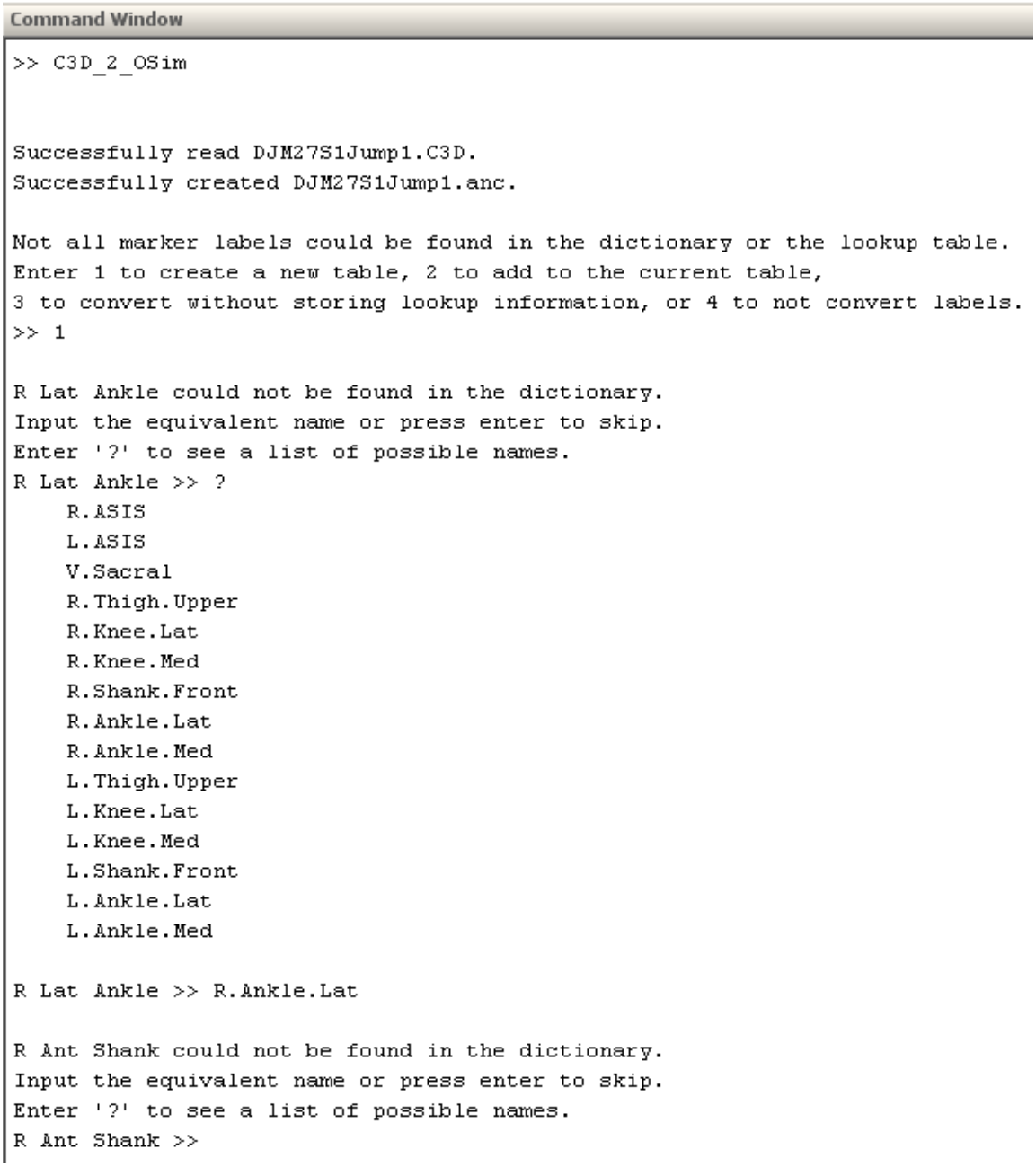


Figure 3.1: Marker Label Renaming Process

3.4.2 Analog Data

Analog data in C3D files most often consists of EMG data and forceplate data. The C3D file does not always contain information as to whether a data channel is EMG or forceplate, but as long as standard naming procedures are used for the analog channels, the channels are automatically separated into EMG and forceplate data. This is done by searching for sets of 6 channels in which the names take the following forms (not case-sensitive, * designates wild card, wild cards must match between channel): *f*x*, *f*y*, *f*z*, *m*x*, *m*y*, and *m*z*. Any such set of channels is forceplate data, and everything else is assumed to be EMG data.

Like the marker data, the analog data has important parameters that are stored in the C3D file. Analog parameters are stored in the 'ANALOG' parameter group and the names of the analog channels can be stored in either the 'LABELS' or the 'DESCRIPTIONS' field. The user must specify which field the analog channel names are located in. The other important fields in the 'ANALOG' group are shown in Table 3.4. As with the marker data, the only required field is 'RATE', but all fields should be present and any duplicate fields will cause the process to fail.

Table 3.4: Analog Parameter Fields from the C3D 'ANALOG' Group

Field Name	Contents
LABELS or DESCRIPTIONS	analog label names
UNITS	the units of each channel
USED	the number of analog channels used
GEN_SCALE	a scale factor that is applied to all channels
SCALE	scale factors that are applied to each channel individually
OFFSET	offset amounts for each analog channel
RATE	the analog data sampling rate

3.5 Writing Intermediate Files

3.5.1 trc Output

The trc file format is used by the IK tool in OpenSim. Normally they are created by Motion Analysis mocap systems (http://www.motionanalysis.com), where the files are in ASCII format and only contain 3D marker data [17,19]. Important information related to the data and the testing conditions are contained in a header, followed by a row of marker labels and by rows of marker coordinates. An example of the top of a trc file is shown in Figure 3.2. Writing the processed marker data from a C3D file into a trc file is a fairly simple process. Most of the header information is found in the C3D parameters. The few parameters that are not and can't be determined any other way, e.g., PathFileType, OrigDataRate, OrigDataStartFrame, and OrigNumFrames, are not important for OpenSim purposes, so generic values can be used. The values written to the trc file consistently match the coordinates recorded by the mocap system and stored in the C3D file.

PathFileType		4	(X/Y/Z)	DJM27S1Jump1.trc												
DataRate	CameraRate	NumFrames		NumMarkers	Units	OrigDataRate		OrigDataStartFrame			OrigNumFrames					
1200.00	120.00	1320	11	mm	1200.00	1	1320									
Frame#	Time	R.Ankle.Lat			R.Shank.Front					R.Knee.Lat				R.Thigh.Upper		
	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	Z2	X3	Y3	Z3	X4	Y4	Z4	X5	Y5
1	0.00000	-275.28391	212.81030		-286.71210	-306.33868	558.45911		-320.80746		-406.10059	605.46960					
2	0.00083	-271.90030	212.73936		-285.31978	-304.44101	557.91441		-320.08127		-403.89654	604.80137					
3	0.00167	-268.50277	212.67073		-283.94545	-302.51251	557.38260		-319.37889		-401.68464	604.14692					

Figure 3.2: Partial trc File Produced from C3D Data

3.5.2 anc Output

An anc file is the Motion Analysis analog data equivalent of a trc file. Such files are not used by OpenSim because OpenSim cannot process GRFs. These files are created from C3D data because the data contained within them is useful for reference and verification. Like trc files, the anc file contains a header with important information, followed by channel names, important channel information, and then rows of samples (see Figure 3.3). However, unlike trc files, not all of the important header information is available in the C3D data. Generic values are assigned to the unimportant fields, e.g., File_Type, Generation #, and Board_Type. The polarity is assumed to be bipolar, although this assumption can cause major problems if it is wrong.

File_Type:	Analog R/C ASCII	Generation#:	1										
Board_Type:	unknown	Polarity:	Bipolar										
Trial_Name:	DJM27S1Jump1	Trial#:	1	Duration(Sec.):	1.099167	#Channels:	14						
BitDepth:	12	PreciseRate:	1200.000000										
Name	sync	event	Fx1	Fy1	Fz1	Mx1	My1	Mz1	Fx2	Fy2	Fz2	Mx2	My2	Mz2
Rate	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200	1200
Range	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096	4096
0.000000		0.0146484	0.0537109	0.00488281	0.00488281	0.0244141	0.0195313	0.0195313						
0.000833		0.00976563	0.0537109	0.00488281	0.00488281	0.0244141	0.0195313	0.0195313						
0.001667		0.00976563	0.0537109	0.00488281	0.00488281	0.0244141	0.0195313	0.0195313						
0.002500		0.00976563	0.0537109	0.00488281	0.00488281	0.0244141	0.0195313	0.0195313						

Figure 3.3: Partial anc File Produced from C3D Data

The important fields that are missing in the C3D data are BitDepth and the Range for each channel. In a Motion Analysis anc file these values are used to convert a digital integer value into its true analog value. The C3D files that are the output of the Vicon system in the Physical Therapy Division at the University of North Carolina at Chapel Hill (UNC) already contain the true analog values in floating point format. This means that the BitDepth can be set to 12, i.e., the most common ADC bit depth, allowing the range to be set at 4096 for each channel. The standard equation for converting digital values from an ADC to the true analog values is given below (Equation 3.1) where D is the digital value and A is the analog value. This means that if the anc file is used and an attempt is made to perform this calculation the true analog data will be retained because $2^{12}/4096 = 1$. The analog values in the anc files consistently match those recorded by the mocap system.

$$A = \frac{D * 2^{BitDepth}}{Range} \tag{3.1}$$

3.6 Processing Ground Reaction Forces

3.6.1 Processing Data

Since OpenSim cannot process force-plate data directly, the data must first be preprocessed to calculate GRFs. This is done using the force-plate calibration information and a set of functions created at Stanford University for preprocessing trc and anc files for use in OpenSim. The functions were modified to process analog data extracted from C3D files.

3.6.2 Writing Data

Stanford functions are also used to write the gencoord mot files and the GRF mot files that are usable by OpenSim. The top of an example gencoord mot file is given in Figure 3.4 and a mot file containing GRFs produced by Stanford's functions is given in Figure 3.5.

```
name DJM27S1Jump1.mot
datacolumns 42
datarows 1321
range 0.000000 1.100000
endheader
                time              pelvis_tx              pelvis_ty              pelvis_tz
          0.00000000             0.00000000             0.00000000             0.00000000
          0.00083000             0.00000000             0.00000000             0.00000000
          0.00167000             0.00000000             0.00000000             0.00000000
```

Figure 3.4: Partial mot Gencoords File Produced from C3D Data

```
name DJM27S1Jump1_grf.mot
datacolumns 19
datarows 1320
range 0.000000 1.099167
endheader
                time         ground_force_vx         ground_force_vy         ground_force_vz
          0.00000000             0.00000000             0.00000000             0.00000000
          0.00083333             0.00000000             0.00000000             0.00000000
          0.00166667             0.00000000             0.00000000             0.00000000
```

Figure 3.5: Partial GRF mot File Produced from C3D Data

3.7 Conclusion and Data Flow Chart

Across different test subjects and testing conditions, the program quickly and accurately converts C3D data into formats that can be applied to OpenSim models. Figure 3.6 shows the output of the program for a successful conversion. If any warnings are given by the program it is because appropriate analog data doesn't exist in the C3D file. The basic data and program execution flow chart is given in Figure 3.7

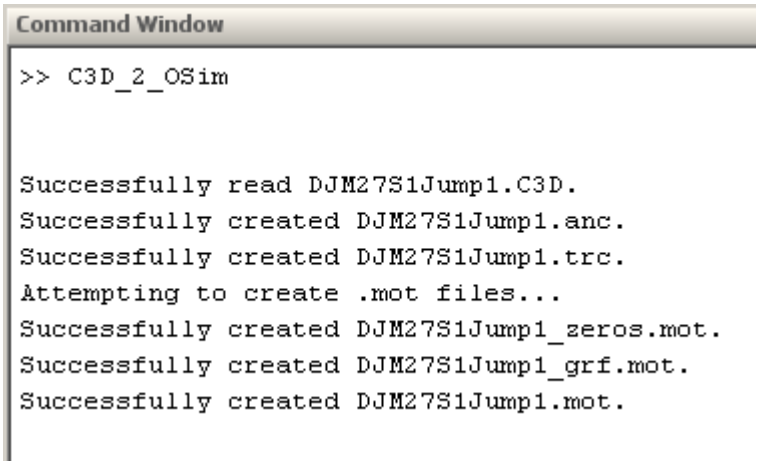


Figure 3.6: Successful Output from C3D Data Conversion

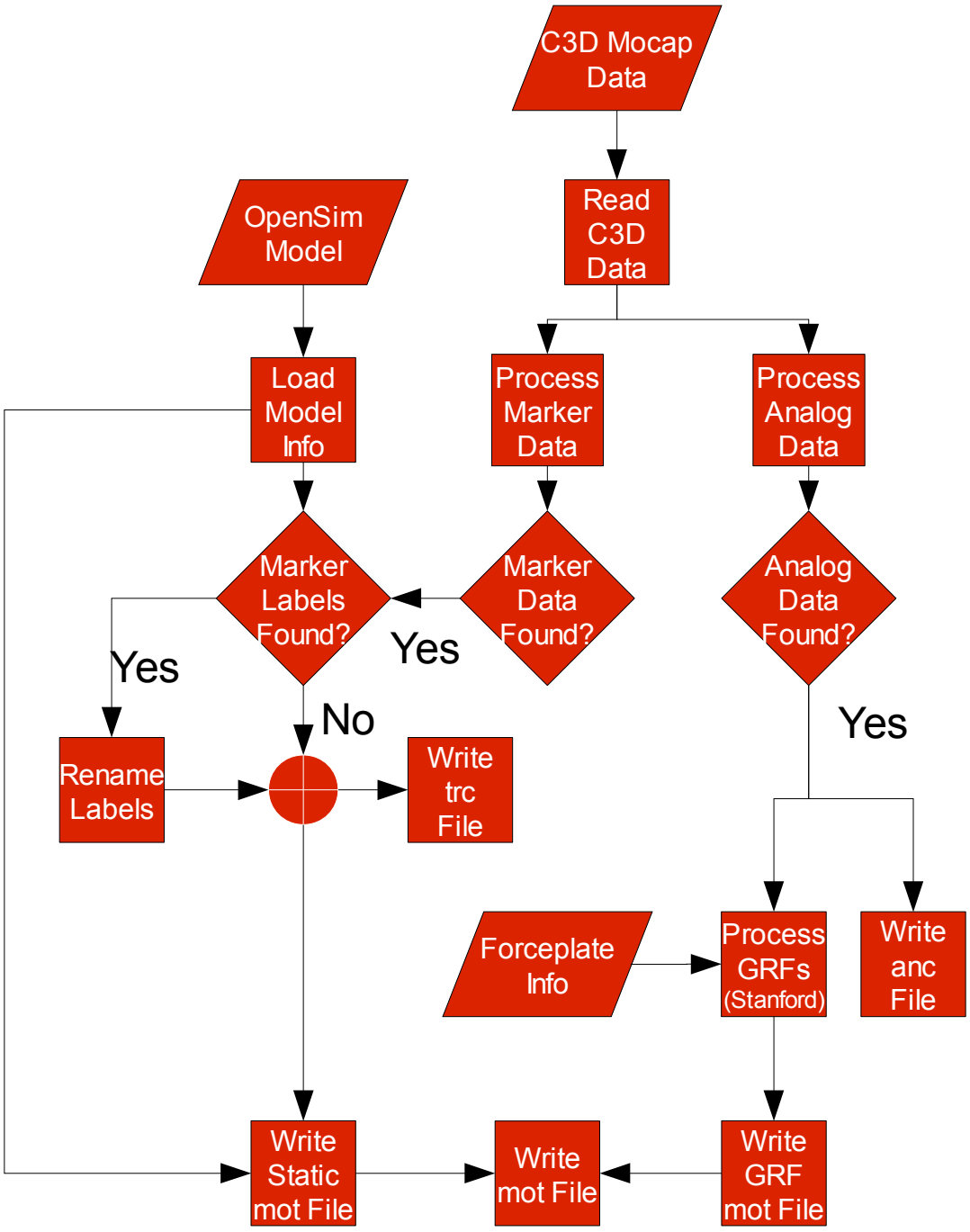


Figure 3.7: C3D Conversion Flow Chart

Chapter 4

OpenSim and C3D Conversion Analysis

4.1 Overview and Goals

The objective of this analysis is to determine not only the effectiveness of converting C3D data and applying it to OpenSim models, but to also determine the accuracy of OpenSim in using the data in some of its more basic tools. In order to do this, two sets of data have been collected: (1) upper body mocap data of a subject reaching, and (2) lower body mocap data of a subject jumping.

Both sets of data were collected by Vicon mocap systems and stored in the C3D format. The C3D data was then converted and applied to OpenSim models using the process described in Chapter 3. The IK tool was run on the data and the resulting joint angles were compared to the joint angles calculated by two other programs: MotionSoft (MotionSoft, Chapel Hill, NC) and Visual3D (C-Motion, Rockville, MD). For the jumping data only the MotionSoft joint angles are available. All C3D mocap data, as well as MotionSoft and Visual3D kinematic data, was collected by Steve Leigh at UNC.

4.2 Reaching Data

4.2.1 OpenSim Results

For the reaching data an upper extremity model was used, one that was developed by Holzbaur et al [20]. This model is freely available from the SimTK library. The result of scaling the model using the C3D data from the test subject's static pose is shown in Figure 4.1. In Figure 4.1 all joints are set to their neutral (zero) positions.

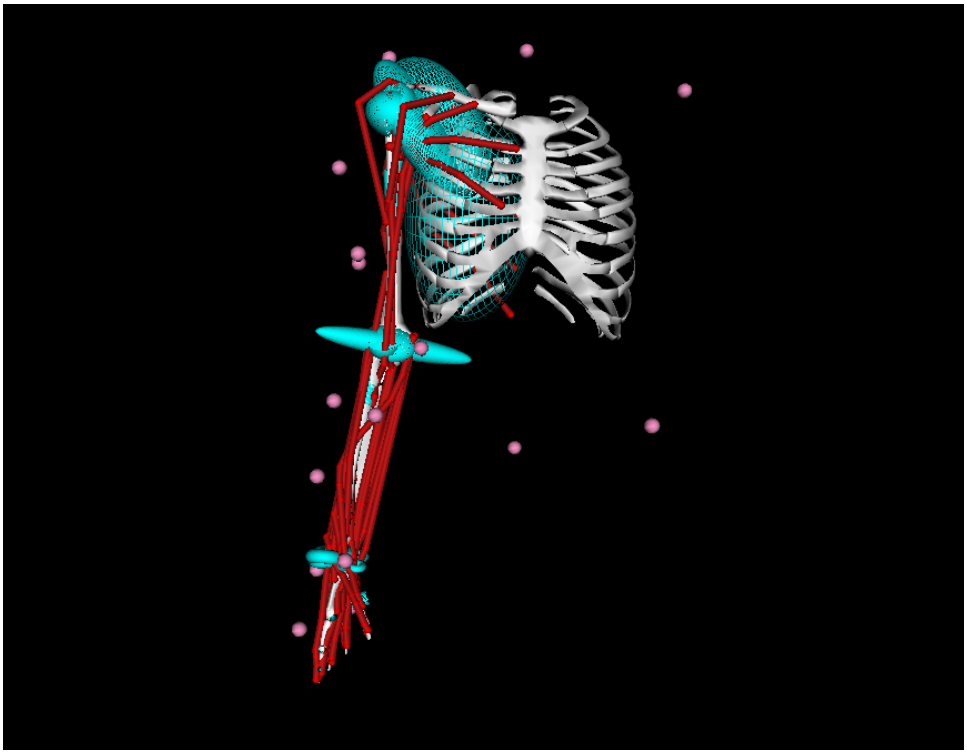


Figure 4.1: Upper Extremity Model in OpenSim Scaled by C3D Data

The model is actually a SIMM model, and SIMM was used to add the marker set used in the mocap system to the model. Once the marker set was added, the model was imported into OpenSim and a few manual changes were made to allow the model to work with the

mocap data. Most alterations involved increasing the range of a joint or renaming a gencoord to reduce ambiguity. The most important change was adding 6 DOF to the joint that is defined between the thorax and the ground (world frame). It is obviously necessary to define a segment on the model wrt the world frame, but this joint was locked in place so that the model could not have any movement at all wrt to the world frame. All changes made to the model are outlined below:

1. Added markers (using SIMM)
2. Changed thorax_ground joint to have 6 DOF
   - 6 gencoords (thorax_tx, thorax_ty, thorax_tz, thorax_roll, thorax_yaw, thorax_pitch)
3. Changed "flexion" gencoord to "wrist_flexion"
4. Changed "deviation" gencoord to "wrist_deviation"
5. Changed "shoulder_elv" gencoord to "shoulder_flexion"
6. Changed "elv_angle" gencoord to "shoulder_abduction"
7. Increased shoulder_rot max from 20 to 90 degrees
8. Decreased shoulder_flexion min to -20 degrees
9. Decreased 2pm_flexion min to -1 degrees
10. Decreased 2md_flexion min to -1 degrees
11. Increased elbow_flexion range from 0 through 130 to -1 through 160 degrees
12. Increased wrist_deviation range from -10 through 25 to -45 through 45 degrees
13. Increased wrist_flexion range from -70 through 70 to -90 through 90 degrees

Once the upper extremity model was established it was scaled using weights of 1000 for markers attached to rigid bodies and 1 for other markers. This essentially ensures that the segments are scaled properly between markers attached at their endpoints. The other markers basically become “tiebreakers” in determining size and position. Mocap C3D data was then loaded and used to calculate joint angles using OpenSim's IK tool. For the IK tool weights of 10 were given to rigid body markers and 1 to other markers. A total of 5 different mocap experiments were loaded, and the joint angles calculated for one experiment are shown in Figure 4.2. In Appendix A, a video of one of the experiments is provided.

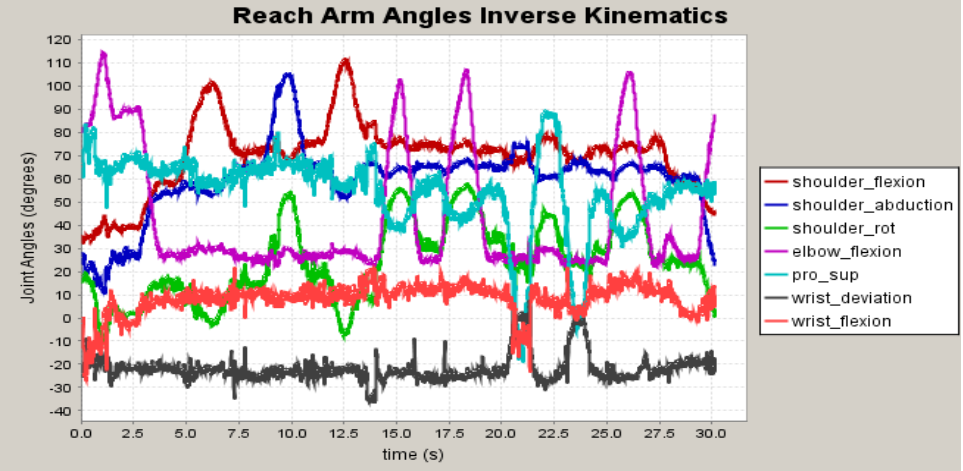


Figure 4.2: OpenSim IK Results for C3D Reaching Data Experiment

4.2.2 MotionSoft and Visual3D Comparison

In order to determine the validity of the IK data calculated by OpenSim, some of the joint angles were compared to values found by MotionSoft and Visual3D. This was a difficult comparison to make because the neutral position (all joint angles at zero) was defined differently by OpenSim (Figure 4.1) than it was by the other software. In addition, some of the axes used to calculate joint angles were reversed. To help fix this problem, a stationary pose was used to determine the differences between the systems' neutral poses. The stationary pose was mocap data of the arm being held still, and the validity of OpenSim's IK calculations for this pose can be verified visually by the snapshot in Figure 4.3.

OpenSim's joint angles from the stationary pose were adjusted to match Visual3D's joint angles. Visual3D was used instead of MotionSoft because MotionSoft did not include shoulder angles, and MotionSoft and Visual3D were expected to have the same axes and neutral position. Adjustments made to OpenSim's angles are outlined in Table 4.1.

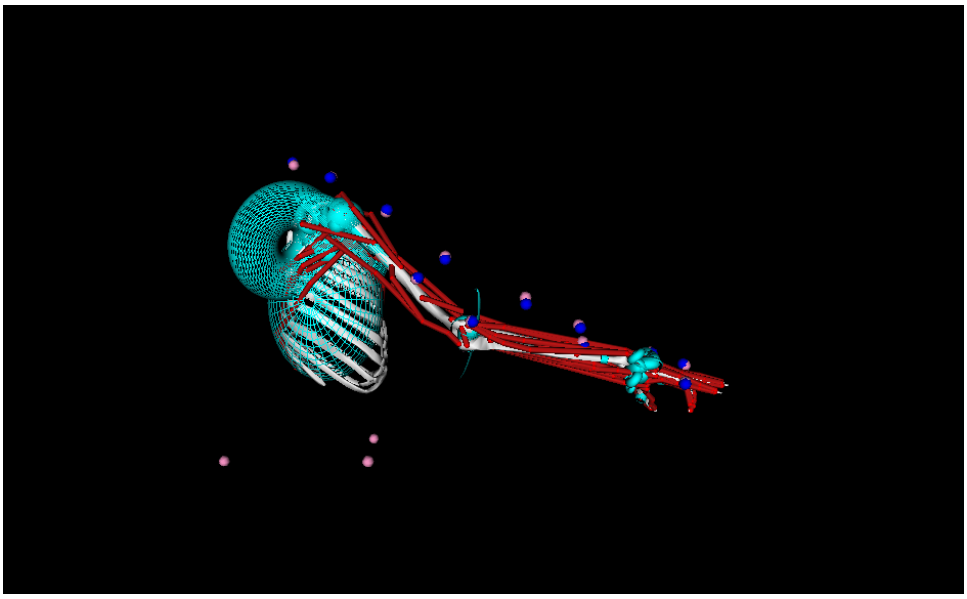


Figure 4.3: Reaching Experiments Stationary Pose in OpenSim

Table 4.1: Adjustments Made to OpenSim's Reaching Joint Angles

	Shoulder Flexion	Shoulder Abduction	Shoulder Rotation	Elbow Flexion	Elbow Pronation	Wrist Deviation	Wrist Flexion
Degrees Shifted	-120	-45	-15	15	75	35	-20
Flipped Axis?	no	no	yes	no	yes	no	no

Once these adjustments were made, joint angles for the different reaching experiments were compared. As shown by Figure 4.4, the results were extremely good for the elbow flexion DOF. The average difference between the OpenSim and Visual3D results is only a few degrees, and MotionSoft's results are only slightly offset. The curves have nearly identical shapes for all 3 systems, which is what is most important.

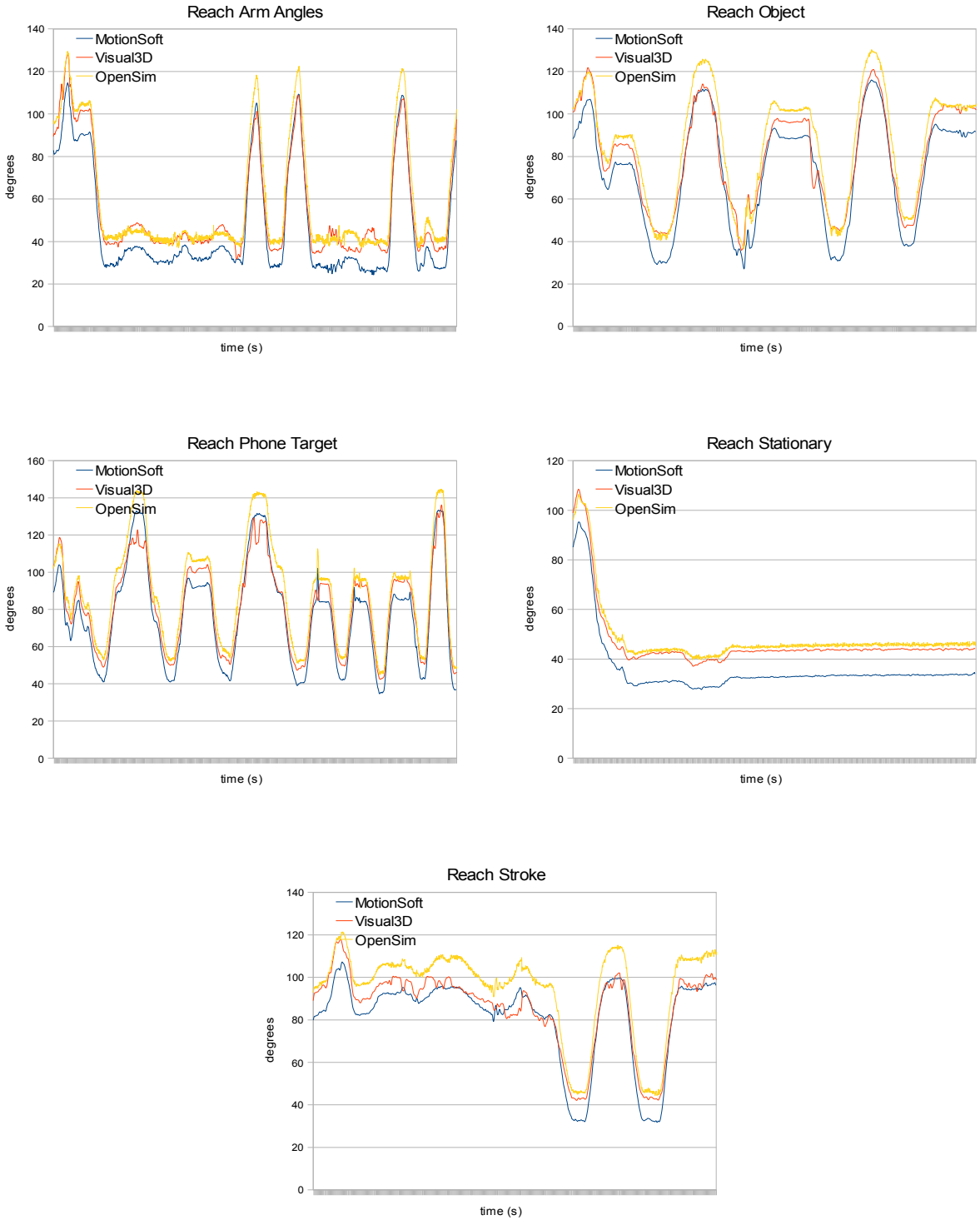


Figure 4.4: Elbow Flexion Comparisons for Reaching Experiments

For the other joint angles (Figures 4.5-4.9), the results were not as good. Shoulder flexion and the wrist DOF matched fairly well between OpenSim and Visual3D, but there were some major differences. Some difference is always expected due to slightly different marker position definitions, axes definitions, etc., but the results should be better than those displayed in many of the charts.

The biggest problem is that in many of the curves the OpenSim joint axis appears to flip halfway through the motion wrt the Visual3D or MotionSoft axis. This is a particular problem with shoulder abduction and shoulder rotation. Without observing the mocap experiment it is hard to determine if this is a problem with OpenSim, Visual3D, and/or MotionSoft. In many cases of the wrist DOF, the MotionSoft axis is reversed from the Visual3D axis, which was totally unexpected.

The most promising aspect of the data is that in general, the same trends are followed by all systems for nearly every joint angle. Sometimes the axes might be reversed or the peaks might be more pronounced in one system, but the same basic shape is still there. For future comparisons more care must be taken to ensure that all systems have the same neutral position and the same joint axes are defined. OpenSim's accuracy can also be observed by comparing the actual motion recorded by the mocap system to the motion reproduced in OpenSim by its IK tool.

Figure 4.5: Joint Angle Comparisons for Reach Arm Angles Experiment

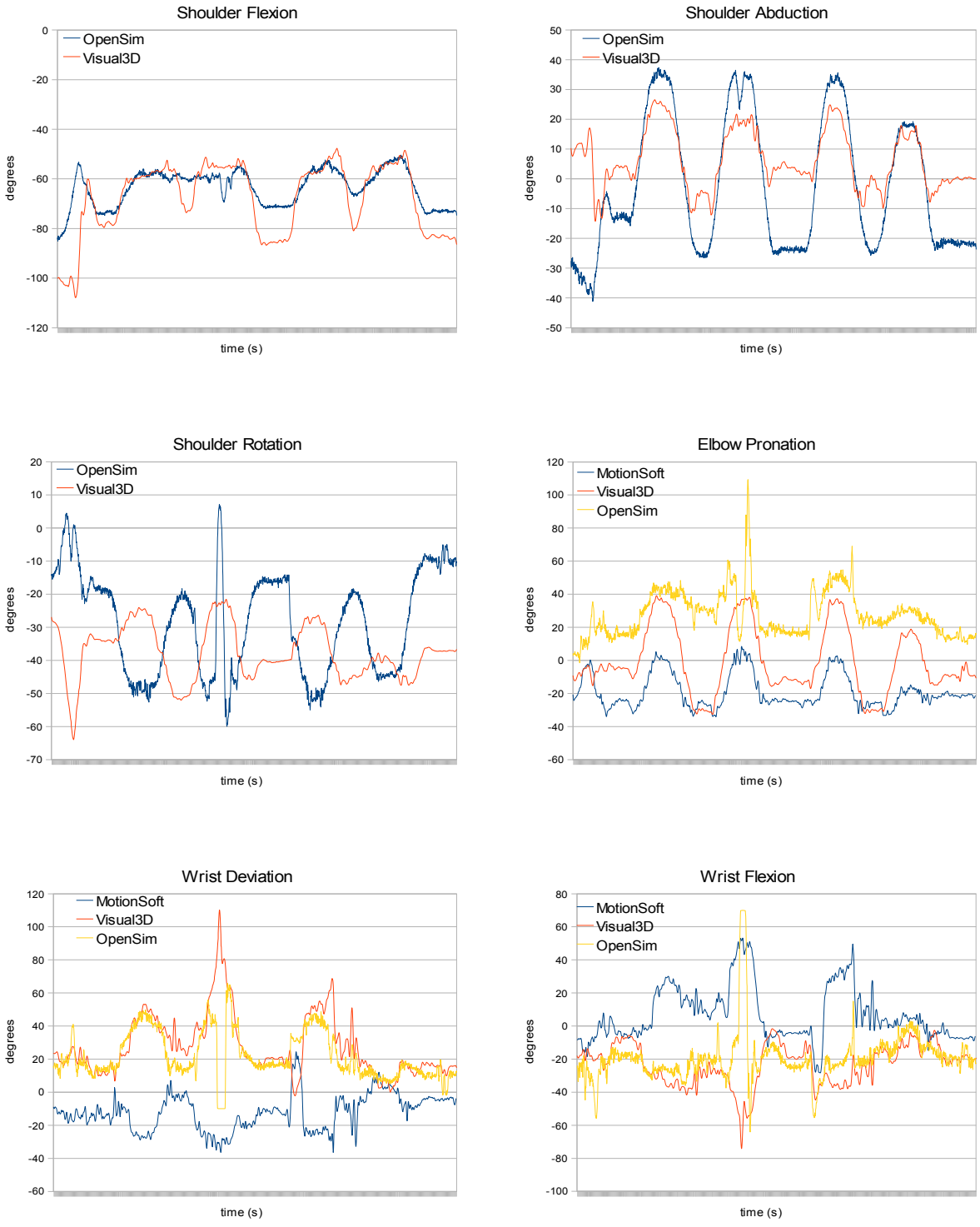


Figure 4.6: Joint Angle Comparisons for Reach Object Experiment

Figure 4.7: Joint Angle Comparisons for Reach Phone Target Experiment

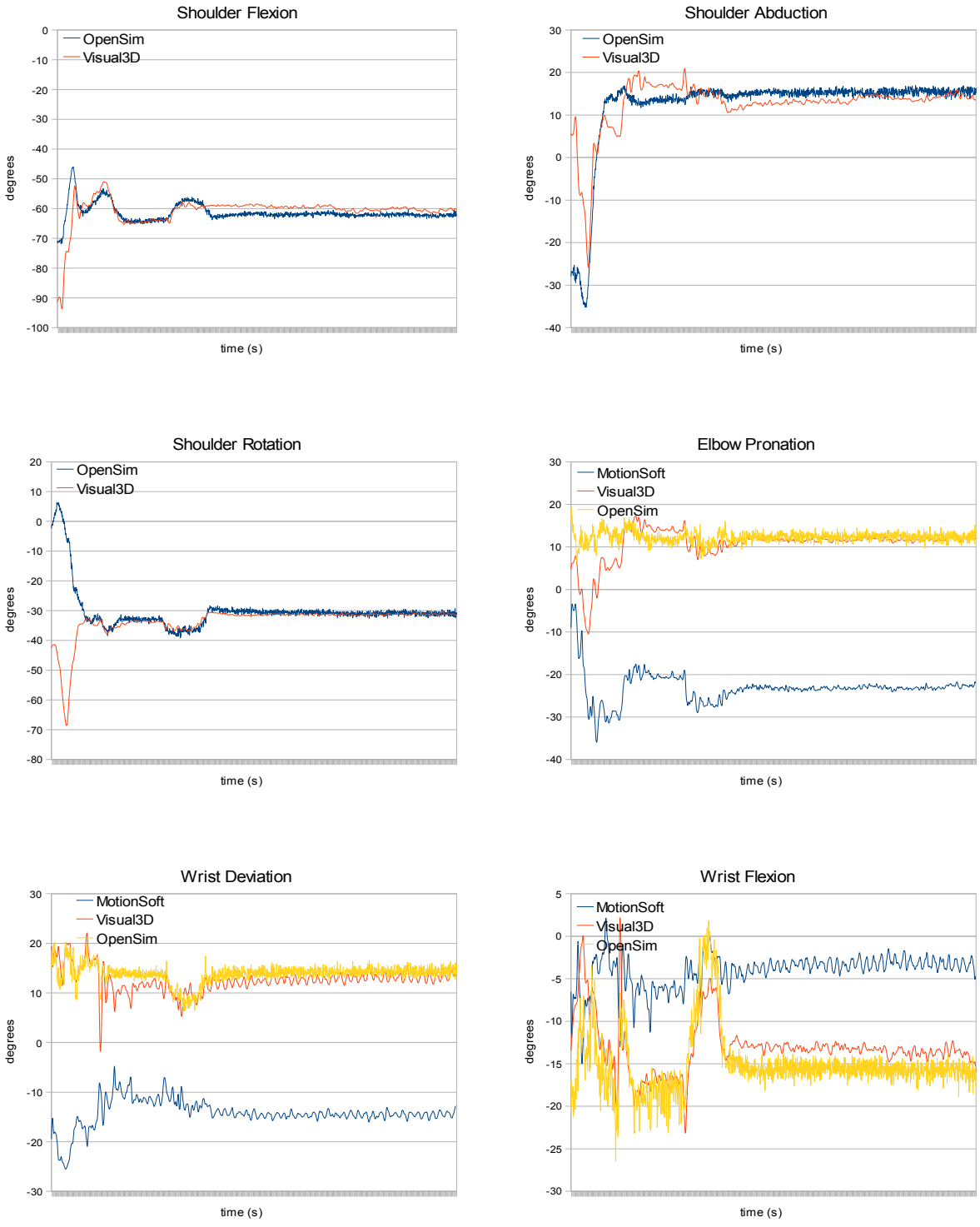


Figure 4.8: Joint Angle Comparisons for Reach Stationary Experiment

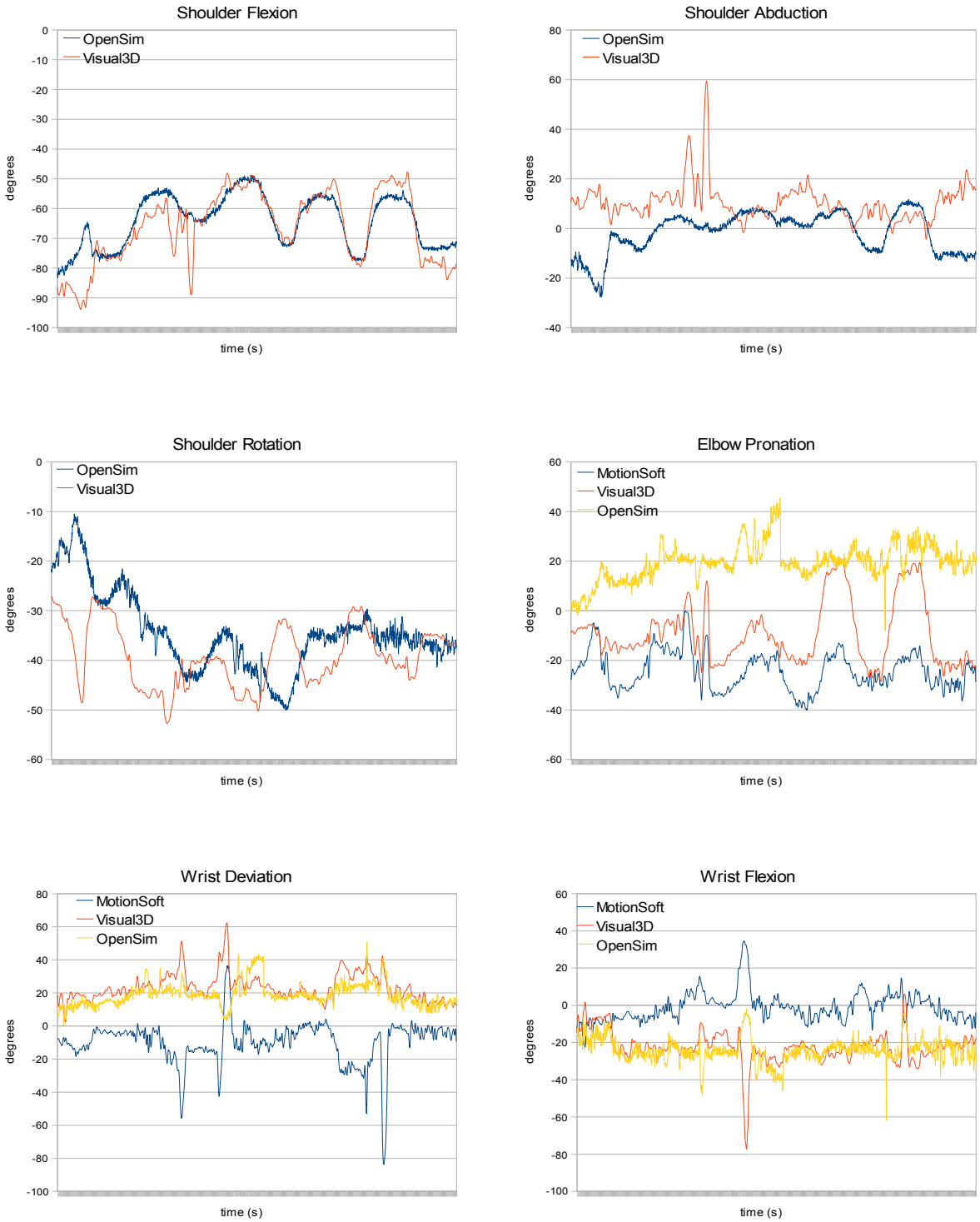


Figure 4.9: Joint Angle Comparisons for Reach Stroke Experiment

4.3 Jumping Data

4.3.1 OpenSim Results

For the jumping data, 3DGait2392, a model included with the standard OpenSim installation, was used. 3DGait2392 is a model of the lower body plus torso, back, and head with 23 DOF and 92 muscles. It is a modified version of the lower extremity model created by Delp et al [21]. No adjustments were required for this model, and the model scaled by test subject DJM27S1's static pose is shown in Figure 4.10. After scaling the model, the mocap data obtained from C3D files was loaded into OpenSim's IK tool. As with the reaching experiments, markers attached to rigid bodies were given a weight of 1000 during scaling and 10 during IK calculations. A total of 3 different jumps were recorded, and the IK results for one of these jumps is shown in Figure 4.11. In Appendix A, a video of one of the experiments is provided.

4.3.2 MotionSoft Comparison

For the jumping experiments Visual3D data was not available, but MotionSoft was still used to compare some of the calculated joint angles. The comparison charts for knee flexion in each trial are in Figure 4.12. As shown by Table 4.2, OpenSim was reasonably consistent with MotionSoft in determining the right and left knee flexion/extension DOF. On average, the data only differed by a few degrees.

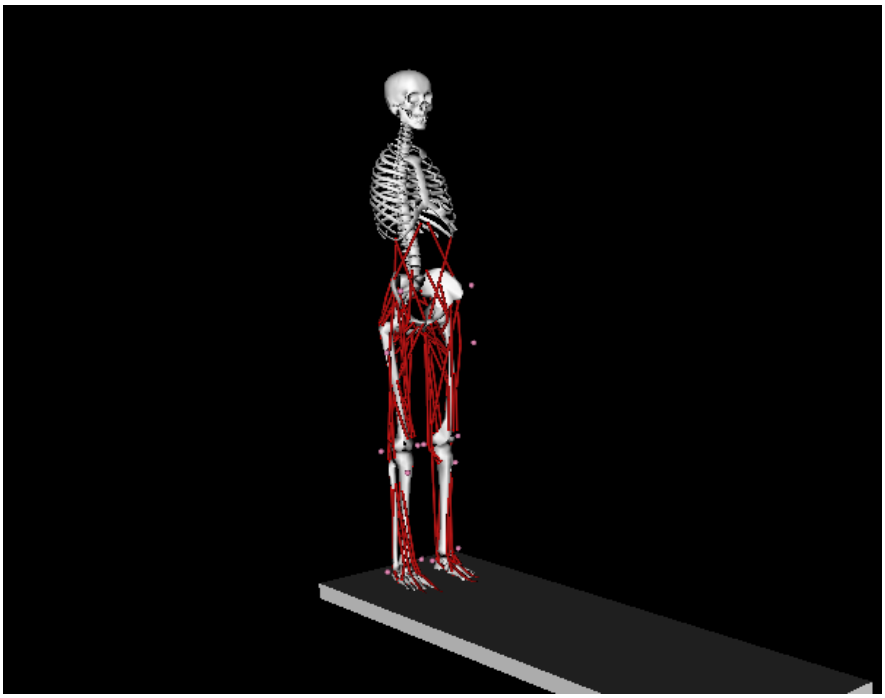


Figure 4.10: 3DGaitModel in OpenSim Scaled by c3d Data

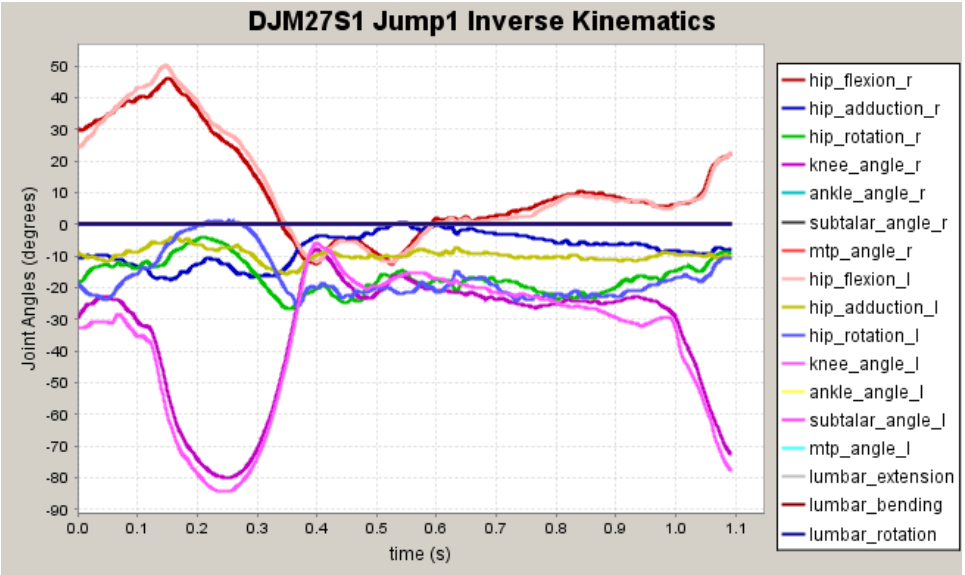


Figure 4.11: OpenSim IK Results for c3d Jumping Experiment

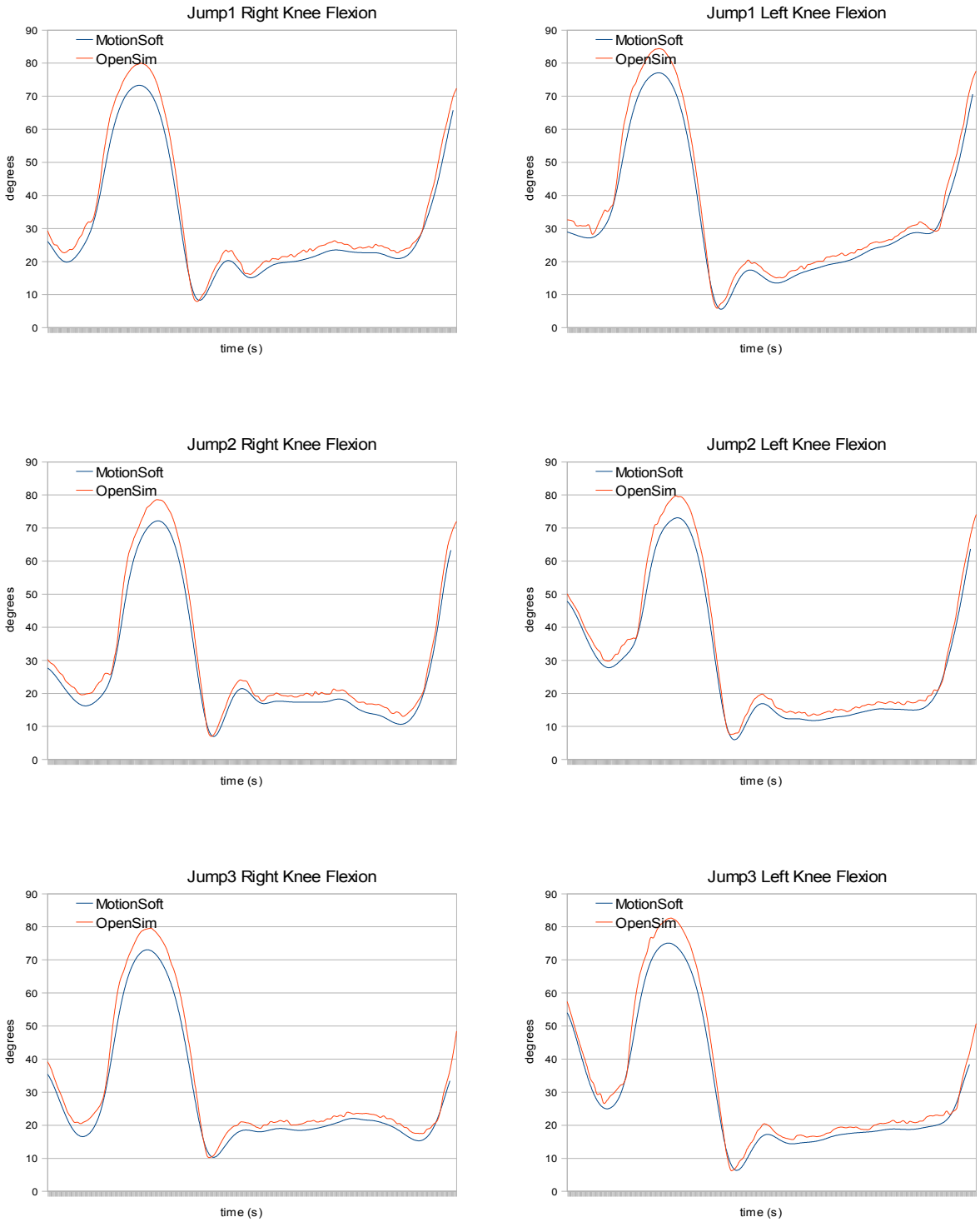


Figure 4.12: Knee Angle Comparisons for Jumping Experiments

Table 4.2: Difference (in degrees) Between OpenSim and MotionSoft Knee Flexions

	Jump1		Jump2		Jump3	
	Mean	Max	Mean	Max	Mean	Max
Right Knee Flexion	3.15	7.86	3.33	8.51	3.13	9.17
Left Knee Flexion	3.26	8.53	2.95	8.06	3.07	8.86

4.3.3 OpenSim ID Results

The C3D data that was loaded for the jumping experiments contained analog forceplate data in addition to the marker coordinates. After the forceplate data was preprocessed to calculate GRFs, the data was loaded into OpenSim. Figure 4.13 shows the GRFs, but it should be noted that these forces were found by using Stanford University's functions in MATLAB, not by OpenSim. In Figure 4.14, the GRFs are shown as force vectors at the point of contact between the OpenSim model and the ground.

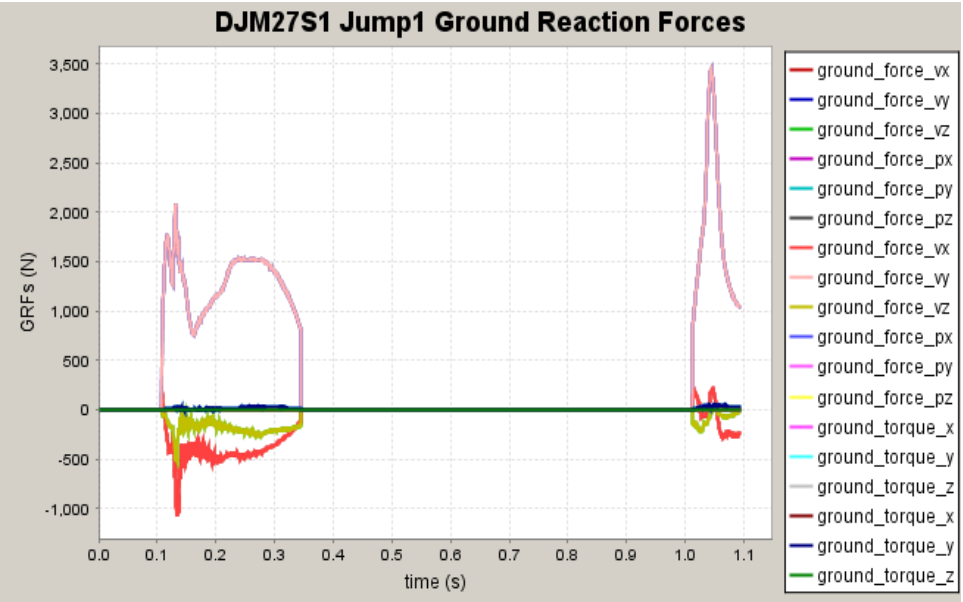


Figure 4.13: GRFs for Jumping Data Plotted in OpenSim

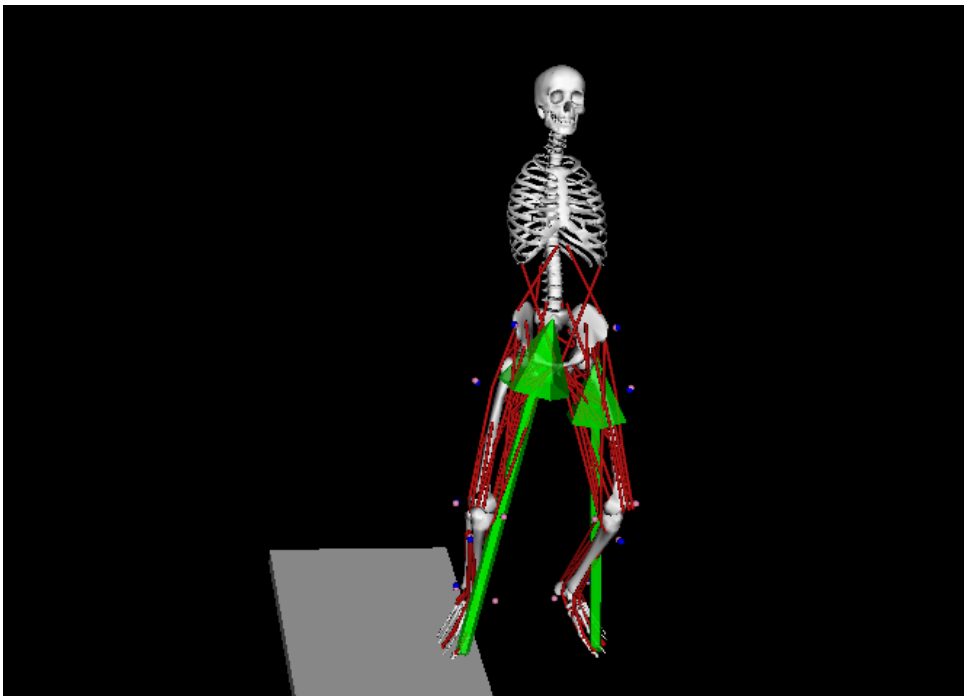


Figure 4.14: GRFs as Force Vectors in OpenSim

When external forces such as GRFs are loaded into OpenSim they can also be used in the ID tool, assuming the model is capable of dynamics (which 3DGait2392 is). The forces must be applied to one of the segments of the model, thereafter OpenSim calculates the forces and torques required for the motion. Figure 4.15 shows the ID calculations from one of the jumping experiments. This is given merely to demonstrate successful ID calculation with mocap data from a C3D file. For truly accurate dynamics calculations, masses and inertias for each segment of the model must also be known [5]. These values were not collected in this experiment.

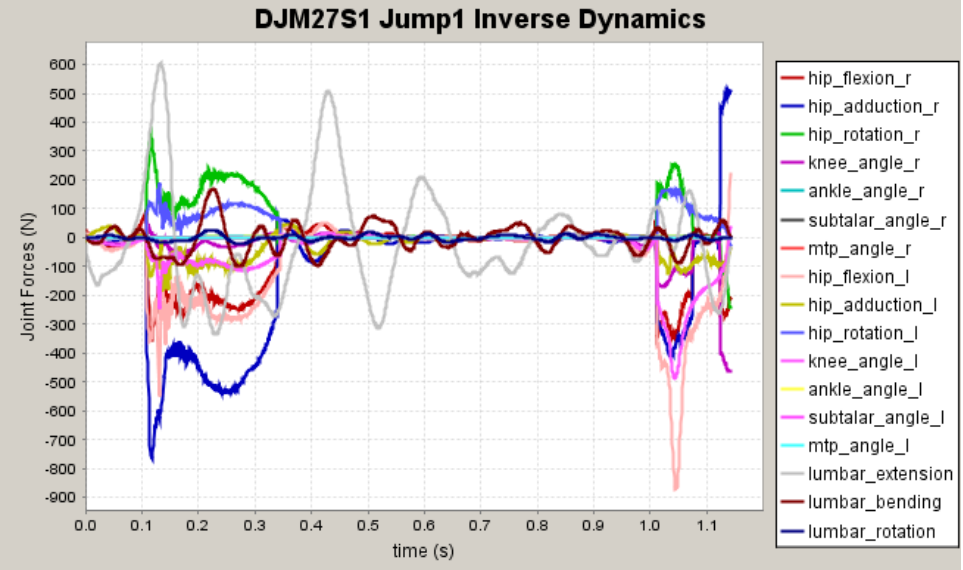


Figure 4.15: OpenSim ID Results for c3d Jumping Experiment

4.4 Conclusions

OpenSim's IK tool performed extremely well for simple joint angles that don't really interact with other DOF, i.e., elbow flexion, knee flexion. For these angles the calculations from OpenSim closely match the calculations of Visual3D and MotionSoft. For more complex joints, such as the shoulder, there were inconsistencies between OpenSim and other analysis software. These inconsistencies are probably mostly due to (1) the definitions of the joint axes and (2) how the segments are tracked by each software package [Steve Leigh, personal communication, July 2008]. Other factors that might contribute to inconsistencies include: (1) differences in the definition of the neutral position, (2) differences in the the definition of the DOF used, or (3) errors in one of the software packages. To determine the cause of the differences and to determine which system gives the most accurate IK

calculations, further experiments and closer observations must be made. For each IK calculation done by OpenSim, the resulting simulated motion appeared to be correct. Yet again, it is impossible to know for sure without observing the true motion recorded by the mocap system.

Chapter 5

Processing of a Large Motion Data Set

5.1 Data Set Description

The purpose here was to demonstrate how motion parameters like those calculated by OpenSim may be efficiently analyzed and reduced to their important features. Because the available OpenSim data was sparse a different data set was used. This data set was collected jointly by Duke University and UNC as part of a project funded by a Whittaker grant to Dr. Martin McKeown at the University of British Columbia, with Dr. Carol Giuliani at UNC and Rachael Brady at Duke as collaborators. For data collection, a VR environment was created in which subjects sat in a chair and faced a screen from which virtual balls were thrown towards them. There were 7 unique testing conditions (see Table 5.2). Some balls were marked as target balls immediately, others were not targets, and some were identified as targets after a delay. Each subject was instructed to hit each target ball with a virtual paddle affixed to their hand [S. Leigh, personal communication (see Appendix D), March 2008]. In all, 12 healthy people and 11 stroke patients were tested, for a total of 250 unique trials and over 5,000 target balls. Further details on the data available are outlined below in Table 5.1.

Table 5.1: Data Available from Stroke Reaching VR Experiments

Subject	Conditions Tested (L=left arm, R=right, B=both)							Dominant Arm	Stroke-affected Arm
	1	2	3	4	5	6	7		
e1	B		B	B*	B*	B	B*	R	L
e2	B		B	L,R*	L,R*	B	L,R*	R	L
e3	B	B	B	B*	B*	B	B*	R	R
e4	B	L	B	B*	B*	B	B*	L	R
e5	B	R	B	B*	B*	B	B*	R	L
e6	R**		R	R*	R*	R	R*	R	L
e7	B	L	L,R**	B*	B*	L	L*	R	R
e8				*	*		*	R	R
e9	B	L	B	B*	B*	B	B*	R	R
e10	B		B	B*	B*	B	B*	R	L
e11	B	R	B	B*	B*	B	B*	R	L
h1	B	R	R	B*	B*	R	B*	R	
h2	B	R	R	B*	B*	R	B*	R	
h3	B	R	R	B*	B*		B*	R	
h4	L	R	R	B*	B*	R	B*	R	
h5	B	L	L	B*	B*	L	B*	L	
h6	B	R		B*	B*	R	B*	R	
h7	B	R	R	B*	B*	R	B*	R	
h8	B	R	R	B*	B*	R	B*	R	
h9	B	R	R	B*	B*	R	B*	R	
h10	B	R	R	B*	B*	R	B*	R	
h11	B	R	R	B	B	R	B	R	
h12	B	R	R	B*	B*	R	B*	R	

**FastTrack Data available, **2 trials completed for this condition*

Table 5.2: Description of Experiment Conditions

Condition Number	Target Balls	Total Balls	Ball Release and Target Identification Methods
1	varies	varies	individual release, varying ball speeds and heights
2	varies	varies	individual release, varying ball speed
3	varies	varies	individual release, varying ball speed, subjects swatted balls
4	20	20	individual release, immediate identification
5	20	50	individual release, targets randomly chosen, identified immediately
6	varies	varies	5 balls released at once, 1 of the 5 a target, identified after a delay, varying ball speed
7	20	100	5 balls released at once, 1 of the 5 a target, identified after a delay

During all trials, VR markers were placed on the subject. The VR program was controlled by Track D, which streamed the marker data and information about the VR balls to Syzergy, a program created by engineers at Duke. The Syzergy program then wrote the data. For testing conditions 4, 5, and 7, additional information was collected using Polhemus FastTrack hardware and Motion Monitor (Innovative Sports Training, Inc.) software. The Syzergy program gives as its output position and orientations coordinates for the subject's trunk, head, and hand. It also gives as an output coordinates of the virtual balls, along with information about when the ball was released, when it was revealed to be a target ball (if it was a target ball), and when, if ever, it was contacted by the subject's virtual paddle. The FastTrack hardware tracked coordinates for the trunk and joint angles for the shoulder and the elbow. It is important to note, though, that the FastTrack data was only collected for conditions 4, 5, and 7 (Table 5.1). These conditions are also the most consistent of the trials,

as the other 4 conditions were mostly used to determine the correct setup to use for each subject on conditions 4, 5, and 7 [S. Leigh, personal communication, March 2008].

5.2 Preprocessing the Data

5.2.1 Formatting the Syzergy Data

Originally, the data collected from the Syzergy system was in a format that was not suitable for statistical analysis or feature extraction. Due to the sheer amount of data collected, over 1.2 gigabytes of ASCII data, manual analysis was not practical. The data was streamed in so that each line of the data contained a time stamp along with the data collected by 1 sensor. The time stamps in this original format are simply labels; and these cannot be converted to actual or relative time. Their only function is to synchronize the data between the various sensors.

To make the Syzergy data useful, a program was created to read in and rearrange the data so that each row contained one sample from all of the sensors. A time stamp was then added to each row that marked the time relative to the beginning of data collection. The resulting data was then output in a csv file for use in spreadsheet applications and other software. It could also be stored as a table in a .mat file for further use in MATLAB. The second storage option also saves space, as it requires around 0.25 gigabytes compared to 4.4 gigabytes for csv files.

5.2.2 Synchronizing the FastTrack Data

In order to use the FastTrack joint angles, the trunk coordinates collected by the Syzergy system were matched with the FastTrack trunk coordinates. This allowed the two sets of data to be synchronized. To complete this process, the FastTrack data was first upsampled from its sampling frequency (30 Hz) to match the 50 Hz sampling frequency of the Syzergy system. Second, both sets of coordinates were temporarily set to a zero mean. This was because the coordinate sets appeared to have different frames of reference. Subtracting out the means does not necessarily give them the same reference frame, but it did help in matching the curves.

The FastTrack trunk coordinates also had random, sometimes severe, spikes of noise. These were due to either electromagnetic interference around the trunk marker or by the marker moving outside the electromagnetic field created by the system's transmitter [S. Leigh, personal communication, April 2008]. To some extent, this noise was removed, but much of it was extremely difficult to eliminate. Since the Syzergy trunk coordinates could be used for analysis purposes, the FastTrack trunk coordinates were only used to synchronize the data, so it was not necessary to fully eliminate the noise. To match the curves, a window of one set of data was moved across the other to find the point at which the MMSE between the curves appeared [22]. Once that point was established, the joint angles were added to the data at the appropriate time.

In most cases the data synchronized successfully despite noise or problems with one or more of the coordinates (Figure 5.1). The biggest problem was that sometimes it appeared

that the two data sets were sampled at different frequencies despite consistent resampling of the data in MATLAB. This caused one of the trunk coordinate sets to appear to be stretched when compared with the second set (Figure 5.2). The data remained synchronized and the results are usable, but still, slight errors must remain. There are just a few instances where all of the coordinate channels had problems and the data could not be synchronized. For example, Figure 5.3 shows a trial where the FastTrack system appears to just be connected to ground, and Figure 5.4 shows a trial where the Syzergy data appears to be random noise. For some reason noise appeared more often in y-coordinates of the Syzergy data, but the data was still good enough for synchronization. Table 5.3 shows trials where there might have been synchronization problems.

Table 5.3: Trials with Possible Synchronization Problems

Subject	Arm	Trials with Apparent Inconsistent Sampling	Trials with Bad Data
e5	Left	7	
e6	Right	4, 7	
e9	Right	4,7	
e9	Left	4,5	
e11	Both	4, 5, 7	
h1	Both	4,5,7	
h2	Both		4,5,7
h5	Both	4,5,7	
h6	Right	4,5,7	
h8	Both	4,5,7	
h11	Right	4,5,7	
h12	Both	4,5,7	

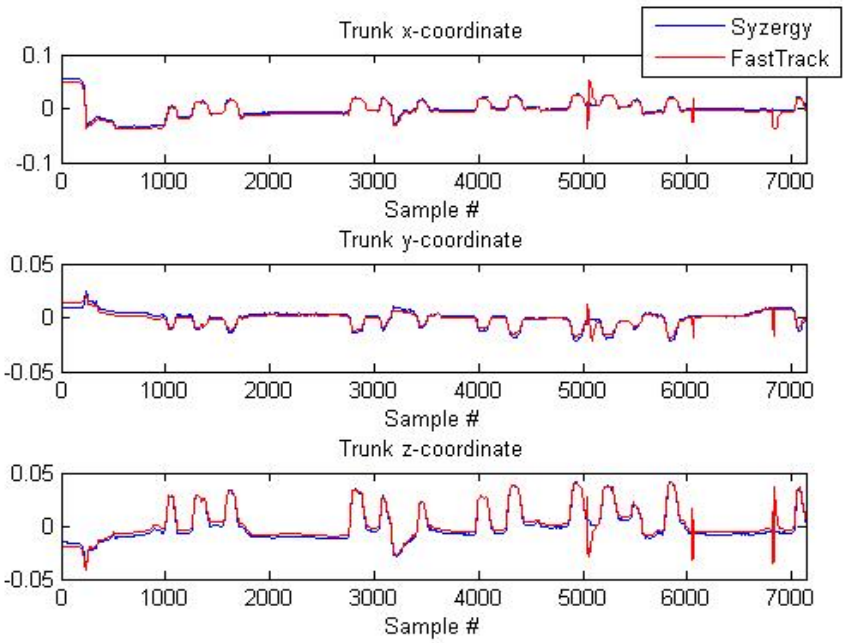


Figure 5.1: Successful Synchronization (Trial e1l2c5)

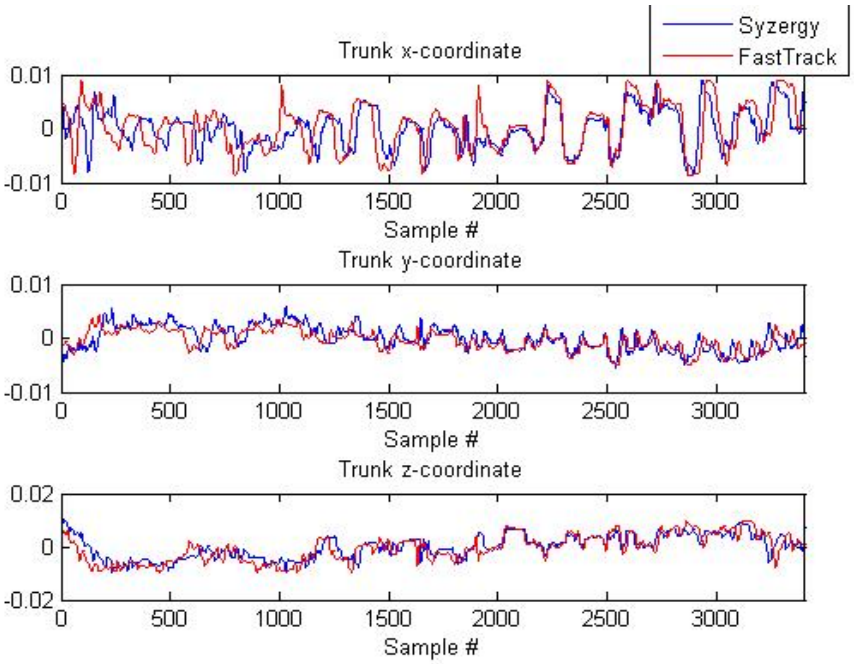


Figure 5.2: Synchronization Off, Inconsistent Sampling (Trial h1l1c7)

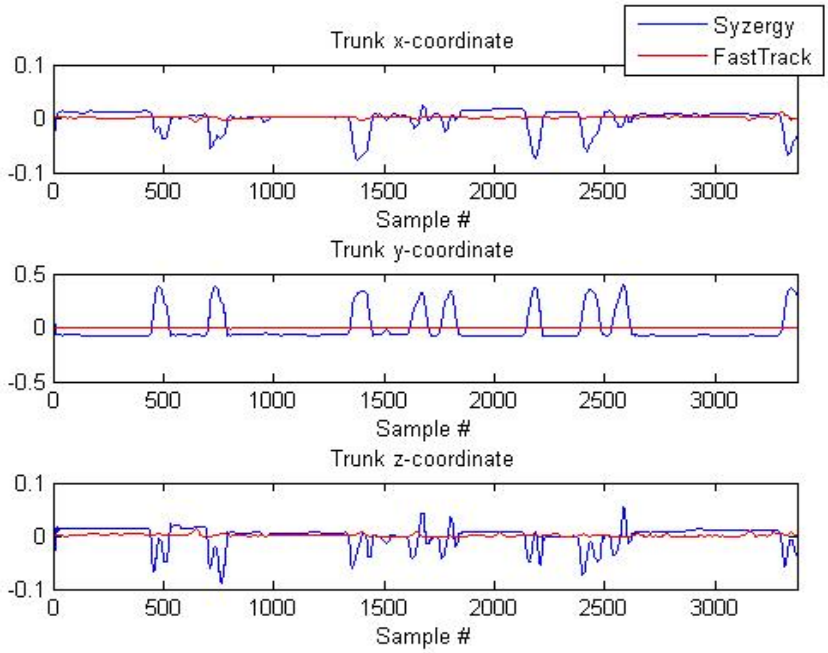


Figure 5.3: Synchronization Failed, FastTrack System Off (Trial h2l1c5)

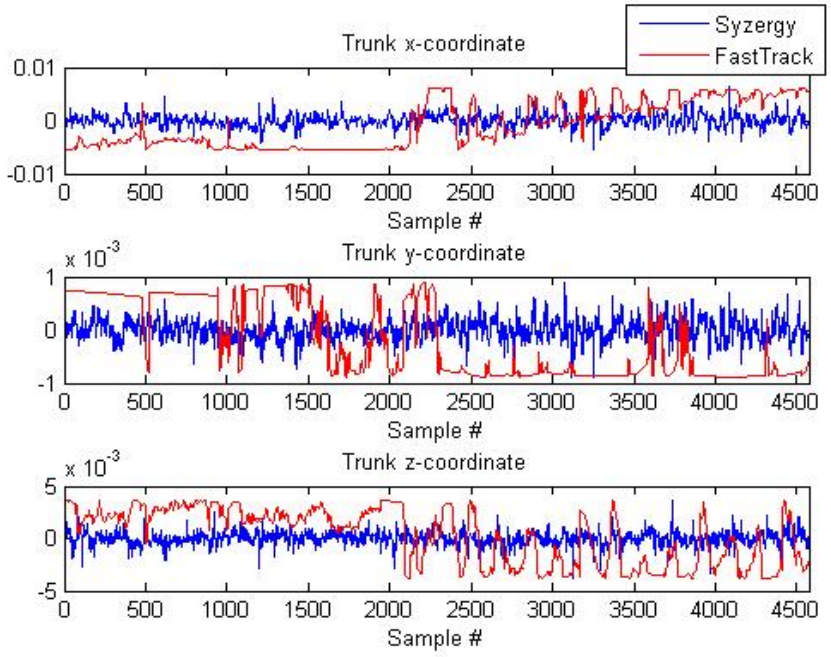


Figure 5.4: Synchronization Failed, Syzergy Data Noisy (Trial h2l2c7)

5.3 Statistics and Data Analysis

5.3.1 Determining Movement Time

To make use of such a large set of data, it was necessary to extract important statistics and features. The most important data collected was related to the subject's hand when it was actually moving towards a target ball in an attempt to make contact. So the first step in analyzing the data was to determine the windows in the data when this hand motion was occurring. This is a much more difficult process than it seems to be, especially when the subject is a stroke victim. The movement could be choppy, and in many cases there might be a false movement that occurred before the true movement towards the target ball. There was also a chance that the hand was still in motion from the previous target ball. To determine the movement time as best as possible, the following steps were taken:

- The end of movement was defined as the time at which contact with the ball is made, or if no contact is made then it was defined as the time at which the ball disappears from view.
- A point X was then defined as either 0.3 seconds prior to the time of contact, or if no contact is made then it was defined as 0.3 seconds prior to the time of peak hand velocity during the second half of the ball's flight. The assumptions made were as follows: (1) true hand movement occurs during the second half of the ball's flight, and (2) peak velocity should occur near the point when subjects attempted to make contact with the ball. The start of movement must occur before this point.

- The start of movement was defined as the first point in time before point X when the hand velocity in the longitudinal direction (towards the ball) was greater than 0 and the hand displacement, i.e., straight line wrt the base frame of reference, in the longitudinal direction was more than 5% more than the minimum displacement that occurred during the ball's flight. This eliminated many false movements while still retaining true movements that were just choppy.

The method for determining movement time can be verified graphically by plotting the hand movement and marking the calculated movement start and end points. Since there were over 5000 target balls, it was unreasonable to graphically test every single target ball. Out of over 50 randomly selected target balls, except for 4 target balls, the movement time was correctly determined. In each of those 4 negative instances, the hand did not really move enough for there to even be considered a movement time. So, when hand movement occurred, the movement time was calculated very reliably. Figure 5.5 shows the hand kinematics when the movement time was correctly calculated. Figure 5.6 shows the trunk and head positions and the joint angles for the same target ball, and Figure 5.7 shows the orientations of the hand, head, and trunk. Figure 5.8 shows a movement time calculation that was incorrect due to a lack of hand movement.

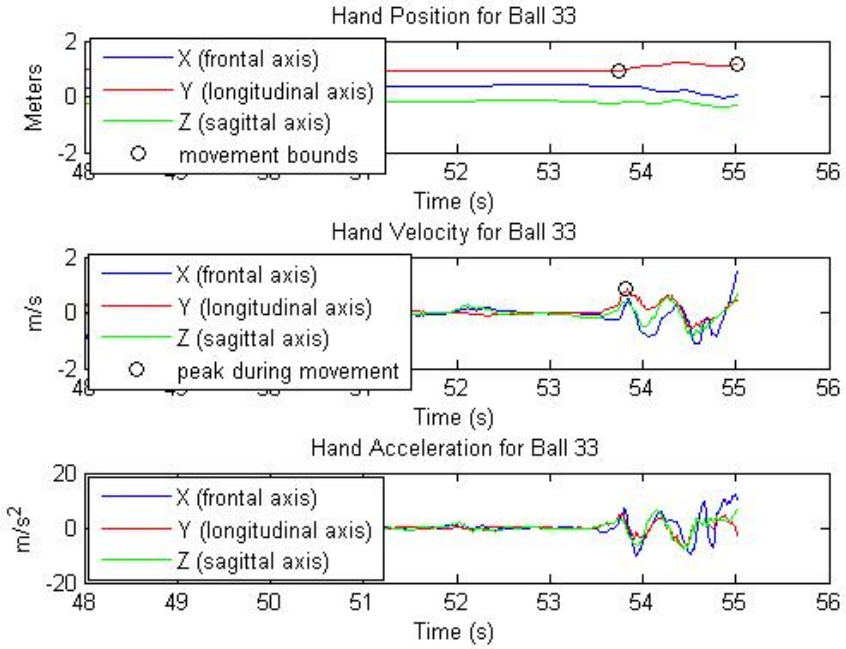


Figure 5.5: Hand Kinematics for One Ball (Trial e10l1c7)

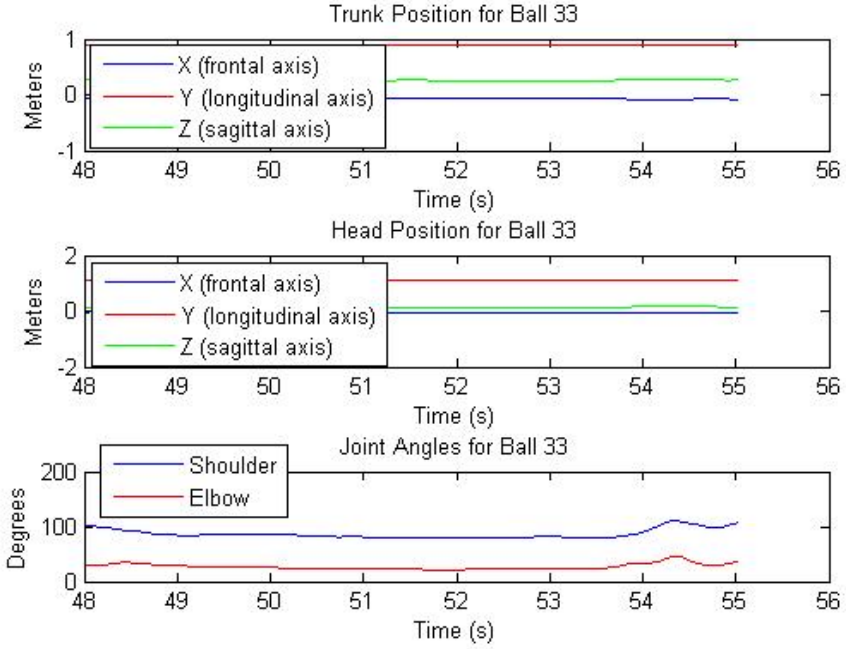


Figure 5.6: Trunk, Head, and Joint Positions for One Ball (Trial e10l1c7)

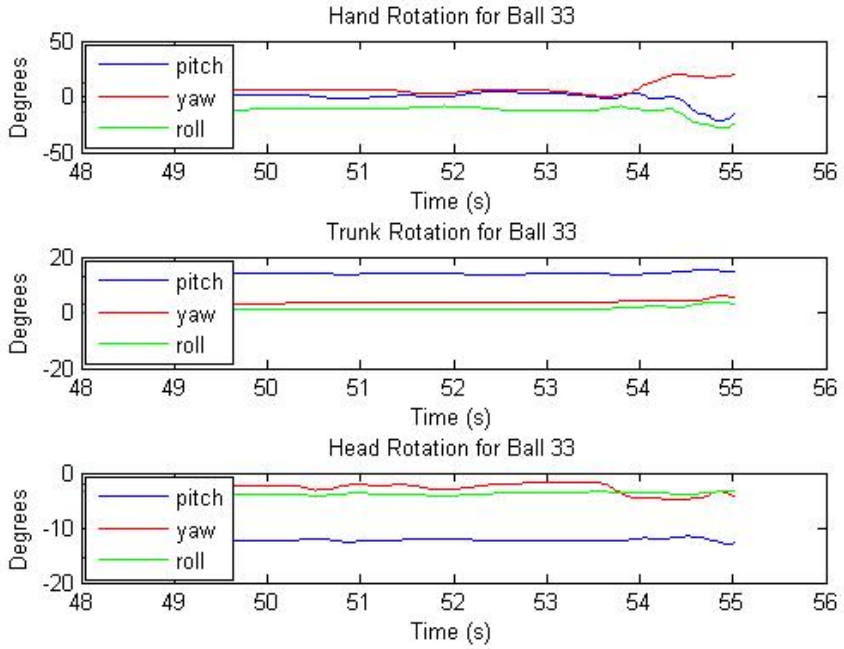


Figure 5.7: Hand, Trunk, and Head Rotations for One Ball (Trial e10l1c7)

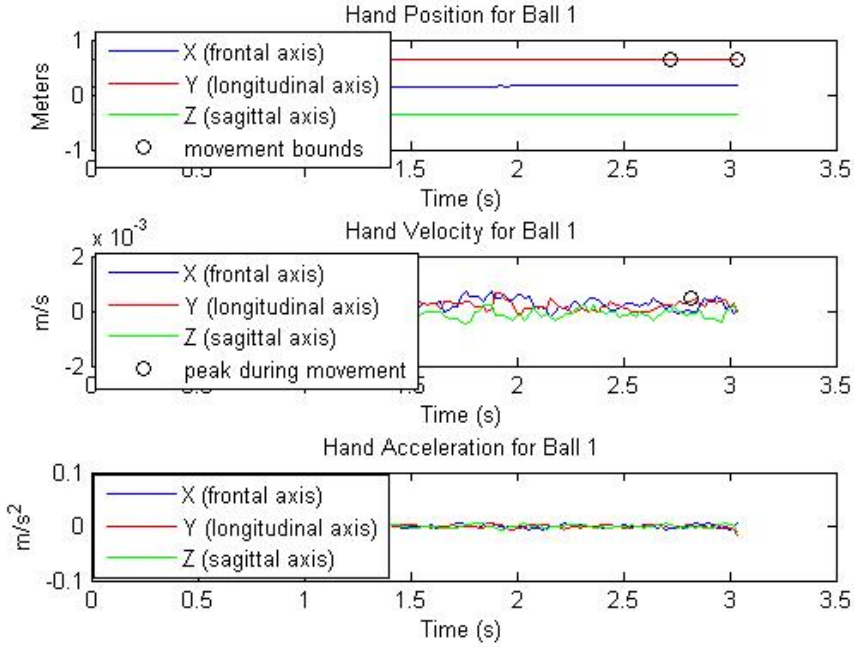


Figure 5.8: Hand Kinematics, Wrong Movement for One Ball (e1l1c7)

Movement time would have been more accurate if the subject had been asked to remain in a neutral "at rest" position for a few seconds at the beginning of each trial. Using this approach displacement would have been defined wrt to this equilibrium position rather than the base of the reference frame. Different methods were applied in an attempt to find the neutral position of the subjects under test. However, there were just too many variables present between subjects to define a neutral position that was reliable enough to improve the accuracy of finding the movement time. The neutral position must be defined correctly more often than the movement time if it is ever to become a useful metric.

5.3.2 Feature Extraction

Once movement time was determined, it was possible to examine movement that occurred during this time period to extract interesting statistics and features. The initial set of features to examine was suggested by Dr. Carol Giuliani of UNC. These she considered to be statistics of interest when looking at the movement of stroke patients [Carol Giuliani and S. Leigh, personal communication (see Appendix D), March 2008]. A few additional features were added after the first set was examined. Table 5.4 shows the means of various features tested for under different testing conditions, for both stroke patients and healthy subjects. Each sample was considered the movement occurring during the flight of a target ball.

Most of the features are fairly self explanatory with the exception of hand velocity smoothness. Many different methods were used to achieve a smooth velocity profile. These included finding the average hand jerk, i.e., a derivative of acceleration, looking at the

number of velocity peaks, summing the magnitude of the high frequency components present in the FFT of the velocity curve, and counting the number of samples during movement at which the longitudinal acceleration is not positive. The last method worked well. The basis for its creation was the fact that once the hand begins moving towards the ball it should continue accelerating towards the ball until contact. Any deceleration or hesitation is "penalized." So, lower numbers in the result correspond to smoother velocities.

Table 5.4 shows the statistics for all 7 testing conditions averaged together, and then testing conditions 4, 5, and 7 individually. This is because, as stated earlier, these conditions contain the FastTrack data and they are also the most consistent. No matter what the testing conditions are, though, the features that are examined follow the same trends when comparing the healthy subjects to the stroke patients. The differences seem to be slightly greater for testing condition 4, though.

Table 5.4: Means of Features Under Different Conditions, Healthy and Stroke Subjects

Feature	Testing Conditions (H = healthy, S = stroke, the number indicates trial number(s))							
	H1-7	S1-7	H4	S4	H5	S5	H7	S7
Hit %	77.8	59.0	89.3	68.9	90.9	76.6	77.9	55.2
Movement Time	1.20	1.62	1.03	1.69	0.94	1.37	1.32	1.69
Peak Velocity	1.13	0.92	1.06	0.84	1.19	0.89	1.17	0.99
Time to Peak Velocity	0.300	0.303	0.298	0.302	0.278	0.297	0.294	0.353
Velocity Smoothness	36.2	46.5	32.1	48.8	30.4	39.9	39.4	47.5
Response Time	0.981	1.05	1.04	1.20	1.14	1.49	0.48	0.17
Min Shoulder Angle	59.8	64.3	61.4	64.0	61.9	65.9	56.3	62.5
Max Shoulder Angle	86.2	84.9	86.6	85.4	84.6	86.7	87.4	82.1
Shoulder Range	26.3	20.6	25.2	21.4	22.7	20.7	31.1	19.6
Shoulder Excursion	34.8	26.3	32.3	27.9	28.6	24.5	43.7	26.3
Min Elbow Angle	44.0	36.8	43.7	35.8	45.2	37.2	43.1	37.4
Max Elbow Angle	83.5	78.4	83.4	77.8	79.2	77.3	88.1	80.1
Elbow Range	44.6	43.0	44.4	44.8	38.1	40.0	51.4	44.6
Elbow Excursion	55.7	51.2	52.7	56.7	44.5	43.7	70.1	53.5
Trunk Excursion	0.025	0.014	0.018	0.014	0.017	0.012	0.022	0.013
Total (integral of) Hand Yaw	9.47	17.0	7.85	18.5	7.00	13.3	11.0	14.3
Total Hand Roll	12.3	28.3	11.0	32.2	10.5	24.6	13.6	29.5
Total Hand Pitch	24.5	33.4	20.7	33.8	18.9	27.0	26.6	32.4
Total Trunk Yaw	3.67	6.46	3.02	7.04	2.70	5.02	3.58	7.12
Total Trunk Roll	4.96	4.36	4.20	3.35	3.34	3.13	5.07	4.49
Total Trunk Pitch	21.1	23.8	18.1	26.0	16.9	20.3	24.0	24.2
Total Head Yaw	2.03	2.21	1.83	2.37	1.62	1.59	2.19	2.14
Total Head Roll	2.66	5.16	2.44	5.26	2.30	3.85	2.72	4.22
Total Head Pitch	21.4	28.9	18.3	29.8	16.8	23.8	23.3	29.1

5.4 Conclusion and Data Flow Chart

The methods used in this chapter, when applied to a large data set, efficiently and effectively extracted meaningful data that proved useful for analyzing. These approaches will enable clinicians to use it for researching the physical impairments caused by stroke, and for developing better rehabilitation methods. Figure 5.9 is a flow chart that outlines the data processing.

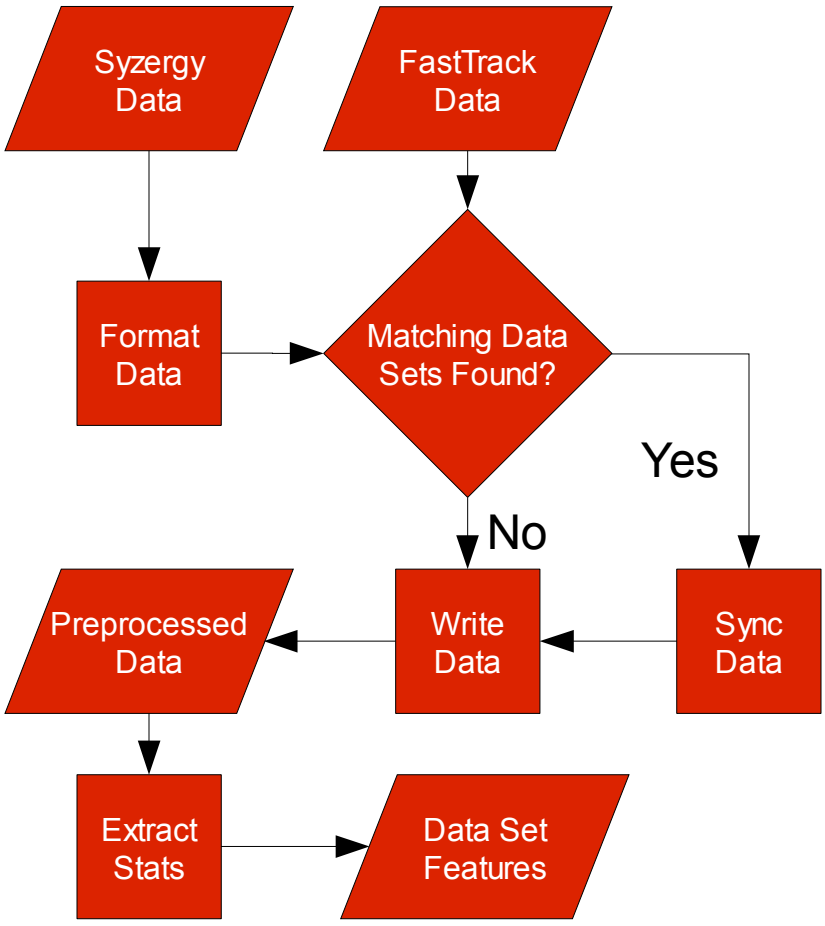


Figure 5.9: Data Processing Flow Chart

Chapter 6

Pattern Classification of Motion Data

6.1 Classifier Overview

As a proof-of-concept, a decision was made to develop a pattern classifier that can examine the data set presented in Chapter 5 and automatically distinguish between the motion of a healthy subject and that of a stroke victim. Obviously, it is not hard for a clinician to make this distinction, however, the development of a classifier will help determine those features of movement that are most affected by stroke. A generic classifier will also provide a foundation for more specific classifiers that will save clinicians time when making diagnoses and developing customized rehabilitation regimens. The code used for classification can be found in Appendix F.

Two different classifiers were developed based on the kNN and SVM algorithms. The SVM used both svmtrain and svmclassify from the MATLAB Bioinformatics Toolbox. With the kNN algorithm a data sample is classified based on what class the majority of the k samples closest to it in the feature space are. This can be useful for noisy data because it can draw very complex boundaries, but it is generally not as good as a SVM. SVMs find the

optimal hyperplanes that correctly divide the training data by looking for the hyperplanes that maximize the division between classes and maximize the distance between themselves and the nearest training samples. By doing this a SVM finds the most general classifier. Compared to kNN, though, training a SVM is much more computationally complex [23,24].

6.2 Feature Analysis

6.2.1 Individual Features

Due to the presence of so many variables in the data set, the following strategy was used to extract good classification features: (1) glean as much information as possible from simple observations of the data, (2) combine this information in (1) with intuition and prior knowledge, (3) extract any feature that has potential, and then evaluate the feature based on its PDF, and (4) conduct a quick performance test with the kNN classifier. In this way well over a hundred possible features were examined. Table 6.1 shows the results of creating a 1-dimensional kNN classifier with k=3 on each feature from Table 5.4, as well as a few others that show good potential. In the classification, the movement occurring for each target ball was 1 sample and the data was randomly split in half between training and verification data.

Another important factor in determining the use of a feature for classification is the PDF of the feature split into the various classes (stroke and healthy in this case). The more separation there is between the class PDFs, the more potential the feature has in working well for general classification [Hamid Krim, personal communication, April 2008]. PDFs of all features discussed are in Figure 6.1 and Appendix B.

Table 6.1: Classification Results in a 1-D kNN Classifier (50 Iterations)

Feature	Mean Error %	Variance
Hit or Miss	79.1	0
Movement Time	46.2	1.8
Peak Velocity	26.7	0.5
Time to Peak Velocity	35.8	0.9
Velocity Smoothness	76.6	1.2
Response Time	48.3	2.6
Min Shoulder Angle	51.9	0.5
Max Shoulder Angle	53.4	0.4
Shoulder Range	53.4	0.4
Shoulder Excursion	52.9	0.4
Min Elbow Angle	51.0	0.5
Max Elbow Angle	52.6	0.5
Elbow Range	53.3	0.4
Elbow Excursion	53.2	0.5
Trunk Excursion	24.0	0.4
Max Trunk Displacement	14.8	0.2
Max Head Displacement	24.7	0.5
Max Sagittal Trunk Displacement	20.1	0.4
Total (integral of) Hand Yaw	25.3	0.6
Total Hand Roll	23.7	0.5
Total Hand Pitch	26.3	0.6
Total Trunk Yaw	26.1	0.5
Total Trunk Roll	27.1	0.5
Total Trunk Pitch	27.3	0.7
Total Head Yaw	27.4	0.5
Total Head Roll	24.3	0.6
Total Head Pitch	26.1	0.5

6.2.2 Multi-Dimensional Classification

As can be seen by the results in Table 6.1, none of the features were outstanding, but if combined correctly they can be quite effective. First, linear combinations can be made of some of the most effective features. This offers the benefit of adding features to the classification without increasing the dimensionality of the feature space. Determining the best features to use and the best coefficients for the linear combination is in large part trial and error. In this way, 4 new features were developed that were extremely effective in the classification exercise. These were added to 2 existing features to form a set of 6 features to be used for classification. The 6 features were chosen based on their PDFs (Figure 6.1), the results of 1-D classification (such as in Table 6.1), and the results of testing the features together in multi-dimensional classifiers. As with movement time, these features would benefit from being defined wrt a neutral position rather than wrt the base of the world frame.

The best of these features was a linear combination of the maximum trunk displacement, the maximum head displacement, and the maximum trunk movement in the sagittal direction. This works because a healthy subject has a wide range of arm motion while keeping their trunk and head steady. A stroke victim might need to lunge forward to reach an object that is flying towards them. The 2nd, 3rd, and 4th features were, respectively, linear combinations of the integrals of hand, trunk, and head rotations. When reaching to swat a ball not much rotation should occur other than the hand pitch, but in stroke patients more rotation occurred. The last 2 features were movement time and hand velocity smoothness. For stroke patients, movement time was longer and the velocity less smooth.

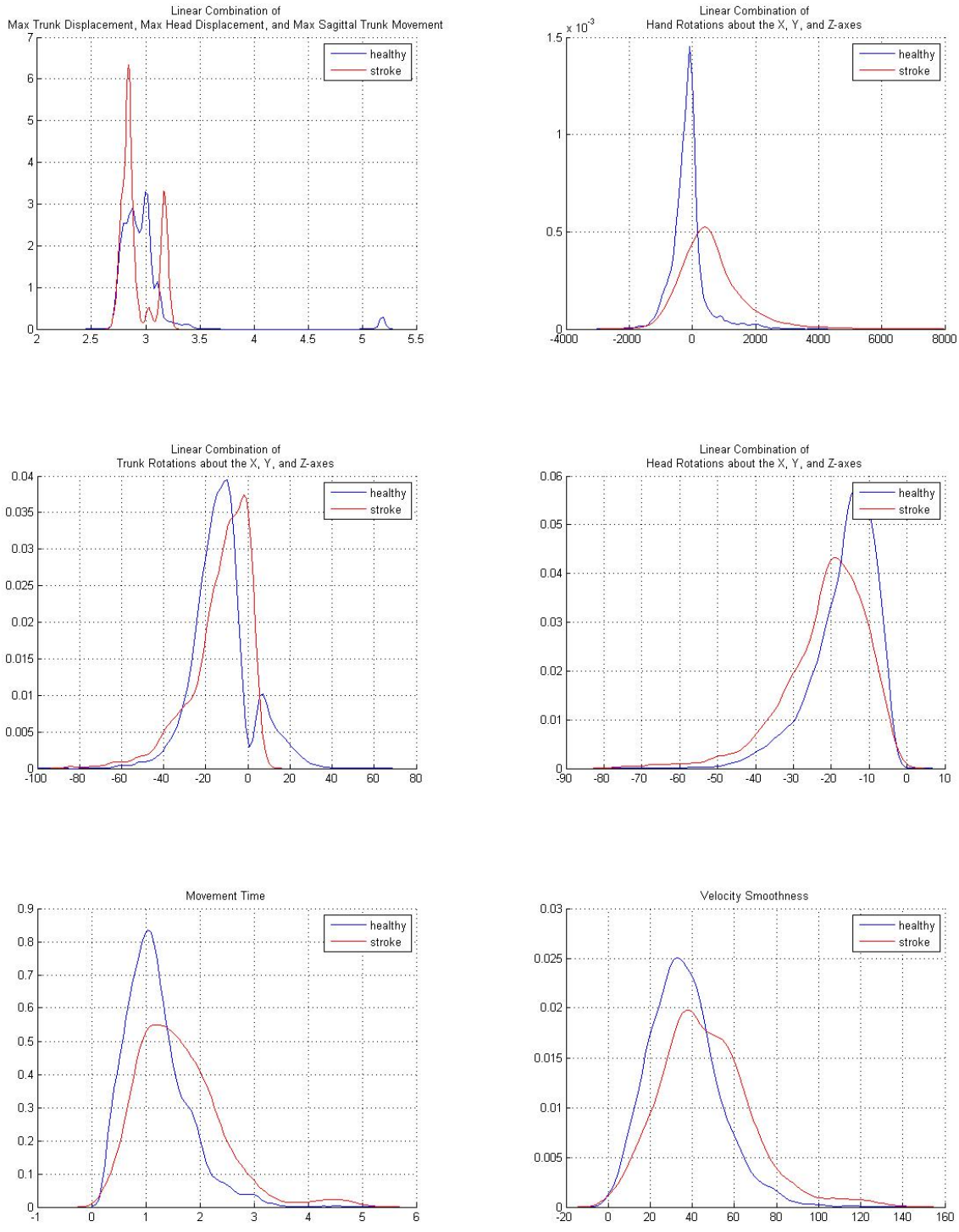


Figure 6.1: PDFs of 6 Features used in Classifier

$$Feature\,1 = 2 * max(|trunk_{position}(t)|) + max(|head_{position}(t)|) - max(|trunk_{z}(t)|) \quad (6.1)$$

$$Feature\,2 = \int |hand_{roty}(t)|dt + \int |hand_{rotz}(t)|dt - \int |hand_{rotx}(t)|dt \quad (6.2)$$

$$Feature\,3 = \int |trunk_{roty}(t)|dt + \int |trunk_{rotz}(t)|dt - \int |trunk_{rotx}(t)|dt \quad (6.3)$$

$$Feature\,4 = \int |head_{roty}(t)|dt + \int |head_{rotz}(t)|dt - \int |head_{rotx}(t)|dt \quad (6.4)$$

$$Feature\,5 = time(move_{end}) - time(move_{start}) \quad (6.5)$$

$$Feature\,6 = \sum f(t), \quad f(t) = \begin{matrix} 1, & hand_{ay}(t) > 0 \\ 0, & hand_{ay}(t) \le 0 \end{matrix} \quad (6.6)$$

The algorithms used for the 6 features being used are given above, where t ranges from the movement start time to the movement end time. Before individual features can be effectively used together, it is necessary for all features to be normalized by subtracting out the mean and dividing by the standard deviation of the training samples. Normalization allows the different features to interact with each other on the same scale. It is important that verification samples are not used in computing the mean or standard deviation of the features, since these values most likely would not be known yet in a real world classification situation.

Figure 6.2 shows the error rate and variance of 50 kNN classification iterations as each feature was added. Variance is important because it indicates the consistency of the results and it is often the worst case scenario, which is the maximum error in this situation, that is most important. To demonstrate the importance of normalizing this feature set for kNN, the error rate and the variance without normalization are also shown.

From the results in Figure 6.2, the 6th feature actually hurt performance. Adding too

many features can often cause performance decreases because it decreases the generality of the classifier and lowers the resolution of the sample space [23]. Too many features also unnecessarily increases the computational complexity. Even the 5th feature shows minimal benefit, but it will be kept for now to see if it is useful to the SVM classifier. If it has absolutely no use, PCA will eliminate it. So the 6th feature was dropped, and the normalized versions of features 1-5 were used in further analysis.

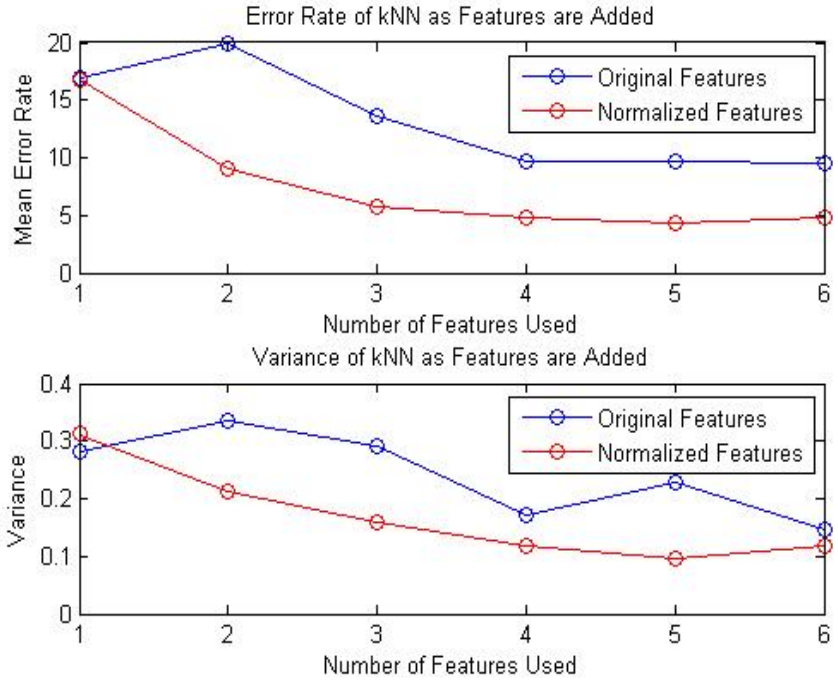


Figure 6.2: Classification Results as Features are Added (50 Iterations)

6.3 Classifier Definitions and Settings

6.3.1 Sample Space Definition

The eventual goal of this classifier was to diagnose the condition of a patient's arm, meaning each trial of each arm was treated as one sample. With the amount of data recorded there is an insufficient number of samples to create a reliable classifier. Also, as stated in Chapter 5, the most important data occurred during the actual movement of the hand towards a target ball. So, in order to simulate a more diverse and more populated sample space, each movement towards a target ball was considered as 1 sample. This is what was done in the feature analysis above. When all test conditions were used it produced over 5000 samples. It should be noted that the best data is in conditions 4, 5, and 7 since the data for those conditions includes joint angles, but using those conditions alone did not produce enough data. The joint angles did show differences between stroke patients and healthy subjects, but for the purposes of classification there were better features available.

One problem with the given data set is that although it contained many samples of motion it only contained 23 patients. So, when classifying each individual motion there was a high risk that the condition of the patient was not being classified, it may simply have been the patients that were being classified. If 100 samples from one patient were in the training data and then another 100 were in the verification data, it is likely that all 200 would be classified together. If a new patient is added to the verification data, though, it is harder to determine how the classifier would perform without re-training.

This problem could be removed by explicitly defining half of the subjects as training data rather than using a random 50% of the samples, but then resolution would be lost in the classifier. It is known that human motion can exhibit different characteristics no matter what the health of the subjects are. A good example of this is throwing a baseball: there are many ways to do it, all of which are healthy. It was important to have many patients in the learning data to account for this diversity in motion. Also, if half of the subjects were explicitly defined as training data, then the ability to choose random verification sets would be lost, which is important in determining the consistency of a classifier.

Another quasi-solution is to average the statistics from the motions towards all target balls in each trial so there will only be one sample per trial, or even one sample per arm. However, there is the possibility that resolution will be lost again and the diversity of the samples will not be good enough for classification. One potential compromise is based on averaging the results of a set number of target balls into 1 sample. Each of these sample space definitions was tested, and the classification results averaged over 50 iterations are given in Table 6.2. With SVM the tests with more than 500 training samples were skipped because the number of samples made the computational complexity unreasonable for analysis purposes.

Table 6.2: Classification Results with Various Sample Definitions (50 Iterations)

			KNN, k=3		SVM, 5th order	
Definition of 1 Sample	Method of Choosing Training Data	# of Samples	Mean Error	Training %	Mean Error	Training %
1 target ball	random	5051	4.40%	50	not tested	
5 target balls	random	1008	5.90%	50	not tested	
8 target balls	random	629	6.20%	50	6.6%	50
10 target balls	random	502	6.20%	50	8.1%	50
1 trial	random	250	8.60%	50	10.1%	50
1 trial	by patient	250	19.70%	50	14.7%	50
1 patient	by patient	43	20.40%	50	19.8%	50

Table 6.2 shows that for the feature space defined (as would be expected with 5 dimensions), a large number (over 500) of samples was needed to get somewhat consistent results. Since kNN needs a higher resolution than SVM to function well, it had a significant drop-off in accuracy when the data was split into groups based on the patient that produced the data rather than being split randomly. SVM had a 4.6% increase in error compared to 11.1% for kNN.

Judging from the accuracy of kNN, it might still seem like the best option would be to treat every target ball as one sample. The questions still exists, though, of whether it is valid to use every target ball as a sample without adding new test subjects to the verification data. kNN clearly had a greater tendency to fall victim to classifying a sample based on which patient performed the motion rather than based on whether the motion exhibited characteristics of a stroke. Also, SVM is in general a better classifier, and since using 1

target ball made the computational costs of training SVM unreasonable, this sample definition was not used for analysis purposes.

Data reduction using PCA will alleviate some of the computational problems associated with SVM, as will using a smaller ratio of training samples to verification samples. The computational costs of training SVMs with a training set larger than 500 samples were fairly high, and although there was a small increase in performance with this many samples, the problems of using that many samples outweighs the benefits during analysis. To ensure that kNN had enough samples and the sample space was diverse, more than 300 samples needed to be in the training set , though, so 8 target balls was defined as 1 sample. This gave 629 total samples. Using more than 1 target ball per sample could also help eliminate noise caused by random movements. After the classifier analysis is finished, however, and the best feature space and the best classifier parameters are determined, it might be good to use fewer target balls as 1 sample if the classifier is put into practice.

6.3.2 Classifier Parameters

A few things were taken into consideration when selecting the parameters for the classifiers. First, as mentioned before, it was known that human motion can exhibit different characteristics no matter what the health of the subjects are. This means that healthy motions would most likely appear in the feature space as clusters rather than one giant mass.

Originally, kNN was done using k=3, and the SVM was set to be 5^{th} order. Using k>1 for kNN should allow it to not be affected as much by noisy data, but keeping k relatively small should prevent a neighboring cluster of data from having as much affect on the

classification. Of course, an odd-numbered k should be used in order to break ties. Setting the SVM to be 5^{th} order should give it enough freedom to define multiple boundaries, but should restrict it enough to prevent overfitting. The order refers to the order of the polynomial that is created as the boundary between classes.

In Table 6.2, the SVM demonstrated poorer performance than kNN in the random data sets, which was not expected. This might have been due to the improper order polynomial being used. Once the feature space was completely defined, a quick test was run to determine the best parameters for kNN and SVM in this classification problem. Figure 6.3 shows the accuracy and variance for kNN as k increased and the accuracy and variance for SVM as the order of the polynomial increased.

For kNN, the most accurate value of k turns out to be 1, which had an error rate of 5.7%. So plain nearest neighbor in this case was actually better than k nearest neighbor. For SVM, the best order polynomial was 3, which had an error rate of 5.5%. As expected, the SVM overtook the kNN in accuracy when the optimal parameters were chosen.

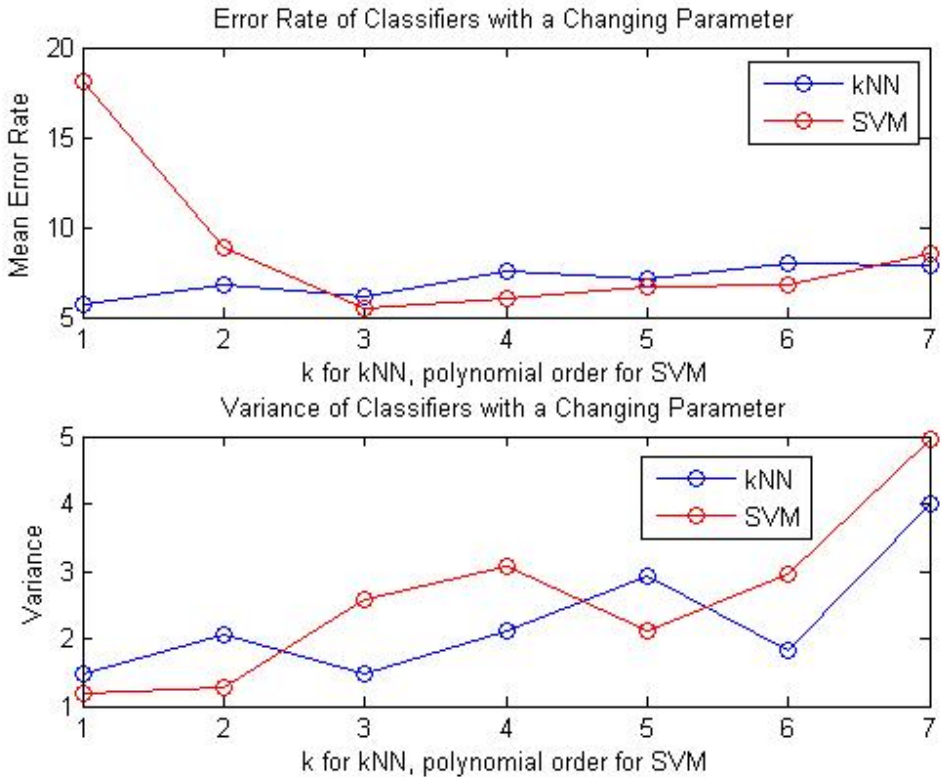


Figure 6.3: Classification Results with Varying Classifier Parameters (50 Iterations)

6.4 Principle Component Analysis of Features

When dealing with high-dimensional feature spaces, it is sometimes desirable to perform data reduction. One of the most common methods of doing this is PCA. PCA transforms the feature space into a data set ordered by the amount that each component contributes to the data's variance. PCA is the optimal linear transformation for performing this task, but has a higher computational complexity than some other methods. Also, if the data is not Gaussian, PCA will merely de-correlate the data [25].

Many times performing PCA allows lower-order components to be removed

altogether from the data because the data assumed to be most important and interesting has been moved into the higher-order components. If this cannot be done, PCA still offers the advantage of being able to better analyze and visualize the data on a 2 or 3 dimensional level.

There are a few techniques for performing PCA, and even some that can operate on non-linear data. The most common method involves finding the eigenvectors of the data's covariance matrix. First the data must be normalized by subtracting out the mean and dividing by the standard deviation. This step was already done with the data used here. Then the eigenvectors of the covariance matrix are found and the data is rearranged in order of decreasing eigenvalue. So the principal component is the one with the highest eigenvalue. The original data is then projected into the principal component subspace. PCA was performed here by using princomp in MATLAB's Statistics Toolbox.

Once the data was reduced using PCA, the performance of kNN and SVM was analyzed for K=1 to 5, where K is the number of principal components used. The results are shown in Figure 6.4. As determined by prior analysis, kNN is k=1 and SVM is 3rd order. A random 50% of the samples was used for training data in both cases. The performance trends show that the 5th principal component was unnecessary, and in the case of SVM it was even harmful. This makes sense based on the eigenvalues shown in Table 6.3. The 1st 3 principal components were shown to be much more important by their large eigenvalues, but the 4th principal component still had a significant impact on classification despite its small eigenvalue. The error rate at K=4 was 5.4% for SVM and 5.9% for kNN, which is statistically equivalent to the results from using the full feature space.

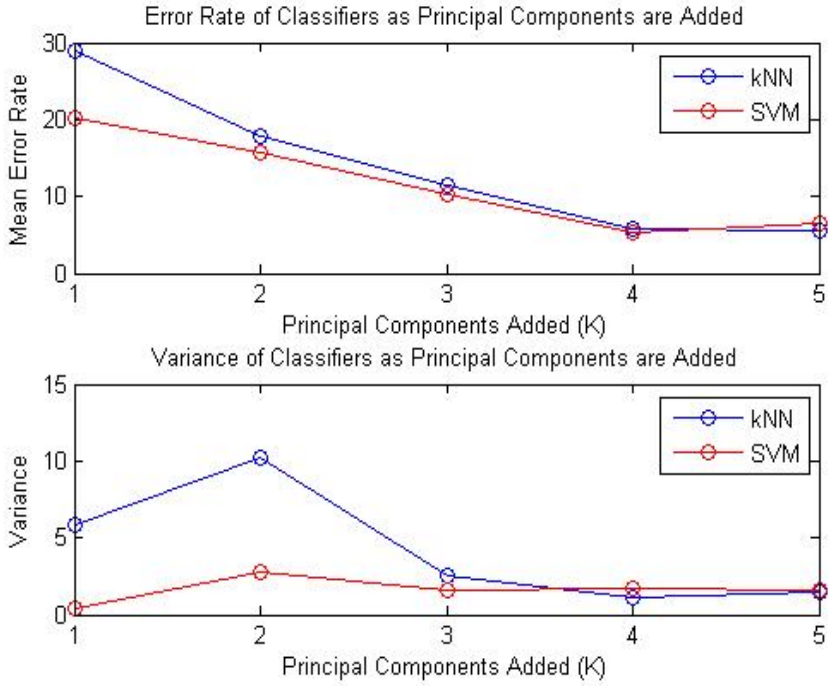


Figure 6.4: Classification Results with PCA (50 Iterations)

Table 6.3: Eigenvalue of Components in PCA of Classifier Features

Principal component #	1	2	2	4	5
Eigenvalue	450740	362130	108941	0.0868	0.0738

Reducing the feature space further by 1 dimension should also have a large benefit on the abilities of the classifiers to operate with fewer training samples. It also decreases the computational complexity. Since the 5th principal component was not used in further analysis, it is desirable to know what the results would have been in reconstructing the original feature space from the 4 components used. PCA provides an optimal transformation, but there still must be an error involved in this reconstruction. The error for this reconstruction is given by the equation below [23], and for K=4 it was computed to be 0.037.

$$e = \frac{1}{2} * \sum \lambda_i, \quad i = (K+1) \rightarrow N \tag{6.7}$$

6.5 Training Analysis

The final decision for the classifier was determining the correct amount of training data to use. This could easily be done if there were an infinite amount of data available, but here, as with most data sets, there were a limited number of samples. So, if the amount of training data was merely increased until the classifier accuracy started decreasing, then the process might run out of data and before that happened the results would become invalid due to the extremely small number of samples that were left for verification.

For the training analysis 25% of the data was selected each time as verification data. The remaining 75% was used as training data and the amount of that 75% actually used for training varied from none of it to all of it. If the performance of the classifier had still been showing significant improvement when the training data was exhausted, then the only conclusion that could have been drawn is that the optimal amount of training data was some amount greater than what was available. The results of these tests are shown in Figure 6.5.

Both curves appear to adhere to the law of diminishing returns. For kNN, the error rate seems to asymptotically approach a number just below 6%. As shown by the graph, the error rate really seems to level off near this asymptote at about 330 samples where the error rate was 5.8%. For SVM, the error rate asymptotically approaches a number just below 5%. This can first be seen graphically at just over 300 samples, where the error rate was 4.9%.

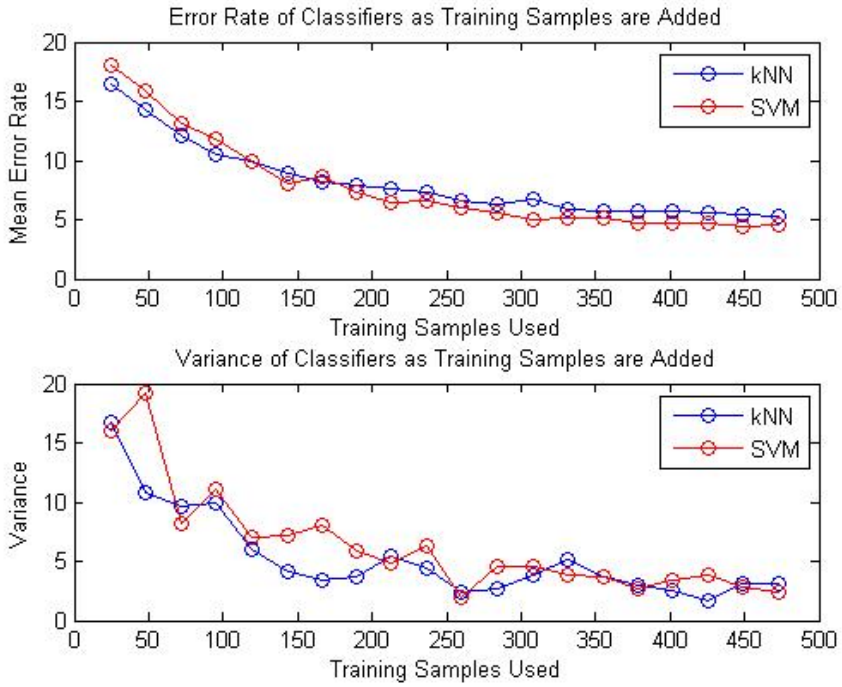


Figure 6.5: Classification Results as Training is Increased (50 Iterations)

6.6 Final Classification Results

6.6.1 Results

With the data set and compilation of samples defined as they were for analysis and training, and all of the parameters for classification and training set to what was found to be their optimal values, the final classification results are shown in Table 6.4. SVM had better and more consistent results than kNN, so a 3rd order polynomial SVM was the best classifier found. To ensure the validity of the classifiers, each set of verification data in the final tests were switched with their corresponding set of training data [Edward Grant, personal communication, May 2008]. Nearly identical results were obtained each time, which is the

desired outcome.

The biggest problem encountered with SVM was the computational complexity in constantly retraining it for analysis. With 629 samples, and 330 of them used for training, the best kNN classifier took an average of just 59.4 milliseconds to execute. The best SVM classifier, with 307 training samples, took an average of 24.9 seconds just to train. Once it was trained, however, it took an average of only 3.4 milliseconds to classify the verification data. So the learning for a SVM was computationally intense, but this doesn't matter as much in application as it does in analysis since once the best settings for the classifier have been determined training only needs to occur once. Once this is done, the classification is actually quicker than kNN.

Table 6.4: Final Classification Results

	Mean Error	Variance	Parameter	PC's Used	Training Samples	Test Samples
SVM	4.9%	4.5	3rd Order	4	307 (49%)	322
kNN	5.8%	5.1	k = 1	4	330 (52%)	299

Lastly, it would be useful to see what effect the various testing conditions had on the classifier. To do this, the data from testing conditions 4, 5, and 7 were each individually classified. Since this greatly decreased the number of samples, it was necessary to revert to defining 1 sample as the movement occurring during 1 target ball, rather than the average of 8 target balls. Because of this, and because of a slightly lower training ratio (around 300 out of 850 samples), the results can not be directly compared to the results in Table 6.4. The performance should still be somewhat similar, though. For a 3rd order SVM classifier, the

results are given in Table 6.5. Conditions 4 and 5 have nearly the same performance, which is good because it shows that the feature space and the classifier are mostly independent of those testing conditions. Condition 7 has slightly worse performance, which might be a result of the ball releases being too complicated for even some healthy subjects to perform well on.

Table 6.5: Classification Results for Individual Testing Conditions (50 Iterations)

	Mean Error	Variance
Condition 4	5.6%	0.8
Condition 5	5.6%	1.0
Condition 7	8.5%	1.1

6.6.2 Final Classification Flow Chart

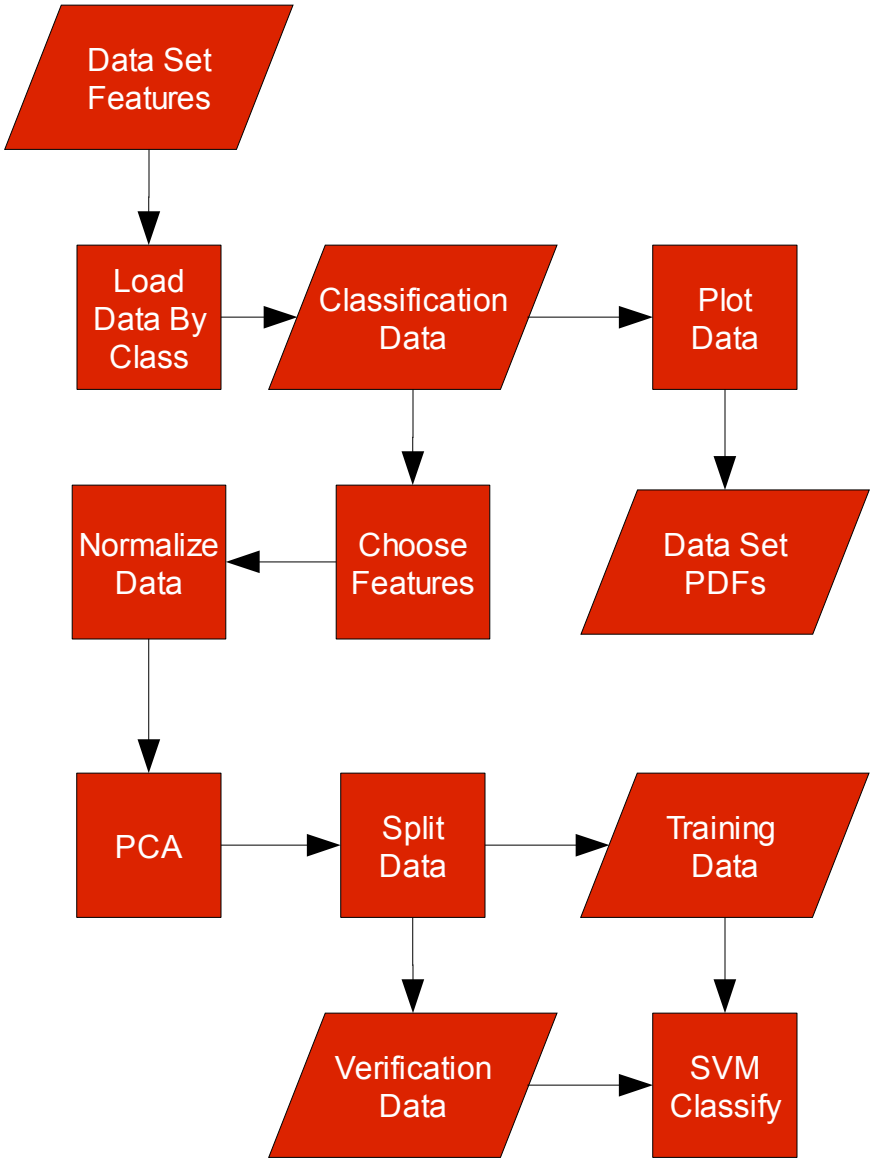


Figure 6.6: Classification Flow Chart

6.6.3 Possible Improvements

As stated earlier, the data was broken up by target ball rather than by test subject. In reality, the classifier needs to determine the condition of a patient's arm rather than the condition of one motion. This could be done by classifying each motion performed by a test subject, and then simply classifying the subject to be in whatever category the majority of the motions are in. The ratio of stroke-classified motions to healthy-classified motions could possibly even be used as a crude measure of the severity of the subject's condition.

The classifier developed demonstrated that computer recognition of a stroke victim's motion is possible, but to really be effective the data set needs to be expanded. The research needs more subjects, and only data from truly stroke-afflicted subjects and truly healthy subjects should be used for classifier training. It severely harms the classifier when a subject who is deemed a stroke victim but has healthy motion (or vice versa) is used for training.

It would also be good if the data collection from the FastTrack system was a little more consistent so that joint angle data would be more reliable. Adding some kind of protection to prevent electromagnetic interference would help as would ensuring the marker stays within the system's electromagnetic field. It is also critical that the FastTrack system and the Syzergy system are without a doubt collecting data at known sampling frequencies.

Finally, it would be good if a few extra parameters are recorded, such as a neutral resting position. Hopefully with these data set improvements, the feature space and some of the techniques developed in this analysis will be useful and could help create an effective tool for diagnosing physical problems caused by stroke.

Chapter 7

Conclusion

7.1 Summary of Results

A system was successfully developed that accurately and efficiently imports C3D mocap data into OpenSim. This will prove highly useful for clinicians and researchers, because it gives them access to open-source musculoskeletal modeling software. OpenSim was shown to be reliable for modeling mocap data, and its IK tool was shown to be reliable at least for simple joints, i.e., knee and elbow. With the data available it was difficult to determine its accuracy for more complicated joints such as the shoulder.

Although software such as OpenSim is good for modeling and analyzing musculoskeletal motion to determine parameters of movement, it does little to help clinicians and researchers understand those parameters. All the data must be reduced to relevant features and analyzed to determine patterns and characteristics of motion. To demonstrate this, a large data set of mocap data from stroke patients and healthy subjects was analyzed. Important statistics in the data were identified and a pattern classifier was developed that could use some of these statistics to identify the motion of a stroke victim.

7.2 Possible Applications and Future Work

The intention of most of the work completed here was not to be an end in itself, but to provide a foundation for future work. The work done to apply C3D data to OpenSim models will help increase the availability of tools that have previously only been accessible through proprietary software. The dynamics and the muscle control calculated by OpenSim data could be applied to determine attachment points and external forces needed for exoskeletons [26]. The development of neural prostheses and FES would also benefit [27,28].

For example, mocap data for a healthy motion could be recorded, then applied to an OpenSim model. The dynamics of that motion would be calculated, which could then be reproduced by an exoskeleton. If dynamics are not available, the kinematics of the motion could be put into a virtual fixture that would create a path of least resistance for reproducing this motion [29]. These techniques would be useful in rehabilitation or in sports training.

The classifier developed for recognizing the distinct patterns of motion exhibited by stroke victims was a proof-of-concept. The goal was to be able to automatically distinguish an arm affected by stroke from a healthy one. This is obviously a distinction that can easily be made by a clinician, but it is hoped that the concept can be expanded upon to help clinicians make more difficult diagnoses and to provide physical therapy that is more accurately tuned to the needs of each patient. This could help improve recovery, lower costs, and provide doctors with valuable time to treat more patients. It is important to note that this is not meant to create a fully automated system; it is meant to be a tool similar to AI-based systems that help doctors diagnosis other medical conditions.

References

[1] Post-Stroke Rehabilitation, (2000), U.S. Dept. Health and Human Services, Public Health Service, National Institutes of Health, Bethesda, Maryland.

[2] Economic Impact of Acute Ischemic Stroke. *Medical News Today*, February 26 2006.

[3] Delp, S.L., Anderson, F.C., Arnold, A. S., Loan, P., Habib, A., John, C., Thelen, D.G. OpenSim: Open-source software to create and analyze dynamic simulations of movement. IEEE Transactions on Biomedical Engineering, vol. 54, pp. 1940-1950, 2007.

[4] "OpenSim Overview", (SimTK), Available: https://simtk.org/home/opensim, (Accessed: 2008, June 19).

[5] Hammer, S, Anderson, C, Guendelman, E, John, C, Reinbolt, J, Delp, S. OpenSim Tutorial #3: Scaling, Inverse Kinematics, and Inverse Dynamics. Neuromuscular Biomechanics Laboratory, Stanford University.

[6] Anderson, F. C. (2007, October 25). import SIMM model using Simbody engine [Msg 2]. Message posted to http://groups.earthlink.com/forum/messages/00025.html

[7] Anderson, F.C., Guendelman, E, Delp, S. Generating a Muscle-Actuated Simulation in OpenSim. Neuromuscular Biomechanics Laboratory, Stanford University.

[8] Thelen DG, Anderson FC. Using computed muscle control to generate forward dynamic simulations of human walking from experimental data. J Biomech 39: 1107-15. (2006)

[9] Thelen DG, Anderson FC, Delp SL. Generating dynamic simulations of movement using computed muscle control. J Biomech 36: 321-328. (2003)

[10] John, C.T., Anderson, F.C., Guendelman, E., Arnold, A.S., Delp, S.L. An algorithm for generating muscle-actuated simulations of long-duration movements, *Biomedical Computation at Stanford (BCATS) Symposium*, Stanford University, 21 October 2006, Poster Presentation.

[11] Loan, P, Delp, S, Smith, K, Blaikie, K. SIMM 4.0 for Windows User's Manual. MusculoGraphics, Inc. June, 2004

[12] MusculoGraphics Brochures, (MusculoGraphics, Inc.), Available: http://www.musculographics.com/contactus/brochures.html, (Accessed: 2008, June 19)

[13] AnyBody Technology A/S. The AnyBody Modeling System Tutorials Version 3.0. September, 2007.

[14] AnyBody Technology Medical/Rehab, (AnyBody Technology), Available: http://www.anybodytech.com/163.0.html. (Accessed: 2008, June 19)

[15] Re-lion, (Re-lion), Available: http://www.re-lion.com/home.html, (Accessed: 2008, January 20).

[16] DynaMechs (Dynamics of Mechanisms): A Multibody Dynamic Simulation Library, (Scott McMillan), Available: http://dynamechs.sourceforge.net/, (Accessed: 2008, January 19).

[17] Thingvold, J, Van Baerle, S. Motion Capture File Format Review. LambSoft, Inc. Minneapolis, MN.

[18] Motion Lab Systems. The C3D File Format User Guide. Updated: 2008, Jaunary 17. Available: ftp://ftp.c3d.org/mls/c3dformat.pdf.

[19] Menache, Alberto. Understanding Motion Capture for Computer Animation and Video Games. San Francisco, CA: Morgan Kaufmann, 1999

[20] Holzbaur KR, Murray WM, Delp SL.: A model of the upper extremity for simulating musculoskeletal surgery and analyzing neuromuscular control., Ann Biomed Eng. 2005 Jun;33(6):829-40. (2005)

[21] Delp, S.L., Loan, J.P., Hoy, M.G., Zajac, F.E., Topp E.L., Rosen, J.M.: An interactive graphics-based model of the lower extremity to study orthopaedic surgical procedures, IEEE Transactions on Biomedical Engineering, vol. 37, pp. 757-767, 1990. (1990)

[22] Papoulis; Pillai. Probability, Random Variables, and Stochastic Processes: 4th Ed. New York: McGraw-Hill, 2002.

[23] Duda, R; Hart, P; Stork, D. Pattern Classification: 2nd Ed. New York: John Wiley & Sons, Inc., 2001.

[24] Burges, Christopher. A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery, 2, 121–167 (1998). Kluwer Academic Publishers, Boston.

[25] Jolliffe, I. T., Principal Component Analysis, 2nd edition. Springer, 2002.

[26] Merritt, Carey. A pneumatically actuated brace designed for upper extremity stroke rehabilitation. Master's Thesis, North Carolina State University, 2003.

[27] Davoodi, R, Brown, I.E., Loeb, G.E. Advanced Modeling Environment for Developing and Testing FES Control Systems. Medical Engineering & Physics 25 (2003) 3-9. April 25, 2002.

[28] Davoodi, R, Urata, C, Hauschild, M, Khachani, M, Loeb, G. Model-Based Development of Neural Prostheses for Movement. IEEE Transactions on Biomedical Engineering, Vol. 54, No. 11, November 2007.

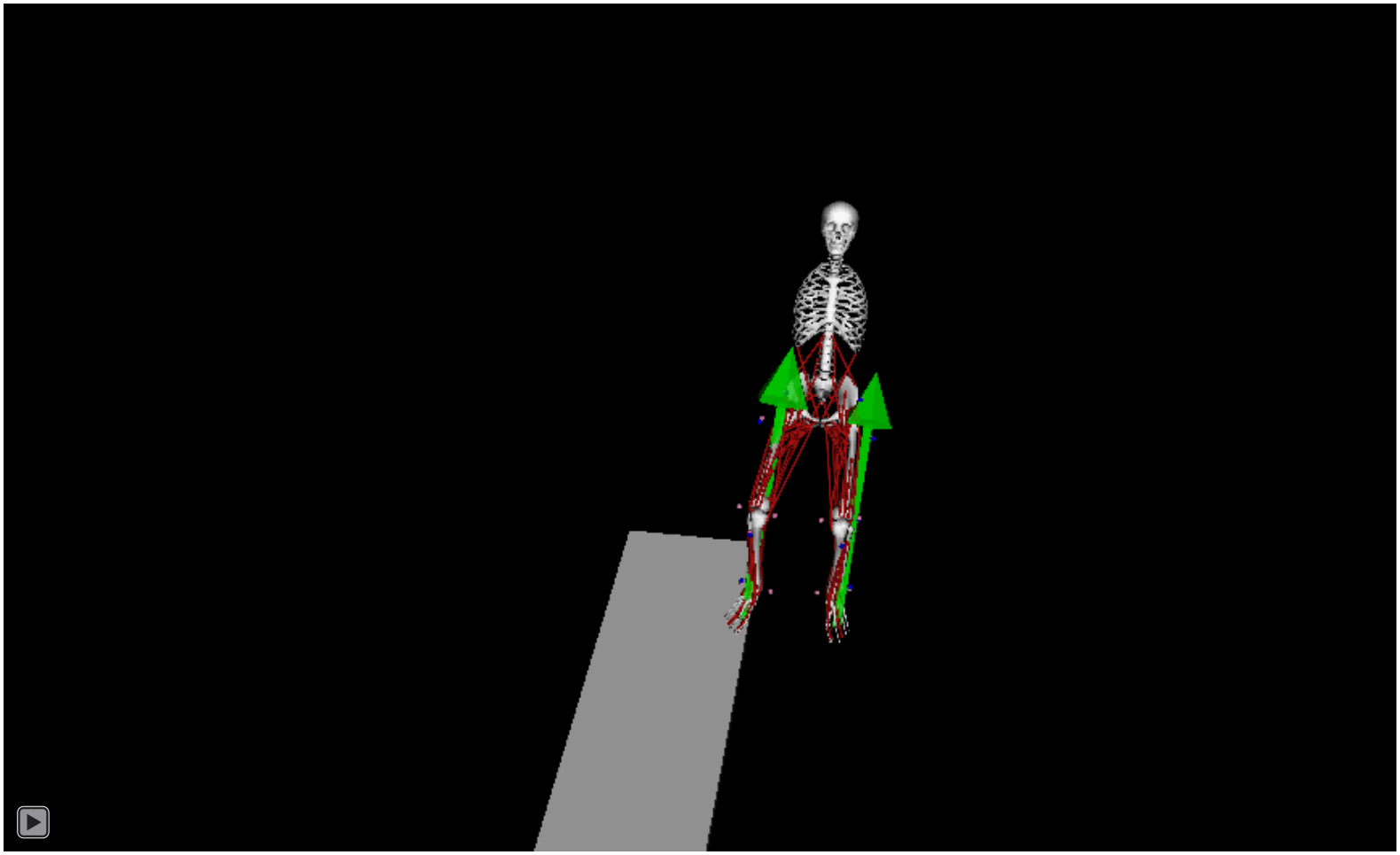
[29] Abbott, J.J.; Hager, G.D.; Okamura, A.M. Steady-hand teleoperation with virtual fixtures The 12th IEEE International Workshop on Robot and Human Interactive Communication, 2003. Proceedings. ROMAN 2003.

Appendices

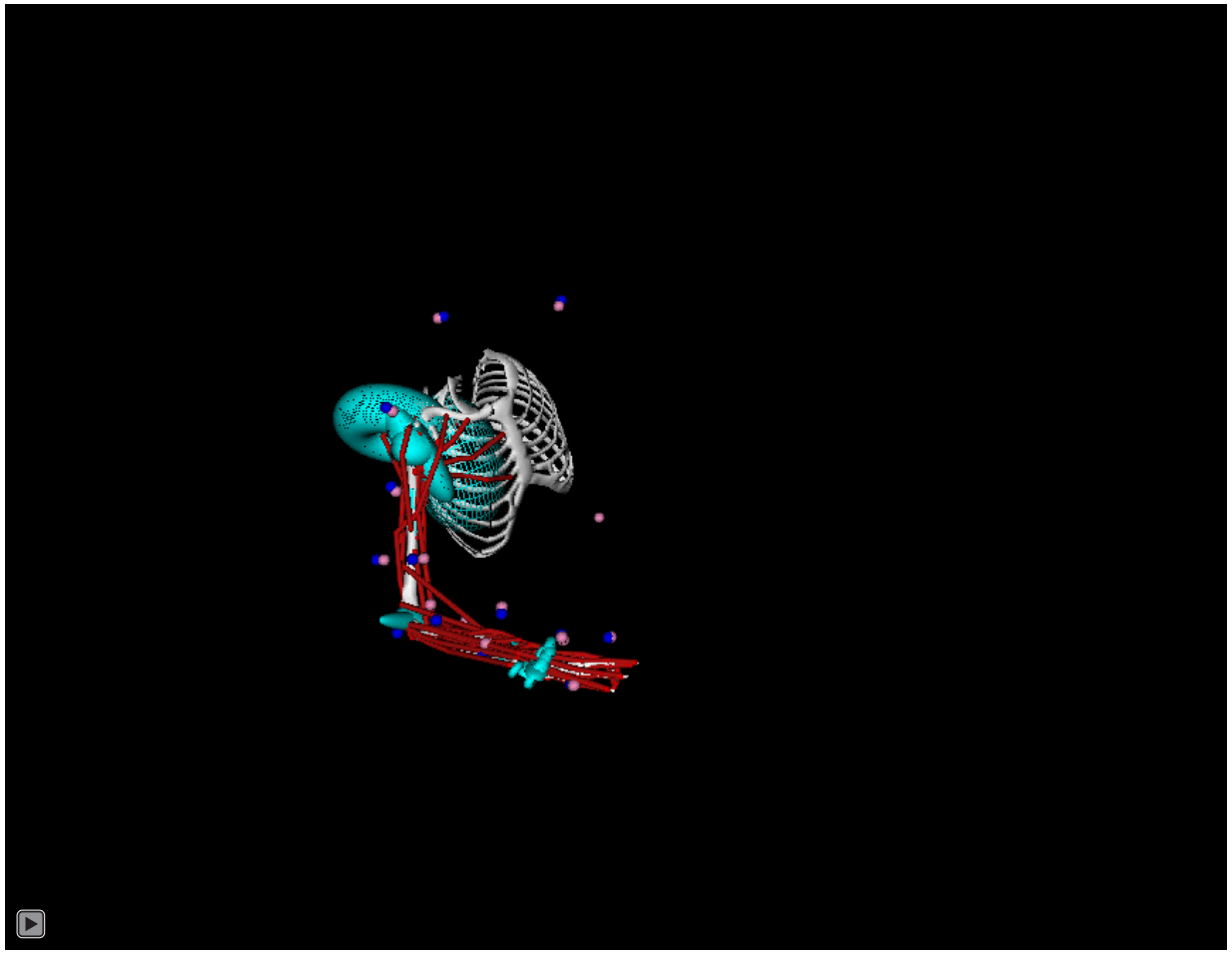
Appendix A – Motion Videos

The following two videos are mocap data played back in OpenSim. The first video is using the gait2392 model that is provided with OpenSim. The mocap data was collected at UNC on a subject who was jumping. The trial name is DJM27S1 Jump1. The second video uses a modified version of the Upper Extremity Model provided with the SimTK library. Motion capture data was also collected at UNC and the subject in this video is repeatedly reaching for and answering a phone that is on a table. The final video is a sample motion video produced by the AnyBody modeling system that was obtained from the AnyBody website.

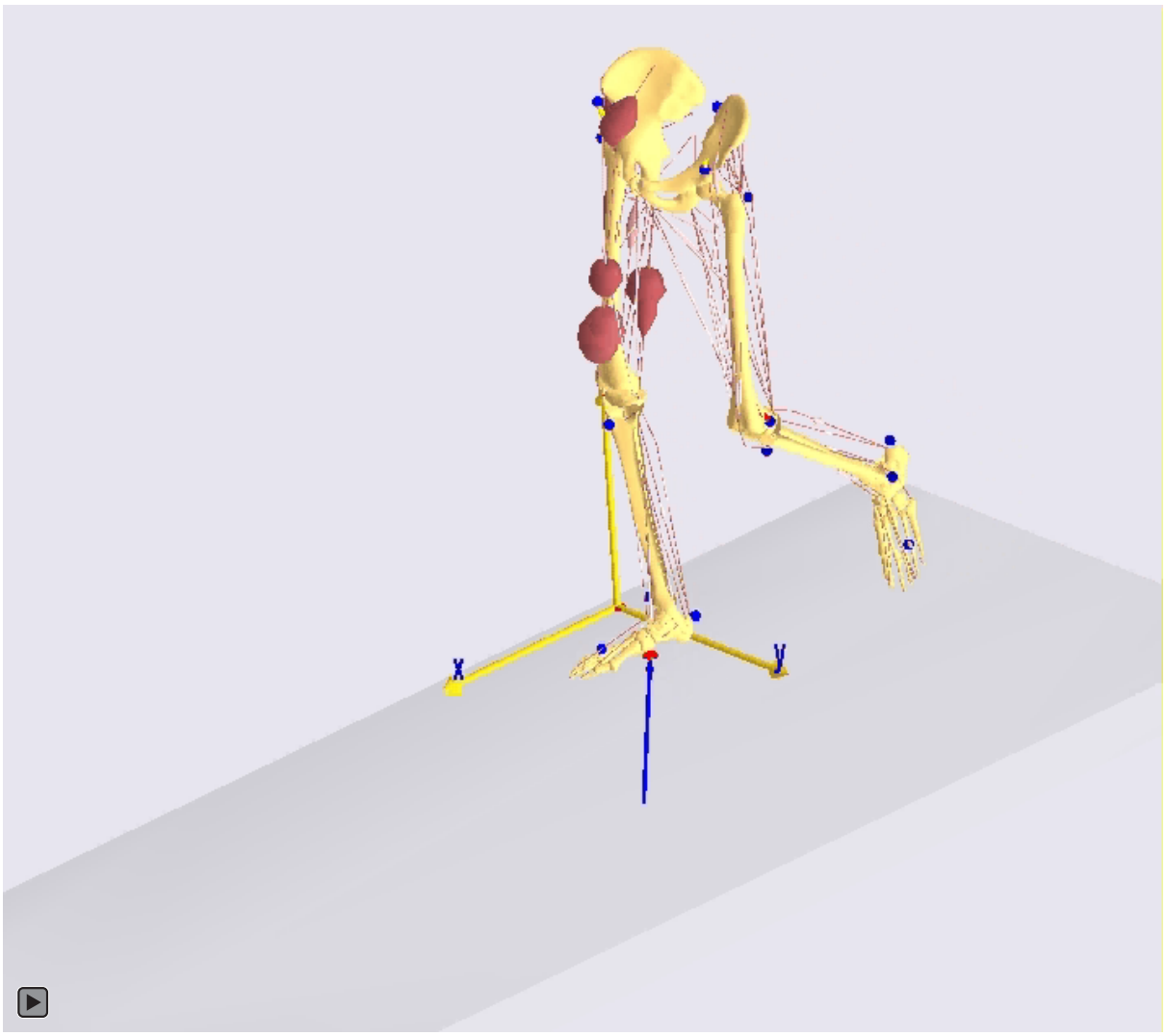
1. DJM27S1 Jump1



2. Reach Arm Angles



3. AnyBody Sample Video



Appendix B – Additional PDFs, VR Data

The PDFs of the following features from the VR Data of chapter 5 are given in this appendix:

1. Peak Velocity
2. Time to Peak Velocity
3. Response Time
4. Minimum Shoulder Angle
5. Maximum Shoulder Angle
6. Shoulder Range
7. Shoulder Excursion
8. Minimum Elbow Angle
9. Maximum Elbow Angle
10. Elbow Range
11. Elbow Excursion
12. Trunk Excursion
13. Maximum Trunk Displacement
14. Maximum Head Displacement
15. Maximum Trunk Sagittal Displacement
16. Total (integral of) Hand Yaw
17. Total Hand Roll
18. Total Hand Pitch
19. Total Trunk Yaw
20. Total Trunk Roll
21. Total Trunk Pitch
22. Total Head Yaw
23. Total Head Roll
24. Total Head Pitch

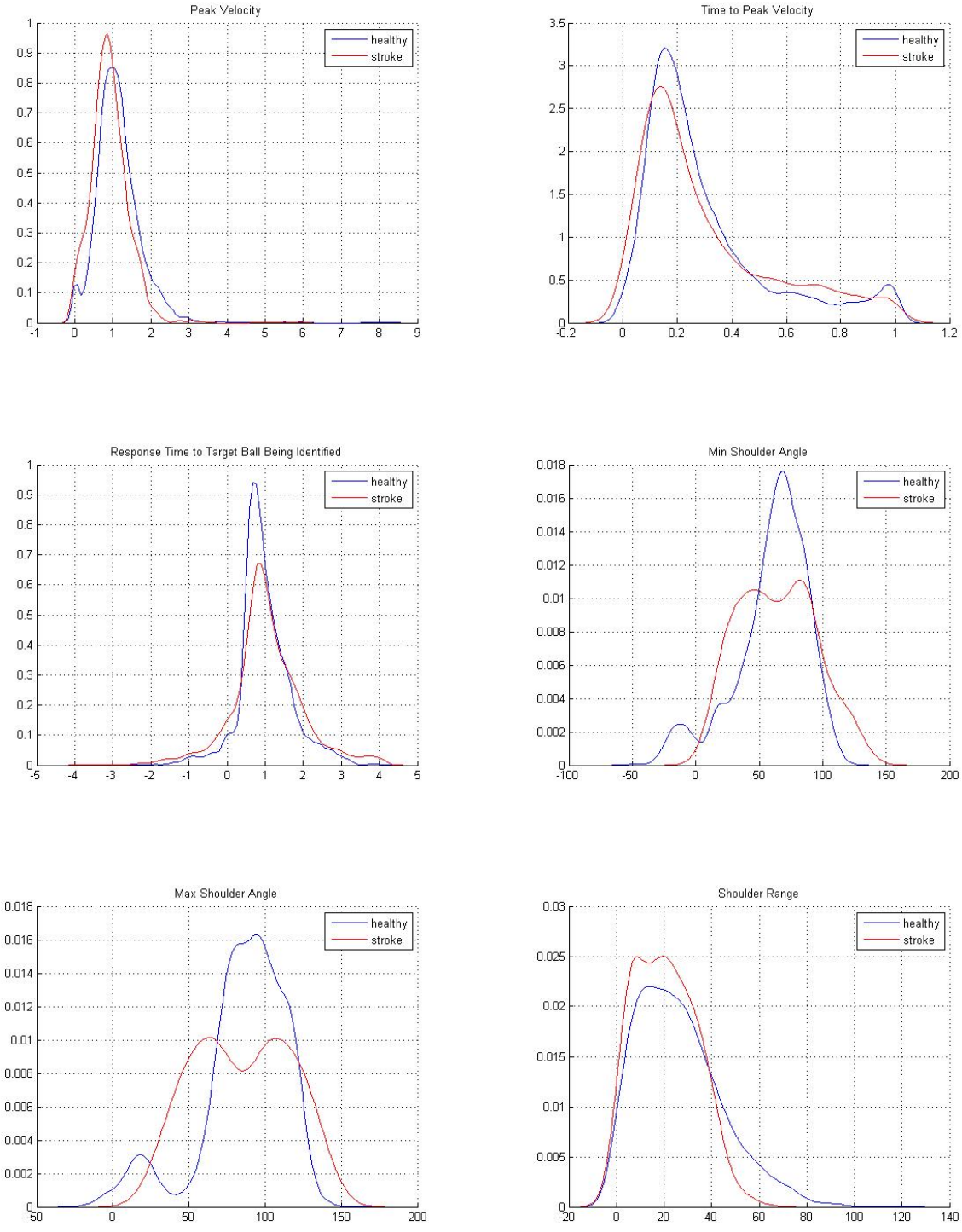


Figure B.1: VR Data PDF Set 1

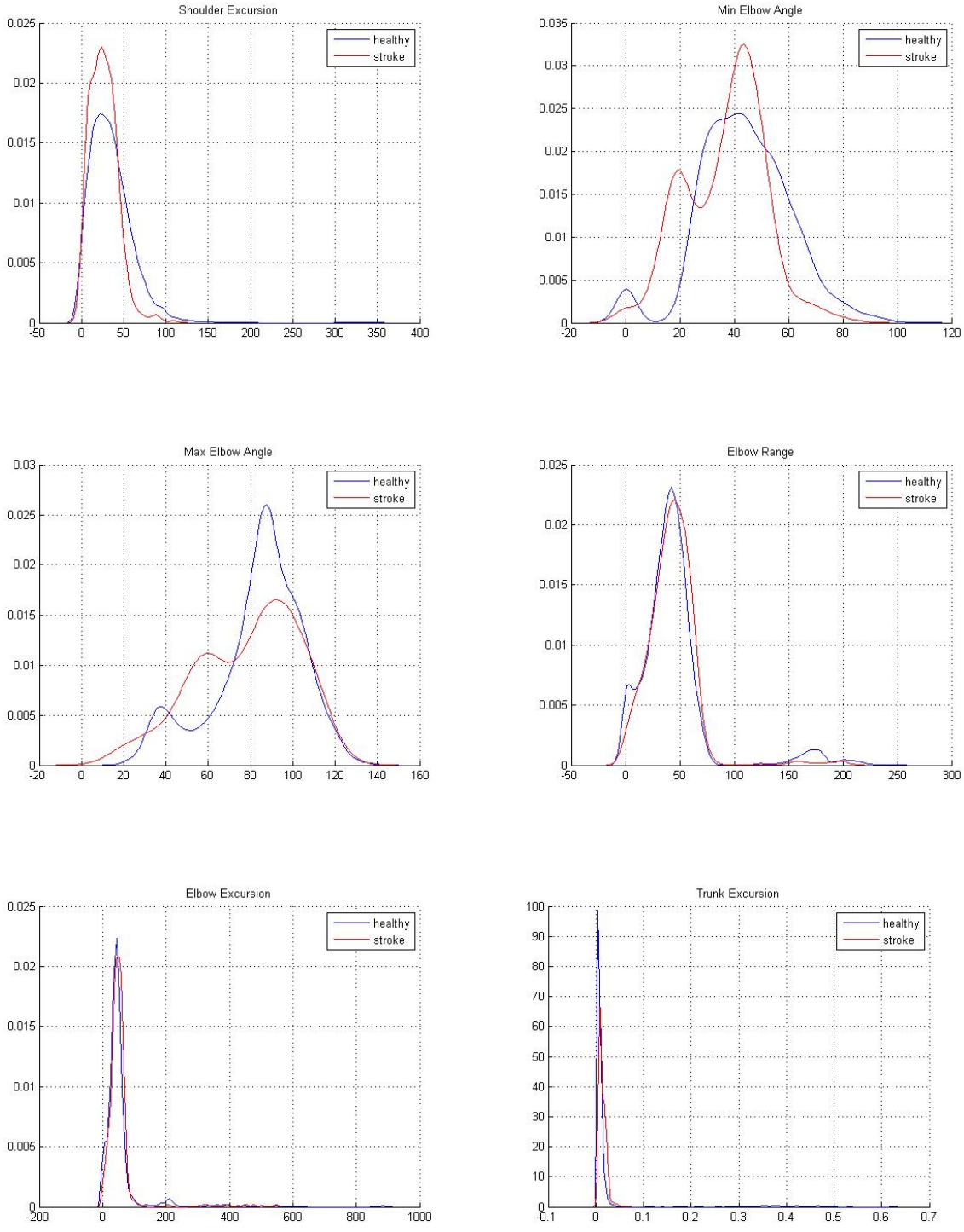


Figure B.2: VR Data PDF Set 2

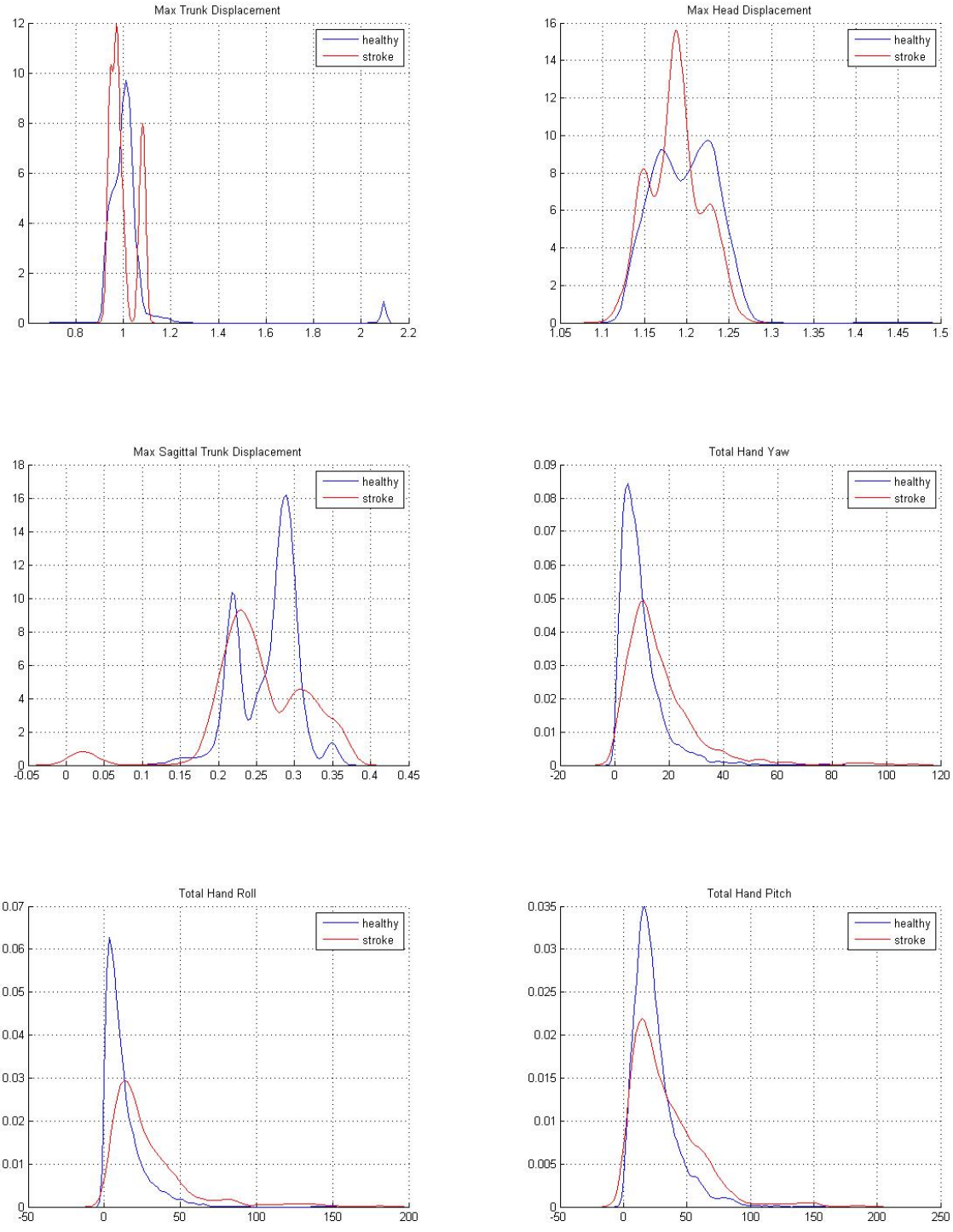


Figure B.3: VR Data PDF Set 3

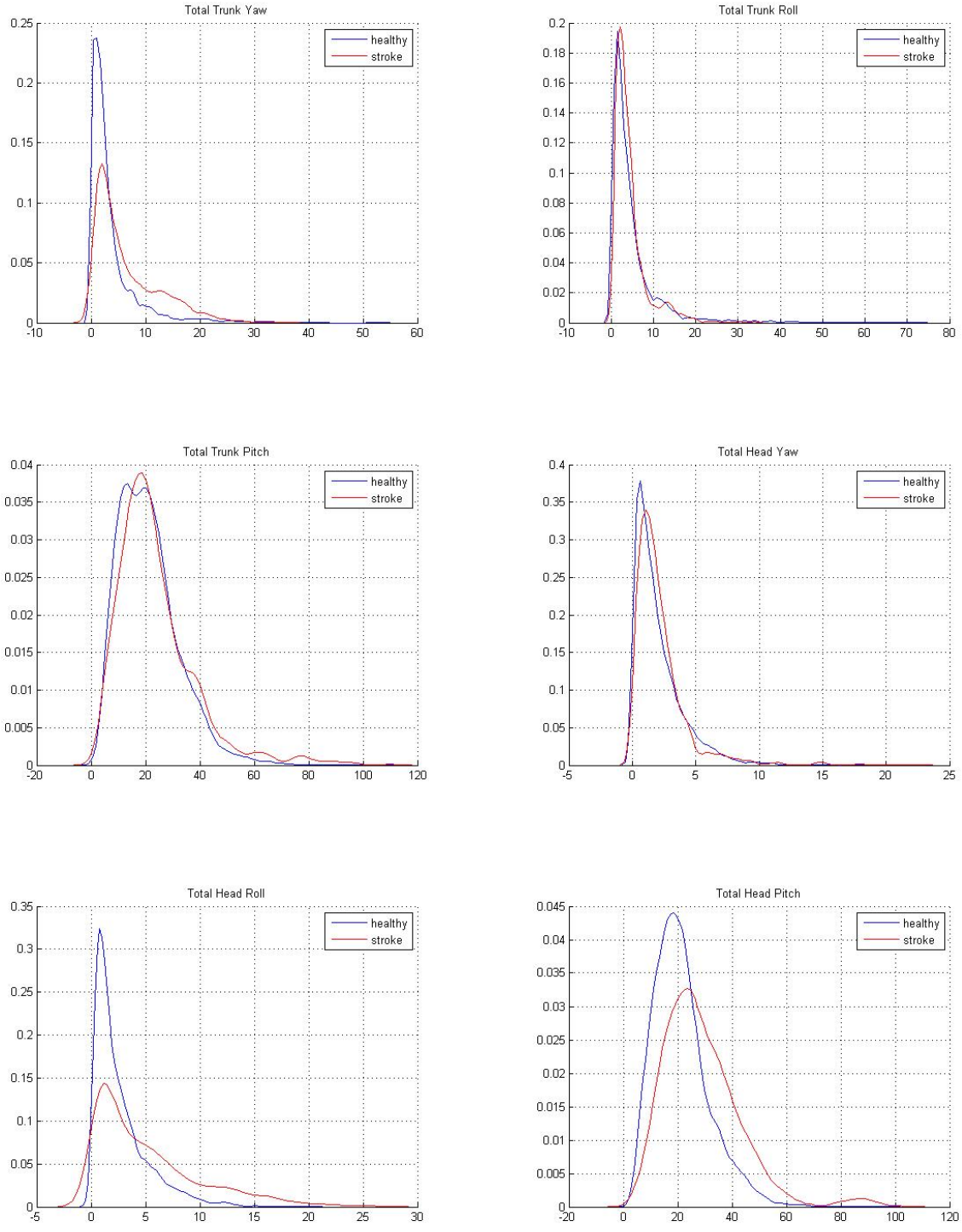


Figure B.4: VR Data PDF Set 4

Appendix C – Program Instructions

C.1 C3D_2_OSim Instructions

Using C3D_2_Osim in MATLAB to convert C3D motion capture files into formats usable by OpenSim

The steps below are mostly initial setup and won't need to be performed many times, some of them only once. If the C3D file is named example.C3D and contains valid marker data, example.trc will be created. If valid general coordinates are loaded from an OpenSim model, then example_zeros.mot will also be created. If example.C3D contains analog data then example.anc will be created. If the analog data contains forceplate data and foreceplate calibration information is provided then example_grf.mot will be created. The two .mot files will then be combined into example.mot. Refer to the documentation in the code for more information on any function. It might also be desirable to change some settings in Stanford's process_grf.m function (filtering options, contact thresholds, etc.).

1) Before using C3D_2_OSim, read the documentation at the top of C3D_2_OSim.m. It might be necessary the first time to experiment with a few parameters that vary between different motion capture systems. For now, hit cancel when the program asks for model dictionaries, lookup tables, and forceplate files. The program should still be able to create .trc and .anc files.

2) A model dictionary needs to be created once for each OpenSim model. To do this, call the function make_OSim_jct(). It will need to load either an OpenSim model that contains markers, or an XML file that contains the model's marker information. If an XML file was loaded, the OpenSim model will need to be loaded as well. Once the marker labels and gencoords have been extracted, they will be saved to a .mat file.

3) If forceplate data is used, a forceplate calibration file must be manually created once for each system setup. To do this, create a structure called FPinfo that contains one structure for each forceplate used (the order in which the forceplates are added should match the order in which they appear in the C3D file). The structure for each forceplate should contain the following fields:

calMatrix - 6x6 calibration matrix for the forceplate
orientationMatrix - matrix for rotating from forceplate axes to mocap system axes
originTranslation - vector for translating to mocap system axes (after rotation)
gain - gain of the forceplate
Note: Bertec4060a.mat is included as an example file that contains 2 forceplates.

4) Call C3D_2_OSim and load the OpenSim dictionary and forceplate files. If conv_lbls is true, the program will try to convert marker labels to the labels used by the OpenSim model. The first time it will ask for each C3D marker's OpenSim equivalent. Once these conversions have been input once, they can be saved in a lookup table that can be reused for other C3D files.

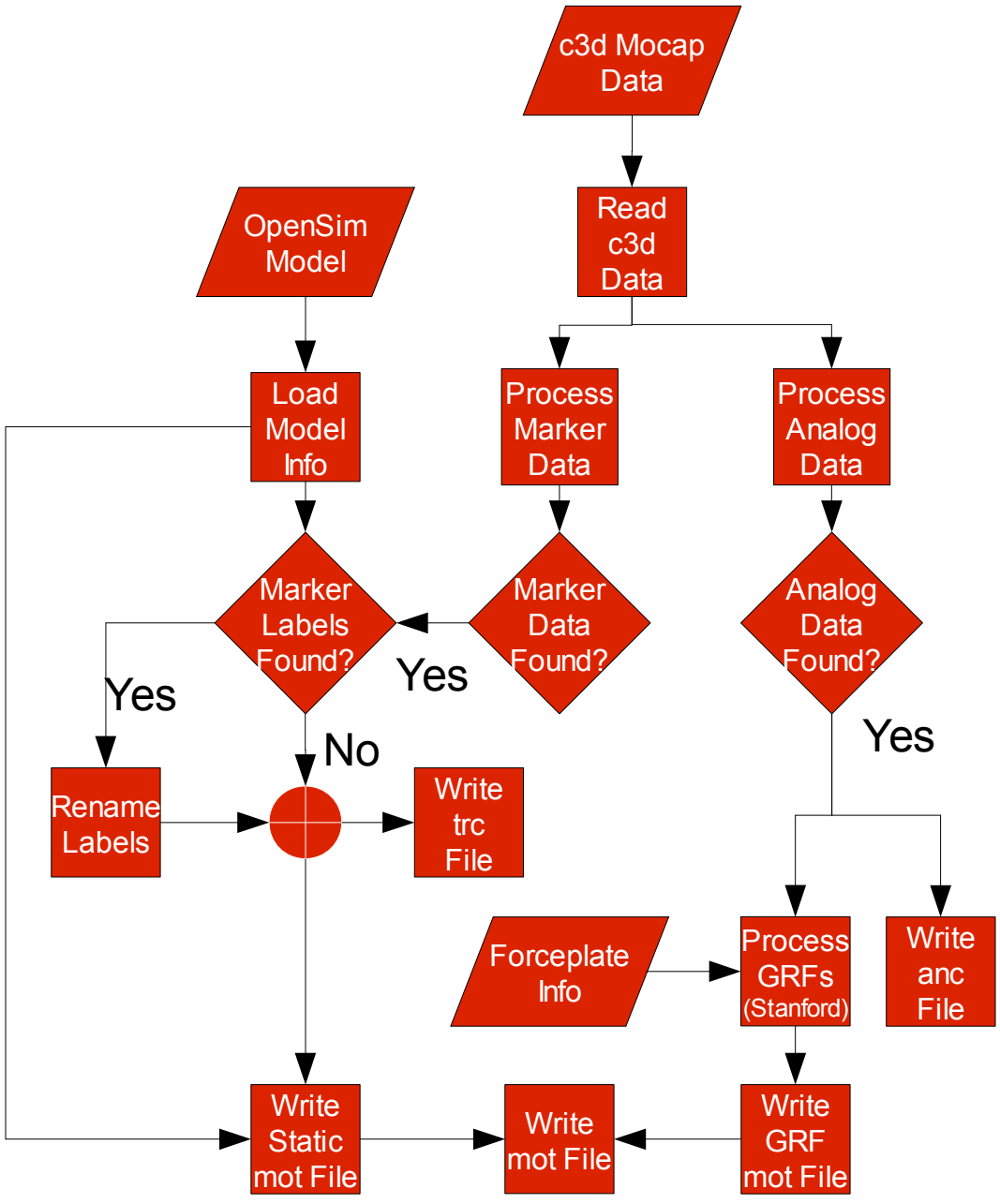


Figure C.1: C3D_2_OSim Execution Flow Chart

C.2 VR Reaching Data Instructions

Using MATLAB to preprocess and analyze stroke reaching VR data collected by Duke and UNC

The steps below outline how to use the provided MATLAB functions to analyze the VR stroke reaching data collected jointly between Duke and UNC. Refer to the documentation in the code for more information on any function. To alter what features of the data are analyzed, analyzeSYG.m will need to be edited, and to alter how features are grouped together into different classes compileFeats.m and loadClassData.m might need to be edited.

1) Call preprocessSYG(out_format, do_plots). out_format determines if the files are output in csv or mat format. do_plots determines whether plots are shown during preprocessing (currently this means that the synchronization plots for the Jason_syncd data would be shown).

2) Call extractFeats(in_format, conditions, verbosity). in_format determines the format of the input files (csv or mat). conditions should be a vector containing the condition numbers (1-7) to use. Input 0 for conditions to use all 7 trial conditions. verbosity determines whether the analysis results will be output and plotted for each file.

3) Call [Data, Group, feat_lbls, class_lbls] = loadClassData(windowSize). windowSize determines how many target balls to average together in order to form one sample. Input 0 to average all target balls from each trial. Currently the samples are divided into healthy arm samples and stroke arm samples. Edit the groups defined by Group and class_lbls at the bottom of loadClassData.m to change the class definitions. New classes can easily be defined as any combination of: healthy subject dominant arms, healthy subject non-dominant, stroke-affected dominant, stroke non-dominant, stroke patient unaffected dominant, and stroke patient unaffected non-dominant.

4a) To further analyze the features based on class, call plotClassFeats(Data, Group, feat_lbls, class_lbls, features_used, reduction). Data, Group, feat_lbls, and class_lbls should come from the output of loadClassData. features_used is a vector containing the numbers of the features that will be analyzed. Input 0 to analyze all features. reduction determines whether the features are normalized and reduced using Principle Component Analysis (PCA) before they are plotted and analyzed.

4b) To test the effectiveness that a k-Nearest Neighbor (kNN) or Support Vector Machine (SVM) pattern classifier might have on a given feature set, call testClassfiy(Data,Group, training_ratio, test_params, features_test, features_set). For more information on the input parameters, please refer to the documentation at the top of testClassify.m

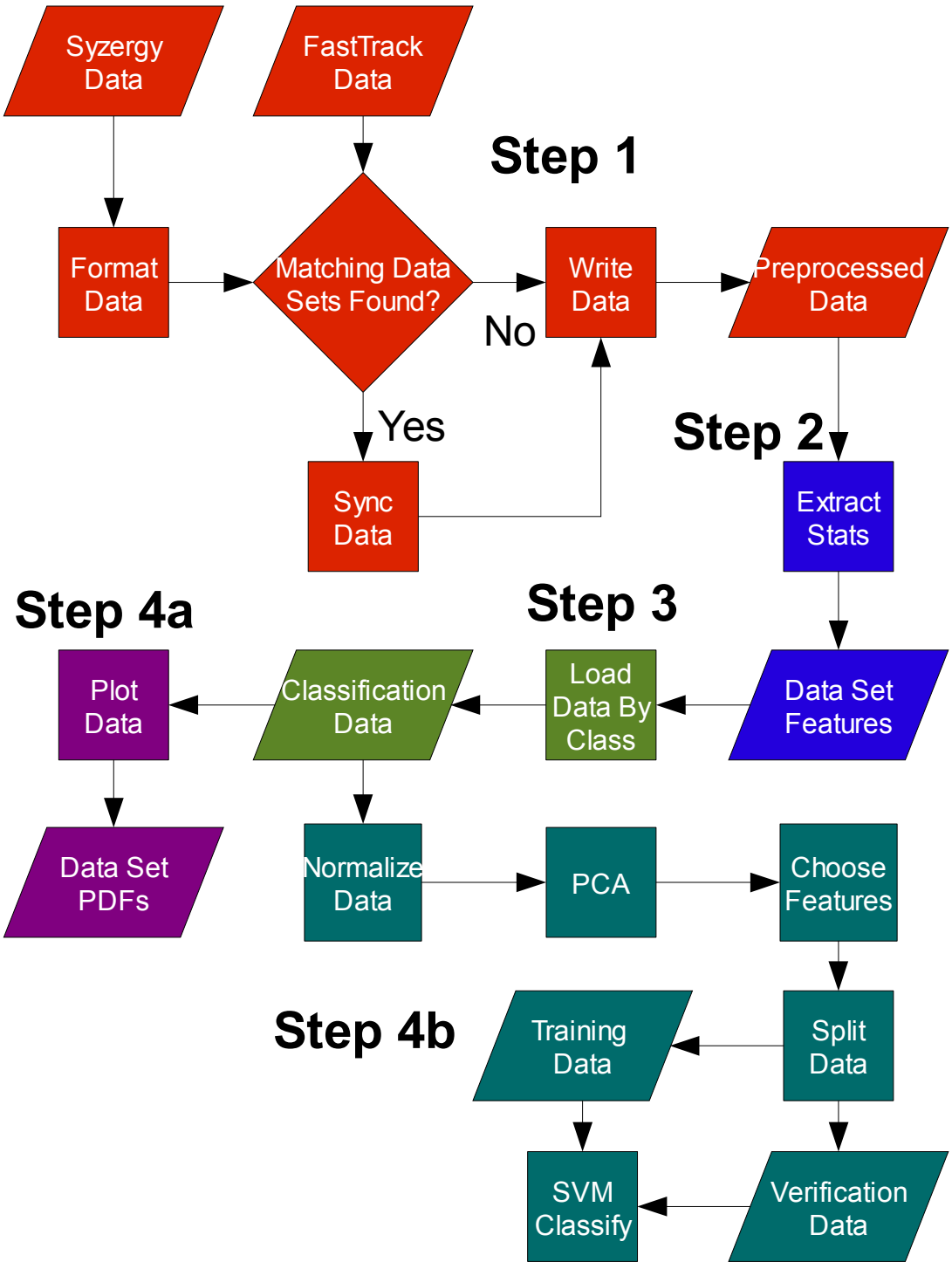


Figure C.2: vr_data Analysis Flow Chart

Appendix D - VR Data File Information

General Information

1) All the relevant files are named in the following way: healthy (H) or experimental (E), # (1 to n), limb (L), # (1= right, 2 = left), condition (C), # (4/5/7). So, H6L1C4 means healthy subject number 1, right limb, condition 4.
2) There were 11 E subjects – people with stroke, and 12 H subjects – age and gender matched healthy normals.
3) Every subject used both limbs to complete the tasks, we are interested in both.
4) There were 7 different conditions, but we are only interested in 4, 5, 7.
5) The data files contain information about the kinematics of the movements made by the balls and the people – more specific information later.
6) Each file for condition 4, 5, 7 contains multiple trials, but always 20 target ball releases. There may be additional balls released, but some were not target balls. Each subject was supposed to reach for target balls only and not to reach for the distractor balls. One trial is defined as a target ball release with or without an associated movement attempt by the person.
   a) For condition 4, 20 balls were released consecutively, all of which were target balls, immediately identified.
   b) For condition 5, 50 balls were released consecutively, 20 of which were target balls, immediately identified, and the order of release was randomised.
   c) For condition 7, 20 releases were made. Each release consisted of 5 balls. One of those 5 balls was the target ball, but there was a time delay before it was revealed as the target ball.
7) Data was collected with multiple systems.
   a) One system recorded the orientations and positions of sensors placed on the head, hand and trunk of the subject as well as the positions of the balls, all relative to a global reference frame. These files are in a strange format – the data need to be arranged into columns so they are useable and meaningful.
   b) One system recorded the orientations of the trunk relative to a global reference frame as well as elbow and shoulder angles. The recording was done at the same time, but the files were not electronically synchronised. We need to synchronise the files using the orientation of the trunk.
   c) One system recorded the EMG activity of many muscles of the patients upper extremities. A pulse should show up in the signals which coincides with a ball release. This should allow the files to be synchronised. These files are monopolar EMG and very messy, so I'm not sure what can be done with them if anything at all.
8) Time is reported in seconds.
9) Effective sampling frequency of the two kinematic systems was 30 Hz.

10) Rotation about any axis relative to the origin is reported in degrees.
11) Translations relative to the origin are reported in feet.
12) All translations and rotations are measured relative to the origin of a global reference frame that was set so that it was on the ground underneath a person sitting in a chair. The chair was facing a VR screen and they saw the balls appear in the screen and fly towards them. The orientation of the global axis system is as follows:
13) Positive Z went from the screen towards the subject, so you will see the Z translation of the ball start of very negative and move towards zero
14) Positive X went from left to right if you face the screen in the negative Z direction
15) Positive Y was straight up

SYG Kinematic Files

1) There is a folder in the repository labelled "syg". Inside this folder are sub-folders labelled E(1-11) and H(1-12). Each of these folders contains entire data files for a subject completing conditions 4, 5 and 7 with each limb. These files have extensions "*.DAT" and are space delimited.
2) These are kinematic data files with information about sensors placed on the head, hand and trunk of each subject, giving positions and orientations of the head, hand and trunk.
3) Also within these files are information about the balls locations.

These are the files that need to be re-formatted before they can be used.

4) The first two columns should show time and global time, but I think there is something wrong with them. We do not need these values because we have the sampling frequency and we can get time increments from that, i.e. each NEW row should be 0.033 seconds incremented from the preceeding row.
5) The third column denotes if the row information is about a sensor or a ball. A zero means the row represents a sensor and a one means the row represents a ball.
6) If the row represents a ball, then the number in the fourth column after the one in the third column shows which release the ball was, starting at zero for the first release.
7) If the row represents a sensor then the fourth column denotes where the sensor was placed; 0 = head, 1 = hand, 2 = trunk.
8) Columns 5, 6 and 7 are the X, Y and Z coordinates of the translation of the middle of the sensor or ball from the origin of the global reference frame – in feet.
9) Columns 8, 9 and 10 are the Euler rotations of the sensors about the X, Y and Z axes. For the balls, column 8 indicates if the ball is a target ball or distractor ball – a 1 indicates the ball is a target ball. Column 9 indicates if the target ball was revealed to the subject as a target ball, a 1 indicates that the ball has been revealed as a target ball, this changes part way through the trial for condition 7. Column 10 indicates if a target ball has been contacted by the subject, a 1 indicates that the target ball has been contacted.
10) The last 16 columns are the 4x4 position and rotation matrix of the sensor, these are not important to us because we will just keep things simple and use columns 5 to 10 for positions and orientations.

This is what we need:

11) The files in their current format are not useful. We need them to be organised so that each row represents one time point and each column represents one unique piece of data. Can you re-arrange the file so that for <u>one row</u>:
    a) The first column is time in seconds, effectively delete the current first two columns and replace them with one showing time based on the sampling frequency of 30 hertz.
    b) The next 18 columns contain the position and orientation data for the sensors, i.e. column 2-4 = position data of the head sensor, 5-7 = rotation data of the head sensor, 8-10 = position data of the hand sensor, 11-13 = rotation data of the hand sensor, 14-16 = position data of the trunk sensor, 17-19 = rotation data of the trunk sensor. We don't need the number of the sensor if the data is organised in this way, as long as they are kept consistent.
    c) All columns after that contain position data about the ball currently in view. You would probably be best doing this by creating three times as many columns as there are balls and each ball has its own set of three columns, i.e. column 20-22 = ball 1, 23-25 = ball 2, etc, and when a ball disappears all values underneath the final position are set to zero. For some of the trials this would mean an awful lot of columns, but I think this is the best way to keep things organised.
12) Please note that if you look at the file *e1l1c1.dat* (not one we need for an actual analysis) that the ball 0 disappears before ball 1 appears (~row 613), so you will probably need to use the data in the third and 4th columns to rearrange the data file. You can't rely on their being 4 rows (head sensor, hand sensor, trunk sensor, ball) just like that throughout the data file.

Jason_Syncd Kinematic Files

1) There is a folder in the repository labelled "Jason_Syncd". Inside this folder are sub-folders labelled E_L1, E_L2, H_L1, and H_L2. Each of these sub-folders contains data files for E or H subject completing conditions 4, 5 and 7 with either limb 1 or 2. These files have extensions "*.EXP" and are tab delimited.
2) These are kinematic data files with information about shoulder flexion/extension, elbow flexion/extension, and the orientation of the trunk.
3) Column 1 is sample number, each sample represents 1/30th of a second (same as the syg files). Column 2 is shoulder elevation, column 3 is elbow flexion, and columns 4, 5 and 6 are the X, Y and Z translations of the trunk sensor.
4) The trunk sensor used in these files was placed right next to the trunk sensor in the syg files, so we need to synchronise these files by matching the translations of the trunk sensor. I think it should be possible to do this if we can graph the 3-D coordinates of both sensors and select where they look the most similar and match the time at that point(s).
5) It would then be nice to create just one data file per trial where all the information from the Jason_Syncd files are in the syg files so they can be analysed.

Analysis of Data

We would generally produce the following to analyse the kinematic data:

1) Hand Path - A 3-D plot of time against positions for the hand sensor.
2) Hand Velocity - A 3-D plot of time against velocity for the hand sensor (requires numerical differentiation, and therefore smoothing – easy to smooth in Matlab, even I can do that!).
3) Joint Angles – The Elbow

4) For each trial/condition/subject, Carol would like to compare the following variables for the hand sensor:

    a) Movement Time (absolute, in seconds) – From when the hand sensor starts to move until the ball is contacted or missed.
    b) Peak Velocity (m/s) - this requires numerical/vector differentiation of the 3-D coordinate data wrt time, then reading off the peak value.
    c) Time to Peak Velocity (% of movement time)
    d) Smoothness of Velocity Trajectory – Difficult, maybe look at acceleration or number of peaks of velocity?
5) Other important variables are:
    a) Response Time (seconds) - Difference in time between the target ball being revealed and the first movement of the hand. Really only informative for condition 7.
    b) Peak, Min, Max and Excursions of the joint angles – both numerically and graphically.
    c) Excursion of the Trunk During Each Trial (distance, plus angle relative to frontal plane).
    d) Ball Speed, start point, and end point.

Appendix E – C3D_2_OSim Code

E.1 Initial Setup

E.1.1 make_OSim_dict.m

```
function make_OSim_dict
% make_OSIM_dict: Reads OpenSim model files to find the model's gen coord
%                 and marker labels.
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load marker label information
[file,dir] = uigetfile({'*.osim','OpenSim model with markers (*.osim)';...
    '*.xml','Scale Tool markers file (*.xml)'},...
    'Choose file with marker information');
fid = fopen([dir file],'r','n');

if (fid==-1)
    error('make_OSim_dict:file','%s could not be opened.',file);
end

temp = fscanf(fid,'%s');
temp = regexpi(temp,'<Markername="(.*?)">','tokens');

i_lim = length(temp);
osim_markers = cell(1,i_lim);
for i=1:i_lim
    osim_markers(i) = temp{i};
end
osim_markers(strcmpi('default',osim_markers)) = [];

fclose(fid);

fprintf('Successfully loaded %d marker labels.\n',length(osim_markers));

%% load gen coord label information
if (~strcmpi(file(max(strfind(file,')):length(file)),'.osim'))
    [file,dir] = uigetfile({'*.osim','OpenSim model (*.osim)'},...
        'Choose file with gen coord information');
end
fid = fopen([dir file],'r','n');

if (fid==-1)
```

```
    error('make_OSim_dict:file','%s could not be opened.',file);
end

temp = fscanf(fid,'%s');
temp = regexpi(temp,'Coordinatename="(.*?)">','tokens');

i_lim = length(temp);
osim_gencoords = cell(1,i_lim);
for i=1:i_lim
    osim_gencoords(i) = temp{i};
end
osim_gencoords(strcmpi('default',osim_gencoords)) = [];

fclose(fid);

fprintf('Successfully loaded %d general
coordinates.\n',length(osim_gencoords));

%% save information
[file_name,file_dir] = uiputfile('*.mat','Save marker dictionary as');
save([file_dir file_name],'osim_markers','osim_gencoords');

return
```

E.2 Main Program

E.2.1 C3D_2_OSim.m

```
function C3D_2_OSim(conv_type, conv_lbls, upsample_markers)
% C3D_2_OSim: Converts a C3D file into files usable by OpenSim.
%             Marker data is placed in a TRC file.
%             Analog data is placed in an ANC file.
%             Processed ground reaction forces are added to make MOT files.
%
% IMPORTANT:
% *Some C3D files put analog data in 32 bit floats, others use 16 bit ints.
%  Trial and error is the easiest way to determine the correct format.
% *Be sure transform is the correct transform matrix for transforming your
%  system's world frame to OpenSim's X-forward, Y-Up, Z-right frame.
%
% Input:
% conv_type            (OPTIONAL) 'single' for one file, 'batch' for a dir,
%                                 'batch_r' for recursive dir
% conv_lbls            (OPTIONAL) 1 to convert label names, 0 otherwise
% upsample_markers     (OPTIONAL) 1 to upsample marker data to analog rate,
%                                 0 otherwise
%
```

```
% Important Parameters:
% transform          3x3 transform matrix for transforming to OpenSim frame
%                    OpenSim axes are x-forward, y-up, and z-right
% c3d_analog_format  analog data format in c3d file ('float32' or 'int16')
% marker_lbl_loc     c3d marker label field ('LABELS' or 'DESCRIPTIONS')
% analog_lbl_loc     c3d analog label field ('LABELS' or 'DESCRIPTIONS')
%
% Defaults:
% conv_type = 'single'
% transform = [-1 0 0; 0 0 -1; 0 1 0];
% conv_lbls = 1
% upsample_markers = 1
% c3d_analog_format = 'float32'
% marker_lbl_loc = 'DESCRIPTIONS'
% analog_lbl_loc = 'DESCRIPTIONS'
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults
if (nargin<1)
    conv_type = 'batch_r';
end
if (nargin<2)
    upsample_markers = 1;
end
if (nargin<3)
    conv_lbls = 1;
end

% parameters
transform = [-1 0 0; 0 0 -1; 0 1 0];
c3d_analog_format = 'float32';
marker_lbl_loc = 'DESCRIPTIONS';
analog_lbl_loc = 'DESCRIPTIONS';

fprintf('\n');

%% check if inputs and parameters are valid
err = '';
if (~(strcmpi(conv_type,'single')||strcmpi(conv_type,'batch')||...
        strcmpi(conv_type,'batch_r')))
    err = 'conv_type must be ''single'', ''batch'' or ''batch_r''\n';
end
if (~isscalar(conv_lbls)||(conv_lbls~=0)&&(conv_lbls~=1))
    err = [err,'conv_lbls must be 1 or 0\n'];
end
if (~isscalar(upsample_markers)||
(upsample_markers~=0)&&(upsample_markers~=1))
    err = [err,'upsample_markers must be 1 or 0\n'];
end
```

```
if (~(strcmpi(c3d_analog_format,'int16')||...
        strcmpi(c3d_analog_format,'float32')))
    err = [err,'c3d_analog_format must be ''int16'' or ''float32''\n'];
end
if (~(strcmpi(marker_lbl_loc,'LABELS')||...
        strcmpi(marker_lbl_loc,'DESCRIPTIONS')))
    err = [err,'marker_lbl_loc must be ''LABELS'' or ''DESCRIPTIONS''\n'];
end
if (~(strcmpi(analog_lbl_loc,'LABELS')||...
        strcmpi(analog_lbl_loc,'DESCRIPTIONS')))
    err = [err,'analog_lbl_loc must be ''LABELS'' or ''DESCRIPTIONS''\n'];
end
if (~strcmp(err,''))
    error('C3D_2_OSim:params',err);
end

%% choose file(s) and load data
lookup_table = []; osim_markers = []; osim_gencoords = []; FPinfo = [];

if strcmpi(conv_type,'single')
    [files,file_dir] = uigetfile('*.c3d', 'open C3D file');
    if (~file_dir)
        return
    end
    files = struct('name',files,'isdir',false);
elseif (strcmpi(conv_type,'batch')||strcmpi(conv_type,'batch_r'))
    file_dir = uigetdir(pwd,'Choose C3D data folder.');
    if (~file_dir)
        return
    end
    files = dir(file_dir);
end

if (conv_lbls==1)
    % load dictionary
    [dict_name,dict_dir] = uigetfile('*.mat',...
        'Open OpenSim model dictionary. Cancel to skip.');
    if (dict_name~=0)
        load([dict_dir dict_name]);
        [lookup_name,lookup_dir] = uigetfile('*.mat',...
            'Open label lookup table. Cancel if none exist.');
        if (lookup_name~=0)
            load([lookup_dir lookup_name]);
        end
    else
        warning('C3D_2_OSim:missingInfo',['Label conversions and some',...
          ' mot files can not be completed without a dictionary loaded.']);
    end

    % load forceplate info
    [temp,temp2] = uigetfile('*.mat',...
        'Open forceplate mat file. Cancel if no forceplates used.');
```

```
    if (temp~=0)
        load([temp2,temp]);
    end
    if (isempty(FPinfo))
        warning('C3D_2_OSim:missingInfo',...
            'No forceplate info loaded, GRF data can not be processed.')
    end
end

%% convert file(s)
settings = struct('osim_markers',{osim_markers},'osim_gencoords',...
    {osim_gencoords},'conv_type',conv_type,'transform',transform,...
    'conv_lbls',conv_lbls,'upsample_markers',upsample_markers,...
    'c3d_analog_format',c3d_analog_format,'marker_lbl_loc',...
    marker_lbl_loc,'analog_lbl_loc',analog_lbl_loc,'FPinfo',FPinfo);
if (sum(files(1).name~=0))
    C3D_2_OSim_helper(files, file_dir, settings, lookup_table);
end

return
```

E.2.2 C3D_2_OSim_helper.m

```
function lookup_table = C3D_2_OSim_helper(files, file_dir, s, lookup_table)
% C3D_2_OSim_helper: Recursive helper function that should only be
%                    called from C3D_2_OSim.m. Gives the ability to
%                    descend recursively into data directories.
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%%
files(strcmp(files(1).name,'.'))=[];
files(strcmp(files(1).name,'..'))=[];
i_lim = length(files);
for i=1:i_lim
    if ((files(i).isdir)&&(strcmp(s.conv_type,'batch_r')))
        lookup_table = C3D_2_OSim_helper(dir(fullfile(file_dir,...
          files(i).name)),fullfile(file_dir,files(i).name),s,lookup_table);
        continue
    end

    file_name = files(i).name;
    if (strcmpi(file_name(max(strfind(file_name,'.'))...
            :length(file_name)),'.c3d'))
        [Markers,VideoFrameRate,AnalogSignals,AnalogFrameRate,Event,...
         ParameterGroup] = readC3D(file_name,file_dir,s.c3d_analog_format);

        [AnalogSignals,q] = processAnalog(AnalogSignals,...
            ParameterGroup,s.FPinfo,s.transform,s.analog_lbl_loc);
```

```
marker_params = processMotion(Markers,VideoFrameRate,...
    ParameterGroup,s.transform,s.marker_lbl_loc);

if (~isempty(AnalogSignals))
    if isfield(marker_params,'data_rate')
        marker_params = upsampleMarkers(marker_params,...
            q.analog_rate/marker_params.data_rate);
    end
   writeANC(AnalogSignals,q.labels,q.num_channels,q.analog_rate,...
        q.scale,[file_name(1:(max(strfind(file_name,'.'))-1)),...
        '.anc'],file_dir);
else
    fprintf('Could not create %s.\n',strrep([file_name(1:(max...
        (strfind(file_name,'.'))-1)),'.anc'],'\','\\'));
end

if (~isempty(marker_params.data))
    if (~isempty(s.osim_markers))
     [marker_params.labels(2:length(marker_params.labels)),temp]...
            = renameLabels(marker_params.labels(2:length...
            (marker_params.labels)),s.osim_markers,lookup_table);
        if (~strcmp(temp,''))
            load(temp);
        end
     writeTRC(marker_params,[file_name(1:(max(strfind(file_name,...
            '.'))-1)),'.trc'],file_dir);
    end
else
    fprintf('Could not create %s.\n',strrep([file_name(1:(max...
        (strfind(file_name,'.'))-1)),'.trc'],'\','\\'));
end

fprintf('Attempting to create .mot files...\n');
if (~isempty(marker_params.data)&&(~isempty(s.osim_gencoords)))
    M = write_static_motion(s.osim_gencoords,marker_params,...
        [file_name(1:(max(strfind(file_name,'.'))-1)),...
        '_zeros.mot'],file_dir);
   fprintf('Successfully created %s.\n',strrep([file_name(1:(max...
        (strfind(file_name,'.'))-1)),'_zeros.mot'],'\','\\'));
end
mot_success = 0; grf_success = 0;
try
 F = process_grf(q,[file_name(1:(max(strfind(file_name,'.'))-1))...
        ,'_grf.mot'],file_dir);
    grf_success = 1;
   fprintf('Successfully created %s.\n',strrep([file_name(1:(max...
        (strfind(file_name,'.'))-1)),'_grf.mot'],'\','\\'));
catch
    fprintf('Could not create %s.\n',strrep([file_name(1:(max...
        (strfind(file_name,'.'))-1)),'_grf.mot'],'\','\\'));
    if (~isempty(q))
        fprintf(['There might be a problem with the ',...
```

```
                'forceplate calibration information.\n']);
            end
        end
        if (grf_success)
            try
              put_forces_in_mot(F,M,[file_name(1:(max(strfind(file_name,...
                    '.'))-1)),'.mot'],file_dir);
               fprintf('Successfully created %s.\n',strrep([file_name(1:...
                    (max(strfind(file_name,'.'))-1)),'.mot'],'\','\\'));
                mot_success = 1;
            catch
                mot_success = 0;
            end
        end
        if (mot_success==0)
            fprintf('Could not create %s.\n',strrep([file_name(1:...
                (max(strfind(file_name,'.'))-1)),'.mot'],'\','\\'));
        end
        fprintf('\n\n');
    end
end

return
```

E.3 Reading and Processing Data

E.3.1 readC3D.m

```
function [Markers,VideoFrameRate,AnalogSignals,AnalogFrameRate,Event,...
    ParameterGroup,CameraInfo,ResidualError] = ...
    readC3D(file_name,file_dir,analog_data_format)
% readC3D:  Reads 3D coordinate/analog data from a C3D file.
%           Modified to work with the C3D_2_OSim package.
%
% Input:
% file_name             name of c3d file
% file_dir              (OPTIONAL) directory where file is located
% analog_data_format    (OPTIONAL) format of analog data in c3d file
%
% Output:
% Markers           3D-marker data [Nmarkers x NvideoFrames x Ndim(=3)]
% VideoFrameRate    Frames/sec
% AnalogSignals     Analog signals [Nsignals x NanalogSamples ]
% AnalogFrameRate   Samples/sec
% Event             Event(Nevents).time ..value  ..name
% ParameterGroup    ParameterGroup(Ngroups).Parameters(Nparameters).data
% CameraInfo        MarkerRelated CameraInfo [Nmarkers x NvideoFrames]
```

```
% ResidualError     MarkerRelated ErrorInfo  [Nmarkers x NvideoFrames]
%
% Defaults:
% file_dir = present working directory (pwd)
% analog_data_format = 'float32'
%
% AUTHOR(S) AND VERSION-HISTORY
% V. 1.0 Creation (Alan Morris, Toronto, Oct 1998) [originally "getc3d.m"]
% V. 2.0 Revision (Jaap Harlaar, Amsterdam, April 2002)
% V. 2.1 Revision (John Kelly, North Carolina State University, March 2008)


%% load defaults if necessary
Markers = []; VideoFrameRate = []; AnalogSignals = []; AnalogFrameRate =[];
Event = []; ParameterGroup = []; CameraInfo = []; ResidualError = [];
if (nargin<2)
    file_dir = [pwd,'\'];
end
if (nargin<3)
    analog_data_format = 'float32';
end
if (nargin==0)
    warning('readC3D:args',...
        'Not enough input arguments. %s could not be read.',...
        strrep(file_name,'\','\\'));
    return
end

%% check if analog_data_format is valid
if (~(strcmpi(analog_data_format,'int16')||...
        strcmpi(analog_data_format,'float32')))
    warning('readC3D:format',...
        'analog_data_format must be ''int16'' or ''float32''.');
    fprintf('\nCan not read %s.\n',strrep(file_name,'\','\\'));
    return
end

%%
% ################################################
% ##                                            ##
% ##    open the file                           ##
% ##                                            ##
% ################################################

fid=fopen(fullfile(file_dir,file_name),'r','n'); % native format (PC-intel)

if fid==-1,
    warning('readC3D:file','%s could not be opened.',file_name);
    return
end

% Reading record number of parameter section
```

```
NrecordFirstParameterblock=fread(fid,1,'int8');
key=fread(fid,1,'int8');                              % key = 80;

if key~=80,
    warning('readC3D:format',...
        '%s does not comply to the C3D format.',file_name);
    fclose(fid);
    return
end

% jump to processortype - field
fseek(fid,512*(NrecordFirstParameterblock-1)+3,'bof');
% proctype: 1(INTEL-PC); 2(DEC-VAX); 3(MIPS-SUN/SGI)
proctype=fread(fid,1,'int8')-83;

if proctype==2,
    fclose(fid);
    % DEC VAX D floating point and VAX ordering
    fid=fopen(fullfile(file_dir,file_name),'r','d');
end

%%
% ################################################
% ##                                            ##
% ##    read header                             ##
% ##                                            ##
% ################################################

fseek(fid,2,'bof');

Nmarkers=fread(fid,1,'int16');                    %number of markers
%number of analog channels x #analog frames per video frame
NanalogSamplesPerVideoFrame=fread(fid,1,'int16');
StartFrame=fread(fid,1,'int16');                  %# of first video frame

EndFrame=fread(fid,1,'int16');                    %# of last video frame

fseek(fid,2,'cof');     % skip maximum interpolation gap allowed (in frame)

%floating-point scale factor to convert 3D-integers to ref system units
Scale=fread(fid,1,'float32');

%starting record number for 3D point and analog data
NrecordDataBlock=fread(fid,1,'int16');

NanalogFramesPerVideoFrame=fread(fid,1,'int16');
if NanalogFramesPerVideoFrame > 0
   NanalogChannels=NanalogSamplesPerVideoFrame/NanalogFramesPerVideoFrame;
else
    NanalogChannels=0;
end
```

```
VideoFrameRate=fread(fid,1,'float32');
AnalogFrameRate=VideoFrameRate*NanalogFramesPerVideoFrame;

%%
% ################################################
% ##                                            ##
% ##    read events                             ##
% ##                                            ##
% ################################################

fseek(fid,298,'bof');
EventIndicator=fread(fid,1,'int16');
if EventIndicator==12345,
    Nevents=fread(fid,1,'int16');
    fseek(fid,2,'cof'); % skip one position/2 bytes
    if Nevents>0,
        clear Event
        Event(Nevents) = struct('time',[],'value',[],'name',[]);
        for i=1:Nevents,
            Event(i).time=fread(fid,1,'float');
        end
        fseek(fid,188*2,'bof');
        for i=1:Nevents,
            Event(i).value=fread(fid,1,'int8');
        end
         fseek(fid,198*2,'bof');
        for i=1:Nevents,
            Event(i).name=cellstr(fread(fid,4,'*char')');
        end
    end
end

%%
% ################################################
% ##                                            ##
% ##    read 1st parameter block                ##
% ##                                            ##
% ################################################

fseek(fid,512*(NrecordFirstParameterblock-1),'bof');

fseek(fid,2,'cof');                     % dat1 and key aren't needed, skip
NparameterRecords=fread(fid,1,'int8');
fseek(fid,1,'cof');                     % proctype isn't needed, skip

clear ParameterGroup
ParameterGroup(NparameterRecords) = ...
    struct('name',[],'description',[],'Parameter',[]);
ParameterNumberIndex = zeros(1,NparameterRecords);
NparameterGroups = 0;

Ncharacters=fread(fid,1,'int8');    % characters in group/parameter name
```

```
GroupNumber=fread(fid,1,'int8');    % id number -ve=group / +ve=parameter

% The end of the parameter record is indicated by <0 characters for
% group/parameter name
while Ncharacters > 0

    if GroupNumber<0 % Group data
        GroupNumber=abs(GroupNumber);
        GroupName=fread(fid,[1,Ncharacters],'*char');
        ParameterGroup(GroupNumber).name=cellstr(GroupName);    %group name
        offset=fread(fid,1,'int16');                %offset in bytes
        deschars=fread(fid,1,'int8');               %description characters
        GroupDescription=fread(fid,[1,deschars],'*char');
        %group description
        ParameterGroup(GroupNumber).description=cellstr(GroupDescription);

        ParameterNumberIndex(GroupNumber)=0;
        NparameterGroups = max(NparameterGroups,GroupNumber);
        fseek(fid,offset-3-deschars,'cof');

    else % parameter data
        clear dimension;
     ParameterNumberIndex(GroupNumber)=ParameterNumberIndex(GroupNumber)+1;
        % index all parameters within a group
        ParameterNumber=ParameterNumberIndex(GroupNumber);

     ParameterName=fread(fid,[1,Ncharacters],'*char');  % name of parameter

        % read parameter name
        if size(ParameterName)>0
            ParameterGroup(GroupNumber).Parameter(ParameterNumber).name=...
                cellstr(ParameterName); %save parameter name
        end

        % read offset
        offset=fread(fid,1,'int16');    %offset of parameters in bytes
        filepos=ftell(fid);             %present file position
      nextrec=filepos+offset(1)-2;    %position of beginning of next record

        % read type
        % type of data: -1=char/1=byte/2=integer*2/4=real*4
        type=fread(fid,1,'int8');
      ParameterGroup(GroupNumber).Parameter(ParameterNumber).datatype=type;

        % read number of dimensions
        dimnum=fread(fid,1,'int8');
        if dimnum==0
            datalength=abs(type);                   %length of data record
        else
            mult=1;
            dimension = zeros(1,dimnum);
            for j=1:dimnum
```

```
        dimension(j)=fread(fid,1,'int8');
        mult=mult*dimension(j);
        ParameterGroup(GroupNumber).Parameter(ParameterNumber)...
            .dim(j)=dimension(j);  %save parameter dimension data
    end
    %length of data record for multi-dimensional array
    datalength=abs(type)*mult;
end


if type==-1 %datatype=='char'

    wordlength=dimension(1);    %length of character word
    if ((dimnum==2) && (datalength>0))
        for j=1:dimension(2)
            %character word data record for 2-D array
            data=fread(fid,[1,wordlength],'*char');
          ParameterGroup(GroupNumber).Parameter(ParameterNumber)...
                .data(j)=cellstr(data);
        end

    elseif ((dimnum==1) && (datalength>0))
        %numerical data record of 1-D array
        data=fread(fid,[1,wordlength],'*char');
     ParameterGroup(GroupNumber).Parameter(ParameterNumber).data...
            =cellstr(data);
    end

elseif type==1    %1-byte for boolean

    Nparameters=datalength/abs(type);
    data=fread(fid,Nparameters,'int8');
  ParameterGroup(GroupNumber).Parameter(ParameterNumber).data=data;

elseif ((type==2) && (datalength>0))            %integer

    Nparameters=datalength/abs(type);
    data=fread(fid,Nparameters,'int16');
    if dimnum>1
     ParameterGroup(GroupNumber).Parameter(ParameterNumber).data...
            =reshape(data,dimension);
    else
     ParameterGroup(GroupNumber).Parameter(ParameterNumber).data...
            =data;
    end

elseif ((type==4) && (datalength>0))

    Nparameters=datalength/abs(type);
    data=fread(fid,Nparameters,'float');
    if dimnum>1
     ParameterGroup(GroupNumber).Parameter(ParameterNumber).data...
```

```
                    =reshape(data,dimension);
                else
                 ParameterGroup(GroupNumber).Parameter(ParameterNumber).data...
                    =data;
                end
            else
                % error
            end

            deschars=fread(fid,1,'int8');               %description characters
            if deschars>0
                description=fread(fid,[1,deschars],'*char');
                ParameterGroup(GroupNumber).Parameter(ParameterNumber)...
                    .description=cellstr(description);
            end
            %moving ahead to next record
            fseek(fid,nextrec,'bof');
        end

       % check group/parameter chars & id number to see if more records present
     Ncharacters=fread(fid,1,'int8'); % characters in next group/parameter name
        GroupNumber=fread(fid,1,'int8'); % id number -ve=group / +ve=parameter
    end

    ParameterGroup = ParameterGroup(1:NparameterGroups);

    %%
    % ################################################
    % ##                                            ##
    % ##    read data block                         ##
    % ##                                            ##
    % ################################################
    %  Get the coordinate and analog data

    fseek(fid,(NrecordDataBlock-1)*512,'bof');

    NvideoFrames=EndFrame - StartFrame + 1;

    Markers = zeros(NvideoFrames,Nmarkers,3);
    CameraInfo = zeros(NvideoFrames,Nmarkers);
    ResidualError = zeros(NvideoFrames,Nmarkers);
    AnalogSignals =
    zeros(NanalogFramesPerVideoFrame*NvideoFrames,NanalogChannels);

    if Scale < 0
        for i=1:NvideoFrames
            for j=1:Nmarkers
                Markers(i,j,1:3)=fread(fid,3,'float32')';
                a=fix(fread(fid,1,'float32'));
                highbyte=fix(a/256);
                lowbyte=a-highbyte*256;
                CameraInfo(i,j)=highbyte;
```

```
            ResidualError(i,j)=lowbyte*abs(Scale);
        end
        for j=1:NanalogFramesPerVideoFrame,
            AnalogSignals(j+NanalogFramesPerVideoFrame*(i-1),...
                1:NanalogChannels)=...
                fread(fid,NanalogChannels,analog_data_format)';
        end
    end
else
    for i=1:NvideoFrames
        for j=1:Nmarkers
            try
                Markers(i,j,1:3)=fread(fid,3,'int16')'.*Scale;
                ResidualError(i,j)=fread(fid,1,'int8');
                CameraInfo(i,j)=fread(fid,1,'int8');
            catch
               warning('readC3D:format',['Either %s is corrupt or the ',...
                'incorrect value of analog_data_format is being used.'],...
                   file_name);
                fclose(fid);
                AnalogSignals = [];
                Markers = [];
                return
            end
        end
        for j=1:NanalogFramesPerVideoFrame,
            AnalogSignals(j+NanalogFramesPerVideoFrame*(i-1),...
                1:NanalogChannels)=...
                fread(fid,NanalogChannels,analog_data_format)';
        end
    end
end

fclose(fid);

fprintf('\nSuccessfully read %s.\n',strrep(file_name,'\','\\'));

return
```

E.3.2 processAnalog.m

```
function [AnalogSignals,q]=...
    processAnalog(AnalogSignals,ParameterGroup,FPinfo,transform,label_loc)
% processAnalog:    Processes analog data collected from a C3D file.
%                   This function is called from C3D_2_OSim.
%                   It has been modified to work with Stanford's functions
%                   for processing ground reaction forces.
%
% Input:
% AnalogSignals     analog data
% ParameterGroup    various parameter information found in C3D files
% FPinfo            (OPTIONAL) forceplate calibration information
% transform         (OPTIONAL) 3x3 matrix for transforming to OpenSim frame
% label_loc         (OPTIONAL) C3D field where analog labels are located
%
% Output:
% AnalogSignals     analog data
% q                 various parameters associated with the analog data
%
% Defaults:
% transform = eye(3)
% data_format = 'float32'
% label_loc = 'DESCRIPTIONS'
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults if necessary
if (nargin<3)
    FPinfo = [];
end
if (nargin<4)
    transform = eye(3);
end
if (nargin<5)
    label_loc = 'DESCRIPTIONS';
end
if (nargin<2)
    warning('processAnalog:args',...
        'Not enough arguments. Could not process analog data.');
    AnalogSignals = [];
    q = [];
    return
end

%% check params
% check if analog data exists
if (isempty(AnalogSignals))
```

```
    warning('processAnalog:data',...
        'Analog data could not be found. Could not process analog data.');
    q = [];
    return
end

% check if inputs are valid
err = '';
if ((size(transform,1)~=3)&&(size(transform,2)~=3))
    err = sprintf('transform must be a 3x3 matrix.\n');
end
if (~(strcmpi(label_loc,'LABELS')||strcmpi(label_loc,'DESCRIPTIONS')))
err = sprintf('%slabel_loc must be ''LABELS'' or ''DESCRIPTIONS''.\n',err);
end
if (~strcmp(err,''))
    warning('processAnalog:args','%sCould not process analog data.',err);
    AnalogSignals = [];
    q = [];
    return
end

%% extract info from c3d file and fill in missing data
mask = uint8(0);
temp = mask;
i_lim = length(ParameterGroup);
for i=1:i_lim
    if ((~isempty(ParameterGroup(i).name))&&...
            strcmpi(ParameterGroup(i).name{1},'ANALOG'))
    j_lim = length(ParameterGroup(i).Parameter);
        for j=1:j_lim
            switch (upper(ParameterGroup(i).Parameter(j).name{1}))
                case upper(label_loc)
                    q.labels = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(1));
                case 'UNITS'
                    q.units = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(2));
                case 'USED'
                    q.num_channels = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(4));
                case 'GEN_SCALE'
                    gen_scale = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(8));
                case 'SCALE'
                    q.scale = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(16));
                case 'OFFSET'
```

```
                    offset = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(32));
                case 'RATE'
                    q.analog_rate = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(64));
            end
            if (temp>mask)
             warning('processAnalog:data',['Analog parameter data has ',...
                 'ambiguous information. Could not process analog data.']);
                AnalogSignals = [];
                q = [];
                return;
            end
        end
    end
end
if (mask~=uint8(127))
    if (bitand(mask,uint8(64))==0)
        warning('processAnalog:data',['Analog parameter data is ',...
            'missing ''RATE'' field. Could not process analog data.']);
        AnalogSignals = [];
        q = [];
        return;
    end
    if (bitand(mask,uint8(4))==0)
        warning('processAnalog:data',['''USED'' field was missing. ',...
            'Number of channels will be manually measured.']);
        q.num_channels = size(AnalogSignals,2);
    end
    if (bitand(mask,uint8(1))==0)
        warning('processAnalog:data',...
           'Label info was missing. Generic channel labels will be used.');
        fprintf(['\tTry changing ''label_loc'' if ',...
            'label information should be present.\n']);
        fprintf(['Channel types must be manually assigned. ',...
            'Enter 0 for EMG and 1 for forceplate.\n']);
        i_lim = num_channels/3;
        q.labels = cell(1,i_lim);
        num_fp = 0;
        num_emg = 0;
        i = 1;
        while (i<=i_lim)
            if ((i_lim-i)>4)
                temp = input(sprintf('channel%d >> ',i),'s');
                if (temp==1)
                    num_fp = num_fp+1;
                    q.labels{i} = sprintf('Fx%d',num_fp);
                    fprintf('\tchannel%d assigned as Fx%d\n',i,num_fp);
                    q.labels{i+1} = sprintf('Fy%d',num_fp);
                    fprintf('\tchannel%d assigned as Fy%d\n',i+1,num_fp);
```

```
                    q.labels{i+2} = sprintf('Fz%d',num_fp);
                    fprintf('\tchannel%d assigned as Fz%d\n',i+2,num_fp);
                    q.labels{i+3} = sprintf('Mx%d',num_fp);
                    fprintf('\tchannel%d assigned as Mx%d\n',i+3,num_fp);
                    q.labels{i+4} = sprintf('My%d',num_fp);
                    fprintf('\tchannel%d assigned as My%d\n',i+4,num_fp);
                    q.labels{i+5} = sprintf('Mz%d',num_fp);
                    fprintf('\tchannel%d assigned as Mz%d\n',i+5,num_fp);
                    i = i+6;
                    continue;
                elseif (temp~=0)
                    fprintf('Invalid input\n');
                    continue;
                end
                num_emg = num_emg+1;
                q.labels{i} = sprintf('emg%d',num_emg);
                fprintf('\tchannel%d assigned as emg%d\n',i,num_emg);
                i = i+1;
            end
        end
    end
    if (bitand(mask,uint8(2))==0)
        warning('processAnalog:data',['''UNITS'' field was missing.',...
            ' ''volts'' will be assumed as the units of all channels.']);
        units = cell(1,q.num_channels);
        for i=1:num_channels
            units{i} = 'volts';
        end
    end
    if (bitand(mask,uint8(8))==0)
        warning('processAnalog:data',['''GEN_SCALE'' field was ',...
            'missing. gen_scale=1 will be assumed.']);
        gen_scale = 1;
    end
    if (bitand(mask,uint8(16))==0)
        warning('processAnalog:data',['''SCALE'' field was missing.',...
            ' scale=1 will be assumed.']);
        q.scale = ones(q.num_channels,1);
    end
    if (bitand(mask,uint8(16))==0)
        warning('processAnalog:data',['''OFFSET'' field was ',...
            'missing. offset=0 will be assumed.']);
        offset = zeros(q.num_channels,1);
    end
end

%% update and process data
AnalogSignals = (AnalogSignals-(offset*ones(1,size(AnalogSignals,1)))');
q.scale = q.scale'*gen_scale;
q.time = ((1/q.analog_rate)*(0:(size(AnalogSignals,1)-1)))';
q.rates = q.analog_rate*ones(1,q.num_channels);
```

```
% separate emg and forceplate data
num_fp = 0;
num_emg = 0;
q.emg.data = [];
i_lim = q.num_channels;
i = 1;
while (i<=i_lim)
    if ((i_lim-i)>4)
        temp = cell(1,6);
        temp{1} = strrep(strrep(lower(q.labels{i}),'x','_'),'f','_');
        temp{2} = strrep(strrep(lower(q.labels{i+1}),'y','_'),'f','_');
        temp{3} = strrep(strrep(lower(q.labels{i+2}),'z','_'),'f','_');
        temp{4} = strrep(strrep(lower(q.labels{i+3}),'x','_'),'m','_');
        temp{5} = strrep(strrep(lower(q.labels{i+4}),'y','_'),'m','_');
        temp{6} = strrep(strrep(lower(q.labels{i+5}),'z','_'),'m','_');
        if (min(strcmp(temp{1},temp))==1)
            num_fp = num_fp+1;
            q.data(num_fp).digitalF = ...
                AnalogSignals(:,i:(i+2))*diag(q.scale(i:(i+2)));
            q.data(num_fp).digitalM = ...
                AnalogSignals(:,(i+3):(i+5))*diag(q.scale((i+3):(i+5)));
            i = i+6;
            continue;
        end
    end
    num_emg = num_emg+1;
    q.emg.labels{num_emg} = q.labels{i};
    q.emg.data = [q.emg.data (AnalogSignals(:,i)*q.scale(i))];
    i = i+1;
end

%% process forceplate data
if (isempty(FPinfo))
    warning('processAnalog:data','Forceplate data could not be processed');
    return
end

% code block pulled from Stanford's read_anc.m and modified
for i=1:num_fp

    % Convert 'reaction' forces and moments to 'action' forces and moments
    temp = -1*[q.data(i).digitalF, q.data(i).digitalM];

    % Convert units (voltages to Newtons, Newtons-m)
    temp = (1 / FPinfo(i).gain) * temp * FPinfo(i).calMatrix;

    % extract F and M
    q.data(i).F = temp(:,1:3);
    q.data(i).M = temp(:,4:6);

    % Transform FP -> Lab -> Model
    q.data(i).F = (transform * FPinfo(i).orientationMatrix *
```

```
q.data(i).F')';
   q.data(i).M = (transform * FPinfo(i).orientationMatrix * q.data(i).M')';

    % used later for translation
    q.data(i).FPorigin_model = transform*FPinfo(i).originTranslation';

end

return
```

E.3.3 processMotion.m

```
function param = ...
   processMotion(Markers,VideoFrameRate,ParameterGroup,transform,label_loc)
% processMotion:    Processes mocap data collected from a C3D file.
%                   This function is called from C3D_2_OSim.
%                   It has been modified to work with Stanford's functions
%                   for processing ground reaction forces.
%
% Input:
% Markers           marker coordinate data
% VideoFrameRate    frame rate of video at time of capture
% ParameterGroup    various parameter information found in C3D files
% transform         (OPTIONAL) 3x3 transform matrix for converting data
% label_loc         (OPTIONAL) C3D field where marker labels are located
%
% Output:
% param             processed marker data and parameter information
%
% Defaults:
% transform = eye(3)
% label_loc = 'DESCRIPTIONS'
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults if necessary
if (nargin<4)
    transform = eye(3);
end
if (nargin<5)
    label_loc = 'DESCRIPTIONS';
end

param.data = Markers;

%% check params
% check if marker data exists
if (isempty(param.data))
```

```
    warning('processMotion:data',...
        'Marker data could not be found. Could not process marker data.');
    return
end

% check if label_loc, transform are valid
err = '\n';
if ((size(transform,1)~=3)&&(size(transform,2)~=3))
    err = 'transform must be a 3x3 matrix.\n';
end
if (~(strcmpi(label_loc,'LABELS')||strcmpi(label_loc,'DESCRIPTIONS')))
   err = [err,'label_loc must be ''LABELS'' or ''DESCRIPTIONS''.\n'];
end
if (~strcmp(err,'\n'))
    fprintf('Could not process marker data.\n');
    param.data = [];
    return;
end

%% extract info from ParameterGroup and fill in missing data
param.video_rate = VideoFrameRate;
mask = uint8(0);
temp = mask;
i_lim = length(ParameterGroup);
for i=1:i_lim
    if ((~isempty(ParameterGroup(i).name))&&...
            strcmpi(ParameterGroup(i).name{1},'POINT'))
        j_lim = length(ParameterGroup(i).Parameter);
        for j=1:j_lim
            switch (upper(ParameterGroup(i).Parameter(j).name{1}))
                case upper(label_loc)
                    param.labels = ParameterGroup(i).Parameter(j).data;
                    param.labels = [{'Time'} param.labels];
                    temp = mask;
                    mask = bitxor(mask,uint8(1));
                case 'USED'
                    param.nummarkers = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(2));
                case 'FRAMES'
                   param.num_frames = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(4));
                case 'SCALE'
                    scale = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(8));
                case 'UNITS'
                    param.units = ParameterGroup(i).Parameter(j).data{1};
                    temp = mask;
                    mask = bitxor(mask,uint8(16));
                case 'RATE'
```

```
                    param.data_rate = ParameterGroup(i).Parameter(j).data;
                    temp = mask;
                    mask = bitxor(mask,uint8(32));
                end
                if (temp>mask)
                    fprintf(['\nMarker parameter data has ambiguous ',...
                        'information. Could not process marker data.\n']);
                    return;
                end
            end
        end
    end
    if (mask~=uint8(63))
        if (bitand(mask,uint8(32))==0)
            warning('processMotion:data',['Marker parameter data is ',...
                'missing ''RATE'' field. Could not process marker data.']);
            return;
        end
        if (bitand(mask,uint8(2))==0)
            warning('processMotion:data',['''USED'' field was missing. ',...
                'Number of markers will be manually measured.']);
            param.nummarkers = size(param.data,2);
        end
        if (bitand(mask,uint8(1))==0)
            warning('processMotion:data',...
                'Label info was missing. Generic marker labels will be used.');
            fprintf(['\tTry changing ''label_loc'' value if label ',...
                'information should be present.\n']);
            i_lim = param.nummarkers/3;
            param.labels = cell(1,i_lim);
            for i=1:i_lim
                param.labels{i} = sprintf('marker%d',i);
            end
        end
        if (bitand(mask,uint8(4))==0)
            warning('processMotion:data',['''FRAMES'' field was ',...
                'missing. Number of frames will be measured.']);
            param.num_frames = size(param.data,1);
        end
        if (bitand(mask,uint8(8))==0)
            warning('processMotion:data',['''SCALE'' field was ',...
                'missing. param.scale=1 will be assumed.']);
            scale = 1;
        end
        if (bitand(mask,uint8(16))==0)
            warning('processMotion:data',['''UNITS'' field was missing.',...
                ' ''mm'' will be assumed as the param.units.']);
            param.units = 'mm';
        end
    end

    %% transforma marker coords
```

```
for i=1:param.nummarkers
    for j=1:param.num_frames
        param.data(j,i,1:3) = transform*reshape(param.data(j,i,1:3),3,1);
    end
end
param.data = param.data*scale;

return
```

E.3.4 upsampleMarkers.m

```
function markers = upsampleMarkers(markers, ratio)
% upsample: Upsamples marker data used by C3D_2_OSim to match analog rate.
%
% Input:
% markers       contains marker information
% ratio         factor to upsample by
%
% Output:
% markers       the upsampled marker information
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008

if ((nargin<2)||(~exist('markers','var'))||(~isfield(markers,'data'))...
     ||(~isfield(markers,'data_rate'))||(~isfield(markers,'num_frames'))...
        ||(~isscalar(ratio))||(~isnumeric(ratio)))
    warning('upsampleMarkers:args',...
        'Input arguments are incorrect. Data could not be upsampled.');
    return
end

m_size = size(markers.data);

data_ind = round(1:ratio:(ratio*m_size(1)));
gap_ind = setdiff(1:round(ratio*m_size(1)),data_ind);

temp = zeros(round(m_size(1)*ratio),m_size(2),m_size(3));
temp(data_ind,:,:) = markers.data;
markers.data = temp;

for i=1:m_size(2)
    for j=1:3
        markers.data(gap_ind,i,j) = ...
           interp1(data_ind, markers.data(data_ind,i,j), gap_ind, 'cubic');
    end
end

markers.data_rate = round(markers.data_rate*ratio);
markers.num_frames = round(markers.num_frames*ratio);
```

```
return
```

E.3.5 renameLabels.m

```
function [labels,table_path] = renameLabels(labels,dict,lookup_table)
% renameLabels: Uses lookup tables to rename labels.
%               This function is called from C3D_2_OSim.
%
% Input:
% labels            labels to be renamed
% dic               dictionary containing acceptable names
% lookup_table      (OPTIONAL) existing lookup table
%
% Output:
% labels            renamed labels
% table_path        path of lookup table if new one created, '' otherwise
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults
choice = -1;
if ((nargin<3)||(isempty(lookup_table)))
    lookup_table = [];
end
table_path = '';

%% attempt to convert labels
i_lim = length(labels);
i = 1;
while (i<=i_lim)
   % label found in dictionary
   if (max(strcmp(labels{i},dict))==1)
       i = i+1;
       continue;
   end

   if (~isempty(lookup_table))
       % label found in lookup table
       ind = strcmp(labels{i},lookup_table(:,1));
       if (max(ind)==1)
           if (~strcmp(lookup_table{ind,2},''))
               labels{i} = lookup_table{ind,2};
           end
           i = i+1;
           continue;
       end
```

```
    % label not found, choice already made
    lookup_length = size(lookup_table,1)+1;
    if (choice~=-1)
        lookup_table{lookup_length,1} = labels{i};
        while (max(strcmp(labels{i},dict))~=1)
            fprintf('\n%s could not be found in the dictionary.\n',...
                labels{i});
            fprintf(['Input the equivalent name or press enter ',...
                'to skip.\nEnter ''?'' to see a list of ',...
                'possible names.\n']);
            temp = input(sprintf('%s >> ',...
                lookup_table{lookup_length,1}),'s');
            while (strcmp(temp,'?'))
                j_lim = length(dict);
                for j=1:j_lim
                    fprintf('\t%s\n',dict{j});
                end
                temp = input(sprintf('\n%s >> ',...
                    lookup_table{lookup_length,1}),'s');
            end
            if (strcmp(temp,''))
                break
            end
            labels{i} = temp;
        end
        if (~strcmp(temp,''))
            lookup_table{lookup_length,2} = labels{i};
        else
            lookup_table{lookup_length,2} = '';
        end
        i = i+1;
        continue;
    end
end

%label not found, figure out what to do
while true
    fprintf(['\nNot all marker labels could be found in the ',...
        'dictionary or the lookup table.\n']);
    fprintf(['Enter 1 to create a new table, 2 to add to the ',...
        'current table,\n3 to convert without storing lookup ',...
        'information, or 4 to not convert labels.\n']);
    choice = str2double(input('>> ','s'));
    if ((choice==1)||(choice==2)||(choice==3)||(choice==4))
        if (choice==4)
            return
        end
        if ((choice==1)||(choice==2))
            [file,dir] = uiputfile('*.mat','Save lookup table as');
            table_path = fullfile(dir,file);
        end
        if (isempty(lookup_table))
```

```
                if (choice==2)
                    fprintf(['No current table exists. A new one ',...
                        'will be created.\n']);
                end
                lookup_table = cell(1,2);
            end

            break;
        else
            fprintf('Invalid input.\n');
        end
    end
end

%% save new lookup table if desired
if (isempty(lookup_table{1,1}))
    lookup_table(1,:)=[];
end
if ((choice~=3)&&(choice~=-1))
    save(table_path,'lookup_table');
end

return
```

E.4 Writing Output

E.4.1 writeANC.m

```
function writeANC(AnalogSignals,channel_labels,num_channels,data_rate,...
    scale,file_name,file_dir)
% writeANC: Creates an ANC file with data collected from a C3D file.
%           This function is called from C3D_2_OSim.
%
% Input:
% AnalogSignals     analog data
% channel_labels    analog channel headers
% num_channels      number of analog channels
% data_rate         analog data rate
% scale             analog scale factor extracted from c3d file
% file_name         name of file to be created
% file_dir          (OPTIONAL) directory where file is to be created
%
% Defaults:
% file_dir = present working directory (pwd)
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008
```

```
%% load defaults if necessary
if (nargin<7)
    file_dir = [pwd,'\'];
end
if (nargin<6)
    warning('writeANC:args',...
        '\nNot enough arguments. %s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

% check if analog data exists
if (isempty(AnalogSignals))
    warning('writeANC:data',...
        '\nAnalog data could not be found. %s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

%% hard-coded values
% these values can't be extracted from C3D files
% they are hard-coded here, change them if necessary to fit your system
gen_num = 1;
board_type = 'unknown';
polarity = 'Bipolar';
trial_name = file_name(1:(max(strfind(file_name,'.'))-1));
trial_num = 1;
bit_depth = 12;

range = scale*(2^bit_depth);

%% open file
fid = fopen(fullfile(file_dir,file_name),'Wt');

% check if file was created successfully
if fid==-1,
    warning('writeANC:file','\n%s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

%% write anc file
% write anc header
fprintf(fid,'File_Type:\t%s\tGeneration#:\t%d\n','Analog R/C
ASCII',gen_num);
fprintf(fid,'Board_Type:\t%s\tPolarity:\t%s\n',board_type,polarity);
fprintf(fid,['Trial_Name:\t%s\tTrial#:\t%d\tDuration(Sec.):\t',...
    '%1.6f\t#Channels:\t%d\n'],trial_name,trial_num,...
    (size(AnalogSignals,1)-1)/data_rate,num_channels);
fprintf(fid,'BitDepth:\t%d\tPreciseRate:\t%1.6f\n\n\n\n\n',...
```

```
    bit_depth,data_rate);

% write anc column labels
fprintf(fid,'Name\t');
fprintf(fid,'%s\t',channel_labels{:});
fprintf(fid,'\nRate\t');
fprintf(fid,'%d\t',data_rate*ones(1,num_channels));
fprintf(fid,'\nRange\t');
i_lim = length(range);
for i=1:i_lim
    if (abs(range(i)-round(range(i)))<0.001)
        fprintf(fid,'%d\t',round(range(i)));
    else
        fprintf(fid,'%1.2f\t',range(i));
    end
end
fprintf(fid,'\n');

% write analog data
i_lim = size(AnalogSignals,1);
time = (1/data_rate)*(0:(i_lim-1));
for i=1:i_lim
    fprintf(fid,'%1.6f\t',time(i));
    fprintf(fid,'%g\t',AnalogSignals(i,:));
    fprintf(fid,'\n');
end

fclose(fid);

fprintf('Successfully created %s.\n',strrep(file_name,'\','\\'));

return
```

E.4.2 writeTRC.m

```
function writeTRC(param,file_name,file_dir)
% writeTRC: Creates a TRC file with data collected from a C3D file.
%           This function is called from C3D_2_OSim.
%
% Input:
% param             various marker and parameter information
% file_name         name of file to be created
% file_dir          (OPTIONAL) directory where file is to be created
%
% Defaults:
% file_dir = present working directory (pwd)
% label_loc = 'DESCRIPTIONS'
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008
```

```
%% load defaults if necessary
if (nargin<3)
    file_dir = [pwd,'\'];
end
if (nargin<2)
    warning('writeTRC:args',...
        'Not enough arguments. %s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

%% check params
% check if marker data exists
if (isempty(param.data))
    warning('writeTRC:data',...
        'Marker data could not be found. %s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

fid = fopen(fullfile(file_dir,file_name),'Wt');

% check if file was created successfully
if fid==-1,
    warning('writeTRC:file','%s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

%% hard-coded values
% these values can't be extracted from C3D files
% they are hard-coded here, change them if necessary to fit your system
path_file_type = 4;
orig_data_rate = param.data_rate;
orig_start_frame = 1;
orig_num_frames = param.num_frames;

%% write trc file
% write trc header
fprintf(fid,'PathFileType\t%d\t(X/Y/Z)\t%s\n',path_file_type,file_name);
fprintf(fid,['DataRate\tCameraRate\tNumFrames\tNumMarkers\t',...
    'Units\tOrigDataRate\tOrigDataStartFrame\tOrigNumFrames\n']);
fprintf(fid,'%1.2f\t%1.2f\t%d\t%d\t%s\t%1.2f\t%d\t%d\n',...
    param.data_rate,param.video_rate,param.num_frames,param.nummarkers,...
    param.units,orig_data_rate,orig_start_frame,orig_num_frames);

% write trc column labels
coord_labels = '';
fprintf(fid,'Frame#\tTime\t');
fprintf(fid,'%s\t\t\t',param.labels{2:length(param.labels)});
```

```
i_lim = param.nummarkers;
for i=0:(i_lim-1)
    coord_labels = sprintf('%sX%d\tY%d\tZ%d\t',coord_labels,i,i,i);
end
fprintf(fid,'\n\t\t%s\n\n',coord_labels);

% write marker data
time = (1/param.data_rate)*(0:(param.num_frames-1));
for i=1:param.num_frames
   fprintf(fid,'%d\t%1.5f\t',i,time(i));
   for j=1:param.nummarkers
       fprintf(fid,'%1.5f\t%1.5f\t%1.5f\t',param.data(i,j,1),...
           param.data(i,j,2),param.data(i,j,3));
   end
   fprintf(fid,'\n');
end

fclose(fid);

fprintf('Successfully created %s.\n',strrep(file_name,'\','\\'));

return
```

E.5 Modified Stanford Code (GRF processing)

E.5.1 write_static_motion.m

```
function q =
write_static_motion(gencoords,marker_params,file_name,file_dir)
% created at Stanford University
% slightly modified to work better with C3D_2_OSim

tstart=0;
rate = marker_params.data_rate;
tend=marker_params.num_frames*(1/rate);

T=round((((rate*tstart):(rate*tend))/rate)*100000)/100000;

q.labels=[{'time'}, gencoords];
q.data=zeros(length(T),(length(gencoords)+1));
q.data(:,1)=T';

write_motionFile(q,file_name,file_dir);

return
```

E.5.2 write_motionFile.m

```
function write_motionFile(q, fname, fdir)
% originally created at Stanford University

fid = fopen(fullfile(fdir,fname), 'Wt');
if fid == -1
       error(['unable to open ', fname])
end

if length(q.labels) ~= size(q.data,2)
       error('Number of labels doesn''t match number of columns')
end

if ~strcmp(q.labels{1},'time')
       error('Expected ''time'' as first column')
end

fprintf(fid, 'name %s\n', fname);
fprintf(fid, 'datacolumns %d\n', size(q.data,2));
fprintf(fid, 'datarows %d\n', size(q.data,1));
fprintf(fid, 'range %f %f\n', min(q.data(:,1)), max(q.data(:,1)));
fprintf(fid, 'endheader\n');

for i=1:length(q.labels)
       fprintf(fid, '%20s\t', q.labels{i});
end
fprintf(fid, '\n');

for i=1:size(q.data,1)
       fprintf(fid, '%20.8f\t', q.data(i,:));
       fprintf(fid, '\n');
end

fclose(fid);

return
```

E.5.3 process_grf.m

```
function [mot, offsets] = process_grf(q, file_name, file_dir, verbosity)
% original version created at Stanford University
% This version has been trimmed down and slightly modified at North
% Carolina State University in order to work more smoothly with C3D_2_OSim.

%% load defaults if necessary
if nargin < 2
    file_name = '';
```

```
end
if (nargin<3)
    file_dir = pwd;
end
if (nargin<4)
    verbosity = 0;
end
if (nargin<1)
    warning('process_grf:args',...
        'Not enough arguments. %s could not be created.',...
        strrep(file_name,'\','\\'));
    return
end

%% set parameters
up_axis = 2;
num_fp = size(q.data,2);

filter_order = 50; % 2 for butter
filter_cutoff = 20;
filter_type = 'none'; % 'none' or 'butter' or 'fir'

% for normal gait
% force_implying_contact = 300;
% cop_F_up_threshold = 200;
% zero_derivative_threshold = 200;
% allowed_gap_threshold = 100;
% range_length_threshold = 0;

% FOR FAST WALKING, THE FOLLOWING MIGHT BE BETTER VALUES:
% force_implying_contact = 300;
% cop_F_up_threshold = 200;
% zero_derivative_threshold = 500;
% allowed_gap_threshold = 50;
% range_length_threshold = 80;

% for jumping
force_implying_contact = 800;
cop_F_up_threshold = 800;
zero_derivative_threshold = Inf;
allowed_gap_threshold = 100;
range_length_threshold = 50;

n = length(q.time);

%% begin processing

q = filter_grf(q, filter_order, filter_cutoff, filter_type);

F_up_deriv = zeros(n, num_fp);
for fp=1:num_fp
       F_up_deriv(1:(n-1), fp) = ...
```

```
          diff(q.data(fp).filteredF(:,up_axis))./diff(q.time);
end
F_up_deriv(n, :) = F_up_deriv(n-1, :);

contact = zeros(n, num_fp);
copcalc = zeros(n, num_fp);

offsets(num_fp) = struct('meanF',[],'meanM',[]);

for fp=1:num_fp

       for step=1:2
            % Find contact ranges/indices
            contactIndices = ...
              find(abs(F_up_deriv(:,fp)) > zero_derivative_threshold | ...
                q.data(fp).filteredF(:,up_axis) > force_implying_contact);
            contactRanges = indices_to_ranges(contactIndices);
            contactRanges = remove_gaps_from_ranges(contactRanges, ...
                allowed_gap_threshold);
            if range_length_threshold > 0
                 contactRanges = prune_short_ranges(contactRanges, ...
                     range_length_threshold);
            end
            contactIndices = ranges_to_indices(contactRanges);
            contact(:,fp) = zeros(n, 1);
          contact(contactIndices,fp) = 100*ones(length(contactIndices), 1);
            q.data(fp).contactIndices = contactIndices;

            % For COP computation, need to narrow down the "contact" range
            % in which we can divide by F_up
            copIndices = contactIndices;
            copRanges = indices_to_ranges(copIndices);
            for i=1:length(copRanges)
                 I = find(q.data(fp).filteredF(copRanges{i},up_axis) > ...
                     cop_F_up_threshold);
                 if isempty(I)
                     copRanges{i} = [];
                 else
                   copRanges{i} = copRanges{i}(I(1)):copRanges{i}(I(end));
                 end
            end
            copIndices = ranges_to_indices(copRanges);
            copcalc(:,fp) = zeros(n, 1);
            copcalc(copIndices,fp) = 120*ones(length(copIndices), 1);
            q.data(fp).copIndices = copIndices;

            % Manually zero the forceplates
            noncontactIndices = setdiff(1:n, contactIndices);

            if isempty(noncontactIndices)
                 warning('process_grf:contactIndices',['did not ',...
                     'find any times with no foot-floor contact ',...
```

```
        '(force plate %d)'],fp);
end
meanF = mean(q.data(fp).F(noncontactIndices,:));
meanM = mean(q.data(fp).M(noncontactIndices,:));

i_lim = length(contactRanges);
peaks = zeros(i_lim);
for i=1:i_lim
     peaks(i) = ...
         max(q.data(fp).filteredF(contactRanges{i},up_axis));
end

offsets(fp).meanF = meanF;
offsets(fp).meanM = meanM;

if verbosity
    fprintf('Forceplate %d step %d', fp, step);
    fprintf_ranges(contactRanges, q.time);
    fprintf('Forces and moments during no contact:');
    fprintf(['MEAN F_forward=%f F_up=%f F_right=%f ',...
        'Mx=%f My=%f Mz=%f'], meanF(1), meanF(2), meanF(3),...
        meanM(1), meanM(2), meanM(3));
    minF = min(q.data(fp).F(noncontactIndices,:));
    minM = min(q.data(fp).M(noncontactIndices,:));
    fprintf(['MIN F_forward=%f F_up=%f F_right=%f ',...
        'Mx=%f My=%f Mz=%f'], minF(1), minF(2), minF(3),...
        minM(1), minM(2), minM(3));
    maxF = max(q.data(fp).F(noncontactIndices,:));
    maxM = max(q.data(fp).M(noncontactIndices,:));
    fprintf(['MAX F_forward=%f F_up=%f F_right=%f ',...
        'Mx=%f My=%f Mz=%f'], maxF(1), maxF(2), maxF(3),...
        maxM(1), maxM(2), maxM(3));

    fprintf('Max overall filtered F_up peak=%f', ...
        max(q.data(fp).filteredF(:,up_axis)));
    fprintf('Average filtered F_up peak=%f', mean(peaks));
end

if step == 1
     if (verbosity); fprintf('Applying DC offset'); end
     Foffset = meanF;
     Moffset = meanM;
     q.data(fp).F = q.data(fp).F - ones(n,1)*Foffset;
     q.data(fp).M = q.data(fp).M - ones(n,1)*Moffset;
     q.data(fp).filteredF = ...
         q.data(fp).filteredF - ones(n,1)*Foffset;
     q.data(fp).filteredM = ...
         q.data(fp).filteredM - ones(n,1)*Moffset;
end

if step == 2
     if (verbosity)
```

```
                    fprintf('Zeroing non-contact forces and torques');
                end
                q.data(fp).F(noncontactIndices,:) = ...
                    zeros(length(noncontactIndices), 3);
                q.data(fp).M(noncontactIndices,:) = ...
                    zeros(length(noncontactIndices), 3);
                if (verbosity); fprintf('Filtering again'); end
              filter_grf(q, filter_order, filter_cutoff, filter_type, fp);
                if (verbosity)
                    fprintf('Manually zeroing filtered data too!');
                end
                q.data(fp).filteredF(noncontactIndices,:) = ...
                    zeros(length(noncontactIndices), 3);
                q.data(fp).filteredM(noncontactIndices,:) = ...
                    zeros(length(noncontactIndices), 3);
            end

            if (verbosity); fprintf(sprintf('\n')); end
        end

        COP = compute_COP(q.data(fp).filteredF, q.data(fp).filteredM, ...
            q.data(fp).FPorigin_model, q.data(fp).copIndices, up_axis);
        COP = fill_gaps(COP, q.data(fp).copIndices);
        T = compute_T_at_COP(COP, q.data(fp).filteredF, ...
            q.data(fp).filteredM, q.data(fp).FPorigin_model);
        q.data(fp).COP = COP;
        q.data(fp).T = T;
end

%% WRITE MOTION FILE
mot.labels = { 'time', ...
    'ground_force_vx', 'ground_force_vy', 'ground_force_vz', ...
    'ground_force_px', 'ground_force_py', 'ground_force_pz', ...
    'ground_force_vx', 'ground_force_vy', 'ground_force_vz', ...
    'ground_force_px', 'ground_force_py', 'ground_force_pz', ...
    'ground_torque_x', 'ground_torque_y', 'ground_torque_z', ...
    'ground_torque_x', 'ground_torque_y', 'ground_torque_z' };
mot.data = [ q.time, q.data(1).filteredF, q.data(1).COP, ...
    q.data(2).filteredF, q.data(2).COP, q.data(1).T q.data(2).T ];
write_motionFile(mot, file_name, file_dir);

return
```

E.5.4 filter_grf.m

```
function q = filter_grf(q, filter_order, filter_cutoff,
filter_type,fpIndices)
% created at Stanford University

if nargin < 5
       fpIndices=1:2;
end

% Filter
% Seems to create overshooting which may be undesirable
if strcmp(filter_type,'butter')
       %fprintf('butterworth filter order=%f cutoff=%f', ...
       %filter_order, filter_cutoff));
       [filterb, filtera] = ...
           butter(filter_order, filter_cutoff/(0.5*q.analog_rate));
       for fp=fpIndices
           q.data(fp).filteredF = filtfilt(filterb, filtera, q.data(fp).F);
           q.data(fp).filteredM = filtfilt(filterb, filtera, q.data(fp).M);
       end
elseif strcmp(filter_type,'fir')
     %fprintf('fir filter order=%f cutoff=%f',filter_order,filter_cutoff));
       B = fir1(filter_order, filter_cutoff/(0.5*q.analog_rate));
       for fp=fpIndices
             q.data(fp).filteredF = filtfilt(B, 1, q.data(fp).F);
             q.data(fp).filteredM = filtfilt(B, 1, q.data(fp).M);
       end
else
       for fp=fpIndices
             q.data(fp).filteredF = q.data(fp).F;
             q.data(fp).filteredM = q.data(fp).M;
       end
end

return
```

E.5.5 indices_to_ranges.m

```
function R = indices_to_ranges(I)
% created at Stanford University

if isempty(I)
       R = {};
       return;
end

R = { [] };
```

```
for i=1:length(I)
	R{end} = [R{end} I(i)];
	if i < length(I) && I(i+1) ~= I(i) + 1
		R = { R{:} [] };
	end
end

return
```

E.5.6 remove_gaps_from_ranges.m

```
function newR = remove_gaps_from_ranges(R, threshold)
% created at Stanford University

newR = {};

if length(R)
	newR = { R{1} };
end

for i=2:length(R)
	if (R{i}(1) - newR{end}(end) - 1) <= threshold
		newR{end} = newR{end}(1):R{i}(end);
	else
		newR = { newR{:} R{i} };
	end
end

return
```

E.5.7 prune_short_ranges.m

```
function newR = prune_short_ranges(R, threshold)
% created at Stanford University

newR = {};

for i=1:length(R)
	if length(R{i}) > threshold
		newR = { newR{:} R{i} };
	end
end

return
```

E.5.8 ranges_to_indices.m

```
function I = ranges_to_indices(R)
% created at Stanford University

I = [];
for i=1:length(R)
       I = [ I R{i} ];
end

return
```

E.5.9 compute_COP.m

```
function COP = compute_COP(F, M, FPorigin, copIndices, up_axis)
% created at Stanford University
% modified slightly to work with C3D_2_OSim

n = size(F,1);

% With respect to the FP origin in the model coordinate system
COP = zeros(n, 3);
COP(copIndices,:) = [M(copIndices,...
    (mod(up_axis,3)+1))./F(copIndices,up_axis),...
    zeros(length(copIndices),1), -M(copIndices,...
    (mod(up_axis-2,3)+1))./F(copIndices,up_axis)];
COP = COP + ones(n,1)*FPorigin';

return
```

E.5.10 fill_gaps.m

```
function outArray = fill_gaps(inArray, validIndices)
% created at Stanford University
% EG, this is based on Allison's interpolate_array

% fill endpoints with average value if endpoints not valid
avg = mean(inArray(validIndices,:));
if validIndices(1) ~= 1
       validIndices = [1 validIndices];
       inArray(1,:) = avg;
end
if validIndices(end) ~= size(inArray,1)
       validIndices = [validIndices size(inArray,1)];
       inArray(end,:) = avg;
end
```

```
gapIndices = setdiff(1:size(inArray,1), validIndices);
validValues = inArray(validIndices,:);

gapValues = interp1(validIndices, validValues, gapIndices, 'cubic');

outArray = inArray;
outArray(gapIndices,:) = gapValues;

return
```

E.5.11 compute_T_at_COP.m

```
function T = compute_T_at_COP(COP, F, M, FPorigin)
% created at Stanford University

n = size(F,1);

T = M - cross((COP - ones(n,1)*FPorigin'), F);

return
```

E.5.12 put_forces_in_mot.m

```
function put_forces_in_mot(F, M, mot_fname, mot_fdir)
% created at Stanford University
% slightly modified to work better with C3D_2_OSim

notI = find_columns_by_label(M.labels, 'ground_force|ground_torque');
I = setdiff(1:length(M.labels), notI);
M.labels = M.labels(I);
M.data = M.data(:,I);

forcecolumns = length(F.labels)-1;

M.labels = {M.labels{:} F.labels{2:end}};
M.data(:,(end+1):(end+forcecolumns)) =
interp1(F.data(:,1),F.data(:,2:end),M.data(:,1));

write_motionFile(M,mot_fname,mot_fdir);

return
```

E.5.13 find_columns_by_labels.m

```
function columns = find_columns_by_label(labels, label_selection)
% created at Stanford University
% label_selection either a cell array of strings or a single string

n = length(labels);

if isa(label_selection, 'char')
	fixed_label_selection = { label_selection };
else
	fixed_label_selection = label_selection;
end

matches = cell(1,n);
for i=1:n
	matches{i} = false;
end
for i=1:length(fixed_label_selection)
	grepresult = regexp(labels, fixed_label_selection{i});
	for j=1:n
		if length(grepresult{j})
			matches{j} = true;
		end
	end
end

columns = [];
for i=1:n
	if matches{i}
		columns = [columns i];
	end
end

return
```

Appendix F – VR Reaching Data Code

F.1 Preprocessing Data

F.1.1 preprocessSYG.m

```
function preprocessSYG(out_format,do_plots)
% preprocess_SYG: Preprocesses SYG Kinematic files and Jason_syncd files.
%
% Input:
% out_format    (OPTIONAL) csv output if out_format&1~=0
%                          and/or mat output if out_format&2~=0
% do_plots      (OPTIOnAL) 1 to show any plots, 0 not to
%
% Defaults:
% out_format = 3
% do_plots = 0
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults if necessary
if (nargin<1)
    out_format = 3;
end
if (nargin<2)
    do_plots = 0;
end

%% choose data directories
syg_dir = [uigetdir(pwd,'Choose syg data directory.'),'\'];
j_syncd_dir = [uigetdir(syg_dir,'Choose Jason_syncd data directory.'),'\'];
out_dir = [uigetdir(j_syncd_dir,...
    sprintf(['Choose output directory.\nWARNING: All output files ',...
    'currently in directory will be deleted.'])),'\'];

% make sure directories exist and remove existing mat data
if (exist(out_dir,'dir'))
    delete(sprintf('%s*.mat',out_dir));
    delete(sprintf('%s*.csv',out_dir));
else
    mkdir(out_dir)
end
if (exist([out_dir,'csv\'],'dir'))
```

```
    delete(sprintf('%s*.mat',[out_dir,'csv\']));
    delete(sprintf('%s*.csv',[out_dir,'csv\']));
else
    mkdir([out_dir,'csv\'])
end
if (exist([out_dir,'mat\'],'dir'))
    delete(sprintf('%s*.mat',[out_dir,'mat\']));
    delete(sprintf('%s*.csv',[out_dir,'mat\']));
else
    mkdir([out_dir,'mat\'])
end

%% begin formatting file(s)
preprocessSYG_helper(syg_dir,j_syncd_dir,out_dir,out_format,do_plots);

fprintf('\n')

return
```

F.1.2 preprocessSYG_helper.m

```
function
preprocessSYG_helper(syg_dir,j_syncd_dir,out_dir,out_format,do_plots)
% preprocessSYG_helper: Recursive helper function that should only be
%                       called from preprocessSYG.m. Gives the ability to
%                       descend recursively into data directories.
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% begin preprocessing data

% remove pwd and parent directory from file list
syg_files = dir(syg_dir);
syg_files(strcmp(syg_files(1).name,'.'))=[];
syg_files(strcmp(syg_files(1).name,'..'))=[];

% loop through all files in directory
i_lim = length(syg_files);
for i=1:i_lim

    % descend into child directory
    if (syg_files(i).isdir)
       preprocessSYG_helper([syg_dir,syg_files(i).name,'\'],j_syncd_dir,...
            out_dir,out_format,do_plots);
        continue
    end

    % if SYG file is found, begin preprocessing
```

```
   syg_name = syg_files(i).name;
if (strcmpi(syg_name(max(strfind(syg_name,'.')):length(syg_name)),'.dat'))

        % read SYG data
        data = readSYG(syg_name, syg_dir);
        if (~isempty(data))

            % if read successfully, reformat data
          fprintf('\n\nSuccessfully read %s.\n',strrep(syg_name,'\','\\'));
            [data num_sensors num_balls ] = formatSYG(data);
            fprintf('Successfully formatted SYG data.\n');

            % if SYG trial 4,5, or 7, search for matching Jason_syncd file
            if (sum(sscanf(syg_name,'%*c%*d%*c%*d%*c%d',1)==[4,5,7])==1)
                % determine directory
                if (strcmp(syg_name(1),'h'))
                    final_syncd_dir = [j_syncd_dir,'H_L',...
                        sscanf(syg_name,'%*c%*d%*c%c',1),'\'];
                else
                    final_syncd_dir = [j_syncd_dir,'E_L',...
                        sscanf(syg_name,'%*c%*d%*c%c',1),'\'];
                end

                % search for matching file
                syncd_files = dir(final_syncd_dir);
                j_lim = length(syncd_files);
                for j=1:j_lim
                    syncd_name = syncd_files(j).name;

                  % if match found, read the data, synchronize it, & add it
                 if (strcmp([syg_name(1:(max(strfind(syg_name,'.'))-1)),...
                          '.exp'],syncd_name))
                        j_syncd = readSyncd(syncd_name,final_syncd_dir);
                        [data,temp] = addSyncd(data,j_syncd,do_plots);
                        if (temp)
                            fprintf(['Successfully added joint angles',...
                                ' from %s.\n'],syncd_name)
                        end
                        break
                    end
                end
            end

            % if out_format&2~=0 save formatted data in mat file
            if (bitand(out_format,uint8(2))~=0)
                save([out_dir,'mat\',syg_name(1:(max(strfind(syg_name,...
                    '.'))-1)),'.mat'],'data','num_sensors','num_balls');
                fprintf('Successfully saved %s.\n',strrep([syg_name(1:...
                    (max(strfind(syg_name,'.'))-1)),'.mat'],'\','\\'));
            end

            % if out_format&1~=0 write formatted data to CSV file
```

```
            if (bitand(out_format,uint8(1))~=0)
              data(isnan(data)) = 0;    % replace NaN with 0 for CSV output
             writeFormattedSYG(data, num_sensors, num_balls,[syg_name(1:...
                 (max(strfind(syg_name,'.'))-1)),'.csv'],[out_dir,'csv\']);
                fprintf('Successfully created %s.\n',strrep([syg_name(1:...
                    (max(strfind(syg_name,'.'))-1)),'.csv'],'\','\\'));
            end

        end
    end
end

return
```

F.1.3 readSYG.m

```
function data = readSYG(file_name, file_dir)
% readSYG: Read SYG data from original dat file.
%
% Input:
% file_name     (OPTIONAL) name of file
% file_dir      (OPTIONAL) file directory
%
% Defaults:
% file_dir = pwd
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults and choose file if not input
if (nargin<2)
    file_dir = pwd;
end
if (nargin<1)
    [file_name,file_dir] = uigetfile('*.dat', 'open SYG .dat file');
end

%% open file
fid = fopen([file_dir file_name],'r','n');

if (fid==-1)
    fprintf('%s could not be opened.\n',file_name);
    data = [];
    return
end

%% read data
data = sscanf(fgetl(fid),'%f')';
data = [data; fscanf(fid,'%f',[length(data),inf])'];
```

```
fclose(fid);

return
```

F.1.4 formatSYG.m

```
function [out, num_sensors, num_balls] = formatSYG(in)
% formatSYG: Reformats SYG data into a usable form.
%
% Input:
% in            the unformatted SYG data
%
% Ouput:
% out           the formatted SYG data
% num_sensors   the number of sensors used in the SYG data
% num_balls     the number of balls present in the SYG data
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% set parameters and format data
Fs = 50;            % sampling frequency
ft2m = 0.3048;      % for converting ft to meters

% fix time
temp = [0;diff(in(:,2))];
temp = [1;find(temp~=0)];
temp2 = max(diff([0;temp;(size(in,1)-temp(end)+1)]))-1;
for j=temp2:-1:0
    in(min(temp+j,size(in,1)),2) = 0:(1/Fs):((length(temp)-1)/Fs);
end

% extract ball data
temp = find(in(:,3)==1);
balls = [in(temp,2) in(temp,4:10)];
balls(:,3:5) = balls(:,3:5)*ft2m;
num_balls = max(balls(:,2))+1;

% extract and rearrange sensor data
temp = setdiff(1:size(in,1),temp);
num_sensors = max(in(temp,4))+1;
in = in(temp,:);
out = zeros(size(in,1)/num_sensors, ((num_sensors+num_balls)*6+6));
temp = find(in(:,4)==0);
out(:,1) = in(temp,2);
for j=1:num_sensors
    out(:,(j*6-4):(j*6+1)) = in(temp,5:10);
    temp = find(in(:,4)==j);
```

```
end
out(:,2:(num_sensors*6+1)) = out(:,2:(num_sensors*6+1))*ft2m;

% rearrange ball data
temp = num_sensors*6+7;
j_lim = num_balls-1;
for j=0:j_lim
    temp2 = balls(balls(:,2)==j,:);
    out((out(:,1)>=min(temp2(:,1)))&(out(:,1)<=max(temp2(:,1))),...
        (j*6+temp):(j*6+temp+5)) = temp2(:,3:8);
end

% fill reserved Jason_syncd area with NaN for easy removal
% if Jason_syncd data not available
ind = num_sensors*6+2;
out(:,ind:(ind+4)) = NaN;

return
```

F.1.5 readSyncd.m

```
function data = readSyncd(file_name, file_dir)
% readSyncd: Read Jason_syncd data from original exp file.
%
% Input:
% file_name     (OPTIONAL) name of file
% file_dir      (OPTIONAL) file directory
%
% Defaults:
% file_dir = pwd
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% choose file if not input
if (nargin<2)
    file_dir = pwd;
end
if (nargin<1)
   [file_name,file_dir] = uigetfile('*.exp', 'open Jason_syncd .exp file');
end

%% open file and read data
fid = fopen([file_dir file_name],'r','n');

if (fid==-1)
    fprintf('%s could not be opened.\n',file_name);
    data = [];
    return
```

```
end

fgetl(fid); fgetl(fid); fgetl(fid); fgetl(fid); fgetl(fid);
data = fscanf(fid,'%*d %f %f %f %f %f',[5 inf])';

fclose(fid);

return
```

F.1.6 addSyncd.m

```
function [data,success] = addSyncd(data,j_syncd,do_plot)
% addSyncd: Synchronizes FastTrack and Syzergy data and adds joint angles.
%
% Input:
% out_format    (OPTIONAL) csv output if out_format&1~=0
%                and/or mat output if out_format&2~=0
%
% Output:
% data          the synchronized data
% success       1 if synchronization was successful, 0 otherwise
%
% Defaults:
% out_format = 3
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% initialize parameters and load defaults if necessary
if (nargin<3)
    do_plot = 0;
end

mmse = inf;             % minimum mean squared error found between curves
ind = [];               % indices for best match between data
test_freqs = 50;        % frequencies to try upsampling FastTrack data to
success = 0;            % changes to 1 if data is successfully synchronized

%% eliminate offset and some bad data
% extract trunk data
syg_trunk = data(:,14:16);
syncd_trunk = j_syncd(:,3:5);

% eliminate bad data occurring at the beginning of Syzergy data
[temp trim_syg] = find((abs(syg_trunk-
ones(size(syg_trunk,1),1)*mean(syg_trunk,1))-
ones(size(syg_trunk,1),1)*std(syg_trunk,1))'<0,1,'first');
[temp2 temp] = find((abs(syg_trunk-
ones(size(syg_trunk,1),1)*mean(syg_trunk,1))-
```

```
ones(size(syg_trunk,1),1)*std(syg_trunk,1))'>0,1,'first');
trim_syg = trim_syg+temp-2;
syg_trunk = syg_trunk((trim_syg+1):end,:);

% find index for eliminating bad data at the beginning of FastTrack data
[temp trim_syncd] = find((abs(syncd_trunk-
ones(size(syncd_trunk,1),1)*mean(syncd_trunk,1))-
ones(size(syncd_trunk,1),1)*std(syncd_trunk,1))'<0,1,'first');
[temp2 temp] = find((abs(syncd_trunk-
ones(size(syncd_trunk,1),1)*mean(syncd_trunk,1))-
ones(size(syncd_trunk,1),1)*std(syncd_trunk,1))'>0,1,'first');
trim_syncd = trim_syncd+temp-2;

% remove offset by making both data sets zero mean
syg_trunk = syg_trunk-ones(size(syg_trunk,1),1)*mean(syg_trunk,1);
syncd_trunk = syncd_trunk-ones(size(syncd_trunk,1),1)*mean(syncd_trunk,1);

% eliminate the larger noise spikes occurring in FastTrack data
for i=1:3
    bad_ind = find((syncd_trunk(:,i)>max(syg_trunk(:,i)))...
            |(syncd_trunk(:,i)<min(syg_trunk(:,i))));
    if (isempty(bad_ind))
        continue
    end
    valid_ind = setdiff(1:size(syncd_trunk,1),bad_ind);
    syncd_trunk(bad_ind,i) = ...
      interp1(valid_ind, syncd_trunk(valid_ind,i), bad_ind, 'cubic');
end

%% match trunk coordinates based on MSE between curves
for freq=test_freqs
    % upsample syncd_trunk to match syg
    syncd_test = upsample(syncd_trunk((trim_syncd+1):end,:),freq/30);

    % determine longer dataset
    if (size(syg_trunk,1)<size(syncd_test,1))
        shorter = syg_trunk;
        longer = syncd_test;
    else
        longer = syg_trunk;
        shorter = syncd_test;
    end

    % slide shorter dataset across longer one in a window that moves to an
    % offset on both sides of the longer data set equal to half its size
    i_lim = size(longer,1);
    offset = round(i_lim*0.5);
    for i=1:i_lim
        window1 = [max(i-offset,1),...
            min(i-offset+size(shorter,1)-1,size(longer,1))];
        window2 = [max(offset-i,1),...
            (window1(2)-window1(1)+max(offset-i,1))];
```

```
        temp = mean(mean((longer(window1(1):window1(2),:)...
            -shorter(window2(1):window2(2),:)).^2));
        if (temp<mmse)
            mmse = temp;
            ind = [window1;window2];
            Fs = freq;
        end
    end
end

%% add joint angles at proper locations

% use first if clause to only add data if a good match is found
% use second if clase to add data no matter what
% if ((~isempty(ind))&&((1/max(max(syg_trunk)))*mmse)<0.01)
if (~isempty(ind))
    j_syncd = upsample(j_syncd,Fs/30);
 if (size(syg_trunk,1)>=size(upsample(syncd_trunk((trim_syncd+1):end,:),...
            Fs/30),1))
        ind = [ind(2,:);ind(1,:)];
    end
    syncd_trunk = upsample(syncd_trunk,Fs/30);
    trim_syncd = floor(trim_syncd*Fs/30);
    ind = ind+[trim_syncd;trim_syg]*ones(1,2);
    ind(:,1) = ind(:,1)-ones(2,1)*(min(ind(:,1))-1);
    data_column = find(isnan(data(1,:)));
    data(ind(2,1):ind(2,2),data_column:(data_column+4)) = ...
        [syncd_trunk(ind(1,1):ind(1,2),:),...
        j_syncd(ind(1,1):ind(1,2),2:-1:1)];
    success = 1;
else
    fprintf('\nSyzergy and FastTrack data could not be synchronized.\n\n');
    return
end

%% plot matched curves if desired
if (do_plot&&(~isempty(ind)))
  syg_curves = data(ind(2,1):ind(2,2),14:16)-ones(ind(2,2)-ind(2,1)+1,1)...
        *mean(data(ind(2,1):ind(2,2),14:16),1);
    syncd_curves = syncd_trunk(ind(1,1):ind(1,2),:);
    figure
    subplot(3,1,1)
    plot(syg_curves(:,1))
    hold on
    plot(syncd_curves(:,1),'r-')
    axis([0 (ind(1,2)-ind(1,1)) 0 1])
    axis 'auto y'
    legend('Syzergy','FastTrack');
    title('Trunk x-coordinate')
    xlabel('Sample #')

    subplot(3,1,2)
```

```
    plot(syg_curves(:,2))
    hold on
    plot(syncd_curves(:,2),'r-')
    axis([0 (ind(1,2)-ind(1,1)) 0 1])
    axis 'auto y'
    title('Trunk y-coordinate')
    xlabel('Sample #')

    subplot(3,1,3)
    plot(syg_curves(:,3))
    hold on
    plot(syncd_curves(:,3),'r-')
    axis([0 (ind(1,2)-ind(1,1)) 0 1])
    axis 'auto y'
    title('Trunk z-coordinate')
    xlabel('Sample #')
end

return
```

F.1.7 upsample.m

```
function data = upsample(data, ratio)
% upsample: Upsample data along each column of data.
%
% Input:
% data          data (1 or 2-dimensional)
% ratio         factor to upsample by
%
% Output:
% data           the upsampled data information
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%%
data_size = size(data);

% space indices apart by the proper amount
data_ind = round(1:ratio:(ratio*data_size(1)));
gap_ind = setdiff(1:round(ratio*data_size(1)),data_ind);

% create gaps in data
temp = zeros(round(data_size(1)*ratio),data_size(2));
temp(data_ind,:) = data;
data = temp;

% fill gaps with interpolation
for i=1:data_size(2)
```

```
    data(gap_ind,i) = interp1(data_ind, data(data_ind,i), gap_ind,
'cubic');
end

return
```

F.1.8 writeFormattedSYG.m

```
function writeFormattedSYG(data, num_sensors, num_balls,...
       file_name, file_dir)
% writeFormattedSYG: Writes formatted SYG and Jason_syncd data to CSV
%
% Input:
% data          the formatted data to write
% num_sensors   the number of SYG sensors the data contains
% num_balls     the number of balls in the SYG data
% file_name     file name to write data to
% file_dir      directory to write data to
%
% Defaults:
% file_dir = pwd
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults and choose file if not input
if (nargin<5)
    file_dir = pwd;
end
if (nargin<4)
    [file_name,file_dir] = uigetfile('*.csv', 'open SYG .csv file');
end

%% open file and write data
% open output file
fid = fopen([file_dir,file_name],'Wt');

if (fid==-1)
    fprintf('%s could not be opened for writing.\n',file_name);
    return
end

% write header
fprintf(fid,'Trial:,%s,,Sensors:,%d,,Balls:,%d\n',...
    file_name(1:(max(strfind(file_name,'.'))-1)),num_sensors,num_balls);
fprintf(fid,'Time,');
fprintf(fid,['Head,,,,,,Hand,,,,,,Syg Trunk,,,,,,Syncd Trunk,,,',...
    'Shoulder Angle,Elbow Angle,']);
fprintf(fid,'Ball%d,,,,,,',1:num_balls);
```

```
fprintf(fid,'\n,');

temp = sprintf('X,Y,Z,rot_x,rot_y,rot_z,');
for j=1:num_sensors
    fprintf(fid,'%s',temp);
end
fprintf(fid,'X,Y,Z,,,');
temp = sprintf('X,Y,Z,target,identified,contact,');
for j=1:num_balls
    fprintf(fid,'%s',temp);
end
fprintf(fid,'\n');

% write data
j_lim = size(data,1);
for j=1:j_lim
    fprintf(fid,'%s\n',sprintf('%f,',data(j,:)));
end

fclose(fid);

return
```

F.2 Extracting Features

F.2.1 extractFeats.m

```
function extractFeats(in_format, conditions, verbosity)
% extractFeats: Extracts features from formatted SYG/syncd kinematic data.
%               Saves features in mat files.
%
% Input:
% in_format     (OPTIONAL) 1 to look for CSV files, 2 to look for mat files
% conditions    (OPTIONAL) vector containing condition numbers to use,
%                          or 0 to use all
% verbosity     (OPTIONAL) 0 for silent analyzation, 1 for full verbosity
%
% Defaults:
% in_format = 2
% conditions = 0
% verbosity = 0
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults if necessary and set parameters
```

```
if (nargin<1)
    in_format = 2;
end
if (nargin<2)
    conditions = 0;
end
if (nargin<3)
    verbosity = 0;
end

%% choose directories
data_dir = [uigetdir(pwd,'Choose data directory.'),'\'];
features_dir = [uigetdir(data_dir,...
  sprintf(['Choose output directory.\nWARNING: All mat files currently',...
    ' in directory will be deleted.'])),'\'];

% make sure directories exist and remove existing mat data
if (exist(features_dir,'dir'))
    delete(sprintf('%s*.mat',features_dir));
else
    mkdir(features_dir)
end
if (exist([features_dir,'h_dom\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'h_dom\']));
else
    mkdir([features_dir,'h_dom\'])
end
if (exist([features_dir,'h_off\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'h_off\']));
else
    mkdir([features_dir,'h_off\'])
end
if (exist([features_dir,'e_stroke_dom\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'e_stroke_dom\']));
else
    mkdir([features_dir,'e_stroke_dom\'])
end
if (exist([features_dir,'e_stroke_off\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'e_stroke_off\']));
else
    mkdir([features_dir,'e_stroke_off\'])
end
if (exist([features_dir,'e_good_dom\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'e_good_dom\']));
else
    mkdir([features_dir,'e_good_dom\'])
end
if (exist([features_dir,'e_good_off\'],'dir'))
    delete(sprintf('%s*.mat',[features_dir,'e_good_off\']));
else
    mkdir([features_dir,'e_good_off\'])
end
```

```
%% begin extracting features
extractFeats_helper(data_dir, in_format, features_dir, conditions,
verbosity);

fprintf('\n')

return
```

F.2.2 extractFeats_helper.m

```
function extractFeats_helper(data_dir, in_format, features_dir, ...
    conditions, verbosity)
% extractFeats: Recursive helper function that should only be
%               called from extractFeats.m. Gives the ability to
%               descend recursively into data directories.
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% begin processing data

% remove pwd and parent directory from file list
files = dir(data_dir);
files(strcmp(files(1).name,'.'))=[];
files(strcmp(files(1).name,'..'))=[];

% loop through all files in directory
i_lim = length(files);
for i=1:i_lim

    % descend into child directory
    if (files(i).isdir)
        extractFeats_helper([data_dir,files(i).name,'\'],...
            in_format,features_dir,conditions, verbosity);
        continue
    end

    % if correct file is found, load data
    file_name = files(i).name;
    read_success = 0;
    if ((in_format==1)&&(strcmpi(file_name(max(strfind(file_name,...
            '.')):length(file_name)),'.csv')))
      [data num_sensors num_balls] = readFormattedSYG(file_name, data_dir);
        if (~isempty(data))
            fprintf('\nSuccessfully read %s.\n',file_name);
            read_success = 1;
        end
    elseif ((in_format==2)&&(strcmpi(file_name(max(strfind(file_name,...
```

```
            '.')):length(file_name)),'.mat')))
        load([data_dir,file_name]);
        if (exist('data','var')&&exist('num_sensors','var')&&...
                exist('num_balls','var'))
         fprintf('\nSuccessfully loaded %s.\n',strrep(file_name,'\','\\'));
            read_success = 1;
        end
    end
    if (~read_success)
        continue
    end

    % if data was loaded correctly, extract features
    temp = sscanf(file_name,'%*c%dl%dc%d',3);
    if (((length(conditions)==1)&&(conditions==0))||...
            (sum(temp(3)==conditions)))
      [features,labels] = analyzeSYG(data,num_sensors,num_balls,verbosity);
        if (strcmp(file_name(1),'h'))
            if (((temp(1)~=5)&&(temp(2)==1))||((temp(1)==5)&&(temp(2)==2)))
                dir_end = [features_dir,'h_dom\']; % healthy dominant arm
            else
              dir_end = [features_dir,'h_off\']; % healthy non-dominant arm
            end
        else
            if ((temp(2)==1)&&sum(temp(1)==[3,7,8,9]))
                % stroke-affected, dominant arm
                dir_end = [features_dir,'e_stroke_dom\'];
            elseif (((temp(2)==1)&&(temp(1)==4))||((temp(2)==2)&&...
                    sum(temp(1)==[1,2,5,6,10,11])))
                % stroke-affected, non-dominant arm
                dir_end = [features_dir,'e_stroke_off\'];
            elseif (((temp(2)==2)&&(temp(1)==4))||((temp(2)==1)&&...
                    sum(temp(1)==[1,2,5,6,10,11])))
                % unaffected, dominant arm
                dir_end = [features_dir,'e_good_dom\'];
            elseif ((temp(2)==2)&&sum(temp(1)==[3,7,8,9]))
                % unaffected, non-dominant arm
                dir_end = [features_dir,'e_good_off\'];
            end
        end
        save([dir_end,file_name(1:(max(strfind(file_name,'.'))-1)),...
            '_features.mat'],'features','labels');
        fprintf('Extracted features to %s.\n',strrep([file_name(1:(max(...
            strfind(file_name,'.'))-1)),'_features.mat'],'\','\\'));
    end
end

return
```

F.2.3 readFormattedSYG.m

```
function [data, num_sensors, num_balls] = ...
    readFormattedSYG(file_name, file_dir)
% readFormattedSYG: Read SYG and Jason_syncd data from formatted CSV file.
%
% Input:
% file_name     (OPTIONAL) name of file
% file_dir      (OPTIONAL) file directory
%
% Defaults:
% file_dir = pwd
%
%
% AUTHOR: John Kelly, North Carolina State University, March 2008


%% load defaults and choose file if not input
if (nargin<2)
    file_dir = pwd;
end
if (nargin<1)
    [file_name,file_dir] = uigetfile('*.csv', 'open SYG .csv file');
end

%% open file
fid = fopen([file_dir file_name],'r','n');

if (fid==-1)
    fprintf('%s could not be opened.\n',file_name);
    data = [];
    return
end

%% read data
data = sscanf(fgetl(fid),'%*s,%*s,%*s,%d,%*s,%d');
num_sensors = data(1); num_balls = data(2);
textscan(fgetl(fid),'%s');
fgetl(fid);
data = fscanf(fid,'%f',[(num_sensors+num_balls)*6+6,inf])';

fclose(fid);

return
```

F.2.4 analyzeSYG.m

```
function [features,labels] = ...
    analyzeSYG(data, num_sensors, num_balls, verbosity)
% analyzeSYG: Analyzes and determines features of SYG and Jason_syncd data.
%
% Input:
% data              the data to analyze
% num_sensors       the number of SYG sensors in the data
% num_balls         the number of balls in the data
% verbosity         (OPTIONAL) 0 to analyze silently, 1 to print or plot
%
% Output:
% features          the features found in the data
% labels            the labels for the features
%
% Defaults:
% verbosity = 0
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults if necessary and set parameters
if (nargin<4)
    verbosity=0;
end

labels = {'Ball #';'Hit Accuracy';'Movement Time';'Peak Velocity';...
    'Time to Peak Velocity';'Velocity Smoothness';...
   'Response Time to Target Ball Being Identified';'Min Shoulder Angle';...
    'Max Shoulder Angle';'Shoulder Range';'Shoulder Excursion';...
    'Min Elbow Angle';'Max Elbow Angle';'Elbow Range';'Elbow Excursion';...
    'Trunk Excursion';'Trunk Pitch';'Ball Speed';...
   'Ball Start Point - X';'Ball Start Point - Y';'Ball Start Point - Z';...
    'Ball End Point - X';'Ball End Point - Y';'Ball End Point - Z';...
    sprintf(['Linear Combination of\nMax Trunk Displacement, Max Head ',...
        'Displacement, and Max Sagittal Trunk Movement']);...
    sprintf(['Linear Combination of\nHand Rotations about the ',...
        'X, Y, and Z-axes']);...
    sprintf(['Linear Combination of\nTrunk Rotations about the ',...
        'X, Y, and Z-axes']);...
    sprintf(['Linear Combination of\nHead Rotations about the ',...
        'X, Y, and Z-axes']);'Max Trunk Displacement';...
    'Max Head Displacement';'Max Sagittal Trunk Displacement';...
    'Total Hand Yaw';'Total Hand Roll';'Total Hand Pitch';...
    'Total Trunk Yaw';'Total Trunk Roll';'Total Trunk Pitch';...
    'Total Head Yaw';'Total Head Roll';'Total Head Pitch'};

Fs = round(1/(data(2,1)-data(1,1)));    % sampling frequency
```

```
loop_control = 1:num_balls;
plot_mode = 0;
head = data(:,2:7);
hand = data(:,8:13);
trunk = data(:,14:19);
ind = 6*(num_sensors)+5;
joints = data(:,ind:(ind+1));
time = data(:,1);
move_thresh = mean(diff(hand(:,2))*Fs)*0.5;

num_features = 40;
features = zeros(num_balls,num_features);

while (1)

%% if requested, plot movement during trial
if (plot_mode==1)
    figure
    subplot(4,1,1)
    plot(time,hand(:,1),'b',time,hand(:,2),'r',time,hand(:,3),'g');
    title('Hand Position')
    legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
    ylabel('Meters')
    xlabel('Time (s)')
    subplot(4,1,2)
    plot(time,trunk(:,1),'b',time,trunk(:,2),'r',time,trunk(:,3),'g');
    title('Trunk Position')
    legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
    ylabel('Meters')
    xlabel('Time (s)')
    subplot(4,1,3)
    plot(time,head(:,1),'b',time,head(:,2),'r',time,head(:,3),'g');
    title('Head Position')
    legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
    ylabel('Meters')
    xlabel('Time (s)')
    subplot(4,1,4)
    plot(time,joints(:,1),'b',time,joints(:,2),'r');
    title('Joint Angles')
    legend('Shoulder','Elbow')
    ylabel('Degrees')
    xlabel('Time (s)')

    figure
    subplot(3,1,1)
    plot(time,hand(:,4),'b',time,hand(:,5),'r',time,hand(:,6),'g');
    title('Hand Rotation')
    legend('pitch','yaw','roll')
    ylabel('Degrees')
    xlabel('Time (s)')
    subplot(3,1,2)
    plot(time,trunk(:,4),'b',time,trunk(:,5),'r',time,trunk(:,6),'g');
```

```
    title('Trunk Rotation')
    legend('pitch','yaw','roll')
    ylabel('Degrees')
    xlabel('Time (s)')
    subplot(3,1,3)
    plot(time,head(:,4),'b',time,head(:,5),'r',time,head(:,6),'g');
    title('Head Rotation')
    legend('pitch','yaw','roll')
    ylabel('Degrees')
    xlabel('Time (s)')
end

%% analyze motion for each ball
for i=loop_control

    % extract ball data
    ind = 6*(num_sensors+i-1)+7;
    ball = data(:,(ind):(ind+5));
    ball_ind = find(ball(:,1)~=0);
    ball = ball(ball_ind,:);

    if (ball(1,4)==0)
        if (verbosity)
            fprintf('Ball %d:\n',i);
            fprintf('not a target ball\n\n');
        end
        continue;
    end
    features(i,1) = i;

    % extract data occurring during ball's flight
    ball_time = time(ball_ind)-time(1);
    hand_p = hand(ball_ind,1:3);
    hand_v = [smooth(diff(hand_p(:,1))),smooth(diff(hand_p(:,2))),...
        smooth(diff(hand_p(:,3)))]*Fs;
    hand_vy = hand_v(:,2);
    hand_a = [smooth(diff(hand_v(:,1))),smooth(diff(hand_v(:,2))),...
        smooth(diff(hand_v(:,3)))]*Fs;
    trunk_p = trunk(ball_ind,1:3);
    head_p = head(ball_ind,1:3);
    joints_p = joints(ball_ind,:);
    hand_rot = hand(ball_ind,4:6);
    trunk_rot = trunk(ball_ind,4:6);
    head_rot = head(ball_ind,4:6);
    head_disp = sqrt(sum(head_p.^2,2));
    trunk_disp = sqrt(sum(trunk_p.^2,2));

    % if subject never reacts to the ball, skip to the next one
    if ((max(hand_vy)<move_thresh)||((max(ball_ind)==size(data,1))&&...
            isempty(find(ball(:,6)==1,1))))
        if (verbosity)
            fprintf('not enough hand movement occurred\n');
```

```
    end
    continue;
end

% movement time
features(i,2) = max(ball(:,6)); % hit or miss
move_end = [find(diff(ball(:,6))==1);length(ball_time)];
move_end = move_end(1); % time of contact or of ball disappearance
ind = round(size(hand_p,1)/2);  % halfway through ball time
 % contact time, or peak vertical displacement from halfway
 % till ball disappearance, minus 15
ind = [find(diff(ball(:,6))==1);find(hand_p(ind:end,2)==...
    max(hand_p(ind:end,2)),1,'last')+ind-1]-15;
% last movement before ind in positive y direction
ind = [find(hand_vy(1:ind(1))>=0,1,'last'),1];
 % last time before ind that vertical displacement is
 % 5% more than min during ball flight
move_start = max([find(hand_p(1:ind(1),2)<...
    (1.05*min(hand_p(1:move_end,2))),1,'last')-1,1]);
move_start = move_start(1);
% time of movement
features(i,3) = ball_time(move_end)-ball_time(move_start);

% peak velocity during movement time and % time to that velocity
features(i,4) = max(hand_vy(move_start:min(move_end,end)));
ind = find(hand_vy(move_start:min(move_end,end))==...
    features(i,4),1,'last')+move_start-1;
features(i,5) = (ind-move_start)/(move_end-move_start);

% if requested, plot response to the ball
if (plot_mode==2)
    figure
    subplot(3,1,1)
    plot(ball_time,hand_p(:,1),'b',ball_time,hand_p(:,2),'r',...
        ball_time,hand_p(:,3),'g');
    hold on; scatter([ball_time(move_start),ball_time(move_end)],...
        [hand_p(move_start,2),hand_p(move_end,2)],'ko');
    title(sprintf('Hand Position for Ball %d',i))
    legend('X (frontal axis)','Y (longitudinal axis)',...
        'Z (sagittal axis)','movement bounds')
    ylabel('Meters')
    xlabel('Time (s)')
    subplot(3,1,2)
    plot(ball_time(2:end),hand_v(:,1),'b',ball_time(2:end),...
        hand_v(:,2),'r',ball_time(2:end),hand_v(:,3),'g');
    ind = find(hand_vy(move_start:min(move_end,end))==...
        features(i,4),1,'last')+move_start-1;
    hold on; scatter(ball_time(ind),hand_vy(ind),'ko');
    title(sprintf('Hand Velocity for Ball %d',i))
    legend('X (frontal axis)','Y (longitudinal axis)',...
        'Z (sagittal axis)','peak during movement')
    ylabel('m/s')
```

```
   xlabel('Time (s)')
   subplot(3,1,3)
   plot(ball_time(3:end),hand_a(:,1),'b',ball_time(3:end),...
       hand_a(:,2),'r',ball_time(3:end),hand_a(:,3),'g');
   title(sprintf('Hand Acceleration for Ball %d',i))
legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
   ylabel('m/s^2')
   xlabel('Time (s)')

   figure
   subplot(3,1,1)
   plot(ball_time,trunk_p(:,1),'b',ball_time,trunk_p(:,2),'r',...
       ball_time,trunk_p(:,3),'g');
   title(sprintf('Trunk Position for Ball %d',i))
legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
   ylabel('Meters')
   xlabel('Time (s)')
   subplot(3,1,2)
   plot(ball_time,head_p(:,1),'b',ball_time,head_p(:,2),'r',...
       ball_time,head_p(:,3),'g');
   title(sprintf('Head Position for Ball %d',i))
legend('X (frontal axis)','Y (longitudinal axis)','Z (sagittal axis)')
   ylabel('Meters')
   xlabel('Time (s)')
   subplot(3,1,3)
   plot(ball_time,joints_p(:,1),'b',ball_time,joints_p(:,2),'r');
   title(sprintf('Joint Angles for Ball %d',i))
   legend('Shoulder','Elbow')
   ylabel('Degrees')
   xlabel('Time (s)')

   figure
   subplot(3,1,1)
   plot(ball_time,hand_rot(:,1),'b',ball_time,hand_rot(:,2),'r',...
       ball_time,hand_rot(:,3),'g');
   title(sprintf('Hand Rotation for Ball %d',i))
   legend('pitch','yaw','roll')
   ylabel('Degrees')
   xlabel('Time (s)')
   subplot(3,1,2)
   plot(ball_time,trunk_rot(:,1),'b',ball_time,trunk_rot(:,2),'r',...
       ball_time,trunk_rot(:,3),'g');
   title(sprintf('Trunk Rotation for Ball %d',i))
   legend('pitch','yaw','roll')
   ylabel('Degrees')
   xlabel('Time (s)')
   subplot(3,1,3)
   plot(ball_time,head_rot(:,1),'b',ball_time,head_rot(:,2),'r',...
       ball_time,head_rot(:,3),'g');
   title(sprintf('Head Rotation for Ball %d',i))
   legend('pitch','yaw','roll')
   ylabel('Degrees')
```

```
        xlabel('Time (s)')

    end

    % trim needed values to movement time
    hand_a = hand_a(move_start:min(move_end,end),:);
    trunk_p = trunk_p(move_start:move_end,:);
    joints_p = joints_p(move_start:move_end,:);
    hand_rot = hand_rot(move_start:move_end,:);
    trunk_rot = trunk_rot(move_start:move_end,:);
    head_rot = head_rot(move_start:move_end,:);
    head_disp = head_disp(move_start:move_end,:);
    trunk_disp = trunk_disp(move_start:move_end,:);

    % velocity smoothness
    % non-positive y velocity during path to ball
    features(i,6) = length(find(hand_a(:,2)<=0));

    % response time to ball being revealed as target
    features(i,7) = ball_time(move_start)-...
        ball_time(find(ball(:,5)==1,1,'first'));

    % analyze joint angles
    ind1 = find(~isnan(joints_p(:,1)));
    ind2 = find(~isnan(joints_p(:,2)));
    if ((length(ind1)<2)||(length(ind2)<2))
        features(i,8) = NaN;
        features(i,9) = NaN;
        features(i,10) = NaN;
        features(i,11) = NaN;
        features(i,12) = NaN;
        features(i,13) = NaN;
        features(i,14) = NaN;
        features(i,15) = NaN;
    else
        features(i,8) = min(joints_p(ind1,1));            % shoulder min
        features(i,9) = max(joints_p(ind1,1));            % shoulder max
        features(i,10) = range(joints_p(ind1,1));         % shoulder range
     features(i,11) = sum(abs(diff(joints_p(ind1,1))));% shoulder excursion
        features(i,12) = min(abs(joints_p(ind2,2)));      % elbow min
        features(i,13) = max(abs(joints_p(ind2,2)));      % elbow max
        features(i,14) = range(joints_p(ind2,2));         % elbow range
        features(i,15) = sum(abs(diff(joints_p(ind2,2))));% elbow excursion
    end

    % analyze trunk movement
  features(i,16) = sum(abs(diff(trunk_disp)));        % cartesian excursion
features(i,17) = sum(abs(diff(trunk_rot(:,1))));    % x-axis rot excursion

    % analyze ball movement
    % mean ball speed
    features(i,18) = mean(sqrt(sum((diff(ball(:,1:3))*Fs).^2,2)));
```

```
    features(i,19) = ball(1,1);         % ball start x
    features(i,20) = ball(1,2);         % ball start y
    features(i,21) = ball(1,3);         % ball stary z
    features(i,22) = ball(end,1);       % ball end x
    features(i,23) = ball(end,2);       % ball end y
    features(i,24) = ball(end,3);       % ball end z

    % 2*max trunk displacement + max head displacement
    % - max trunk z-direction movement
    features(i,25) = 2*max(abs(trunk_disp))+max(abs(head_disp))...
        -max(abs(trunk_p(:,3)));
    % hand rotations
  features(i,26) = sum(abs(hand_rot(:,2)))/Fs+sum(abs(hand_rot(:,3)))/Fs...
        -sum(abs(hand_rot(:,1)))/Fs;
    % trunk rotations
    features(i,27) = sum(abs(trunk_rot(:,2)))/Fs+...
        sum(abs(trunk_rot(:,3)))/Fs-sum(abs(trunk_rot(:,1)))/Fs;
    % head rotations
  features(i,28) = sum(abs(head_rot(:,2)))/Fs+sum(abs(head_rot(:,3)))/Fs...
        -sum(abs(head_rot(:,1)))/Fs;

    features(i,29) = max(abs(trunk_disp));   % max trunk displacement
    features(i,30) = max(abs(head_disp));    % max head disp
    features(i,31) = max(abs(trunk_p(:,3))); % max sagittal trunk disp
    features(i,32) = sum(abs(hand_rot(:,2)))/Fs;% hand yaw excursion
    features(i,33) = sum(abs(hand_rot(:,3)))/Fs;% hand roll excursion
    features(i,34) = sum(abs(hand_rot(:,1)))/Fs;% hand pitch excursion
    features(i,35) = sum(abs(trunk_rot(:,2)))/Fs;% trunk yaw excursion
    features(i,36) = sum(abs(trunk_rot(:,3)))/Fs;% trunk roll excursion
    features(i,37) = sum(abs(trunk_rot(:,1)))/Fs;% trunk pitch excursion
    features(i,38) = sum(abs(head_rot(:,2)))/Fs;% head yaw excursion
    features(i,39) = sum(abs(head_rot(:,3)))/Fs;% head roll excursion
    features(i,40) = sum(abs(head_rot(:,1)))/Fs;% head pitch excursion

    % if desired, print analysis for the ball
    if (verbosity)
        for j=1:num_features
            fprintf('%s: %g\n',labels{j},features(i,j));
        end
        fprintf('\n');
    end
end

% eliminate bad data points (no movement, or not target ball)
features(sum(features(:,2:end),2)==0,:) = [];

% if desired, prompt for data plotting
if (verbosity)
    fprintf(['\nEnter a ball number to plot the response.\n',...
        'Enter 0 to plot the entire trial.\nEnter -1 to quit.\n']);
    in = input('input >> ','s');
    in = sscanf(in,'%d');
```

```
    if ((strcmp(in,''))||(in<-1)||(in>num_balls))
        fprintf('\nIllegal value entered.\n');
        continue;
    elseif (in==-1)
        fprintf('\n');
        return
    elseif (in==0)
        plot_mode = 1;
        loop_control = [];
    else
        plot_mode = 2;
        loop_control = in;
    end
else
    return
end

end

return
```

F.3 Loading Features

F.3.1 loadClassData.m

```
function [Data, Group, feat_lbls, class_lbls] = loadClassData(windowSize)
% loadClassData: Loads data and feature spaces for use in classification.
%
% Input:
% windowSize    the number of target balls to average together
%               use 0 to average all target balls from each trial
%
% Output:
% Data          MxN matrix with M samples and N features per sample
% Group         vector of length M containing the category of each sample
% feat_lbls     labels describing each feature in Data
% class_lbls    labels describing each class in Group
%
% Defaults:
% windowSize = 1
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults
if (nargin<1)
```

```
    windowSize = 1;
end

features_dir = [uigetdir(pwd,sprintf('Choose features directory.')),'\'];

%% compile and load classes
% loop through each class directory
for i=1:6
    switch (i)
        case 1
            class_name = 'h_dom';
        case 2
            class_name = 'h_off';
        case 3
            class_name = 'e_stroke_dom';
        case 4
            class_name = 'e_stroke_off';
        case 5
            class_name = 'e_good_dom';
        case 6
            class_name = 'e_good_off';
    end

    % compile and load class
    cur_dir = [features_dir,class_name,'\'];
    if (exist([cur_dir,class_name,'.mat'],'file'))
        delete([cur_dir,class_name,'.mat']);
    end
    compileFeats([class_name,'.mat'], windowSize, cur_dir);
    load([cur_dir,class_name,'.mat']);

    % store class
    switch (i)
        case 1
            h_dom = features;
        case 2
            h_off = features;
        case 3
            e_stroke_dom = features;
        case 4
            e_stroke_off = features;
        case 5
            e_good_dom = features;
        case 6
            e_good_off = features;
    end
end
feat_lbls = labels;

%% concatenate classes into Training and Test data

Data = [...
```

```
    e_stroke_dom;...
    e_stroke_off;...
    h_dom;...
    e_good_dom;...
    h_off;...
    e_good_off;...
    ];

Group = [...
    2*ones(size(e_stroke_dom,1),1);...
    2*ones(size(e_stroke_off,1),1);...
    ones(size(h_dom,1),1);...
    ones(size(e_good_dom,1),1);...
    ones(size(h_off,1),1);...
    ones(size(e_good_off,1),1)...
    ];

class_lbls = {'healthy';'stroke';};

return
```

F.3.2 compileFeats.m

```
function compileFeats(class_file_name, windowSize, features_dir)
% compileFeats: Compiles individual feature samples into a category
%               Should only be called from loadClassData.m.
%
% Input:
% class_file_name   file name to output the compiled category to
% windowSize        the number of samples to average together
%                   use 0 to average all samples from each trial
% features_dir      the directory where the feature samples are contained
%
% Defaults:
% windowSize = 1
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults and set parameters
if (nargin<2)
    windowSize = 1;
end
if (nargin<3)
    features_dir = [uigetdir(pwd,'Choose features directory.'),'\'];
end

%% check directory
files = dir(features_dir);
```

```
if ((length(files)<3)||((length(files)<4)&&...
        (strcmp(files(3).name,'class.mat'))))
    return
end
files(strcmp(files(1).name,'.'))=[];
files(strcmp(files(1).name,'..'))=[];

%% begin compiling features
out = [];
i_lim = length(files);
% load all samples
for i=1:i_lim
    load([features_dir,files(i).name]);
    if (windowSize==0)
        features = mean(features,1);
    end
    out = [out;features];
end
features = [];

if (windowSize==0)
    windowSize = 1;
end
% compile samples based on windowSize
i_lim = windowSize*floor(size(out,1)/windowSize);
for i=1:windowSize:i_lim
    features = [features;mean(out(i:(i+windowSize-1),:),1)];
end

% save compiled result
save([features_dir,class_file_name],'features','labels');

return
```

F.4 Analyzing Features

F.4.1 plotClassFeats.m

```
function plotClassFeats(Data,Group,feat_lbls,class_lbls,...
    features_used,reduction)
% plotClassFeats: Plots the class PDFs for a set of features.
%
% Input:
% Data          MxN matrix with M samples and N features per sample
% Group         vector of length M containing the category of each sample
% feat_lbls     (OPTIONAL) labels describing each feature in Data
% class_lbls    (OPTIONAL) labels describing each class in Group
% features_used (OPTIONAL) vector containing indices for features to plot
% reduction     (OPTIONAL) reduction>0 to normalize, reduction=2 for PCA
%
% Defaults:
% feat_lbls = num2cell(feat_lbls)
% class_lbls = num2cell(unique(Group))
% features_used = 1:size(Data,2)
% reduction = 0
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults and prepare data
if (nargin<4)
    class_lbls = num2cell(unique(Group));
end
if ((nargin<5)||((length(features_used)<2)&&(features_used==0)))
    features_used = 1:size(Data,2);
end
if (nargin<6)
    reduction=0;
end
if (nargin<3)
    feat_lbls = num2cell(features_used);
end

colors = {'b','r','k','g','c','m'};
Data = reduceFeats(Data,0,reduction);

if (reduction==2)
    i_lim = size(Data,2);
    feat_lbls = cell(i_lim);
    for i=1:i_lim
        feat_lbls{i} = sprintf('Principle Component %d',i);
```

```
    end
end

%% plot PDFs
% loop through features
for i=features_used
    figure
    grid on
    hold on

    % loop through classes
    for j=unique(Group)'
        ind = find((Group==j)&(~isnan(Data(:,i))));
        [f,x] = ksdensity(Data(ind,i));
        fprintf('%s, %s\n',feat_lbls{i},class_lbls{j});
        fprintf('Mean: %f\tStd Dev: %f\n\n',...
            mean(Data(ind,i)),std(Data(ind,i)));
        plot(x,f,colors{j})
    end

    title(feat_lbls{i})
    legend(class_lbls);
end

return
```

F.4.2 reduceFeats.m

```
function Data = reduceFeats(Data,train_ind,option)
% reduceFeats: Reduces a set of features.
%
% Input:
% Data          MxN matrix with M samples and N features per sample
% train_ind  vector indicating samples to use for determining the reduction
% option        option>0 to normalize, option=2 to do PCA
%
% Defaults:
% option = 2
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults
if ((nargin<2)||((length(train_ind)<2)&&(train_ind==0)))
    train_ind = 1:size(Data,1);
end
if (nargin<3)
    option = 2;
end
```

```
%% begin reduction
% normalize Data
if (option>0)
    i_lim = size(Data,2);
    for i=1:i_lim
        train = Data(train_ind,i);
        Data(isnan(Data(:,i)),i) = mean(train(~isnan(train)));
        train = Data(train_ind,i);
        Data(:,i) = (Data(:,i)-mean(train))/std(train);
    end
end

% do PCA
if (option==2)
    train = Data(train_ind,:);
    coeff = princomp(train);
    Data = Data*coeff;
end

return
```

F.4.3 testClassify.m

```
function results = testClassify(Data,Group,training_ratio,test_params,...
    features_test,features_set)
% testClassify: Tests classification on a set of data.
%
% Input:
% Data           MxN matrix with M samples and N features per sample
% Group          vector of length M containing the category of each sample
% training_ratio (OPTIONAL) ratio (0 to 1) of training data to total
% test_params    (OPTIONAL) vector indicating options for testing
%                test_params(1): 0 to test each feature
%                                1 to test a features set many times
%                test_params(2): 0 for kNN, 1 for SVM
%                test_params(3): 0 to not alter data at all
%                                1 to normalize data, and 2 to do PCA
%                test_params(4): k for kNN, polynomial order for SVM
%                test_params(5): number of iterations to test
% features_test  (OPTIONAL) vector containing indices for features to test
% features_set   (OPTIONAL) indices for the features set for each test
%
% Defaults:
% training_ratio = 0.5
% test_params = [1,0,2,3,50]
% features_test = 1:size(Data,2)
% features_set = []
%
%
```

```
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% load defaults
if (nargin<3)
    training_ratio = 0.5;
end
if (nargin<4)
    test_params = [1,0,2,3,50];
end
if (nargin<5)
    features_test = 1:size(Data,2);
%     features_test = [25,26,27,28,3,6];
end
if (nargin<6)
    features_set = [];
end
switch (length(test_params))
    case 1
        test_params = [test_params,0,2,3,50];
    case 2
        test_params = [test_params,2,3,50];
    case 3
        test_params = [test_params,3,50];
    case 4
        test_params = [test_params,50];
end

%% run classification tests
% test each feature once
if (bitand(test_params(1),uint16(1))==0)
    results = zeros(size(Data,2),test_params(5));
    for i=1:test_params(5)
        for j=features_test
            for k=1:1
            % make new feature set and new training set
            features_used = [features_set,j];
            if (k==1)
                [train_ind, test_ind] = ...
                    crossvalind('holdOut',Group,1-training_ratio);
            else
                temp = train_ind;
                train_ind = test_ind;
                test_ind = temp;
            end

            % reduce data
            reduced = ...
               reduceFeats(Data(:,features_used),train_ind,test_params(3));
            if (test_params(3)==2)
                reduced = reduced(:,1:4);
            end
```

```
            %kNN or SVM
            if (bitand(test_params(2),uint16(1))==0)
                [temp2,temp2,temp] = ...
                    kNN(reduced(test_ind,:),reduced(train_ind,:),...
                    Group(train_ind),test_params(4),0,Group(test_ind));
            else
                temp = svmtrain(reduced(train_ind,:),Group(train_ind),...
                    'Kernel_Function','polynomial','Polyorder',...
                   test_params(4),'QuadProg_Opts',optimset('MaxIter',500));
                temp = sum(Group(test_ind)==...
                    svmclassify(temp,reduced(test_ind,:)))/sum(test_ind);
            end

            % tabulate results
            temp = 100-temp;
            results(j,i) = results(j,i)+temp;
            fprintf('Feature %d had %1.2f%% error.\n',j,temp);
            end
        end
    end
    results = [mean(results,2),min(results,[],2),...
        max(results,[],2),var(results,0,2)];
end

% test all features many times
if (bitand(test_params(1),uint16(1))>0)
    results = zeros(test_params(5),1);
    for i=1:test_params(5)
        for k=1:1
        % make new training set
        if (k==1)
            [train_ind, test_ind] = ...
                crossvalind('holdOut',Group,1-training_ratio);
        else
            temp = train_ind;
            train_ind = test_ind;
            test_ind = temp;
        end

        % reduce data
        reduced = ...
            reduceFeats(Data(:,features_test),train_ind,test_params(3));
        if (test_params(3)==2)
            reduced = reduced(:,1:4);
        end

        % kNN or SVM
        if (bitand(test_params(2),uint16(1))==0)
            [temp2,temp2,temp] = ...
                kNN(reduced(test_ind,:),reduced(train_ind,:),...
                Group(train_ind),test_params(4),0,Group(test_ind));
```

```
        else
            temp = svmtrain(reduced(train_ind,:),Group(train_ind),...
              'Kernel_Function','polynomial','Polyorder',test_params(4),...
                'QuadProg_Opts',optimset('MaxIter',500));
            temp = 100*sum(Group(test_ind)==...
                svmclassify(temp,reduced(test_ind,:)))/sum(test_ind);
        end

        % tabulate results
        temp = 100-temp;
        results(i) = results(i)+temp;
        fprintf('Test %d had %1.2f%% error.\n',i,temp);
        end
    end
    results = [mean(results),min(results),max(results),var(results)];
end

return
```

F.4.4 kNN.m

```
function [result,k,accuracy_best] = ...
    kNN(Sample,Training,Group,k,verbosity,Truth,labels)
% kNN:          Uses k Nearest Neigbor to classify data. If k is a vector
%               and the Truth is provided, the accuracy and the best k
%               is also determined.
%
% Input:
% Sample    test data, MxN matrix with M samples and N features per sample
% Training  training data, also an MxN matrix
% Group  vector of length M containing the category of each training sample
% k         (OPTIONAL) scalar or vector of k values to use
% verbosity (OPTIONAL) 0 to classify silently, 1 to print results
% Truth  (OPTIONAL) vector containing the true category of each test sample
% labels    (OPTIONAL) labels for the categories
%
% Output:
% result     vector containing the classification given to each test sample
% k             the best k found from the input k vector
% accuracy_best the best accuracy found
%
% Defaults:
% k = 3
%
%
% AUTHOR: John Kelly, North Carolina State University, April 2008


%% set parameters and load defaults if necessary
accuracy_best = 0; % keeps track of the best accuracy
```

```
k_best = 0;        % best k value
result_best = [];  % vector containing the classification given to each
sample

% determine the number of categories and temporarily offset them to start
at 1
temp = unique(Group);
group_offset = min(Group)-1;
Group = Group-group_offset;

if (nargin<4)
    k=3;
end
if (nargin<5)
    verbosity=1;
end
if (nargin<6)
    truth_exist = 0;
else
    truth_exist = 1;
end
if (nargin<7)
    i_lim = length(unique(Group)');
    labels = cell(i_lim,1);
    for i=unique(Group)'
        labels{i} = sprintf('class %d',temp(i));
    end
end

%% begin classifying
for i=k
    neighbors = zeros(size(Sample,1),i);

    % determine the nearest Training points to each Sample point
    j_lim = size(Sample,1);
    for j=1:j_lim
        dist = ...
           sqrt(sum((ones(size(Training,1),1)*Sample(j,:)-Training).^2,2));
        [temp ind] = sort(dist);
        neighbors(j,:) = ind(1:i);
    end

    % determine the classification of each Sample point
    result = Group(neighbors);
    result = mode(result,2);

    % print out the classification results
    if (verbosity)
        fprintf('\nk = %d:\n',i);
        temp = 100/size(Sample,1);
        for j=unique(Group)'
            fprintf('%1.2f%% of the samples are %s.\n',...
```

```
                temp*length(find(result==j)),labels{j});
        end
    end

    result = result+group_offset;

    % if the true class values are input, determine the accuracy
    if (truth_exist)
        temp = 100*sum(Truth==result)/length(Truth);
        if (verbosity)
            fprintf('This classification was %1.2f%% accurate.\n',temp);
        end
        if (temp>accuracy_best)
            k_best = i;
            result_best = result;
            accuracy_best = temp;
        end
    end

end

% if it exists, print out the best accuracy found
if (verbosity&&~isempty(result_best))
    result = result_best;
    k = k_best;
    fprintf('\nThe best accuracy was %1.2f%%, which was at k = %d.\n',...
        100*sum(Truth==result)/length(Truth),k);
end

fprintf('\n');

return
```