Adjusting and Logging Rheo Knee Parameter Values Using a Smartphone Application

Thomas Andrew Edwards

Final thesis for B.Sc. degree in Mechatronic Engineering Technology
Faculty of Electrical and Computer Engineering
School of Engineering and Natural Sciences
University of Iceland
Adjusting and Logging Rheo Knee Parameter Values Using a Smartphone Application

Thomas Andrew Edwards

24 ECTS thesis submitted in partial fulfillment of a Baccalaureus Scientiarum degree in Mechatronic Engineering

Advisors
Sverrir Guðmundsson
Þórður Halldórsson
Magnús Oddsson

Final thesis for B.Sc. degree in Mechatronic Engineering Technology
Faculty of Electrical and Computer Engineering
School of Engineering and Natural Sciences
University of Iceland
Reykjanesbær, May 2015
Adjusting and Logging Rheo Knee Parameter Values Using a Smartphone Application
Rheo Knee Smartphone Application
24 ECTS thesis submitted in partial fulfillment of a Baccalaureus Scientiarum degree in Mechatronic Engineering

Copyright © 2015 Thomas Andrew Edwards
All rights reserved

Keilir Institute of Technology
School of Engineering and Natural Sciences
University of Iceland
Grænásbraut 910
235 Reykjanessbær

Telephone: + (354) 578 4000

Bibliographic information:

Printing: Háskólaprent ehf.
Fálkagata 2, 107 Reykjavík, May 2015
Abstract

A smartphone application, including its architecture, is presented which can establish a Bluetooth connection with a Rheo Knee. The Rheo Knee is a prosthetic knee produced by Össur which is designed for use by transfemoral amputees. The smartphone application includes functionality which allows it to transfer parameter data from the Rheo Knee to a Google sheet database, as well as adjust parameter values within the Rheo Knee. The adjustment of these values results in different levels of damping in the prosthetic knee joint. These damping magnitudes are used to control the amount of flexion in the knee during stance, as well as the amount of impact experienced as the knee reaches full extension in swing.

The smartphone application can be used to select different parameter values by adjusting the position of a marker within an interactive two-dimensional graph. The initial scheme that was used to map the marker’s position to particular parameter values is evaluated. Test results are analyzed and used as a benchmark for a revised mapping scheme. Also discussed is the efficiency of the application in transferring parameter data from the knee to the Google sheet database.

Aspects of the application’s software architecture can be used as an aid in the development of commercial software which serves a similar purpose. This can either be by direct refinement and integration of programing code, or through the utilization of similar approaches as discussed in this paper.
Útdráttur

Kynnt er hönun smáforrits (e. application or app) fyrir snjallsíma sem getur haft samskipti við Rheo hnéið með Bluetooth samskiptum. Rheo hné er gervileggur framelíður af Össur fyrir fólk sem misst hefur fólegg fyrir ofan hné. Smáforritið býður uppá möguleika til að senda gögn og breytur frá Rheo Knee í töflu gagnagrunn frá Google auk þess að geta stillt breytur inni Rheo hnénu. Stillningar á þessum breytum hafa áhrif á mótsóðumbyndun í Rheo hnjálönum. Breytturnar fyrir mótsóðustillingarnar eru notaðar til að stilla hversu mikið hnéið beygist eftir að stigið er í fótinn og einnig hversu mikið högg finnst þegar réttist úr hnénu í sveiflu.

Hægt er að nota smáforritið til þess að stilla þessar breytur með því að breyta staðsetningu bendils í gagnvirku, tvívíðu myndriti. Upphaflegu aðferðinni, sem notuð var til þess að tengja breytturnar við myndritið er lýst og próf á henni framkvæmd. Niðurstöðurnar úr þeim prófunum eru notaðar sem viðmið til að betrumbæta aðferðina. Einnig er rætt um skilvirkni aðferðarinnar sem smáforritið notar til þess að senda gögn í miðlægann gagnagrøn. Marga eiginleika úr þessu smáforriti væri hægt að nota til hliðsjónar við gerð á smáforriti sem hafa ætti svipaða eiginleika, ætlað fyrir notendur. Þetta væri hægt að gera sem viðbót við forritið sem skrifð var, eða nota svipaðar aðferðir og kynntar voru í þessu verkefni.
To my wife, Rakel
And my children Fannar, Eyþórr, and Jökull—

As with everything I accomplish, you are my inspiration
# Table of Contents

List of Figures ........................................................................................................ xiii

List of Tables............................................................................................................ xv

Abbreviations ............................................................................................................. xvii

Acknowledgements .................................................................................................... xix

1 Introduction ........................................................................................................ 21
  1.1 Objectives ........................................................................................................ 22

2 Background ........................................................................................................... 25

3 Materials ............................................................................................................... 27
  3.1 Rheo Knee Prosthesis ....................................................................................... 27
    3.1.1 Components .............................................................................................. 27
    3.1.2 Control Algorithm ..................................................................................... 30
    3.1.3 Communication protocol .......................................................................... 33
  3.2 Android ............................................................................................................ 35
    3.2.1 Android Activity Lifecycle ........................................................................ 35
    3.2.2 Android Fragment Class .......................................................................... 36
    3.2.3 Android AlertDialogs Class .................................................................. 36
    3.2.4 Android Graphical Components ............................................................... 36

4 Implementation ..................................................................................................... 39
  4.1 Android Rheo Knee Application .................................................................... 39
    4.1.1 Functional Overview ................................................................................. 39
    4.1.2 Program Structural Overview .................................................................. 41
    4.1.3 RheoKneeBluetooth Class ....................................................................... 41
    4.1.4 GraphView and GraphFragment Class ..................................................... 44
    4.1.5 MainActivity Class .................................................................................. 45
  4.2 Google Sheets Database .................................................................................... 46
  4.3 Mapping Graph Position to Parameter Values ................................................ 46
    4.3.1 Qualitative Aim ......................................................................................... 46
    4.3.2 Quantitative Implementation .................................................................... 48

5 Evaluation ............................................................................................................. 51
  5.1 Methods ............................................................................................................ 51
    5.1.1 Testing of Parameter Logging .................................................................... 51
    5.1.2 Testing of the Interactive Graph Scheme ................................................ 51
  5.2 Results .............................................................................................................. 54
    5.2.1 Parameter Logging Results ....................................................................... 54
    5.2.2 Interactive Graph Scheme Results ............................................................ 54
  5.3 Discussion ......................................................................................................... 71
    5.3.1 Parameter Logging .................................................................................... 71
5.3.2 Interactive Graph Scheme ................................................................. 72

6 Conclusion ........................................................................................................ 75

7 References ......................................................................................................... 79

Appendix A – MainActivity Code ...................................................................... 81

Appendix B – RheoKneeBluetooth Code ........................................................ 97

Appendix C – GraphView Code ..................................................................... 119

Appendix D – GraphFragment Code .............................................................. 129

Appendix E – Gantt Chart ............................................................................. 131
List of Figures

Figure 1-1: Adjustment Software User Interface [3] ................................................................. 22
Figure 2-1: Stages of the human gait cycle ........................................................................... 25
Figure 3-1: Rheo Knee by Össur [4] ......................................................................................... 27
Figure 3-2: Cylinders and Disks of Actuator Sketch .............................................................. 28
Figure 3-3: Rheo Knee's Magnetorheological Actuator [2] ....................................................... 29
Figure 3-4: Rheo Knee State Machine [2] .................................................................................. 31
Figure 3-5: Index and Value Packaging ..................................................................................... 34
Figure 3-6: The Activity Class’s States and Event-Handler ...................................................... 36
Figure 4-1: Android App GUI ..................................................................................................... 39
Figure 4-2: Discovered Devices List ......................................................................................... 40
Figure 4-3: Connected App GUI ............................................................................................... 40
Figure 4-4: RheoKneeBluetooth Diagram ............................................................................... 42
Figure 4-5: Initial Layout of Interactive Graph ......................................................................... 44
Figure 4-6: Stance Stability Setting ........................................................................................... 47
Figure 4-7: Stance Dynamic Setting .......................................................................................... 47
Figure 4-8: Swing Stability Setting ............................................................................................ 47
Figure 4-9: Swing Dynamic Setting .......................................................................................... 47
Figure 5-1: Prosthesis Setup ...................................................................................................... 51
Figure 5-2: Protractor ................................................................................................................ 51
Figure 5-3: Gait Data Obtained with the Rheo Knee set at Stable and a Walking Speed of 2.5 km/h ................................................................................................................................. 55
Figure 5-4: Gait Data Obtained with the Rheo Knee set at Mid-Stable and a Walking Speed of 2.5 km/h ................................................................................................................................. 56
Figure 5-5: Gait Data Obtained with the Rheo Knee set at Mid-Stable and a Walking Speed of 5 km/h ................................................................................................................................. 57
Figure 5-6: Gait Data Obtained with the Rheo Knee set at Default and a Walking Speed of 2.5 km/h .......................................................... 58

Figure 5-7: Gait Data Obtained with the Rheo Knee set at Default and a Walking Speed of 5 km/h .......................................................... 59

Figure 5-8: Trial Five Gait Data Showing a Deviant Stride (State 3 not Entered) .......... 60

Figure 5-9: Gait Data Obtained with the Rheo Knee set at Mid-Dynamic and a Walking Speed of 2.5 km/h .................................................. 61

Figure 5-10: Trial Six Gait Data Showing a Deviant Stride (States 2 and 3 not Entered) .......................................................... 62

Figure 5-11: Gait Data Obtained with the Rheo Knee set at Mid-Dynamic and a Walking Speed of 5 km/h .................................................. 63

Figure 5-12: Gait Data Obtained with the Rheo Knee set at Dynamic and a Walking Speed of 2.5 km/h .................................................. 64

Figure 5-13: Trial Eight Gait Data Showing a Deviant Stride (State 3 not Entered) ....... 65

Figure 5-14: Trial Eight Gait Data Showing a Deviant Stride (States 2 and 3 not Entered) ........................................................................ 66

Figure 5-15: Gait Data Obtained with the Rheo Knee set at Dynamic and a Walking Speed of 5 km/h .................................................. 67

Figure 5-16: Trial Nine Gait Data Showing a Deviant Stride (State 3 not Entered) ........ 68

Figure 5-17: Trial Nine Gait Data Showing a Deviant Stride (States 2 and 3 not Entered) .......................................................... 69
List of Tables

Table 3-1: Relevant Current Control Functions and Parameters [7] ............................................. 32
Table 3-2: Relevant Rheo Commands and Responses ................................................................. 33
Table 4-1: Graph Parameter Range and Default Values ............................................................... 48
Table 4-2: X-axis Mapping Scheme .............................................................................................. 49
Table 4-3: Y-axis Mapping Scheme .............................................................................................. 50
Table 5-1: Evaluated Marker Positions ......................................................................................... 52
Table 5-2: Recorded Trial Parameters .......................................................................................... 53
Table 5-3: Average Stance Flexion per Trial ................................................................................. 70
Table 5-4: Summary of Deviant Strides per Trial ......................................................................... 71
Table 6-1: Revised X-axis Mapping Scheme ............................................................................... 75
Table 6-2: Revised Y-axis Mapping Scheme ............................................................................... 76
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>MR</td>
<td>Magnetorheological</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SKD</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor-Transistor Logic</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>USCI</td>
<td>Universal Serial Communication Interface</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank the many employees of Össur who provided me with their insight during the course of my work on this project. With special thanks to Magús Oddson for providing me with the opportunity to work on such a project.

I would also like to thank Þórður Halldórsson for greatly easing the daunting task of constructing intelligible text regarding software architecture. His help during the execution of this project was invaluable.
1 Introduction

The process of human walking is an immensely complex and difficult task. We are unique in our use of bipedal locomotion and spend a relatively long period of our first years acquiring this ability. As pointed out by Inman, Ralston, and Todd in *Human Walking*,

The mastering of the erect bipedal type of locomotion is a relatively prolonged affair and appears to be a learned process, not the result of inborn reflexes. … No one can watch the struggles of an infant as it first attempts to stand, holding onto the edge of a chair or tightly grasping in its hand the supporting fingers of a parent, without feeling that this is pure experimentation rather than the maturation of an inborn reflex [1].

This passage implies that unlike many animals, some of whom walk within moments of birth, humans must rely on learning rather than instinct in their endeavor to walk upright.

This fact is equally true for amputees as they learn the new skill of walking upright on a prosthetic leg. Fortunately, the technology of prosthetics has evolved greatly since the use of simple peg legs. Mechanical knee systems have existed for hundreds of years and development of electronically-controlled prosthetic knees began in the 1970s. These prosthetics are more responsive to the dynamic process of human walking than their purely mechanical predecessors. For instance, unlike purely mechanical systems, in electronic systems the resistive torque supplied by the knee joint can easily be made to vary throughout the progression of the gait cycle. Electronic systems can also be programmed to react to circumstances outside of the normal gait, such as stumbling [2].

The Rheo Knee is a computer-controlled prosthetic knee which has been produced by Össur for about a decade. Since its creation, the Rheo Knee has evolved through multiple generations, with the current version being the Rheo Knee 3. Despite many technological improvements to the original Rheo Knee, no centralized database logging mechanism is available for the current version. This has limited the amount of information which can be collected from users in the field.

Better access to user information could prove advantageous to Rheo knee users, and to the engineers working to enhance Rheo Knee function. For instance, direct access by engineers in Iceland to individual unit information from around the world, could help them more efficiently redress faulty units. This would reduce the engineer’s dependency on middle-men relaying particular fault errors and configuration settings.

Development of future versions of the Rheo Knee could also be positively impacted by access to field data. The information could be used to create statistical models of various phenomenon of interest to engineers. For instance, one aspect of this project was to simplify the software used to configure damping magnitudes of the Rheo Knee’s knee-joint to a particular patient’s preferences. The current adjustment software requires nine different control parameters to be set individually. Each setting can be adjusted by moving a marker along the length of a line. The left end of the line represents that setting’s lowest acceptable
value and the right it’s highest. These parameters must be set so as the user can take maximum advantage of the prosthetics bionics, while still feeling confident on the knee. The user interface for the current version of the software can be seen in Figure 1-1.

![Figure 1-1: Adjustment Software User Interface](image)

This project attempts to make this process more intuitive by creating software which will allow for the adjustment of four of these parameters by positioning a point within an interactive two-dimensional graph. This requires that a relationship be found between the different parameters that will allow for them to be dependent on a single controlling value. This project utilized the knowledge and experience with the knee, of engineers at Össur in an attempt to find this relationship. However, statistical models may provide a more objective approach to fine-tuning the software than engineer experience alone.

### 1.1 Objectives

There are many ways in which the creation of a centralized database logging scheme, and the simplification of the configuration software could be addressed. However, this paper explores the possibility of creating a smartphone application (app) to facilitate the transfer of data between the Rheo Knee and a central database. Data will be transferred between the smartphone and Rheo Knee over a Bluetooth connection. Once the data is contained in the smartphone it will be uploaded to a database using an internet connection. The creation of the database itself, is also within the scope of this paper.

Moreover, The smartphone app should incorporate an interactive two-dimensional graph which can be used to adjust damping magnitudes of the Rheo Knee’s knee-joint at different stages of the stride. The damping magnitudes are configured by positioning a marker within the app’s two-dimensional graph. The aim of this configuration process is to allow for users to maximize their use of the knee’s bionic features while still feeling confident. This paper investigates a scheme for converting the position of this marker into control parameter values which apply proper damping though out the stride.

Completion of the objectives outlined above, provides benchmark information concerning the practicality of using a smartphone app to collect Rheo Knee data over a Bluetooth
connection, as well as establishes a baseline scheme for mapping the position of the marker within the graph to Rheo Knee control parameter values. Also of value are many elements of the app software, which can be used for future testing or refined for use in commercial products.
2 Background

All prosthetic legs attempt to replace the functionality of the biological limb in some respect. Therefore, an understanding of the biological human gait is a prerequisite to the development of this paper. Five distinct stages can be found within the human gait, as shown in Figure 2-1. The first three stages of Figure 2-1 comprise the stance phase.

The stance phase of the gait is the time between heel strike and toe off, during which the foot is in contact with the ground. The last two stages of Figure 2-1 comprise the swing phase. During the swing phase of the gait cycle, the leg leaves the ground and is swung forward to take the next step [1].

The stages of the stance phase are stance flexion, stance extension, and pre-swing. Stance flexion begins when the heel of the foot first contacts the ground. During this part of the gait the knee flexes, or bends, slightly to absorb the impact of heel strike. Next, stance extension occurs with the knee straightening while supporting the body. Prior to the foot leaving the ground at toe-off, the knee begins to flex again during pre-swing, which is essential for ground clearance during swing phase [1].

The stages of the swing phase are swing flexion and swing extension. Once toe-off occurs the knee flexes greatly during the swing flexion stage to prevent the toe from stubbing the ground as the foot swings forward. The final stage is swing extension when the knee straightens in preparation for heel strike [1].
3 Materials

After gaining an appreciation of the natural human gait, efforts of this project shifted towards understanding the existing components which were involved in the development of the smartphone app. The first component was the Rheo Knee itself and a second was the software development kit (SDK) which would be used to develop the app. After these components were satisfactorily understood, development of the smartphone application was undertaken. Then a simple database logging scheme was created. Finally the system was tested.

3.1 Rheo Knee Prosthesis

The Rheo Knee is designed for transfemoral amputees, meaning the leg has been amputated above the knee. The key component of the Rheo Knee is its magnetorheological-damping knee-hinge actuator. This damping actuator allows for flexion control by applying suitable resistive torque to the knee at different stages of the gait cycle. Information from electronic sensors within the knee is used, alongside control firmware, to adjust this damping. This section describes elements of both the hardware and firmware systems on the Rheo Knee.

3.1.1 Components

The Rheo Knee contains multiple sensors, a MSP430F5438A microcontroller, and an actuator. The location of these components can be seen in the Figure 3-1 below.

![Figure 3-1: Rheo Knee by Össur [4]](image)
In Figure 3-1 the actuator is labeled with the number one. Number two shows the location of an angle sensor. Three indicates the location of one of two load-cells, the second of which is located on the opposite face of the knee. The circuit boards, which contain the micro-controller, a labeled with the number four. Finally number 5 shows the location the knee’s battery.

**Actuator**

The damping actuator within the Rheo Knee uses a magnetorheological fluid in order to control rotational damping. A magnetorheological (MR) fluid consists of oil which has small iron particles suspended within it. When a magnetic field is passed through the fluid the iron particles cause an increase in the fluid’s viscosity. This increase in viscosity is accompanied by an increase in the fluids ability to apply resistive shear force and it is this attribute which is exploited by the Rheo Knee damping actuator.

The Rheo Knee takes advantage of this phenomenon by using two sets of thin annulus shaped disks. Each set consists of multiple disks which are fixed to one of two cylindrical supports. An outer cylindrical support with a larger diameter has splines on its inner wall. An inner cylindrical support with a smaller diameter has splines on its outer wall. Figure 3-2 shows a qualitative sketch of these components.

![Figure 3-2: Cylinders and Disks of Actuator Sketch](image)

The disks are inserted between the cylinders, alternating between the inner and out spline attachments, with a small gap left between each disk. Side plates are used to seal both ends of the cylinder assembly and the remaining space between the disks is filled with MR fluid. Despite the outer cylinder forming a fluid tight seal with the side plates, it is still free to rotate about its central axis independent of the side plates. An electromagnet is located at the center of the inner cylinder. An illustration of this assembly can be seen in Figure 3-3.
When no current is supplied to the electromagnet the MR fluid’s viscosity and shear resistance remains low and the Outer Cylinder is free to rotate about the knee axis. As the current increases, so too does the viscosity and shear resistance until the Outer Cylinder is held fast. In the Rheo Knee setup, the Outer Cylinder is connected to the amputee’s leg stump and the rest of the prosthetic is fixed to the side plates; in this way the prosthetic is allowed to flex about the knee axis in a controlled manner.

**Microcontroller**

The MSP430F5438A is a microcontroller produced by Texas Instruments. This microcontroller is well suited for systems which are required to run in real time. The Rheo Knee’s controller runs a Real Time Operating System (RTOS) which helps to manage access of different control processes to the central processing unit. The RTOS simplifies the task of ensuring that control threads are executed in a deterministic manner.

Many peripherals are contained within the microcontroller but the Universal Serial Communication Interface (USCI) was of most interest to the project at hand. Bluetooth communication is accomplished by the Rheo Knee through a USCI which is connected to an external Bluetooth interface. The USCI and Bluetooth interfaces communicate using Universal Asynchronous Receiver/Transmitter (UART) mode, which is one of many modes handled by the microcontroller. The external Bluetooth interface converts a radio frequency (RF) signal into a standard transistor-transistor logic (TTL) signal which is then forwarded to the USCI port. The current system is implemented for point to point communication over Classic Bluetooth.

**Load cells**

Two load cells are employed within the knee. One load cell is located in the front of the knee and the other is located aft of the knee. These are used to determine both the axial and moment loads placed on the prosthetic. The axial load is found by adding the values of the signals from the two load cell. The moment experienced by the prosthetic is calculated by finding the difference between these two signals. The axial load is used to
determine if the prosthetic is in contact with the ground and supporting the user. The moment is used to determine if the user is in early or late stance phase.

**Angle Sensor**

The current version of the Rheo Knee is equipped with a hall-effect sensor which can measure the angle of the knee joint. A zero angle indicates that the knee is in its non-flexed or extended position. The knee can be flexed to a maximum angle of 120 degrees. During normal gait cycle the knee has a target maximum flexion of 60 degrees, achieved during swing phase.

**Speaker and Vibrator Warning Modes**

The Rheo Knee produces sound and vibration in order to communicate basic information to the user. When the Rheo Knee is powered on it indicates this by producing a beep, vibrating, or both. The same method is used to indicate when the knee is about to power off. On normal shutdown the power button must be held and the Rheo Knee will produce four vibrations/beeps before shutting down. The system will also use these methods to broadcast low battery warnings. The two warning modes, i.e. vibrate and sound, can be individually toggled active or inactive.

**3.1.2 Control Algorithm**

**State Machine**

The backbone of the Rheo Knee control system is a state machine that uses sensory information to detect changes between the gait stages shown in Figure 3-4. Three primary attributes control the transitions of the state machine, i.e. axial load, moment load, and knee angle. A fourth input to the state machine is implemented into the current version of the Rheo Knee. This is a signal produced by an accelerometer and gyro. The data from the gyro allows for users to practice a less uniform gait by helping to safely transition the state machine from stance to swing phase. The normal progression of the state machine is best explained by tracking its progress through the gait cycle.

Axial load sensed by the load-cells is used to differentiate between the stance phase and the swing phase of the gait. Within stance phase different sub-stages are determined using the moment read from the load-cells and angular velocity read from the hall sensor. In early stance the prosthetic is anterior to the user’s center of gravity, which causes a flexion moment in the knee actuator. That is to say that if the actuator did not resist the moment at this point in the gait the knee joint would flex and the angle detected by the hall sensor would increase. As the stride progress through the stance phase, the center of gravity of the user shifts over the prosthetic until the prosthetic is posterior to the user. During this transition the moment on the knee shifts to an extension moment, which tends to straighten the knee.

When axial load is removed from the prosthetic at toe off the state machine shifts into swing phase, which starts with stage four in Figure 2-1. During this stage, damping is removed from the actuator and the knee is allowed to flex greatly as the thigh is swung forward. The transition to the next stage in swing phase occurs when the prosthetic begins to extend after reaching maximum flexion. This transition is governed by
information from the hall sensor [7]. Figure 3-4 illustrates the transition requirements between states within the state machine.\(^1\)

![State Machine Diagram](https://via.placeholder.com/150)

**Figure 3-4: Rheo Knee State Machine [2]**

**State Damping Behavior**

Within each state the damping behavior of the actuator is uniquely controlled by altering parameter values and/or functions which are used to derive the electrical current pushed through the actuator. Table 3-1 shows the functions and parameters used to control the actuator current in each state.

\(^1\) Figure 3-4 depict an outdated version of the state machine. The current firmware runs a revised version, with the most notable change being the inclusion of information gathered from the gyro and accelerometer. For the purpose of this paper the state machine depicted in Figure 3-4 is sufficient.
Table 3-1: Relevant Current Control Functions and Parameters² [7]

<table>
<thead>
<tr>
<th>State</th>
<th>Current Control Functions</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>$\tau = (\tau_s + \omega) * S_{1_d}$</td>
<td>$S_{1_d}$</td>
</tr>
<tr>
<td>State 2</td>
<td>$\tau = (\tau_s + \omega) * S_{2_d}$</td>
<td>$S_{2_d}$</td>
</tr>
<tr>
<td>State 3</td>
<td>$\tau = 0$</td>
<td>NA</td>
</tr>
<tr>
<td>State 4</td>
<td>$\tau = (\omega * S_{4_d}) * \left(\frac{(\alpha_c - \alpha_s)}{\alpha_t}\right)$</td>
<td>NA</td>
</tr>
<tr>
<td>State 5</td>
<td>$\tau = (\omega * S_{5_d}) * \left(1 + \frac{(\alpha_{da} - \alpha_c)}{\alpha_{da}} * S_{5_gain}\right)$</td>
<td>$\alpha_{da}$, $S_{5_gain}$</td>
</tr>
</tbody>
</table>

$\tau$: Torque of actuator
$\tau_s$: Value used to set minimum actuator torque
$\omega$: Angular velocity of actuator
$\alpha_c$: Measured angle of actuator
$\alpha_s$: Calibrated zero offset
$\alpha_t$: Maximum target flexion angle
$\alpha_{da}$: Active damping angle
$S_{1\_d}$: Damping coefficient in state 1
$S_{2\_d}$: Damping coefficient in state 2
$S_{3\_d}$: Damping coefficient in state 3
$S_{4\_d}$: Damping coefficient in state 4
$S_{5\_d}$: Damping coefficient in state 5
$S_{5\_gain}$: Damping gain

On heel strike the damping provided by the actuator must be sufficient enough to prevent buckling but a range of damping values above this magnitude can be applied. The damping coefficient of State 1 can be adjusted to achieve different effects. Lower damping values applied in State 1 allow for flexion to occur. Just as in the natural gait, actuator flexion during stance flexion helps to absorb the impact of heel strike. Although this is a positive attribute of the prosthetic, some users prefer to have higher damping values and little flexion at this point in the gait, as it is more stable.

As the user’s center of gravity shift over the prosthetic, the gait cycle enters stance extension and the knee begins to extend. Damping at this stage prevents excessive impact as the actuator reaches full extension. The amount of damping necessary to prevent excessive impact depends on the size and strength of the user. Overly aggressive damping during stance extension could cause difficulty in fully extending the knee. During this stage of the gait cycle, damping magnitudes are adjusted using the damping coefficient of State 2.

Damping during swing extension accomplishes a similar objective as in stance. As the Rheo Knee approaches full extension during swing extension, the damping levels increase to reduce impact on full extension. The two parameters listed in Table 1, i.e.

² Control function are calculated in arbitrary units. This is due to the fact that calculations are carried out on raw bit-value data.
active damping angle and damping gain, can be adjusted to control the angle of application and magnitude of damping during this state. As the knee is extending, the active damping angle governs at what angle the damping will begin to take effect. The magnitude of damping is controlled by the damping gain. The amount of impact allowed on full extension is a matter of user preference. Some users wish to have almost no terminal impact. Others find that some terminal impact is beneficial as it indicates that the knee has reached full extension and is ready to bear their weight.

### 3.1.3 Communication protocol

The Rheo Knee Bluetooth communication protocol consists of a set of input commands to the knee and response messages returned from the knee. All command messages sent to the Rheo Knee must end with a newline character (‘\n’), ASCII code 0x0A. This is needed as the Rheo Knee will parse the input stream using this character. Some commands sent to the knee must be followed with a parameter index and others by both an index and a value. The index refers to the position of the parameter within a parameter-array list contained in the Rheo Knee’s memory.

Most of the commands recognized by the knee are paired with a response message. The content and exact syntax of the response messages varies depending on the command being responded to. However, all response messages contain a tag followed by a content message. Each tag is composed of a unique set of characters ending with a newline. The content message may or may not end with a newline character. Command and response message pairs of interest are shown in Table 3-2, along with a description of their function.

<table>
<thead>
<tr>
<th>Command</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“STARTCOMM\n”</td>
<td>Multiple initialization response messages</td>
<td>Used to initialize communication and query the current state of the Rheo Knee</td>
</tr>
<tr>
<td>“S\n” + index\n + value\n</td>
<td>“S\n” + six-byte content³</td>
<td>Sets a specific parameter by index.</td>
</tr>
<tr>
<td>“TESTWARNING\n”</td>
<td>NA</td>
<td>Causes active warning modes to actuate</td>
</tr>
<tr>
<td>“NEWMODE\n”</td>
<td>“MSG\n” + “Newmode\n”</td>
<td>Set the communication mode within the Rheo Knee</td>
</tr>
</tbody>
</table>

³ The response message’s six-bytes contain information regarding a parameter’s index and value. The six-byte content mirrors the index and value of the input command but is formatted differently. The response notifies the device contacting the Rheo Knee that the set operation was successful.
When the Rheo Knee receives the “STARTCOMM\n” command it responds by outputting a complete list of active knee parameters. Depending on knee settings the number of active parameters varies, but usually there are around 344. The responds begins with the Rheo outputting a list of id-codes, one code for each active parameter. Then response messages containing an “Sn” tag and a six-byte content package are output for each parameter. The content package indicates the index and value of each parameter. After all the parameter information has been output, two “MSG\n” tagged messages are sent. The first “MSG\n” tag is followed by a content string containing the firmware version. The second “MSG\n” tag is followed by the content string “connected\n”, which ends the “STARTCOMM” response. This information can be used by the receiving device to initialize a copy of the parameter array contained within the Rheo Knee. This allows for the receiving device to maintain the same parameter indexing and state as the Rheo Knee.

All “Sn” tag response messages contain a six-byte content package indicating the index and value of a parameter. The Rheo Knee constructs this package from a 4-byte floating point value and a 2-byte short index. During the packaging process the six most significant bits (MSB) of the short index value are dropped. Then the floating point value and the remaining bits from the short index value are compressed into six-bytes with the MSB of each byte set high. This package is then sent to the receiving device which must reverse this process to unpack the content. The packaging process is illustrated in Figure 3-5.

In this diagram the position of the bits of the floating point and short variables within the six-byte content message are indicated below the dotted line. It can be seen that byte 5 of the content message contains bit information from both variables, this is indicated by the red and green colors. The bit indexes of 10 thru 15 of the short variable are missing from the content message, as is indicated by the lack of green indexing with
these values within the six-byte content message. The white colored “H”s indicate the MSB of each byte in the context package has been set high.

The “S\n” input command, not to be confused with the “S\n” tag response message, must be followed by an index and value. The “S\n” input command is used to set the value of a parameter at a specific index. Once the set command is received by the Rheo, the parameters value will be set and an “S\n” tag response message with the new value will be returned. The response confirms that the command was received and processed.

The “NEWMODE” command sets the communication mode of the Rheo Knee. This command’s response message uses the “MSG\n” tag followed by the content message string “Newmode\n”. There are two communication modes within the knee and Newmode must be set in order to utilize the “TESTWARNING” command. The “TESTWARNING” command returns no response message over the Bluetooth channel but instead activates the enabled warning modes. That is to say, the Rheo Knee will produce an audible beep and a vibration if both warning modes are enabled.

3.2 Android

The Android SDK includes a set of software packages designed to function as an app development tool. The software packages contain classes which can be aggregated into a larger app program. These classes perform a multitude of purposes, including hardware management, program encapsulation, and graphical user interface (GUI) creation.

3.2.1 Android Activity Lifecycle

A primary function of Android is to provide a platform on which to run an app. The Android operating system controls the lifecycle of apps running on a device. The Activity class is an Android class specially designed to control the app’s execution [8]. In order to gain access to the functionality of the Activity class, an app’s program code must contain a class which is derived from it.

Open Android apps can be in one of three lifecycle states, i.e. active, paused, or stopped. The active state means that the app is visible on the screen and has focus. When an app is visible but has lost focus, as when it is covered by a pop-up window, it moves to the paused state. Finally when the app is no longer visible on the screen, it is in the stopped state. The Android SDK provides Application Programming Interfaces (API) which allow for managing the app’s transition between states using various event-handlers [8]. These handlers are inherited from Android Activity class and can be overridden in the inheriting class. The operating system then calls these event-handler when the app’s state is changed [9]. An overview of app lifecycle and event-handlers are presented in Figure 3-6.
3.2.2 Android Fragment Class

The Fragment class contains similar lifecycle states and event-handlers as the Activity class. Classes which are derived from the Fragment class must be hosted by an Activity class; that is, they cannot run independently. The Fragment class allows for modular portions of code to be easily incorporated into an Activity class. The lifecycle of a Fragment object is tied to the lifecycle of an Activity object in which it is contained [9].

3.2.3 Android AlertDialogs Class

The Android AlertDialogs class allows for pop-up windows to be easily constructed and displayed. Through its APIs, various content, e.g. scrollable lists, text, and buttons, can be easily incorporated into the pop-up window.

3.2.4 Android Graphical Components

The Android SDK provides many GUI elements which can be easily incorporated into an app. The layout of these graphical components within an app’s GUI are described by Extensible Markup Language (XML) files. XML prescribes an encoding method which can be used by Android to create a GUI layout [9]. Android graphical elements consist not only of the graphics displayed in the GUI but also of event-handlers which allows for control over that particular component. For example, a button graphical element can be included in a GUI layout and the event-handler can be used to develop the program logic which is executed when the button is clicked.
All standard Android GUI components are derived from the `View` class. This class provides functionality for configuring graphical components and event handling. A custom GUI component can be constructed by inheriting from the Android `SurfaceView` class which itself is derived from the `View` class. The `SurfaceView` class provide the tools for drawing custom GUI components and incorporating event handling [8].
4 Implementation

4.1 Android Rheo Knee Application

According to the book, Android for Programmers, at the end of 2013 Android held 81.3% of the smartphone market share [8]. The Android SDK used for the development of the Rheo Knee app is open source and free of charge. This section outlines the Rheo Knee app, also referred to as the app, in terms of program structure and functional purpose.

4.1.1 Functional Overview

The functionality of the Rheo Knee app is explored by explaining the behavior of different components of the app’s GUI. This GUI, as it appears when the app is first launched, is depicted in Figure 4-1.

![Figure 4-1: Android App GUI](image)

After launching, the app must be connected to a Rheo Knee. When the Connect button is pressed, the app performs a Bluetooth discovery process. This action causes a pop-up window to appear which lists the Bluetooth devices, including any Rheo Knee, which have been detected, as shown in Figure 4-2.
A Rheo Knee name can then be selected by pressing the device’s name contained in the list. This will cause the pop-up window to close. The connection process then downloads approximately 344 parameter values and id-codes from the selected Rheo Knee. This information is stored in two arrays, one for values and one for id-codes. Relevant variable values, including Name, Firmware Version, and various settings, are then displayed by the GUI, see Figure 4-3.

The gray marker within the interactive two-dimensional graph reflects the information uploaded from the Rheo Knee. The X coordinate for this marker is derived from the stance control parameters $S1_d$ and $S2_d$, shown in Table 3-1. The Y coordinate is determined based on the swing control parameters $\alpha_{da}$ and $S5_{gain}$, also shown in Table 3-1. From this it
can be seen that the X and Y-axis are associated with stance and swing damping control, respectively.

After the connection process is complete, a position within the two-dimensional graph can be selected by touching the screen within the graph view. The yellow marker will position itself to indicate the last point touched on the graph. When the Set button is pressed, commands are sent to the Rheo Knee which adjust the four damping control parameters to reflect the position of the yellow marker. After each parameter is set, the Rheo Knee responds with a message containing the index and new value of that parameter. This response is used to update the values within the arrays in the app’s memory. As each array value is updated the gray marker shift along the graph’s axes to indicate these new values. The precise mapping of graph position to parameter values is discussed later.

The State button allows access to a pop-up window which displays a list of the 344 parameter id-codes and values. The sound and vibrate checkboxes allow for the indicator mode to be adjusted. Although the Rheo system itself will allows for the sound and vibrate indicator modes to be simultaneously turned off, care has been taken to ensure that the app will not disable an indicator mode if it is the only one active.

When the Log State button is pressed, an internet connection is used to upload to a database all information displayed in the State pop-up window. The database consists of a Google spreadsheet which has been supplemented with a logging-script. Google spreadsheets is a cloud-based spreadsheet service provided by Google. When a Google spreadsheet is created it is stored on a web server. The spreadsheet can then be accessed using a Uniform Resource Locator (URL) obtained upon its creation. A URL is an address used to locate information on the web [10]. The logging-script allows data to be inserted into the spreadsheet by amending a formatted string, containing the data, to the end of a URL used to access the spreadsheet. A more in-depth explanation of the database will be given later.

**4.1.2 Program Structural Overview**

The app was programmed in three main sections. The first section developed was a class which could establish and manage a Bluetooth communication socket between the smartphone and Rheo Knee. The second class developed was derived from the Android SurfaceView class. This class is used to generate the graphics of the interactive two-dimensional graph and is contained within a Fragment class. Lastly, these components were then utilized as member objects in a main class which inherits from the Android Activity class. The full code is contained within the appendix.

**4.1.3 RheoKneeBluetooth Class**

The RheoKneeBluetooth class establishes and manages the Bluetooth connection between the Rheo Knee and the smartphone. In order to accomplish this task the class makes use of multiple Android APIs designed for Bluetooth communication, as well as Android APIs for safely transmitting data between the multiple threads on which the class is running. Also incorporated into the class are callback functions used to send data processed in the RheoKneeBluetooth class to any class which contains it as a member object. Figure 4-4 show an overview of the RheoKneeBluetooth class and its relation to outside objects.
The green rectangle in the figure each represent a class. The rectangle on the right represents the RheoKneeBluetooth class. The labels within this class represent key aspects of the class. The rectangle on the left represents a class which implements the RheoKneeBluetooth class. The implementing class contains no labels as it is not the focus of this section. The dotted lines indicate on which thread different aspects of the diagram are running. The arrows describe the flow of data between different objects. The remainder of this section gives an in-depth explanation of this image.

Before the functionality of the RheoKneeBluetooth class can be used, an object of its type must be created in the implementing class. The constructor is run when the implementing class instantiates the RheoKneeBluetooth object. When the RheoKneeBluetooth class is instantiated, it in turn obtains a BluetoothAdapter object.

The BluetoothAdapter class is an Android class which represents the physical Bluetooth hardware of the device on which the app is running. This class allows for control of the physical Bluetooth hardware and can perform such tasks as enabling the Bluetooth adapter, device discovery, and device connection [9]. If the BluetoothAdapter object indicates that the Bluetooth is disabled at the time of RheoKneeBluetooth’s instantiation, a request is sent to the Android operating system to enable it. The operating system returns a result message to the class which constructed the RheoKneeBluetooth object, indicating the success or failure of the attempt to activate the Bluetooth.

When the constructor has completed, device discovery can be started from the implementing class. The RheoKneeBluetooth object performs the discovery process. During this process it constructs and populates the pop-up window shown in Figure 4-2 with the device names and media access control (MAC) addresses it finds. A MAC address is a unique identifier associated with each Bluetooth enabled device.
When a device name is selected from the list, the BluetoothAdapter object uses the MAC address to initialize a BluetoothDevice object associated with that device. Android provides the BluetoothDevice class which represents a remote Bluetooth device. The BluetoothDevice object is needed to open a Bluetooth communication socket with a remote device. A Bluetooth communication socket is represented by the BluetoothSocket object and is the channel through with data are transferred between the devices [9].

A new thread is created to establish the Bluetooth communication socket with the remote device. This thread, referred to here as the Connection Thread, is needed as the Bluetooth connection process is a blocking process. A blocking process is one which causes its executing thread to stop until the process is complete. This means that the thread will be stuck establishing the connection until it has finished. If this process was performed on the same thread as that which generates the GUI, the app would be unresponsive until the process was complete. The Android operating system will kill apps which are unresponsive for an extended amount of time [9].

During this connection process the RheoKneeBluetooth object will notify the implementing class of changes to the connection status of the app. The connection status can be disconnected, connecting, or connected. This information can be used within the implementing class to do such things as display pop-up windows which indicate the status to the app user.

Once the connection has been established the Communication Thread is used to monitor the Bluetooth socket for incoming messages. Android provides functionality for reading from the socket. This process is also a blocking process, as the read command will hold the thread until there is something to read from the socket [9].

As mentioned earlier, the Rheo Knee sends message tags followed by message content. As all message tags from the Rheo Knee end with a newline character, the Communication Thread will continue reading bytes from the socket until a newline character is received. The tag is then used by the thread to processes the content portion of the message. Currently only two tags are needed for the functionality of the app, i.e. “MSG” and “S”.

If a “MSG” tag is received then the thread will obtain the content portion of the message by reading to the next newline character. The content is then passed from the Communication Thread to a Handler object defined in the RheoKneeBluetooth class and running on the main thread. The Android Handler class is designed for safe messaging between threads. This class allows for two threads to talk without being in sync. That is to say, the sending thread can store a message in a queue held by the Handler which the receiving thread can then access later.

The main thread then processes the message by forwarding it to a callback function in the implementing class. The RheoKneeBluetooth class implements callback function by using a set of abstract function declarations. An abstract function simply delineates the function’s name and parameters but does not provide program logic. These function must be overridden in the implementing class to add this logic. Callback functions eliminate the need for the implementing class to poll the RheoKneeBluetooth class for messages; instead the process is event driven, with the RheoKneeBluetooth class’s abstract functions being implemented as message handlers in the implementing class.
If the “S” tag is received by the Connection Thread then the next six bytes containing a parameter index and value are read, as explained in section 3.1.3. The compressed information is extracted by a function running on the Connection Thread before being passed to the Handler in the main thread. After being received by the Handler, the messages is forwarded to a callback function in the implementing class associated with “S” tag messages.

Sending messages to the knee from the smartphone is a straightforward process. This is achieved simply by passing the intended message to the RheoKneeBluetooth class’s send function from the implementing class. The entire process is carried out within the main thread.

4.1.4 GraphView and GraphFragment Class

The interactive two dimensional graph is implemented into the app using two classes, i.e. the GraphFragment and GraphView classes. The GraphFragment class is derived from the Android Fragment class. This class provides a platform through which the graphics outlined by the GraphView class can be displayed. The GraphFragment is also used to control threads in which the logic of the GraphView class is run. The GraphFragment’s onResume handler, seen in Figure 3-6, starts a thread loop inside the GraphView class which handles events and generates the graphics for the interactive graph. This loop must be stopped when the app loses the focus, which is accomplished using the GraphFragment’s onPause handler, also seen in Figure 3-6.

The GraphView class is derived from the Android SurfaceView class. When the app is launched the GraphView will calculate the position of the elements within the interactive graph based on the size of the space allotted to it. Information about the size of the allotted space is gained through an API inherited from the SurfaceView class. In this way the graph can size itself properly for the screen on which it is running. The initial layout of the graph can be seen in Figure 4-5.

![Figure 4-5: Initial Layout of Interactive Graph](image)

After the position of the elements within the graph have been calculated a display loop is started in a new thread, referred to here as the Graph Thread. The Graph Thread then uses...
an Android SurfaceHolder object to draw the elements onto a canvas. A SurfaceHolder object allows for graphics to be generated in a thread other than the GUI thread and then be passed to the GUI thread for display. On every iteration of the display loop the position of graphical elements are recalculated and passed for displaying on the GUI thread.

When the user touches the screen, an event is generated and dispatched to a SurfaceView event handler function which has been overridden in the GraphView class. This function is passed the coordinates of the last point touched, which are used to update the position of the yellow marker within the graph. During the next iteration through the display loop the position of the yellow marker within the graph view is updated to the last spot touch on the graph. The coordinates of the yellow marker are public members of the GraphView class and can be accessed by outside classes.

The GraphView class also contains a function for displaying a gray marker at a specific point in the graph. This function is used to display the current setting of the Rheo Knee when a connection has been established. Once the location of this marker has been set, it will be displayed on the graph during the next iteration through the display loop.

When the app is not in focus on the device’s screen, it enters the paused or stopped state shown in Figure 3-6. In these states the SurfaceHolder can no longer display graphics to the screen and therefore the display loop must be terminated. This is achieved using the onPause function of the GraphFragment class, as stated earlier.

### 4.1.5 MainActivity Class

The onCreate handler within the MainActivity class is the first event handled after the launching of the app. When the app is launched the MainActivity’s onCreate event handler instantiates the RheoKneeBluetooth and GraphFragment objects. With the exception of the interactive graph, this class also controls the behavior of all graphical components of the app’s GUI, shown in Figure 4-1.

All calls to the RheoKneeBluetooth class are performed by this class, with most initiated by clicking a button on the GUI. The connect button causes the RheoKneeBluetooth class to start the connection process by making a call to device discovery, represented by an arrow in Figure 4-4. The RheoKneeBluetooth class notifies the MainActivity class when the connection has been established. At this point the “STARTCOMMn” command is sent to the Rheo Knee through the RheoKneeBluetooth class. The response messages are then used to initialize parameter arrays and GUI fields maintained by the MainActivity. The last task in the connection process is to send the “NEWMODEn” command to the Rheo Knee to set it communication mode.

When the set button is pressed the coordinates of the yellow marker are retrieved from the GraphView class by the MainActivity class. These coordinates are then converted to corresponding Rheo Knee parameters values. A message is sent through the RheoKneeBluetooth class to set these parameters accordingly in the knee. The response message from the Rheo Knee is then received by the RheoKneeBluetooth class, processed, and forwarded to the MainActivity. The MainActivity then updates the position of the gray marker within the GraphView class.
The information contained in the parameter arrays, obtained through the “STARTCOMM\n” command, is displayed by the MainActivity class when the State button is pressed. This same information is appended to the end of a URL when the Log State button is pressed. The MainActivity uses the URL to send the information to the Google spreadsheet by requesting the Android operating system to dispatch the URL request using a web browser.

The check boxes associated with the warning mode of the Rheo Knee are also initialized using the information contained within the parameter arrays. When either of the check boxes controlling the warning modes are pressed, the MainActivity sends a command through the RheoKneeBluetooth class to change the warning mode in the Rheo Knee. The parameter lists are then updated by the response message past to the MainActivity from the RheoKneeBluetooth class.

4.2 Google Sheets Database

The Google spreadsheet, introduced in section 3.3.1, is used for logging data and contains 441 columns each with a unique header. The name of the Rheo Knee unit being logged is held in the first column. Column two contains the date that the data were logged. The remaining columns are knee parameter columns, each used to hold the value of a single parameter. The parameter column headers mirror the id-codes assigned to the parameters within the Rheo Knee firmware.

A script maintained under an open-source license was provided by Össur for use in this project. The script, which is incorporated into the spreadsheet, controls the parsing of the formatted string appended to the Google sheet’s URL by the app. The appended string indicates the values of different columns contained within the spreadsheet by setting the header equal to a value, e.g. “RheoName = Rheo 341261”. Each column header and value pair is separated by the word “and”. Entering the URL with the appended string in a browser will add a single row of data to the spreadsheet. Column values indicated within the formatted string will be placed in the proper column of that row. The word “Undefined” will be placed in a column if its value is not contained within the formatted string.

4.3 Mapping Graph Position to Parameter Values

4.3.1 Qualitative Aim

The X and Y-axes of the interactive graph are used to adjust the damping of the knee actuator in the stance and swing phases of the gait cycle. With stance damping adjusted according to the position of the yellow marker on the X-axis and swing damping adjusted based on the Y-axis. The general scheme of the graph is to allow for the swing and stance phase behaviors to be independently shifted between stable and dynamic settings.

The yellow marker positioned on the negative end of the X-axis, see Figure 4-6, corresponds to a stance stable setting. This setting will provide a large amount of damping during state 1 of the state machine, which prevents most stance flexion on heel strike. When the knee begins to experience an extension moment as state 2 is entered, the damping is set low allowing for ease of extension. This setting is designed to require less effort from the user. A small flexion angle during stance flexion provides stability on heel strike, while low damping during stance extension helps to ease the extension process.
The stance dynamic setting is achieved by moving the yellow marker towards the positive end of the X-axis, see Figure 4-7. This setting allows for more stance flexion by lowering the damping during state 1. During state 2, damping is increased by the stance dynamic setting, slowing the angular velocity as the knee reaches full extension. The characteristics of this setting are intended to give full advantage of bionics designed into the Rheo Knee. The flexion provided during state 1 mirrors natural stance flexion; this flexion helps to absorb the impact of heel strike. More damping during state 2 is needed to prevent excessive terminal impact on full extension at faster walking speeds. The stance dynamic setting may be better suited for agile users.

Placing the yellow marker at the negative end of the Y-axis, shown in Figure 4-8, will affect the swing phase behavior of the Rheo Knee, setting it to a swing stable setting. With this setting the damping values are kept low as the knee approaches full extension during the state 5 of the state machine. Low damping values during this point in the gait cycle allow for angular velocity to be better maintained. At slower walking speed this characteristic will help to ensure full extension before the prosthetic begin to bear weight on heel strike. As the walking speed increases terminal impact at full extension will begin to become noticeable to the user. As stated earlier some users utilize terminal impact as an indication of full extension. In this way they can be confident that the knee is fully extended and ready to bear weight.

The other end of the Y-axis configures the knee to a swing stable setting. This setting causes damping to take effect earlier during swing extension and with a greater magnitude. The end result is a low terminal impact. As can be seen from the state 5 current control function in Table 3-1, damping at slower walking speeds should not be overly aggressive. This is due to the fact that the damping is proportional to the angular velocity of the knee.
4.3.2 Quantitative Implementation

The MainActivity is used to interpret the information passed between the GraphView and RheoKneeBluetooth classes. This is done by mapping the position of the marker within the graph to parameter values used to control damping within the Rheo Knee. The X and Y-axes of the graph span an interval of -100 to 100 units and it is the coordinates of the yellow marker’s position which must be converted to usable damping parameter values. The aim of the mapping scheme presented here is to explore the limits of the stable and dynamic setting characteristic discussed above.

The Rheo Knee firmware places bounds on the four parameter values used to achieve the stable and dynamic settings, shown in Table 3-1. A parameter value set outside of these bounds will be corrected by the firmware before being used in the control system. The bounds are used to eliminate extreme behavior which could cause the Rheo Knee to be unsafe, e.g. bucking on stance. The allowable range of the four parameter used in the interactive graph scheme are shown in Table 4-1, along with their default values. The parameters are set to default at the end of the manufacturing process.

Table 4-1: Graph Parameter Range and Default Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1_d</td>
<td>[0,100]</td>
<td>50</td>
</tr>
<tr>
<td>S2_d</td>
<td>[0,100]</td>
<td>30</td>
</tr>
<tr>
<td>α_da</td>
<td>[0,10]</td>
<td>2.7</td>
</tr>
<tr>
<td>S5_gain</td>
<td>[0,45]</td>
<td>23</td>
</tr>
</tbody>
</table>

S1_d: Damping coefficient in state 1  
S2_d: Damping coefficient in state 2  
α_da: Active damping angle  
S5_gain: Damping gain

Only a subset of the allowable ranges for these parameters were mapped to positions within the two-dimensional interactive graph; save the active damping angle whose full range was utilized. The experience of engineers at Össur was relied on when choosing the subset of values for each parameter which would be incorporated into the interactive graph. The aim was to choose values just outside of customary bounds, in order to push the Rheo Knees behavior beyond typical settings.

Table 4-2 shows the association between the graph’s marker placement on the X-axis and the control parameter values. The values of the damping coefficients S1_d and S2_d, which control stance phase behavior, are adjusted using this axis. Both the damping coefficients
are directly proportional to the damping magnitudes within their relative states. The mapping functions shown in the table convert a position on the graph into a parameter value.

**Table 4-2: X-axis Mapping Scheme**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Lower Bound</th>
<th>Zero</th>
<th>Upper Bound</th>
<th>Mapping Function</th>
<th>X ≤ 0</th>
<th>X &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Axis</td>
<td>-100</td>
<td>0</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>S1_d</td>
<td>100</td>
<td>50</td>
<td>30</td>
<td>50 + 50 * (-x / 100)</td>
<td>50 - 20 * (x / 100)</td>
<td></td>
</tr>
<tr>
<td>S2_d</td>
<td>10</td>
<td>30</td>
<td>30</td>
<td>30 + 20 * (x / 100)</td>
<td>S2_d = 30</td>
<td></td>
</tr>
</tbody>
</table>

X-Axis: The horizontal axis of the interactive graph
S1_d: Damping coefficient in state 1
S2_d: Damping coefficient in state 2

Table 4-2 shows that the mapping functions are linear on either side of the origin on the X-axis. The behaviors associated with the stance stable setting are most pronounced at the negative end of the X-axis. Here the *state 1 damping coefficient* is set to its higher bound of 100, producing high damping and low knee flexion on heel strike. The low damping coefficient of state 2, ten at this end of the X-axis, should result in lower damping within that state and an easier extension process.

As the yellow marker on the graph moves from the negative extreme towards the zero point of the X-axis, both coefficients shift towards their default values. As can be seen from the mapping function the parameter values change linearly with respect to axis positioning between -100 and zero.

As the yellow marker passes the X-axis origin into positive values, the *state 2 damping coefficient* takes the constant value of thirty. This was deemed to be a reasonable upper limit by Össur’s engineers, as it is sufficient to eliminate terminal impact for most users during stance extension. The mapping function of the *state 1 damping coefficient* shows that its coefficient will continue to decrease in value as the marker moves toward the right of the X-axis. This should result in an increase in stance flexion, while the stance extension stays fixed.

Table 5 shows the association of the marker’s position on the Y-axis and the swing control parameter values. The *active damping angle* controls the angle at which the state 5 current control function, shown in Table 3-1, takes effect [7]. As stated earlier, the full extension angle is considered to be the zero angle. Any angle greater than zero will indicate some amount of knee flexion. During swing extension the angle is decreasing from some positive value towards zero when the knee is fully extended, see Figure 2-1. The *damping gain* is a scalar which is used to control the magnitude of the damping in the actuator.
Table 4-3: Y-axis Mapping Scheme

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Lower Bound</th>
<th>Zero</th>
<th>Upper Bound</th>
<th>Mapping Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y-Axis</td>
<td>-100</td>
<td>0</td>
<td>100</td>
<td>--</td>
</tr>
<tr>
<td>$\alpha_{da}$</td>
<td>0</td>
<td>23</td>
<td>45</td>
<td>$23 - 23 \times (-y / 100)$</td>
</tr>
<tr>
<td>$S5_{gain}$</td>
<td>0</td>
<td>2.7</td>
<td>5.4</td>
<td>$2.7 - 2.7 \times (-y / 100)$</td>
</tr>
</tbody>
</table>

Y-Axis: The vertical axis of the interactive graph
$\alpha_{da}$: State 5 Active damping angle
$S5_{gain}$: State 5 Damping gain

The lower bounds of the Y-axis minimize damping during swing extension. As the marker is moved towards the positive end of the Y-axis both the active damping angle and the damping gain increase. This results in damping being applied at an earlier point during swing extension and with a greater magnitude. The origin of the Y-axis correlates to the default values of both parameters. Again the experience of engineers at Össur was relied upon to determine the range of parameter values incorporated into the mapping scheme.
5 Evaluation

5.1 Methods

5.1.1 Testing of Parameter Logging

The first step in the process of testing the app’s ability to log data was to launch the app. The app was then connected to the Rheo Knee using the app’s connect button. The data loaded to the smartphone during the connection process was transferred to the Google sheet database using the Log State button. Once the app had finished uploading the values to the Google sheet, multiple arbitrary parameters were selected and their values compared between the smartphone and spreadsheet. The same Rheo Knee was used to repeat this process six times. The connection and logging process was not directly timed but a rough estimate was noted by observation.

5.1.2 Testing of the Interactive Graph Scheme

Össur employs multiple above-the-knee amputees whom assisted in the development of new products. The services of one of these individuals was engaged to evaluate the app’s parameter setting capabilities. Tests were carried out with the socket adapter, to which the Rheo Knee is connected, set five degrees off of straight alignment. That is to say that when the user stood with the Rheo Knee fully extended, the posterior angle between the Rheo Knee and socket was 175°, see Figure 5-1. This angle was measured using the protractor seen in Figure 5-2.

![Figure 5-1: Prosthesis Setup](image1.png)  
![Figure 5-2: Protractor](image2.png)

Five different marker positions within the two-dimensional interactive graph were independently tested. These five positions were chosen with the goal of testing the widest
range of settings on each axis in the fewest trials. For each of the five marker positions tested, the coordinates are shown in Table 5-1.

**Table 5-1: Evaluated Marker Positions**

<table>
<thead>
<tr>
<th>Marker Position Number</th>
<th>Mode Name</th>
<th>X-axis Coordinate</th>
<th>Y-axis Coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stable</td>
<td>-98</td>
<td>-95</td>
</tr>
<tr>
<td>2</td>
<td>Mid-Stable</td>
<td>-51</td>
<td>-51</td>
</tr>
<tr>
<td>3</td>
<td>Default</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Mid-Dynamic</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>Dynamic</td>
<td>97</td>
<td>94</td>
</tr>
</tbody>
</table>

During each test the yellow marker was moved to one of the points specified in the table above. Then the *Set* button was pressed. The position of the gray marker, which displays the current settings, was noted to make sure the process was successful. The Rheo Knee was then reset by switching the unit off and on.

Once the Rheo Knee had been reset after configuration via the smartphone app, a connection was established between a Rheo Toolbox software and the Rheo Knee. The Rheo Toolbox software allows for parameter of interest to be flag for recording. Once the recording process is initiated the values of flagged parameters will be logged into the Rheo Knee’s memory every five milliseconds. When the recording session is terminated, the values are uploaded to the PC which is running the Rheo Toolbox software. This data can then be saved to a text file and viewed in Matlab. For each trial an identical set of parameters were flagged for recording, seen in Table 5-2.
**Table 5-2: Recorded Trial Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Name in Firmware</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Pls-&gt;t</td>
<td>Time in seconds of on time</td>
</tr>
<tr>
<td>State 1 damping coefficient</td>
<td>Pls-&gt;c1_damping</td>
<td>$S_1 \text{d}$ in Table 3-1</td>
</tr>
<tr>
<td>State 2 damping coefficient</td>
<td>Pls-&gt;c2_damping</td>
<td>$S_2 \text{d}$ in Table 3-1</td>
</tr>
<tr>
<td>Used to set c1_damping</td>
<td>Pls-&gt;c1_FixedDamping</td>
<td>Indirect access to $S_1 \text{d}$</td>
</tr>
<tr>
<td>Used to set c2_damping</td>
<td>Pls-&gt;c2_FixedDamping</td>
<td>Indirect access to $S_2 \text{d}$</td>
</tr>
<tr>
<td>State 5 Active Damping Angle</td>
<td>Pls-&gt;sw2_softimpactgain</td>
<td>$a_{da}$ in Table 3-1</td>
</tr>
<tr>
<td>State 5 Damping Gain</td>
<td>Pls-&gt;sw2_softimpactangle</td>
<td>$S_5 \text{gain}$ in Table 3-1</td>
</tr>
<tr>
<td>State Variable</td>
<td>Pls-&gt;state_var</td>
<td>The state of the state machine</td>
</tr>
<tr>
<td>Knee Angle</td>
<td>Pls-&gt;qknee_joint</td>
<td>Angle of knee</td>
</tr>
<tr>
<td>Knee Angular Velocity</td>
<td>Pls-&gt;qdknee_joint</td>
<td>Derivative of knee angle</td>
</tr>
<tr>
<td>Moment</td>
<td>Pls-&gt;moment</td>
<td>Difference between load-cells</td>
</tr>
<tr>
<td>Ground Contact</td>
<td>Pls-&gt;on_ground</td>
<td>Ground contact</td>
</tr>
<tr>
<td>Load</td>
<td>Pls-&gt;forces_sens</td>
<td>Sum of load-cells</td>
</tr>
<tr>
<td>Actuator Current</td>
<td>Pls-&gt;current</td>
<td>Electrical current output to actuator</td>
</tr>
</tbody>
</table>

With these parameters being recorded, two trials were run for each marker position in Table 5-1. During both of these trials the tester was asked to walk on a horizontally level treadmill; with one trial being conducted at a speed of 2.5 km/h and the other at 5 km/h. After obtaining data from a single marker position at these two speeds, the Rheo Knee was once again reset by turning it off and on. The entire process was then repeated until data was collected for all marker positions from Table 5-1.
5.2 Results

5.2.1 Parameter Logging Results

The process of connecting the app and the Rheo Knee, during which data is uploaded to the smartphone, took roughly one to two seconds. However, some connection attempts failed. The time required to upload data from the smartphone to the Google spreadsheet varied between roughly 8 to 15 seconds. The values of all parameters which were compared between the app and the Google spreadsheet matched.

5.2.2 Interactive Graph Scheme Results

The following sub-sections display the recorded results from each treadmill trials in graph form. Each graph includes a title which indicates the marker position and treadmill speed of that trial. Within each graph the knee angle is designated by the solid blue line in units of degrees, the magnitude of which is delineated on the Y-axis. The high peaks of the blue line within these graphs delineate swing phase knee flexion. Due to calibration issues, full knee extension correlation to -1.5 degrees on the Y-axis. The X-axis is in units of seconds.

The state of the state machine is also shown in the graph. The state is displayed using a dotted red line. This line tends to appear in a “staircase” pattern, with the lowest level corresponding to state 1 and the highest to state 5. It can be used to determine the progression of the walking cycle throughout the duration of the graph. The position of this line with respect to the Y-axis is arbitrary.

Ground contact is delineated by the green dotted line. As this parameter’s value is Boolean, the line only has two levels. The lower level indicated that the prosthetic is in the swing phase and not in contact with the ground. The upper level indicates stance phase and that the prosthetic is contacting the ground. The position of this line with respect to the Y-axis is also arbitrary.

Each of the following sub-sections show the results from one trial. Within each sub-section a graph displaying a sample of three typical strides from that trial is shown. The figure is followed by text which indicates the total number of strides taken for that trial. In these graphs, stance flexion, which occur after heel strike, is labeled with an indications of the amount of flexion achieved.

If significant deviations from the average stride are present within a trial’s results, a sample of each type of deviation is shown within that trial’s sub-section. The text following this figure describes the nature of the deviation and indicate the number of strides within that trial that contained similar abnormalities.
Trial One: Marker Position 1 at 2.5 km/h

Figure 5-3: Gait Data Obtained with the Rheo Knee set at Stable and a Walking Speed of 2.5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 1, shown in Table 5-1, and a walking speed of 2.5 km/h. A total of 27 strides were taken during this trial with no significant deviations from the data shown in Figure 5-3.

4 Only one trial was satisfactorily run for this marker position. The 5 km/h trial at this position had an improperly adjusted Rheo Knee.
Trial Two: Marker Position 2 at 2.5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 2, shown in Table 5-1, and a walking speed of 2.5 km/h. A total of 31 strides were taken during this trial with no significant deviations from the data shown in Figure 5-4.
Trial Three: Marker Position 2 at 5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 2, shown in Table 5-1, and a walking speed of 5 km/h. A total of 15 strides were taken during this trial with no significant deviations from the data shown in Figure 5-5.
Trial Four: Marker Position 3 at 2.5 km/h

Figure 5-6: Gait Data Obtained with the Rheo Knee set at Default and a Walking Speed of 2.5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 3, shown in Table 5-1, and a walking speed of 2.5 km/h. A total of 14 strides were taken during this trial with no significant deviations from the data shown in Figure 5-6.
Trial Five: Marker Position 3 at 5 km/h

Figure 5-7: Gait Data Obtained with the Rheo Knee set at Default and a Walking Speed of 5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 3, shown in Table 5-1, and a walking speed of 5 km/h. A total of 55 strides were taken during this trial with four strides deviating significantly from the data shown in Figure 5-7. A graph depicting the deviant strides is shown on the next page.
Figure 5-8: Trial Five Gait Data Showing a Deviant Stride (State 3 not Entered)

The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 3, shown in Table 5-1, and a walking speed of 5 km/h. In Figure 5-8, three full strides are depicted with the middle stride indicating that the state machine did not enter state 3. The missing state corresponds to the pre-swing stage of the gait cycle, as shown in Figure 2-1. In total 4 out of 55 strides contained this abnormality.
Trial Six: Marker Position 4 at 2.5 km/h

Figure 5-9: Gait Data Obtained with the Rheo Knee set at Mid-Dynamic and a Walking Speed of 2.5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 4, shown in Table 5-1, and a walking speed of 2.5 km/h. A total of 37 strides were taken during this trial with one stride deviating significantly from the data shown in Figure 5-9. A graph depicting the deviant stride is shown on the next page.
The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 4, shown in Table 5-1, and a walking speed of 2.5 km/h. In Figure 5-10, three full strides are depicted with the middle stride indicating that the state machine did not enter states 2 and 3. The missing states correspond to the stance extension and preswing stages of the gait cycle, as shown in Figure 2-1. In total 1 out of 37 strides contained this abnormality.
Trial Seven: Marker Position 4 at 5 km/h

Figure 5-11: Gait Data Obtained with the Rheo Knee set at Mid-Dynamic and a Walking Speed of 5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 4, shown in Table 5-1, and a walking speed of 5 km/h. A total of 23 strides were taken during this trial with no significant deviations from the data shown in Figure 5-11.
Trial Eight: Marker Position 5 at 2.5 km/h

Figure 5-12: Gait Data Obtained with the Rheo Knee set at Dynamic and a Walking Speed of 2.5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 2.5 km/h. A total of 45 strides were taken during this trial with eleven strides deviating significantly from the data shown in Figure 5-9. Graphs depicting the deviant strides are shown on the next two pages.
The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 2.5 km/h. In Figure 5-13, three full strides are depicted with the middle stride indicating that the state machine did not enter state 3. The missing state corresponds to the pre-swing stage of the gait cycle, as shown in Figure 2-1. In total 2 out of 45 strides contained this abnormality.
The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 2.5 km/h. In Figure 5-14, three full strides are depicted with the middle stride indicating that the state machine did not enter states 2 and 3. The missing states correspond to the stance extension and pre-swing stages of the gait cycle, as shown in Figure 2-1. In total 9 out of 45 strides contained this abnormality.
Trial Nine: Marker Position 5 at 5 km/h

Figure 5-15: Gait Data Obtained with the Rheo Knee set at Dynamic and a Walking Speed of 5 km/h

The above graph depicts three typical strides from trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 5 km/h. In Figure 5-15, it can be seen that during the transition between the second and third stride, the state machine briefly jumps to state 2 after heel strike. This phenomenon happened in 25 out of the 53 total strides taken during this trial. Also of note is the fact that consistently the knee did not reach full extension before heel strike. Not present in the Figure 5-15 are eight deviant strides which are shown on the following two pages.
Figure 5-16: Trial Nine Gait Data Showing a Deviant Stride (State 3 not Entered)

The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 5 km/h. In Figure 5-16, three full strides are depicted with the middle stride indicating that the state machine did not enter state 3. The missing state corresponds to the pre-swing stage of the gait cycle, as shown in Figure 2-1. In total 4 out of 53 strides contained this abnormality.
The above graph depicts a deviant stride contained in trial data obtained with the Rheo Knee set using marker position 5, shown in Table 5-1, and a walking speed of 5 km/h. In Figure 5-17, three full strides are depicted with the middle stride indicating that the state machine did not enter states 2 and 3. The missing states correspond to the stance extension and pre-swing stages of the gait cycle, as shown in Figure 2-1. In total 4 out of 53 strides contained this abnormality.
## Summary of Trial Results

*Table 5-3: Average Stance Flexion per Trial*

<table>
<thead>
<tr>
<th>Trial Number</th>
<th>Position and Speed</th>
<th>Average Stance Flexion Over Ten Typical Strides[^5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stable 2.5 km/h</td>
<td>2.2°</td>
</tr>
<tr>
<td>2</td>
<td>Mid-Stable 2.5 km/h</td>
<td>3.2°</td>
</tr>
<tr>
<td>3</td>
<td>Mid-Stable 5 km/h</td>
<td>2.8°</td>
</tr>
<tr>
<td>4</td>
<td>Default 2.5 km/h</td>
<td>4.9°</td>
</tr>
<tr>
<td>5</td>
<td>Default 5 km/h</td>
<td>4.2°</td>
</tr>
<tr>
<td>6</td>
<td>Mid-Dynamic 2.5 km/h</td>
<td>5.1°</td>
</tr>
<tr>
<td>7</td>
<td>Mid-Dynamic 5 km/h</td>
<td>4.4°</td>
</tr>
<tr>
<td>8</td>
<td>Dynamic 2.5 km/h</td>
<td>6.6°</td>
</tr>
<tr>
<td>9</td>
<td>Dynamic 5 km/h</td>
<td>7.4°</td>
</tr>
</tbody>
</table>

[^5]: The zero-calibration error in the Rheo Knee has been corrected for by adding 1.5° to all results.
Table 5-4: Summary of Deviant Strides per Trial

<table>
<thead>
<tr>
<th>Trial Number</th>
<th>Position and Speed</th>
<th>Percent of Strides Without State 3</th>
<th>Percent of Strides Without States 2 and 3</th>
<th>Total Percent of Deviant Strides</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stable 2.5 km/h</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>Mid-Stable 2.5 km/h</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>Mid-Stable 5 km/h</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>Default 2.5 km/h</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>Default 5 km/h</td>
<td>7%</td>
<td>0%</td>
<td>7%</td>
</tr>
<tr>
<td>6</td>
<td>Mid-Dynamic 2.5 km/h</td>
<td>0%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>7</td>
<td>Mid-Dynamic 5 km/h</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>Dynamic 2.5 km/h</td>
<td>4%</td>
<td>20%</td>
<td>24%</td>
</tr>
<tr>
<td>9</td>
<td>Dynamic 5 km/h</td>
<td>7%</td>
<td>7%</td>
<td>14%</td>
</tr>
</tbody>
</table>

5.3 Discussion

5.3.1 Parameter Logging

The process of uploading data from the Rheo Knee to the smartphone over a Bluetooth connection was efficient enough to transfer all parameter information within about one to two seconds. Once the information was uploaded to the smartphone, it took roughly 8 to 15 seconds to upload this data to the Google spreadsheet database. Although these time seem reasonable, the current smartphone app can only provide a snapshot of the state of any given Rheo Knee unit. This is because only information regarding the state of the Rheo Knee at the moment of connection is currently transferred.

It is conceivable that the data transfer process between the Rheo Knee and smartphone is quick enough to upload larger amounts of data in a reasonable amount of time. If so, it could be used to collect recorded trial data in the same way that the Rheo Toolbox software does. This could provide a portable tool for easily conducting trials in the field. In order to test the transfer time of a larger amounts of data over the Bluetooth connection the current smartphone app would have to be altered but much of the base programing code could be reused.
5.3.2 Interactive Graph Scheme

It can be seen from the data gathered over the different trials that the stance flexion angle tended to increase as the position of the yellow marker advanced from the stable towards the dynamic end of the interactive graph’s X-axis. This can be seen in Table 5-3. The minimum average stance flexion of 2.2° was experienced with the stance stable mode setting. The maximum average stance flexion of 7.4° was experienced with the stance dynamic setting. It should be noted that during the trial which obtained 7.4° stance flexion, full knee extension was not obtained before heel strike. This may have effect the stance flexion angle results. However, the general stance flexion trend behaved as expected with respect to the interactive graph scheme. This is shown by the increase in stance flexion as the yellow marker shifted from stable to dynamic along the X-axis.

It is difficult to draw firm conclusions, based on the obtained data, of the effect of the interactive graph scheme on stance extension behavior. This is due to the fact that the angle of stance flexion varies between the trials run, which makes comparison of stance extension over the trials difficult. As the effect of stance extension damping is subtle, it may be best to rely on user feedback to determine its proper values within the interactive graph scheme.

The effect of the interactive graph’s Y-axis on swing damping can be seen by examining the swing extension behavior within the different trials. A lack of swing extension damping in the stable and mid-stable modes is exhibited by the solid blue line, representing the knee flexion angle, as it drops toward zero from the high peaks which occur during swing phase. In Figure 5-3 thru Figure 5-5, the graph oscillates slightly about the zero point after it reaches full extension. This indicates that swing extension was underdamped and that full extension produced an impact noticeable to the user. These oscillations subside by trial 3 when the Rheo Knee has been set using the default parameter values, corresponding to the origin of the interactive two-dimensional scheme. The same regions of graph in Figure 5-15 show the slope of the blue line increase, becoming less steep, as it approaches full extension. The damping is so aggressive in Figure 5-15 that full extension is not reached before heel strike. Despite the dynamic mode’s damping levels being too aggressive, the general swing extension trend behaved as expected with respect to the interactive graph scheme, with damping levels increasing as the yellow marker shifted from the stable to dynamic mode along the Y-axis.

Two distinct stride deviations were evident in the trial results. In one the state machine never enters state 3. Instead of the normal state progression from state 2 into state 3, the state machine reverts to state 1 before entering state 4, as seen in Figure 5-8, Figure 5-13 and Figure 5-16. This deviation can be referred to as “late stance locking,” as the low damping magnitudes of state 3 are not applied before toe-off. This type of deviant stride presents a danger of toe-stubbing during swing phase as it reduces the swing flexion angle.

The second stride deviation seen in the test results was one in which neither states 2 or 3 are entered. This deviation can be referred to as “damped buckling” as the knee never enters stance extension but instead continually flexes under state 1 damping conditions. During these types of deviant strides the user is force to rapidly transfer their weight to their sound

---

6 See Table 5-1 for mode names. “Stance” or “Swing” before a mode name refer to the X and Y-axis of that mode. With “Stance” refering to the X-axis and “Swing” referring to the Y-axis.
leg in order to halt further flexion of the prosthetic knee. This abnormality interrupts the normal rhythm of the gait cycle, or worse, result in a fall.

Both types of deviant strides were absent in trials conducted using the Stable and Mid-Stable settings, as seen in Table 5-4. The first cases of “late stance locking” occurred with the Rheo Knee set to the Default setting. The exact cases of “late stance locking” have not been determined. Further testing and analysis is needed in order to determine if the interactive graph scheme is the cause of this problem and if so how it could be altered in order to remedy the problem.

Strides effected by “damped buckling” became prominent in the trials conducted using the dynamic mode setting, as seen in Table 5-4. The lower damping magnitudes applied to the knee joint during early stance is most likely the cause of this problem. This low damping magnitudes results in a large knee flexion angle early in the gait cycle which makes it difficult for the user to shift their weight over the prosthetic in order to achieve stance extension. Increasing the damping magnitude of the dynamic mode setting may reduce the number of strides effected by “damped buckling.”

As seen in Figure 5-15, during trial nine the state machine briefly entered state 2 just after heel strike in 25 of 53 strides. This was a consequence of full knee extension not being reached before the stance phase was entered. Lowering the swing extension damping magnitudes of the dynamic mode setting, so as full extension is reached before heel strike, would most likely minimize the presence of this phenomenon.
6 Conclusion

The Bluetooth connection between the smartphone and Rheo Knee proved to be efficient enough for the purposes of transmitting a snapshot of the status of the Rheo Knee to a centralized Google sheet database. In order to test the limits of the connection speed, the app could be modified to incorporate functionality, similar to that of the Rheo Toolbox software, for uploading recorded data files. In the event that the Bluetooth connection proves capable of transferring these data files in an acceptable amount of time, a more robust centralized database will have to be created, as the current Google sheet scheme is most likely too slow for larger amounts of data.

The general principles incorporated into the interactive two-dimensional graph allowed for the Rheo Knee to be progressively shifted between a stable and dynamic mode. Trials conducted using the stable end of the graph resulted in uniform strides with low stance flexion and a noticeable impact as full extension occurred before heel strike. The dynamic end of the graph produced some undesirable stride characteristic. Despite low damping magnitudes in early stance increased the stance flexion angle, many strides were affected by “damped buckling.” This problem could be remedied by raising the state 1 damping magnitudes applied at the dynamic end of the graph’s X-axis.

Similarly the damping magnitudes applied by the dynamic end of the Y-axis should be adjusted. In this case large damping magnitudes resulted in full extension not being reached at the end of the swing phase in trial 9. Lowering these damping magnitudes should allow full extension to be reached, which should also help to eliminate abnormalities in the stride, such as the momentary jumps to state 2 shown in Figure 5-15.

Table 6-1 and Table 6-2 show a revised mapping scheme for converting the position of the yellow marker within the graph into parameter values. The upper bounds of the state 1 damping coefficient and the state 5 damping gain have been adjusted in the hope of elevating the undesired stride characteristics mentioned above. Tests will need to be run on the mapping scheme outlined below to see if further adjustment is needed.

Table 6-1: Revised X-axis Mapping Scheme

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Lower Bound</th>
<th>Zero</th>
<th>Upper Bound</th>
<th>Mapping Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Axis</td>
<td>-100</td>
<td>0</td>
<td>100</td>
<td>X &lt; = 0</td>
</tr>
<tr>
<td>S1_d</td>
<td>100</td>
<td>50</td>
<td>40</td>
<td>50 + 50 * (-x / 100)</td>
</tr>
<tr>
<td>S2_d</td>
<td>10</td>
<td>30</td>
<td>30</td>
<td>30 + 20 * (x / 100)</td>
</tr>
</tbody>
</table>

X-Axis: The horizontal axis of the interactive graph
S1_d: Damping coefficient in state 1
S2_d: Damping coefficient in state 2
Table 6-2: Revised Y-axis Mapping Scheme

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Lower Bound</th>
<th>Zero</th>
<th>Upper Bound</th>
<th>Mapping Function</th>
<th>Y &lt; = 0</th>
<th>Y &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y-Axis</td>
<td>-100</td>
<td>0</td>
<td>100</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>( \alpha_{da} )</td>
<td>0</td>
<td>23</td>
<td>45</td>
<td>( 23 - 23 \times (-y / 100) )</td>
<td>( 23 + 22 \times (y / 100) )</td>
<td></td>
</tr>
<tr>
<td>( S_5_{gain} )</td>
<td>0</td>
<td>2.7</td>
<td>4.0</td>
<td>( 2.7 - 2.7 \times (-y / 100) )</td>
<td>( 2.7 + 1.3 \times (y / 100) )</td>
<td></td>
</tr>
</tbody>
</table>

Y-Axis: The vertical axis of the interactive graph
\( \alpha_{da} \): State 5 Active damping angle
\( S_5_{gain} \): State 5 Damping gain

The current app software can be used to test the revised mapping scheme outline in the tables above. Portions of the software could also be refined and incorporated into a commercial app, particularly the RheoKneeBluetooth and/or GraphView classes. At a minimum, the collected data and aspects of the software architecture contained in this paper can be used to ease the development of future software intended to serve a similar purpose.
7 References


Appendix A – MainActivity Code

```java
package com.ossur.rheodirect;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.TimeZone;

import com.ossur.rheodirect.RheoKneeBluetooth.OnMessageReceivedListener;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.net.Uri;
import android.os.Bundle;
import android.os.PowerManager;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.res.Configuration;

public class MainActivity extends Activity {

    //GUI Components================================
    private ImageView connectedView; // ImageView for connect button
    private ImageView disconnectedView; // ImageView for disconnect button
    private CheckBox vibrateCheckBox; // ImageView for sound toggling button
    private CheckBox soundCheckBox; // ImageView for vibrate toggling button
    private TextView rheoNameView; // ImageView for Rheo Name
    private TextView firmwareView; // ImageView for Firmware Version
    private RheoKneeBluetooth rheoBT; // Class object for communication
    private GraphFragment graphFragment; // Fragment for displaying graph
    private ProgressDialog progress; // Dialog for connection wait
    private Button setButton;
    private Button viewButton;
    private Button logButton;
    private ListView variableListView;

    PowerManager pm;
```
PowerManager.WakeLock wl;

//Message Received Variables====================
private String msg; // String used to hold MSG tag messages
private Float varValue; // used for storing incoming visible variables
    // values
private Short varIndex; // used for storing incoming visible variable
    // indexes
private ArrayList<String> variableCodeList; // used to store list of
    // variable codes on startComm
private ArrayList<Float> variableValueList; // used to store list of
    // variable values on startComm

//For displaying list of variables from Rheo
private ArrayList<String> variableList; // list used to hold both
    // index and value of each variable
private ArrayList<String> variableCodeList;
private ArrayList<Float> variableValueList;

private ArrayAdapter<String> variableAdapter; // used to connect the
    // above list to a list view

private int warningMode; // tracks the warning mode of the Rheo knee
private boolean soundOn; // tracks state of check box for sound
private boolean vibrateOn; // tracks state of check box for vibrate

private static final Integer IS_DISCONNECTED = 0;
private static final Integer IS_CONNECTING = 1;
private static final Integer IS_CONNECTED = 2;
private boolean masterIsConnected;

////////////////////////////////////
/////////////////////////////////
// Possible improvement:
// Description: Sets various variables
// Parameters:
// Returns:
/////////////////////////////////////////////////////////////////////
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //Class object which handles communication with Rheo knee over
    //bluetooth
    rheoBT = new RheoKneeBluetooth(MainActivity.this);

    //Lists and adapters for holding and displaying current Rheo
    //variable states
    variableList = new ArrayList<String>();
    variableAdapter = new ArrayAdapter<String>(this,
                                            R.layout.list_item, variableList);
//Main layout view variables
connectedView = (ImageView) findViewById(R.id.connectView);
disconnectedView = (ImageView) findViewById(R.id.disconnectView);
vibrateCheckBox = (CheckBox) findViewById(R.id.vibrateCheckBox);
soundCheckBox = (CheckBox) findViewById(R.id.soundCheckBox);
rheoNameView = (TextView) findViewById(R.id.rheoNameView);
firmwareView = (TextView) findViewById(R.id.firmwareView);
graphFragment = (GraphFragment) getFragmentManager().
    findFragmentByld(R.id.graphFragment);
setButton = (Button) findViewById(R.id.setButton);
viewButton = (Button) findViewById(R.id.rheoStateButton);
logButton = (Button) findViewById(R.id.autoButton);

//Set various click listeners
connectedView.setOnClickListener(connectedViewListener);
disconnectedView.setOnClickListener(disconnectedViewListener);
vibrateCheckBox.setOnClickListener(vibrateCheckBoxListener);
soundCheckBox.setOnClickListener(soundCheckBoxListener);
setButton.setOnClickListener(setButtonListener);
viewButton.setOnClickListener(viewButtonListener);
logButton.setOnClickListener(logButtonListener);
rheoBT.setOnMessageReceivedListener(
rheoBtOnMessageReceivedListener);
rheoBT.setOnConnectionStateChanged(
rheoBtOnConnectionStateChangedListener);

//Initialization of variables for state of Rheo
vibrateCheckBox.setClickable(false);
soundCheckBox.setClickable(false);
setButton.setClickable(false);
logButton.setClickable(false);
soundOn = false;
vibrateOn = false;
masterIsConnected = false;

//WakeLock will prevent the phone from sleeping
pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
wl = pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My tag");
}
Possible improvement:

Description: Receives the messages sent from the RheoKneeBluetooth. Messages fall into three categories:

- messageS - an English sentence. Most of these are currently ignored. Only the message indicating the Firmware version is handled.
- value and index - the value and index of an individual Rheo variable. This information is sent if the variables are "flagged" for "Showing" or if a command was sent to change the value of a variable in the Rheo Knee.
- varCodes and varValues - This is a list of all variable indices and values which are passed after a startComm is called.

Parameters: See above
Returns: None

private OnMessageReceivedListener rheoBtOnMessageReceivedListener =
new OnMessageReceivedListener()
{
    @Override
    public void onMessageReceived(char[] messageS)
    {
        msg = new String(messageS);
        if(msg.contains("VersionNumber:"))
        {
            firmwareView.setText(firmwareView.getText() + " " + msg.substring(msg.indexOf(':')+1));
        }
    }

    @Override
    public void onMessageReceived(Float value, Short index)
    {
        varValue = value;
        varIndex = index;

        //set the value in the variableValueList to the new value.
        variableValueList.set(index, value);
        variableList.set(index, variableCodeList.get(index) + " = " + Float.toString(variableValueList.get(index)));
        variableAdapter.notifyDataSetChanged();
        setEstimated();
    }

    @Override
    public void onMessageReceived(ArrayList<String> varCodes, ArrayList<Float> varValues)
    {
        variableCodeList = varCodes;
        variableValueList = varValues;
    }
}
```java
for (int index = 0; index < variableCodeList.size(); index++)
{
    variableList.add(variableCodeList.get(index) + " = " + Float.toString(
    variableValueList.get(index)));
}
```

/////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description: Callback indicating the connection status
//      with the Rheo knee
//      --called by RheoKnee Class when ConnectedThread.
//      Run is started
//  Parameters: Integer connectionState - variable indicating
//        connection state
//  Returns: None

private OnConnectionStateChangedListener
rheoBtOnConnectionStateChangedListener =
new OnConnectionStateChangedListener()
{

    @Override
    public void onConnectionStateChanged(Integer connectionState)
    {
        connectionState(connectionState.intValue());
    }
};
```

/////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description: Sets the GUI attributes based on connection state
//      It is also used to display a "Progress" window
//      when connecting Called by rheoBtOnConnectionStateChanged in this activity
//      Sends "STARTCOMM" command after Bluetooth socket is opened.
//  Parameters:  int _connectionState - represents the connection
//        state of the app to the Rheo knee
//  Returns: None

private void connectionState(int _connectionState)
{
    if (_connectionState == IS_CONNECTING)
    {
        connectedView.setImageResource(
```
R.drawable.connected_on);
disconnectedView.setImageResource(R.drawable.disconnected_off);
rheoNameView.setText(rheoNameView.getText() + 
    rheoBT.rheoName);
rheoBT._sendMessage("STARTCOMM\n".getBytes());
progress = ProgressDialog.show(this, "Connecting", 
    "Please Wait", true);
}
else if(_connectionState == IS_CONNECTED)
{
    masterIsConnected = true;
    progress.dismiss();
    vibrateCheckBox.setClickable(true);
    soundCheckBox.setClickable(true);
    setButton.setClickable(true);
    logButton.setClickable(true);
    initializeConnection();
    graphFragment.graphView.drawEstimate = true;
}
else if(_connectionState == IS_DISCONNECTED)
{
    masterIsConnected = false;
    connectedView.setImageResource(R.drawable.connected_off);
    disconnectedView.setImageResource(
        R.drawable.connected_on);
    rheoNameView.setText(getResources().getString(
        R.string.rheo_name_textview));
    firmwareView.setText(getResources().getString(
        R.string.firmware_textview));
    vibrateCheckBox.setClickable(false);
    soundCheckBox.setClickable(false);
    vibrateCheckBox.setChecked(false);
    soundCheckBox.setChecked(false);
    setButton.setClickable(false);
    logButton.setClickable(false);
    graphFragment.graphView.drawEstimate = false;
}

////////////////////////////////////////////////////////////////////
// Possible improvement:
////////////////////////////////////////////////////////////////////
// Description:  Sets the Rheo comm to "NEWMODE" in order to
// use "TESTWARNING" command. It also sets the
// checkboxes, soundOn and vibrateOn, values
// based on the value of 'g6' variable in
// variableValueList. Called after startComm
// sends data.
// Parameters:  None
// Returns:   None
////////////////////////////////////////////////////////////////////
//Initialize variables after startComm sends data
private void initializeConnection()
{
    //set comm to NEWMODE in order to gain access to TESTWARNING
    //command.
rheoBT._sendMessage("NEWMODE\n".getBytes());
setEstimated();

warningMode = variableValueList.get(variableCodeList.indexOf("g6")).intValue();

//if warning modes are off, then turn on both.
if(warningMode == 0)
{
    soundOn = true;
    vibrateOn = true;
    rheoBT._sendMessage("Sn".getBytes());
rheoBT._sendMessage((variableCodeList.indexOf("g6") + "3n").getBytes());
soundCheckBox.setChecked(true);
vibrateCheckBox.setChecked(true);
}
else if(warningMode == 1)
{
    soundOn = true;
    vibrateOn = false;
    soundCheckBox.setChecked(true);
vibrateCheckBox.setChecked(false);
}
else if(warningMode == 2)
{
    soundOn = false;
    vibrateOn = true;
    vibrateCheckBox.setChecked(true);
soundCheckBox.setChecked(false);
}
else if(warningMode == 3)
{
    soundOn = true;
    vibrateOn = true;
    soundCheckBox.setChecked(true);
vibrateCheckBox.setChecked(true);
}
private void setEstimated()
{
    float x = 0;
    float y = 0;

    Float c1Damping = (float) 0;
    Float c2Damping = (float) 0;
    Float swingPoint = (float) 0;
    Float swingResistance = (float) 0;

    //Retrieve c1 and c2 damping
    c1Damping = variableValueList.get(variableCodeList.indexOf("Fq"));
    c2Damping = variableValueList.get(variableCodeList.indexOf("Fr"));

    //Retrieve SwingPoint and SwingResistance
    swingPoint = variableValueList.get(variableCodeList.indexOf("EO"));
    swingResistance = variableValueList.get(
        variableCodeList.indexOf("EP"));

    //Bound swingPoint between 0 and 5.7
    if(swingResistance > 5.7)
        swingResistance = (float) 5.7;

    //=============Set the x coordinate
    //if working in top half of x axis
    if(c1Damping <= 50)
    {
        //if x is larger then graph x-bound
        if( c1Damping < 30)
            x = 1;
        else
            x = (float) (0.5 + 0.5 *(1 - (c1Damping-30)/20));
    }
    //if working in the lower half of the x axis
    else
    {
        //if c2 is less then lower bound
        if(c2Damping < 10)
            c2Damping = (float) 10;

        //Map c1 and c2 to the lower half of the graph
        float c1Scaled = (1 - (c1Damping - 50)/ 50);
        float c2Scaled = ( (c2Damping-10) / 20);

        //if the scaled value of c2 is larger then c1
        //Set x to the lower scaled value plus half the distance
//between the two.
if(c2Scaled < c1Scaled)
{
    x = (float) (0.5 * (c2Scaled + ((c1Scaled - c2Scaled)
                     / 2)));
} else
{
    x = (float) (0.5 * (c1Scaled + ((c2Scaled - c1Scaled)
                     / 2)));
}

//=============Set the y coordinate
float swPointScaled = 0;
float swResistanceScaled = 0;
if(swingPoint >= 23)
    swPointScaled = (float) (0.5 * ((45 - swingPoint) / 22.0));
else
    swPointScaled = (float) (0.5 + 0.5 * (1 - (swingPoint/23)));
swResistanceScaled = (float) (1 - (swingResistance/ 5.4));
if(swResistanceScaled > swPointScaled)
    y = swPointScaled + (swResistanceScaled - swPointScaled);
else
    y = swResistanceScaled + (swPointScaled -
                        swResistanceScaled);

graphFragment.graphView.setEstimated(x, y);

} //////////////////////////////////////////////////////////////////////////////////////////
// Possible improvement:
// Description: Result passed back to the main activity from a
// startActivityForResult called in RheoKneeBluetooth
// class
// Parameters: int requestCode - the code used during the request
// to I.D. it
// int resultCode - the result of the connection.
// int data - not used
// Returns: None
////////////////////////////////////////////////////////////////////////////////////////

public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    rheoBT.onActivityResult(requestCode);
    //String message = Boolean.toString(rheoBT.getbluetoothActive());
}
private OnClickListener viewButtonListener =
    new OnClickListener()
{
    @Override
    public void onClick(View v) {
        AlertDialog.Builder deviceDiaBuilder = new AlertDialog.Builder(MainActivity.this);
        deviceDiaBuilder.setTitle("Variables");
        variableAdapter.notifyDataSetChanged();
        deviceDiaBuilder.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which){
        }
    });
    deviceDiaBuilder.setAdapter(new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
    });
    variableAdapter.notifyDataSetChanged();
    AlertDialog alert = deviceDiaBuilder.create();
    alert.show();
    }
};
// Possible improvement:
// Description: Calculates the position of the set marker in the graphView. It maps the markers position to an intended value for "stance flexion damping", "stance extension damping", "swing terminal resistance", and "terminal swing point angle". The mapping is linear on either side of 0 on both axis.

// ---------------------------------------------------mapping---------------------------------------------------
// X Axis: -100        0          100
//     stance flex damp: 100       50           30
//     stance extend damp: 10       30           30
// Y Axis: -100        0          100
//     swing resistance: 0       2.7          5.4
//     swing point: 0       23           45
// Parameters:
// Returns:

private OnClickListener setButtonListener =
    new OnClickListener()
{

    @Override
    public void onClick(View v)
    {
        int xValue = graphFragment.graphView.xPoint;
        int yValue = graphFragment.graphView.yPoint;

        //constrain xValue and yValue between -100 and 100
        if(xValue > 100)
            xValue = 100;
        if(xValue < -100)
            xValue = -100;
        if(yValue > 100)
            yValue = 100;
        if(yValue < -100)
            yValue = -100;

        float c1Damping = 0;
        float c2Damping = 0;

        float swingResistance = 0;
        float swingPoint = 0;

        if(xValue <= 0)
        {
            c1Damping = (float) (50 + 50 * (-xValue / 100.0));
            c2Damping = (float) (30 + 20 * (xValue / 100.0));
        }
        else
        {
            c1Damping = (float) (50 - 20 * (xValue / 100.0));
            c2Damping = 30;
        }
    }
}
if(yValue >= 0)
{
    swingResistance = (float) (2.7 + 2.7 * (yValue / 100.0));
    swingPoint = (float) (23 + 22 * (yValue / 100.0));
}
else
{
    swingResistance = (float) (2.7 - (2.7 * (-yValue / 100.0)));
    swingPoint = (float) (23 - (23 * (-yValue / 100.0)));
}

rheoBT._sendMessage("S\n".getBytes());
rheoBT._sendMessage((variableCodeList.indexOf("Fq") + " " + Float.toString(c1Damping) + "\n").getBytes());

try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}

rheoBT._sendMessage("S\n".getBytes());
rheoBT._sendMessage((variableCodeList.indexOf("Fr") + " " + Float.toString(c2Damping) + "\n").getBytes());

try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}

rheoBT._sendMessage("S\n".getBytes());
rheoBT._sendMessage((variableCodeList.indexOf("EP") + " " + Float.toString(swingResistance) + "\n").getBytes());

try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    e.printStackTrace();
}

rheoBT._sendMessage("S\n".getBytes());
rheoBT._sendMessage((variableCodeList.indexOf("EO") + " " + Float.toString(swingPoint) + "\n").getBytes());
rheoBT._sendMessage("TESTWARNING\n".getBytes());
public void onClick(View v)
{
    if(!masterIsConnected)
    {
        rheoBT.findDevices();
    }
}
}

private OnClickListener disconnectedViewListener =
    new OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        if(masterIsConnected)
        {
            rheoBT.close();
        }
    }
}

////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  //
//  Description: This is the event handler for the log button
//  It creates a URL used for logging and sends an
//  intent to launch the browser
//  //
//  Parameters: None
//  Returns: None
////////////////////////////////////////////////////////////////////
private OnClickListener logButtonListener =
    new OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        SimpleDateFormat dateFormatGmt = new SimpleDateFormat("dd-MM-yyyy HH:mm");
        dateFormatGmt.setTimeZone(TimeZone.getTimeZone("GMT + 0"));
        String urlString = "+";
        urlString = urlString + "https://script.google.com/macros/s/AKfycbzPb804m_"
        + "NrWd4XVJo9D0Xw9he9zUm9O6o_pZvaPGQXxMQ8gLA/exec?";
        urlString = urlString + "Name=" + rheoBT.rheoName + ";";
urlString = urlString + "Date=" + dateFormatGmt.format(new Date()) + ";

for(int x = 0; x < variableCodeList.size(); x++)
{
    urlString = urlString + variableCodeList.get(x) + "=" + variableValueList.get(x).toString() + ";
}

Intent webIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(urlString));

startActivity(webIntent);

private OnClickListener vibrateCheckBoxListener = new OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        vibrateOn = !vibrateOn;
        updateWarningMode((CheckBox)v);
    }
};

private OnClickListener soundCheckBoxListener = new OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        soundOn = !soundOn;
        updateWarningMode((CheckBox)v);
    }
};

////////////////////////////////////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description:  Used to update warningMode in Rheo knee when a checkbox is changed.
//  Parameters:  CheckBox v - reference to the checkbox which was changed
//  Returns:    None
////////////////////////////////////////////////////////////////////////////////////////////////////
private void updateWarningMode(CheckBox v)
{
    if(!soundOn && !vibrateOn)
    {

v.setChecked(true);
if(v.getText()==getResources().getString(R.string.soundCheck))
    soundOn = true;
else
    vibrateOn = true;

Toast btFailed = Toast.makeText(this, "Cannot Disable Warning System",
    Toast.LENGTH_SHORT);
btFailed.show();
return;
}
else if(!soundOn && vibrateOn)
{
    warningMode = 2;
    rheoBT._sendMessage("S\n".getBytes());
    rheoBT._sendMessage(
        variableCodeList.indexOf("g6") + " 2\n").getBytes();
    rheoBT._sendMessage("TESTWARNING\n".getBytes());
}
else if(soundOn && !vibrateOn)
{
    warningMode = 1;
    rheoBT._sendMessage("S\n".getBytes());
    rheoBT._sendMessage(
        variableCodeList.indexOf("g6") + " 1\n").getBytes();
    rheoBT._sendMessage("TESTWARNING\n".getBytes());
}
if(soundOn && vibrateOn)
{
    warningMode = 3;
    rheoBT._sendMessage("S\n".getBytes());
    rheoBT._sendMessage(
        variableCodeList.indexOf("g6") + " 3\n").getBytes();
    rheoBT._sendMessage("TESTWARNING\n".getBytes());
}

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // Inflate the menu; this adds items to the action bar if it is
    // present.
    getMenuInflater().inflate(R.menu.main, menu);
    return false;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_settings) {

}
return true;

return super.onOptionsItemSelected(item);

@override
public void onPause()
{
    super.onPause();
    if(wl.isHeld())
        wl.release();

    if(graphFragment.graphView.graphThread != null)
        graphFragment.graphView.stopGraphThread();
}

@override
public void onResume()
{
    super.onResume();
    wl.acquire();
    //graphFragment.graphView.startGraphDrawing();
}

@override
public void onDestroy()
{
    super.onDestroy();

    //graphFragment.graphView.stopGraphThread();
    if(masterIsConnected == true)
    {
        rheoBT.close();
    }
}
Appendix B – RheoKneeBluetooth Code

```java
package com.ossur.rheodirect;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.UUID;
import android.app.Activity;
import android.app.AlertDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Handler;
import android.os.Message;
import android.widget.ArrayAdapter;
import android.widget.Toast;

public class RheoKneeBluetooth {

    private BluetoothAdapter adapter;
    private boolean bluetoothActive;
    private boolean waitingForResult = false;
    private int connectionState;
    private static final int REQUEST_ENABLE_BT = 1;
    private Context activity;
    private ArrayList<String> deviceArray;
    private ArrayAdapter<String> deviceAdapter;

    public String rheoName;
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    private ConnectThread connectRheoBT = null;
    private ConnectedThread connectedRheoBT = null;
    private static final int MESSAGE_READ = 1;
    private static final int FLOAT_READ = 2;
    private static final int SHORT_READ = 3;
    private static final int CONNECTING_STATE_READ = 4;
    private static final int VARIABLE_CODE_LIST_READ = 5;
    private static final int VARIABLE_VALUE_LIST_READ = 6;
```
private static final Integer IS_DISCONNECTED = 0;
private static final Integer IS_CONNECTING = 1;
private static final Integer IS_CONNECTED = 2;

//------------------Constructor and getMethods------------------

////////////////////////////////////////////////////////////////////////
// Possible improvement: May need to change this to the onCreate
// method inherited from Fragment.
//
// Description: Constructor for RheoKneeBluetooth class. This
// class manages a Bluetooth(BT) communication
// between an Android app and the Rheo Knee. This
// method retrieves a BluetoothAdapter and insures
// that the device has BT capability.
//
// Parameters: act -- holds the context of the implementing Activity.
// Returns: None
////////////////////////////////////////////////////////////////////////
public RheoKneeBluetooth(Context act)
{
    //set local "activity" to context of calling activity
    activity = act;

    // ArrayList and Adapter for displaying discovered devices
    //in AlertDialog
deviceArray = new ArrayList<String>();
deviceAdapter = new ArrayAdapter<String>(activity,
                                        R.layout.list_item,
                                        deviceArray);

    //set the connection state to false
    connectionState = IS_DISCONNECTED;

    //retrieve a Bluetooth adapter
    adapter = BluetoothAdapter.getDefaultAdapter();

    //check to see if the device supports Bluetooth
    if(adapter == null)
    {
        bluetoothActive = false;
    }
    else// if device supports Bluetooth
    {
        // check to see if the Bluetooth is disabled
        if(!adapter.isEnabled())
        {
            //if Bluetooth is disabled send intent message to
            // prompt enabling
            Intent enableBtIntent = new Intent(
                BluetoothAdapter.ACTION_REQUEST_ENABLE);

            // this line uses the main_activity to receive the
            //callback result.
            ((Activity) activity).startActivityForResult(
                enableBtIntent, REQUEST_ENABLE_BT);
        }
    }
}
public boolean getBluetoothActive() {
    return bluetoothActive;
}

public int getRequestEnableBT() {
    return REQUEST_ENABLE_BT;
}

public String getRheoName() {
    return rheoName;
}
//--------------Device Discovery and Pairing--------------

/////////////////////////////////////////////////////////////////
// Possible improvement:
// Description: This is a callback used to inform the implementing
// Activity of changes to the connection state.
// It allows for progress"bars" to be used during
// connection.
// Interface: onConnectionStateChangedListener
// Parameters: Integer connectionState - specifies
// the connection state
// as DISCONNECTED,
// CONNECTING, OR
// CONNECTED
// Returns: None
/////////////////////////////////////////////////////////////////
private OnConnectionStateChangedListener onConnectionStateChangedListener;

public interface OnConnectionStateChangedListener
{
    public void onConnectionStateChanged(Integer connectionState);
}

public void setOnConnectionStateChanged(
    OnConnectionStateChangedListener listener)
{
    onConnectionStateChangedListener = listener;
}

/////////////////////////////////////////////////////////////////
// Possible improvement:
// Description: Is called from the implementing Activity which
// should forward the onActivityResult returned by
// the startActivityForResult called in this classes
// constructor. The result indicates if BT was
// enabled or not.
// Parameters: int result -- the result passed to the
// implementing Activity.
// Returns: None
/////////////////////////////////////////////////////////////////
public void onActivityResult(int result)
{
    if(result == Activity.RESULT_OK)
        bluetoothActive = true;
    else if(result == Activity.RESULT_CANCELED)
        bluetoothActive = false;
    else
        // error
        bluetoothActive = false;
waitingForResult = false;
}

////////////////////////////////////////////////////////////////////////
// Possible improvement:
//
// Description: - Registers a mReceiver BroadcastReceiver which
// receives intent messages from the
// BluetoothAdapter when a device is discovered.
// The message contains information about the
// discovered device.
// - Builds and displays an AlertDialog which
// displays an AdapterList of discovered devices.
// The AlertDialog will call the connectDevice
// method with the proper MAC address if one is
// selected from the list.
// - Starts the BluetoothAdapter discovery process.
// - Cancel Button clears AdapterList devices and
// cancels discovery
// Parameters: None
// Returns: None
////////////////////////////////////////////////////////////////////////
public void findDevices()
{
    //activateBluetooth();

    ((Activity) activity).registerReceiver(mReceiver, filter);

    AlertDialog.Builder deviceDiaBuilder = new
            AlertDialog.Builder(activity);
    deviceDiaBuilder.setTitle("Select a Device");

    //Clear the deviceArray
    deviceArray.clear();
    deviceAdapter.notifyDataSetChanged();

    deviceDiaBuilder.setNegativeButton(
            "Cancel", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    //Cancel discovery because it will slow down the connection
                    adapter.cancelDiscovery();
                }
            });

    deviceDiaBuilder.setAdapter(
            deviceAdapter, new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    String deviceMac = deviceArray.get(which).toString();
```java
int indexOf = deviceMac.indexOf('
');
rheoName = ' ' + deviceMac.substring(0, indexOf);

indexOf = deviceMac.indexOf('
') + 1;
deviceMac = deviceMac.substring(indexOf);
connectDevice(deviceMac);

deviceAdapter.notifyDataSetChanged();
AlertDialog alert = deviceDiaBuilder.create();
alert.show();
adapter.startDiscovery();

```
private void connectDevice(String deviceMAC)
{
    // Only create a new ConnectThread object if it does not exist.
    if(connectRheoBT == null)
    {
        BluetoothDevice mdevice =
            adapter.getRemoteDevice(deviceMAC);
        connectRheoBT = new ConnectThread(mdevice);
    }

    if(!connectRheoBT.isAlive())
        connectRheoBT.start(); // calls the run() method from 
        // the thread
}

Class ConnectThread extends Thread

// Possible improvement: Try to create secureRFcommSocket

// Description: - Attempts to create a BT socket with given MAC 
// address
// - Method run -- Creates new thread which runs 
// to completion after the following
// -- Cancels discovery process
// -- Attempts to connect to device 
// (blocking method)
// -- Calls manageConnectedSocket
// if connection was successful
// - Method cancel -- Closes the socket

Methods: Constructor
// Parameters: BluetoothDevice device - Device 
// to connect to.
// Returns:
//
// run
// Parameters: None 
// Returns: None 

// cancel
// Parameters: None 
// Returns: None
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    //private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        setName("ConnectThread");
        // Use a temporary object that is later assigned to
        // mmSocket, because mmSocket is final
        BluetoothSocket tmp = null;
        //mmDevice = device;

        // Get a BluetoothSocket to connect with the given
        // BluetoothDevice
        try {
            // MY_UUID is the app's UUID string, also used by
            // the server code
            tmp = device.createInsecureRfcommSocketToServiceRecord(
                    UUID.fromString("00001101-0000-1000-8000-00805F9B34FB"));
        } catch (IOException e) {
            mmSocket = tmp;
        }
    }

    //Cancels discovery and tries to open connection with
    //bluetooth
    public void run() {
        //Cancel discovery because it will slow down the
        //connection
        adapter.cancelDiscovery();

        try {
            // Connect the device through the socket. This will
            // block until it succeeds or throws an exception
            mmSocket.connect();
        } catch (IOException connectException) {
            // unable to connect; close the socket and get out
            try {
                mmSocket.close();
            } catch (IOException closeException) {
            }

            Toast btFailed = Toast.makeText(
                    activity, "Bluetooth failed",
                    Toast.LENGTH_LONG);
            btFailed.show();
            return;
        }

        // Do work to manage the connection (in a separate thread)
        RheoKneeBluetooth.this.manageConnectedSocket(mmSocket);
        connectRheoBT = null;
    }

    //Will cancel an in-process connection, and close the socket
public void cancel()
{
    try{
        mmSocket.close();
    } catch(IOException e){}
}

//--------------Manage Bluetooth Connection-------------------

private OnMessageReceivedListener onMessageReceivedListener;
public interface OnMessageReceivedListener
{
    public void onMessageReceived(char[] messageS);
    public void onMessageReceived(Float value, Short index);
    public void onMessageReceived(
        ArrayList<String> _variableCodes,
        ArrayList<Float>   _variableValues);
}

public void setOnMessageReceivedListener(
    OnMessageReceivedListener listener)
{
    onMessageReceivedListener = listener;
}

public void _sendMessage (byte[] msg)
{
if(!connectedRheoBT == null)
    connectedRheoBT.write(msg);
else
{
    Toast btFailed = Toast.makeText(
        activity, "Not paired to device",
        Toast.LENGTH_LONG);
    btFailed.show();
}

/////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description: - Creates ConnectedThread object
//                - Starts new thread for monitoring BT input
//                buffer.
//  Parameters: - BluetoothSocket mmSocket - the connected
//                socket to a device
//  Returns: - None
/////////////////////////////////////////////////////////////
private void manageConnectedSocket(BluetoothSocket mmSocket)
{
    connectedRheoBT = new ConnectedThread(mmSocket);
    connectedRheoBT.start();
}

/////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description: Receive the callback from the handler. It is
//               used to receive byte arrays passed to a
//               handler by the BT input thread. The handler
//               allows for safe transfer of data between
//               threads. The field Message.what indicate the
//               data being sent. The message is forwarded
//               to the onMessageReceivedListener in the
//               implementing Activity.
//  Parameters: Message msg - sent from the connection
//               thread by way of handler. Contains object holding
//               a message from the Rheo
//  Returns: Boolean - indicates if handler is still
//            needed. Return false if handling
//            is still needed.
/////////////////////////////////////////////////////////////
private Handler mHandler = new Handler(new Handler.Callback() {
    byte[] msgHolder;
    Float value = (float) 0;
    Short index = 0;
    ArrayList<String> variableCodes;
    ArrayList<Float> variableValues;

    @Override
    public boolean handleMessage(Message msg)


//Arrays.fill(msgHolder, (byte) 0);

if (msg.what == MESSAGE_READ)
{
    msgHolder = (byte[])msg.obj;
    onMessageReceivedListener.onMessageReceived(
        new String(msgHolder).toCharArray());
}
else if (msg.what == FLOAT_READ)
{
    value = (Float) (msg.obj);
}
else if (msg.what == SHORT_READ)
{
    index = (Short) (msg.obj);
    onMessageReceivedListener.onMessageReceived(
        value, index);
}
else if (msg.what == CONNECTING_STATE_READ)
{
    //Inform implementing of Connection State changed
    connectionState = ((Integer) msg.obj);

    onConnectionStateChangedListener.
        onConnectionStateChanged(connectionState);
}
else if (msg.what == VARIABLE_CODE_LIST_READ)
{
    variableCodes = (ArrayList<String>) msg.obj;
}
else if (msg.what == VARIABLE_VALUE_LIST_READ)
{
    variableValues = (ArrayList<Float>) msg.obj;
    onMessageReceivedListener.onMessageReceived(
        variableCodes, variableValues);
}
return false;
}};
// Class ConnectedThread extends Thread

// Possible improvement:  
// - Brake down the logic in the run method into multiple methods.
// - Change from endless loop to looping based on variable accessible from outside the thread.

// Description:  
// - Attempts to open an Input and Output stream to the device
// - Method run:  
//   -- Calls startComm
//   -- Then endless loop on new thread
//   {
//     -- Remove all '\n' characters at beginning of buffer (blocking method)
//     -- Read from mmInStream until '\n' and place in messageBuffer
//     -- Copy messageBuffer to byteArray
//     -- Check to see if it is a "$\n" tag
//       -- Remove 6 bytes from mmInStream and place in messageBuffer
//     -- Copy messageBuffer to byteArray
//     -- Unpack the 6 byteArray using unpackVariableValue method, returns Float and Short
//     -- Pass Float to main thread using mHandler(FLOAT_READ)
//     -- Pass Short to main thread using mHandler(SHORT_READ)
//     -- Else if message is not "$\n" tag
//       -- Remove all '\n'
//       -- Read from mmInStream until '\n' and place in messageBuffer
//     -- Copy messageBuffer to byteArray
//     -- Pass byteArray to main thread using mHandler(MESSAGE_READ)
//   }
// - Method startComm:  
//   -- Handles the initialization of variables sent from Rheo on connection.
// - Method ReadToEol:  
//   -- Read input stream to next '\n'.
// - Method readIndexValue:  
//   -- Reads index and value for variables sent from Rheo.(Six bytes)
// - Method write:  
//   -- Used to write bytes to BT mmOutStream.
// - Method cancel:  
//   -- Closes the socket

// Methods:  
// Constructor
// Parameters:  BluetoothSocket - socket connected to a device.
// Returns:  
// run
// Parameters:  None
// Returns:  None
// Handler:  Passes connecting status
// Passes variable initialization
// Passes connected status
// Passes incoming bytes to
// main thread with handler

startComm:
Parameters: None
Returns: None

ReadToEol:
Parameters: byteArray - An array for holding incoming bytes
Returns: int - the number of bytes read

readIndexValue:
Parameters: byteArray - An array for holding incoming bytes
Returns: int - the number of bytes read

write
Parameters: bytes - byte array to send over BT to Rheo Knee
Returns: None

cancel
Parameters: None
Returns: None

private class ConnectedThread extends Thread
{
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;
    public boolean scanInput = true;
    public boolean threadDone = false;

    private ArrayList<String> inVariableCodes = new ArrayList<String>();
    private ArrayList<Float> inVariableValues = new ArrayList<Float>();

    // Watches the inputStream. Need to add a return value to run() which is thrown if an IOException is thrown
    public ConnectedThread(BluetoothSocket socket)
    {
        setName("ConnectedThread");
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp
        // object because member streams are final
        try{
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) {}//

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
}
public void run()
{
    byte[] byteArray = new byte[500]; // buffer store
    //for the stream
    ByteBuffer messageBuffer = ByteBuffer.allocate(500);
    byte[] buffByte = new byte[1];
    byte[] EOF = "\n".getBytes();
    int bCount = 0; // bytes returned from read();
    byte[] varValue = new byte[4];
    byte[] varIndex = new byte[2];

    //Send message to mainActivity of connecting state.
    mHandler.obtainMessage(CONNECTING_STATE_READ,
        bCount, -1, IS_CONNECTING).sendToTarget();
    startComm();
    mHandler.obtainMessage(VARIABLE_CODE_LIST_READ,
        bCount, -1, inVariableCodes).sendToTarget();
    mHandler.obtainMessage(VARIABLE_VALUE_LIST_READ,
        bCount, -1, inVariableValues).sendToTarget();

    //Send message to mainActivity of connected state.
    mHandler.obtainMessage(CONNECTING_STATE_READ,
        bCount, -1, IS_CONNECTED).sendToTarget();

    // Keep listening to the InputStream until an
    //exception occurs
    while(scanInput)
    {
        try {
            sleep(10);
        } catch (InterruptedException e2) {
            e2.printStackTrace();
        }

        bCount = readToEol(byteArray);

        //check to see if it is a value response.
        if( new String(byteArray,0,bCount).equals("S"))
        {
            //Read next six bytes
            bCount = readIndexValue(byteArray);

            //Unpack six bytes to short and float
            unpackVariableValue(byteArray, varValue, varIndex);

            //Create object from primitive types
            Float objF_varValue = Float.valueOf(
                ByteBuffer.wrap(varValue).order(
                    ByteOrder.LITTLE_ENDIAN).getFloat());
            Short objS_varIndex = Short.valueOf(
                ByteBuffer.wrap(varIndex).order(
                    ByteOrder.LITTLE_ENDIAN)).
mHandler.obtainMessage("FLOAT_READ", bCount, -1, objF_varValue).sendToTarget();

try {
    sleep(10);
} catch (InterruptedException e2) {
    e2.printStackTrace();
}

mHandler.obtainMessage("SHORT_READ", bCount, -1, objS_varIndex).sendToTarget();

//if it MSG. Read MSG until EOF
else if (new String(byteArray, 0, bCount).equals("MSG")) {
    bCount = readToEol(byteArray);
    mHandler.obtainMessage("MESSAGE_READ", bCount, -1, byteArray).sendToTarget();
} else {
    //remove all EOF characters from the
    //inputStream
    do {
        try {
            mmInStream.read(buffByte, 0, 1);
        } catch (IOException e) {
            connectedRheoBT.scanInput = false;
            break;
        }
    } while (buffByte[0] == EOF[0]);
    messageBuffer.put(buffByte);
    for (bCount = 0; buffByte[0] != EOF[0]; bCount++) {
        try {
            mmInStream.read(buffByte, 0, 1);
        } catch (IOException e) {
            connectedRheoBT.scanInput = false;
            break;
        }
        //Put the new byte into the messageBuffer
        messageBuffer.put(buffByte[0]);
    }
    //Tag with "error"
Arrays.fill(byteArray, (byte) 0);
System.arraycopy("error: ".getBytes(),
    0, byteArray, 0, 7);

//Clear the byteArray then
//reads the number of bytes from the
//messageBuffer that were just written to it.
messageBuffer.flip();

messageBuffer.get(byteArray, 7, bCount);
messageBuffer.clear();
try {
    if(mmSocket.isConnected())
        mmInStream.skip(mmInStream.available());
} catch (IOException e) {
    connectedRheoBT.scanInput = false;
    e.printStackTrace();
    break;
}
mHandler.obtainMessage(MESSAGE_READ, bCount,
    -1, byteArray).sendToTarget();
}

public void startComm()
{
    int bCount = 0; //the number of bytes read
    byte[] byteArray = new byte[500]; // array for holding
    // send bytes
    byte[] bNumVars = new byte[2];
    short sNumVars; // number of variables being passed
    // on startComm
    byte[] varValue = new byte[4]; // used to retrieve
    // float bit-pattern
    // from unpack
    byte[] varIndex = new byte[2]; // used to retrieve
    // short bit-pattern
    // from unpack

    //Read to EOL
    bCount = readToEol(byteArray);
    bCount = readToEol(byteArray);
    bCount = readToEol(byteArray);

    //If SVNI was read
    if( new String(byteArray,0,bCount).equals("SVNI"))
    {
        //read to EOL
        bCount = readToEol(byteArray);

        String numVar = new String(byteArray, 0, bCount);
// Read number of variables
sNumVars = Short.parseShort(numVar);

// Read in list of variable codes
for (Short index = 0; index < sNumVars; index++)
{
    bCount = readToEol(byteArray);
    inVariableCodes.add(new String(
        byteArray, 0, bCount));
}

// Read in list of variable values
for (Short index = 0; index < sNumVars; index++)
{
    bCount = readToEol(byteArray);
    // Could add a check here to see if 'S' was received. Also could be that some 'S' msg are not followed by needed six bytes
    // read in the six next bytes
    bCount = readIndexValue(byteArray);
    // Unpack six bytes to short and float
    unpackVariableValue(
        byteArray, varValue, varIndex);
    // Create object from primitive types
    Float objF_varValue = Float.valueOf(
        ByteBuffer.wrap(varValue).order(
            ByteOrder.LITTLE_ENDIAN)
        .getFloat());
    inVariableValues.add(objF_varValue);
}

// Read MSG - "VersionNumber..." and "Connected"
for (int index = 0; index < 2; index++)
{
    bCount = readToEol(byteArray);
    if (new String(byteArray, 0, bCount).equals("MSG"))
    {
        bCount = readToEol(byteArray);
        mHandler.obtainMessage(
            MESSAGE_READ, bCount, -1, 
            byteArray)
        .sendToTarget();
        try {
            sleep(100);
        } catch (InterruptedException e2) {
            e2.printStackTrace();
        }
    }
}
else// if SVNI fails it could be the device is not an RHEO {
    //Remove leading '\n' then read to next '\n'
    private int readToEol(byte[] byteArray) {
        ByteBuffer messageBuffer = ByteBuffer.allocate(500);
        byte[] buffByte = new byte[1];
        byte[] EOF = "\n".getBytes();
        int bCount = 0; // bytes returned from read();
        //remove all EOF characters from the inputStream
        do{
            try {
                mmInStream.read(buffByte, 0, 1);
            } catch (IOException e) {
                connectedRheoBT.scanInput = false;
                break;
            }
        } while (buffByte[0] == EOF[0]);
        messageBuffer.put(buffByte);
        //remove characters from the inputStream and add
        //them to the messageBuffer until EOF
        for (bCount = 0; buffByte[0] != EOF[0]; bCount++) {
            try {
                mmInStream.read(buffByte, 0, 1);
            } catch (IOException e) {
                connectedRheoBT.scanInput = false;
                break;
            }
            //Put the new byte into the messageBuffer
            messageBuffer.put(buffByte[0]);
        }
        //Clear the byteArray then
        //reads the number of bytes from the messageBuffer
        //that were just written to it.
        messageBuffer.flip();
        Arrays.fill(byteArray, (byte) 0);
        messageBuffer.get(byteArray, 0, bCount);
        messageBuffer.clear();
        return bCount;
    }
}
// reads the six bytes
private int readIndexValue(byte[] byteArray) {
    ByteBuffer messageBuffer = ByteBuffer.allocate(500);
    byte[] buffByte = new byte[1];
    int bCount = 0;

    // read six bytes from the inputs
    for (bCount = 0; bCount < 6; bCount++) {
        try {
            mmInStream.read(buffByte, 0, 1);
        } catch (IOException e) {
            connectedRheoBT.scanInput = false;
            break;
        }

        // Put the new byte into the messageBuffer
        messageBuffer.put(buffByte[0]);
    }

    // Clear the byteArray then
    // reads the number of bytes from the messageBuffer
    // that were just written to it.
    messageBuffer.flip();
    Arrays.fill(byteArray, (byte) 0);
    messageBuffer.get(byteArray, 0, bCount);
    messageBuffer.clear();
    return bCount;
}

// Call this from the main activity to send data to the remote device
public void write(byte[] bytes) {
    try {
        mmOutStream.write(bytes);
    } catch (IOException e) {};
}

// Call this from the main activity to shutdown the connection
public void cancel() {
    try {
        mmSocket.close();
    } catch (IOException e) {};
}
private void unpackVariableValue(byte[] byteSequence, byte[] value, byte[] index) {
    byte[] shortIndex = new byte[2];
    byte[] floatValue = new byte[4];
    Arrays.fill(shortIndex, (byte) 0);
    Arrays.fill(floatValue, (byte) 0);
    byte holder1;
    byte holder2;
    holder1 = byteSequence[0];
    holder2 = byteSequence[1];
    shortIndex[1] = (byte) ((holder2 >>> 5) & 0x03);
    shortIndex[0] = (byte) (((holder2 << 3) & 0x80) | (holder1 & 0x7f));
    holder1 = byteSequence[2];
    floatValue[3] = (byte) ((holder2 << 4) | ((holder1 >>> 3) & 0x0f));
    holder2 = byteSequence[3];
    floatValue[2] = (byte) (holder1 << 5) | ((holder2 >>> 2) & 0x1f);
    holder1 = byteSequence[4];
    floatValue[1] = (byte) (holder2 << 6) | ((holder1 >>> 1) & 0x3f);
    holder2 = byteSequence[5];
    floatValue[0] = (byte) (holder1 << 7) | (holder2 & 0x7f);
    System.arraycopy(floatValue, 0, value, 0, 4);
    System.arraycopy(shortIndex, 0, index, 0, 2);
public void close() {
    connectedRheoBT.cancel();
    ((Activity) activity).unregisterReceiver(mReceiver);
    connectionState = IS_DISCONNECTED;
    onConnectionStateChangedListener.onConnectionStateChanged(connectionState);
}

// Possible improvement:

// Description: Used to clean up onDestroy. Must remember to
// include it in onPause or onDestroy of
// mainActivity

// Parameters:  None
// Returns:     None

//
package com.ossur.rheodirect;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Align;
import android.graphics.Point;
import android.graphics.RadialGradient;
import android.graphics.Shader;
import android.graphics.drawable.Drawable;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;
import android.widget.ImageView;

public class GraphView extends SurfaceView
    implements SurfaceHolder.Callback
{
    private static final String TAG = "GraphView"; // for logging errors

    private Activity activity; // calling activity
    public GraphThread graphThread;

    private Line xAxis; // x axis of graph
    private Line yAxis; // y axis of graph
    private Point position; // point on graph
    private Point estimated;

    private Paint majorAxisPaint;
    private Paint minorAxisPaint;
    private Paint positionPaint;
    private Paint estimatePaint;
    private Paint backgroundPaint;
    private Paint underWash;
    private Paint textPointPaint;
    private Paint textAxisPaint;

    private int majorAxisWidth;
    private int minorAxisWidth;
    private int positionRadius;

    private int xAxisDistance;
    private int xAxisStart;
private int xAxisEnd;
private int yAxisDistance;
private int yAxisStart;
private int yAxisEnd;

private int dynamicLabelX;
private int dynamicLabelY;

private int stableLabelX;
private int stableLabelY;

private int stanceLabelX;
private int stanceLabelY;

private int swingLabelX;
private int swingLabelY;

private SurfaceHolder mHolder = null;

// used for display coordinates only
public int xPoint;
public int yPoint;

public int graphViewWidth;
public int graphViewHeight;
public boolean drawEstimate = false;

public GraphView(Context context, AttributeSet attrs) {
    super(context, attrs);
    activity = (Activity) context; // store reverence to MainActivity

    // register SurfaceHolder.Callback listener
getholder().addCallback(this);

    // initialize axis and position
    xAxis = new Line();
    yAxis = new Line();
    position = new Point();
    estimated = new Point();

    // construct Paints for drawing axis and position
    majorAxisPaint = new Paint();
    minorAxisPaint = new Paint();
    positionPaint = new Paint();
    estimatePaint = new Paint();
backgroundPaint = new Paint();
underWash = new Paint();
textPointPaint = new Paint();
textAxisPaint = new Paint();
//setEstimated((float) 0.5, (float) 0.5);

////////////////////////////////////////////////////////////////////////////////////////////////////
// Possible improvement:
//
// Description: Calculates and sets different sizes and positions
// of GUI elements
// Parameters:
// Returns:
////////////////////////////////////////////////////////////////////////////////////////////////////
//Called when view size has changed, including when it first added
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    super.onSizeChanged(w, h, oldw, oldh);

    graphViewWidth = w;
    graphViewHeight = h;

    majorAxisWidth = 4;
    minorAxisWidth = 1;
    positionRadius = w / 36;

    xAxisDistance = h / 2;
    xAxisStart = 0;
    xAxisEnd = w ;

    yAxisDistance = w / 2;
    yAxisStart = 0;
    yAxisEnd = h;

    xAxis.start.set(xAxisStart, xAxisDistance);
    xAxis.end.set(xAxisEnd, xAxisDistance);

    yAxis.start.set(yAxisDistance, yAxisStart);
    yAxis.end.set(yAxisDistance, yAxisEnd);

    position.set(w/2, h/2);
    positionRadius = h/20;

    xPoint = 0;
    yPoint = 0;

    underWash.setColor(Color.BLACK);
    backgroundPaint.setShader(new LinearGradient(0, 0, w, h, getResources().getColor(R.color.background_holo_light),
                                           getResources().getColor(R.color.ossur_blue),
                                           Shader.TileMode.CLAMP));

    majorAxisPaint.setStrokeWidth(majorAxisWidth);
majorAxisPaint.setColor(Color.BLACK);
minorAxisPaint.setStrokeWidth(minorAxisWidth);
minorAxisPaint.setColor(Color.BLACK);

positionPaint.setStrokeWidth(positionRadius);
positionPaint.setShader(new RadialGradient(position.x, position.y, positionRadius,
Color.WHITE, getResources().getColor(R.color.ossur_yellow), Shader.TileMode.CLAMP));
estimatePaint.setStrokeWidth(positionRadius / 2);
estimatePaint.setShader(new RadialGradient(estimated.x, estimated.y, positionRadius/2,
Color.WHITE, Color.GRAY, Shader.TileMode.CLAMP));
textPointPaint.setTextSize(w/25);
textPointPaint.setAntiAlias(true);
textAxisPaint.setTextSize(w/30);
textAxisPaint.setAntiAlias(true);

stableLabelX = (int) (graphViewWidth/35.0);
stableLabelY = (int) (graphViewHeight - textAxisPaint.getTextSize()*0.5);
dynamicLabelX = (int) (graphViewWidth - graphViewWidth/35.0);
dynamicLabelY = (int) (textAxisPaint.getTextSize()*1.3);
stanceLabelX = (int) (graphViewWidth - graphViewWidth / 30);
stanceLabelY = (int) graphViewHeight / 2;
swingLabelX = (int) graphViewWidth / 2;
swingLabelY = (int) graphViewHeight - graphViewHeight / 30;

@Override
public boolean onTouchEvent(MotionEvent e) {
    int action = e.getAction();
    if(action == MotionEvent.ACTION_DOWN ||
        action == MotionEvent.ACTION_MOVE)
    {
        position = new Point((int) e.getX(),(int) e.getY());
        positionPaint.setShader(new RadialGradient(position.x, position.y, positionRadius,
            Color.WHITE, getResources().getColor(R.color.ossur_yellow), Shader.TileMode.CLAMP));
    }
}
R.color.assur_yellow, Shader.TileMode.CLAMP));

xPoint = (int)(200 * (position.x/(float)graphViewWidth)) - 100;
yPoint = (int)(-200 * (position.y/(float)graphViewHeight)) + 100;

if(xPoint < -60)
{
    textPointPaint.setTextAlign(Align.LEFT);
}
else if(xPoint > 60)
{
    textPointPaint.setTextAlign(Align.RIGHT);
}

return true;

////////////////////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description:
//  Parameters:
//  Returns:
////////////////////////////////////////////////////////////////////////////////////
@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width, int height)
{

}

////////////////////////////////////////////////////////////////////////////////////
//  Possible improvement:
//  Description: Starts a new thread for updating the graph GUI
//  Parameters:  holder - SurfaceHolder
//  Returns:
////////////////////////////////////////////////////////////////////////////////////
@Override
public void surfaceCreated(SurfaceHolder holder)
{
    mHolder = holder;
    graphThread = new GraphThread(holder);
    graphThread.start();
}
@Override
public void surfaceDestroyed(SurfaceHolder holder)
{
    mHolder = null;
    boolean retry = true;
    while(retry)
    {
        try
        {
            graphThread.join(); // wait for cannonThread to finish
            retry = false;
        }
        catch (InterruptedException e)
        {
            Log.e(TAG, "Thread interrupted", e);
        }
    }
}

private void drawGraph(Canvas canvas)
{
    //Draw back ground colors
    canvas.drawRect(0, 0, graphViewWidth, graphViewHeight, underWash);
    canvas.drawRect(0, 0, graphViewWidth, graphViewHeight, backgroundPaint);

    //Draw minor axis on Y
    for(float x = 8; x < graphViewHeight;
        x=(float) (x+(graphViewHeight-16)/10.0))
    {
        canvas.drawLine(xAxis.start.x+8, x, xAxis.end.x-8, x, minorAxisPaint);
    }

    //Draw minor axis on X
    for(float x = 8; x < graphViewWidth;
        x=(float) (x+(graphViewWidth-16)/10.0))
```java
    canvas.drawLine(x, yAxis.start.y+8, x, yAxis.end.y-8, minorAxisPaint);
}

//Draw major axis
canvas.drawLine(xAxis.start.x+8, xAxis.start.y, xAxis.end.x-8, xAxis.end.y, majorAxisPaint);
canvas.drawLine(yAxis.start.x, yAxis.start.y+8, yAxis.end.x, yAxis.end.y-8, majorAxisPaint);

//Draw the estimated point
if(drawEstimate)
    canvas.drawCircle(estimated.x, estimated.y, positionRadius, estimatePaint);

//Draw stable and dynamic labels
textAxisPaint.setColor(Color.RED);
textAxisPaint.setTextAlign(Align.LEFT);
canvas.drawText(getResources().getString(R.string.stable), stableLabelX, stableLabelY, textAxisPaint);

textAxisPaint.setTextAlign(Align.RIGHT);
canvas.drawText(getResources().getString(R.string.dynamic), dynamicLabelX, dynamicLabelY, textAxisPaint);

//Draw the axis labels
textAxisPaint.setColor(getResources().getColor(R.color.ossur_yellow));
textAxisPaint.setTextAlign(Align.RIGHT);
canvas.drawText(getResources().getString(R.string.stance_axis), stanceLabelX, stanceLabelY, textAxisPaint);

canvas.save();
canvas.rotate(-90, swingLabelX, swingLabelY);
textAxisPaint.setTextAlign(Align.LEFT);
textAxisPaint.setTextSkewX((float)-.25);
canvas.drawText(getResources().getString(R.string.swing_axis), swingLabelX, swingLabelY, textAxisPaint);
canvas.restore();

//Draw the position point
canvas.drawCircle(position.x, position.y, positionRadius, positionPaint);
canvas.drawText(getResources().getString(R.string.point_position), xPoint, yPoint), position.x, position.y, textPointPaint);
```
public void setEstimated(float x, float y)
{
    estimated.x = (int) (x * graphViewWidth);
    estimated.y = (int) (y * graphViewHeight);
    estimatePaint.setShader(new RadialGradient(
        estimated.x, estimated.y, positionRadius,
        Color.WHITE, Color.GRAY, Shader.TileMode.CLAMP));
}

public void stopGraphThread()
{
    if(graphThread != null)
        graphThread.setRunning(false);
    boolean retry = true;
    while(retry)
    {
        try
        {
            graphThread.join();//wait for cannonThread to finish
            retry = false;
        }
        catch (InterruptedException e)
        {
            Log.e(TAG, "Thread interrupted", e);
        }
    }
}

public void stopGraphDrawing()
{
    if(graphThread != null)
        graphThread.setDrawing(false);
}
public void startGraphThread()
{
    if (mHolder != null)
    {
        graphThread = new GraphThread(mHolder);
        graphThread.start();
    }
}

private class GraphThread extends Thread
{
    private SurfaceHolder surfaceHolder;
    private boolean threadIsRunning = true;
    private boolean threadIsDrawing = true;

    public GraphThread (SurfaceHolder holder)
    {
        surfaceHolder = holder;
        setName("GraphThread");
    }

    public void setRunning(boolean running)
    {
        threadIsRunning = running;
    }

    public void setDrawing(boolean drawing)
    {
        threadIsDrawing = drawing;
    }

    @Override
    public void run()
    {
        while (threadIsRunning)
        {
            Canvas canvas = null;
            if (threadIsDrawing)
            {
                //some how need to wait for the canvas
            }
        }
    }
}
try{
    canvas = surfaceHolder.lockCanvas();
    synchronized(surfaceHolder)
    {
        drawGraph(canvas);
    }
}
finally{
    if(canvas != null)
    {
        surfaceHolder.unlockCanvasAndPost(canvas);
    }
}

///////////////////////////////////////////////////////////////
// Possible improvement:
//
// Description: Class used to draw the line within the graphView
//
// Parameters:
// Returns:

private class Line{
	public Point start = new Point(); // start Point --
		// (0,0) by default
	public Point end = new Point(); // end Point --
		// (0,0) by default
} // end inner-class Line

Appendix D – GraphFragment Code

```java
package com.ossur.rheodirect;

import android.app.Fragment;
import android.media.AudioManager;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class GraphFragment extends Fragment {
    public GraphView graphView;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);
        View view = inflater.inflate(R.layout.fragment_graph, container, false);
        graphView = (GraphView) view.findViewById(R.id.graphView);

        return view;
    }

    // set up volume control once Activity is created
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
    }

    @Override
    public void onPause() {
        super.onPause();
        if(graphView.graphThread != null)
            graphView.stopGraphThread();
    }

    @Override
    public void onResume() {
        super.onResume();
        if(graphView != null)
            graphView.startGraphThread();
    }
}
```
Appendix E – Gantt Chart