DESIGN AND VALIDATION OF A FALL DETECTION APPLICATION FOR iOS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Biomedical Engineering

by

Connor Lewis Mosley

March 2016

COMMITTEE MEMBERSHIP

TITLE:	Design and Validation of a Fall Detection Application for iOS
AUTHOR:	Connor Lewis Mosley
DATE SUBMITTED:	March 2016
COMMITTEE CHAIR:	Robert Szlavik, Ph.D. Associate Professor of Biomedical Engineering
COMMITTEE MEMBER:	Robert Crockett, Ph.D. Associate Professor of Biomedical Engineering
COMMITTEE MEMBER:	Lily Laiho, Ph.D. Associate Professor of Biomedical Engineering

ABSTRACT

Design and Validation of a Fall Detection Application for iOS

Connor Lewis Mosley

Despite significant preventative efforts, falls continue to be a major source of morbidity and mortality among the elderly. Additionally, the fear of falling can be a major obstacle to independent living for otherwise self-sufficient individuals. This fear is significantly heightened in individuals who have sustained a fall and often results in self-imposed restrictions on mobility and exercise, causing weakening in these individuals and further exacerbating the danger. Much time has been spent developing alert systems in an attempt to mitigate these problems. Unfortunately these systems typically involve dedicated monitoring centers and therefore often come with substantial upfront and recurring costs. This thesis proposes a solution to these problems by implementing fall detection and alert capabilities on a smartphone, devices that are quickly becoming ubiquitous in today's society. This solution has the potential to quell the fears of many elderly people and their families, while allowing them to maintain their independence at little expense. Detailed herein is the process of designing, developing, and validating this fall detection application. The final application was written in Objective-C for iOS and tested on an iPhone.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1 INTRODUCTION

The inspiration for this project came from a senior design project undertaken by two classmates and myself. For that project, we were tasked with creating a remote fall detection system. The system had to be capable of automatically detecting when a user had sustained a fall, and then contacting help for the user. Our implementation consisted of an Arduino Duemilanove microcontroller, outfitted with an accelerometer and an XBee radio module, to be worn by the user. The device measured accelerations in all three axes and compared each to a preset threshold value specific to that axis. The system would declare a fall if any of the accelerations exceeded their preset threshold. The device would sound an alarm when a fall was detected, and the user was given a certain amount of time to abort the distress call via a button on the device. If the abort timer elapsed without the user pressing the button, then the device used the XBee radio module to send a message to its counterpart radio attached to a computer within approximately 100 feet. The computer would then display a message declaring the user had sustained a fall. This was meant to serve as a proof of concept, with the computer acting as a surrogate for a base station attached to a telephone landline [39].

This thesis is a continuation of the same concept, but with the implementation being an application on a smartphone. There is ample rationale for choosing a smartphone as the platform for this project. Smartphones already

contain the necessary hardware required for a fall detection system. These devices are equipped with high performance accelerometers, as well as the obvious telephone functionality. The devices themselves are relatively small and designed to be carried on the user's person. Additionally, smartphones are becoming ubiquitous in today's society, especially in the aging baby boomer generation [66, 67, 56, 57]. Smartphones could provide a cheaper and more convenient platform for fall detection than current expensive monitoring systems or dedicated devices.

1.1 Project Justification

In the United States, falls are the leading cause of injury related visits to the emergency room, and the primary etiology of accidental deaths in persons over the age of 65. Falls account for 70% of accidental deaths in persons 75 years of age and older. Additionally, more than 90% of hip fractures occur as a result of falls. Approximately 9,500 deaths annually in older Americans are associated with falls. Those who survive these falls experience significant morbidity, with hospital stays almost twice as long as for patients admitted for other reasons. Five to 15% of falls result in a major injury, including head trauma, soft tissue injuries, fractures, and dislocations [46].

Many falls do not result in significant physical injury, but the psychological impact can still be great. These falls often result in a fear of falling which leads to self-imposed restrictions on movement; further decreasing mobility and physical

strength, leading to an increased risk of falling [46, 43, 50]. This becomes a vicious cycle that many elderly people find difficult to break free from, and one that can quickly lead to some type of assisted care living. Many elderly people dread admission into a residential home and the loss of independence associated with it [41].

Studies have uncovered several risk factors associated with falls [47, 68, 69]. Many falls are linked to a person's physical condition or medical problems [69]. Restricted physical activity in the past five years and poor leg strength were found to be predictors of falls [68]. Cognitive impairment and the use of medications have also been identified as important risk factors associated with falls [47]. Certain medical problems, such as a history of stroke, postural hypotension, and disabilities in the lower extremities, can increase the risk of recurrent falls [47]. Graafmans declared mobility impairment as the strongest risk factor and potentially the easiest to change in a prevention program that includes exercise [47]. Exercise has the potential to improve balance and muscle strength, and also may increase bone mineral density [47].

In light of these findings, a wearable fall-detection device may not only save lives in cases of emergency, but also give elderly people the confidence and security to continue living on their own and to stay active. In this way, such a device can play a large role in preventing falls from occurring in the first place.

2 BACKGROUND

This chapter is intended to provide background information for this project. The following sections discuss the current solutions to the problem of falls in the elderly and fall detection in general, the concept behind this project and the requirements for its success, as well as the iOS platform. Additionally, a cloud communications platform named Twilio is discussed, as it was utilized in the final project design.

2.1 Current Solutions

There have been many attempts to address the problem of falls among the elderly. The most well established solutions revolve around emergency alert systems where the user must activate the system's emergency services. There has also been a significant amount of research into methods of automatically detecting falls. The combination of automatic fall detection within alert systems is a fairly recent advancement that is yet to gain traction in the marketplace.

2.1.1 Alert Systems

There are a number of alert systems currently on the market, offering various options for different prices. Some examples of established companies are Life Alert, Alert1, and Philips Lifeline. Most offer 24-hour monitoring centers, which users can connect to via button press on a pendant or bracelet. Some

companies offer devices capable of fall detection, allowing for the monitoring center to be notified if the user is unable to press the alert button. Several companies even offer options for emergency services on the go, usually through a small device that works similar to a cell phone. Each of these companies requires users to pay a monthly subscription in order to use their services. These subscriptions range from about $25 per month to over $55 per month, depending on what kind of services and options are selected. Additionally, customers often have to purchase the pendant or device that will be used for the fall detection or alert system [58, 2, 53].

2.1.2 Fall Detection Research

A considerable amount of research has been placed on fall detection methods and their efficacy [35, 37, 38, 49, 51, 54, 55, 65]. This research has been focused in several distinct areas. One of the main areas explored has been fall detection via sensors attached to the subject's body. These systems often utilize accelerometers, gyroscopes, or a combination of the two [35, 37, 38, 51, 54, 65]. With these sensor-based systems, a common method for declaring a fall is to monitor for movement exceeding what would be experienced in activities of daily living (ADL). This is done by determining and setting threshold levels for either accelerations (for accelerometers) or angular velocities (for gyroscopes) and declaring a fall when these thresholds have been exceeded [37, 38, 51]. Other research has focused on determining the user's orientation to declare a

fall. For this method, attached sensors are used primarily to detect whether the user is lying down or not [54, 55]. This type of system can be supplemented with floor sensors to enhance accuracy. There has also been research into image processing of video signals to detect falls. This typically either involves detecting a lying posture using scene analysis, or detecting abrupt movements using vector analysis. [55, 65]

Each of these methods has their own challenges and shortcomings. For systems using attached sensors to monitor for sudden and abrupt movement, the main challenge is in regards to determining the threshold values for distinguishing a fall. The difficulty arises in trying to set the thresholds low enough so that even the smallest fall will register as such, but high enough so that the system wont produce false alarms during normal daily activities. While failure to detect a fall could be a catastrophic failure, a system that produces lots of false positives is also deeply flawed. In fact, a high rate of false alarms has been found to be the main reason for rejection of fall detection systems by users [55].

Determining falls based on user orientation can also be problematic particularly in the case of falls that don't result in the user lying on the floor; for instance if a fall occurs on a staircase or up against a wall. Additionally, these systems are not ideal for monitoring falls in the elderly, as their sleep patterns are often irregular and the body orientation sensors cannot be worn while lying down. Coupling these systems with floor sensors helps to mitigate the problems, but

greatly increases the cost of the system and limits the usable area to rooms containing the floor sensors.

Image processing methods have their own set of drawbacks, as well. They require a substantial initial investment and have challenges in regards to unpredictable lighting conditions and camera placement, which must ensure that the subject remains in the field of view at all times. Additionally, there are privacy concerns, considering the need for cameras throughout the individual's living space [55]. Furthermore, a complex system of cameras, powered by mains electricity, could pose a potential problem in the case of a power outage. One can imagine such an event being a likely time for an elderly individual to sustain a fall. Therefore, the system would need to have a robust, and likely expensive, backup system of generators or batteries.

2.2 Concept

As the baby boomer generation continues to age, there will be an ever-increasing need for fall-alert systems for the elderly. As discussed, currently available systems have some significant drawbacks; not the least of which is expensive upfront and recurring costs. A system based on a technologically suitable platform and increasingly accessible device has the potential to correct these drawbacks and make the system available to more users. This is the rationale behind choosing smartphones as a platform, as it satisfies both these conditions.

Smartphones are quickly becoming ubiquitous in our society. According to Nielsen, over 70% of all Americans own a smartphone [56]. Among individuals aged 55 to 64, 61.1% currently own a smartphone, and this number is increasing [56]. In fact, according to Pew Research, U.S. smartphone ownership has increased by 29% since 2011 [66]. Even seniors 65 and older have seen an 8% increase in ownership in just the last year [66].

The devices themselves are particularly well suited for the task of fall detection and alert. Each device comes equipped with high precision, on-board accelerometers and gyroscopes. These sensors give the phone the capability of detecting rapid movements associated with falls, as well as recognizing orientation if necessary. Additionally, smartphones have the obvious built in capability to act as a phone, as well as send text messages and accomplish other forms of communication.

2.3 Project Requirements

The following sections detail the requirements for the project in terms of functionality and performance.

2.3.1 Functionality

The requirements for functionality have been broken down into two categories. First, the minimum functionality required: these are the functions the device must perform in order to be an acceptable fall detection solution. Second,

desired functionality: this includes any extra functionality that would be included in an ideal solution.

At a minimum, the system must be able to detect a fall and alert an outside party. The app must enable the phone to perform these actions without user input. The user must be notified of an impending distress call via alarm and/or vibration. Additionally, the user must be given time and a means by which to cancel the automatic distress call in the event of a false alarm. The final minimum requirement is that the app must be able to remain running while in the user's pocket, or contained in a case, where touch input to the screen may or may not be received.

Beyond the bare minimum requirements for functionality, a number of goals were set for desired functionality. Ideally, the user would be able to input his/her own list of emergency contacts into the app. In the event of a fall, the app would automatically notify the user's emergency contacts, as well as emergency medical services (EMS). The user's contacts would be notified via SMS or other text based messaging. This would decrease the likelihood of a missed connection and free up the telephone capability for EMS services.

Another desired function is for the app to provide a means of contacting EMS upon the user's request. This would allow the user to quickly call for help in the event of any emergency that was not registered as a fall by the phone. This function would need to be easily accessible to the user while also requiring

explicit selection, so as not to be accidentally triggered without the user's consent or knowledge.

Additionally, it would be ideal if the application could run in the background. This would mean that once fall detection is initiated the app would continue to monitor for falls, even while the user is using other apps. Moreover, this would allow the user to place the phone in the lock state while the app continues to run.

2.3.2 Performance

The following sections detail and discuss a reasonable level of performance that was set as a base requirement for this project. There are two primary metrics for measuring the performance of a fall detection system: sensitivity and specificity. The system must meet the requirements for these metrics, laid out in the following sections, to be considered a success. Additionally, the longevity (battery life) is also an important measure of performance when dealing with a battery powered, mobile detection system.

2.3.2.1 Sensitivity

The sensitivity of a fall detection system is defined as the percentage of real falls that are declared as such. Written as a formula [55]:

$$Sensitivity = \frac{TP}{TP + FN} \tag{1}$$

where:

- True positive (TP): a fall occurs, the device detects it
- False negative (FN): a fall occurs but the device does not detect it

For this system to be considered a success, the tested sensitivity must be 100%. That is, any conceivable fall scenario, with the potential to result in incapacitation, must be registered as a fall by the application. A fall that renders the user unable to contact help and is not regarded as a fall by the application would constitute a catastrophic failure of the system.

2.3.2.2 Specificity

Specificity is the percentage of non-fall events that are regarded as such, and therefore not declared as falls. Written as a formula [55]:

$$Specificity = \frac{TN}{TN + FP} \tag{2}$$

where:

- True negative (TN): a normal (non-fall) movement is performed, the device does not declare a fall
- False positive (FP): the device announces a fall, but one did not occur

A perfect system would have a specificity of 100%. Specificity is essentially the false alarm rate. The fewer false alarms the better. Studies have shown that a high false alarm rate is a major contributor to users rejecting fall detection devices [55]. Fewer than five false alarms per day would be acceptable for this

application. Furthermore, the device must show 100% specificity while performing the most common ADL, such as walking, sitting, bending over, etc.

2.3.2.3 Sensitivity vs Specificity

With a threshold-based system, there is some inevitable trade off between sensitivity and specificity. The lower the fall threshold is set, the less chance there is of a fall going undetected; however the chance of false alarms increases. Conversely, with a higher fall threshold there is a decreased chance of false alarms but an increased chance that a fall can occur and register below the set threshold. A threshold must be selected so that a balance is struck between maximizing sensitivity while also keeping false alarms to a minimum.

While both sensitivity and specificity are important, sensitivity takes precedence. Sensitivity has this higher priority because an undetected fall is a more critical error than a false positive. If a false positive occurs, the likely outcome is that the user manually overrides the system and aborts any distress messages or calls. The worst-case scenario is that the user does not catch the false alarm and erroneous distress messages and calls are sent out. This is obviously undesirable and could cause undue stress, but it is not life threatening. On the other hand, a false negative, or undetected fall, would force the user to manually activate the distress call. In the event of the user being unconscious or otherwise unable to operate the phone, this would constitute a catastrophic failure of the system and result in the user being left without help.

2.3.2.4 System Longevity

With a system such as this, where the user relies on a battery-powered device for monitoring, it is important that the device be able function for an extended period of time without needing to be recharged. Obviously, the longer the battery can last, the better. An extended operation time of 12 hours without recharge was set as a minimum requirement. This would allow for continuous monitoring throughout most of an elderly individual's waking day.

2.4 iOS Platform

The iOS operating system was developed by Apple to run on their proprietary devices, such as the iPhone. From a conceptual standpoint the operating system is composed of the following four abstraction layers from lowest level of abstraction to highest: Core OS layer, Core Services layer, Media layer, and Cocoa Touch layer. This operating system manages the device hardware and provides the technologies required to implement native apps [16].

The term "native app" refers to apps that are installed physically on the device, and therefore always available to the user; this is in contrast to web apps. These native apps are built using the iOS system frameworks and the Objective-C programming language and run directly on iOS. A framework is a directory that contains a dynamic shared library and the resources (such as header files,

images, and helper apps) needed to support that library. Apple delivers most of its system interfaces in these special packages [16].

2.4.1 iOS's Layered Architecture

At the highest level, iOS acts as an intermediary between the underlying hardware and the apps that run on that hardware. This means that apps don't communicate directly with the hardware, but rather they communicate through a set of well-defined system interfaces. This simplifies the app writing process and allows apps to work consistently on devices with different hardware capabilities [16].

The implementation of iOS technologies can be viewed as a set of abstraction layers. The lower layers contain fundamental services and technologies. The upper layers build upon the lower layers and provide more sophisticated services. When writing apps for iOS, it is prudent to utilize higher-level frameworks whenever possible. These higher levels are there to provide object-oriented abstractions for lower-level constructs, making it easier and less laborious to write code, as potentially complex features are encapsulated. Situations do arise where the programmer has to use lower-level frameworks if they require features not exposed by the higher-level frameworks [16].

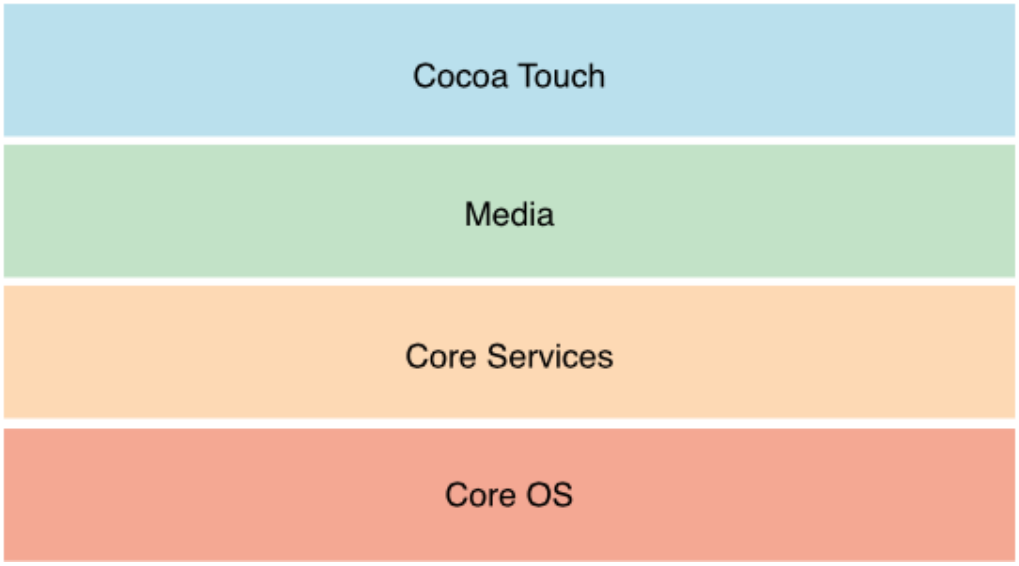

Figure 1 – The abstraction layers of iOS; the top corresponds to the highest level of abstraction, with the bottom being the lowest level of abstraction [16].

2.4.1.1 Cocoa Touch Layer

The Cocoa Touch layer contains many key frameworks for building iOS applications. These frameworks help define the appearance of the app as well as provide basic app infrastructure and support for key technologies such as multitasking, touch-based input, push notifications, and many high-level system services [16].

One very key technology contained within this layer is storyboards. Storyboards allow the developer to design the entire user interface graphically in one place. This allows for viewing of all the views and view controllers and how they work together [16].

2.4.1.2 Media Layer

The media layer contains the graphics, audio, and video technologies used to implement multimedia experiences within an app [16].

2.4.1.3 Core Services Layer

The Core Services layer contains fundamental system services for apps. Key among these frameworks are the Core Foundation and Foundation frameworks, which define the basic types that all apps use. This layer also contains individual technologies to support features such as location, iCloud, social media, and networking. This layer also includes some high-level features such as block objects, which are essentially anonymous functions packaged together with the data that the function operates on [16].

2.4.1.4 Core OS Layer

The Core OS layer contains most of the low-level features that most other technologies are built upon. Even if a developer doesn't use these technologies directly, they are most likely being used by other frameworks [16].

2.4.2 Object Oriented Programming and Objective-C

The Objective-C language is a computer programming language designed to enable sophisticated object-oriented programming and it is the main language native iOS apps are written in. It is defined as a small but powerful set of

extensions to the ANSI C language [21, 22]. These extensions, however, make object-oriented programming in Objective-C sufficiently different from procedural programming in ANSI C [21, 22, 24].

Most object-oriented environments consist of several parts: an object-oriented programming language, a library of objects (e.g. Objective-C application frameworks collectively known as Cocoa), a suite of development tools, and a runtime environment [21].

Object orientation provides a concrete grouping between the data and the operations that can be performed with the data – in effect giving the data behavior. It groups operations and data into modular units called objects and lets the developer combine objects into structured networks to form a complete program. Every object has both state (data) and behavior (operations on data). Objects and object interactions are the basic elements of design [21, 22, 24].

Object-oriented programming languages don't lose any of the virtues of structures and functions from procedural programming in C – they go a step further by adding a unit of abstraction at a higher level, a unit that hides the interaction between a function and its data. Objects completely encapsulate their data, effectively hiding it and allowing the user to think of them solely in terms of their behavior. The hidden data structure unites all the functions that share access to it. So an object is more than a collection of random functions; it's a bundle of related behaviors that are supported by shared data [22].

Object-oriented programming environments typically come with class libraries. There are well over two hundred classes in the Cocoa libraries. Typically a group of library classes work together to define a partial program structure; these classes constitute a software framework or kit [21, 22].

The developer uses these system frameworks in several ways. First and foremost, frameworks are used by initializing and arranging instances of framework classes within a program. The developer can also define subclasses of framework classes. Lastly, newly defined classes can work together with classes defined in the framework [21, 22, 24].

2.4.3 iOS Design Patterns and Techniques

In iOS, the system frameworks provide critical infrastructure for apps and in most cases are the only way to access the underlying hardware. These frameworks use many specific design patterns and assume that the developer is familiar with them [15].

The most important design patterns iOS developers must know are:

- ***Model-View-Controller*** – this design pattern governs the overall structure of the app
- ***Delegation*** – this design pattern facilitates the transfer of information and data from one object to another

- ***Target-Action*** – this design pattern translates user interactions with buttons and controls into code that the app can execute
- ***Block Objects*** – the developer uses block objects to implement callbacks and asynchronous code
- ***Sandboxing*** – all iOS apps are placed in sandboxes to protect the system and other apps. The structure of the sandbox affects the placement of the app's files and has implications for data backups and some app-related features

2.4.3.1 Model-View-Controller

The Model-View-Controller design pattern (MVC) is a high-level pattern in that it concerns itself with the global architecture of an application and classifies objects according to the general roles they play in an application. The MVC design pattern considers there to be three types of objects: model objects, view objects, and controller objects. MVC defines the roles that these types of objects play in the application and their lines of communication. The three types of objects are separated from each other only by abstract boundaries and communicate with each other across those boundaries [8].

Model objects encapsulate data and basic behaviors. These objects hold the apps data and define the logic that manipulates that data. Any data that is part of the persistent state of the application (whether stored in files or databases) should reside in the model objects once the data is loaded into the

application. Ideally, a model object has no explicit connection to the user interface used to present and edit it [8].

View objects present information to the application user. These objects know how to display, and might allow users to edit, the data from the application's model. The view shouldn't be responsible for storing the data it is displaying. View objects tend to be reusable and configurable, and they provide consistency between applications. The UIKit framework in Cocoa Touch defines a large number of view objects and provides many of them in the Interface Builder library. A view should ensure that it is displaying the model correctly and therefore it usually needs to know about changes to the model. However, since model objects should not be tied to specific view objects, they need a generic way of indicating that they have changed. This is where controller objects come in [8].

Controller objects tie the model to the view by acting as an intermediary between the two types of objects within an application. Controllers are often in charge of making sure the views have access to the model objects that they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the lifecycles of other objects [8].

A view controller is a controller that concerns itself mostly with the view layer. View controllers are traditional controller objects in the MVC design pattern, but they also do much more. They "own" the interface (the views) and

their primary responsibilities are to manage that interface and communicate with the model. Whenever an iOS app displays a user interface, the displayed content is managed by a view controller or a group of view controllers coordinating with each other. Therefore, view controllers provide the skeletal framework on which the developer builds the app [8, 31, 32].

A view controller manages a set of views – a discreet portion of the app's user interface. The view controller acts as the central coordinating agent for this view hierarchy, handling exchanges between the views and any relevant controller or data objects. Action methods concerned with the data displayed in a view are typically implemented in a view controller [8].

2.4.3.2 Target-Action

Cocoa Touch uses the target-action mechanism for communication between a control and another object. The role of a control on a user interface is to interpret the intent of the user and instruct some other object to carry out that request. Target-action provides the mechanism by which an event (ex: button click) is translated into an instruction that is specific to the application. The target is the receiving object, and is usually an instance of a custom class. The action is the message that the control sends to the target. The target, being the object that is interested in the user event, is the one that imparts significance to that event [8, 15].

2.4.3.3 Blocks

In Objective-C, a class defines an object that combines data with related behavior. However, sometimes it makes sense just to represent a single task or unit of behavior, rather than a collection of methods. Blocks are a language level feature added to Objective-C, which allow the developer to create distinct segments of code that can be passed around to methods or functions as if they were values [15, 24].

Blocks are Objective-C objects that have the ability to capture values from the enclosing scope, making them similar to closures or lambdas in other programming languages. Blocks can also take arguments and return values just like methods and functions. Additionally, you can pass blocks as arguments to methods or functions [24].

Blocks can be used to simplify concurrent tasks. A block represents a distinct unit of work, combining executable code with optional state captured from the surrounding scope. This makes it ideal for asynchronous invocation; rather than having to figure out how to work with low-level mechanisms like threads, the developer can simply define the tasks using blocks and then let the system perform those tasks as processor resources become available. They are also commonly used for callbacks, defining the code to be executed when a task completes [9, 24].

2.4.3.4 Anatomy of iOS Applications (Core App Objects)

UIKit provides the infrastructure for all apps, but it is the developer's custom objects that define the specific behavior of the application. Apps consist of a handful of specific UIKit objects that manage the event loop and the primary interactions with iOS. Through a combination of subclassing, delegation, and other techniques, the developer modifies the default behavior defined by UIKit to implement the app [15].

In addition to customizing the UIKit objects, the developer is also responsible for providing or defining other key sets of objects. The largest set of objects is the app's data objects, the definition of which is entirely the responsibility of the developer. Another responsibility is to provide a set of user interface objects, of which UIKit provides numerous classes to help [15].

From the time the app is launched by the user, to the time it exits, the UIKit framework manages much of the app's core behavior. At the heart of the app is the UIApplication object, which receives events from the system and dispatches them to the custom code for handling [15].

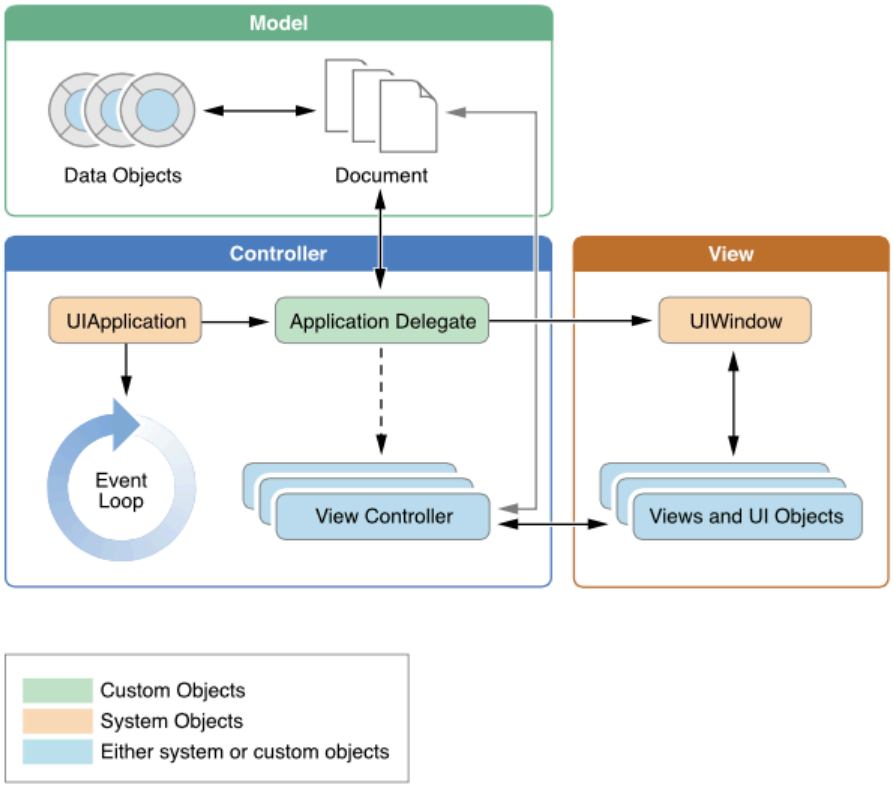

Figure 2 – Key objects in an iOS app [15].

Figure 2 above shows the key objects in every iOS app and their basic connections to each other. The figure is also a good illustration of the model-view-controller design pattern. The following sections detail the roles each of the key objects play within an application.

2.4.3.4.1 UIApplication Object

The developer uses the UIApplication object essentially as is – that is, without subclassing it. This is a controller object that manages the event loop and coordinates other high-level app behaviors. The custom app-level logic resides in the app delegate object, which works in tandem with this object [4, 15].

2.4.3.4.2 App Delegate Object

The app delegate is a custom object created at app launch time, usually by the UIApplicationMain function. The primary job of this object is to handle state transitions within the app. For example it is responsible for launch-time initialization and handling transitions to and from the background. The app delegate can also be used to handle other app-related events. If the UIApplication object doesn't handle an event, it dispatches the event to the app delegate for processing [4, 15].

2.4.3.4.3 Documents and Data Model Objects

Data model objects store the app's content and are specific to that app. Apps can also use document objects to manage some or all of their data model objects [15].

2.4.3.4.4 View Controller Objects

View controller objects manage the presentation of the app's content on screen. A view controller manages a single view and its collection of subviews [15].

2.4.3.4.5 UIWindow Object

A UIWindow object coordinates the presentation of one or more views on a screen. Most apps only have one window, which presents content on the main

screen, but apps may have an additional window for content displayed on an external display. To change the content of an app, a view controller is used to change the views displayed in the corresponding window. The window itself is never replaced. In addition to hosting views, windows work with the UIApplication object to deliver events to the views and view controllers [15].

2.4.3.4.6 View, Control, and Layer Objects

Views and controls provide the visual representation of the app's content. A view is an object that draws content in a designated area and responds to events within that area. Controls are a specialized type of view responsible for implementing familiar interface objects such as buttons, text fields, and toggle switches [15].

Layer objects are actually data objects that represent visual content. Views use layer objects intensively behind the scenes to render their content. Custom layer objects can also be added to the interface to implement complex animations and other types of sophisticated visual effects [15].

2.4.3.5 The App Lifecycle

The following sections describe the application lifecycle.

2.4.3.5.1 The Main Function

Every computer program must have an entry point. For C-based applications, that entry point is the main function. This holds true for iOS apps, however, the developer does not write the main function. The main function is generated automatically by Xcode (Apple's integrated development environment) as part of any basic project. The developer should almost never change the implementation of the main function provided by Xcode.

The job of the main function is hand control off to the UIKit framework. The UIApplicationMain function handles this process by creating the core objects of the app, loading the user interface from the available storyboard files, calling the custom code for some initial setup, and putting the app's run loop into motion. It is the developer's responsibility to provide the storyboard files and the custom initialization code [4, 6, 15, 27].

2.4.3.5.2 The Main Run Loop

An app's main run loop processes all user-related events. The UIApplication object sets up the main run loop at launch time and uses it to process events and handle updates to the view-based interfaces. The main run loop executes on the app's main thread, ensuring that user-related events are processed serially in the order in which they were received [4, 9, 15].

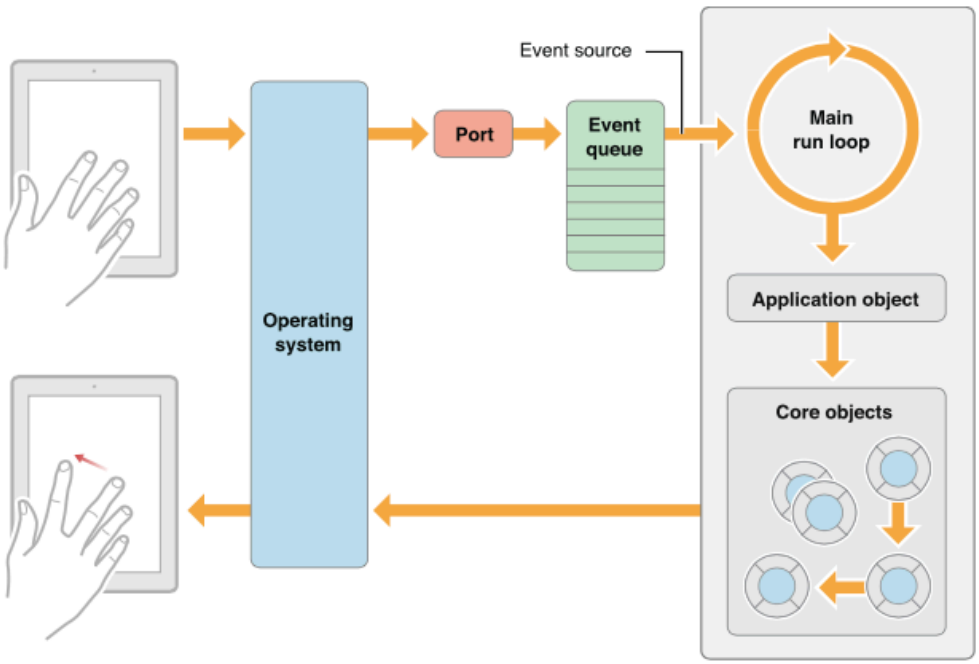

Figure 3 – How events are processed in the main run loop [15].

Figure 3 above shows the architecture of the main run loop and how user events result in actions taken by the app. As a user interacts with the device, events related to those interactions are generated by the system and delivered to the app via a special port set up by UIKit. Events are queued internally by the app and dispatched on-by-one to the main run loop for execution [15].

The UIKit object is the first object to receive the event and make the decision about what needs to be done. A touch event is usually dispatched to the main window object, which in turn dispatches it to the view in which the touch occurred. Other events might take slightly different paths through various app objects. Many types of events can be delivered in an iOS app, a lot of which are delivered using the main run loop. Some events are sent to a delegate object or passed to a block provided by the developer [15].

2.4.3.5.3 Execution States for Apps

At any given moment, an app is in one of the states listed in table 1 below.

Table 1 – Application States [4].

Not Running	The app has not been launched or was running but was terminated by the system.
Inactive	The app is running in the foreground but is currently not receiving events. (It may be executing other code though.) An app usually stays in this state only briefly as it transitions to a different state.
Active	The app is running in the foreground and is receiving events. This is the normal mode for foreground apps.
Background	The app is in the background and executing code. Most apps enter this state briefly on their way to being suspended. However, an app that requests extra execution time may remain in this state for a period of time. In addition, an app being launched directly into the background enters this state instead of the inactive state.
Suspended	The app is in the background but is not executing code. The system moves apps to this state automatically and does not notify them before doing so. While suspended, an app remains in memory but does not execute any code. When a low-memory condition occurs, the system may purge suspended apps without notice to make more space for the foreground app.

The system moves the app from state to state in response to actions happening throughout the system. For example, when the user presses the home button, a phone call comes in, or any of several other interruptions occurs, the currently running apps change state in response. Figure 4 shows the paths that an app takes when moving from state to state.

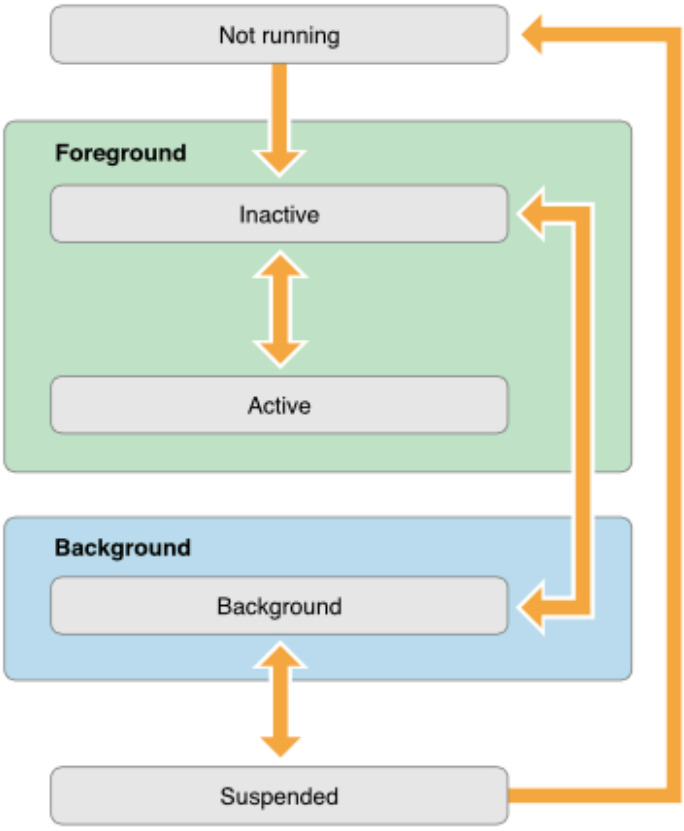

Figure 4 – State changes in iOS [4].

2.4.3.5.4 Background Execution

When a user is not actively using an app, the system moves it to the background state. For many apps, the background state is only a brief stop on the way to being suspended. Suspending apps is a way of improving battery life and allowing the system to devote important resources to the current foreground app. Most apps can move to the suspended state easily, however, there are legitimate reasons for apps to continue running in the background. Some of these reasons include tracking the user's position, playing audio, or downloading content in the background [4, 15].

There are three techniques offered by iOS for background execution. First, apps that start a short task in the foreground can ask for time to finish that task when the app moves to the background. Second, apps that initiate downloads in the foreground can hand off management of those downloads to the system, thereby allowing the app to be suspended or terminated while the download continues. Third, apps that need to run in the background to support specific types of tasks can declare their support for one or more background execution modes [4, 15].

Apps must request specific permission to run in the background without being suspended. In iOS, only specific app types are allowed to run in the background. Apps that are permitted background execution are:

- Apps that play audible content to the user while in the background
- Apps that record audio content from the background
- Apps that keep users informed of their location or otherwise track the user's location at all times
- Apps that support Voice over Internet Protocol (VoIP)
- Apps that need to download and process new content regularly
- Apps that receive regular updates from external accessories

Apps that implement these services must not only declare the services they support, but also use the system frameworks to implement the relevant aspects of those services. Declaring the services lets the system know which services the

app uses, but in some cases it is the system frameworks that actually prevent the application from being suspended [4].

2.4.4 Survey of Relevant iOS Frameworks

The following sections briefly detail some of the iOS frameworks relevant to this fall detection application. Each of the described frameworks was used in some capacity during the development of the application.

2.4.4.1 Cocoa Touch Frameworks

The following sections describe frameworks that reside in the Cocoa Touch layer.

2.4.4.1.1 Address Book UI Framework

The Address Book UI framework is an Objective-C programming interface that is used to display standard system interfaces for creating new contacts and for editing and selecting existing contacts. The framework simplifies the work needed to display contact information in the app and makes sure that the app uses the same interfaces as other apps, ensuring consistency across the platform [16].

2.4.4.1.2 Message UI Framework

The Message UI framework provides support for composing email or SMS messages from within the app. The support for composition consists of a view controller interface that is presented in the app. The fields of this view controller can be populated programmatically to set the recipients, subject, body, content, and any attachments to be included with the messages. The user, however, must manually send the message; iOS does not allow for this to be done programmatically [16].

2.4.4.1.3 UIKit Framework

The UIKit framework provides crucial infrastructure for implementing graphical, event-driven apps in iOS. UIKit provides support for the following [16]:

- Basic app management and infrastructure, including the app's main run loop
- User interface management, including support for storyboards and nib files
- A view controller model to encapsulate the contents of the user interface
- Objects representing the standard system views and controls
- Support for handling touch and motion-based events
- Support for a document model that includes iCloud integration
- Graphics and windowing support, including support for external displays
- Multitasking support
- Printing support

- Support for customizing the appearance of standard UIKit controls
- Support for text and web content
- Cut, copy, and paste support
- Support for animating user interface content
- Integration with other apps on the system through URL schemes and framework interfaces
- Accessibility support for disabled users
- Support for the apple Push Notification service
- Local notification scheduling and delivery
- PDF creation
- Support for custom input views that behave like the system keyboard
- Support for creating custom text views that interact with the system keyboard
- Support for sharing content through email, Twitter, Facebook, and other services

In addition to providing the fundamental code for apps, UIKit also incorporates support for some device-specific features, such as the following [16]:

- The built-in camera
- The user's photo library
- Device name and model information
- Battery state information
- Proximity sensor information

- Remote control information from attached headsets

2.4.4.2 Media Layer Frameworks

The following sections describe the frameworks of the media layer and the services they offer.

2.4.4.2.1 AV Foundation Framework

The AV Foundation framework provides a set of Objective-C classes for playing, recording, and managing audio and video content. This framework is used to integrate media capabilities seamlessly into an app's interface [16].

2.4.4.2.2 Core Audio

Core Audio is a family of frameworks that provides native support for handling audio. These frameworks support the generation, recording, mixing, and playing of audio in apps [16].

2.4.4.2.3 Quartz Core Framework

The Quartz Core framework contains the Core Animation interfaces. Core Animation is an advanced compositing technology that makes it easy to create view-based animations that are fast and efficient. The compositing engine takes advantage of the underlying hardware to manipulate a view's contents efficiently and in real time [16].

2.4.4.3 Core Services Frameworks

The following sections describe some of the frameworks of the Core Services layer and the services they offer.

2.4.4.3.1 Core Foundation Framework

The Core Foundation framework is a set of C-based interfaces that provide basic data management and service features for iOS apps. This framework includes support for the following:

- Collection data types (arrays, sets, etc)
- Bundles
- String management
- Date and time management
- Raw data block management
- Preferences management
- URL stream manipulation
- Threads and run loops
- Port and socket communication

The Core Foundation framework is closely related to the Foundation framework, which provides Objective-C interfaces for the same basic features [16].

2.4.4.3.2 Core Motion Framework

The Core Motion framework provides a single set of interfaces for accessing all motion-based data available on a device. The framework supports accessing both raw and processed data using a set of block-based interfaces [16].

2.4.5 Limitations of iOS

The following sections discuss some limitations of iOS with respect to the requirements of this project.

2.4.5.1 Continued Execution in the Background

Ideally this application would operate silently in the background, allowing the user to execute other apps and even lock the screen while continuing to monitor for falls. IOS, however, requires that specific permission be granted to an app in order for it to execute prolonged tasks in the background [4]. Unfortunately, iOS only grants this permission to certain types of apps, and only for specific services. Continuous monitoring of accelerometer output is not one of the background tasks approved by iOS. Additionally, the app must use system frameworks to implement these background services [4], ruling out the possibility of “tricking” the system into allowing background execution of non-approved tasks.

Another potential issue for trying to run this type of app in the background is the resource precedence given by iOS to the foreground app. That is, major hardware resources, such as processor cycles and RAM, are supplied to the foreground app as needed by that app. This means that any resources being used by apps in the background may be reclaimed without warning [4, 15]. This can result in the background app being suspended and purged from memory without the user knowing [4]. This would obviously set the stage for potentially catastrophic failures in a fall detection application.

For these reasons, the only viable solution is to opt for a fall detection application that runs in the foreground. This is an acceptable solution for this application because it provides for the largest margin of safety, and also has the added benefit of reminding the user that the application is running whenever they look at their phone.

2.4.5.2 Programmatic Sending of SMS Messages

This app relies on the ability to notify emergency contacts in the event of a fall. While iOS allows for a phone call to be initiated programmatically via an in app URL, this limits the application to notifying a single emergency contact. It would be ideal to be able to notify several secondary contacts via SMS text message, in addition to notifying a primary contact via phone call. Unfortunately, while iOS allows the fields of an SMS message (recipient, subject, message body) to be populated programmatically, it does not allow for the message to

actually be sent automatically [16]. The user must actively approve the message and elect to send it via a button touch. This is obviously not a viable option for a fall detection application that needs to alert emergency contacts on behalf of a potentially unconscious user.

Fortunately there is another solution for accomplishing the programmatic sending of SMS messages. This solution involves setting up an intermediate server on the Internet that uses an online SMS sending service called Twilio. This is discussed in detail in the next section.

2.5 Twilio

Twilio is a cloud communications, or infrastructure as a service, company that allows software developers to programmatically send and receive phone calls and text messages using its web service APIs. Twilio's services are accessed over HTTP and are billed based on usage [70].

2.5.1 The Twilio Connection

1. User initiates a call or sends a text message to a Twilio phone number
2. The Twilio platform accepts the incoming call or message and relays it in standard HTTP
3. The custom web application receives the HTTP request from Twilio and provides XML instructions about how to proceed

2.5.2 Twilio Client for iOS

Twilio Client for iOS is an Objective-C library that enables voice or text communications with landlines or other devices, including web browsers and mobile devices. There are three major pieces in a Twilio Client app [70]:

- The iOS app that uses the Twilio Client library (libTwilioClient.a) to add Voice over IP (VoIP) features to the application
- A server-side, web application to grant capabilities to the Client iOS app, and to orchestrate Twilio's telephony services
- A TwiML application to provide Twilio's cloud services and to connect the iOS app with the server-side app deployed on public internet
    - This application is run by Twilio and receives the incoming connection from the iOS app and routes to the URL where the back-end application is deployed

These components are illustrated in figure 5 below.

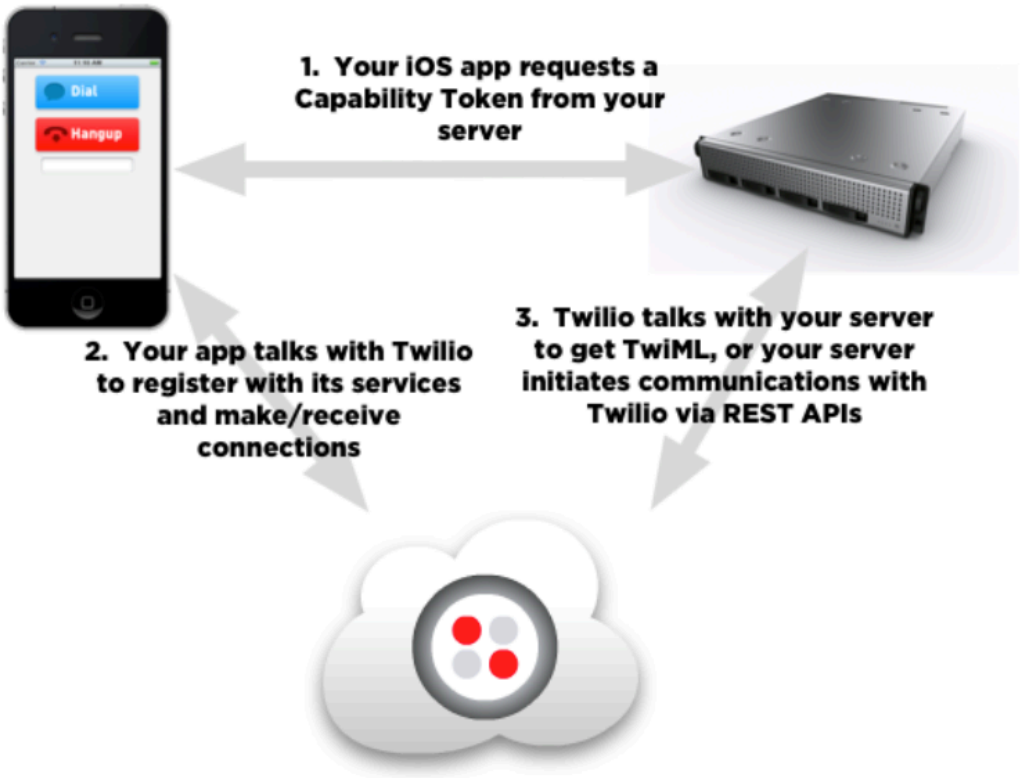

Figure 5 – Diagram of the main components of a Twilio Client Application [70].

2.5.2.1 Twilio Client iOS Application

The Twilio Client iOS application must be configured for Voice over IP (VoIP) and for communicating with Twilio. The primary class for connecting to Twilio services from the iOS app is TCDevice. This class coordinates service authorization with Twilio, listens for incoming connections, and establishes outgoing connections. Connections to Twilio, whether incoming or outgoing, are represented by instances of the class TCConnection. Additionally, status callbacks are provided to objects that implement the delegate protocols TCDeviceDelegate and TCConnectionDelegate [70].

2.5.2.2 Back-End Application Server

The back-end application server is deployed on the public Internet and serves two main purposes. First, it is responsible for supplying capability tokens for the iOS Client application. Capability tokens are what Twilio Client relies on to sign communications from devices to Twilio. They allow for Twilio capabilities to be added to mobile applications without exposing the AuthToken in the client-side environment. The token is created on the server and specifies what capabilities are being granted to the device. All tokens have a limited lifetime for protection [70].

The other main responsibility of the application server is to serve up the TwiML application and or make REST API calls to orchestrate Twilio's telephony services. TwiML, or the Twilio Markup Language, is a set of instructions that are used by the custom web application to tell Twilio what to do when an incoming connection is received. When a connection to a Twilio number is established, Twilio looks up the URL associated with that number and makes a request to that URL. Twilio reads the TwiML instructions at that URL to determine what to do (such as send SMS messages to a list of contacts).

2.5.2.3 TwiML Application

Twilio Client connections aren't made to a specific phone number. For this reason, Twilio relies on a TwiML Application within a Twilio account to determine how to interact with the back-end server. This TwiML application stores a set of

URLs, which have requests sent to them when a device initiates a Twilio Client connection [70].

2.5.2.4 Integration with Fall Detection Application

These pieces come together to give the fall detection application the capability to programmatically send text messages to a list of emergency contacts. The iPhone is set up to establish a connection with Twilio. The phone will not communicate with the other devices directly, but rather it will instruct Twilio to fetch TwiML from the web server to handle the connections to the other devices. The script deployed on the web server configures the messages to be sent by Twilio. Twilio relies on the TwiML Application within a Twilio account to determine how to interact with the web server. This application is really just a convenient way to store a set of URLs. So when the iPhone initiates a Twilio Client connection, a request is made to one of these URL properties, stored in the application within the account. Twilio then uses the TwiML response fetched from the script on the web server that was pointed to by the URL. This script directs what happens with the client connection [70].

2.6 G-Force Testing Application

This section discusses the creation of a simple iOS application to test the reading and processing of acceleration data from the phone's onboard accelerometers.

2.6.1 Purpose

This preliminary application was created with several purposes in mind. First, it was the most basic and logical place to start development. The final application hinges on the basic principle of reading and analyzing accelerometer data. An application that simply reads accelerations from the phone's hardware was a key first step toward the final goal of a fall detection application. The other main reason for the creation of this simple application was for basic testing purposes.

While the final application would undoubtedly have to be tested thoroughly, there were variables it made sense to test first on a simple application. For instance, the accelerometer sampling speed was an unknown variable prior to testing. The phone's accelerometers use a significant amount of battery, which is dependent on the sampling speed. Therefore a balance must be struck so that the frequency is kept low enough to preserve battery life, while also being high enough to provide adequate resolution for fall detection. Another important variable that needed to be tested was the most effective phone placement on the individual. The placement needs to be such that the accelerometer disturbance is maximized during a fall but minimized during normal daily activities.

Another function of this preliminary app was to characterize various movements with respect to the accelerations recorded by the phone. Specifically,

characterizations of activities of daily living and various fall events had to be obtained. This information is important for determining whether an app based on accelerometer data from a smartphone could reliably distinguish a fall from a non-fall event. This was an important proof-of-concept step in order to justify moving on to full app development.

Beyond proving the concept, this information is critical for setting the threshold accelerations for declaring when a fall has occurred, and for determining the most efficient algorithm for making that determination. There were several options with a threshold-based algorithm that needed to be explored. The most basic option being the algorithm that was used in the original senior design project upon which this thesis was based [39]. That method involved setting acceleration thresholds for the x, y, and z-axes independently. Then if an acceleration is registered outside the set thresholds for any of the three axes, a fall is declared. Written in pseudocode, this algorithm looks like:

declare fall if (x_thresh_low > x_accel > x_thresh_hi) or
(y_thresh_low > y_accel > y_thresh_hi) or
(z_thresh_low > z_accel > z_thresh_hi)

A possible alteration to this algorithm would be to set a single threshold value and compare that to the sum of all three accelerations. This would look like:

declare fall if sum_thresh_low > (x_accel + y_accel + z_accel) >
sum_thresh_hi

Additionally, taking the absolute value of the accelerations prior to summing them was another option worth exploring. This would look like:

declare fall if (abs(x_accel) + abs(y_accel) + abs(z_accel)) > abs_sum_thresh

2.6.2 Creation

In order to create this simple application for testing, a sample project, available on Apple's developer website, was acquired as a starting point. This template project, called "Motion Graphs", demonstrates how to use the push method to receive data from the Core Motion framework [19]. The default behavior of the app is to collect accelerometer, gyroscope, and device motion data and display graphs of the data in real-time.

The Motion Graphs project comes with the following classes and protocol:

Table 2 – Classes and Protocols included with the Motion Graphs sample project [19].

GraphView	A UIView subclass that provides the ability to plot accelerometer, gyroscope, and device motion data.
GraphViewController	A view controller that handles the display of accelerometer, gyroscope, and device motion data. Depending on the argument that's passed into its initWithMotionDataType: method, it can display graph(s) generated from one of the three data types.
MotionGraphsAppDelegate	A standard implementation of the UIApplication Delegate Protocol. The UIApplicationDelegate protocol defines methods that are called by the singleton UIApplication object in response to important events in the lifetime of the app. The app delegate works alongside the app object to ensure the app interacts properly with the system and with other apps. Specifically, the methods of the app delegate give the app a chance to respond to important changes. The app delegate is effectively the root object of the app.

The main functionality provided by this sample project is the ability to access and process motion events from the phone's hardware. Specifically, the data collected by the phones accelerometer is of interest. In actuality, the "accelerometer" is made up of three accelerometers: one for each axis – x, y, and z. Each measures changes in velocity over time along a linear path. Combining the three allows for detection of device movement in any direction [13].

Apple's Core Motion framework allows access to the accelerometer, gyroscope, and device motion classes [13, 16, 19]. The framework is primarily responsible for accessing raw accelerometer data and passing that data to the app for handling. The CMMotionManager class is the central access point for

Core Motion [13]. An instance of this class is created, an update interval is specified, a request is made to start updates, and the motion events are handled as they are delivered.

The CMMotionManager class offers two approaches for obtaining motion data: pull and push. With the pull method, an app requests that updates start and then periodically samples the most recent measurements of motion data. With the push method, an app specifies an update interval and implements a block for handling the data. Next, it requests that updates start and passes Core Motion an operating queue and the block. Core Motion delivers each update to the block, which executes as a task in the operation queue. Pull is the recommended method for most apps, as it is generally more efficient and requires less code. However, push is appropriate for apps that cannot miss a single sample measurement, as is the case with a fall detection application [13].

Although the Motion Graphs sample app provided a great starting point, modifications were necessary to get a usable application for preliminary testing. The default behavior of the sample app is to display a running graph of data being collected in real time. The user is able to select which data to display (device motion, accelerometer, or gyroscope) and what the update interval will be. For our purposes, only the accelerometer data is of interest, so the other functionality was stripped out. Additionally, since this app would be used for data collection, a control was added to the user interface, enabling the tester to start and stop data collection more precisely. Furthermore, while the default app

provided an elegant looking running graph of the data being collected, this function was unnecessary and imprecise. The default app was also not useful for data analysis, as the data was only stored temporarily and inaccessible. Therefore, the display was modified to remove the graph and replace it with a display of the current acceleration in each direction. This display allows for the tester to verify that the app is running and sampling accelerometer data [19].

For thorough testing it was necessary to be able to access entire collected data sets and analyze them. This required creating a persistent data store within the app that could be downloaded and accessed via computer. For this task, a property list was chosen as the persistent data store. Property lists organize data into named values and lists of values using several object types. These types provide the means to produce data that is meaningfully structured, transportable, storable, and accessible, but still as efficient as possible. The property list programming interfaces provided by Cocoa (Apple's framework for building software programs to run on iOS) allow for conversion of hierarchically structured combinations of these basic objects to and from XML. The XML data can be saved to disk and later used to reconstruct the original objects. The process of converting the data from its runtime (object) form to a static representation that can be stored in the file system (XML) and back, is called serialization and deserialization, respectively. The NSPropertyListSerialization class provides methods for serialization and deserialization and automatically takes account of

endianness (the required order of bytes per unit of memory) on different processor architectures [25].

3 TEST RESULTS AND DISCUSSION

This chapter presents and discusses the results from the testing done using the standalone g-force testing application. The results are discussed in the context of the fall detection application design.

3.1 Punching Bag as Human Test Analogue (Backward Fall)

The first set of testing was done with the primary aim of discovering the ideal accelerometer sampling speed. Four different speeds were selected for testing: 10 updates per second, 20 updates per second, 40 updates per second, and 100 updates per second (the fastest possible). This set of testing was performed using a punching bag as a human analogue. The iPhone was secured to the punching bag via a neoprene waist pack that held the phone in a Velcro pouch. Accelerometer data was collected while simulating falls by tipping the punching bag and letting it fall to the floor. This test was designed for its repeatability rather than its realistic approximation of a fall event. In order to isolate the variable of interest (accelerometer sampling speed), multiple falls needed to be simulated that were as close to identical as possible. This test allowed for a comparison of falls of very similar speed, impact, and trajectory, while varying the accelerometer sampling speed between tests. This test was also the first opportunity to experiment with different fall detection algorithms on the raw acceleration data.

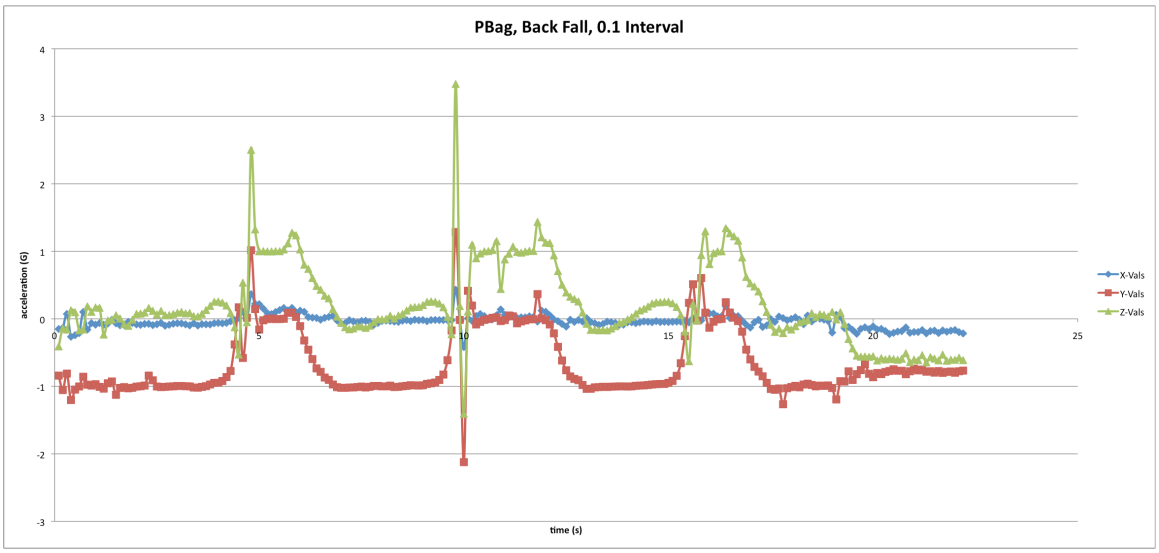

Figure 6 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the backward direction and an update interval of 0.1 (10 per second).

Figure 6 shows the iPhone's accelerometer output over a short time interval, during which three falls were simulated. The x-axis displays the time passed in seconds, while the y-axis displays the output from the accelerometer in g-forces. One g-force, or G, is equal to the force exerted by the earth's gravitational field (9.81m/s^2). We see with the phone at rest (prior to fall events and after reset of bag) the output of the accelerometer in the x and z directions are approximately zero, while the output in the y direction is about negative one. This is what we would expect the acceleration to be with the phone at rest in a vertical position. That is, with the x and z axes parallel to the ground, gravitational acceleration will have no effect on them, while the y axis is perpendicular to the ground with gravitational acceleration working in the negative direction, as seen in figure 7. During the falls we see a spike in acceleration, known as an impact shock. This shock is most pronounced in the z direction for this test, as this is the direction facing down when the bag impacts the ground. While one fall registers a

peak acceleration of over three g-forces in the z-axis, another fall barely peaks over one g. This result is probably due to the relatively slow accelerometer update speed. Impact shock forces occur over a very short period of time. If the accelerometer updates occur too infrequently, then the peak acceleration may occur between updates and not be recorded. This test seems to indicate that an update speed of 10Hz is too slow.

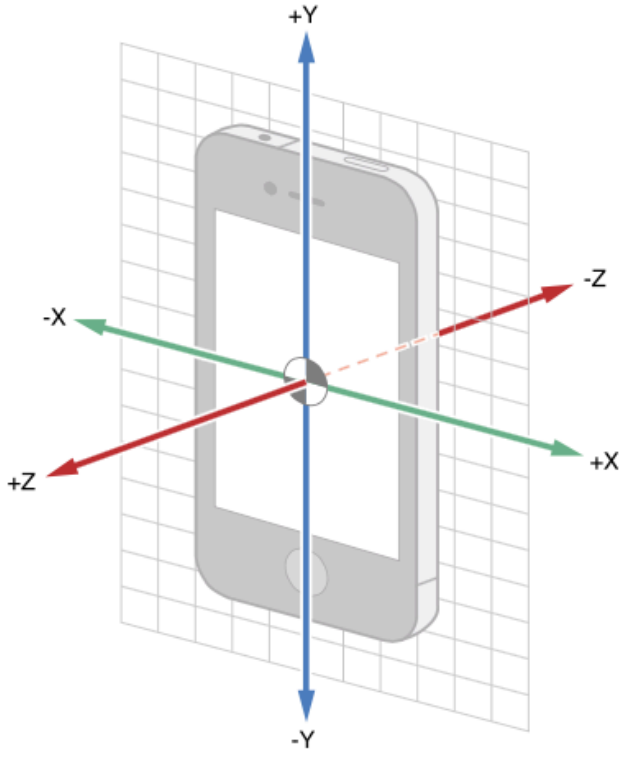

Figure 7 – Diagram illustrating the three axes of an iPhone.

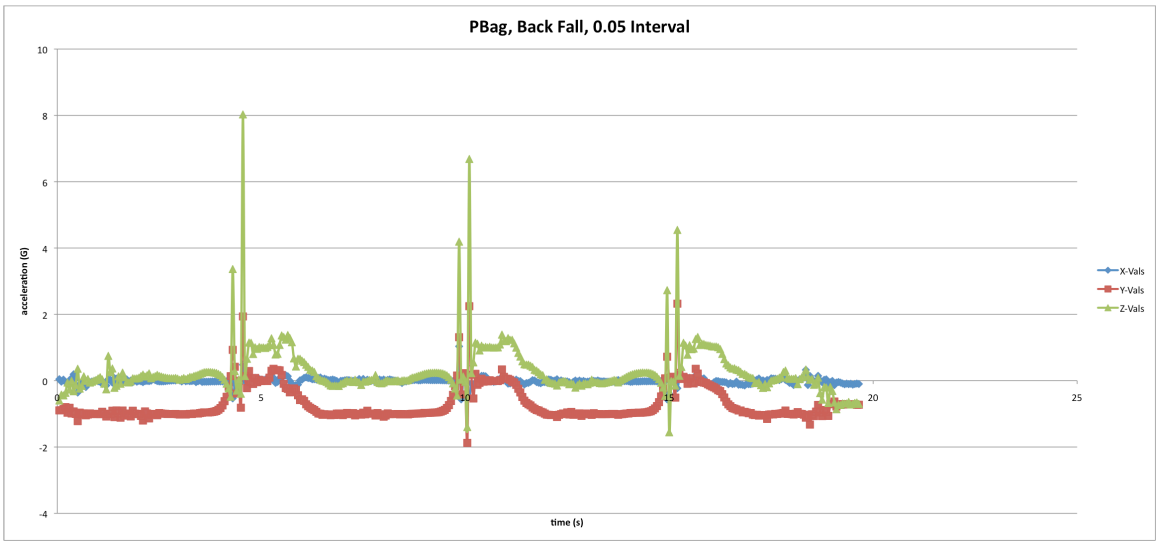

Figure 8 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the backward direction and an update interval of 0.05 (20 per second).

Figure 8 shows the same test as the previous graph, except the accelerometer update frequency has been doubled from 10Hz to 20Hz. We can see that the resolution of this graph is noticeably improved over the previous one. Impact spikes are now peaking between 4 and 8 g-forces in the z direction, a sizeable increase over the previous test's peaks of 1 to 3 g-forces. Additionally, we now see accelerations in the y direction peaking over 2 g-forces in visible spikes that were mostly absent from the first graph. It is also interesting to note that each fall event is now appearing as 2 spikes in the z direction; the slower update speed was unable to register both spikes.

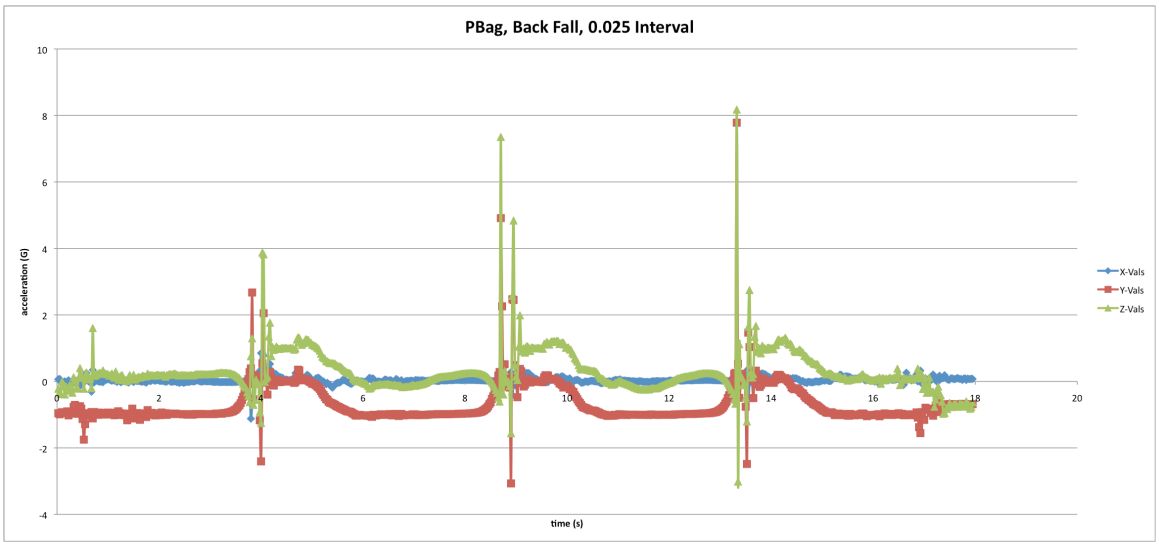

Figure 9 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the backward direction and an update interval of 0.025 (40 per second).

Figure 9 again shows the same test, but with the frequency increased now to 40Hz. This graph looks similar to the previous graph, except the spikes in the y direction are more pronounced and with greater amplitude. Additionally, bounce back accelerations in the y direction are much more pronounced, with accelerations registering past -2 g-forces for each fall. Also of note is how closely the spikes in the y and z direction correspond with each other, with the y-axis accelerations having only slightly decreased amplitude compared to the z-axis. This finding supports the idea that that a fall detection algorithm using a sum of the accelerations in multiple axes may be an improved metric over individual thresholds for each axis.

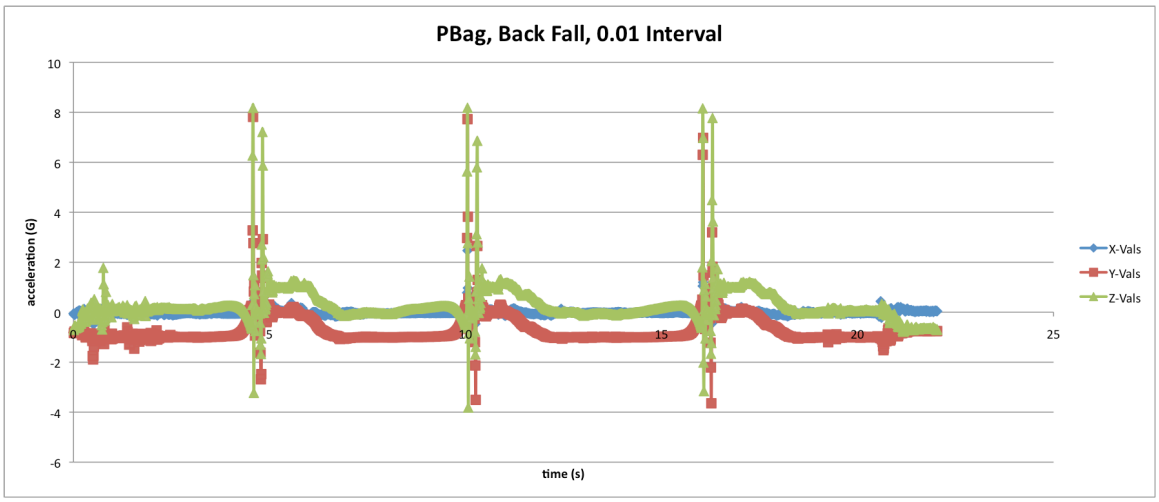

Figure 10 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the backward direction and an update interval of 0.01 (100 per second).

Figure 10 shows the test with the accelerometer updating at its maximum speed of 100Hz. This graph shows similar shape and maximum amplitudes to the previous graph. The main difference here is a higher degree of consistency in the peak accelerations being captured for each fall. This is an intuitive result when considering the very short time interval over which impact shocks occur. Even with 40 updates per second, as seen in figure 9, the maximum peak acceleration may not be captured for every fall. This does not necessarily mean that the maximum update speed must be used for a reliable system. The accelerometer uses a significant amount of power and therefore it is desirable to use as slow an update interval as possible to maximize battery life. As long as the update interval being used can reliably detect falls past a certain g-force threshold, then it is acceptable.

This first set of testing was done primarily to assess the accelerometer sampling speed and determine a rough minimum sampling speed that would be

acceptable. From this testing it was determined that a minimum sampling speed of 40Hz should be used to maximize battery life while retaining sufficient resolution to not miss major acceleration events. A sampling speed slower than this does not provide adequate resolution and could result in reliability issues when attempting to detect minor fall events.

This set of testing also lends credence to the idea that altering the fall detection algorithm could improve its reliability. Specifically of interest is the fact that during a fall event, acceleration spikes can be seen in multiple axes. An algorithm that uses the aggregate of the three axes and compares it to a single threshold value would raise the threshold value, allowing for minor spikes and fluctuations in any individual axis to be ignored, while still responding to falls, as these would register in multiple axes. The next set of testing, involving falls in the sideways direction, was important for testing and developing this algorithm.

3.2 Punching Bag as Human Test Analogue (Sideways Fall)

This test was performed in order to observe how fall direction would affect the impact spikes seen in the three axes and to test some alterations to the fall detection algorithm. Specifically, these alterations involve using an aggregate of the accelerations in all three axes and comparing it to a single threshold value.

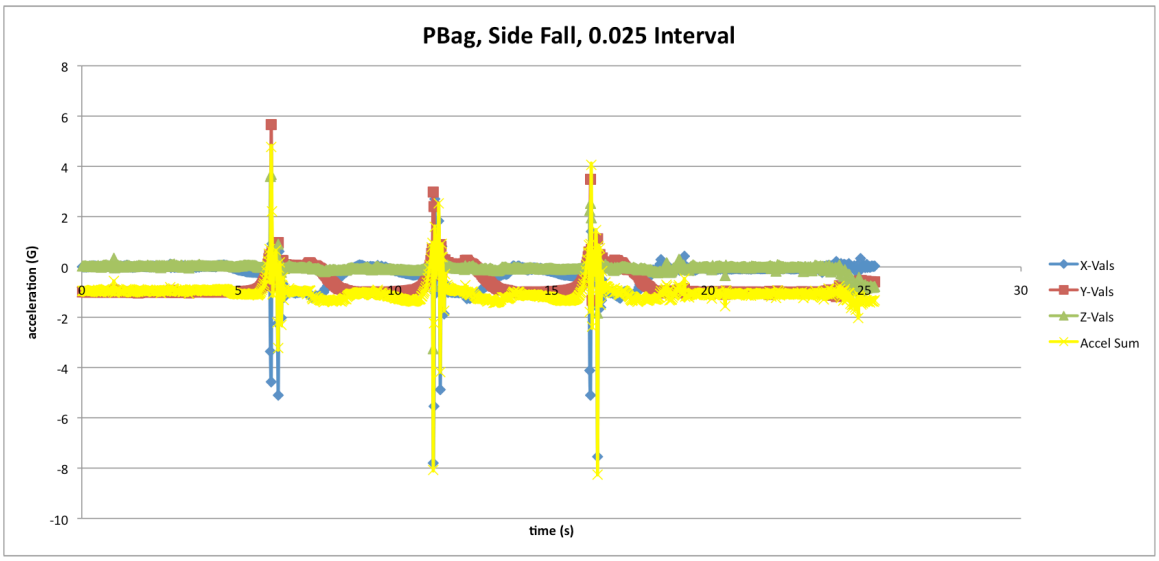

Figure 11 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the sideways direction and an update speed of 40Hz. Included in yellow are the sums of the accelerations in the three axes.

The first thing to notice about the graph in figure 11 is that it shows spikes in the x and y directions during fall events, with little activity in the z direction. This is in contrast to the backwards fall tests, where the spikes were seen in the y and z directions, with little activity in the x. This result is intuitive, given the relative positioning of the phone upon impact for these two tests. Another thing to note is that in this case, the acceleration spikes in the x-axis are in the negative direction. These negative spikes are of particular interest when considering an aggregate of accelerations that takes the sum of the three axes as a metric for determining a fall. This aggregate value is shown in yellow on the graph. It is apparent that this aggregate metric is not an improvement for determining falls, as its spikes are never significantly greater in amplitude than any of the individual axes. This lack of improvement is due to the fact that spikes in individual

directions can be either positive or negative and therefore potentially cancel each other out when summed in an aggregate.

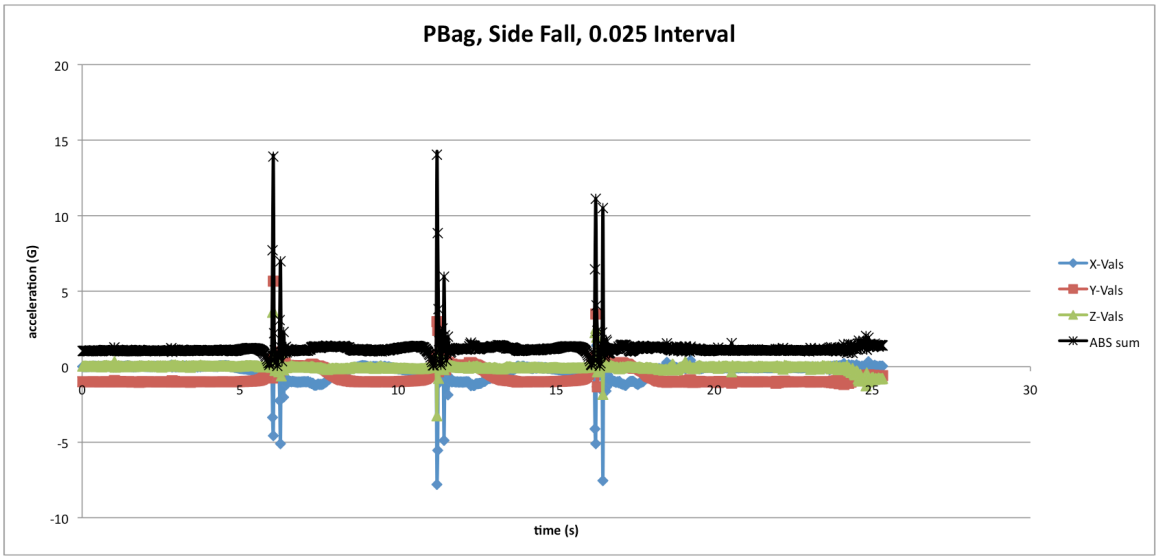

Figure 12 – Graph of accelerations in each axis, during three simulated falls with a punching bag, in the sideways direction and an update speed of 40Hz. Included in black are the sums of the absolute accelerations in the three axes.

Figure 12 shows data from the same test, however an absolute aggregate is displayed instead of the simple sum aggregate used in the previous graph. For this metric, absolute values of the accelerations in each axis were taken before being summed together for an aggregate value. You can see that this metric shows significantly increased amplitude for the impact spikes during falls. Where individual spikes peak in amplitude from 3 to 8 g-forces (positive or negative), this new absolute aggregate reaches peaks between 11 and 14 g-forces. Subsequent testing will show if this increased sensitivity is also seen in activities of daily living. If not, then using the absolute aggregate will significantly improve the reliability of fall detection.

3.3 Human Test Subject

Subsequent testing was all performed with the iPhone attached to a real human subject. These tests involved performing various activities of daily living (ADL) as well as executing different fall scenarios. One objective of these tests was to find the ideal phone placement on the individual to maximize g-forces experienced by the phone during a fall and minimize those experienced during ADL. Additionally, the data from these tests was used to construct the g-force thresholds such that they reliably detect any fall that could potentially result in injury, while also ignoring all ADL.

3.4 Activities of Daily Living Testing

This set of testing focused on obtaining g-force profiles for normal daily activities, or activities of daily living (ADL). This testing was crucial for establishing threshold values for determining falls; the threshold should be set above the maximum accelerometer disturbance observed during ADL, as to not cause excessive false alarms. These tests were also important for evaluating the absolute aggregate acceleration value as an improved metric for fall detection. This aggregate value showed significantly higher spike values for falls during the previous set of testing. However, for it to improve reliability, the aggregate should not show significantly higher spikes than any single axis during ADL. These tests were also important for determining the best positioning of the phone on the

individual. Again, with the aim of maximizing reliability, a position where the phone can be worn so that accelerometer disturbance is minimized during ADL but maximized during falls is desired.

3.5 Walking

Walking is obviously a very common daily activity and has the potential to cause significant false alarms if the fall threshold is set below an acceleration level that is frequently surpassed during walking. It is important to characterize this activity for different placement positions in order to compare them to each other, as well as to the acceleration profiles seen during falls.

Three placement positions of interest were identified: chest, hip, and thigh-pocket. Attachment at the chest was the location chosen in the original design project [39]. This position is high above the ground and therefore will most likely experience significant impact shock during a fall. Additionally, the attachment to the upper torso should minimize most activities of daily living, although it may be susceptible to disturbances during activities such as lying down or bending over. The hip was selected for its proximity to a person's center of gravity, making it unlikely to experience significant disturbances during all ADL. The thigh pocket was selected as a practical and convenient placement for the user. It can't be ignored that the convenience of using the application will play a role in its usefulness. If users find it too inconvenient to use, then it is useless to those individuals. Placement of the phone in the user's thigh pocket may result in

significant accelerometer disturbance during ADL as well as potentially lessened disturbance during falls, due to its lower placement on the individual (both negatives for our system). However, if this position proves to be adequate, then its added convenience may actually give it an edge over other placements.

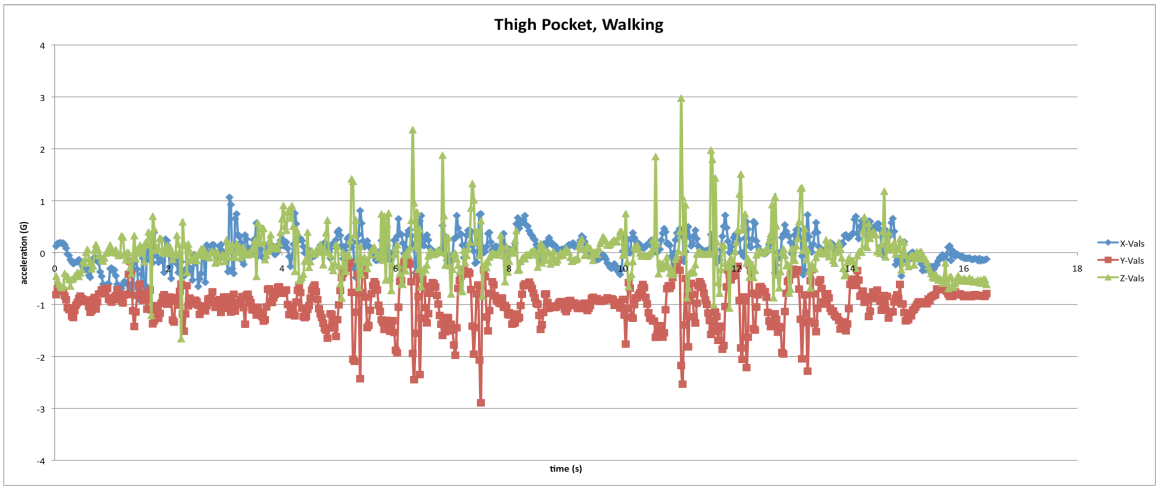

Figure 13 – Graph of accelerations in each axis, during walking, with thigh pocket placement.

Figure 13 shows data collected with the phone placed in the user's thigh-pocket during a walk down a hallway and back. This placement permitted the phone to swing somewhat during walking, impacting the subject's thigh and resulting in the acceleration spikes seen primarily in the positive z-direction and negative y-direction. A number of these spikes exceed an amplitude of 2 g-forces, with several approaching 3 g-forces.

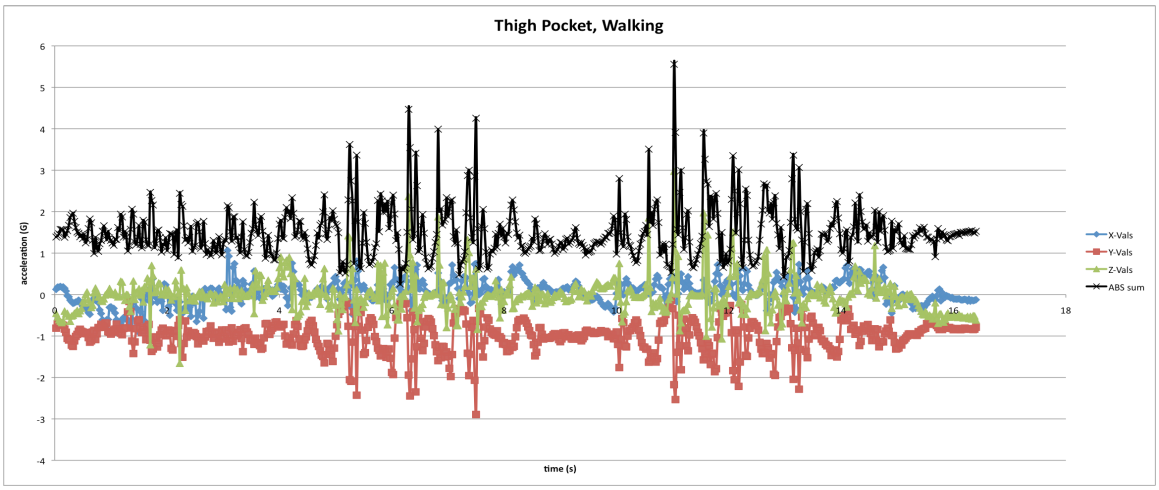

Figure 14 – Graph of accelerations in each axis, during walking, with thigh pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

This graph is from the same test, but with the absolute aggregate acceleration metric added in. This aggregate does show increased amplitude over the individual axes, however not to the extent that the amplitude was increased for the previously discussed fall-simulation testing. This is an encouraging result for this absolute aggregate value as an improved fall detection metric. This phone placement resulted in several spikes exceeding 4 g-forces and one exceeding 5 g-forces. This is a fairly significant disturbance for an ADL, however if fall spike values are consistently higher than the spikes seen here, then this configuration may prove to be acceptable.

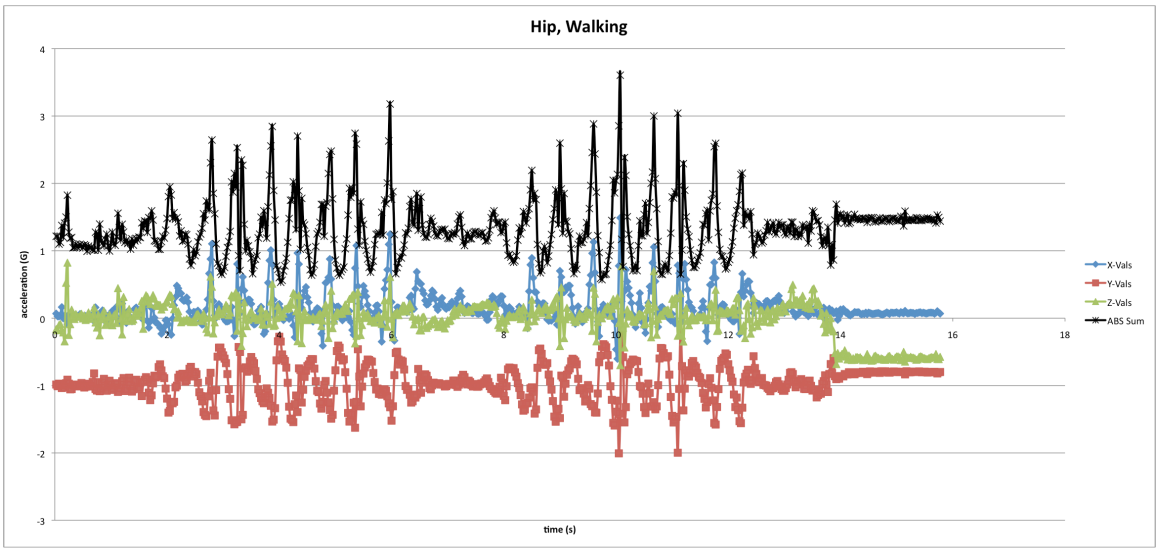

Figure 15 – Graph of accelerations in each axis, during walking, with attachment. Included in black are the sums of the absolute accelerations in the three axes.

Figure 15 displays data from the subject walking with the phone attached securely at the hip. It is clear that this attachment method results in less accelerometer disturbance when walking than placement in the thigh-pocket. This result is most likely due to two factors. Firstly, the secure attachment eliminates the swing and impacts that were seen in the previous test. Secondly, a person's hip simply experiences less movement during walking than does one's thigh. Again, what will be important is the difference between accelerations experienced during ADL and those experienced during falls. Therefore, we won't know if this placement is superior to that of the thigh-pocket until we characterize falls with both placements.

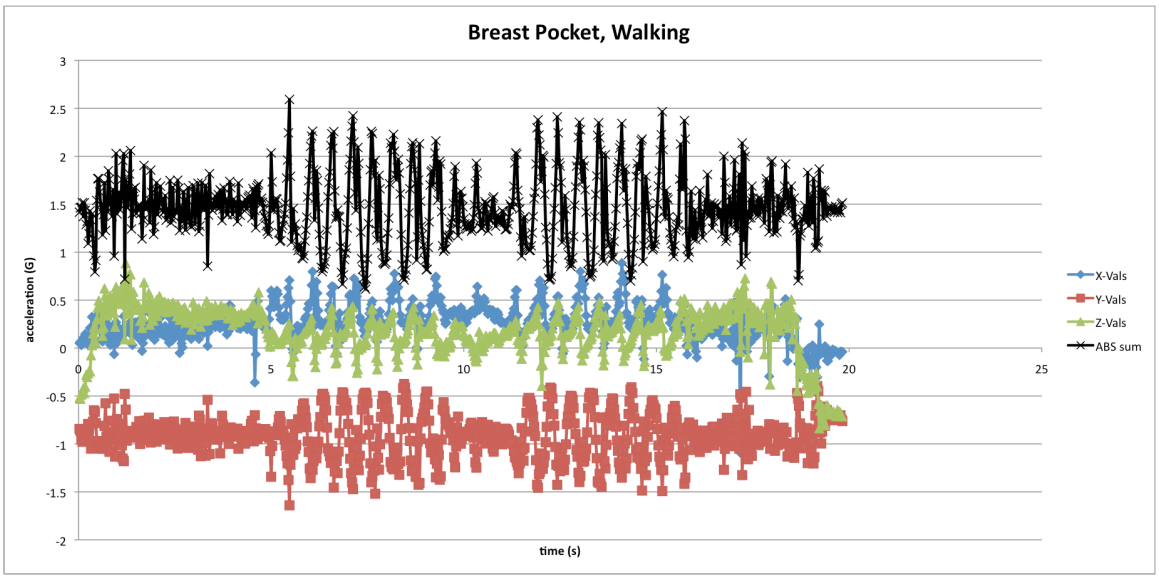

Figure 16 – Graph of accelerations in each axis, during walking, with breast pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 16 shows data of the subject walking with the phone placed in a somewhat loose breast pocket. Counter intuitively, this placement actually showed a fairly significant decrease in accelerometer disturbance over that of the firm hip attachment. It was expected that this test would closely resemble that of the previous test, as both tests used attachment to the subject's torso. However, it seems that the chest experiences more subtle motion than does the subject's hip. Additionally, the placement in a somewhat loose pocket may be absorbing some of the shock and therefore smoothing the accelerations experienced by the phone. It will be crucial to see if this smoothing effect is seen during falls as well; if it isn't, then this may be the superior attachment method.

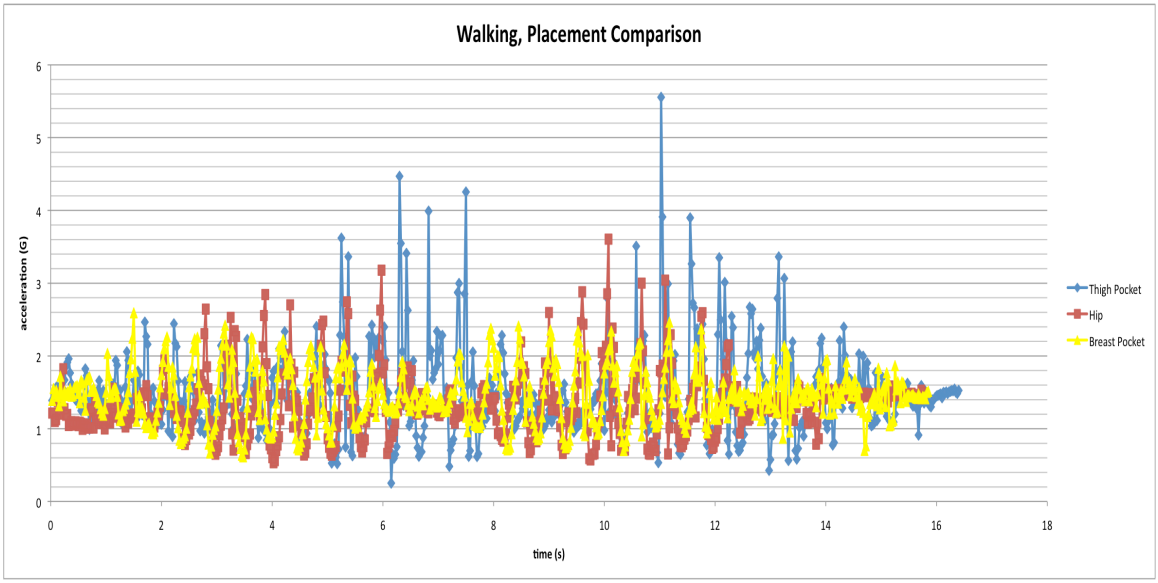

Figure 17 – Graph comparing aggregate accelerations experienced during walking for the three placement positions.

Figure 17 shows an overlay of the absolute aggregate acceleration values from each of the previous placement tests. This provides for a good visual comparison of how each placement position affects the accelerometer disturbance seen during walking. It is clear that the thigh-pocket placement causes the most disturbance on average and the greatest maximum disturbance. It also seems to be the most erratic in the range of values seen. Whereas, the breast pocket placement shows very consistent values, while also having the lowest maximum. The hip attachment configuration seems to be approximately in the middle as far as average and maximum disturbance.

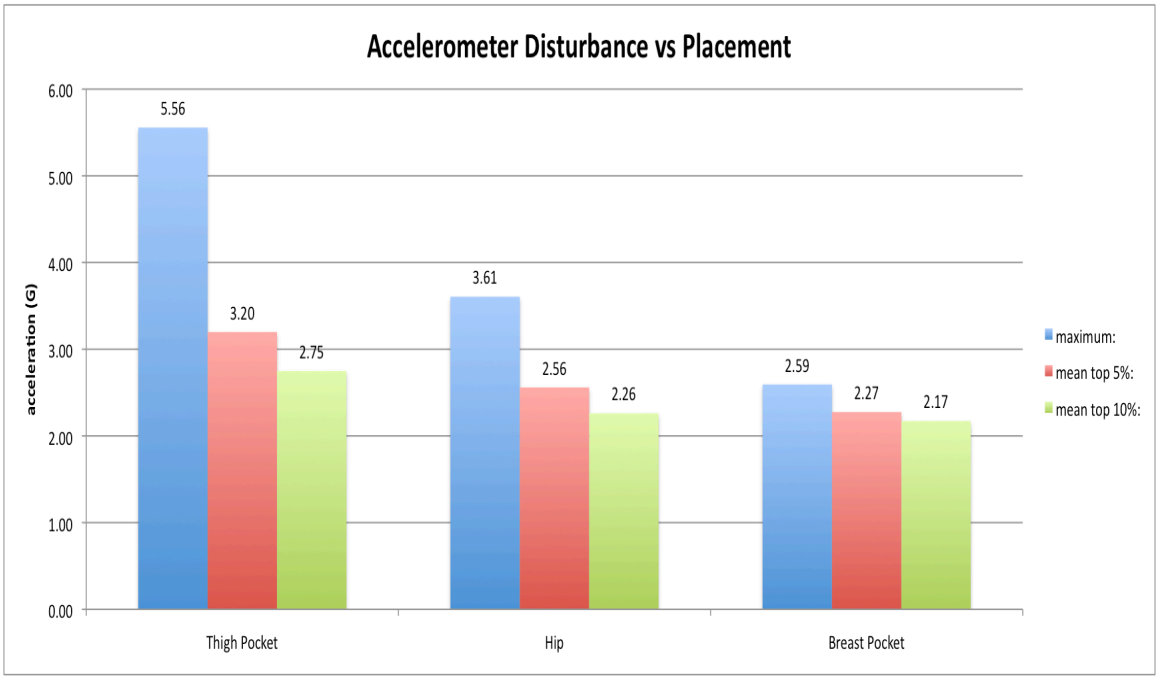

Figure 18 – Graph comparing the maximum acceleration, mean of top 5% of accelerations, and mean of top 10% of accelerations, for the three placement positions during walking.

Figure 18 provides another way of visualizing the comparison between the phone placement positions. For each position, a bar graph displays the maximum absolute aggregate acceleration recorded during the walking test, the mean of the top 5% of values, and the mean of the top 10% of values. All of these values are of interest. Although the maximum value used here will contain potential outliers, this does not nullify its usefulness. In fact, outliers are actually of particular interest in this case, as they represent potential false alarms during real world use of the device. We can see from the graph, the thigh-pocket configuration experienced the greatest maximum aggregate acceleration, followed by the hip placement, and then the breast pocket placement. Also of interest was the average accelerometer disturbance of each position during walking. However, a simple average of all the data is not sufficient, as the test

includes several periods at the beginning middle and end where the subject wasn't walking, and therefore the data from these areas pulled down the overall means. Additionally, the amount of time not walking was not equal for each test and therefore affected the means unequally. A more useful and interesting metric was the mean of the top 5% and 10% of values for each test. For theses metrics, the same trend is seen as with the maximum values: thigh pocket greatest, hip attachment middle, and breast pocket lowest. The consistency of the breast pocket configuration is also reaffirmed by this test with the maximum and both mean values being very close to one another. This is in contrast to the hip configuration and especially the thigh-pocket configuration, where the maximum value is much higher than the mean values.

These walking tests demonstrate further promise of the absolute aggregate acceleration as an improved fall detection metric over accelerations in the individual axes. This improvement is shown by the general lack of significant spikes for this important ADL. Furthermore, the testing shows that all three of the selected placement positions remain viable options. However, the breast pocket configuration seems to show the most promise, as it displays the least accelerometer disturbance during walking. Subsequent testing of more ADL, as well as thorough fall testing, will complete the picture and determine if these early assessments hold true.

Table 3 – Analysis of Variance Between Placement Positions for Walking Test.

ANOVA: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Thigh Pocket	656	975.5911865	1.487181687	0.343673567
Hip	631	846.6631927	1.34178002	0.182850014
Breast Pocket	634	921.4251099	1.453351908	0.142960313

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	7.395761306	2	3.697880653	16.46380679	8.13845E-08	3.000416196
Within Groups	430.7955737	1918	0.22460666			
Total	438.191335	1920				

Table 3 above shows an analysis of variance (ANOVA) test performed on the data from the walking ADL. This is a statistical test comparing the three placement positions; its purpose is to determine if the differences seen in the accelerations between the placement positions are statistically significant. The null hypothesis for this test is that there is no difference between the means for the three placement positions. The alternate hypothesis is that not all the means are the same. In order to reject the null hypothesis, the F statistic should be higher than F critical and the P-value should be below the chosen alpha value of .05. The table shows that F is indeed greater than F critical, and the P-value is 8.14×10^{-8}, indicating the probability that the differences seen in the means are merely due to sampling error is extremely low. Therefore we can reject the null hypothesis and accept that the means are not all equal. This test does not,

however, tell us whether all three means are different, or just one different from the others. To determine this we need to use a t-Test.

Table 4 – t-Tests Comparing the Three Placement Positions.

	t-Tests		
Placement Position	*Hip vs Breast Pocket*	*Thigh Pocket vs Breast Pocket*	*Thigh Pocket vs Hip*
P-value	5.02774E-07	0.108395858	1.99957E-07

Table 4 above shows the results from three t-tests performed on the acceleration data from the walking tests. The three tests were performed comparing each placement position to the other. For these tests, as with the previous ANOVA, the null hypothesis is that there is no difference between the means of the data sets being compared. In order to reject this null hypothesis, the P-value must be less than the chosen alpha value of .05. Two of the t-tests resulted in substantially lower P-values, indicating that the null hypothesis can be rejected for the Hip vs Breast Pocket and for the Thigh Pocket vs Hip tests. In other words, there is a statistically significant difference between those placement positions. However, we are unable to reject the null hypothesis for the Thigh Pocket vs Breast Pocket in this case. The P-value obtained for that comparison was not found to be less than the chosen alpha, and therefore the variance between these two sets of data could be due to sampling error. This P-value could be due to the fact that the variances within the Thigh Pocket and Breast Pocket placement groups are relatively large. These within-group variances are increased by the way the walking test was performed. That is, the test included a period of walking and a period of standing still. Since we are really

only concerned with comparing the average peak accelerations for the different positions, another t-test was performed on the top 5% of accelerations from each group. The results from this test are shown below.

Table 5 – t-Tests Comparing the Top 5% of Values for the Three Placement Positions.

	t-Tests (top 5% values)		
Placement Position	*Hip vs Breast Pocket*	*Thigh Pocket vs Breast Pocket*	*Thigh Pocket vs Hip*
P-value	3.01307E-05	4.1019E-09	9.02912E-06

Table 5 shows the t-test run on the subsets of data containing only the top 5% of accelerations for each placement position. This test allows us to determine whether there is a statistically significant difference in the accelerometer disturbance experienced by the device in the various placement positions. All three tests resulted in P-values that were considerably less than the chosen alpha value, allowing us to reject the null hypothesis and declare that there is in fact a statistically significant difference between the placement positions.

3.6 Bending Over

The previous set of ADL testing, concerned with walking, demonstrated that each of the selected phone placement positions remain viable, with the breast pocket showing the most promise. This set of testing involves the test subject bending over, as if to pick something up off the floor. This could be a test where the breast pocket placement shows more accelerometer disturbance than the other positions, due to its location on the upper torso.

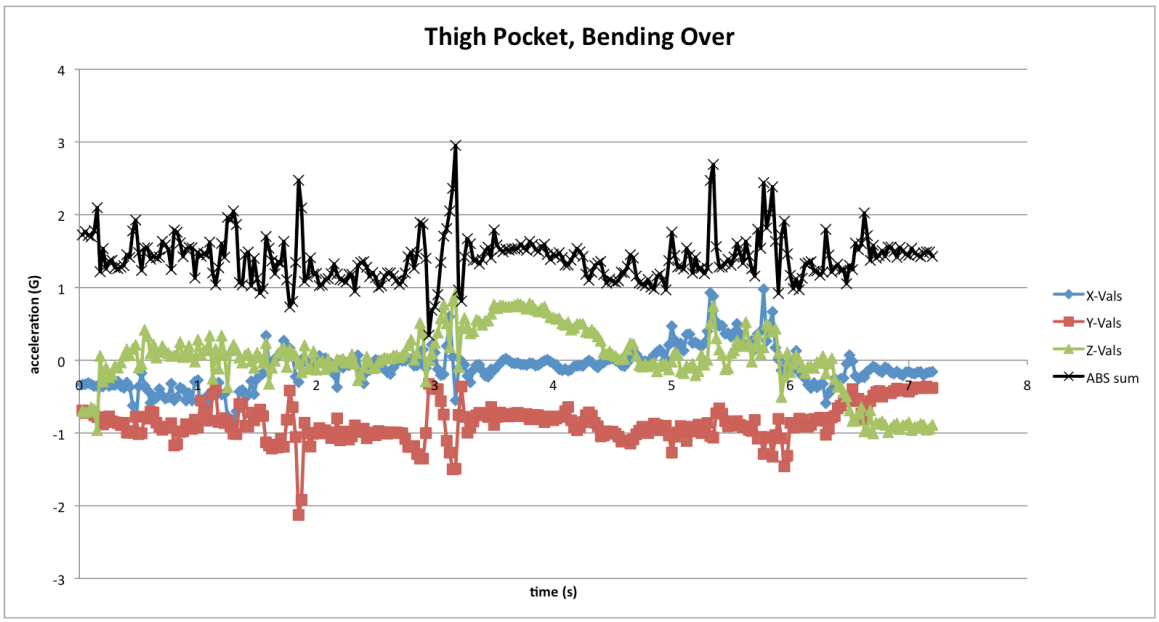

Figure 19 – Graph of accelerations in each axis while bending over, with thigh pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 19 shows data collected with the phone placed in the user's thigh pocket while bending over. This activity only causes minor accelerometer disturbance, as the absolute aggregate value never exceeds three g-forces. Additionally, this test demonstrates the absolute aggregate metric's ability to devalue non-fall accelerations, which tend to be uniaxial. Just prior to the two-second mark, we can see there is a spike exceeding negative two g-forces in the y-axis. This, however, corresponds with only a small spike in the absolute aggregate value, which is only a few tenths of a g-force greater in absolute amplitude than the spike in the y-direction. This is due to the fact that the x and y axes were relatively unaffected by the movement, and therefore didn't add much to the absolute aggregate value. This is in contrast to what we saw for the fall events, where each axis was affected by the impact shock.

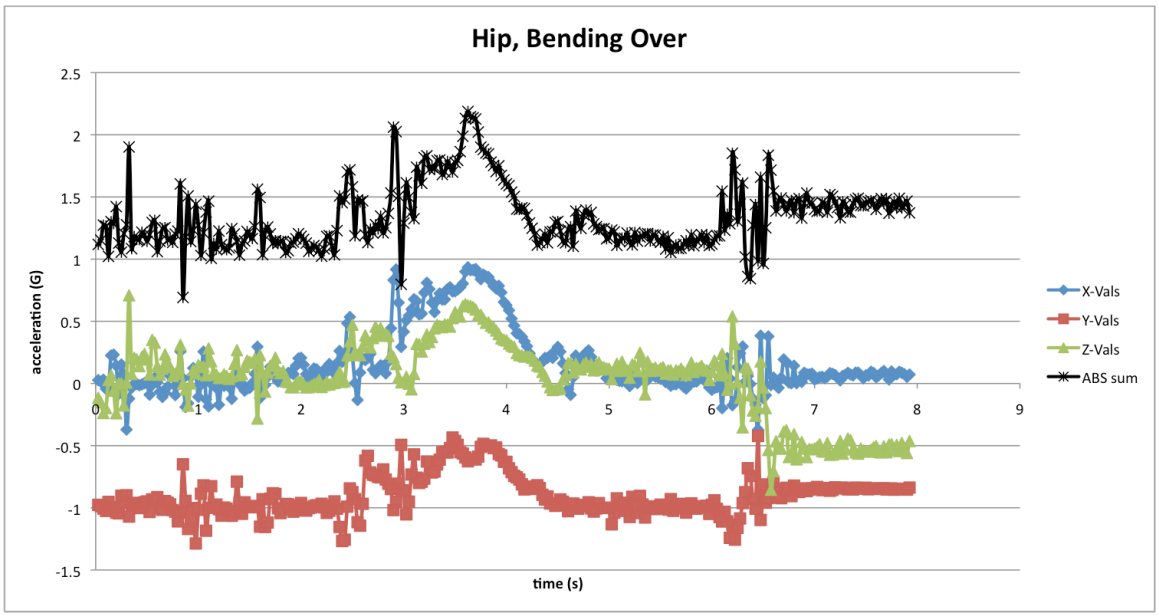

Figure 20 – Graph of accelerations in each axis while bending over, with hip attachment. Included in black are the sums of the absolute accelerations in the three axes.

Figure 20 shows the same test, except with the phone now attached firmly to the user's hip. This placement shows even less accelerometer disturbance, as the absolute aggregate value barely exceeds two g-forces.

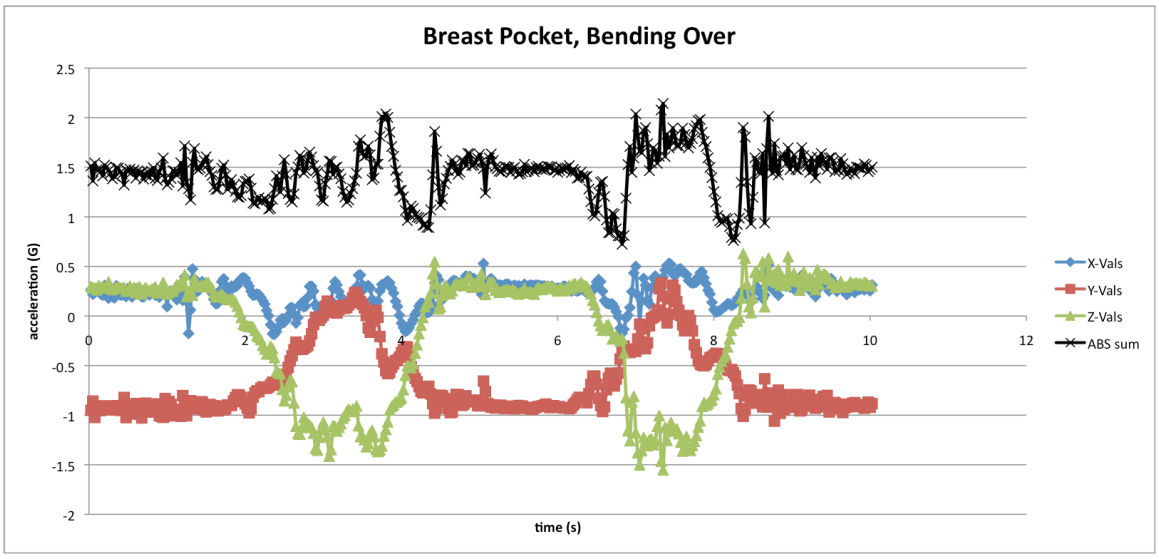

Figure 21 – Graph of accelerations in each axis while bending over, with breast pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 21 shows the user bending over with the phone placed in the breast pocket. Counter intuitively, this placement shows the least amount of accelerometer disturbance of the three positions. It was anticipated that this placement would show the greatest accelerometer displacement for this test, considering that bending over results in more movement occurring at the chest than at the hip or thigh. However, it seems that the smooth movements seen in the upper torso prevent large accelerations from being registered.

3.7 Sitting Down

Another very common ADL, and one that was identified as being potentially problematic, is sitting down. For this test, it was difficult to anticipate which placement would fair the best.

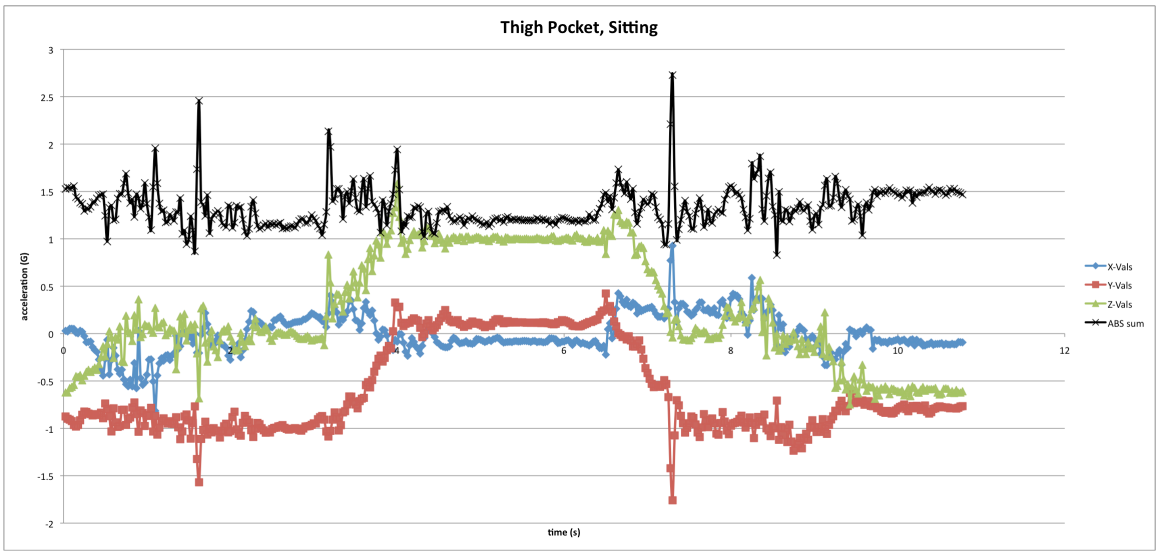

Figure 22 – Graph of accelerations in each axis while sitting down, with thigh pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 22 shows the data collected with the phone placed in the subject's thigh pocket while sitting down and then standing back up. For the thigh pocket placement, this ADL caused less accelerometer disturbance than the previous tests of walking and bending over.

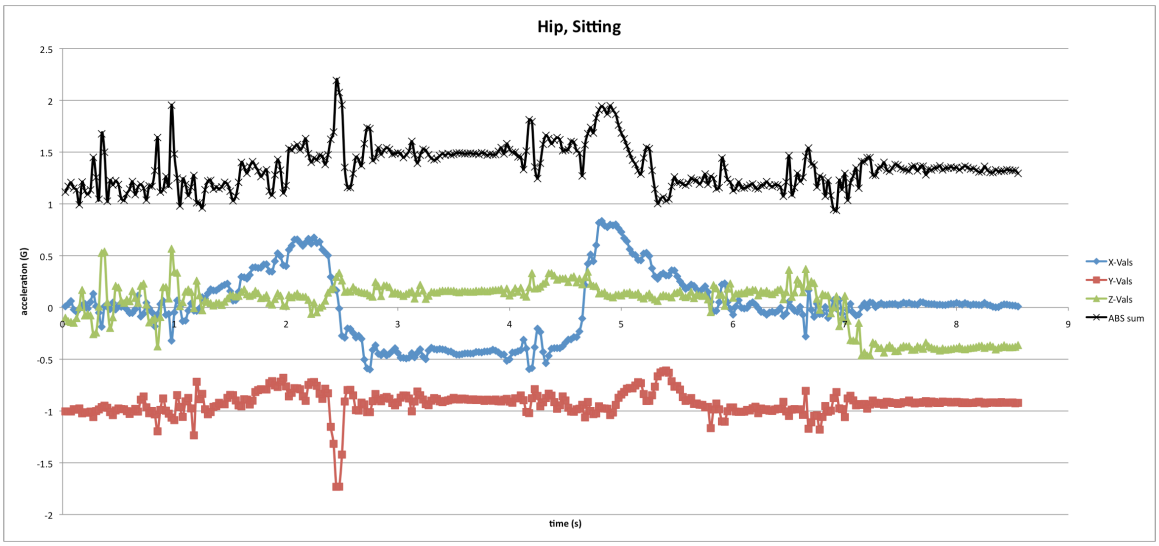

Figure 23 – Graph of accelerations in each axis while sitting down, with hip attachment. Included in black are the sums of the absolute accelerations in the three axes.

Figure 23 shows the same test, but this time with the hip attachment. The same trends from previous ADL testing hold true here, as the maximum acceleration recorded is less than that seen with the thigh pocket placement.

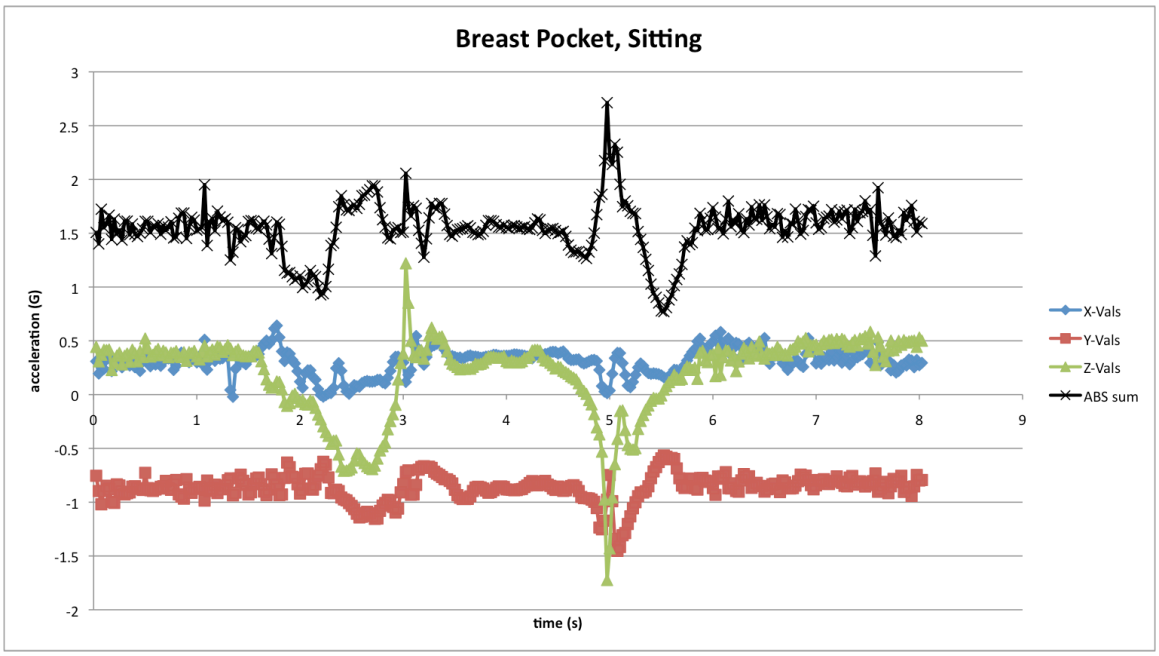

Figure 24 – Graph of accelerations in each axis while sitting down, with breast pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 24 shows the same test with the phone placed in the user's breast pocket. The actual act of sitting down has little effect on the accelerations. However, when the user stands back up, we observe a slight spike in the absolute aggregate acceleration. This spike in the absolute aggregate is primarily due to a negative spike in the z-axis, caused from the user bending forward and raising their body upward at the same time. This activity still only resulted in a moderate accelerometer disturbance of just over 2.5 g-forces.

3.8 Other ADL

The previous three ADL were identified as the main three of interest due to their frequency of occurrence. However, for the sake of due diligence, several

more ADL were tested with some of the device placement positions to ensure there weren't any significant and non-intuitive results.

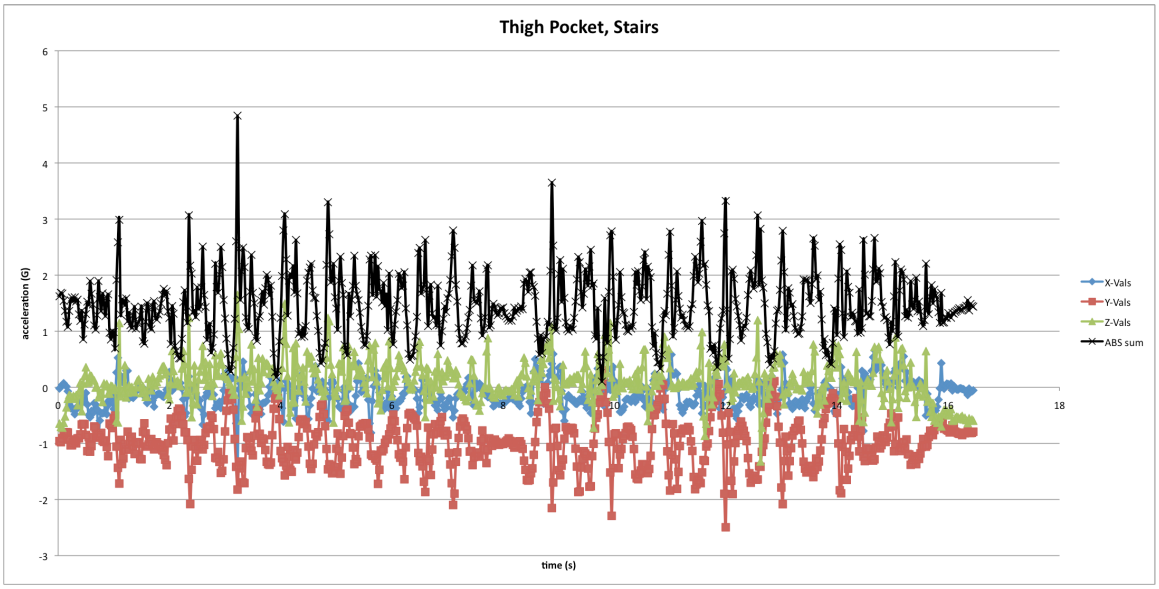

Figure 25 – Graph of accelerations in each axis while descending and ascending stairs, with thigh pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Figure 25 shows a test with phone placement in a thigh-pocket with the user descending and ascending a flight of stairs. This test reaffirms the thigh-pocket placement as potentially unsuitable. We can see significant accelerometer disturbance in the aggregate value, as it exceeds three g-forces numerous times and even approaches five g-forces on one occasion.

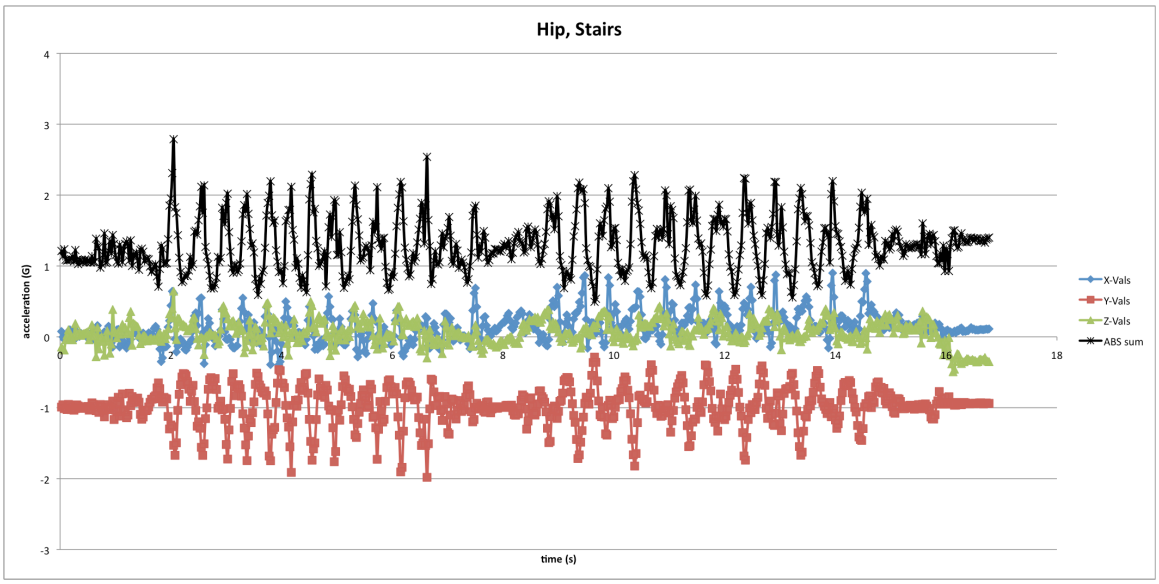

Figure 26 – Graph of accelerations in each axis while descending and ascending stairs, with hip attachment. Included in black are the sums of the absolute accelerations in the three axes.

This test was repeated with the hip placement to ensure that the act of ascending and descending the stairs itself wasn't responsible for the acceleration spikes. This notion was confirmed, as we can see in Figure 26; the accelerations seen for this phone placement were much more moderate than those seen with the thigh pocket placement. In fact, for both phone placements, the act of ascending and descending the stairs affected the accelerometer less than for walking.

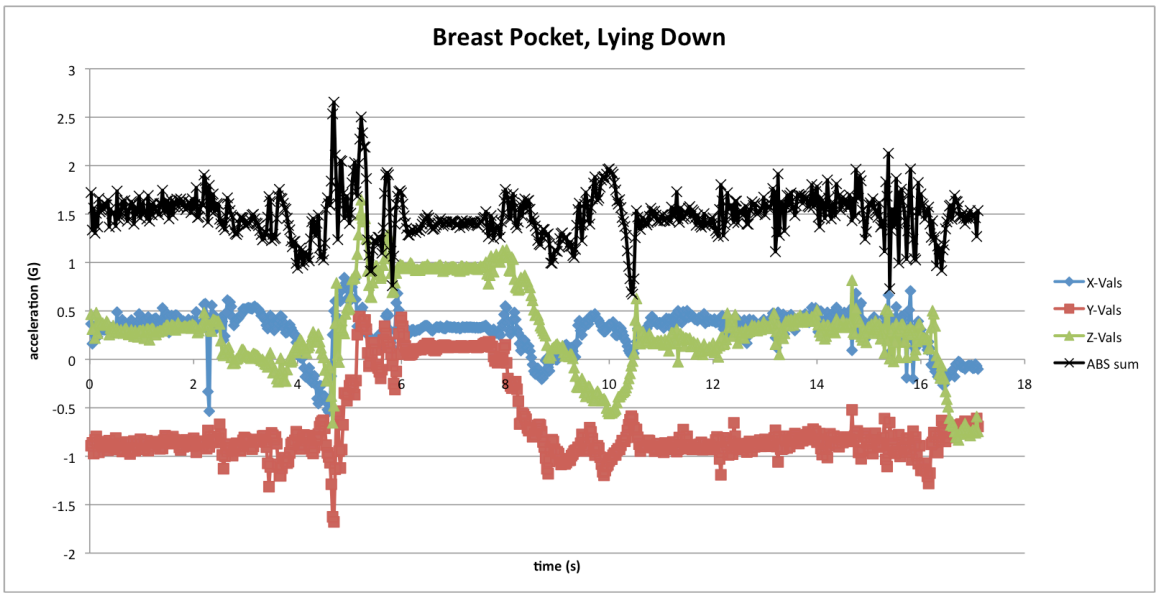

Figure 27 – Graph of accelerations in each axis while lying down, with breast pocket placement. Included in black are the sums of the absolute accelerations in the three axes.

Previous ADL testing seemed to highlight the breast pocket placement as potentially being the most ideal placement. In light of this placement potentially being the best option, it was important to ensure that the act of lying down wouldn't cause significant disturbance with this placement. Figure 27 above shows the results from this test. With absolute aggregate accelerations barely exceeding 2.5 g-forces, this placement remains the likely best option.

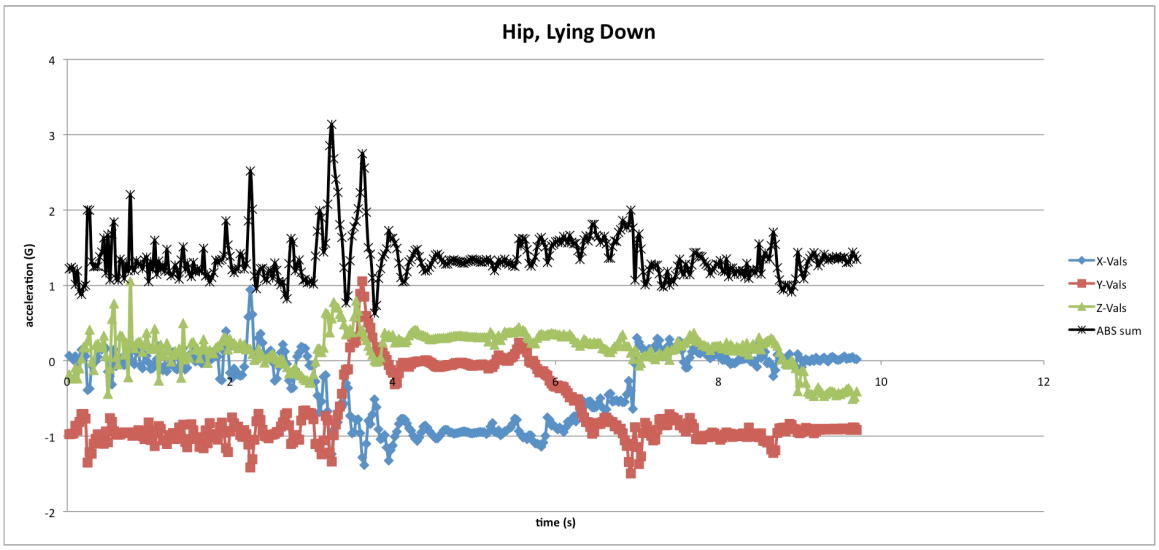

Figure 28 – Graph of accelerations in each axis while lying down, with hip attachment. Included in black are the sums of the absolute accelerations in the three axes.

For the sake of comparison, this test was repeated again with the hip placement, seen in Figure 28 above. Again, we see that the breast pocket placement outperforms the hip placement, as the absolute aggregate accelerations peak over three g-forces with the hip placement.

3.9 Fall Testing

This section describes the extensive fall testing that was performed and discusses the data collected during this testing. The hip attachment was used to characterize a number of different falls that were determined to be plausible; as well as to identify some potential "worst-case scenario" falls. In this instance, "worst-case-scenario" does not mean a fall with the potential to cause the greatest injuries. On the contrary, the worst-case scenario here is a fall that could potentially occur while causing minimal disturbance to the phone's accelerometer. This is the scenario in which a fall could potentially go unnoticed

by the application if the threshold value is set too high. These worst-case scenario falls were repeated with the other placement positions for comparison.

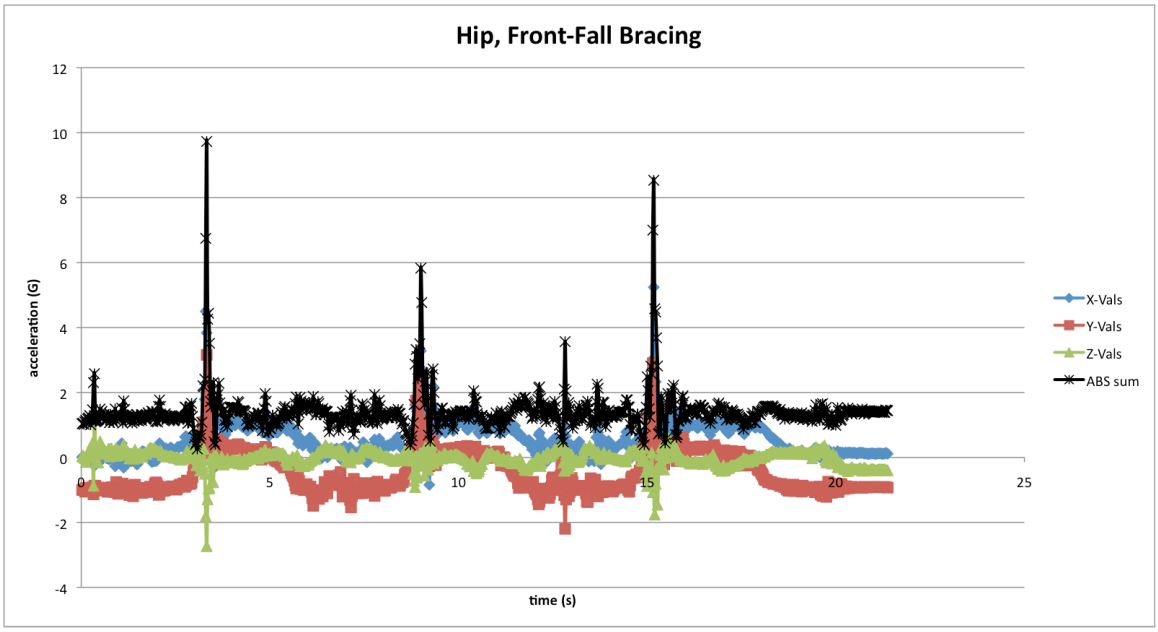

Figure 29 – Graph of accelerations in each axis during three performed falls in the forward direction, with hip attachment and using the arms to lessen the impact. Included in black are the sums of the absolute accelerations in the three axes.

Figure 29 shows data collected with the user falling forward and bracing for impact using their arms. For this test, and all subsequent fall testing, the subject fell onto a padded mat. This was done mainly to prevent injury to the test subject. While this safety precaution makes for a slightly less realistic fall scenario, it does not falsely prop up system performance. In fact, this test scenario is more challenging for the application than a real world scenario; as the mat has the effect of lessening the accelerations experienced by the phone, and therefore makes the falls more difficult to distinguish from ADL.

This test was meant to simulate a fall if the user were to trip, stumble and fall forward, but be able to extend their arms in an attempt to lessen the fall. These are complicated motions and events to repeat, and hence they register differently on the phone's accelerometers, which is reflected in the absolute aggregate accelerations with peaks of 9.7, 5.8, and 8.5 g-forces for the three falls respectively. Despite the variation, all three falls register as significant events by the accelerometer.

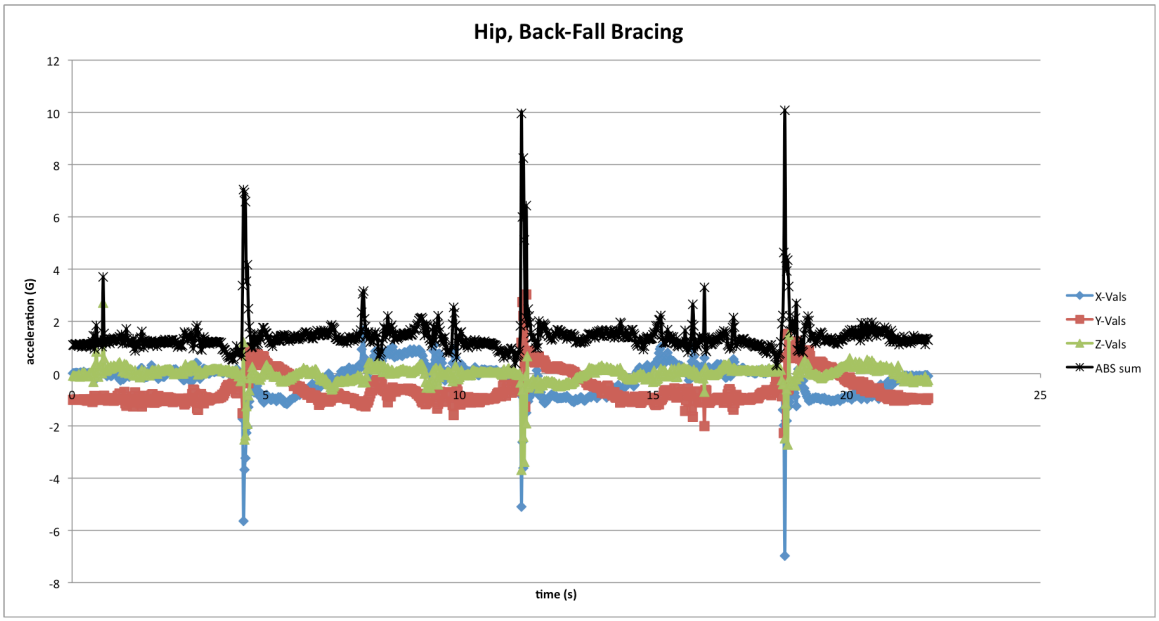

Figure 30 – Graph of accelerations in each axis during three performed falls in the backward direction, with hip attachment and using the arms to lessen the impact. Included in black are the sums of the absolute accelerations in the three axes.

Figure 30 above shows the results from a test that simulated a user falling backward and bracing for impact using their arms. Again, three falls were simulated. We can see the absolute aggregate acceleration reaches 7 g-forces on the first fall and exceeds 10 g-forces on the subsequent two falls.

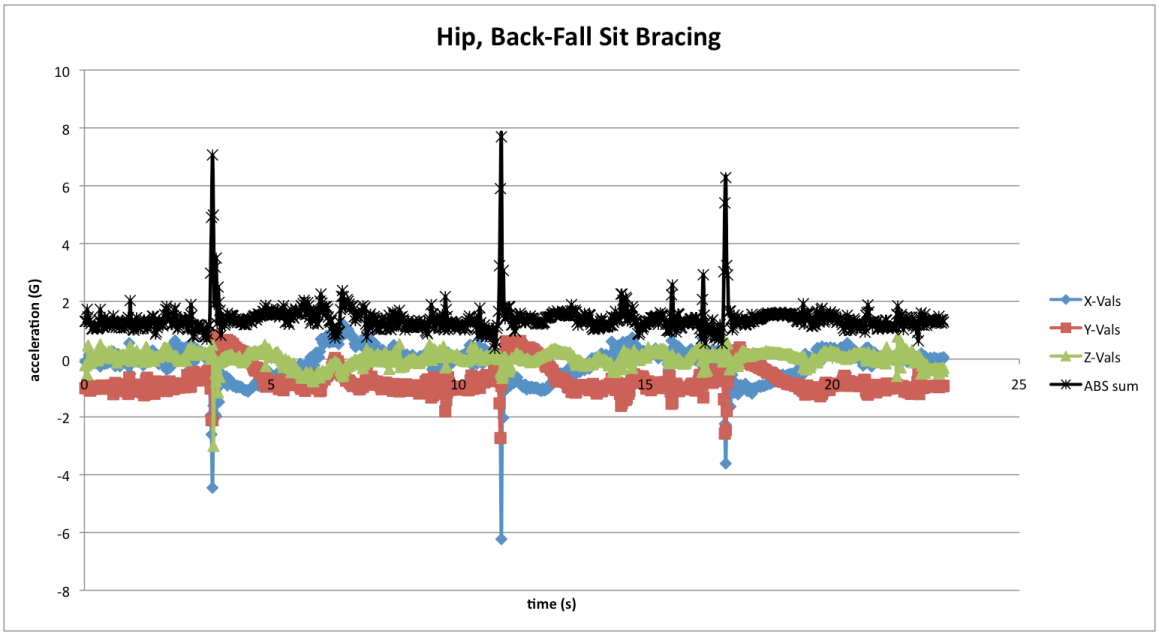

Figure 31 – Graph of accelerations in each axis during three performed falls in the backward direction, with hip attachment and using the arms to lessen the impact. The user comes to rest in a sitting position for this test. Included in black are the sums of the absolute accelerations in the three axes.

Figure 31 shows the results of a test that was similar to the previous test in that it involved the user bracing for a fall backward. This test, however, simulated the user coming to rest in a sitting position. As we can see from the graph, this slightly decreases the accelerometer disturbance, with the three falls registering between six and eight g-forces.

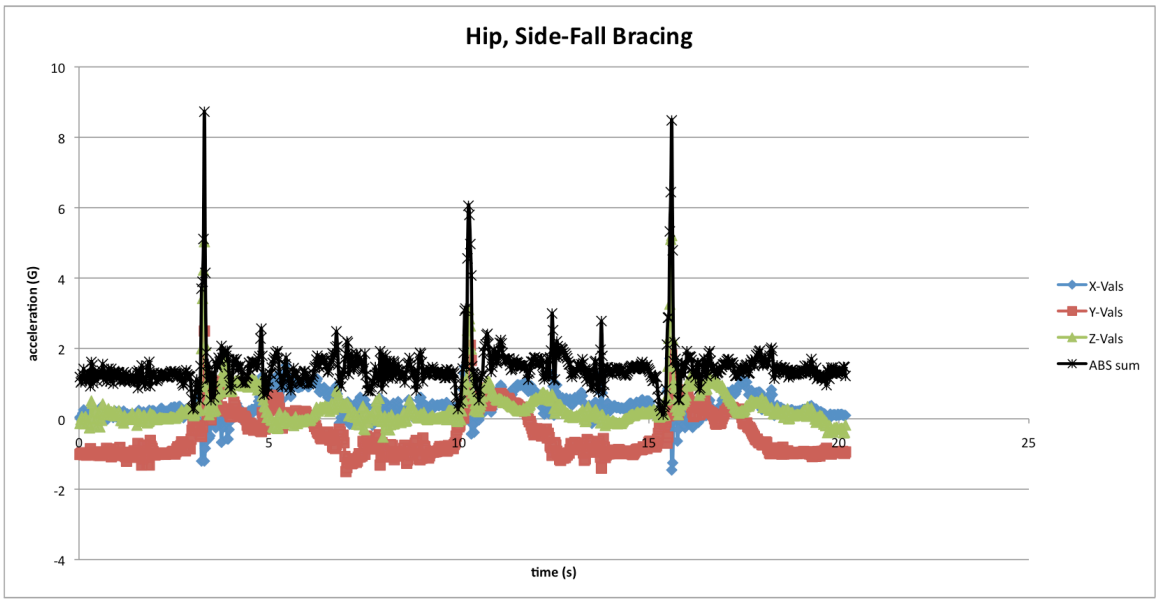

Figure 32 – Graph of accelerations in each axis during three performed falls in the sideways direction, with hip attachment and using the arms to lessen the impact. Included in black are the sums of the absolute accelerations in the three axes.

Figure 32 shows data collected with the user falling sideways and bracing for impact. Again, we observe significant spikes in the absolute aggregate acceleration.

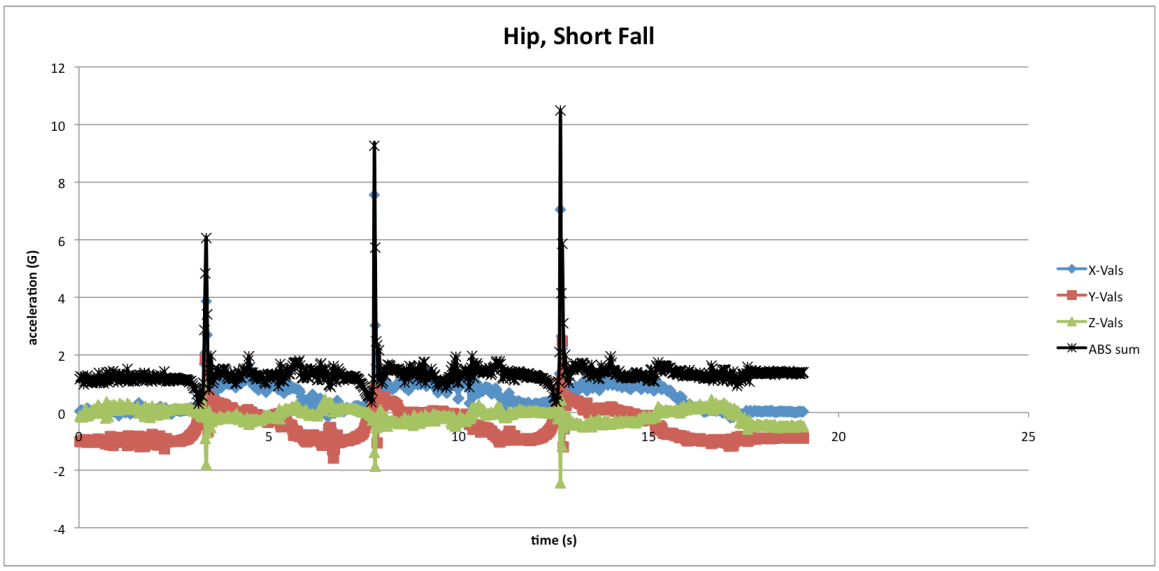

Figure 33 – Graph of accelerations in each axis during three performed falls forward with the subject starting on the knees. Included in black are the sums of the absolute accelerations in the three axes.

Figure 33 represents an attempt to simulate a worst-case scenario fall. This would be a fall in which the user is already in close proximity to the floor. For instance if the user were bending down or kneeling to pick something up and lost balance or consciousness. To simulate this worst-case scenario, the test subject started in a position on the knees and fell forward onto a padded mat. We can see in the graph, even with this short, low-impact fall onto a padded surface, there is a significant accelerometer disturbance.

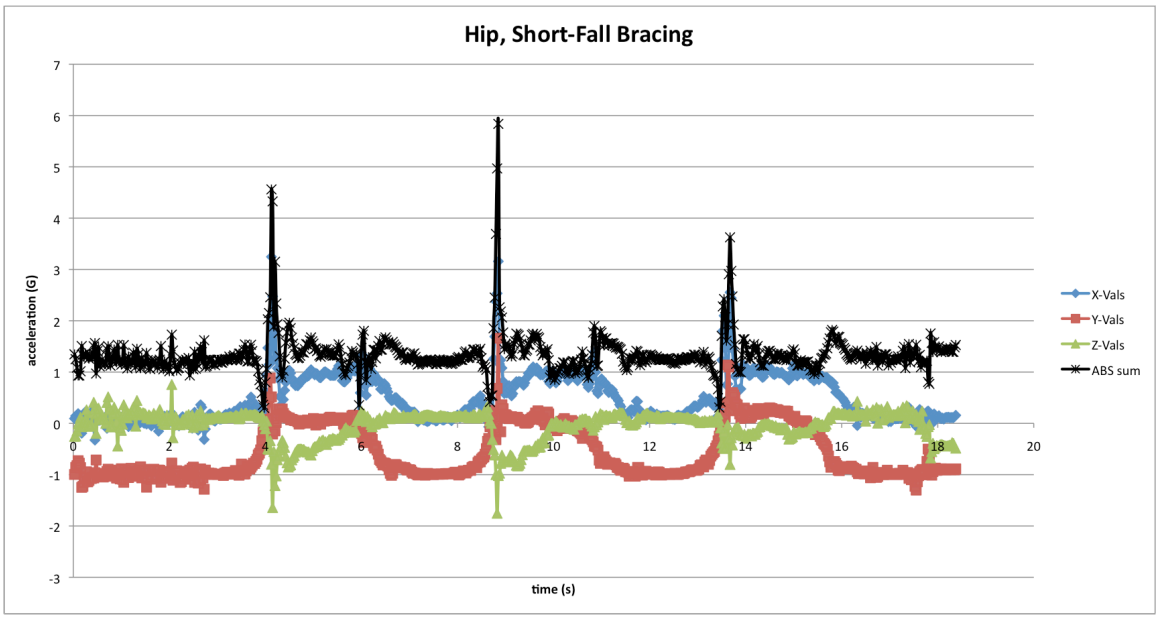

Figure 34 – Graph of accelerations in each axis during three performed falls forward with the subject starting on the knees and bracing for impact with the arms. Included in black are the sums of the absolute accelerations in the three axes.

Figure 34 represents an attempt to take the worst-case scenario even further. While the previous test simulated a very short fall where the user potentially loses consciousness (a situation where a distress call would be warranted and potentially vital) this test represents the same short, low-impact fall, but with the user retaining consciousness and therefore being able to brace for the impact with their arms. It is very unlikely that this type of fall would result in injury and even less likely that it would render the user incapable of operating their phone. In other words, this test goes beyond what would be considered a plausible fall event that the app would need to detect and alert emergency contacts. However, it is a worthwhile test because if the system can reliably detect this type of fall, then it should have no problem detecting any real world fall.

You can see from the graph that this test resulted in the lowest accelerometer disturbance of any of the fall tests, validating its design as the worst-case scenario. In fact, the third fall only recorded an absolute aggregate acceleration spike of 3.62 g-forces, almost identical to the maximum absolute aggregate acceleration seen for the hip attachment during walking, 3.61 g-forces. This indicates that the hip attachment position could experience excessive false alarms if the threshold value was set with the aim of detecting excessively minor falls. It's possible that greater reliability is achievable with one of the other placement positions; specifically the breast pocket placement, since it showed lower accelerometer disturbance during all ADL.

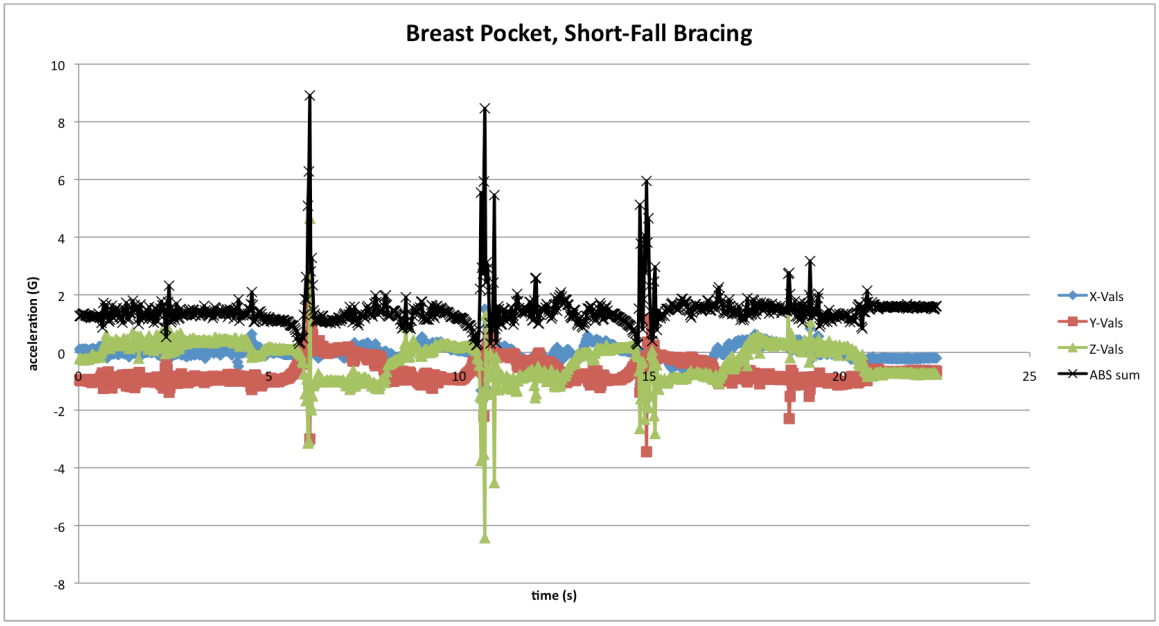

Figure 35 – Graph of accelerations in each axis during three performed falls forward with the subject starting on the knees and bracing for impact with the arms. Included in black are the sums of the absolute accelerations in the three axes.

Figure 35 shows the worst-case scenario test repeated with the breast pocket placement. This test showed greater amplitude than for the hip attachment position.

3.10 ADL vs Fall: Differential

As has been mentioned, the main criterion for a reliable fall detection system is the ability to distinguish between ADL and fall events. For a threshold based system, this means that there needs to be a significant and consistent difference between the accelerations experienced by the device during ADL, and those experienced during falls. This metric will be the key to determining which placement position is the most advantageous. We compared the walking ADL with the short fall bracing fall event, as walking produced the greatest accelerometer disturbance of the ADL, and the short fall bracing test represents the worst-case scenario fall.

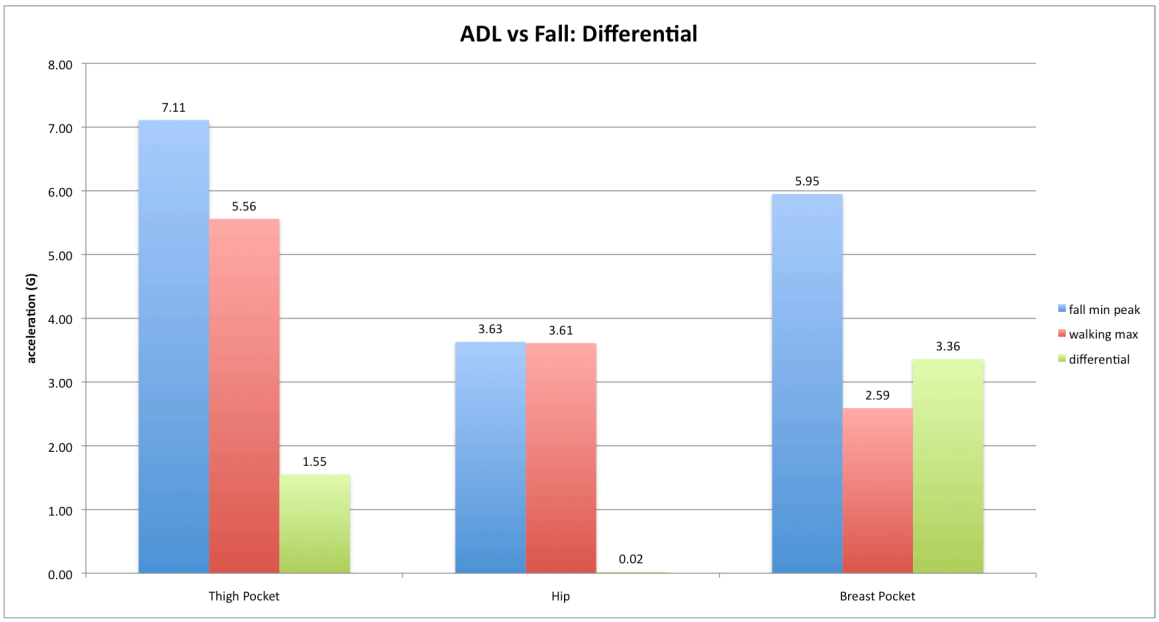

Figure 36 – Graph comparing the minimum fall impact acceleration and maximum walking acceleration for the three placement positions. The third bar for each position shows the differential between the maximum ADL and minimum fall impact.

Figure 36 illustrates the comparison of ADL and falls with respect to the three placement positions that were tested. Shown in the graph for each position is the maximum absolute aggregate acceleration recorded during walking (the most energetic ADL), the minimum peak absolute aggregate acceleration recorded during the short-fall-bracing test (the least energetic, worst-case scenario fall test), and the difference of these accelerations. This difference is a key metric for determining how robust and reliable the system will be with the given placement position. If the difference between these values is substantially large, then it is feasible for a threshold to be set which will easily distinguish between any conceivable fall and normal ADL. This comparison provides some surprising results. Firstly, the thigh pocket placement actually showed a fairly significant differential of more than one and a half g-forces, making it a potentially

viable placement position. More surprising than that, however, was that the hip placement position showed virtually no difference between the maximum acceleration recorded during walking and the minimum peak acceleration recorded for a fall. This means that there is basically no ideal threshold. Either the threshold is set high enough to avoid excessive false alarms during walking and risks a fall occurring that is undetected; or the threshold is set low enough to detect any potential falls, but significant false alarms occur during walking. Both of these results are unacceptable. The breast pocket placement, although recording a smaller peak fall acceleration than the thigh-pocket placement, had a much lower maximum acceleration during walking, and therefore had the largest differential. With a differential of over 3 g-forces, there is plenty of room to set a threshold value that should register any potential fall while also causing few to no false alarms. These results are sufficient to declare the breast pocket placement as the attachment method of choice.

Furthermore, these results are enough to select a threshold value for the application. With breast pocket placement, and using the sum of the absolute values of the three axial accelerations as the fall detection metric, a threshold of four g-forces is selected. This threshold is almost 1.5g-forces greater than the maximum acceleration experienced during ADL testing, a sizeable margin to keep false alarms to a minimum. It is also nearly two g-forces below the smallest acceleration spike recorded during fall testing. This margin should ensure that no fall goes undetected.

4 DESIGN IMPLEMENTATION

The following sections detail the implementation of the final fall detection application. The design is broken up into two main parts: the user interface, and the app's inner workings and how the app logic is organized into its various classes.

4.1 User Interface

Designing a good user interface is a critical early step in creating a successful application. The user needs to be able to access all the app features while being able to interact with the app in the simplest way possible. The user interface needs to be designed with the user in mind to make it efficient, clear, and straightforward.

Apple's integrated development environment, Xcode, provides a tool called storyboards, which simplifies the process of implementing the user interface [33]. This tool allows the developer to design and implement the user interface in a single step, in a graphical environment. It allows for immediate feedback on what is working and what isn't, and changes are instantly visible [15].

A user interface is constructed using a number of different objects, paramount of which are views and view controllers. Views essentially include everything that the user sees and interacts with onscreen, from text and images

to buttons and other controls [32]. View controllers manage sets of views onscreen and can also manage other view controllers [31].

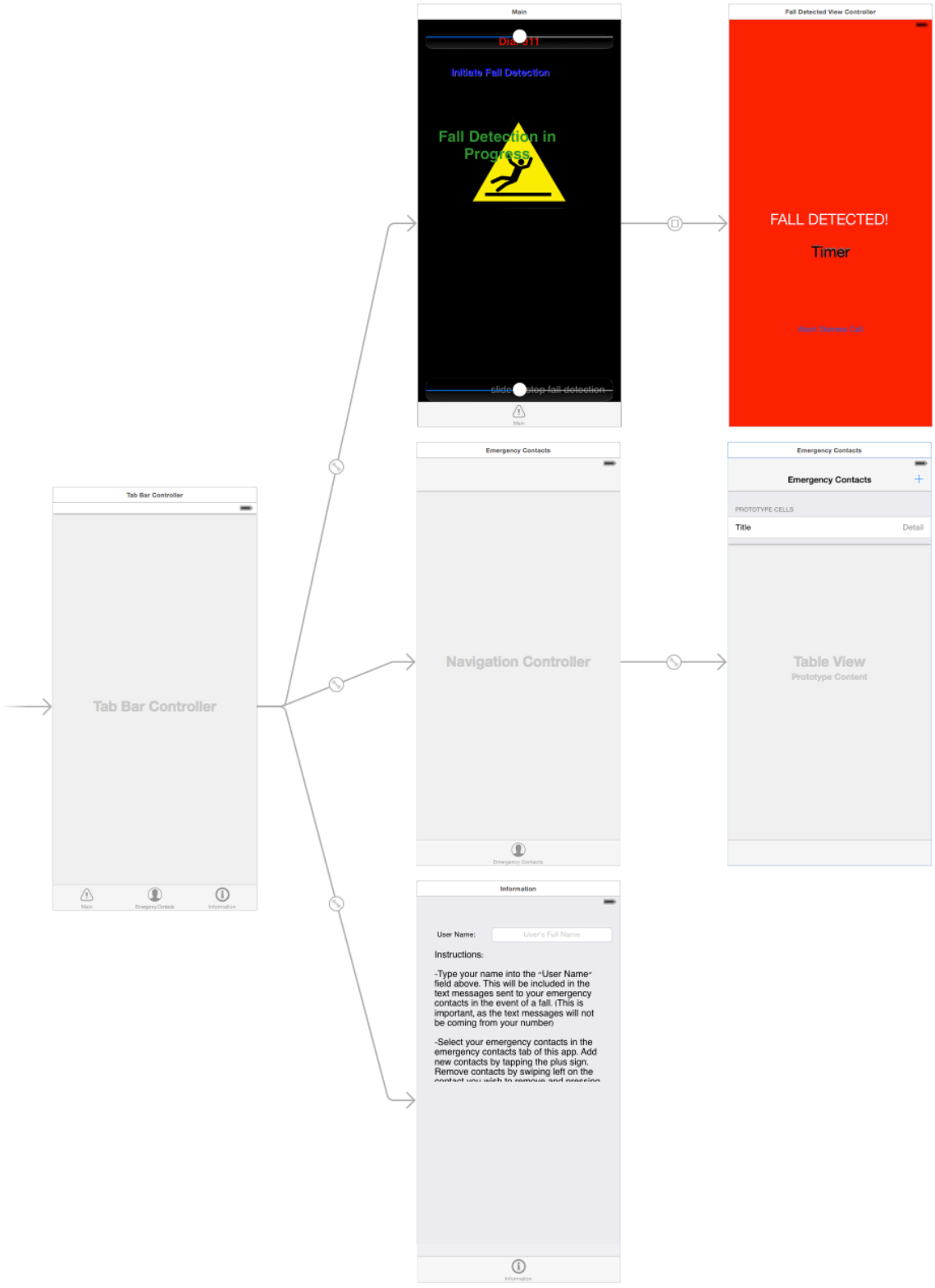

Figure 37 – The storyboard layout of the Fall Detection application's user interface.

Figure 37 shows the storyboard layout for this project. It includes most of the views and view controllers that comprise the app's user interface, as well as the connections between these objects. Not all of the views from the app are included here because some are created programmatically as opposed to being created using storyboards. The basic infrastructure for the app's user interface, however, is built from this storyboard.

At the left of the figure is a view controller known as a tab bar controller. This is a special kind of container view controller. Container view controllers contain content owned by other view controllers [30]. These other view controllers are explicitly assigned to the container view controller as its children [31]. A tab bar controller is a container view controller used to divide an app into several distinct modes of operation [30, 31]. Selecting a tab causes the tab bar controller to display the associated view controller's view on the screen. In this app, the tab bar controller allows for easy navigation between the app's main screen and two additional screens: one for selecting emergency contacts, and one that lists instructions for operating the app. The tab bar controller provides tabs at the bottom of the screen with icons and titles for moving between screens.

At the top of the figure is the content view controller for the app's main screen. This view controller contains numerous views, which include several controls: a button and two sliders. The button allows the user to activate fall detection by initiating accelerometer sampling. The two slider controls are

configured in the style of the iPhone's "slide to unlock" slider. This prevents these controls from being activated unintentionally, as this screen will be active while the phone is in the user's pocket. The slider at the bottom of the screen becomes visible once the user activates fall detection, and its purpose is to deactivate fall detection. The slider at the top of the screen is an emergency call slider that automatically dials 911 for the user in the event of an emergency.

At the top right of figure 37 is the fall detected view controller. This content view controller is presented modally in the event of a detected fall. This is to say that its collection of views will slide onscreen overtop of the main screen when it is presented. This view controller contains several views and one control. The screen displays a "FALL DETECTED!" label as well as the time remaining on a 30 second timer, which gives the user time to abort the distress call and messages in the event of a false alarm. The user accomplishes this via a button, labeled "Abort Distress Call."

In the center of the figure is the Emergency Contacts navigation controller. A navigation controller is another special type of container view controller [30]. This type of container manages a stack-based collection of content view controllers [30, 31]. In our case this container view controller manages a content view controller responsible for displaying the list of emergency contacts as well as the content view controller where the user can select which contacts to include in the list. While the management of these child view controllers is the navigation controller's primary job, it is also responsible for managing several views of its

own. Specifically, the controller manages the navigation bar at the top of the screen, which includes the button for adding a new contact to the list and a button to cancel the addition of a new contact.

In the right center of figure 37 is the Emergency Contacts table view controller. This is a special type of content view controller designed specifically for managing tabular data [30, 31]. It has built-in support for many standard table-related behaviors such as selection management, row editing, and table configuration [31]. For this app, the table consists of a primary contact and four additional contacts. This view controller has a control at the top right of the screen for adding contacts to the table. When the user selects this control, another view controller is presented onscreen. This view controller is not shown in the figure because it is created programmatically, rather than through storyboards. The view controller is an instance of the ABPeoplePickerNavigationController class. This content view controller manages a set of views that allow the user to select a contact or one of its contact-information items (such as a phone number) from the address book.

Lastly, at the bottom of the figure is the information view controller. This is a content view controller that displays a scrolling text field view containing information about how to operate the app. Additionally, this view controller has a control for the user to enter his/her name. This is important for the proper functioning of the app. When the distress text messages are sent out to the emergency contacts via Twilio, they will be coming from a Twilio phone number

as opposed to the user's cell number. By adding the user's name in this text field, the distress text messages can include who the user is to eliminate confusion.

4.2 App Logic

A user interface can't do much without any programming instructions backing it. After the user interface is created, the developer defines how the user can interact with what they see by writing code to respond to user actions in the interface.

IOS apps are based on event-driven programming, where the flow of the app is determined by events: system events or user actions [6, 16, 27]. The user performs actions on the interface, which trigger events in the app. These events result in the execution of the app's programming instructions and manipulation of its data. The app's response to user action is then reflected back in the interface.

The following sections briefly describe the custom classes that contain the programming instructions driving the fall detection application.

4.2.1 AppDelegate

The app delegate is the heart of any app's custom code. Specifically, the methods of the app delegate give the app a chance to respond to important changes, such as state transitions and incoming notifications [4]. It is a subclass of the UIResponder class and conforms to the UIApplicationDelegate protocol. The UIResponder class defines an interface for objects that respond to and

handle events. The UIApplicationDelegate protocol defines methods that are called by the singleton UIApplication object in response to important events in the lifetime of the app. The app delegate works alongside the app object to ensure the app interacts properly with the system and with other apps.

The app delegate is effectively the root object of any app. Like the UIApplication object, the app delegate is a singleton object and is always present at runtime. A singleton is a class for which there exists only a single instantiated object. This object is generally used to model a shared resource. Although the UIApplication object does most of the underlying work to manage the app, the developer decides the app's overall behavior by providing appropriate implementations of the app delegate's methods [4]. The app delegate performs several crucial roles:

- It contains the app's startup code
- It responds to key changes in the state of the app. Specifically, it responds to both temporary interruptions and to changes in the execution state of the app, such as when it transitions from the foreground to the background
- It responds to notifications originating from outside the app, such as remote notifications (also known as push notifications) low-memory warnings, and more
- It determines whether state preservation and restoration should occur and assists in the preservation and restoration process as needed

- It responds to events that target the app itself and are not specific to the app's views or view controllers
- It can be used to store the app's central data objects or any content that doesn't have an owning view controller

The Fall Detection app delegate implements a very important method for creating and accessing a singleton instance of the CMMotionManager class. This object is the application's gateway to the motion data originating in the phone's accelerometers.

4.2.2 ViewController

The ViewController class is a custom subclass of UIViewController and is the view controller for the main screen of the app's user interface. The UIViewController class provides the infrastructure for managing the views of an iOS app [4]. Specifically view controllers manage a set of views that make up a portion of the app's user interface. They are responsible for loading and disposing of those views, for managing the interactions with those views, and for coordinating responses with any appropriate data objects. View controllers also coordinate their efforts with other controller objects—including other view controllers—and help manage the app's overall interface [31]. A view controller's main responsibilities include:

- Updating the contents of the views, usually in response to changes in the underlying data

- Responding to user interactions with views
- Resizing views and managing the layout of the overall interface

A view controller is tightly bound to the views it manages and takes part in the responder chain used to handle events. View controllers are also UIResponder objects and are inserted into the responder chain between the new controller's root view and that view's superview, which typically belongs to a different view controller. If none of the view controller's views handle an event, the view controller has the option to handle the event or pass it along to the superview [31].

The main view controller here has a number of responsibilities. First and foremost, it controls all the views and their behavior in the app's main screen. This includes two sliders, a logo, a background, and a button to initiate fall detection. The view controller presents these views and controls their behavior. For instance, when fall detection is initiated, the background color fades from grey to black, the "Initiate Fall Detection" button disappears, the "Stop Fall Detection" slider appears, and a message is displayed onscreen, notifying the user that fall detection has been initiated.

The view controller also provides critical functionality to the controls displayed on the main screen. At the top of the screen is a slider for the user to activate in the case of an emergency where a fall is not registered. When activated, the view controller displays a message onscreen that asks the user if

they are in need of emergency services. The user can cancel the request and dismiss the message by selecting “No.” If the user selects “Yes,” then the view controller uses a Universal Resource Locator (URL) to open the iPhone’s telephone application and place an outgoing call to 911.

Another critical function of the app is controlled from within this view controller. It is responsible for sampling the phone’s accelerometer, analyzing that data, and declaring when a fall has occurred. When the user initiates fall detection, via the button on the main screen, a special method is triggered within the view controller. This method sets the accelerometer update interval and starts the updates on an operation queue with a specified handler. This handler is a block that is invoked with each update to handle the new accelerometer data. When this block detects an acceleration consistent with a fall, the view controller performs a segue to the fall detected view controller, thereby bringing it onscreen.

4.2.3 FallDetectedViewController

This is the view controller presented onscreen when a fall has been detected by the main view controller. It is a subclass of the UIViewController class and conforms to the TCDeviceDelegate protocol [70]. This custom view controller class has a number of important responsibilities within the app, in addition to every view controller’s usual responsibilities of managing its views onscreen and coordinating with data and controller objects.

This view controller manages views for a “Fall Detected!” label, an “Abort Distress Call!” button, and a label displaying the time remaining on a 30-second timer. The button simply dismisses the view controller and returns control to the main view controller. The view controller is responsible for several things in relation to the 30-second timer. The stored value for the time remaining must be decremented and the view displaying it updated with the new value. The view controller also activates the phone’s vibrate function every second during the countdown to help notify the user in case of a false alarm. Additionally, an audio file is played three times during the countdown, with an audio message alerting the user of a detected fall.

The most crucial functionality contained within this view controller occurs when the 30-second timer expires. It is responsible for programmatically sending text messages to the user’s emergency contacts and placing a phone call to the user’s primary emergency contact. It does this by first initiating a connection with Twilio, retrieving a capability token from the custom web server, and initializing the phone as a TCDevice, using the capability token. An instance of TCDevice is the object that knows how to interface with Twilio services [70]. The view controller then provides Twilio with a list of parameters for the connection, which include the user’s name and the phone numbers for the emergency contacts. These parameters are routed through Twilio to the Python script on the web server, which triggers the text messages to be sent by Twilio. The view controller

then opens the iPhone's telephone application and places a call to the user's primary contact.

4.2.4 EmergencyContacts

This is the view controller presented onscreen when the user selects the emergency contacts tab. This class inherits from UITableViewController and conforms to the ABPeoplePickerNavigationControllerDelegate protocol. The UITableViewController class creates a controller object that manages a table view [30]. Subclassing UITableViewController gives the custom class support for common table formatting, as well as providing some basic methods, such as row editing.

Conforming to the ABPeoplePickerNavigationControllerDelegate protocol allows the emergency contacts view controller to properly communicate with the ABPeoplePickerNavigationController that it creates. When the user selects the plus symbol at the top right of the screen, this triggers an action message to be sent to this view controller. It then creates a new instance of an ABPeoplePickerNavigationController, sets itself as that controller's delegate, and then presents the view controller onscreen. When the user selects a phone number from their contacts, that number and the contact's name are relayed back to the view controller, as the ABPeoplePickerNavigationController's delegate.

The table view controller displays the name and phone number of the selected emergency contacts in two sections. Displayed in one section by itself is the primary contact, whom will receive a distress text message and a phone call in the event of a fall. Below the primary contact is a second section displaying the list of four additional contacts, whom will only receive a distress text message.

4.2.5 InformationViewController

This is the view controller presented onscreen when the user selects the information tab. This custom class is a subclass of UIViewController and conforms to the UITextFieldDelegate protocol. This protocol defines how a text field communicates with its delegate view controller.

This view controller serves a few basic purposes. First, it displays a scrolling text field, which contains instructions for the user on how to properly operate the app. The other main function of this view controller is to provide a text field for the user to input his/her name. The view controller stores the text string for use in the distress text messages.

4.2.6 Contact

The Contact class is a very simple data object class for encapsulating the data associated with the emergency contacts. This class is a subclass of NSObject, which is the root class of most Objective-C class hierarchies and provides objects with the basic ability to behave as an Objective-C object and to

interface with the runtime system [21, 27]. The custom Contact class defines two properties for storing a contact's name and phone number.

4.2.7 MyManager

The MyManager class is a data source singleton object class, which manages the main data of the application. It is used throughout the code in order to retrieve objects that reside in persistent storage. This includes the emergency contact objects and the username string. These pieces of data are stored in a property list as XML data so that they can be saved to disk and preserved when the app is terminated.

It is the responsibility of the custom data manager to read in the property list from the disk and make its data available to other objects within the app. The MyManager class defines a singleton object so that there is only one instance of the object, which can be accessed from anywhere in the code.

5 VALIDATION

The following sections detail some of the steps taken to validate the final app design. To determine the sensitivity and specificity of the system, the protocol laid out by Noury et al was followed [55]. The tests are listed below in table 3. These tests were performed with the final configuration, which was determined by the previously discussed accelerometer testing. That is, an accelerometer update speed of 40Hz, the breast-pocket placement, and a fall detection algorithm comparing the sum of the absolute values of the three axial accelerations to a threshold value of four g-forces, were used.

Table 6 – Tests for the evaluation of fall detection systems [55].

Category	Name	Outcome
Backward fall (both legs straight or with knee flexion)	Ending sitting	**Positive**
	Ending lying	**Positive**
	Ending in lateral position	**Positive**
	With recovery	**Negative**
Forward fall	On the knees	**Positive**
	With forward arm protection	**Positive**
	Ending lying flat	**Positive**
	With rotation, ending in the lateral right position	**Positive**
	With rotation, ending in the lateral to the left position	**Positive**
	With recovery	**Negative**
Lateral fall to the right	Ending lying flat	**Positive**
	With recovery	**Negative**
Lateral fall to the left	Ending lying flat	**Positive**
	With recovery	**Negative**
Syncope	Vertical slipping against a wall finishing in sitting position	**Negative**
Neutral	To sit down on a chair then to stand up (consider the height of the chair)	**Negative**
	To lie down on the bed then to rise up	**Negative**
	Walk a few meters	**Negative**
	To bend down, catch something on the floor, then to rise up	**Negative**
	To cough or sneeze	**Negative**

Table 6, from “Fall Detection – Principles and Methods,” details 20 different tests to be performed on a fall detection device in order to calculate that device’s sensitivity and specificity. Ten of the tests have an expected positive fall detected outcome, while the other ten have an expected negative fall detected outcome; this is reflected in the rightmost column of the table. The column on the left shows the general category for each test, while the center column gives a more detailed description of each test. Note that “with recovery” indicates that a fall does not actually occur, as the subject recovers prior to falling; these tests essentially constitute the test subject stumbling but not falling.

5.1 Fall-Detection Testing (Sensitivity)

This section details the fall-detection testing that was performed on the final app design. This testing included ten of the twenty separate scenarios outlined by Noury et al. The expected outcome of each scenario was a positively detected fall. Each fall scenario was performed ten times, resulting in 100 performed falls. All falls were performed onto a padded surface, as not to injure the test subject.

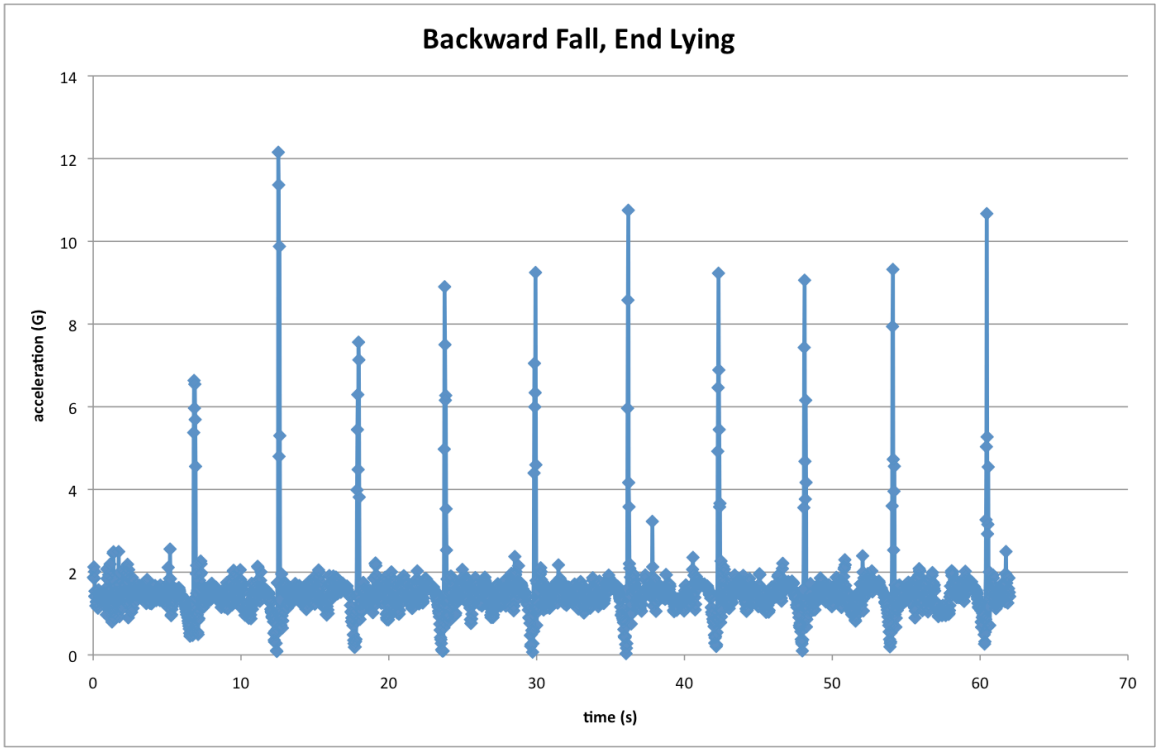

Figure 38 – Graph displaying absolute aggregate acceleration collected during 10 consecutive falls in the backward direction, with the test subject ending in a lying position.

Figure 38 above shows the results from the backward fall test, where the subject comes to rest lying on the floor. The data displayed is the absolute aggregate acceleration metric, which constitutes the sum of the absolute values

of the accelerations in each axis. We can clearly see ten accelerations spikes from the ten separate falls. Each fall registers over four g-forces, and therefore would be declared a positive fall, as was the expected result.

Figure 38 is representative of all the fall testing that was performed during the validation process. That is, the device was able to consistently determine when a fall had occurred. Of the 100 falls performed during testing, 100 true positives and 0 false negatives were recorded. These results were used to calculate the application's sensitivity:

$$Sensitivity = \frac{TP}{TP + FN} = \frac{100}{100 + 0} = 100\% \quad (3)$$

5.2 False Alarm Testing (Specificity)

This section details the false alarm testing that was performed on the system. It includes the remaining ten scenarios outlined by Noury et al. These ten scenarios had an expected negative outcome. In other words, the device should not register a fall while performing these activities. Again, each scenario was performed 10 times, resulting in 100 events recorded.

In addition to the scenarios laid out by Noury et al, extended false alarm testing was also performed. This involved the test subject wearing the device for an extended period of time while it monitored for falls. The subject was instructed to proceed with their normal daily activities with the exception of any athletic activities, such as running, jumping, etc.

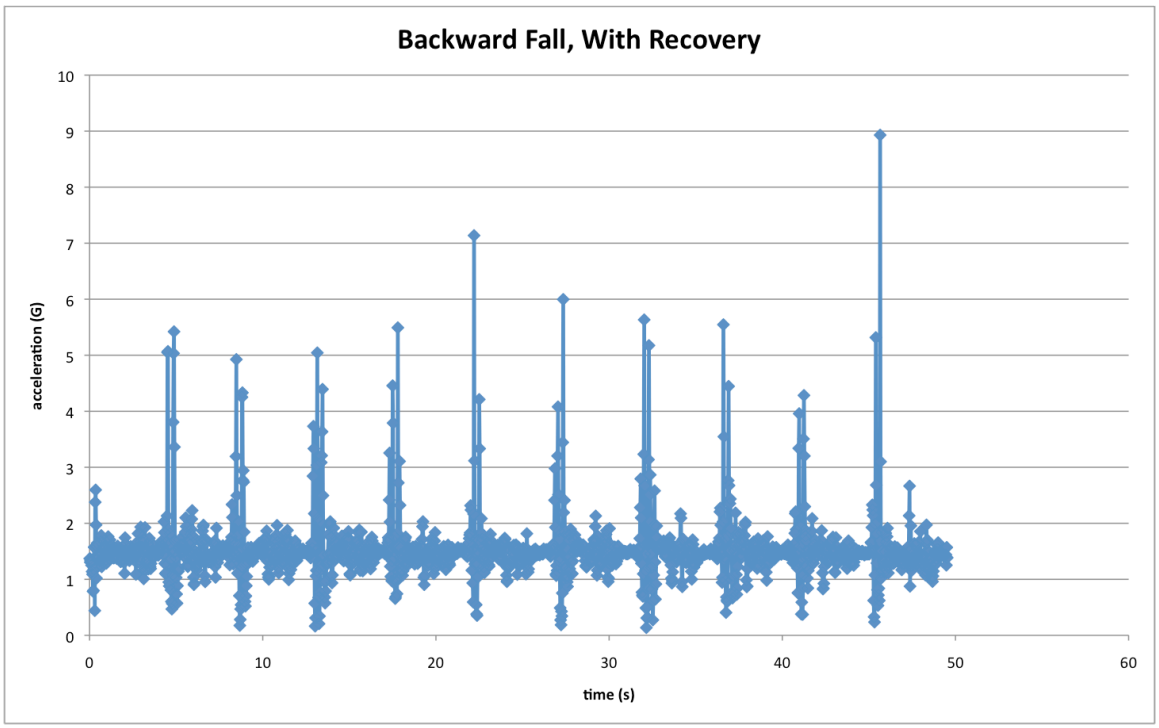

Figure 39 – Graph displaying absolute aggregate acceleration collected during 10 consecutive near falls in the backward direction, with the test subject recovering balance prior to falling.

Figure 39 above shows one of the non-fall scenarios outlined by Noury et al. This scenario involves a backward fall where the subject recovers prior to actually falling. That is, they essentially stumble and catch themselves. For this test, the test subject was instructed to lean back as far as they could, while still being able to regain balance and control. Noury et al determined this to be a scenario where the device should not detect a fall. By their standard, our application failed this test, as it recorded accelerations exceeding the fall threshold on all 10 fall-recovery events. It is debatable, however, whether this result really constitutes a failure. While it is important for the application to limit the number of false alarms, this really only applies to normal daily activities. The user always has the option to abort the distress call. Having to abort distress

calls could become a nuisance if daily activities are triggering alarms and causing the user to have to abort frequently. However, a near fall event is not a normal daily activity, and has the potential to frighten the user. When viewed in this light, the application registering a fall may actually instill confidence in the user that if a real fall were to occur, the device would register it and alert their emergency contacts.

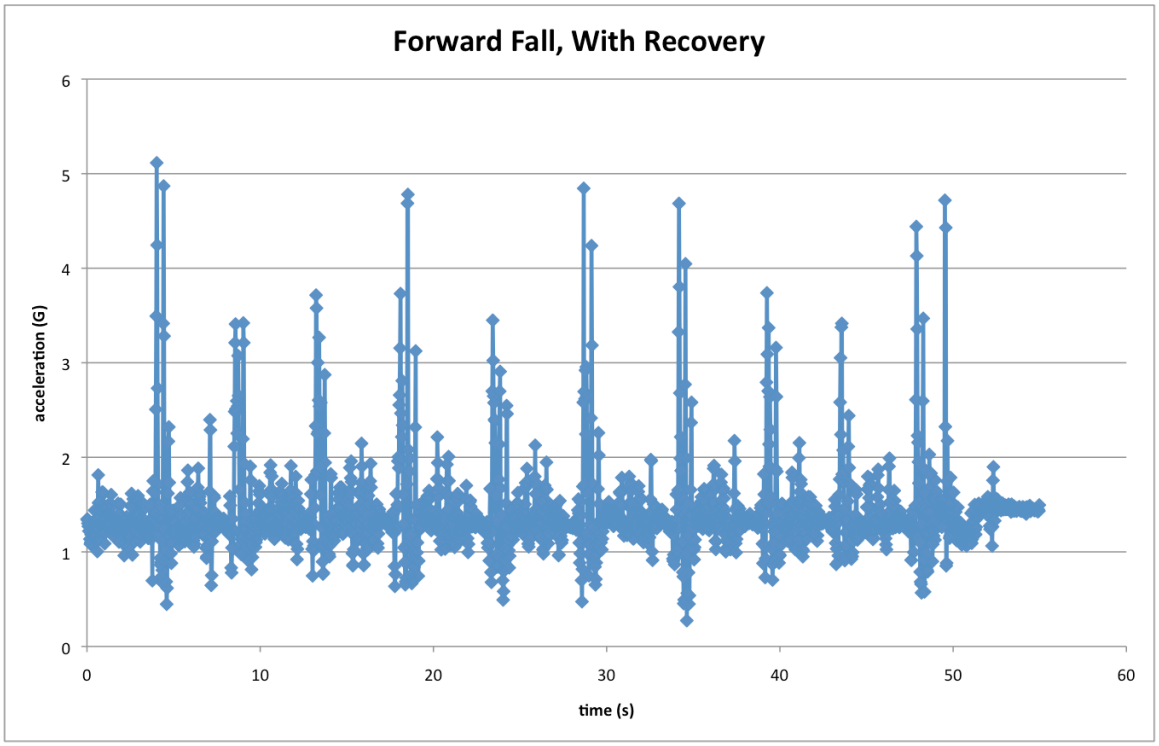

Figure 40 – Graph displaying absolute aggregate acceleration collected during 10 consecutive near falls in the forward direction, with the test subject recovering balance prior to falling.

Figure 40 shows the data collected while performing another non-fall scenario. This scenario was similar to the last one, in that it was another near fall with recovery, but this time in the forward direction. Of the ten fall recovery events, five exceeded the application's fall threshold and five did not. This again

would constitute a failure by the Noury et al criteria. However, as stated previously, a near fall event is a special circumstance, and therefore outside the realm of activities of daily living. In these special circumstances, false alarms are acceptable.

Three more of the scenarios were performed that can be considered special circumstances. There were two more fall recovery events, lateral right and lateral left, where the system registered three false positives and six false positives, respectively. The other scenario was one in which the user leaned back into a wall and slipped down the wall finishing in the sitting position on the floor. In this test the system recorded three out of ten false positives. So, for the special scenario non-fall testing, the system recorded 27 false positives out of 50 events. This resulted in a specificity of 46%, shown below.

$$Specificity = \frac{TN}{TN + FP} = \frac{23}{23 + 27} = 46\% \quad (4)$$

These special scenarios only constitute half of the non-fall events laid out by Noury et al, however. The other five scenarios are ones that fall into the realm of activities of daily living, and therefore are scenarios in which false alarms would be considered to be a nuisance. For these tests, a low specificity would be unacceptable.

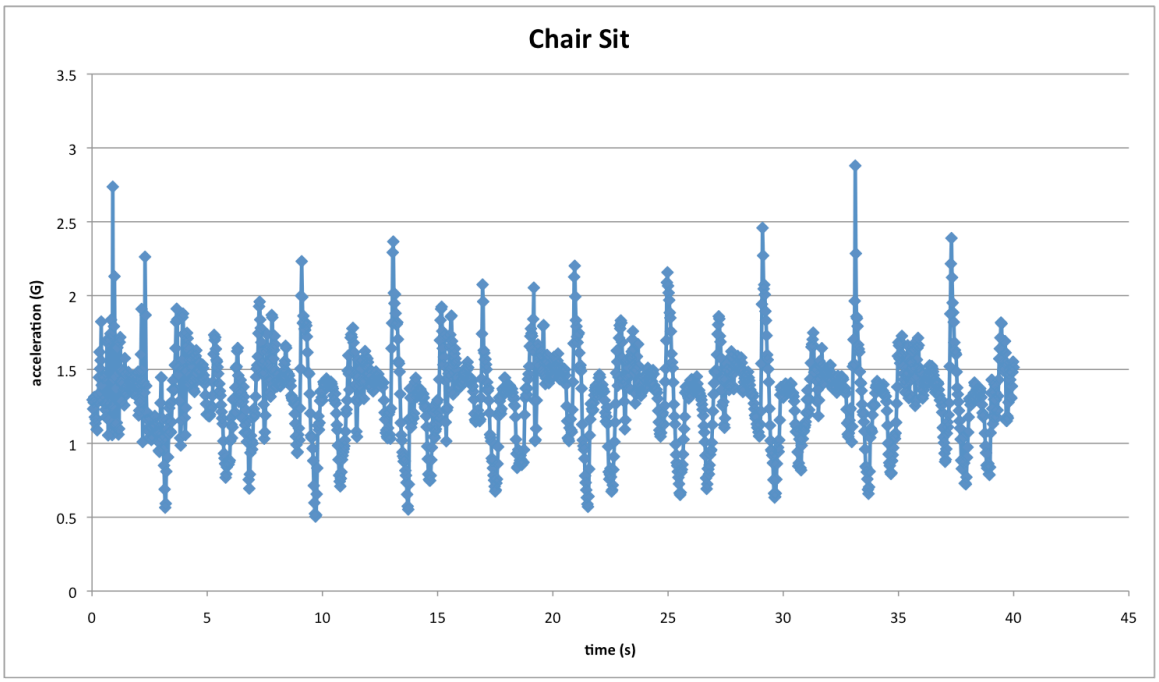

Figure 41 – Graph displaying absolute aggregate acceleration collected during 10 consecutive sit and stand actions performed by a test subject.

Figure 41 above shows the data collected while the test subject sits down and stands up from a chair ten consecutive times. As we can see, none of the accelerations recorded during this test exceed the fall threshold of four g-forces. This result is indicative of the rest of the testing performed for activities of daily living, including laying down and standing up from a bed, walking a few meters, bending down to pick something up off the floor, and coughing/sneezing. For all of these tests, no false positives were recorded, for a result of 100% specificity.

For the sake of completeness, the results from all the non-fall testing were combined to get the specificity shown below.

$$Specificity = \frac{TN}{TN + FP} = \frac{73}{73 + 27} = 73\% \quad (5)$$

A specificity of 73% does not constitute a failure, however, as discussed previously, the true specificity of the system is greater than this when considering only plausible activities of daily living. This sentiment is backed up by the results of the extended false alarm testing. During a 12-hour period of the test subject wearing the device while performing normal daily activities, zero false alarms were recorded.

5.3 Longevity Testing (Battery Life)

The purpose of this test was to verify that the system met the minimum criteria for battery life. In the requirements section, a minimum continuous operation time of 12 hours was set. This minimum requirement was met during the period of extended false alarm testing, discussed in the previous section. During that 12-hour test period, the battery percentage on the iPhone used, went from a fully charged 100% down to 18%. This exceeded the minimum requirement. It should also be noted that the iPhone used for the testing was more than three years old and therefore has a decreased battery life when compared to a new device. Additionally, there are products that would allow the user to further extend the battery life of their device, such as cases containing auxiliary batteries.

6 CONCLUSIONS

This chapter contains a brief discussion on some ways to extend and improve upon the work done for this project, as well as some closing thoughts on what was accomplished in this thesis.

6.1 Future Work

There are several ways that the work from this thesis can be extended and improved upon. While the final application met all of the minimum requirements laid out in this thesis, some ideal functionality could not be incorporated. For instance, the ability for the app to function in the background would be ideal. This would allow the user to operate their smartphone as they normal would, completely unimpeded, while still being afforded the protection of the fall detection application. While iOS does not normally allow for this type of app to run in the background, it may be possible with special permissions granted from Apple.

Beyond adding functionality, there is some significant work that would need to go into this project in order to make it a viable application for customers. First, there would need to be a solution to the monetary issues related to the Twilio service used by the app to programmatically send text messages. This service typically charges a small fee per message delivered. It would need to be decided how this fee would be passed on to the application user. Additionally, in

order to make the application ready for the App Store, a significant amount of work would need to go into more thorough testing and refactoring of the application code [3, 11]. These would be worthwhile activities, as this app has the potential to enhance the quality of life for those living in fear of falling.

6.2 Conclusion

This thesis demonstrates the viability of a fall detection application for the smartphone platform. The testing performed on the final design proves its efficacy as a fall detection solution and shows its potential for protecting and enhancing the lives of individuals at risk of sustaining a life-threatening fall. All of the goals laid out at the beginning of this project were met, and in some cases exceeded. While there is no perfect solution for a complicated problem such as that of falls in the elderly, the solution detailed in this thesis is a practical one with potential to be improved even further.

BIBLIOGRAPHY

1. “911 Wireless Services.” *Federal Communications Commission*. N.p., 2014. Web. 20 Jan. 2015.

2. “Alert-1 Medical Alert Systems.” N.p., n.d. Web. 22 Apr. 2015.

3. Apple, Inc. *App Distribution Guide: About App Distribution*. N.p., 2014. Web.

4. Apple, Inc. *App Programming Guide for iOS*. N.p., 2014. Web.

5. Apple, Inc. *Audio Session Programming Guide*. N.p., 2014. Web.

6. Apple, Inc. *Cocoa Fundamentals Guide*. N.p., 2010. Web.

7. Apple, Inc. *Coding Guidelines for Cocoa*. N.p., 2012. Web.

8. Apple, Inc. *Concepts in Objective- - C Programming*. N.p., 2012. Web.

9. Apple, Inc. *Concurrency Programming Guide*. N.p., 2011. Web.

10. Apple, Inc. *Core Data Tutorial for iOS*. N.p., 2010. Web.

11. Apple, Inc. *Developing for the App Store*. N.p., 2013. Web.

12. Apple, Inc. *Entitlement Key Reference: About Entitlements*. N.p., 2012. Web.

13. Apple, Inc. *Event Handling Guide for iOS*. N.p., 2011. Web.

14. Apple, Inc. *Framework Programming Guide*. N.p., 2013. Web.

15. Apple, Inc. *iOS App Programming Guide*. N.p., 2011. Web.

16. Apple, Inc. *iOS Technology Overview*. N.p., 2010. Web.

17. Apple, Inc. *iTunes Connect Developer Guide 7.4*. N.p., 2012. Web.

18. Apple, Inc. *Key-Value Coding Programming Guide*. N.p., 2012. Web.

19. Apple, Inc. “MotionGraphs.” 2012. Web.

20. Apple, Inc. *Multimedia Programming Guide*. N.p., 2014. Web.

21. Apple, Inc. *The Objective C Programming Language*. N.p., 2008. Web.

22. Apple, Inc. *Object- Oriented Programming with Objective- C*. N.p., 2010. Web.

23. Apple, Inc. *Performance Overview*. N.p., 2013. Web.

24. Apple, Inc. *Programming with Objective C*. N.p., 2012. Web.

25. Apple, Inc. *Property List Programming Guide: About Property Lists*. N.p., 2010. Web.

26. Apple, Inc. *SDK Compatibility Guide*. N.p., 2010. Web.

27. Apple, Inc. *Start Developing iOS Apps Today*. N.p., 2014. Web.

28. Apple, Inc. *Table View Programming Guide*. N.p., 2013. Web.

29. Apple, Inc. *Threading Programming Guide*. N.p., 2010. Web.

30. Apple, Inc. *View Controller Catalog for iOS*. N.p., 2014. Web.

31. Apple, Inc. *View Controller Programming Guide for iOS*. N.p., 2011. Web.

32. Apple, Inc. *View Programming Guide for iOS*. N.p., 2011. Web.

33. Apple, Inc. *Xcode Overview*. N.p., 2008. Web.

34. Apple, Inc. *Your Third iOS App : iCloud*. N.p., 2012. Web.

35. Bagalà, Fabio et al. “Evaluation of Accelerometer-Based Fall Detection Algorithms on Real-World Falls.” *PLoS ONE* 7.5 (2012): 1–9. Web.

36. Blake, a J et al. “Falls by Elderly People at Home: Prevalence and Associated Factors.” *Age and ageing* 17.6 (1988): 365–372. Web.

37. Bourke, a. K., and G. M. Lyons. “A Threshold-Based Fall-Detection Algorithm Using a Bi-Axial Gyroscope Sensor.” *Medical Engineering and Physics* 30.1 (2008): 84–90. Web.

38. Bourke, A.K., J.V. O’Brien, and G.M. Lyons. “Evaluation of a Threshold-Based Tri-Axial Accelerometer Fall Detection Algorithm.” *Gait & Posture* 26.2 (2007): 194–199. Web.

39. Carroll, Cameron, Connor Mosley, and Mike Machado. *Remote Fall Detection: Final Project Report*. San Luis Obispo: N.p., 2011. Print.

40. Chatelier, Pierre. “From C ++ to Objective-C.” (2009): 1–71. Web.

41. Draper, Dr. Richard. “Prevention of Falls in the Elderly.” *Journal of the American Geriatrics Society* (2014): 1–10. Web.

42. “Falls and Older Adults.” *NIH Senior Health*. N.p., 2015. Web. 21 Apr. 2015.

43. “Falls Prevention Facts.” *National Council on Aging*. N.p., 2014. Web. 22 Jan. 2015.

44. “Falls.” *World Health Organization*. N.p., 2012. Web. 20 Jan. 2015.

45. “FCC Adopts Text-to-911 Rules.” *Federal Communications Commission* (2014): 143. Web.

46. Fuller, George F. “Falls in the Elderly.” *American Family Physician* 61.7 (2000): 2159–2168. Web.

47. Graafmans, W C et al. “Falls in the Elderly: A Prospective Study of Risk Factors and Risk Profiles.” *American Journal of Epidemiology* 143.11 (1996): 1129–1136. Web.

48. “How Often Falls Occur.” N.p., 2012. Web. 20 Jan. 2015.

49. Igual, Raul, Carlos Medrano, and Inmaculada Plaza. “Challenges, Issues and Trends in Fall Detection Systems.” *Biomed. Eng. Online* 12.66 (2013): n. pag. Web.

50. “Important Facts about Falls.” *Centers for Disease Control and Prevention*. N.p., 2015. Web. 3 Nov. 2015.

51. Kim, Myung-chul et al. “Developed System of Fall down Estimation Using a Bi-Axial Gyro Sensor.” *Recent Advances in Computational Intelligence, Man-Machine Systems and Cybernetics* (2009): 69–72. Web.

52. Krishnaswamy, Dr. B, and Dr. Gnanasambandam Usha. “Falls in Older People: National/regional Review India.” *Department of Geriatric Medicine, Madras Medical College and Government General Hospital* (2006): 1–19. Web.

53. “Life Alert.” N.p., 2015. Web. 22 Apr. 2015.

54. Li, Qiang et al. “Accurate, Fast Fall Detection Using Gyroscopes and Accelerometer-Derived Posture Information.” (2009): 138–143. Web.

55. Noury, N et al. “Fall Detection - Principles and Methods.” *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2007): 1663–1666. Web.

56. “Mobile Millennials: Over 85% of Generation Y Owns Smartphones.” *Nielsen*. N.p., 2014. Web. 21 Apr. 2015.

57. “Mobile Technology Fact Sheet.” *Pew Research Center Internet, Science and Tech*. N.p., 2015. Web. 22 Apr. 2015.

58. “Philips Lifeline Medical Alert Systems.” N.p., n.d. Web. 22 Apr. 2015.

59. Prudham, D, and J. Grimley Evans. “Factors Associated with Falls in the Elderly: A Community Study.” *Age and Ageing* 10 (1981): 141–146. Web.

60. “Pursuant to the Next Generation 911 Advancement Act of 2012 Legal and Regulatory Framework for Next Generation 911 Services Report to Congress and Recommendations Federal Communications Commission.” *Federal Communications Commission*. N.p., 2012. Web. 20 Jan. 2015.

61. Sampson, Rana. “Misuse and Abuse of 911.” *Center for Problem-Oriented Policing*. N.p., 2002. Web. 21 Jan. 2015.

62. Sartini, M. et al. “The Epidemiology of Domestic Injurious Falls in a Community Dwelling Elderly Population: An Outgrowing Economic Burden.” *European Journal of Public Health* 20.5 (2010): 604–606. Web.

63. “Senior Health & Wellness Blog 10 Shocking Statistics About Elderly Falls.” N.p., 2012. Web. 21 Jan. 2015.

64. Siracuse, Jeffrey J. et al. “Health Care and Socioeconomic Impact of Falls in the Elderly.” *The American Journal of Surgery* 203.3 (2012): 335–338. Web.

65. Sixsmith, Andrew, and Neil Johnson. “A Smart Sensor to Detect the Falls of the Elderly.” *Pervasive Computing* (2004): 1–6. Web.

66. Smith, Aaron. “A Portrait of Smartphone Ownership.” *U.S. Smartphone Use in 2015* (2015): n. pag. Web.

67. Smith, Aaron et al. “U.S. Smartphone Use in 2015.” *The Smartphone Difference* (2015): 60. Web.

68. Sohng, Kyeong-Yae et al. “Risk Factors for Falls among the Community-Dwelling Elderly in Korea.” *Taehan Kanho Hakhoe Chi* 34.8 (2004): 1483–1490. Web.

69. Todd, Chris, and Dawn Skelton. “What Are the Main Risk Factors for Falls amongst Older People and What Are the Most Effective Interventions to Prevent These Falls ?” *World Health* March (2004): 28. Web.

70. “Twilio.” N.p., n.d. Web. 20 Jan. 2015.

71. “What Causes Falls in the Elderly ? How Can I Prevent a Fall ?” *American Academy of Family Physicians* 61.7 (2000): 2173–2174. Web.

Appendix A – AppDelegate Header

```
//
//  AppDelegate.h
//  FallDetection
//
//  Created by Connor Mosley on 11/3/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <CoreMotion/CoreMotion.h>
@import AVFoundation;
@class TCDevice;

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic, readonly) CMMotionManager
    *sharedManager;

@end
```

Appendix B – AppDelegate Source

```
//
//  AppDelegate.m
//  FallDetection
//
//  Created by Connor Mosley on 11/3/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "AppDelegate.h"

@interface AppDelegate ()
{
    CMMotionManager *motionmanager;
}
@end

@implementation AppDelegate

- (CMMotionManager *)sharedManager
{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        motionmanager = [[CMMotionManager alloc] init];
    });
    return motionmanager;
}


- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    NSError *setCategoryError = nil;
    BOOL success = [[AVAudioSession sharedInstance]
                    setCategory: AVAudioSessionCategoryPlayback
                    error: &setCategoryError];
    if (!success) { /* handle the error in setCategoryError */ }

    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    // Sent when the application is about to move from active to
inactive state. This can occur for certain types of temporary
interruptions (such as an incoming phone call or SMS message) or
when the user quits the application and it begins the transition
to the background state.
}
```

```
- (void)applicationDidEnterBackground:(UIApplication
*)application {
    // Use this method to release shared resources, save user
data, invalidate timers, and store enough application state
information to restore your application to its current state in
case it is terminated later.
    // If your application supports background execution, this
method is called instead of applicationWillTerminate: when the
user quits.
}

- (void)applicationWillEnterForeground:(UIApplication
*)application {
    // Called as part of the transition from the background to
the inactive state; here you can undo many of the changes made on
entering the background.
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    // Restart any tasks that were paused (or not yet started)
while the application was inactive. If the application was
previously in the background, optionally refresh the user
interface.
}

- (void)applicationWillTerminate:(UIApplication *)application {
    // Called when the application is about to terminate. Save
data if appropriate. See also applicationDidEnterBackground:.
    [self stopUpdates];
}

- (void)stopUpdates
{
    CMMotionManager *mManager = [(AppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];

    if ([mManager isAccelerometerActive] == YES) {
        [mManager stopAccelerometerUpdates];
    }
}

@end
```

Appendix C – ViewController Header

```
//
//  ViewController.h
//  FallDetection
//
//  Created by Connor Mosley on 11/3/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <CoreMotion/CoreMotion.h>

@interface ViewController : UIViewController

@end
```

Appendix D – ViewController Source

```
//
//  ViewController.m
//  FallDetection
//
//  Created by Connor Mosley on 11/3/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "ViewController.h"
#import "AppDelegate.h"
#import "FallDetectedViewController.h"

static const NSTimeInterval accelerometerMin = 0.025;

@interface ViewController () <TCDeviceDelegate>
//For slide to unlock
@property (weak, nonatomic) IBOutlet UISlider *slideToUnlock;
@property (weak, nonatomic) IBOutlet UIButton *initiateButton;
@property (weak, nonatomic) IBOutlet UILabel *myLabel;  //label
    within slider
@property (weak, nonatomic) IBOutlet UIImageView *Container;
//slide bar
@property (weak, nonatomic) IBOutlet UILabel *inProgressLabel;
//label indicating dall detection in progress
@property (weak, nonatomic) IBOutlet UIImageView *cautionLabel;

//For slide to dial 911
@property (weak, nonatomic) IBOutlet UISlider *slideToDial;
@property (weak, nonatomic) IBOutlet UILabel *EMSLabel;  // call
     911 label within slider
@property (weak, nonatomic) IBOutlet UIImageView *EMSSlideBar;
//container for EMS slider

@property NSTimer *labelFadeTimer;
@property (weak, nonatomic) IBOutlet UIView *backgroundView;


@end

@implementation ViewController

BOOL UNLOCKED = NO; //global variable for slider

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically
       from a nib.
```

```
    _backgroundView.backgroundColor = [UIColor
      groupTableViewBackgroundColor];

    _cautionLabel.alpha = 0.8;

    // initialize custom UISlider (you have to do this in
      viewdidload or applicationdidfinishlaunching.
    UIImage *stetchLeftTrack= [[UIImage
                                imageNamed:@"Nothing.png"]
         stretchableImageWithLeftCapWidth:30.0 topCapHeight:0.0];
    UIImage *stetchRightTrack= [[UIImage
                                imageNamed:@"Nothing.png"]
        stretchableImageWithLeftCapWidth:30.0 topCapHeight:0.0];
    [_slideToUnlock setThumbImage: [UIImage
                                imageNamed:@"SlideToStop.png"]
                                forState:UIControlStateNormal];
    [_slideToUnlock setMinimumTrackImage:stetchLeftTrack
                      forState:UIControlStateNormal];
    [_slideToUnlock setMaximumTrackImage:stetchRightTrack
                             forState:UIControlStateNormal];
    // initialize 911 slider
    UIImage *stetchLeftTrackB= [[UIImage
                                imageNamed:@"Nothing.png"]
         stretchableImageWithLeftCapWidth:30.0 topCapHeight:0.0];
    UIImage *stetchRightTrackB= [[UIImage
                                imageNamed:@"Nothing.png"]
         stretchableImageWithLeftCapWidth:30.0 topCapHeight:0.0];
    [_slideToDial setThumbImage: [UIImage
                                imageNamed:@"SlideToStopB.png"]
                                forState:UIControlStateNormal];
    [_slideToDial setMinimumTrackImage:stetchLeftTrackB
                                forState:UIControlStateNormal];
    [_slideToDial setMaximumTrackImage:stetchRightTrackB
                                forState:UIControlStateNormal];
    _slideToDial.value = 1.0; //slider starts at the right
    _EMSSlideBar.alpha = 0.3; //make slide container somewhat
                                 transparent until screen dims

    //hide slider
    _slideToUnlock.hidden = YES;
    _initiateButton.hidden = NO;
    _Container.hidden = YES;
    _myLabel.hidden = YES;
    _inProgressLabel.hidden = YES;

}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
```

```
}

- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
    [self stopUpdates];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (void)startUpdatesWithSliderValue
{
    NSTimeInterval updateInterval = accelerometerMin;

    [UIApplication sharedApplication].idleTimerDisabled = YES;
    CMMotionManager *mManager = [(AppDelegate *)[[UIApplication
                    sharedApplication] delegate] sharedManager];

    ViewController * __weak weakSelf = self; //weak
                                  version of self (ViewController)
    if ([mManager isAccelerometerAvailable] == YES) {
        [mManager setAccelerometerUpdateInterval:updateInterval];
        [mManager
startAccelerometerUpdatesToQueue:[NSOperationQueue mainQueue]
withHandler:^(CMAccelerometerData *accelerometerData, NSError
*error) {
[weakSelf checkFallX:accelerometerData.acceleration.x
y:accelerometerData.acceleration.y
z:accelerometerData.acceleration.z];
        }];
    }


}

- (IBAction)stopFallDetection {

    if (!UNLOCKED) {

        if (_slideToUnlock.value ==1.0) {  // if user slide far
                                       enough, stop the operation
            // Put here what happens when it is unlocked

            _slideToUnlock.hidden = YES;
            _initiateButton.hidden = NO;
            _Container.hidden = YES;
            _myLabel.hidden = YES;
```

```
            _inProgressLabel.hidden = YES;
            UNLOCKED = YES;

            [UIView animateWithDuration:0.7 delay:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
                _backgroundView.backgroundColor = [UIColor
                                groupTableViewBackgroundColor];
                _cautionLabel.alpha = 0.8;
                _EMSSlideBar.alpha = 0.3;
            }
                                completion:nil];



            [self.labelFadeTimer invalidate];

            [self.tabBarController.tabBar setHidden:NO];

            [self stopUpdates];

        } else {
            // user did not slide far enough, so return back to 0
                position

[UIView animateWithDuration:0.2 delay:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
                [_slideToUnlock setValue:0.0 animated:YES];
                _myLabel.alpha = 1.0;
            }
             completion:nil];



        }
    }
}

- (IBAction)callEMS {
    if (_slideToDial.value ==0.0) {
        //user slid all the way, prompt user with message to call
           911? Then call if user selects yes.
        NSString *str = @"Are you in need of Emergency Services?
                Press Yes to dial 911. Press No to cancel.";
        UIAlertView *EMSalert = [[UIAlertView alloc]
initWithTitle:@"Call 911?" message:str delegate:self
cancelButtonTitle:@"No" otherButtonTitles:@"Yes", nil];
        [EMSalert show];
    }
    else {
        //user did not slide far enough, so return back to 1
```

```
            position
        [UIView animateWithDuration:0.2 delay:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
            [_slideToDial setValue:1.0 animated:YES];
            _EMSLabel.alpha = 1.0;
        }
                         completion:nil];
    }
}

- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 1) {
        _slideToDial.value = 1.0;
        _EMSLabel.alpha = 1.0;
        NSString *phoneNumber = @"tel://611";
        [[UIApplication sharedApplication] openURL:[NSURL
URLWithString:phoneNumber]];
    }
    else {
        [UIView animateWithDuration:0.2 delay:0.0
options:UIViewAnimationOptionCurveEaseOut animations:^{
            [_slideToDial setValue:1.0 animated:YES];
            _EMSLabel.alpha = 1.0;
        }
          completion:nil];
    }
}


- (void)stopUpdates
{
    CMMotionManager *mManager = [(AppDelegate *)[[UIApplication
sharedApplication] delegate] sharedManager];

    if ([mManager isAccelerometerActive] == YES) {
        [mManager stopAccelerometerUpdates];
        }
}

- (IBAction)fadeLabel:(id)sender {
    _myLabel.alpha = 1.0 - _slideToUnlock.value;
}

- (IBAction)fadeEMSLabel:(id)sender {
    _EMSLabel.alpha = _slideToDial.value;
}

-(void)checkFallX:(double)x y:(double)y z:(double)z
```

```
{
if ((ABS(x) + ABS(y) + ABS(z)) > 4) {
        [self stopUpdates];
        [self performSegueWithIdentifier:@"mySegue" sender:self];
    }
}

- (IBAction)startAccelerometer:(id)sender {
    _slideToUnlock.hidden = NO;
    _initiateButton.hidden = YES;
    _Container.hidden = NO;
    _myLabel.hidden = NO;
    _myLabel.alpha = 1.0;
    _inProgressLabel.hidden = NO;
    _inProgressLabel.alpha = 0.0;
self.labelFadeTimer = [NSTimer scheduledTimerWithTimeInterval:3
target:self selector:@selector(timerFireMethod:) userInfo:nil
repeats:YES];
    [UIView animateWithDuration:2.0 delay:0.0
options:UIViewAnimationOptionCurveEaseIn animations:^{
        _backgroundView.backgroundColor = [UIColor blackColor];
        _cautionLabel.alpha = 0.2;
        _EMSSlideBar.alpha = 1.0;
    }
                    completion:nil];


    UNLOCKED = NO;
    _slideToUnlock.value = 0.0;
    NSString *str = @"Please place phone in pocket without
                      pressing Home or Lock button";
    UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Fall Detection Initiated" message:str
delegate:self cancelButtonTitle:nil otherButtonTitles:@"OK",
nil];
    [alert show];
    [self.tabBarController.tabBar setHidden:YES];
    [self startUpdatesWithSliderValue];
}


- (void)timerFireMethod:(NSTimer *)timer
{
    // Fade in the label right away
    [UIView animateWithDuration:1.0
                          delay: 0.0
                        options:
UIViewAnimationOptionCurveEaseOut
                     animations:^{
                         _inProgressLabel.alpha = 1.0;
```

```
                    }
                    completion:^(BOOL finished){
                        // Wait one second and then fade out the
                           label
                        [UIView animateWithDuration:1.0
                                              delay: 0.5

   options:UIViewAnimationOptionCurveEaseIn
animations:^{
              _inProgressLabel.alpha = 0.0;
            }
             completion:nil];}];
}


-(IBAction)unwindToMainView:(UIStoryboardSegue *)segue
{
    [self startUpdatesWithSliderValue];
}

@end
```

Appendix E – FallDetectedViewController Header

```
//
//  FallDetectedViewController.h
//  FallDetection
//
//  Created by Connor Mosley on 11/6/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "TwilioClient.h"
@import MessageUI;
@import AudioToolbox;
@import AVFoundation;

@interface FallDetectedViewController : UIViewController
                <MFMessageComposeViewControllerDelegate>

@property TCDevice* myPhone;
@property TCConnection* myConnection;

@end
```

Appendix F – FallDetectedViewController Source

```
//
//  FallDetectedViewController.m
//  FallDetection
//
//  Created by Connor Mosley on 11/6/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "FallDetectedViewController.h"
#import "ViewController.h"
#import "MyManager.h"
#import "Contact.h"

@interface FallDetectedViewController ()<TCDeviceDelegate>{
    AVAudioPlayer *_audioPlayer;
}

@property NSTimer *distressCallTimer; //initial timer allowing
                                        for user to abort
@property NSTimer *makeCallTimer;     //timer allowing for twilio
                                    texts to send before call is made
@property (weak, nonatomic) IBOutlet UILabel *timerLabel;

@end

@implementation FallDetectedViewController

int timeLeft;

- (void)viewDidLoad {
    [super viewDidLoad];
    //create operation and queue for Twilio initialization
    NSInvocationOperation *twilioInitOp = [[NSInvocationOperation
                                        alloc] initWithTarget:self
                        selector:@selector(initTwilio) object:nil];
    NSOperationQueue *aQueue = [[NSOperationQueue alloc] init];
    [aQueue addOperation:twilioInitOp];

    // Construct URL to sound file
    NSString *path = [NSString
                stringWithFormat:@"%@/fallDetectedSound.aif",
                [[NSBundle mainBundle] resourcePath]];
    NSURL *soundUrl = [NSURL fileURLWithPath:path];

    // Create audio player object and initialize with URL to
        sound
    _audioPlayer = [[AVAudioPlayer alloc]
                        initWithContentsOfURL:soundUrl error:nil];
```

```
    // Set volume (relative to overall system volume, which only
       user can set)
    _audioPlayer.volume = 1.0;

    // Play sound
    [_audioPlayer play];

    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
//make phone vibrate
    timeLeft = 30;
    self.timerLabel.text = [NSString stringWithFormat:@"%d",
                            timeLeft];
    // Do any additional setup after loading the view.
    self.distressCallTimer = [NSTimer
scheduledTimerWithTimeInterval:1 target:self
                     selector:@selector(timerFireMethod:)
                     userInfo:nil repeats:YES];

}

- (void)initTwilio {
    //serve Twilio token and initialize TCDevice
#if TARGET_IPHONE_SIMULATOR
    NSString *name = @"tommy";
#else
    NSString *name = @"jenny";
#endif

//#warning replace this URL with your own server
    //check out https://github.com/twilio/mobile-quickstart to
      get a server up quickly
    NSString *urlString = [NSString
                           stringWithFormat:@"https://arcane-
                           forest-
                           4376.herokuapp.com/token?client=%@",
                           name];
    NSURL *url = [NSURL URLWithString:urlString];
    NSError *error = nil;
    NSString *token = [NSString stringWithContentsOfURL:url
                     encoding:NSUTF8StringEncoding
                     error:&error];
    if (token == nil) {
        NSLog(@"Error retrieving token: %@", [error
         localizedDescription]);
    } else {
        self.myPhone = [[TCDevice alloc]
                        initWithCapabilityToken:token
                        delegate:self];
    }
```

```
}

- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}


#pragma mark - Navigation

// In a storyboard-based application, you will often want to do a
   little preparation before navigation
- (void)prepareForSegue:(UIStoryboardSegue *)segue
sender:(id)sender {
    // Pass the selected object to the new view controller.
    // End Twilio Connection
    [self.myConnection disconnect];
    // Stop sound
    [_audioPlayer stop];
    [self.distressCallTimer invalidate];
    [self.makeCallTimer invalidate];
}


- (void)timerFireMethod:(NSTimer *)timer
{
    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
//make phone vibrate

    if (timeLeft > 0) {
        //replay audio alert at 20 and 10 seconds left
        if (timeLeft == 20) {
            [_audioPlayer play];
        }
        else if (timeLeft == 10) {
            [_audioPlayer play];
        }

        timeLeft--;
        self.timerLabel.text = [NSString stringWithFormat:@"%d",
                                timeLeft];
    } else {
        [self.distressCallTimer invalidate];    //invalidate
           timer, otherwise timerFireMethod will continue to
```

```
            execute

        [self sendTexts];
    }
}

- (void)sendTexts {

    // for access to shared manager
    MyManager *sharedManager = [MyManager sharedManager];

    NSMutableArray *phoneNumbers;
    NSMutableArray *names;
    //get object from array
    phoneNumbers = [NSMutableArray
                    arrayWithCapacity:[sharedManager.gContactItems
                    count]];
    names = [NSMutableArray
             arrayWithCapacity:[sharedManager.gContactItems
             count]];
    for (int i=0; i<[sharedManager.gContactItems count]; ++i) {
        Contact *tempContact;
        tempContact = [sharedManager.gContactItems
                       objectAtIndex:i];
        [phoneNumbers addObject:tempContact.contactPhoneNumber];
        [names addObject:tempContact.contactName];
    }

    NSMutableDictionary *params = [[NSMutableDictionary alloc]
                                   init];
    // Set parameter for user's own name
    [params setValue:sharedManager.userName forKey:@"name"];

    switch ([phoneNumbers count]) {
        case 5:
            [params setValue:[phoneNumbers objectAtIndex:0]
                                   forKey:@"To1"];
            [params setValue:[phoneNumbers objectAtIndex:1]
                                   forKey:@"To2"];
            [params setValue:[phoneNumbers objectAtIndex:2]
                                   forKey:@"To3"];
            [params setValue:[phoneNumbers objectAtIndex:3]
                                   forKey:@"To4"];
            [params setValue:[phoneNumbers objectAtIndex:4]
                                   forKey:@"To5"];
            break;

        case 4:
            [params setValue:[phoneNumbers objectAtIndex:0]
                                   forKey:@"To1"];
```

```
            [params setValue:[phoneNumbers objectAtIndex:1]
                                forKey:@"To2"];
            [params setValue:[phoneNumbers objectAtIndex:2]
                                forKey:@"To3"];
            [params setValue:[phoneNumbers objectAtIndex:3]
                                forKey:@"To4"];
            break;

        case 3:
            [params setValue:[phoneNumbers objectAtIndex:0]
                                forKey:@"To1"];
            [params setValue:[phoneNumbers objectAtIndex:1]
                                forKey:@"To2"];
            [params setValue:[phoneNumbers objectAtIndex:2]
                                forKey:@"To3"];
            break;

        case 2:
            [params setValue:[phoneNumbers objectAtIndex:0]
                                forKey:@"To1"];
            [params setValue:[phoneNumbers objectAtIndex:1]
                                forKey:@"To2"];
            break;

        case 1:
            [params setValue:[phoneNumbers objectAtIndex:0]
                                forKey:@"To1"];
            break;


        default:
            break;
    }

    self.myPhone.outgoingSoundEnabled = NO;
    self.myPhone.incomingSoundEnabled = NO;
    self.myPhone.disconnectSoundEnabled = NO;
    self.myConnection = [self.myPhone connect:params
                                delegate:nil];
    self.myConnection.muted = YES;

    //make call after estimated time to send texts
    self.makeCallTimer = [NSTimer
                          scheduledTimerWithTimeInterval:12
                          target:self
                          selector:@selector(makeCall)
                          userInfo:nil repeats:NO];
}

- (void)makeCall {
```

```
    // for access to shared manager
    MyManager *sharedManager = [MyManager sharedManager];
    Contact *tempContact;
    tempContact = [sharedManager.gContactItems objectAtIndex:0];
NSString *numberToCall = [tempContact.contactPhoneNumber
                          stringByReplacingOccurrencesOfString:@"
                          " withString:@""];
    NSString *phoneNumber = [NSString
                             stringWithFormat:@"tel://%@",
                             numberToCall];
    [[UIApplication sharedApplication] openURL:[NSURL
                                URLWithString:phoneNumber]];
}


@end
```

Appendix G – EmergencyContacts Header

```
//
//  EmergencyContacts.h
//  FallDetection
//
//  Created by Connor Mosley on 12/21/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <UIKit/UIKit.h>
@import AddressBook;
@import AddressBookUI;

@interface EmergencyContacts : UITableViewController
                    <ABPeoplePickerNavigationControllerDelegate>

- (IBAction)showPicker:(id)sender;

@end
```

Appendix H – EmergencyContacts Source

```
//
//  EmergencyContacts.m
//  FallDetection
//
//  Created by Connor Mosley on 12/21/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "EmergencyContacts.h"
#import "Contact.h"
#import "AddContact.h"
#import "MyManager.h"

@interface EmergencyContacts ()

@property NSMutableArray *contactItems;

//for use in getting objects from array of contacts in
updatePlist method
@property NSMutableArray *phoneNumbers;
@property NSMutableArray *names;


@end

@implementation EmergencyContacts

- (IBAction)showPicker:(id)sender {
    ABPeoplePickerNavigationController *picker =
                [[ABPeoplePickerNavigationController alloc] init];
                 picker.peoplePickerDelegate = self;

    [self presentViewController:picker animated:YES
                      completion:nil];
}

- (void)peoplePickerNavigationControllerDidCancel:
          (ABPeoplePickerNavigationController *)peoplePicker
{
    [self dismissViewControllerAnimated:YES completion:nil];
}


-
(void)peoplePickerNavigationController:(ABPeoplePickerNavigationC
                                        ontroller *)peoplePicker
                       didSelectPerson:(ABRecordRef)person
                                 property:(ABPropertyID)property
```

```
    identifier:(ABMultiValueIdentifier)identifier {
                NSString* mobile;
    if (property == kABPersonPhoneProperty) {
        ABMultiValueRef multiPhones = ABRecordCopyValue(person,
                                              kABPersonPhoneProperty);
        for(CFIndex i = 0; i < ABMultiValueGetCount(multiPhones);
i++) {
            if(identifier == ABMultiValueGetIdentifierAtIndex
(multiPhones, i)) {
                CFStringRef phoneNumberRef =
                 ABMultiValueCopyValueAtIndex(multiPhones, i);
                NSString *phoneNumber = (__bridge NSString *)
                                         phoneNumberRef;
                CFRelease(phoneNumberRef);
                mobile = [NSString stringWithFormat:@"%@",
                           phoneNumber];

            }
        }
    }

    [self displayPerson:person number:mobile];
    [self dismissViewControllerAnimated:YES completion:nil];
}


- (void)displayPerson:(ABRecordRef)person number:(NSString
*)number
{
    NSString* name = (__bridge_transfer
                      NSString*)ABRecordCopyValue(person,
                      kABPersonFirstNameProperty);

    Contact *contact = [[Contact alloc] init];
    contact.contactName = name;
    contact.contactPhoneNumber = number;

    [self.contactItems addObject:contact];
    [self.tableView reloadData];
    //add contact to shared manager
    MyManager *sharedManager = [MyManager sharedManager];
    [sharedManager.gContactItems addObject:contact];
    [self updatePlist];

}


- (void)viewDidLoad {
    [super viewDidLoad];
    self.contactItems = [[NSMutableArray alloc] init];
```

```
    [self loadInitialData];

    // Uncomment the following line to preserve selection between
presentations.
    // self.clearsSelectionOnViewWillAppear = NO;

    // Uncomment the following line to display an Edit button in
the navigation bar for this view controller.
    // self.navigationItem.rightBarButtonItem =
self.editButtonItem;
}

- (void)loadInitialData {
    // for access to shared manager
    MyManager *sharedManager = [MyManager sharedManager];

    self.contactItems = [NSMutableArray
                         arrayWithArray:sharedManager.gContactItems];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 2;
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    if (section == 0) {
        return 1;
    }
    else {
    return 4;
    }
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if (section == 0)
        return @"Primary Contact";
    else
        return @"Additional Contacts";
```

```
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
           dequeueReusableCellWithIdentifier:@"ListPrototypeCell"
           forIndexPath:indexPath];


    switch (indexPath.section) {
        case 0:
            if (self.contactItems.count > 0) {
                Contact *contact = [self.contactItems
                                    objectAtIndex:0];
                cell.textLabel.text = contact.contactName;
                cell.detailTextLabel.text =
                                    contact.contactPhoneNumber;
                [cell layoutSubviews];
                return cell;
            }
            else {
                cell.textLabel.text = @"Primary Contact";
                cell.detailTextLabel.text = @"";
                [cell layoutSubviews];
                return cell;
            }
            break;

        case 1:
            if (self.contactItems.count > (indexPath.row + 1)) {
                Contact *contact = [self.contactItems
                           objectAtIndex:(indexPath.row + 1)];
                cell.textLabel.text = contact.contactName;
                cell.detailTextLabel.text =
                           contact.contactPhoneNumber;
                [cell layoutSubviews];
                return cell;
            }
            else {
                cell.textLabel.text = @"Additional Contact";
                cell.detailTextLabel.text = @"";
                [cell layoutSubviews];
                return cell;
            }
            break;
        default:
            return cell;
            break;
    }
}
```

```
// Override to support editing the table view.
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        //check if row to delete is blank row already (outside
bounds of data array
        if ((indexPath.section + indexPath.row) >=
             self.contactItems.count) {
            [self.tableView reloadData];
            return;
        }
        // Delete the row from the data source
        [self.contactItems removeObjectAtIndex:(indexPath.row +
                                                indexPath.section)];
        //delete the row from the shared singleton data manager
        MyManager *sharedManager = [MyManager sharedManager];
        [sharedManager.gContactItems
     removeObjectAtIndex:(indexPath.row + indexPath.section)];
        //delete row from table view
//if deletion occurrs for primary contact and it's the only
contact, then it fades to place holder contact
                [tableView reloadRowsAtIndexPaths:@[indexPath]
                withRowAnimation:UITableViewRowAnimationFade];
        //must reload table data unless we're erasing from the
          end
        if ((indexPath.section + indexPath.row) <
                self.contactItems.count) {
                [self.tableView reloadData];
        }
//update Plist
        [self updatePlist];
    } else if (editingStyle == UITableViewCellEditingStyleInsert)
{
        // Create a new instance of the appropriate class, insert
it into the array, and add a new row to the table view
    }
}

- (void)updatePlist {
    // for access to shared manager
    MyManager *sharedManager = [MyManager sharedManager];
    //get object from array
    self.phoneNumbers = [NSMutableArray
      arrayWithCapacity:[sharedManager.gContactItems count]];
    self.names = [NSMutableArray
      arrayWithCapacity:[sharedManager.gContactItems count]];
    NSString *userName = sharedManager.userName;
```

```
    for (int i=0; i<[sharedManager.gContactItems count]; ++i) {
        Contact *tempContact;
        tempContact = [sharedManager.gContactItems
                       objectAtIndex:i];
        [self.phoneNumbers
         addObject:tempContact.contactPhoneNumber];
        [self.names addObject:tempContact.contactName];
    }

    //write out property list code
    NSError  *error;
    NSString *rootPath =
[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
 NSUserDomainMask, YES) objectAtIndex:0];
    NSString *plistPath = [rootPath
                   stringByAppendingPathComponent:@"Data.plist"];
    NSDictionary *plistDict = [NSDictionary
      dictionaryWithObjects:[NSArray arrayWithObjects:
           self.names, self.phoneNumbers, userName, nil]

forKeys:[NSArray arrayWithObjects: @"Names", @"Phones", @"User",
                                    nil]];
NSData *plistData = [NSPropertyListSerialization
                     dataWithPropertyList:plistDict
                format:NSPropertyListXMLFormat_v1_0
                options:0
                error:&error];
    if(plistData) {
        [plistData writeToFile:plistPath atomically:YES];
    } else {
        NSLog(error);
    }
}

@end
```

Appendix I – InformationViewController Header

```
//
//  InformationViewController.h
//  FallDetection
//
//  Created by Connor Mosley on 2/21/15.
//
//

#import <UIKit/UIKit.h>

@interface InformationViewController : UIViewController
                                       <UITextFieldDelegate>

@end
```

Appendix J – InformationViewController Source

```
//
//  InformationViewController.m
//  FallDetection
//
//  Created by Connor Mosley on 2/21/15.
//
//

#import "InformationViewController.h"
#import "MyManager.h"
#import "Contact.h"

@interface InformationViewController ()
@property (weak, nonatomic) IBOutlet UITextField *userName;

@end

@implementation InformationViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view.
    _userName.delegate = self;
    //load user name if it already exists
    MyManager *sharedManager = [MyManager sharedManager];
    if (![sharedManager.userName  isEqual: @""]) {
        self.userName.text = sharedManager.userName;
    }

}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];

}

- (void)updatePlist {
    // for access to shared manager
    MyManager *sharedManager = [MyManager sharedManager];
    //get object from array
    NSMutableArray *phoneNumbers = [NSMutableArray
        arrayWithCapacity:[sharedManager.gContactItems count]];
```

```
    NSMutableArray *names = [NSMutableArray
        arrayWithCapacity:[sharedManager.gContactItems count]];
    for (int i=0; i<[sharedManager.gContactItems count]; ++i) {
        Contact *tempContact;
        tempContact = [sharedManager.gContactItems
                       objectAtIndex:i];
        [phoneNumbers addObject:tempContact.contactPhoneNumber];
        [names addObject:tempContact.contactName];
    }

    //write out property list code
    NSError  *error;
    NSString *rootPath =
[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES) objectAtIndex:0];
    NSString *plistPath = [rootPath
           stringByAppendingPathComponent:@"Data.plist"];
    NSDictionary *plistDict = [NSDictionary
           dictionaryWithObjects:
                          [NSArray arrayWithObjects: names,
                          phoneNumbers, self.userName.text, nil]

forKeys:[NSArray arrayWithObjects: @"Names", @"Phones", @"User",
                                   nil]];
    NSData *plistData = [NSPropertyListSerialization
                         dataWithPropertyList:plistDict
                         format:NSPropertyListXMLFormat_v1_0
                         options:0
                         error:&error];
    if(plistData) {
        [plistData writeToFile:plistPath atomically:YES];
    } else {
        NSLog(error);
    }
}

- (void)textFieldDidEndEditing:(UITextField *)textField
{
        MyManager *sharedManager = [MyManager sharedManager];
        sharedManager.userName = (NSMutableString
                                  *)self.userName.text;
        [self updatePlist];
}
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}

@end
```

Appendix K – Contact Header

```
//
//  Contact.h
//  FallDetection
//
//  Created by Connor Mosley on 12/21/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface Contact : NSObject

@property NSString *contactName;
@property NSString *contactPhoneNumber;

@end
```

Appendix L – Contact Source

```
//
//  Contact.m
//  FallDetection
//
//  Created by Connor Mosley on 12/21/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "Contact.h"

@implementation Contact

@end
```

Appendix M – MyManager Header

```
//
//  MyManager.h
//  FallDetection
//
//  Created by Connor Mosley on 12/22/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface MyManager : NSObject{
    NSMutableArray *gContactItems;
    NSMutableString *userName;
}

@property (nonatomic, retain) NSMutableArray *gContactItems;
@property (nonatomic, retain) NSMutableString *userName;

+ (id)sharedManager;

@end
```

Appendix N – MyManager Source

```
//
//  MyManager.m
//  FallDetection
//
//  Created by Connor Mosley on 12/22/14.
//  Copyright (c) 2014 Connor Mosley. All rights reserved.
//

#import "MyManager.h"
#import "Contact.h"

@interface MyManager ()

@end

@implementation MyManager

@synthesize gContactItems;
@synthesize userName;

#pragma mark Singleton Methods

+ (id)sharedManager {
    static MyManager *sharedMyManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedMyManager = [[self alloc] init];
    });
    return sharedMyManager;
}

- (id)init {
    if (self = [super init]) {
        gContactItems = [[NSMutableArray alloc] init];
        userName = [[NSMutableString alloc] init];
        NSMutableArray *phoneNumbers;
        NSMutableArray *names;

        //read in property list
        NSError *errorDesc = nil;
        NSPropertyListFormat format;
        NSString *plistPath;
        NSString *rootPath =
        [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                             NSUserDomainMask,
                                            YES) objectAtIndex:0];
        plistPath = [rootPath
                     stringByAppendingPathComponent:@"Data.plist"];
```

```
        if (![[NSFileManager defaultManager]
            fileExistsAtPath:plistPath]) {
            plistPath = [[NSBundle mainBundle]
                         pathForResource:@"Data"
                         ofType:@"plist"];
        }
        NSData *plistXML = [[NSFileManager defaultManager]
                            contentsAtPath:plistPath];
        NSDictionary *temp = (NSDictionary
                              *)[NSPropertyListSerialization
                               propertyListWithData:plistXML
            options:NSPropertyListMutableContainersAndLeaves
                                                 format:&format
                                                 error:&errorDesc];
        if (!temp) {
            NSLog(@"Error reading plist: %@, format: %u",
                    errorDesc, format);
        }
        names = [NSMutableArray arrayWithArray:[temp
                objectForKey:@"Names"]];
        phoneNumbers = [NSMutableArray arrayWithArray:[temp
                objectForKey:@"Phones"]];
        self.userName = [temp objectForKey:@"User"];

        //end read in property list

        for (int i=0; i<[phoneNumbers count]; ++i) {
            Contact *item = [[Contact alloc] init];
            item.contactName = [names objectAtIndex:i];
            item.contactPhoneNumber = [phoneNumbers
                                       objectAtIndex:i];
            //add item 1 to sharedManager
            [self.gContactItems addObject:item];
        }
    }
    return self;
}

- (void)dealloc {
    // Should never be called, but just here for clarity really.
}
@end
```

Appendix O – Python Web Script

```
import os
from flask import Flask, request
from twilio.util import TwilioCapability
from twilio.rest import TwilioRestClient
import twilio.twiml

# Account Sid and Auth Token can be found in your account
dashboard
ACCOUNT_SID = 'ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
AUTH_TOKEN = 'YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY'

# TwiML app outgoing connections will use
APP_SID = 'APZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'

CALLER_ID = 'phone number'
CLIENT = 'jenny'

app = Flask(__name__)

@app.route('/token')
def token():
  account_sid = os.environ.get("ACCOUNT_SID", ACCOUNT_SID)
  auth_token = os.environ.get("AUTH_TOKEN", AUTH_TOKEN)
  app_sid = os.environ.get("APP_SID", APP_SID)

  capability = TwilioCapability(account_sid, auth_token)

  # This allows outgoing connections to TwiML application
  if request.values.get('allowOutgoing') != 'false':
     capability.allow_client_outgoing(app_sid)

  # This allows incoming connections to client (if specified)
  client = request.values.get('client')
  if client != None:
    capability.allow_client_incoming(client)

  # This returns a token to use with Twilio based on the account
and capabilities defined above
  return capability.generate()

@app.route('/sms', methods=['GET', 'POST'])
def sms():
  """ This method routes calls from/to client
"""
  """ Rules: 1. From can be either client:name or PSTN number
"""
  """        2. To value specifies target. When call is coming
"""
```

```
    """           from PSTN, To value is ignored and call is
"""
    """           routed to client named CLIENT
"""
    resp = twilio.twiml.Response()
    from_value = request.values.get('From')
    to = request.values.get('To1')
    sender_name = request.values.get('name')
    if not (from_value and to):
        return str(resp.say("Invalid request"))
    from_client = from_value.startswith('client')
    caller_id = os.environ.get("CALLER_ID", CALLER_ID)
    if not from_client:
        # PSTN -> client
        resp.dial(callerId=from_value).client(CLIENT)
    elif to.startswith("client:"):
        # client -> client
        resp.dial(callerId=from_value).client(to[7:])
    else:
        # client -> PSTN
        # resp.dial(to, callerId=caller_id)
        account_sid = os.environ.get("ACCOUNT_SID", ACCOUNT_SID)
        auth_token = os.environ.get("AUTH_TOKEN", AUTH_TOKEN)
        client = TwilioRestClient(account_sid, auth_token)
        message = client.messages.create(to=to, from_="+18057197081",
body="%s has fallen and he can't get up!" %sender_name)
        to = request.values.get('To2')
        if (to):
            message = client.messages.create(to=to,
from_="+18057197081", body="%s has fallen and he can't get up!"
%sender_name)
        to = request.values.get('To3')
        if (to):
            message = client.messages.create(to=to,
from_="+18057197081", body="%s has fallen and he can't get up!"
%sender_name)
        to = request.values.get('To4')
        if (to):
            message = client.messages.create(to=to,
from_="+18057197081", body="%s has fallen and he can't get up!"
%sender_name)
        to = request.values.get('To5')
        if (to):
            message = client.messages.create(to=to,
from_="+18057197081", body="%s has fallen and he can't get up!"
%sender_name)
        return str(resp)

@app.route('/', methods=['GET', 'POST'])
def welcome():
```

```
    resp = twilio.twiml.Response()
    resp.say("Regina is a SUPERSTAR!!!")
    return str(resp)

if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app.run(host='0.0.0.0', port=port, debug=True)
```