# A Forearm Controller and Tactile Display

by

David Sachs

B.A. Physics
Oberlin College (2001)
B.Mus. Piano Performance
Oberlin Conservatory (2001)

Submitted to the Program in Media Arts and Sciences
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of

Master of Science
in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005



Author……
Program in Media Arts and Sciences
School of Architecture and Planning
September 1, 2005

Certified by……
Tod Machover
Professor of Music and Media
Thesis Supervisor

Accepted by……...
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

# A Forearm Controller and Tactile Display

by

David Sachs

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on September 1, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science
in Media Arts and Sciences

## Abstract

This thesis discusses the design and implementation of ARMadillo, a simple virtual environment interface in the form of a small wireless device that is worn on the forearm. Designed to be portable, intuitive, and low cost, the device tracks the orientation of the arm with accelerometers, magnetic field sensors, and gyroscopes, fusing the data with a quaternion based Unscented Kalman Filter. The orientation estimate is mapped to a virtual space that is perceived through a tactile display containing an array of vibrating motors. The controller is driven with an 8051 microcontroller, and includes a BlueTooth module and an extension slot for CompactFlash cards.

The device was designed to be simple and modular, and can support a variety of interesting applications, some of which were implemented and will be discussed. These fall into two main classes. The first is a set of artistic applications, represented by a suite of virtual musical instruments that can be played with arm movements and felt through the tactile display, The second class involves utilitarian applications, including a custom Braille-like system called Arm Braille, and tactile guidance. A wearable Braille display intended to be used for reading navigational signs and text messages was tested on two sight-impaired subjects who were able to recognize Braille characters reliably after 25 minutes of training, and read words by the end of an hour.

Thesis Supervisor: Tod Machover
Title: Professor of Music and Media

**A Forearm Controller and Tactile Display**

by

David Sachs

The following people served as thesis readers:

Thesis Supervisor……...

Tod Machover
Professor of Music and Media
Massachusetts Institute of Technology

Thesis Reader…….

Joseph Paradiso
Associate Professor of Media Arts and Sciences
Sony Career Development Professor
Massachusetts Institute of Technology

Thesis Reader…….

Hugh Herr
Assistant Professor of Media Arts and Sciences
Massachusetts Institute of Technology

# Acknowledgements

To **Tod Machover**, for supporting me and giving me the benefit of the doubt in some of my strange and unexpected research paths, and for his bravery in trying out my electrocutaneous displays.

To my readers, **Joe Paradiso** and **Hugh Herr,** for their time and flexibility.

To **Neil Gershenfeld,** for teaching me how to make stuff.

To my group members**, Adam, Ariane, Diana, Hugo, Marc, Mary, Mike, Rob, and Tristan**, for their knowledge and friendship, and for their support and time in these last few difficult weeks.

To the **Responsive Environments Group** for letting me borrow their soldering iron and learn from their designs.

To all the **people** who got **coffee** with me when my eyes couldn't focus.

To my **parents**, for their faith.

To my **sister**, for her wisdom.

To **Ellen**, for her help, patience, and love, and for her strength and courage in these last weeks. And thank you, **Minhee**, for keeping Ellen happy.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

In contrast to pure virtual reality, in which a user's senses are bombarded with artificial input such that the real world is obliterated, and a new world is synthesized and sensed, mixed reality involves a juxtaposition of an artificial world and the real world, and a participant can interact with both simultaneously. This thesis describes the design and implementation of ARMadillo, a device that allows a user to carry the virtual part of this world on the arm. ARMadillo was designed to disappear psychologically, be worn under the clothing, and potentially be perceived as a part of the arm that can be ignored or used at will. Just as the rest of the skin, the eyes, and the ears are tuned to receive sensory input from the real world, the skin of the forearm would now be tuned to receive input from a false one, which could be influenced with arm movements. This environment could potentially contain any set of virtual objects, media, or controls; they could be utilitarian in nature, or intended for artistic expression.

These two categories, both being of interest, will be expanded in this thesis. The first category, artistic expression, was implemented in the form of a set of virtual musical instruments. These instruments can be felt with the arm wearing the controller, and played by moving that arm. As the device does not restrain movement or cover the hands, these virtual instruments could be played in addition to real ones. This system was designed to harvest good qualities from two very different types of instruments: tangible ones, which can be felt and played virtuosically, but root the performer to a physical object; and intangible ones, which free the performer's body, but are very difficult to master. The virtual instruments described in this thesis are designed to be tangible via the tactile display, without restricting the performer's movement, or being rooted to an external object or location.

The second category of utilitarian applications involves a Braille-based guidance system called Arm Braille, in which the device is used to receive text related to navigation, such as signs that indicate the nature of doorways, hallways, street intersections, and so on. The intention of this project is to remove the Braille signs one usually finds labeling doors, and place them on the arm of the user, thus removing the

necessity of feeling the wall next to each door. Such a system would, in practice, rely on a position sensing external system, which exists in many forms, but was not implemented as part of this thesis. Only the Braille display itself and a wearable tactile compass were implemented, and the Braille display was tested on two sight-impaired subjects, who were each able to read individual characters reliably after about 25 minutes of training, and demonstrated rudimentary word reading in their first hour.

As the controller and its applications draw from a wide variety of fields, the rest of this introduction will attempt to touch on each of these fields in order to bring the reader up to date, and to help build a foundation from which one can understand the design motivation and constraints for the controller and its applications.

The hardware design of Arnadillo consists of two important sub-systems: a motion tracker, and a tactile display, both of which are seen in Figure 1-1. Commonly used technologies for each of these will be described, along with some example applications. As a set of virtual instruments was designed for this controller, this introduction will continue with descriptions of a few prior electronic music systems that were particularly relevant in the design of the forearm controller. This section will also touch on some interesting recent research in haptics as it relates to music. Finally, to prepare the reader for the Braille-based guidance applications, a quick Braille primer will be given, followed by a summary of Braille display technology.

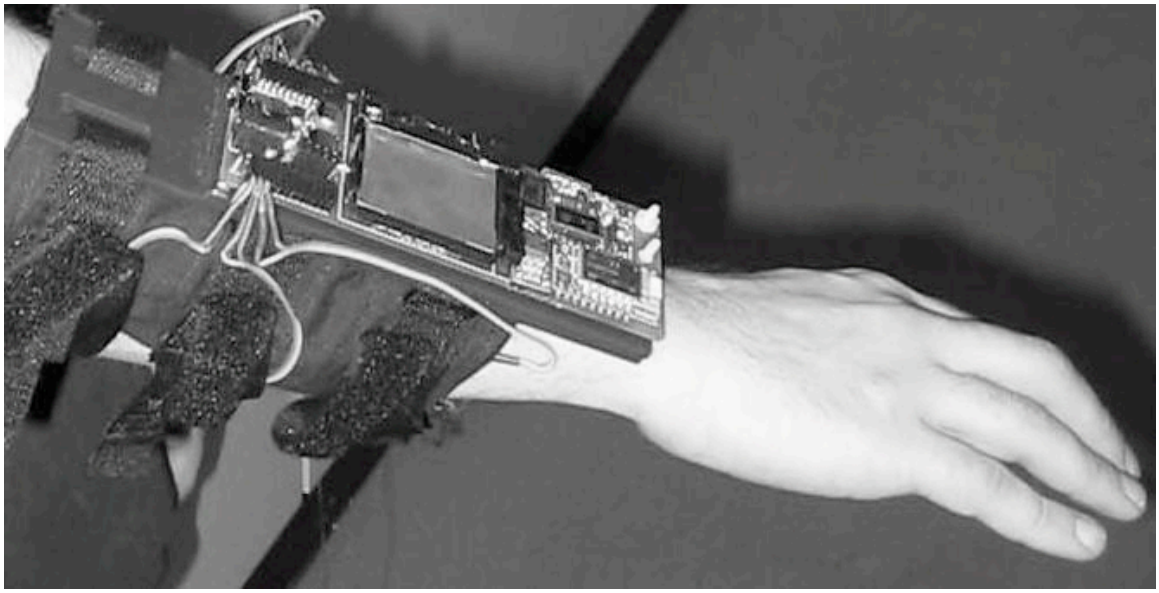


**Figure 1-1: The forearm controller and tactile display**

## 1.1 Motion Tracking Technologies

### 1.1.1 Sensing with a Mechanical Exoskeleton

One way to sense the movement of a human body is to attach a robotic device to it that is free to move and bend in the same manner, and sense the state of the device rather than attempting to directly sense the state of the human body. Using an exoskeleton to track movement can be bulky and awkward, but the reward is one of high update rates and accuracy, and the potential for force feedback systems, which will be described later in this chapter.

Such devices usually rely on measurements of resistance, or optical measurements. In the case of the former, a rotating joint will contain a variable resistor, or potentiometer, such that the resistance changes as the joint bends. In the second case, an optical system is used in which a joint contains a light source and a light detector, with a pattern of notches between them that can be detected. As the joint bends, the pattern of notches changes, and the subsequent pattern of light is altered and can be measured. The

first system tends to be cheaper, but suffers from friction and wear due to a physical wiper that must scrape across a resistive material. The optical system requires no such scraping, and will tend to last longer and involve less friction [Fra04].

A common example of a system that uses potentiometers to track motion is the PHANToM, a desktop robotic arm that can be manipulated, and tracks the position and orientation of its appendage. This device will be referred to again in the section on force feedback [Sen05]. When an exoskeleton is necessary for other reasons, such measurement techniques become even more appealing. For example, at MIT, an ankle orthosis was designed to track foot movement, and actively adjust it to minimize drop-foot, a condition that makes it difficult to raise the foot at the ankle. In this case, as an exoskeleton brace was necessary anyway, the obvious design choice was to measure the ankle joint rotation mechanically with a rotating variable resistor [BH04].

### 1.1.2 Sensing From a Base Station

To sense absolute position, Galilean relativity demands that there be an absolute reference frame from which position is to be compared. In general, a base station must be set up first. This station or array of stations typically transmits ultrasound, infrared light, or radio waves, which are received and measured by the object that is to be tracked. Alternatively, the object can contain a transmitter, and the base station can contain a receiver or array of receivers [Fra04].

Clearly, such a system is easier to implement when the base stations have already been set up. This is the case with GPS, in which the user measures position relative to a set of satellites that are already in place. Unfortunately, GPS delivers poor results indoors and near buildings. Similar types of GPS-like systems for use indoors have been implemented or proposed. One interesting one modifies the existing fluorescent lights within a building to generate an optical identification tag, allowing a receiver to measure its position relative to the lights [Tal05].

It should be noted that the base stations don't have to be far from the tracked object; for use in a wearable system, base stations could be worn on the body. For example, Laetita Sonami created a music controller called the Lady's Glove, which included an ultrasound emitter on the belt and another on the shoe, and a receiver on the hand, allowing the performer's hand position to be measured relative to her body [Cha97].

### 1.1.3 Inertial Sensing

Inertial sensing systems usually include some combination of accelerometers, magnetic field sensors, and gyroscopes, and are commercially available as modules, for example, by the company Intersense [Ise05]. They are sometimes called "sourceless" sensors, because they do not require a base station, as do the previously described systems. When measuring position, the accelerometers can be integrated twice to provide the needed value, but such measurements degrade after a matter of seconds, due to the inherent problem with integrating the noise within a sensor [Fra04]. This can be seen in Fig. 1-2, in which an acceleration of zero and an initial velocity and position of zero should ideally result in a stable position estimate that remains at zero. However, a slight amount of Gaussian noise in the acceleration measurement causes the velocity to wander, and the position to wander even more. In this case, the position estimate is only reasonably close to its true value during the first ten seconds or so.

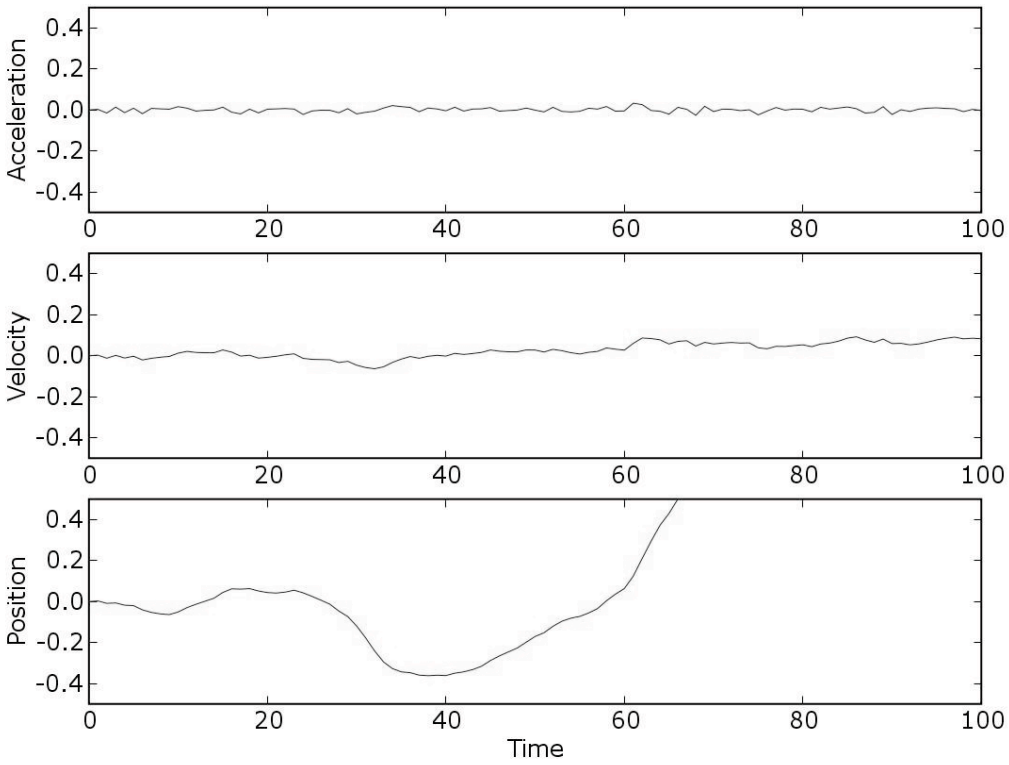



**Figure 1-2: Integrating acceleration and velocity**

Similarly, gyroscopes give a measurement of angular velocity, which can be integrated once to give an angular position, but the same problem of noise integration arises. For this reason, inertial sensors are not usually used to measure position by themselves; when necessary, they are more often combined with a measurement of absolute position, such as that generated by a GPS device. Inertial sensors can then be used to "dead-reckon" between GPS measurements [Fra04].

When measuring orientation, inertial sensors can be combined with magnetic field sensors, and are an attractive choice due to their lack of moving parts or need for a base station. When used as orientation sensors, they are not exactly "sourceless": it is more accurate to say that they measure an attitude with respect to the Earth's reference frame, and that they sense the Earth. Accelerometers are used to measure the acceleration of gravity, giving an orientation with respect to the vertical axis of gravitation, and magnetic

field sensors measure the Earth's magnetic field, giving an orientation with respect to this magnetic field vector. When combined, an accurate orientation can be retrieved, and gyroscopes can be added to sense quick rotational movements.

It is interesting to note that such systems would not work away from the Earth's gravitational and magnetic fields, and would be extremely inaccurate near the Earth's magnetic poles, where the magnetic field vector is almost parallel to the direction of gravity. In addition, the horizontal component of the angular position cannot be said to correspond exactly to magnetic north when the device is used indoors, due to the magnetic field generated by a typical building. or by other local ferrous material [RLV03]. However, they can still be used to sense an absolute orientation with respect to an unknown reference frame, if the literal direction of north is not needed.

ARMadillo uses inertial sensors, the design for which will be described in the next chapter. The design drew from that of a previous project known as the Stack, a modular sensing architecture that contained an inertial measurement unit [Ben98], [BP05].

## 1.2 Tactile Displays

### 1.2.1 Electrocutaneous Displays

In the quest for a truly complete tactile display that can deliver any sensation the skin can receive, the most appealing solution is one in which the display communicates directly to the nerves in the skin. Delivering a variety of effects such as pressure, vibration, or heat, would ideally involve producing electrical stimulations that send those messages to the brain. No moving parts or expensive polymers should be necessary; all could be accomplished with electrodes and software. In reality, such a display has not yet been created, but some promising research should be mentioned.

Some interesting analysis has been done of the way external electrical stimuli are interpreted by the human brain. Research at the University of Tokyo has shown that an electrical stimulation can be broken down into tactile "primary colors", in which certain

different types of nerve endings are targeted separately. For example, to generate a sense of pressure, the Merkel cells would be targeted; to generate vibration, the Meissner or Pacinian corpuscles would be targeted. These nerve endings would be activated selectively by using different combinations of anodic or cathodic current within an array of electrodes [Kaj01], [Kaj99].

Many practical problems need to be considered in order to safely implement an electrocutaneous display. Pushing current on the order of 5mA through highly resistive skin can require high voltages and might generate painful resistive heating if the skin is too dry. This problem can be ameliorated with conductive gel, but this makes the display potentially inconvenient and messy. Preventing these sudden painful sensations requires minimizing the energy dissipation due to heat in the electrical waveform [KKT02].

Some work has been done at the University of Wisconsin that demonstrates the ability of people to explore and identify geometric patterns conveyed in arrays of electrodes. Interestingly, in this experiment, subjects with "excessively high electrode-skin resistance" were excused. This serves as a reminder that the display technology at the time of this experiment was not sufficient to comfortably be used by anyone; the resistive heating effect can be quite painful if the skin impedance is too high [KTB97].

Despite these potential pitfalls and risks, there have been complete displays created using electrical stimulation. An interesting one is "SmartTouch", created at the University of Tokyo. An electrocutaneous display consisting of 16 electrodes was combined with a pressure sensor for controlling the intensity of a sensation through feedback, and optical sensors for receiving information. The device was worn on a fingertip and enabled the wearer to feel a figure that had been drawn on paper, by converting the output of the optical sensor to an electrical stimulation [Kaj03], [Kaj04].

While electrocutaneous displays are fascinating, they were abandoned during the course of this research due to problems with safety and pain control.

### 1.2.2 Thermal Displays

A relative newcomer to the tactile display world, thermal displays could provide an interesting addition to the more common vibration and force displays. In the virtual reality field, adding a perceived temperature to a virtual object could make an environment more realistic. For displays designed to convey abstract information, even a single thermoelectric device and temperature sensor could add an extra dimension to a multimodal tactile display; an array of heating elements could potentially generate extremely interesting sensations.

A typical application of a thermal display would be the simulating of thermal properties of a material in order to add realism to a virtual object. For example, steel and nylon have very different thermal conductivities and heat capacities, and such properties could be simulated in a display. Research at MIT has shown that subjects could distinguish between such materials when their heat capacities differed by about a factor of four [JB03]. A display could simulate such a heat capacity by generating a transient thermal sensation upon contact with a virtual object.

One of the problems in generating meaningful thermal sensations is the unintuitive way in which heat is interpreted by the brain. For example, for low dosages, increasing the surface area that has been heated, while decreasing the intensity of the heat, will not change the perceived sensation. This strange sensory mapping that sums up the total heat felt, with little attention paid to the locality of the heat, serves the important function of helping the body regulate temperature. Localized heat sensations aren't apparent until the heat becomes intense enough that the body is more concerned with protecting against burning than with temperature regulation [JB02].

### 1.2.3 Force Feedback Displays

To truly simulate the presence of a virtual object, force feedback is required. It is not enough to generate sensations on the skin, such as pressure, vibration, or heat; a physical object would prevent a limb from moving through it, and so should a virtual object. For a haptic display to be truly convincing, it must be able to restrict the motion of the user. Such a display is currently too unwieldy to satisfy the constraints set forward by this thesis, as it would consist of a complex robotic system with powerful motors. However, the applications of force feedback are among the most interesting and useful of the tactile displays, and will therefore be discussed here.

Probably the most commonly used haptic feedback device is the PHANToM, by SensAble. This device is commercially available, and is a pen-like appendage attached to a small, reactive robot arm with six degrees of freedom [Sen05]. By measuring the movement of the arm and selectively providing resistance via its actuators, the PHANToM can create the illusion of a movement by the pen across a complex surface, or through a medium.

A second, commonly used force feedback device is a glove called CyberGrasp, by the company Immersion. This glove has a small exoskeleton attached to its back, which contains actuators that can apply a force of up to about 12N, perpendicular to the fingertip [Imm05]. Using feedback with these actuators, this glove can simulate the presence of an object in the hand by restraining the fingers from closing all the way.

Some interesting and useful applications of these two force feedback devices include a sound design controller for a granular synthesis system [BA02], training and assessing surgeons as they make virtual sutures [MBA01], aiding in telemanipulation of objects by transmitting haptic information from a robot arm to a force feedback display [Tur98], scientific visualization of vector fields [Dur98], and stroke therapy and rehabilitation [Riz05].

Although force feedback was not used in this thesis, there is great potential for creating virtual instruments with such devices. The final display type that was used in this research was vibrotactile, and although it was capable of generating sensations and mapping them to points in a virtual environment, it was not capable of restraining the motion of a user, and therefore could be said to simulate complex virtual media through which a user's arm could move, but not solid virtual objects capable of stopping the arm.

### 1.2.4 Vibrotactile Displays

Vibrotactile displays are low cost, easy to implement, and surprisingly effective. Cheap vibrating motors, such as those typically used in cellphones, can be made by simply adding an asymmetrical weight to an ordinary motor. Similarly, vibrating voice coils can be made by adding weight to a speaker voice coil, so that an electrical impulse causes the membrane to deliver momentum to the actuator, rather than simply pushing the air and creating sound. A major difference in driving these two types of actuators is that voice coils can be driven with any waveform and at any frequency, giving the system more control over the type of vibration, whereas vibrating motors must be driven with pulse-width modulation, giving the system some control over the vibration, but certainly not in any linear sense. Vibrating motors, on the other hand, tend to be more powerful for generating transient jolting effects, and cheaper when bought off the shelf. In addition, a carefully crafted PWM system can counteract the inherent awkwardness of controlling a spinning object.

An interesting project that allows users to develop their own tactile language was called ComTouch. In this project, two cell phones were outfitted with vibrating actuators and force sensitive resistors, such that one user could send a vibration to another user. This allowed a certain amount of flexibility and expressiveness. Different users developed different systems for encoding agreement, emphasis, or signaling turn-taking in the course of a conversation [Cha02].

A recent artistic application involving a full body suit of vibrating motors intended to add a tactile component to music was designed by Eric Gunther, and called

SkinScape [Gun01]. As the lower frequencies of music ordinarily have a subtle tactile component, controlling a vibratory sensation along with sound can be aesthetically quite effective. Similar, Skinscape-inspired applications of passive tactile art implemented on a smaller, denser display will be discussed later in this thesis.

Wearable tactile displays offer interesting possibilities for conveying useful information privately [Lin04], [GOS01], [TY04]. Many of these are intended specifically as navigation aids, such as ActiveBelt, which contains a GPS module, a magnetic field sensor, and a belt containing a set of vibrating actuators. To navigate, one must simply turn until the actuator at the belt buckle is vibrating, and walk forward [TY04].

Finally, vibrotactile displays have been used in real-time to provided feedback when controlling a system. While vibration is not usually as convincing as force feedback for simulating an object, the mind can adapt to any lack of realism, and the feedback can therefore still be useful. In one project, vibration was used to aid in a teleoperation involving a peg insertion. Although force feedback might have been more convincing in this case, vibration is certainly easier to implement, cheaper, and was shown to reduce peak forces during the insertion [Deb04].

## 1.3 Some Relevant Music Controllers

Although the literature on music controllers is vast and fascinating, the following review of this field will focus on those instruments that are hands-free, as that constraint is one of the most difficult, and had an unforgiving role in shaping the conceptual design of the forearm controller. It should be emphasized that the term "hands-free" is intended to mean that the user has no object or fabric in contact with the hands at any time, and is free to grasp objects with them or move the fingers without consequence. Clearly, however, if arm movements are what controls the device, then the hands are not free to accomplish any task while the controller is being used. The decision to make the forearm controller hands-free was partially a subjective one, rooted in the author's own revulsion to glove-like controllers and desire to be free of objects and spaces that constrain movement.

Aside from the convenience of being able to move naturally and ignore the existence of the controller until the moment at which it is to be triggered, what is the advantage of leaving the hands free, but not the forearm? In a sense, the forearm controller does not necessarily use up any of the body's commonly accessed resources. Movements are not restricted when the instrument is not being used, and the skin of the forearm is not typically dedicated to the reception of other information. The controller can potentially be forgotten and ignored, and treated as part of the body. It can become a new skin that allows the user to pereive and interact with a virtual environment. All other skills are still in the game; for example, a user could play a real piano or violin while wearing the controller, and simultaneously interact with the space around the physical instrument.

Before the parade of previous research in this field begins, homage must be made to the ultimate hands-free controller that has dominated almost all genres of music for all of recorded history: the voice. Singers have physical advantages over other instrumentalists in that they are not rooted to a spot or bound to any object; they can move around on stage, and have no external instrument to put down when not singing. These are the advantages that the author sought in the design of the forearm controller.

### 1.3.1 The Theremin

In 1920, Leon Theremin unleashed his "aetherphone" at the Physico Technical Institute. This instument, now referred to as a Theremin, was the first open-air controller, consisting of two antennae that measured the electric field from the player's hands. Moving the right hand relative to its antenna controls the pitch of the instrument, and moving the left hand controls the volume [Cha97]. The design of the instrument makes gradually changing, sliding notes particularly easy to play; as a result, this electric sliding sound has become the hallmark of the Theremin, probably best known by its use as a sound effect for alien spaceships in 1950s science fiction movies. While the Theremin does free up the body of the performer locally, it is still rooted to one spot containing a box with electronics, and two antennae. Other instruments will soon be discussed that are

wearable, truly allowing the performer the freedom to move while onstage.

The Theremin's virtuoso, Clara Rockmore, was one of the few who attempted to master an instrument that is especially difficult to play due to its lack of tactile feedback [Cha97]. The fact that the Theremin is a simple, yet interesting instrument that suffers so notably from this lack of feedback has pushed several researchers to use it as a testbed for adding a haptic channel to a device that doesn't have one [RH00], [Mod00].

### 1.3.2 The Termenova

The Fretted Theremin implemented later in this thesis uses ARMadillo to provide a tactile solution to the virtuosity problem presented by the Theremin, and it follows in the footsteps of other haptics research, such as that presented by [RH00] and [Mod00]. However, it is worth considering other, non-tactile approaches to the problem.

For example, a system of visual frets could be implemented, allowing a performer to see notes in space. This approach was taken by [Has01], in which a Theremin-like instrument dubbed the Termenova used a system of lasers and capacitive sensing to track a user's hand, and allows the lasers to serve the additional purposes of providing a set of visual frets for the performer, and interesting imagery for the audience.

While a setup involving a large hardware installation does not satisfy the design constraints met by the forearm controller, the visual frets used in the Termenova solve a virtuosity problem not dealt with in this thesis, or in the work of [RH00] and [Mod00]; they allow a performer to see all the notes in space at once, allowing them to easily choose one and move to it. For a true virtuosic instrument, a combination should ideally be used, such that a user can see and feel the instrument, and choose to access their visual and tactile cues however they wish.

### 1.3.3 The Sensor Chair

A recent instrument created at the Media Lab for the magicians Penn and Teller was an electric field sensing device called the Sensor Chair [Mac96], [PG97]. The player sat in a chair containing a transmit electrode in the seat, and waved hands in the air near an array of receiving electrodes built into the arms of the chair. Unlike the Theremin, in which the antennae sense the highly irregular parasitic capacitance between the device and the player's hands, the Sensor Chair makes contact with the player through the seat electrode, and uses this to its advantage by sending an easily detected signal through the performer's body. This allows it to filter out stimuli not generated by the player touching the transmit electrode [PG97].

While the Theremin had two one-dimensional controls, one for each hand and antenna pair, the Sensor Chair uses an array of electrodes to generate a single two-dimensional space in front of the player [PG97]. Any continuous audio or MIDI sounds can be mapped to this region of space; for example, a one-hand Theremin-like mapping could be created in which the player controls pitch by moving along the X axis and volume by moving along the Y axis. Other possibilities include a percussion set, in which the player strikes the space in different locations, each of which have been mapped to a percussion sound.

### 1.3.4 The Conductor's Jacket

Teresa Marrin created a Conductor's Jacket at MIT, which monitored several physiological functions in addition to motion tracking [Mar00]. The jacket contained four EMG sensors, a respiration sensor, a heart rate monitor, a galvanic skin response sensor, a temperature sensor, and included an eight-sensor magnetic position tracking system. This system was intended to measure the gestures and physiological responses of a conductor

and translate them directly into music.

Such a system is interesting in the context of this thesis because it addresses some of the same design constraints in that it is intended to be discreet. Expanding the forearm controller to track more of the performer's body would naturally tend toward a system like the Conductor's Jacket, and building a tactile display into such an interface, while replacing the magnetic tracking system with one that is independent of a base station, would have interesting consequences.

### 1.3.5 Glove Controllers

Glove controllers should be briefly mentioned, as they have become a staple of computer music controllers since the introduction of the Mattel Power Glove in the 1980s [Cha97]. Michel Waisvisz built a system called "Hands", which contained metal plates strapped under the hands, with keys that responded to finger touch, and more elaborate sensors tracking thumb position and relative hand distance [Cha97]. Soon after, Laetita Sonami created the Lady's Glove, containing a set of flex sensors for tracking the fingers and the wrist, and magnetic sensors in the fingertips and a magnet in the thumb, allowing her to measure the distance from each fingertip to the thumb. She also used a pressure sensor to detect the thumb pressing against the index finger, and an ultrasonic tracking system that measured the distance between her hand and her belt, and her hand and her shoe [Cha97].

The classification of gloves within the framework of hands-free instruments is tricky, because one must first determine whether a gloved hand is free. Gloves were eventually rejected by the author, as they did not quite meet the constraint of being discreet enough to be worn constantly without interfering with natural movement. Although gloves could, in theory, be made thin and flexible enough to be worn regularly without being troublesome, those with embedded bend sensors do not typically meet this constraint.

### 1.3.6 Musical Trinkets

An alternative to tracking the hand with a glove is to track the fingers individually with rings. Such a system could allow a similar data stream, and expressive control, without the restrictive feeling of an enclosed hand. A system of resonant tags built into rings was used in "Musical Trinkets", and allows this sort of tracking within the small range of a magnetic coil. The tags themselves are simple, consisting only of an inductor and a capacitor, and require no power [Par03].

### 1.3.7 Other Hands-Free Music Controllers

There are other measurement techniques that have little in common with the hardware used in this thesis, but involve an innovative approach to solving a similar problem: the control of a system without rooting the body or hands to a specific location in space. For example, some researchers have been attempting to measure brainwaves directly, in order to allow a performer to control a musical system mentally. This is an extremely attractive possibility that suffers from a general lack of understanding of EEG signals, but may eventually prove to be fruitful [MB05]. While this example is hardly similar to the forearm controller, it should serve as a reminder that there may be other solutions to the design problems presented here.

## 1.4 Musical Applications of Tactile Feedback

It is well known by musicians that the sense of touch is crucial to virtuosic performance. As a pianist, it is easy to imagine performing without the use of the eyes, but impossible to imagine performing without being able to physically feel the instrument. Even for music that requires moving the hands quickly from one part of the keyboard to another, a technique that requires looking at the keyboard, it is often

necessary to choreograph the manner in which touch and vision will interact. For example, if the left hand must execute a small jump along the piano keyboard, while the right hand has a larger jump, the player must prepare to look at the destination of the right hand, and find the left hand's landing place by feel. For these types of advanced pieces, the visual and tactile channels are controlled and serve specific purposes.

### 1.4.1 Feedback in Keyboard Instruments

After years of research, electronic keyboards finally contain a physical action system that enables them to feel somewhat like a real piano keyboard. It is telling that such complicated physical modeling and mechanical design was required to satisfy the needs of pianists. In fact, the haptic feedback of a keyboard can dictate whether a piece is playable or not; if the action is too heavy, fast passages become difficult because pressing down the keys requires more force. However, if the action is too light, these passages are also difficult because the muscles on the back of the hand are required to lift the fingers after striking a note, rather than letting the spring action from the keys themselves push the fingers back to their starting positions.

Claude Cadoz created a force-feedback keyboard using 16 keys powered by mechanical actuators [Cad88]. More recently, researchers in Italy built a similar system, using voice coil motors to provide an active force function, allowing them to generate any action by simply changing the software. For example, by instructing the motors to resist with a greater force as the key was depressed, a spring-like piano action could be generated. By programming a large resistance during the initial movement of the key, followed by very little resistance once the key had started moving, a harpsichord-like plucking sensation could be generated [OP02].

### 1.4.2 Tangible Theremins

With open-air controllers such as the Theremin, the only physical feedback

available is the body's own sense of where it's limbs are, or egolocation; this becomes the primary tactile channel for controlling the instrument [RH00]. Rovan and Hayward hypothesized that open-air controllers would benefit from tactile feedback, and added a single vibrating actuator to their system, in the form of a ring worn on the fourth finger. They also tested a system that allowed the performer to feel vibration feedback through the feet [RH00].

In recent research, Sile O'Modhrain demonstrated that, with a Theremin, adding force feedback that was correlated to pitch was helpful in playing the instrument. By generating an easily understood haptic map of the space used by the Theremin, she was able to open the tactile channel that had been broken by the original design of the instrument [Mod00].

## 1.5 Aesthetic Tactile Applications

While the above research deals with the utility of tactile feedback in music, it should be noted that there is an aesthetic component which can make the playing of an instrument more pleasurable. A common complaint about music controllers in general is that the music can't be felt through the controller; this sense of being coupled to the music through physical feedback and the vibration of the instrument can be quite powerful.

In Sile O'Modhrain's research, an attempt was made at creating a virtual bowed stringed instrument, in which frictional and normal forces were simulated. While the friction simulation was not realistic enough to aid in the playing of the instrument, and was even found to detract from the player's abilities in some cases, the players commented that they preferred the feel of the instrument when the simulated friction was activated [Mod00]. Even a tactile feedback system that is not practically useful in playing the instrument is, apparently, preferable to not having any feedback. In short, successful instruments should feel good as well as sound good.

The sense of vibration that is associated with music can also be decoupled from the controller and simply displayed on the skin as part of an artform. This would not be

considered feedback, but still an application of a tactile display with respect to music. Eric Gunther used his vibrating full body suit, SkinScape, to write music with a complete tactile score in addition to the audio one, opening up a new field of tactile composition, in which the skin of the human body receives the artwork as the ears receive the music [Gun01].

## 1.6 Tactual Text

Tactile reading is, obviously, a useful field for helping those who are seeing-impaired to be independent. However, it should be noted that there are merits to tactile reading that could extend beyond this population. Reading with the skin is convenient in that it doesn't require dedicating senses that could be used for long range perception, such as sight and hearing. It can be done in the dark, and doesn't cause eyestrain. In fact, since reading involves receiving information that is within reach of the skin, it could be argued that, particularly for linear text that does not require extensive scanning and jumping, the sense of touch is actually the most logical choice for the job. With that in mind, some possible systems will be examined.

### 1.6.1 Introduction to Braille

The most commonly used tactile representation of text was created by Louis Braille, a student at the Royal Institute for the Young Blind in Paris, in order to replace an awkward system based on simply raising letters within text. In 1821, he met Charles Barbier, a soldier in the French Army, who was working on a system of "night writing", based on raised dots, for military use [Bur91].

Barbier's system was not easily learned, partially because it was based on an overly complicated 12-dot system, and did not include any punctuation. Over the next few years, Braille devised his own system, and by 1829 had developed a six-dot system that could represent text, punctuation, and music notation. He published it as a "Method

of Writing Words, Music, and Plain Songs by Means of Dots, for Use by the Blind and Arranged for Them", and continued to improve the system over the course of his life [Bur91]. It is interesting to note that music may have been one of the primary motivating factors in the development of Braille.

With six dots, Braille has 64 characters in total, including the empty character, which represents a space. The entire set of Braille characters is shown in Table 1-1, along with their Braille ASCII representation, which was created as a standard for generating Braille text files to be read from tactile displays [Brl94]. An extended, eight-dot Braille does exist, and is used for electronic Braille displays, with the lowest row used to indicate bold or underlined text, or the position of the cursor.

| Braille/ASCII/Dec. | Braille/ASCII/Dec. | Braille/ASCII/Dec. | Braille/ASCII/Dec. |
|---|---|---|---|
| (SPACE) 32 | ⠴ 0 48 | ⠈ @ 64 | ⠰ P 80 |
| ⠮ ! 33 | ⠂ 1 49 | ⠁ A 65 | ⠸ Q 81 |
| ⠐ " 34 | ⠆ 2 50 | ⠃ B 66 | ⠼ R 82 |
| ⠩ # 35 | ⠒ 3 51 | ⠉ C 67 | ⠎ S 83 |
| ⠫ $ 36 | ⠲ 4 52 | ⠙ D 68 | ⠞ T 84 |
| ⠿ % 37 | ⠢ 5 53 | ⠑ E 69 | ⠥ U 85 |
| ⠷ & 38 | ⠖ 6 54 | ⠋ F 70 | ⠧ V 86 |
| ⠄ ' 39 | ⠶ 7 55 | ⠛ G 71 | ⠺ W 87 |
| ⠣ ( 40 | ⠦ 8 56 | ⠓ H 72 | ⠭ X 88 |
| ⠜ ) 41 | ⠔ 9 57 | ⠊ I 73 | ⠽ Y 89 |
| ⠪ * 42 | ⠱ : 58 | ⠚ J 74 | ⠵ Z 90 |
| ⠬ + 43 | ⠰ ; 59 | ⠅ K 75 | ⠪ [ 91 |
| ⠠ , 44 | ⠣ < 60 | ⠇ L 76 | ⠳ \ 92 |
| ⠤ - 45 | ⠿ = 61 | ⠍ M 77 | ⠻ ] 93 |
| ⠨ . 46 | ⠜ > 62 | ⠝ N 78 | ⠘ ^ 94 |
| ⠌ / 47 | ⠹ ? 63 | ⠕ O 79 | ⠸ _ 95 |

**Table 1-1: Braille ASCII**

### 1.6.2 Grade 1 and Grade 2 Braille

Note that only the letters of the alphabet correspond to the meanings of their Braille ASCII characters. The ASCII character '=', for example, corresponds to a Braille character ⠿ , which is a contraction for the word 'for'. With 64 characters, the most common form of literary Braille has a complicated set of contractions that are optimized so that the most common words, prefixes, and suffixes, are shortened whenever possible. For example, the word "the" has its own character, ⠮ . The character for "K", or ⠅ , when

appearing by itself, is assumed to stand for the word “knowledge”. This Braille system is known as “Grade 2” Braille [Brl94].

### 1.6.3 Braille Display Hardware

The dominant Braille display hardware involves dots that are lifted by piezoelectric reeds, which are driven by power supplies in the neighborhood of 300V. These piezo displays are usually designed as add-ons to ordinary computer keyboards, and might have 40 characters in a single row. In addition, it is now possible to buy self-sufficient Braille PDAs [Hum05].

There have been several attempts at making a Braille display in which the user keeps the hand in one place, and the text scrolls across it. One that looks promising is a NIST project involving a rotating wheel with Braille dots popping up along its outer rim [Rob00]. In such a display, the Braille characters would literally be moving across the finger.

Two more attempts involve virtual Braille characters that only seem to move. In one experiment, electrocutaneous stimulation was used to simulate the entire tactile sensation [Kaj01a]. In another, lateral deformation of an array of actuators was used to simulate movement, and create the sense of the dragging friction one would feel when moving a finger across a Braille character [Pas04].

## 1.7 Interesting Non-Braille Text Systems

While Braille was chosen as the basis for the text system presented in this thesis, other possibilities do exist. Braille was not chosen because it is the best tactile system for transmitting text, but because it is already known by many sight-impaired people, and the goal was to create a text-based system that would leverage prior skills and allow Arm Braille to be quickly learned. Another approach, taken by [Gel60], [Tan96], and

[MHK00], is to create a completely new tactile language. These will be briefly described for comparison with Braille and Arm Braille.

### 1.7.1 Optimizing the six dots

The six-dot system used in Braille was assembled intuitively by Louis Braille without the help of the mathematics of information theory, entropy, and models of perception. An optimized six-dot system may exist that has certain advantages over that used in traditional Braille, and such possibilities were explored by [MHK00]. More specifically, they chose to optimize the system so that the dots are used at equal or near-equal frequencies; this optimization doesn't help a user read or learn any faster, but it might allow the displays themselves to last longer, do to the even spread of wear throughout the pins.

However, the six dot system [MHK00] presented would certainly be markedly harder to read than the Grade 1 system traditionally used in Braille and Arm Braille. For example, the power optimized system suggested by [MHK00] uses all the single dot characters ⠁ ⠂ ⠄ ⠈ ⠐ and ⠠, which are hard to distinguish visually, and certainly hard to distinguish tactually. Traditional Grade 1 Braille, on the other hand, only uses the one single-dot character ⠁ for "a". While Grade 2 Braille does use all 64 combinations of six dots, there are many complex safeguards for ensuring that similar combinations of characters are prevented by context, and in certain cases, a Grade 1 word will be used within a Grade 2 passage if any tactile ambiguity would be found in the Grade 2 encoding [Brl94]. The approach is certainly interesting, and should be applied to a more complex model that includes tactile percption as well as energy efficiency.

### 1.7.2 Vibratese

Using vibrating motors to display text opens up possibilities for varying timing and intensity in addition to location when organizing a set of tactile characters. A tactile

language that showed no similarity at all to Braille was constructed in the 1950s, and called “Vibratese”. It used a set of five vibrating actuators located on the four corners of a rectangle on the torso, with a fifth actuator in the center. Each motor could be driven with three durations of 100ms, 300ms, and 500ms, and three intensities labeled “soft”, “medium”, and “loud” [Gel60], [Tan96].

This mapping allowed the system to include 45 characters, each involving a single vibratory pulse. Taking advantage of a sparse motor configuration over a large surface area ensured that the location of a motor was easily distinguishable. Vibratese was taught to several subjects, one of whom reached a peak reading speed of 38 words per minute, where words averaged five characters each [Gel60], [Tan96].

### 1.7.3 The Tactuator

A non-Braille system of tactile text was created in [Tan96], and called the “Tactuator”. This system was different from Vibratese and Arm Braille in that it used force-feedback rather than vibration to display the text, utilizing a system of motors and sensors that interfaced with three of a user’s fingers. The display was able to generate a set of 120 elements, each 500ms long, involving a variety of frequencies and waveforms.

The Tactuator was tested on three subjects, including the author of [Tan96]. The other two users were able to master the set in 20 and 27 hours respectively, and were able to receive information at an estimated rate of 12 bits/second.

# 2 The Forearm Tracking System

The hardware design of ARMadillo was implemented so as to meet requirements for utility, convenience, size, and other parameters. The final controller board was driven by an 8051 microcontroller and included a set of sensors for arm tracking, a BlueTooth wireless module, a CompactFlash card slot, and a set of motor drivers capable of controlling up to sixteen vibrating motors. The physical layout of the system is shown below, in Figure , as it appears when unfolded.

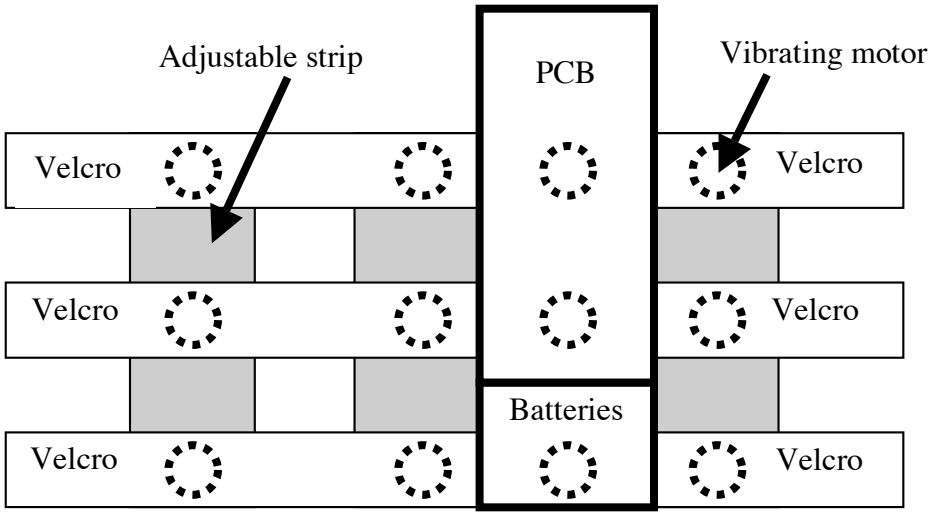



**Figure 2-1: High-level hardware layout**

The controller was designed to be self-contained and modular. The CompactFlash card slot gave the device the potential to interface to gigabytes of storage, GPS data, or a WiFi stream, and a set of headers allowed the controller to be interfaced to other wearable boards. Though most of the applications described in depth in this thesis involve external processing and memory, and therefore don't require these extensions, some projects that were implemented in the early stages of development were independent of an external computer, and required additional memory extensions via a CompactFlash card.

When possible, the design of the sensor suite in the controller was borrowed from the Stack architecture by Ari Benbasat, Joe Paradiso, and Stacy Morris [BP05], which included an IMU containing three axes of accelerometers and three axes of gyroscopes. The layout of the Stack IMU was altered to allow the entire board to lie flat, and combined with additional sensors for detecting three axes of magnetic field, and integrated into the rest of the forearm controller PCB.

A simplified schematic is shown in Figure 2-2.

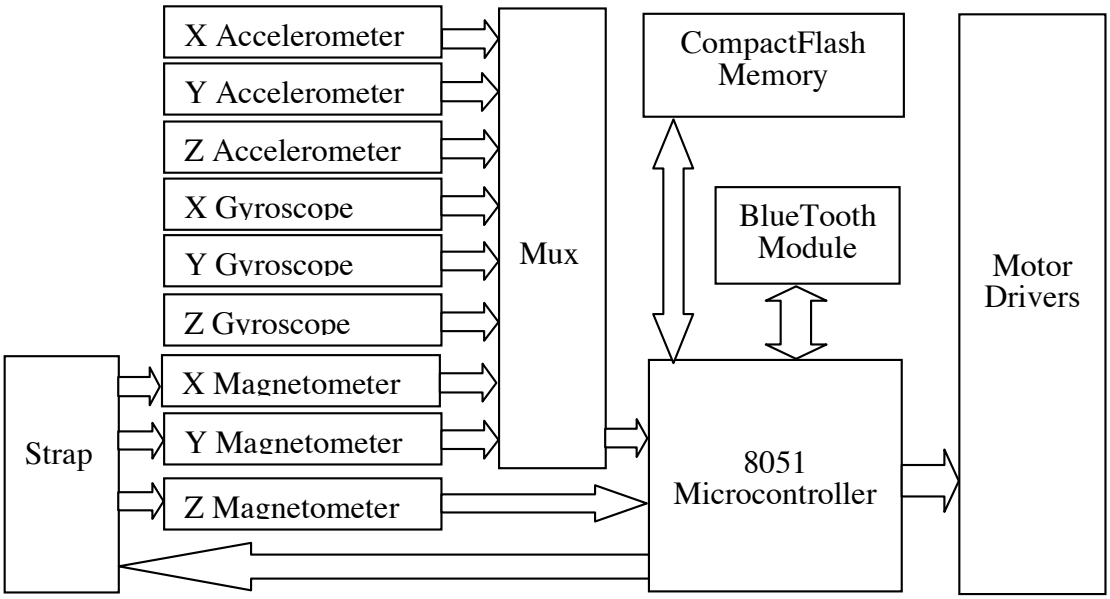



**Figure 2-2: Simplified hardware schematic**

The strapping mechanism on the left side of Figure 2-1 is required to keep the magnetometers from drifting due to external magnetic fields. It sends a brief pulse containing a large current through the magnetometers, resetting them. This can be done at varying time intervals, before every update, or when saturation has been detected [Hon05].

## 2.1 Design Constraints

The design of the forearm controller was constrained to meet the following goals:

1. It must be trivial to put on and remove, and therefore exist in one piece that can be both felt and controlled.

2. It must be possible to wear it discreetly beneath clothing.

3. It must be possible for it to disappear psychologically, and be approached as a new physiological sense.

In order for a single object to be felt and controlled, the hands or arms are logical choices, as the nerves are more dense in this part of the body, and the hands and arms are unrivaled in their control ability [LL86]. Separating the display from the controller would allow for other possibilities, such as controlling with the hands while feeling with a display located on the torso; however, such a device would be inconvenient to put on and take off, and was rejected. The hands were also rejected, although they are obviously the primary tools for both controlling and tactile sensing. For a device to be worn in everyday life, a hand-based system such as a glove controller would be awkward, and was not considered.

The requirement for the device to disappear psychologically is, perhaps, the most important one. The final implementation of the forearm controller satisfied these constraints, and was designed to emulate a wristwatch in that it could be forgotten during natural movement, and had no need to be taken off or put down.

## 2.2 Physiological Considerations

In the design of a tactile display, the types of actuators and their locations must be carefully chosen, as all parts of the human anatomy do not respond equally to tactile

stimulation. In addition, the manner in which the brain responds to and tracks such stimulation must be taken into account in order to ensure that a text display will be legible. A useful background for such physiology is given in the *Handbook of Human Perception and Performance* [LL86].

### 2.2.1 Choosing the Forearm

Traditionally, the spatial resolution for a given part of the body was tested with a two-point threshold test, in which a user is asked to determine whether one or two points are being stimulated on a given section of the skin, as the stimuli are executed over an increasingly large surface area [LL86]. However, for vibratory displays such as the one used in this research, more pertinent studies can be examined. [CCB01] and [PJ05] tested the spatial pattern recognition on the forearm and the torso using vibrating motors, and their results were considered in the design of this tactile display. The torso and the forearm are both attractive sites for such displays, as the skin is sensitive and not typically dedicated to the reception of other information.

While [PJ05] concluded that the torso was superior to the forearm in recognizing patterns in arrays of vibrating motors, the forearm was ultimately chosen as the site for this thesis. The torso does not provide a useful site for user control, whereas the forearm can be moved in complex ways, as well as having a sensitive stretch of unused skin. The design constraints dictated that the final device should be small, in one piece, and be used for both input and output. While the torso might be effective in a system that uses a separate device for input, such a design was not considered.

However, to design an effective arm display, it is worth examining the context in which [PJ05] rejected the forearm as their site. Their array of motors was similar to the one used in this thesis, in that it was a 3-by-3 matrix. However, their spacing was 24mm, and [CCB01] determined experimentally that a forearm display with a spacing of 50mm fared markedly better than one with a spacing of 25mm in a pattern recognition test. The spacing chosen for the display in this thesis was 50mm, so as to leverage the conclusions of [CCB01].

[PJ05] chose to use a small display so that it would fit even a slim forearm. It is assumed that this design constraint was chosen to keep the display on the underside of the forearm, an understandable decision given that the underside is hairless, and hairless skin is known to be more sensitive and has a better spatial resolution [LL86]. The larger forearm display used in this thesis tended to wrap around the forearm for most users. However, the positive consequence of this wrapping is that some of the motors lay on the bones on either side of the forearm, and felt quite different from the motors that lay directly on the underside of the forearm, making it easier to differentiate between motors at these locations. This also leverages the results of [CCB01], in which it was discovered that locating tactile stimuli was easier when an identifiable "anchor point", such as a wrist or an elbow, was near the stimulus. In this case, expanding the display to a spacing of 50mm created three anchor points, the wrist bone, the radius, and the ulna. In addition, motors not triggering the detection of these anchor points must necessarily be directly under the forearm.

One of the most difficult forearm patterns to identify in [PJ05] will be reexamined in section 3.3.2. It will be noted that, although the pattern was also difficult to identify in its first implementation in this thesis, adding a subtle change using PWM and timing caused the pattern to become startlingly clear. Therefore, while simple patterns should certainly be used as benchmarks when comparing parts of the body, it should also be understood that adding a subtle complexity to a sensation can greatly clarify it, and such subtleties are certainly functions of the anatomy on which the sensations are being displayed.

### 2.2.2 Active and Passive Touch

When transmitting a tactile image, it is important to differentiate between active and passive touch, as defined by Gibson [LL86]. Passive touch, in the context of this thesis, involves a tactile sensation that is displayed without control from the user, whereas active touch requires control. In a sense, a passive sensation can be thought of as

a series of images that are displayed on the user's skin, whereas an active sensation is closer to a virtual object or medium that has been created for a user to explore.

More importantly, while many displays and experiments use sensations that are fixed in time and are therefore passive, including the test done by [PJ05], it has been shown that active sensations are easier to perceive [LL86]. Most of the sensations in this thesis, and all of the sensations used in the final implementation of Arm Braille, are active, in that the user is expected to provide some control, and can explore the tactile objects or characters at will.

## 2.3 Mechanical Considerations

### 2.3.1 The PCB

The controller design was limited by the shape of the forearm on which it was to be worn. The length of the board was therefore quite flexible, but the width could not exceed that of a small human forearm, and the height of the board had to be minimized so that the controller could be easily worn under clothing. The limiting factor for choosing the width and the height of the board was the CompactFlash card slot, and components on the top of the board were chosen and arranged such that they didn't exceed these dimensions. In addition, the board layout was structured such that the bottom side contained only surface mount components with a very small height. Keeping all the tall components on one side of the board allowed the final controller to be as thin as possible.

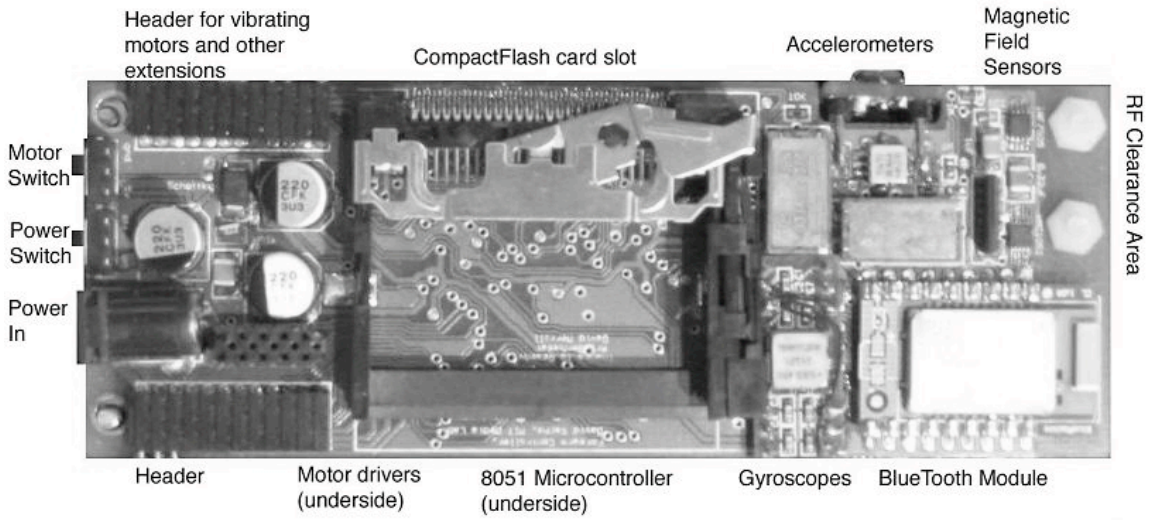



**Figure 2-3: Forearm controller PCB**

The inertial sensors had to be placed carefully, so that all three axes were observable for each of the three sets of sensors. In addition, they had to be positioned close to each other, so that the moment arm of one inertial sensor relative to another would be negligible. Finally, the accelerometers in particular had to be placed as far as possible from the vibrating motors, which generate negligible noise in the gyroscopes and magnetic field sensors, but a noticeable addition of noise in the acceleration measurements. For this reason, the vibrating motors that were located directly under the board were eventually removed. Finally, the BlueTooth module required a clearance area void of metal parts along the axis of its antenna [Blu05]. This region can be seen at the right side of Figure 2-3, containing only two plastic mounting screws.

### 2.3.2 The Tactile Display

The board was mounted on an acrylic platform and connected to a configurable fabric tactile display, with a simple Velcro design that allowed it to be attached to arms of varying widths while maintaining a thin profile. The Velcro design fixed the location of a set of vibrating motors on either side of the PCB, and directly beneath it, forcing the

wearer to center the tactile display on the back of the forearm, simply by centering the PCB itself; the motors on the underside of the forearm were embedded in a movable fabric strip that could be adjusted to allow for thicker or thinner arms.

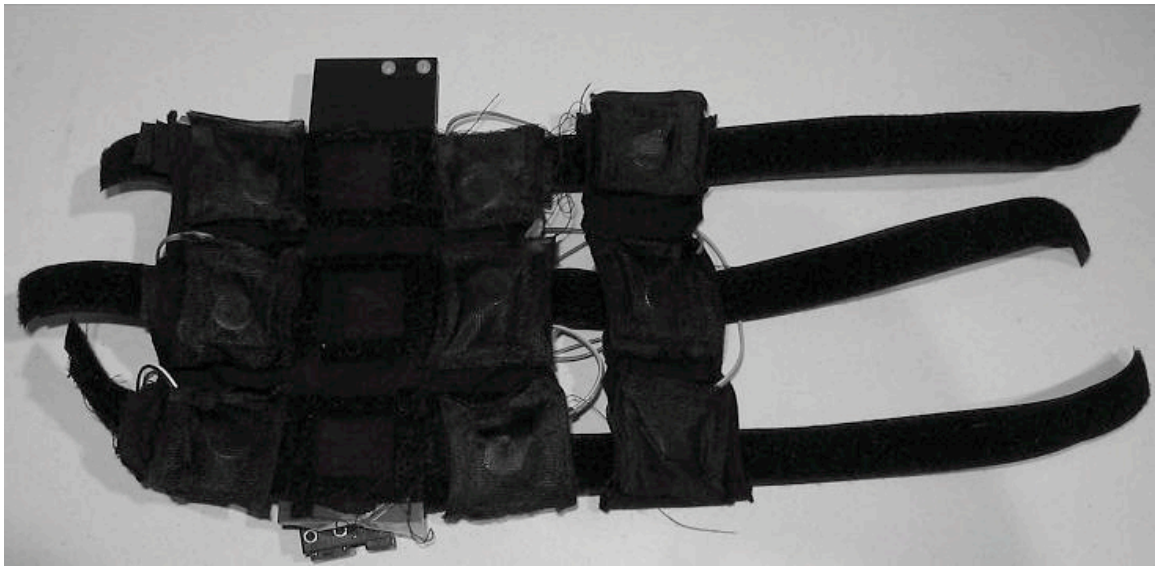


**Figure 2-4: Velcro design**

The vibrating actuators were only 0.14" thick, and could be integrated easily into the thin fabric interface that was used in the final version, as is shown below. They were driven by open-drain latches, allowing the maximum applied voltage to be varied during the application development. This voltage was eventually fixed at 9V.

**Figure 2-5: Vibrating motor**

Finally, an adapter for powering the device from a battery pack was constructed, and when the batteries were used, they were placed on the side of the board closer to the elbow, reducing the rotational inertia of the board relative to that joint, and making it easier for a user to make sudden angular movements. Two rectangular 9V batteries were placed in parallel to increase the available current for the large transients occasionally required by the motors, which sometimes exceeded 1A, and to maximize the life of the battery pack. This life was not measured, as the batteries were only used for demo purposes, and the original pack still functions.

## 2.4 Sensor Selection and Calibration

As it was decided early on that ARMadillo would be constrained to be compact, wireless, and function without an external base station or exoskeleton, the best choice for the sensing method was a suite of inertial sensors containing three axes of accelerometers, three axes of magnetic field sensors, and three axes of gyroscopes. Such a system can only truly measure three degrees of orientation, although the accelerometers were also used to measure quick, linear transient movements. A position sensing addition similar to the one used in the Lady's Glove [Cha97] was considered, which would involve ultrasound emitters and sensors located at various places on the body, but this method was eventually abandoned, due to a determination to keep the controller in one piece which could be easily put on and removed.

The sensors used were as follows:

| Sensor | Measurement | Axes | Num of Sensors Used |
|---|---|---|---|
| ADXL202 | Acceleration | 2 | 2 |
| HMC1052 | Magnetic Field | 2 | 1 |
| HMC1051Z | Magnetic Field | 1 | 1 |
| ENC-03J | Angular Velocity | 1 | 2 |
| ADXRS150 | Angular Velocity | 1 | 1 |

**Table 2-1: Sensors**

The accelerometers were calibrated by turning them until they pointed down, and then turning them until they pointed up, and recording the maximum and minimum output values due to gravity for each sensor.

The magnetic field sensors were more difficult to calibrate. They were turned in a horizontal plane, using the accelerometer values as a reference, until they reached their maximum or minimum values. At the equator, this would correspond to the true maximum and minimum values of the magnetic field vector, but in Boston the vector runs at an angle with the horizontal, known as the dip angle, and this irregularity is coupled with the magnetic distortion typically found inside buildings. To find the true maximum and minimum, the sensors were then turned in a vertical plane until they reached their maximum and minimum values. The values directly in between were biased to zero, and the output was scaled to vary from 1 to -1 for stationary orientations. Due to the different dip angles that occurred in different rooms that the forearm controller was used in, the dip angle was tracked dynamically in the filter.

The gyroscopes were calibrated by attaching to a custom test jig and turning them on a stepper motor with 200 steps per rotation. This allowed a specific angular velocity to be generated, and the resulting gyroscope measurements taken. Four angular velocities were used, and the resulting scale factors averaged to find a final scale factor. The initial gyroscope bias was found by taking measurements from the gyroscopes while they were stationary. However, it was found that the gyroscope bias tended to drift, particularly

during the first minute after they were turned on, so the gyroscope bias was also tracked dynamically in the final filter.

## 2.5 Firmware

When writing the early versions of the firmware for ARMadillo, all of the processing was done on board the 8051. This included rudimentary motion tracking algorithms, software drivers for the vibrating motors, and accessing the CompactFlash card for any extra memory that was needed. Hard-coding these processes into the 8051 was tedious and limiting, but provided true independence and very low latency. Later, to allow for more complex filtering and rapid prototyping of the software, the firmware was reduced to a simpler program that could relay the sensor data to a PDA or a laptop in a backpack, and drive the motors with pulse width modulation. The more powerful processor in the external computer would handle the filtering in C++, and the tactile mapping in Python.

### 2.5.1 Hard-Coding Projects

The main advantage of coding projects directly into the microcontroller is independence from external processing, and three large projects were implemented in such a way. The most elaborate was a complete Braille E-Book, which read Braille ASCII files from a CompactFlash memory card and displayed the results in an Arm Braille system that will be described later. The final Braille system that was tested on users used external processing and a Python script to log the movements of the users, but the initial implementation was a completely independent 8051 system.

Two musical projects were also written directly into the 8051: a tactile mapping of musical passages by Stravinsky and Xenakis, and a simple wearable remote control system for selecting sounds, both of which will be described in the next chapter. These

used the BlueTooth module to communicate with the external Max/MSP patch being controlled, but required no feedback; all the motion tracking and tactile mapping was done on board the forearm controller.

The advantages of this approach were most noticeable when using the BlueTooth module. The BlueTooth serial channel requires 30-50ms to reverse directions, and therefore a single BlueTooth module limited the update rate to about 10 Hz when in feedback mode. Adding a second module would easily fix this problem, but would have made the board more costly. Therefore, the BlueTooth module was only used in the feedforward applications that were just described. A new revision to this board should have a solution that allows for a feedback wireless system, using two wireless modules if necessary.

### 2.5.2 Using an External Processor

The disadvantages of using a hard-coded system are that, due to the difficulties in debugging complex 8051 systems, and the limitations posed by the 22MHz clock speed, the applications must be kept simple, and take longer to implement and to alter. Though it may be possible to eventually hard-code the most complex of the applications listed here, and should certainly be possible if the controller board were revised with a more powerful processor, for research purposes it made more sense to do the programming that requires speed or rapid prototyping on an external computer, and reduce the 8051 code as much as possible.

The final firmware consisted of two routines that ran simultaneously, with their computations interlaced. The first routine polled the sensors, while the second simultaneously controlled the motor's PWM. The microcontroller used has a 12-bit A/D converter, and the entire system was limited by the data bit rate, which had to be compressed as much as possible. The magnetic field sensors were "strapped", or reset, for each sensor update. The strapping had to occur while polling the other sensors, because the magnetic field sensor data was invalid for several microseconds after being strapped.

### 2.5.3 The Feedback Data Packets

With nine sensors, each delivering 12 bits of information, the body of the sensor packet was 108 bits, or 13.5 bytes, rounded up to 14 bytes with the last half of the last byte, or nybble, set arbitrarily to 0x0. In the final implementation of the display, nine motors were used, each with a four bit PWM value that dictated the intensity at which that motor vibrated. This required a data packet of 36 bits, or 4.5 bytes, rounded up to five bytes. The last nybble was set arbitrarily to 0x8.

These two final nybbles, 0x0 and 0x8, were used to determine whether the controller and computer were in sync. If the controller were to find that the last nybble in the PWM packet was not 0x8, it would go into a loop in which it would wait patiently until it received a byte containing 0x8. It was the responsibility of the external computer to actively reconnect to the forearm controller. If the computer discovered that the last nybble of the sensor packet was not 0x0, or if 10 ms were to pass without a byte from the controller, the computer would begin sending a stream of bytes containing 0x8, until the controller responded with a sensor packet. This allowed the two to remain in sync and regain communication in case of a disruption due to a large burst of noise. The bit rate used was 57.6kbps, but with the overhead due to Python scripting, Max/MSP audio processing, and OpenGL rendering, the final update rate was about 200Hz, where each update consists of one sensor packet followed by one PWM packet. The packets are summarized in Table 2-2.

| **Sensor Packet (from forearm)** | |
|---|---|
| *Bits* | *Description* |
| 36 | Accelerometer Data |
| 36 | Gyroscope Data |
| 36 | Magnetometer Data |
| 4 | End of Packet (0x0) |
| **Motor PWM Packet (to forearm)** | |
| *Bits* | *Description* |
| 8 | PWM For Motors 1 and 2 |
| 8 | PWM For Motors 3 and 4 |
| 8 | PWM For Motors 5 and 6 |
| 8 | PWM For Motors 7 and 8 |
| 4 | PWM For Motor 9 |
| 4 | End of Packet (0x8) |

**Table 2-2: Data packets**

The PWM values were unpacked in the controller to five numbers varying from 0 to 15. These were stored in registers within the microcontroller, and compared repeatedly to a four-bit counter. Whenever a PWM value was above the counter, the corresponding motor would be turned on; when it was below the counter, the motor would be turned off. This allowed all nine motors to be controlled simultaneously with one PWM counter.

## 2.6 Sensor Fusion

While the applications implemented in this thesis were run either in 8051 assembler, if they were to be independent of external processing, or in Python, allowing for fast modification and rapid prototyping, the sensor fusion for ARMadillo was implemented in C++ and imported into the Python-based applications as a module. The code for this filter is given in the Appendix, along with code for an Unscented Kalman Filter and a Square Root Unscented Kalman Filter, all implemented with the assumption that the noises involved were Gaussian and additive. The final Kalman filter was able to iterate by itself in 0.48 milliseconds, averaged over 100,000 iterations, on a 2.2GHz Celeron processor, giving it a potential update frequency of about 1600Hz; however, in the applications described in this thesis, the data rate of 57.6kbps limited the update rate to about 380Hz, and further overhead by the Python scripts, OpenGL rendering, and Max/MSP real-time audio processing, brought the final update rate down to about 200Hz.

### 2.6.1 Euler Angles

There are many ways of representing orientation, and each has its advantages and disadvantages. Two will be described in this thesis. Euler angles are probably the most common method, and accomplish their task by breaking up a rotation into three parts, known as roll, pitch (or elevation), and yaw (or azimuth). These three rotations are easier to visualize intuitively than the quaternion methods that will be described subsequently, and the language used to describe Euler angles will therefore be found in this thesis; however, mathematical problems that arise when using these angles made them somewhat awkward to use in an arm tracking scenario. The three Euler angle rotations are shown in Figures 2-6, 2-7, and 2-8.

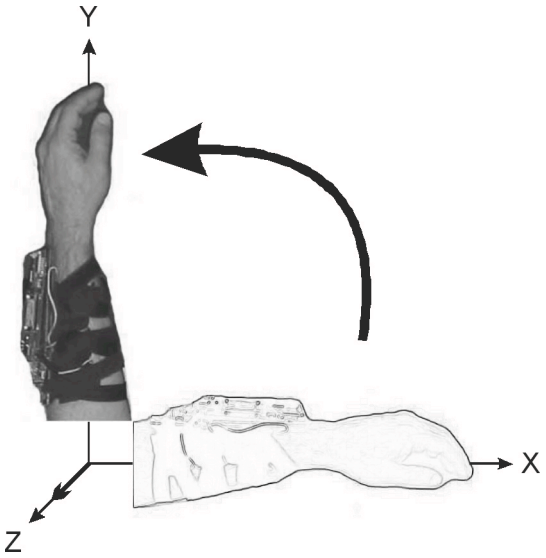



**Figure 2-6: Pitch, around the Z axis**

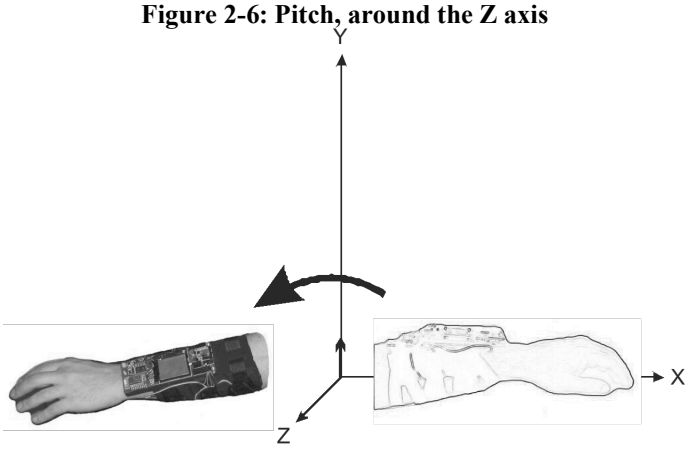



**Figure 2-7: Yaw, around the Y axis**

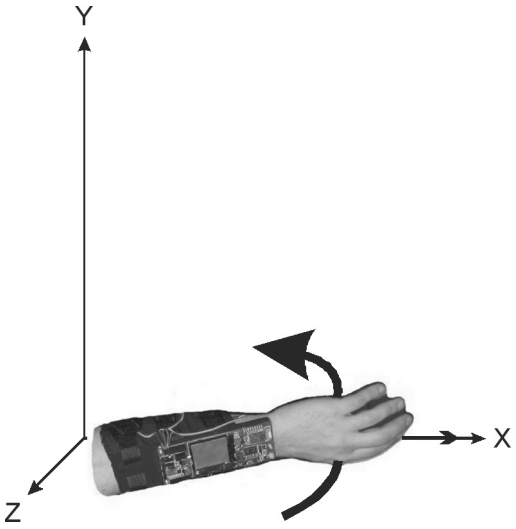



**Figure 2-8: Roll, around the X axis**

This method is problematic when the pitch is 90°, or when the arm shown above points along the Y axis. In this case, roll and yaw describe the same movement. Numerically, a singularity occurs here, which causes division by zero at 90°, and inaccurate estimation close to 90°. Despite this, there have been many effective implementations of Euler angle based inertial tracking systems, such as [Fox96], which were considered in the development of the filter used in this thesis. In some applications, the singularity found at 90° can be ignored if it will never be encountered in practice. For example, tracking systems for cars or underwater vehicles such as boats and submarines can survive a singularity, as long as the singularity is not placed on the yaw axis [TH03].

### 2.6.2 Quaternions

A second method for modeling an orientation with three degrees of freedom without a singularity, is a four component structure known as a quaternion. Intuitively, the concept of modeling three degrees of rotational freedom with a four component vector-like structure can be made clearer by examining an analogous single degree of freedom. When such a rotation, shown in Figure 2-8, is expressed as a single angle, it has

a discontinuity where 360° becomes 0°. Expressing it as a two dimensional vector has the advantage of eliminating the discontinuity, with the added complexity of an additional number to keep track of, and a normalization constraint. A 1DOF orientation is shown in Figure 2-9, with its traditional 1-dimensional angular representation of 45°, and 2-dimensional vector representation of (0.71, 0.71).

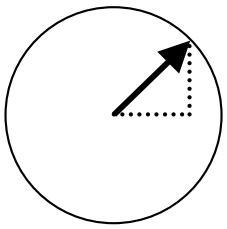


**Figure 2-9: Angle and vector representations**

It is not so easy to draw the 3DOF case, in which roll, pitch, and yaw are all being tracked; nevertheless, the corresponding quaternion can be thought of as sitting on the surface of a four-dimensional hypersphere. Rotation quaternions are not strictly four-dimensional vectors, however; their structure is cleverly designed to simplify rotational kinematics. Specifically, a quaternion (w, x, y, z) has a scalar component and three imaginary components, and can be expressed as [Sho94]:

$$q = w + xi + yj + zk$$

Where i, j, and k are unit length, imaginary, and orthogonal, such that [Sho94]:

$$i^2 = j^2 = k^2 = ijk = -1$$
$$ij = -ji = k$$
$$jk = -kj = i$$
$$ki = -ik = j$$

Representing a vector in quaternion form is as simple as leaving out the scalar part of the quaternion [Sho94]:

$$v = (x, y, z) \rightarrow q = (0, x, y, z)$$

A rotation quaternion can represent a rotation of a certain angle around a certain axis. Given an angle of θ and a unit axis (x, y, z), the corresponding quaternion is constructed as follows [Sho94]:

$$q = (\cos\frac{\theta}{2}, x\sin\frac{\theta}{2}, y\sin\frac{\theta}{2}, z\sin\frac{\theta}{2})$$

Rotation quaternions are always normalized, a necessary constraint for keeping them on the surface of a hypersphere. The normalization constraint can be proved as follows:

$$|q| = \cos^2\frac{\theta}{2} + x^2\sin^2\frac{\theta}{2} + y^2\sin^2\frac{\theta}{2} + z^2\sin^2\frac{\theta}{2}$$
$$|q| = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}(x^2 + y^2 + z^2)$$
$$|q| = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2} = 1$$

Given two successive rotations, $q_1 = (w_1, x_1, y_1, z_1)$ and $q_2 = (w_2, x_2, y_2, z_2)$, the final rotation is computed using quaternion multiplication, which is defined as [Sho94]:

$$q_1 q_2 = q_{final} = (w_{final}, x_{final}, y_{final}, z_{final})$$

$$w_{final} = w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2$$

$$x_{final} = w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2$$

$$y_{final} = w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2$$

$$z_{final} = w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2$$

This is used in the context of this thesis to apply more than one rotation in succession. In Figures 2-10 and 2-11, a 90° rotation about the Z axis, or (0, 0, 1), is followed by a 180° rotation about the Y axis, or (0, 1, 0):

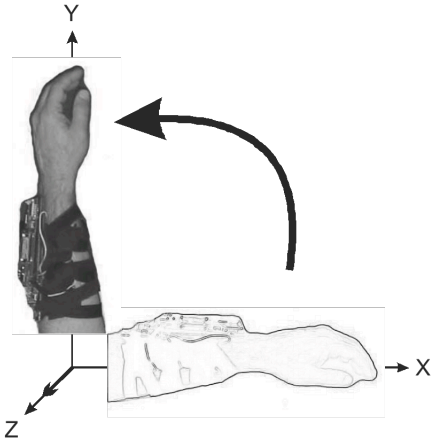



**Figure 2-10: Rotation A**

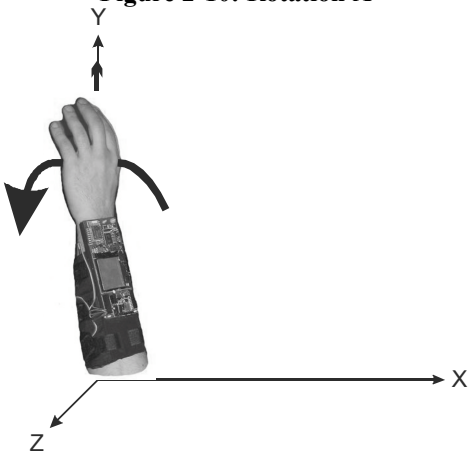



**Figure 2-11: Rotation B**

The quaternions assembled would be:

$$q_A = (\cos\frac{90°}{2}, 0, 0, \sin\frac{90°}{2}) = (0.71, 0, 0, 0.71)$$
$$q_B = (\cos\frac{180°}{2}, 0, \sin\frac{180°}{2}, 0) = (0, 0, 1, 0)$$

Applying quaternion multiplication gives us a single quaternion that reaches the same final orientation as the last two:

$$q_B q_A = q_{final} = (0, 0.71, 0.71, 0)$$

The axis and angle can be extracted from this as follows:

$$\theta = 2\arccos(w_{final}) = 2\arccos(0) = 180°$$
$$x_{axis} = \frac{x_{final}}{\sin\frac{\theta}{2}} = 0.71$$
$$y_{axis} = \frac{y_{final}}{\sin\frac{\theta}{2}} = 0.71$$
$$z_{axis} = \frac{z_{final}}{\sin\frac{\theta}{2}} = 0$$

This reduced the two rotations to a single one, a 180° rotation about an axis that is 45° between the X axis and the Y axis. As can be seen in Figure 2-12, this reduced rotation has an effect identical to the two rotations it was derived from:

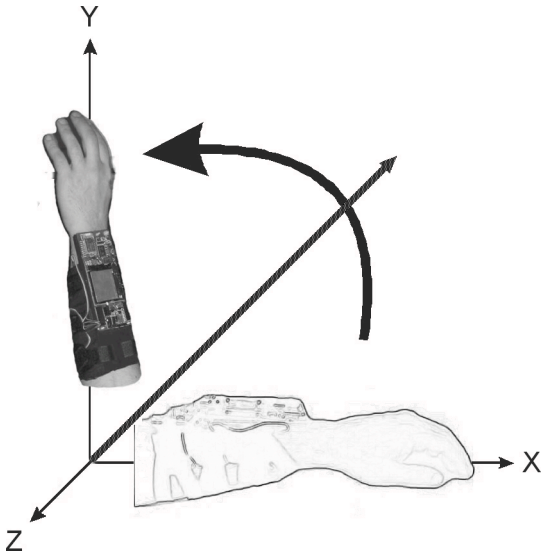



**Figure 2-12; The Combined Rotation**

In the context of this thesis, the forearm controller is assumed to have an initial condition of pointing along the axis (0, 0, -1), with the PCB on top. The Kalman filter returns a quaternion that expresses the rotation from this initial condition to the orientation estimate. If the filter returns the identity quaternion, (1,0,0,0), the orientation estimate has been computed to be this initial condition.

### 2.6.3 Extracting Useful Information From Quaternions

The axis-angle information encoded in the quaternion is enough to control an OpenGL animation, but it may be useful to extract other forms of information from the quaternion, such as an orientation in vector form. One way to do this is to use quaternion multiplication to rotate one vector into a new vector. Given a vector A in quaternion form (0, Ax, Ay, Az) that is rotated a certain angle about a certain axis given by the quaternion q, the rotated vector can be computer by:

$$B = qAq^{-1}$$

In which the normalized rotation quaternion is inverted by negating the imaginary part [Sho94]:

$$q = (w, x, y, z)$$
$$q^{-1} = (w, -x, -y, -z)$$

In the example depicted in Figures 2-10, 2-11, and 2-12, since the quaternion form of the orientation vector that points in the direction of the user's fingers is (0, 1, 0, 0), the new orientation vector is given by:

$$v = q(0,1,0,0)q^{-1} = (0,0,1,0)$$

This is the quaternion form of the vector (0, 1, 0), which describes the new orientation as pointing straight up. From here, other information such as pitch and yaw can be retrieved by looking at the orientation vector's projection in various planes. However, in this computation, the information about roll has been lost; it is not clear from the result that the PCB is on the positive X side of the forearm. If this information is desired, it can be retrieved by also rotating a normal vector, shown below:

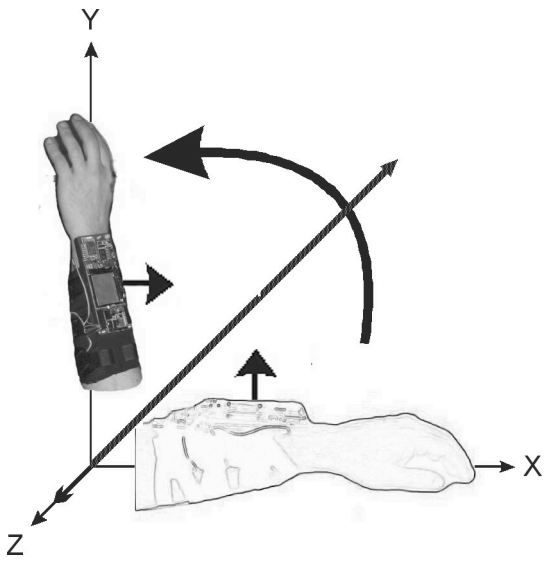



**Figure 2-13: Rotating normal vectors**

The initial normal vector is (0, 1, 0), or (0, 0, 1, 0) in quaternion form, and is rotated into:

$$qNq^{-1} = (0,1,0,0)$$

Now the complete orientation of the forearm controller has been described using two vectors, an orientation vector and a normal vector.

Further useful information can also be retrieved by combining the resulting vectors with the angular velocity, which is also estimated by the Kalman filter. In some of the virtual reality applications discussed later, it was desirable to generate the sensation of movement through a medium. This movement was emulated by allowing a vibration to appear on the side of the forearm that was leading in the movement; that is, if the thumb side of the device was leading the gesture, then a sensation would appear on that side. The computation had to be independent of the absolute direction of the movement.

This was accomplished by examining the flux through the forearm controller. First, two normal vectors were rotated to their new orientations using quaternion multiplication, one representing the surface of the PCB, and one representing the edge of the PCB. Then the flux through the PCB's surface and it's edge were computed by taking the dot product of the angular velocity ω with each normal vector.

$$N_1 = q(0,0,1,0)q^{-1}$$
$$N_2 = q(0,0,0,1)q^{-1}$$
$$flux_1 = N_1 \bullet \omega$$
$$flux_2 = N_2 \bullet \omega$$

The two results were compared, and the one with the larger magnitude was chosen; for example, if the angular velocity vector had a larger projection along the edge of the circuit board then along it's surface, a vibrating motor on the edge of the board would chosen to display this sensation. The sign of the flux dictates whether a motor on the thumb side of the board would be chosen, or one on the opposite side.

In some applications, a rolling gesture along the axis of the forearm was used as a trigger. A naïve way to detect this would be to look directly at the gyroscope along that axis, and allow the trigger to occur if that gyro value went above a threshold. This approach does work, but is problematic because in order to avoid false triggers, the threshold would have to be set high enough to avoid inaccuracies in the gyroscope bias, or Gaussian sensor noise, and the signal data would have to be low-pass filtered to avoid triggers due to noise spikes. The gesture would then have to be both sufficiently vigorous in order to exceed the threshold, and long in duration to allow the trigger to propagate through the low-pass filter.

A better approach is to take the dot product of the angular velocity ω with the orientation vector x, giving the projection of the angular velocity along the axis through the user's forearm. A metric was created using the ratio of this projection to the magnitude of the flux, as was described earlier:

$$\frac{x \bullet \omega}{\sqrt{flux_1^{\,2} + flux_2^2}}$$

Comparing the rolling angular velocity to the flux allows the trigger detection mechanism to filter out large sweeping gestures that include some rolling angular velocity, but other components as well. Using this metric allowed the trigger detection mechanism to be sensitive enough to respond to relatively gentle rolling gestures, while avoiding responding to other gestures. Obviously, division by zero has to be blocked in this case.

Finally, although the Kalman filter described in the next section only estimates orientation, one linear parameter was determined to be useful enough to be extracted. The motion is a quick, forward punching gesture along the forearm axis, similar to that of pressing a button. Such a movement is useful for virtual control panel applications, in which a user can orient the controller to point at a button on the surface of the environment sphere, and then push it with a quick linear movement. To extract this, the computer must determine that an accelerometer jumped, but that the orientation did not change. The Kalman filter is responsible for determining that, although the accelerometer

jumped, the magnetic field sensors and gyroscopes did not, and therefore the orientation should remain constant.

To accomplish this motion detection, accelerometer values due to orientation relative to gravity are calculated, and compared to the true accelerometer values in the form of an innovation similar to the complete measurement innovation that will be described in the following section. To calculate the error between the predicted accelerometer value and the true value, the vector corresponding to gravity, g=(0, 0, -1, 0), is rotated and subtracted from the accelerometer measurement vector a:

$$e = a - qgq^{-1}$$

Only the value corresponding to the error in the accelerometer aligned with the forearm axis is examined. If this measurement is far enough from zero, it must mean one of two things: either a linear acceleration along the forearm axis is in progress, or the arm is being swung with a large centripetal acceleration. To filter out this second possibility, the condition is set that the angular velocity must be below a threshold value. The resulting metric can be used to identify "button pushing" movements.

## 2.7 Kalman Filtering

There are many methods for fusing sensor data, but the most commonly used method, particularly for analyzing inertial sensors, is the Kalman filter. Originally published in 1960, the Kalman filter can intuitively be thought of as a way of combining a measurement with a prediction in the form of a weighted average. This allows the filter to reduce the amount of noise in the final estimates by comparing the noisy sensor measurements to a model of what is physically possible [ZM05], [BH97].

Kalman filters are recursive, which means that their estimate depends on the previous estimate, which has important consequences. For example, compare the

following two possible configurations of a set of inertial sensors, assuming that the controller board is at the equator, where the magnetic field vector is horizontal. In the first configuration, shown in Figure 2-14, the board is horizontal and pointing north; in the second configuration, it is still pointing north, but has been rotated 180° about the magnetic field vector's axis, and is accelerating upwards linearly at 2g, but has not yet started to move, shown in Figure 2-15.

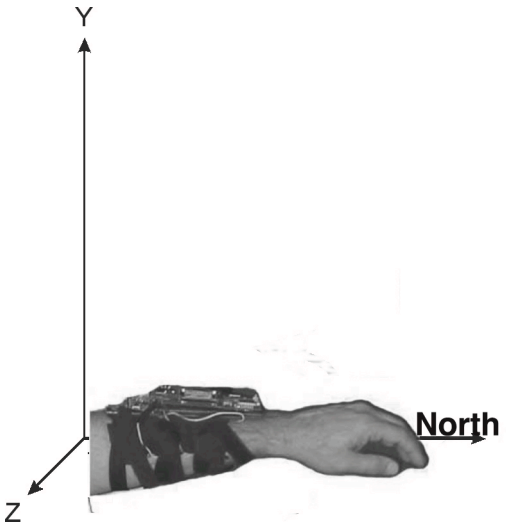



**Figure 2-14: Tricky configuration #1**

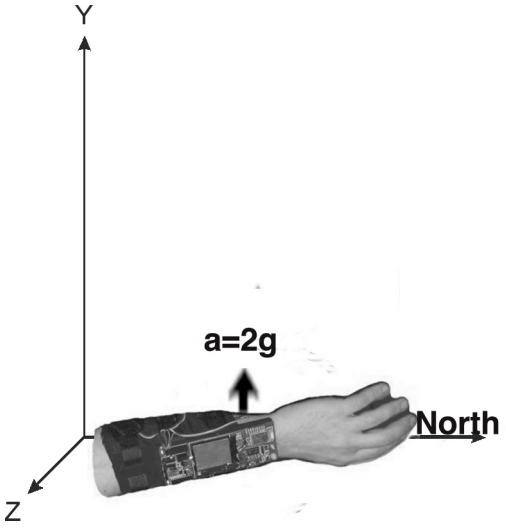



**Figure 2-15: Tricky configuration #2**

Surprisingly, the two configurations have identical sensor values. Since neither configuration involves an angular velocity, the gyroscopes read zero. As their orientations differ by a rotation about the magnetic field vector's axis, the magnetic field sensors will read identical values; that is, the sensor pointing north would read one, assuming the magnetic field vector to be normalized, and the other two sensors would read zero in both cases, being perpendicular to the vector being measured.

The X and Z axis accelerometers are oriented horizontally in both configurations, and read zero. Finally, the Y accelerometer reads g in the first configuration, and $-g + 2g = g$ in the second configuration. This demonstrates in an unlikely but revealing scenario how a linear acceleration can, for a short time period, throw off a physical model that assumes that the accelerometers only deliver data related to orientation. A filter that simply examined the sensor values and made no reference to the previous state of the controller would have no way of differentiating between the two configurations. However, a Kalman filter would combine the measurements with knowledge of the previous state of the controller, Clearly the two paths that the sensors would have to take to enter one of these configurations would be quite different.

### 2.7.1 The Extended Kalman Filter

The original Kalman filter is the optimal solution for linear recursive estimation problems, but several techniques exist for linearizing a non-linear problem so that it can be filtered. The resulting filter is no longer optimal, but it can still be quite effective. The two methods that will be discussed are the Extended Kalman Filter and, in more depth, the Unscented Kalman Filter [WM01].

Extended Kalman Filters are designed to linearize the problem by examining a (usually first-order) linear approximation to the system itself, which can then be propagated analytically through the Kalman filter, using the assumption that all the probability densities are Gaussian. The linearization of the system dynamics can be fairly

difficult to implement, due to the necessary construction of Jacobians [ZM05], [BH97]. Some effective quaternion based Extended Kalman Filters have been constructed by [Dum99], [Mar01]; these filters are complementary filters, and make use of the fact that the normalization constraint of rotation quaternions is easier to linearize when angular differences are small. For example, [Dum99] uses a Gauss-Newton algorithm to estimate the orientation due to the accelerometers and magnetic field sensors. This is done outside the EKF; the Kalman Filter is used to estimate the error between this orientation and that given by the gyroscopes.

### 2.7.2 The Unscented Kalman Filter

A newer approach to this problem is to treat the Kalman filter as a particle filter. The Unscented Kalman Filter approximates the Gaussians themselves as sets of sigma points, each of which can be propagated through the filter individually. It typically has the same order of complexity as a first-order Extended Kalman Filter, but is accurate to a second-order. One advantage to this method is that no Jacobians need to be constructed, making the implementation simpler, as the state and measurement update functions can be non-linear [WM01], [MW01]. Previous quaternion based filters for navigation include those by [Kra03], [CM03], and [MJ04].

There seems to be some discussion as to whether the UKF is truly better than the EKF for all problems. In one example, researchers found that the UKF produced negligibly better results, but required much more computation; that is, although the two may of the same order of complexity, the difference in the required computation can be quite different in practice [Lav03]. However, another project showed that the UKF was noticeably more accurate than the EKF, and required slightly less computation [CM03]. As most criticisms of the UKF were with respect to its required computations, but not to its accuracy, the UKF was chosen for this project. As its final speed was great enough so that it was not the bottleneck in the total implementation speed, the EKF was not considered.

The UKF particle approximation can be illustrated as follows, for a 1-dimensional measurement, such as the position of an object along a line:

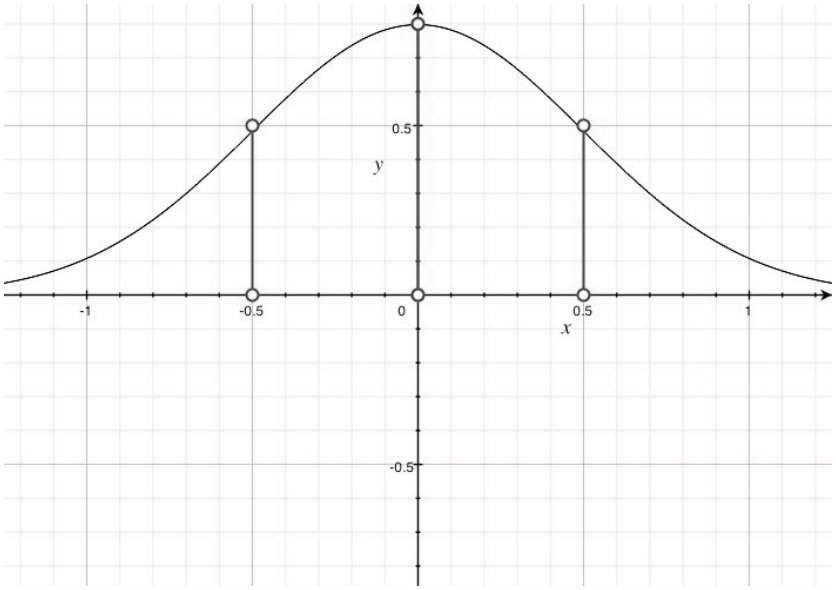



**Figure 2-16: Approximation of a 1-dimensional Gaussian**

To approximate information about the Gaussian, the three points shown are used to represent the mean and the covariance. For a 2-dimensional measurement, such as the X and Y positions of an object in a plane, more points must be used because the two dimensions of the measurement might not be known to the same accuracy, e.g. the X location of the object might be known more precisely than the Y location. The representation would be illustrated as follows:

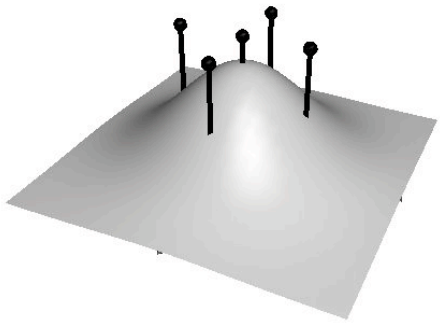


**Figure 2-17: Approximation of a 2-dimensional Gaussian**

Clearly, for an n-dimensional vector, 2n+1 sigma points must be used to approximate it using these methods. Various weighting methods can be chosen that give the mean more or less weight with respect to the other sigma points, and some implementations leave out the mean entirely [WM01].

A simplified algorithm for the UKF is as follows:

1. Approximate the current state using 2n+1 sigma points.

2. For each sigma point, use the state model to predict the next state. This step produces another set of 2n+1 points that should be centered around the mean of the predicted next state.

3. Find the mean and covariance of the points produced in step 2. This is the prediction of the next state.

4. Approximate the predicted state with a new set of 2n+1 sigma points.

5. For each sigma point, use the observation model to predict the sensor measurements that would be expected. This step produces another set of 2n+1 points that should be centered around the predicted measurements.

6. Find the mean and covariance of the points produced in step 5. This is the prediction of the sensor measurements.

7. Combine the covariance matrices of the predicted state and measurements, which produces the Kalman gain, a matrix that will be used to determine the relative weights of the measurements and the predicted state elements.

8. Subtract the predicted measurements from the true sensor measurements, producing a vector called the innovation, which should be close to zero if the model of the system is good. Combine the predicted state and the innovation vector in a weighted average, using the Kalman gain. This produces the final state estimation. This will be the state used in step 1 of the next iteration.

9. Calculate the covariance matrix of the final state estimation. This will be used to produce the sigma points in step 1 of the next iteration [WM01].

### 2.7.3 The Square-Root Unscented Kalman Filter

The version of the UKF used for this project was the Square Root UKF, which tends to be more efficient. This is because in the UKF, the Cholesky factor of the covariance matrix is used to produce the sigma points. For a matrix A, the Cholesky factor C is a triangular matrix that satisfies the equation:

$$A = C^T C$$

Finding C from A can be a time consuming calculation. The Square Root UKF uses the Cholesky factor all the way through, never actually using the complete covariance matrix. This ensures that Cholesky decomposition never has to be used, and it also makes the filter more stable in that the covariance matrix is more likely to remain positive definite [WM01].

The Square Root UKF starts with an initial state and Cholesky factor of the intial covariance matrix, as follows:

$$\dot{x}_0 = E[x_0]$$
$$S_0 = chol\{E[(x_0 - \dot{x}_0)(x_0 - \dot{x}_0)^T]\}$$

The rest of the algorithm (with a summary in English that is analogous to that given in the previous algorithm) is as follows [WM01]:

1) $\chi_{k-1} = [\hat{\mathbf{x}}_{k-1}, \hat{\mathbf{x}}_{k-1} + \gamma\mathbf{S}_k, \hat{\mathbf{x}}_{k-1} - \gamma\mathbf{S}_k]$

2) $\chi^*_{k|k-1} = \mathbf{F}[\chi_{k-1}]$

3) $\hat{\mathbf{x}}^-_k = \sum_{i=0}^{2L} W_i^{(m)} \chi^*_{i,k|k-1}$

$S^-_k = qr\{[\sqrt{w_1^{(c)}}(\chi^*_{1:2L,k|k-1} - \hat{x}^-_k), \sqrt{\mathbf{R}^v}]\}$

$S^-_k = cholupdate\{S^-_k, \chi^*_{0,k} - \hat{x}^-_k, W_0^{(c)}\}$

4) $\chi_{k|k-1} = [\hat{\mathbf{x}}^-_k, \hat{\mathbf{x}}^-_k + \gamma\mathbf{S}^-_k, \hat{\mathbf{x}}^-_k - \gamma\mathbf{S}^-_k]$

5) $y_{k|k-1} = \mathbf{H}[\chi_{k|k-1}]$

6) $\hat{\mathbf{y}}^-_k = \sum_{i=0}^{2L} W_i^{(m)} y_{i,k|k-1}$

$S_{\tilde{y}_k} = qr\{[\sqrt{w_1^{(c)}}(y_{1:2L,k} - \hat{\mathbf{y}}_k), \sqrt{\mathbf{R}^n_k}]\}$

$\mathbf{S}_{\tilde{y}_k} = cholupdate\{\mathbf{S}_{\tilde{y}_k}, y_{0,k} - \hat{\mathbf{y}}_k, W_0^{(c)}\}$

7) $$\mathbf{P}_{\mathbf{x}_k\mathbf{y}_k} = \sum_{i=0}^{2L} W_i^{(c)}[\chi_{i,k|k-1} - \hat{\mathbf{x}}_k^-][y_{i,k|k-1} - \hat{\mathbf{y}}_k^-]^T$$

$$\kappa_k = (\mathbf{P}_{\mathbf{x}_k\mathbf{y}_k} / \mathbf{S}_{\tilde{\mathbf{y}}_k}^T) / \mathbf{S}_{\tilde{\mathbf{y}}_k}$$

8) $$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \kappa_k(\mathbf{y}_k - \hat{\mathbf{y}}_k^-)$$

9) $$\mathbf{U} = \kappa_k \mathbf{S}_{\tilde{\mathbf{y}}_k}$$

$$\mathbf{S}_k = cholupdate\{\mathbf{S}_k^-, \mathbf{U}, -1\}$$

The constants in the above algorithm are given by:

$$W_0^{(m)} = \lambda/(L+\lambda)$$
$$W_0^{(c)} = \lambda/(L+\lambda) + (1-\alpha^2+\beta)$$
$$W_{i,i\neq 0}^{(m)} = W_{i,i\neq 0}^{(c)} = \frac{1}{2(L+\lambda)}$$
$$\gamma = \sqrt{(L+\lambda}$$
$$\lambda = \alpha^2(L+\kappa) - L$$

The constant parameter $\alpha$ is set to a small number, typically in the range $1 \geq \alpha \geq 10^{-4}$, and $\beta$ is set to 2 for Gaussian distributions, L is the length of the state vector.

Some details and notational issues in the above filter should be explained. When generating sigma points, as in steps 1, the result is a matrix in which each column vector represents a sigma point that is a complete n-dimensional state vector. The first of these is the mean state vector, and the next n are located a positive distance from the mean in directions given by the covariance matrix; the final n vectors are located a negative distance from the mean. The same sort of matrix is constructed in step 4.

When passing this matrix through the state update model in step 2, the matrix is broken up into its individual vectors, and for each of these a predicted state vector is calculated; the final set of predicted vectors are then assembled into another matrix. Interestingly, this sort of calculation can be quite intense for large state vectors, but lends itself well to parallel computation. [WM02].

In step 3, a new mean state vector and associated covariance matrix Cholesky factor is calculated. The operation given for the mean is a simple weighted average of the individual column vectors produced in the previous step. However, simply summing the vectors has consequences when using quaternions, so this step had to be altered. This alteration will be explained in the next section.

Updating the Cholesky factor requires some interesting linear algebra. A standard UKF filter would calculate the true covariance matrix from the sigma points in the usual way, and then use Cholesky decomposition to find the Cholesky factor. A faster method uses a combination of QR decomposition and Cholesky factor updating or downdating to update the Cholesky factor without ever actually computing the covariance matrix.

Cholesky factor updating and downdating seem to be unusual enough that they didn't make the cut when the LINPACK FORTRAN routines were ported to the LAPACK routines that are commonly packaged with scientific libraries, such as GNU GSL, which was used in the code given in the Appendix. As a result, a custom C port of the LINPACK functions DCHUD and DCHDD were included with the code. Note that downdating is risky because it can fail if the covariance matrix is not positive definite. This happened frequently enough during prototyping to be worrying, but once a final stable filter had been settled on, this never happened again in practice.

Finally, the matrix right divide operation required in step 7 can be implemented with a back substitution algorithm, because the Cholesky factor matrices are triangular. The code in the Appendix uses the LAPACK routine called DTRSM to solve this efficiently.

### 2.7.4 Finding the Mean Quaternion

Finding the mean of an angular quantity can be surprisingly tricky. In Fig. 2-17A, the angles 1° and 359° are averaged to give the incorrect result of $\frac{1° + 359°}{2} = 180°$; clearly, the correct answer is 0°. Using 2-dimensional vectors to represent this problem gives a better result, as can be seen in Fig. 2-17B.

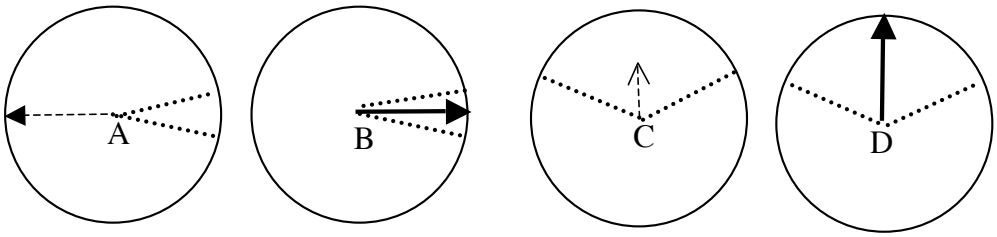



**Figure 2-18: Averaging with vectors and angles**

However, one problem with using vectors to represent angles is that the result of averaging normalized vectors doesn't necessarily yield a normalized vector, and rotation quaternions must always be normalized. This problem becomes painfully obvious in the next example, Fig 2-17C. Here, using angles produces the correct answer, $\frac{10° + 170°}{2} = 90°$, as can be seen in Fig. 2-17D. Averaging vectors here produces a new vector that points in the right direction, but doesn't represent a valid rotation. A naïve solution is to simply normalize the result; however, in practice this only works when averaging rotations will small angular differences between them.

The same problem occurs when averaging quaternions; simply averaging each of the four numbers individually does not produce a normalized quaternion, and normalizing the result by brute force is only accurate when the quaternions being averaged are relatively similar to each other. One method, used by [Kra03], is to switch back and forth from a quaternion representation to an error angle representation, where the error angle is given by a unit vector representing the axis of rotation, scaled up to a magnitude representing the angle of rotation.

These error angles fail when the angle is large, due to periodicity problems, but are accurate for small errors, and can be averaged in the usual way, due to the fact that the three elements are truly independent. This representation can be combined with the quaternion representation to produce a reliable average using a gradient descent convergence algorithm. This technique involves 7 steps:

1. Start with a guess for the mean quaternion. This initial guess can be anything, but the algorithm will converge faster if the guess is close. In this filter, the starting guess is the previously computed estimate of the mean.

2. Remove this guess from each of the quaternion sigma points being averaged. Intuitively, a set of rotations with an estimated mean of q are being rotated so that they have a zero mean. This is done by:

$$q_e = q_i q_m^{-1}$$

3. Convert the quaternion sigma points into error angle vectors, as follows:

$$\theta = 2\arccos(w_i)$$
$$x_{axis} = \frac{x_i}{\sin\frac{\theta}{2}}$$
$$y_{axis} = \frac{y_i}{\sin\frac{\theta}{2}}$$
$$z_{axis} = \frac{z_i}{\sin\frac{\theta}{2}}$$
$$v_e = \theta(x_{axis}, y_{axis}, z_{axis})$$

4. Average the error angles, by summing and dividing by the number of points.

5. Convert the mean error angle vector into an error quaternion:

$$\theta_m = |v_m|$$
$$q_e = (\cos\frac{\theta}{2}, \frac{v_m}{|v_m|}\sin\frac{\theta}{2})$$

6. Adjust the mean with the error quaternion, generating a better estimate of the mean:

$$q_m = q_m q_e^{-1}$$

7. Loop back to step two

Although there are quadratic convergence algorithms that may converge faster than gradient descent in general, [Kra03] demonstrated that, for finding a mean quaternion, gradient descent converges to an acceptable value in four iterations.

### 2.7.5 Additive Gaussian Noise and Quaternions

The form of the UKF used in this thesis assumes that all noises introduced are Gaussian. However, adding Gaussian noise to a rotation quaternion will not, typically, produce another rotation quaternion, due to the normalization constraint. Simply ignoring this inconsistency and normalizing the final quaternion can produce acceptable results if the noises are small, but a better solution is to use the error angle representation when dealing with covariance matrices; this has the advantage that it allows additive noise to be used, and in addition, it reduces the size of the covariance matrices that are used in the filter.

This doesn't completely eliminate problems due to using large amounts of Gaussian noise, as the error angle representation can exhibit periodicity when the noise levels approach 360°; but it does allow for more flexibility than adding Gaussian noise to a normalized quaternion.

Finally, it should be noted that there are forms of the UKF that use non-additive noise, but these require expanding the state vector to includ the measurement and process noise, and are therefore less desirable due to the increased computational complexity [WM02].

### 2.7.6 The State Update Function

The state vector used in this thesis is (q, ω, α), where q is a quaternion orientation, and ω and α are 3-dimensional vectors estimating angular velocity and angular acceleration, respectively. The angular acceleration is updated first, as follows:

$$\alpha' = \alpha$$

Note that this effectively estimates the angular acceleration as a random walk, not as a constant. Although the vector is not altered in the update function, it will later have noise added to it, so the complete model is given by:

$$\alpha' = \alpha + \nu$$

where ν is zero mean Gaussian noise. This noise component is not actually added in the state update function itself.

The angular velocity is updated next:

$$\omega' = \omega + \alpha t$$

Finally, the quaternion orientation is updated by first combining the angular velocity and angular acceleration to produce an error angle perturbation:

$$e = \omega t + \frac{\alpha t^2}{2}$$

Then the previous quaternion is updated by multiplying it with an error quaternion derived from the error angle [Kra01]:

$$\theta = |e|$$
$$q_e = (\cos\frac{\theta}{2}, \frac{e}{|e|}\sin\frac{\theta}{2})$$
$$q' = qq_e$$

### 2.7.7 The Observation Estimate Update

In this step, the SRUKF uses the predicted state vector to estimate a predicted observation, which will later be compared with sensor values. The accelerometer values are predicted by rotating a vector representing the accelerometer values that would occur if the controller were in its initial condition. This condition occurs when the forearm is pointing in the negative Z direction with the PCB on the top, pointing in the positive Y direction. The accelerometer values would then be (0, 1, 0), or (0, 0, 1, 0) in quaternion form. To produce the new accelerometer values, these are rotated as follows:

$$a = q(0,0,1,0)q^{-1}$$

This only estimates the acceleration due to gravity; angular accelerations, linear accelerations, and centripetal accelerations are ignored in this model, and hopefully filtered out by the SRUKF.

The magnetic field sensor values are then predicted as follows:

$$m = qhq^{-1}$$

The quaternion h represents a "constant" magnetic field vector in quaternion form which, in reality, may not remain constant, and must be tracked as well, as will be described in the next section.

Finally, the predicted gyroscope values are calculated by rotating the angular velocity vector in quaternion form:

$$g = q\omega q^{-1}$$

### 2.6.8 Low-Pass Filter Tracking

Certain values that are ideally constant, such as the bias of the gyroscopes and the direction and magnitude of the magnetic field vector, are in fact not as constant as one would like. The gyroscope values drift over time, and drift quite a lot in the first minute or so of use when the gyroscope has been cold for a while. This can cause a huge error in practice, because incorrectly estimating the angular velocity causes an angular position error that increases over time. Estimating the gyro bias can, in some cases, be as elaborate as estimating the orientation itself, due to the long period of time for which some trackers need to be autonomous (e.g. autonomous underwater vehicles) [HAM98]. For this thesis, it is assumed that tracking orientation for several days without any external communication would be an unlikely use of the forearm controller.

To correct for this, the angular velocity is assumed to be zero when averaged over a long enough time period, a reasonable assumption for human motion, in which angular movements tend to be transient. A low-pass filter or running average with a bin of 500 data points, or roughly 2.5 seconds, was used to track the gyroscope bias, which produced acceptable results.

The magnetic field vector didn't drift in the same way, but still had to be estimated each time the controller was used, because it could be different in different

rooms. As the exact direction of north was not needed in any of the applications used, the dip angle was low-pass filtered instead, after being calculated from the accelerometer vector a and the magnetic field sensor vector m:

$$\theta_d = \frac{a \bullet m}{|a||m|}$$

This equation gives incorrect results when the device has an acceleration that is not due to gravity. A simple way to filter this out that works often enough to be useful is to simply ignore dip angle calculations that occur when the magnitude of the accelerometer vector is far from one g.

### 2.7.8 Evaluating the Filter

Two data sets are presented in order to evaluate the capabilities of the Square Root UKF described previously. Figure 2-19 shows the filter's output with the forearm controller being held stationary, as compared to the raw, integrated gyroscope data and the raw accelerometer data. All sets are normalized so that they would show a value of zero with the PCB horizontal, and a value of one with the PCB vertical and pointing upwards. The gyroscope data seems smooth, but wanders off due to the integration of the noise within it; the bias is not being tracked here, but is being fixed at whatever its average value was during the last calibration test. The 1600 data points represent a total time of about 8 seconds. The accelerometer data is centered at the correct value, but is very noisy. The filter output can be seen at the bottom, reducing the noise considerably,

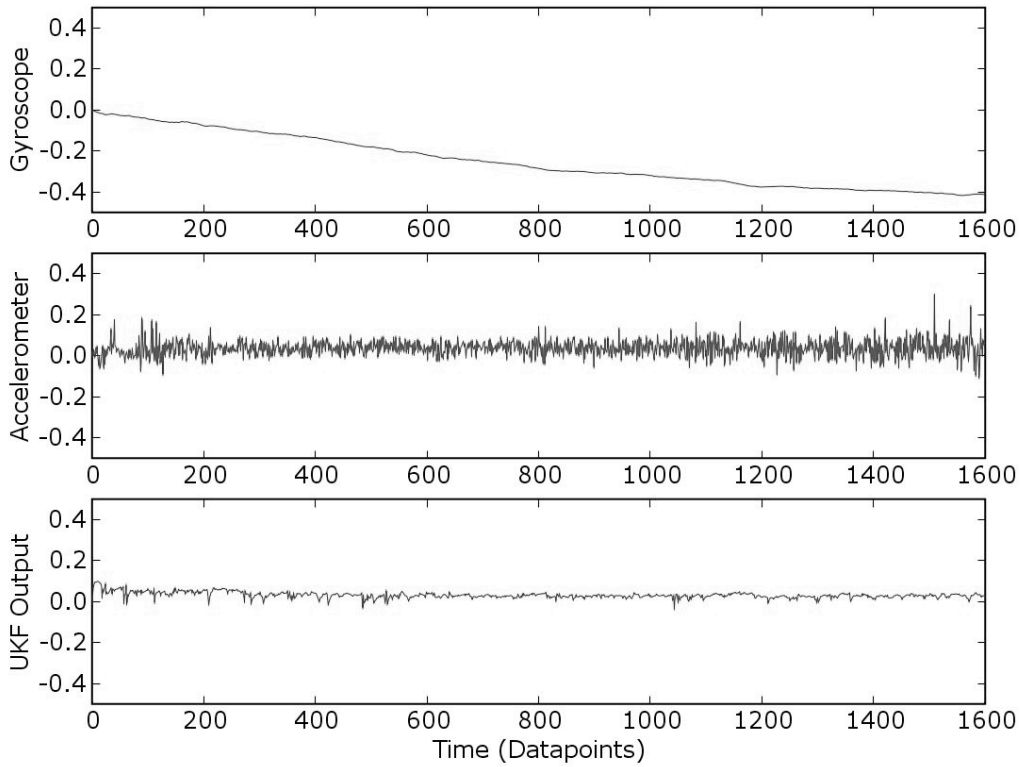



**Figure 2-19: SRUKF output with a stationary controller**

For the second test, the author repeatedly swung the arm wearing the controller from horizontal to vertical in a pitching movement that was executed as rapidly as was possible without risking damage to the controller board or the author. The results are shown in Figure 2-20. The gyroscope data has the correct shape to some approximation, but wanders off once again due to the noise integration. It also seems to overshoot in one or both directions; this is probably due to the scale factor of the gyroscope no longer being linear when very high angular velocities are applied.

The accelerometer data, once again, settles around the correct endpoints of the rotation; but here we can see large transients due to the angular accelerations applied to the PCB. The negative spikes cannot be seen in their entirety, but they descended to about -4g, which corresponds to -4 in this normalized graph. Even in these extreme conditions, where the inertial sensors are being driven passed their ratings and out of their linear ranges, the SRUKF somehow seems to acquire enough correlated data to assemble

a square wave that varies from zero to one. Note that this experiment was performed by a human, and not a robotic test jig, and some of the corresponding error is due to this. For example, around the 1000th data point, all three datasets drop below zero; this is most likely due to the author not precisely arriving at a horizontal orientation.

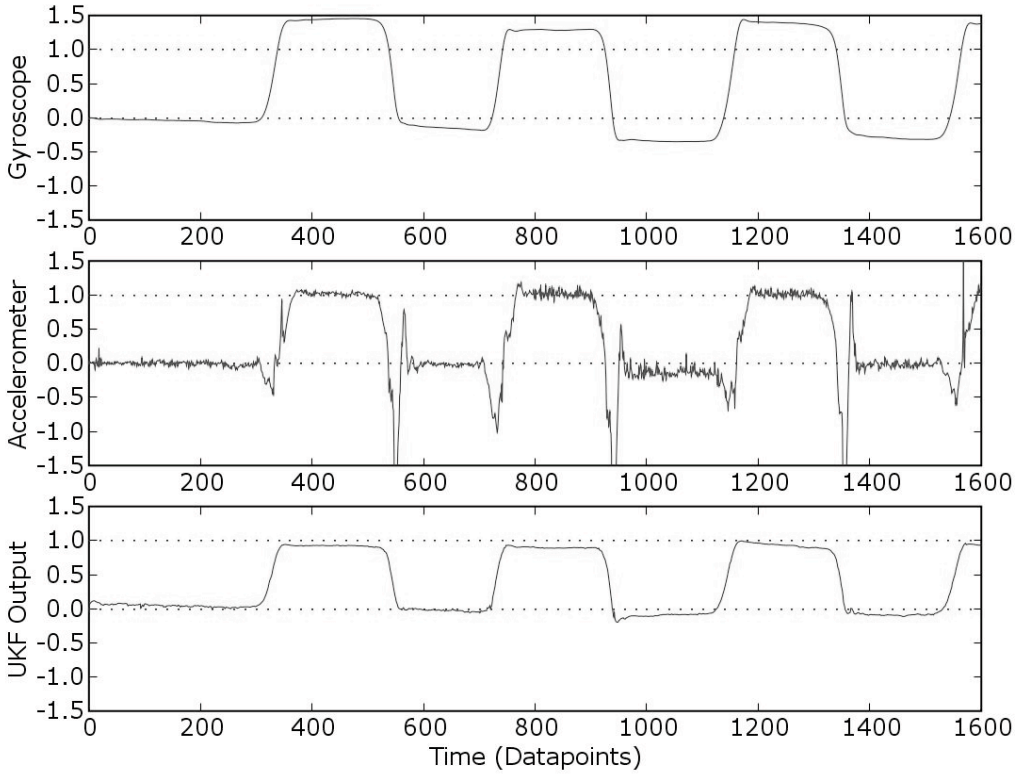



**Figure 2-20: SRUKF output with a rapidly moving controller**

# 3 Artistic Applications: Wearable Virtual Instruments

A set of virtual instruments was created using ARMadillo as the hardware platform. These instruments were implemented using a UDP socket interface between Python/C++ code that tracked the forearm and returned tactile feedback, and a Max/MSP program that provided the real-time audio processing. The instruments created were intended to be simple and to highlight certain features made possible by using tactile feedback. The idea is not to think of the forearm controller as an instrument that is being played, but as an augmentation to an open-air controller such as the Theremin.

Using such a device rather than a true open-air controller allows the user to feel the virtual environment; using the forearm instead of a more traditional glove should make it easier for a user to ignore the existence of the device completely, and think in terms of an environment in which the user can feel invisible objects and play them. With the hands free, there is also the potential for a user to play a real, physical instrument, while playing a virtual instrument at the same time. As the inertial sensing system used can only measure orientation, all virtual environments can be visualized as existing on the surface of a sphere surrounding the user, as can be seen in Figure 3-1.

**Figure 3-1: Virtual Environment sphere**

The environments created were:

1. A Passive Tactile Art Display
2. A Virtual Control Panel
3. A Virtual Crash Cymbal
4. A Fretted Theremin
5. A Gesture-Based Sound Design System

## 3.1 Summary

Each of these was designed to highlight specific attributes of the controller, and will be described in depth in the sections that follow. However, the objectives of each project are briefly summarized below:

1. A Passive Tactile Art Display: for this project, the inertial sensors are dormant, as no user control is required. The focus of the passive display is the vibrating motors themselves, which are used to display abstract tactile art on the user's forearm, rather than useful control feedback information they transmitted in other projects.

2. A Virtual Control Panel: in this section, a simple remote control device was hard-coded into the controller in 8051 assembler. This project was meant to demonstrate the possibilities of using the controller without an external computer doing all the processing; the remote control is truly independent, wireless and portable, unlike the other projects, which use a complex Kalman filter and Python scripting. This project is, in essence, a complete portable virtual reality environment that could be built into a wristwatch like wearable device. In theory it could control anything, using the BlueTooth module, but was chosen to control the selection of a set of audio samples.

3. A Virtual Crash Cymbal: this was the first complete VR environment created using the entire system, including the Kalman filter, OpenGL rendering, Python-based tactile feedback scripting, and Max/MSP audio processing. An invisible crash cymbal was created that could be felt, struck, dampened, and brushed by the user's arm.

4. A Fretted Theremin: this was designed to focus on the utility of tactile feedback, in adding frets to an instrument that has been traditional extremely hard to play due to its inherent intangibility. The frets allow a user to feel a note and then play it, or move quickly from one section of the instrument to another, while maintaining a sense of the physicality of the system.

5. A Gesture-Based Sound Design System: finally, the controller was interfaced to a sound design system as a collaborative project with Adam Boulanger. This system allowed a user to design sounds using gestures, by finding a desirable sound in space, and selecting it in order to "zoom in" on it and begin changing other parameters.

## 3.2 Delivering Interesting Vibrotactile Sensations

Before exploring the individual instruments, an overview should be made of the techniques used in creating tactile sensations with the vibrating motors. Generating an interesting sensation on a dense vibrotactile display is unexpectedly complicated, once physical features of the display are taken into account, such as the rotational inertia of the motors, the static friction holding a motor in place prior to vibration, the conduction of vibration from one part of the display to another, and the danger of overheating when vibrations last too long.

### 3.2.1 The Response of a Vibrating Motor to PWM

Pulse width modulation was used to create varying intensities of vibration in the display. For example, the first signal shown in Figure 3-2 generates a soft vibration, the second a more intense one, and the third a sort of vibratory crescendo.

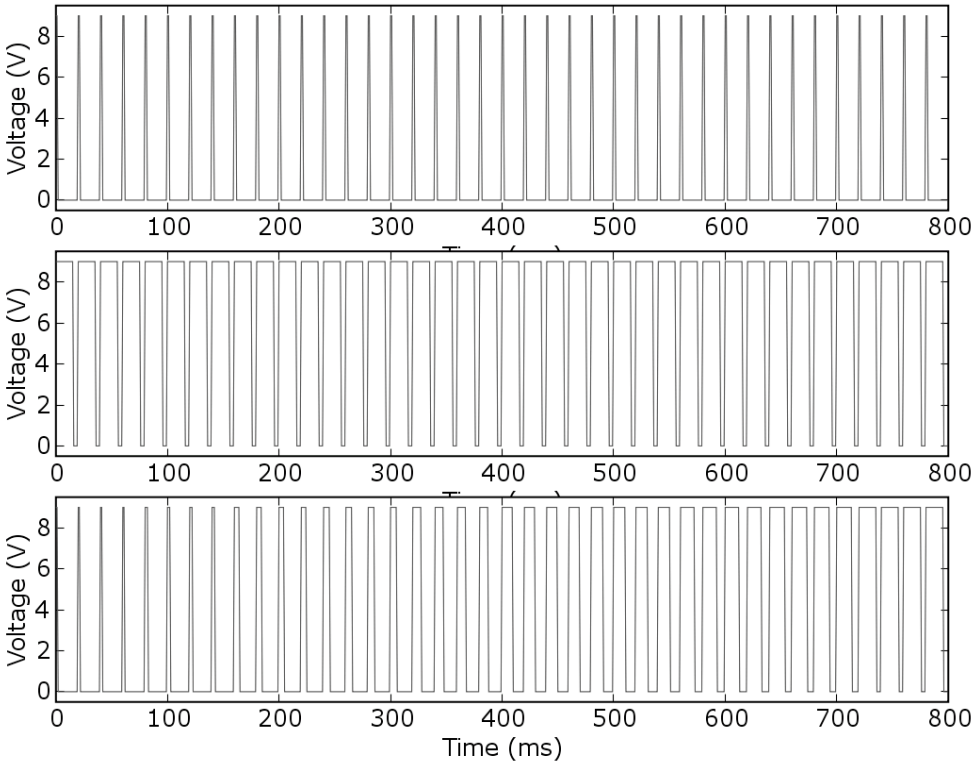



**Figure 3-2: Example PWM waveforms**

Choosing the pulse width required examining properties of the motor's rotational inertia. For example, a pulse width must be large enough to provide a torque that is sufficient for countering the static friction in the motor; that is, there is some pulse width that is so small that it will not start the motor turning. Once this static friction has been defeated, the motors will turn, but too slowly to generate a detectable vibration. Increasing the pulse width creates a vibrating sensation that is related in intensity to the pulse width. The motors are rated at 3V, but up to 9V is used in moderation in these applications. However, pushing the limits of the motors can result in overheating or mechanical damage to the solder joints connecting the motors to their wires.

In Table 3-1, the typical response of a vibrating motor to a voltage is given; it should be understood that these numbers change from motor to motor. The maximum voltage used is 9V, so a pulse width of 0.33, when low-pass filtered by the coil in the

motor, corresponds to the 3V maximum rating given by the manufacturer. These averaged voltages are given rather than the pulse-widths, but all voltages were generated by creating an appropriate PWM.

| ***Average Voltage*** | ***Effect*** |
|---|---|
| <0.1 | Not rotating. |
| 0.1 | Idling. Rotating, but not vibrating. |
| 0.3 | Soft vibration |
| >0.3 | Vibration increases with pulse width. |
| 3 | Rating of the motor. |
| 5 | Motor begins to heat up. |
| 7 | Motor begins to vibrate free of its connectors. |

**Table 3-1: Effects of PWM**

### 3.2.2 Jumpstarting a Vibrating Motor

It must be possible to start and stop the vibration suddenly in a VR environment in which the user might move quickly and touch a virtual object. For a low-latency but gentle vibration, simply generating a PWM signal corresponding to the desired final vibration intensity is insufficient, because the motors have some rotational inertia, and therefore there is a delay required to accelerate the motor to its final angular velocity.

There are essentially two methods for reducing this delay. The first is to allow the motor to turn at a lower, "idling" angular velocity prior to the time of vibration. This ensures that the static friction has already been defeated, because the motor is turning. In addition, it ensures that the acceleration required to reach the final desired angular velocity is smaller, because the velocity doesn't have to change as much.

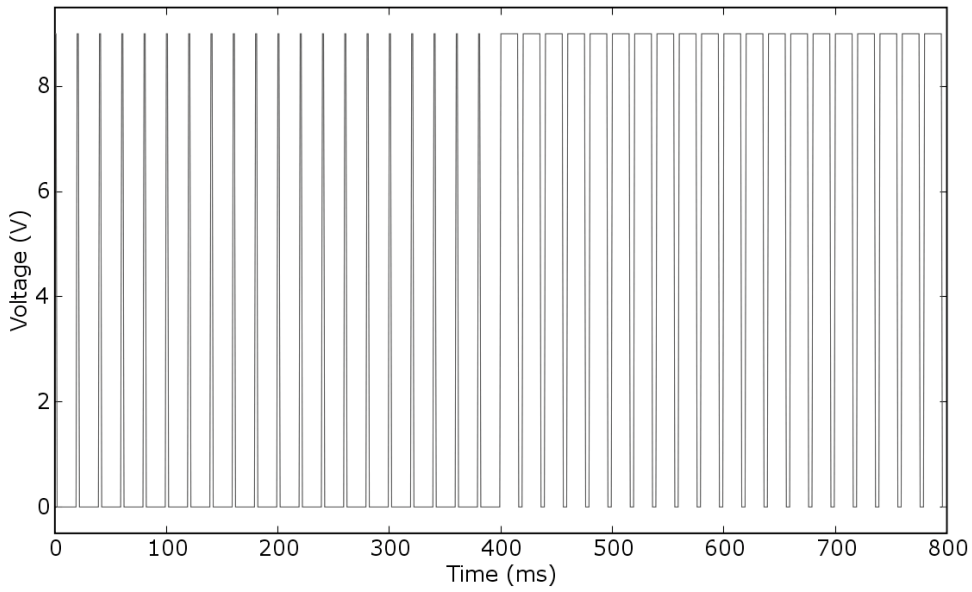



**Figure 3-3: Idling PWM**

This technique is shown in Figure 3-3, with an idling rotation in progress before 400ms, and a discernable vibration beginning at 400ms. The difference between the two pulse widths has been greatly exaggerated. The disadvantage of using the above system is that it requires dissipation of power when no vibration is occurring, and knowledge that a motor is going to vibrate soon, complicating the timing of the VR environment. The alternative to this complex timing is to allow all the motors to idle all the time, which dissipates even more unnecessary energy. Finally, as all the motors are slightly different, the small idling window will vary from one motor to the next. That is, a pulse width that causes one motor to turn without vibrating noticeably might not be powerful enough to turn a second motor at all, and might generate a small, noticeable vibration in a third motor.

An easier method is to create a short spike at the beginning of the signal, and then drop down to the desired final pulse width, as can be seen in Figure 3-4.

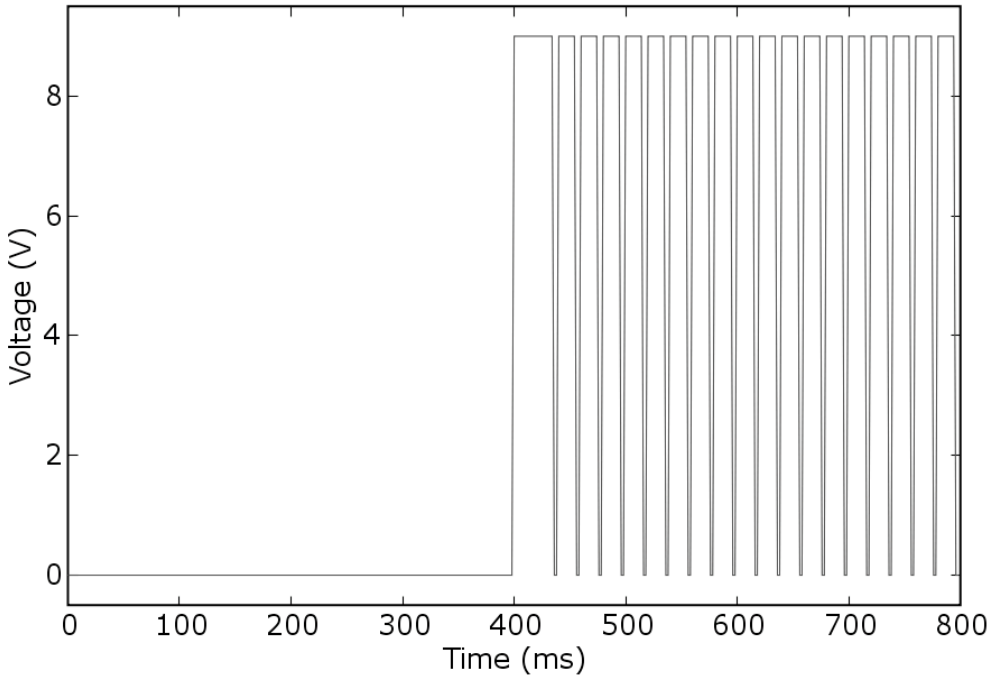



**Figure 3-4: Jumpstarting PWM**

The 9V spike at the beginning of the signal defeats the static friction in the motor, and accelerates the motor to its new angular velocity quickly; the signal then drops to its final pulse width and allows the motor to vibrate normally.

### 3.2.3 Avoiding Overheating

The spikes described in the previous section are highly effective and can be used for strong jolting effects in addition to simply being used to quickly start a less intense vibration. When given spikes, the motors react so strongly that it can feel as if the arm is being pushed, creating a temporary illusion of force feedback. However, driving the motors past their 3V rating for extended periods of time is dangerous for both the motors, which can vibrate free of their connecting wires, and for the user, who will experience

discomfort when the motors heat up within fabric that has been tightly bound to the forearm.

It is important to set time limits for all intense vibrations, so that the motors will never vibrate long enough to heat up. Only very soft vibrations can be used for extended periods of time, and even these should be given time limits.

### 3.2.4 Hysteresis, Unwanted Vibration Feedback, and Jitter

One problem that can easily occur when triggering a vibration with a gesture is feedback due to the accelerometers. This occurs because a new orientation produces a vibration which creates a transient in the accelerometer; this transient, if it is not removed by the Kalman filter, will be interpreted as a new orientation and can, in turn, generate another vibration. Such feedback, when not properly controller, was observed to generate vibration indefinitely, even when the controller was subsequently moved to an orientation in which no vibration should have occurred. This feedback was reduced, but not completely removed, when the motors directly under the PCB were not used.

A second type of disorienting effect can take place if a single threshold is determined for starting and stopping vibration. When the controller is positioned close to that threshold, the sensor noise will cause the computed orientation to drift repeatedly across the threshold, creating a jarring set of jittery vibrations that start and stop. This is simulated in Figure 3-5, with the line composed of crosses representing a noisy orientation estimate, and the solid line representing the resulting state of the system, based on the threshold crossings.

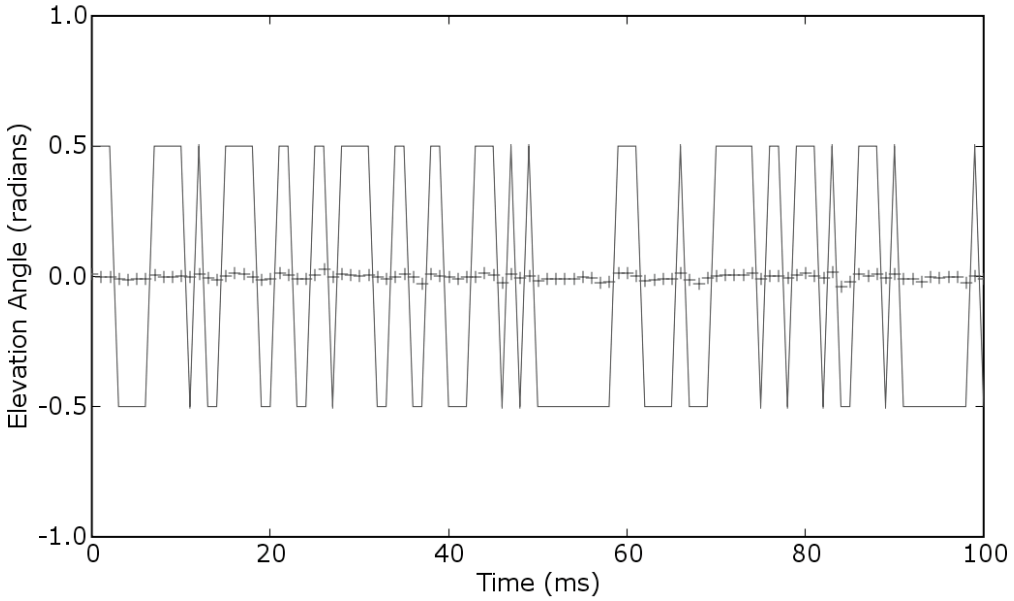



**Figure 3-5: Vibration jitter across a threshold**

One solution to the problems of feedback and jitter is to create two thresholds, one for entering a region and one for exiting a region. A vibrating motor might be active when the controller is in region A, but not when the controller is in region C. When the controller is in region B, in between regions A and C, the motor should be in a state of hysteresis, in which it is vibrating if it came from region A, but not vibrating if it came from region B. This is shown in Figure 3-6, in which the controller moves from one region toward another, and stops immediately upon crossing a threshold. The noise orientation estimate can be seen as a set of crosses, and the solid line is the state of the system, which changes only once, as is desired. This is made possible by the change of the threshold itself, indicated by the dotted line, which drops from 0.1 to -0.1 as the state changes.

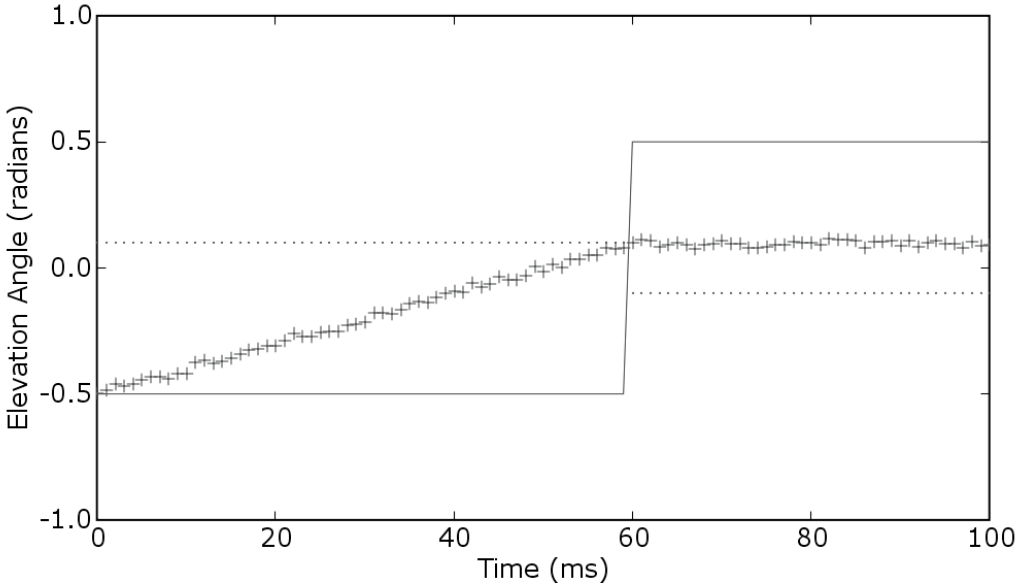



**Figure 3-6: Hysteresis defeating jitter**

When this method is used, if the hysteresis region is larger than the drift created by the sensor noise (taking into account the additional noise generated by the vibrating motors), there is no one place where the controller can be held such that the noise will cause the computed orientation to switch back and forth across two vibratory states, and the jolts that might otherwise create vibratory feedback will be less dangerous. As will be seen in later sections, every boundary created in every virtual environment in this thesis has some hysteresis region to provide stability.

For future research, it might be interesting to make a closer examination to the reaction an accelerometer has to a vibrating motor. The measurement noise matrix in the Kalman filter could be adjusted to take this into account, preventing the orientation estimate from reacting to the vibrating motors. In a sense, the vibration could be thought of as a control input, modeled as Gaussian noise. In addition, it might be possible to monitor the vibration with the accelerometers. The feed-forward methods for generating sensations suffer because each vibrating motor is slightly different; it might be possible to generate more reliable sensations by using a feedback method, where the vibration is increased until the accelerometer noise reaches a certain variance.

### 3.2.5 The Locations of the Motors

Although the forearm controller is capable of driving up to sixteen motors, the applications in this thesis used only twelve in preliminary development stages, and only nine in the more advanced applications. The final nine motors are arranged with three on the left side of the forearm, three on the underside, and three on the right side. Their locations are shown in the left half of Figure 3-7, with the locations of the other three motors shown in the right half of Figure 3-7.

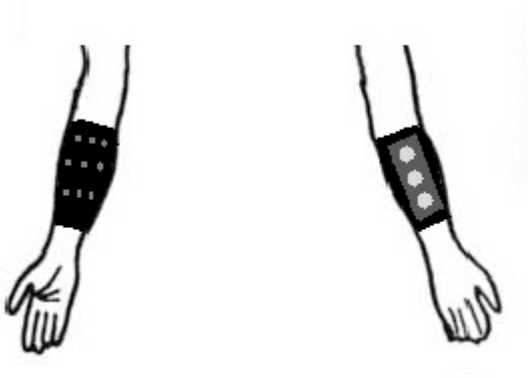


**Figure 3-7: Location of the motors on the forearm**

In the examples that follow, vibrational patterns will be depicted as in Figure 3-8.

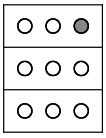


**Figure 3-8: Vibrating motor notation, example #1**

In such diagrams, it is always assumed that the palm is facing into the paper. The above diagram indicates that only the motor closest to the user's fifth finger is vibrating, if the device is being worn on the right hand.

For the examples that deal with a twelve motor setup, the three motors on the back of the forearm, directly under the PCB, are also used. These motors provided a powerful effect but were not clearly localized, as they tended to make the entire acrylic platform vibrate. They were used in combination with motors on the underside of the arm, to provide intensity to a sensation. To convey a sense of the vagueness of the sensation they provide, they are depicted as large shaded regions, as in Figure 3-9.

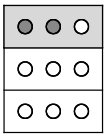


**Figure 3-9: Vibrating motor notation, example #2**

In this example, the shaded region at the top indicates that the motor on the back of the forearm, nearest to the hand, is vibrating, generating an intense, unlocalized sensation near the top of the controller. In addition, the two darkened circles indicate that two motors under the arm are vibrating, one at the center of the forearm, nearest to the hand, and one nearest to the thumb, assumbing the controller is on the right arm.

### 3.2.6 Localized Sensations and Textures

With a large and dense array containing many actuators, it is important to determine combinations of tactile sensations that feel meaningful, and constrain the vibratory output so that it stays within these types of patterns. There are certain combinations of vibration that might seem as though they could be perceived meaningfully by the user, but in practice they are less interesting. For example, take the seemingly logical sequence shown in Figure 3-10:

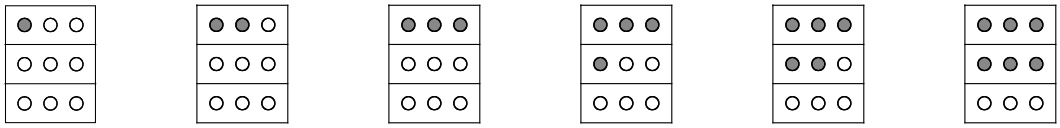


**Figure 3-10: Not so interesting sensation**

The intention of this pattern is clearly to create a sensation of increasing vibration and increasing stimulated area. However, this pattern would fail in its intention. The first event would be felt as a sharp, localized vibration, as it is intended. The second and third would also feel as they were intended, a decrease in localization, and an increase in vibrational intensity. However, the last three patterns would not be distinguishable as shapes, but only as vibration intensity. That is, it would be clear that the total vibration is increasing throughout the sequence, but by the fourth step, the entire forearm display would be shaking, and any sense of shape would have vanished.

More effective tactile sequences can be categorized as localized sensations and textures. The configurations shown in Figure 3-11 can be felt as localized vibrations.

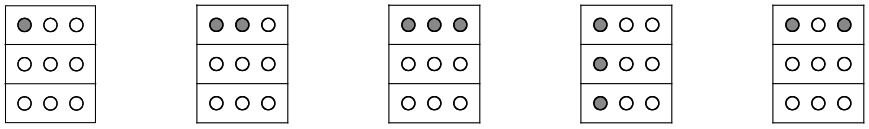


**Figure 3-11: Localized sensations**

The last configuration is interesting, because it is surprisingly clear as a set of two separate actuators, one on each side of the wrist. The clarity in this case is due to the fact that these motors are lying against anchor points at the wrist bones, where the vibration feels sharper and more localized [CCB01]. In contrast, the configuration in Figure 3-12 does not involve two motors at anchor points, and therefore does not feel like two distinct vibrating points.

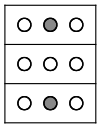


**Figure 3-12: Not a localized sensation**

When a sensation is not felt as a sharp, localized event, it may still be interesting. For example, allowing all the motors to vibrate gently with a rising and falling intensity creates a sort of breathing effect. Pulsing all the motors with a large transient creates a vibratory jolt that makes the user's arm feel as though it has been shoved. Two other interesting textural patterns are given as examples. In this case, it is assumed that a given row represents a pattern that repeats very quickly. Figure 3-13 represents a sort of chaotic tickling sensation.

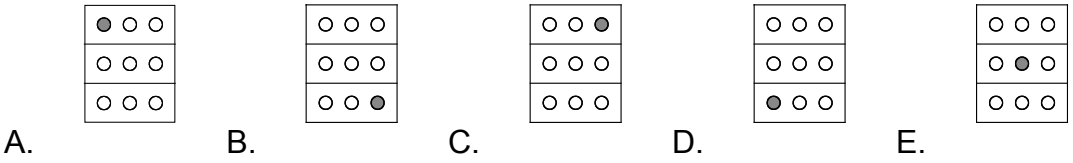



**Figure 3-13: Tickling pattern**

The series shown in Figure 3-14 feels like a massage.

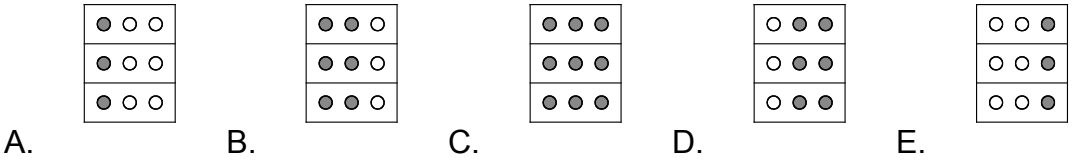



**Figure 3-14: Massaging pattern**

## 3.3 Passive Tactile Art

In the first musical application discussed, the user listens to music while experiencing a form of passive tactile art; just as the music occurs in time without control from the user, tactile sensations are displayed on the user's skin without control. In this application, ARMadillo is used as a tactile display, but not as a forearm controller. The application was designed to demonstrate the artistic capabilities of the vibrating motors, using them to stimulate emotion rather than to inform the user of some tactile environment. This project draws inspiration from work in the field of tactile art by Eric Gunther [Gun01], which uses a full body suit containing a less dense array of vibrating motors. The work described here attempts to expose similar possibilities when using a dense array of actuators.

Two musical fragments were played, each with a vibrotactile choreography that was designed to correspond to the piece in some satisfying way. This choreography was was created by hand and hard-coded the first piece, a logical mapping is made in which there are clear relationships between specific vibrating motors and certain notes; In the second piece, a textural approach was taken.

### 3.3.1 Stravinsky: The Rite of Spring

This fragment was taken from the bassoon solo at the beginning of Igor Stravinsky's "The Rite of Spring" [Str21]. The fragment, shown in Figure 3-15, is tonal and easily understandable as a set of notes comprising a melody, which will not be the case for the example discussed in the next section.

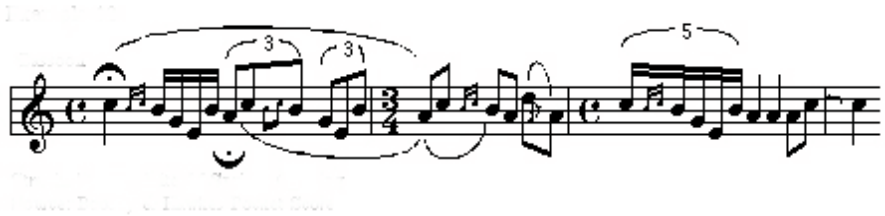


**Figure 3-15: Opening passage of "The Rite of Spring"**

A mapping was chosen for this fragment, such that each note corresponded to a vibrating motor, though some motors had more than one note associated with them. The melody is simple and modal; it can be summarized as an arpeggio containing C-B-G-E with an ornament on the C and B, The two subsequent notes A and D serve respectively as a sort of tonic or resolution point, and the climactic peak of the melody.

The chosen mapping is given in Figure 3-16.

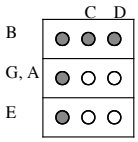



**Figure 3-16: Tactile mapping for the Stravinsky passage**

There were a few factors that motivated this mapping. As it had been decided that only one motor would be on at a time, and would represent a pitch, it was desirable to generate sensations that were as localized as possible. Such sensations are easiest to produce along points located near bone; therefore only motors that were in such positions were used. As the notes in the piece sound as if they are moving in clear directions, it was important to generate an analogous direction within the vibration. However, a three-by-three array does not lend itself easily to long, extended movements in a single direction. This linearity was faked by creating a movement that started at the hand, moved along the wrist, and then moved down the forearm bone. Although this movement is not strictly in a single direction, it was conceptually linear enough to provide the user with the sensation of a clear movement in some direction. The location of the motors also made

use of the wrist and forearm bones as anchor points [CCB01], making it easier to determine the locations of the motors and the direction of the sensation.

In addition to mapping pitches to motors, the amplitude of the musical fragment was translated directly to PWM, so that the vibrating motors increase or decrease in intensity along with the music. For example, the long note F beginning "The Rite of Spring" is played with a crescendo. In sync with this, the corresponding vibrating motor would be driven with a very small pulse width that gradually grew in magnitude, generating a crescendo in vibration that coincided with the audio crescendo.

### 3.3.2 Xenakis: Polytope de Cluny

A second musical example, "Polytope de Cluny" by Iannis Xenakis [Xen72], was chosen as a testbed for textural patterns. Rather than associate specific notes or sounds with specific vibrating motors and locations on the arm, the vibrotactile augmentation to the Xenakis was composed of more complex, abstract sensations derived from combinations of motors.

The fragment that was used contained two sections. The first involved a deep, low pitched extended noise, sounding like wind blown directly across a microphone. This sound texture was matched with a sort of gentle breathing effect in the vibrating motors, increasing and decreasing the intensity of all of them together. Another effect that was derived for this section is depicted in Figure 3-17.

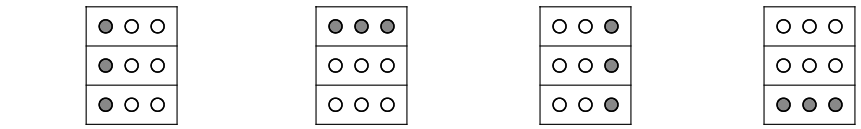



**Figure 3-17: Xenakis twisting sensation**

Each configuration displayed above was faded in slowly using PWM, and faded out as the next configuration was faded in. This was felt as a strange twisting sensation.

The second section of the Xenakis fragment sounded as though it had been derived from the high-pitched noises generated by dropping shards of broken glass. As the piece itself moves from a static, deep sound to a fast moving, high pitched sound, the vibrations chosen did likewise. The pattern shown in Figure 3-18 was created for this section; each labeled event lasts about 250ms.

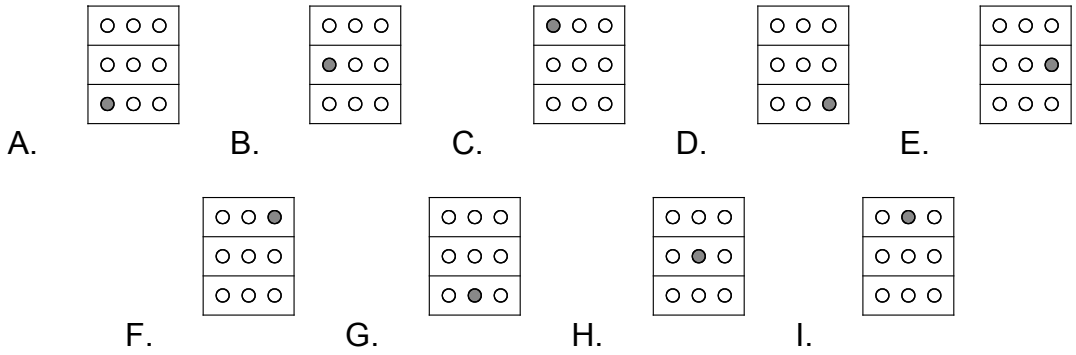



**Figure 3-18: Xenakis, "broken glass" sensation**

The above pattern is displayed twice, once as is shown, and once as its mirror image; then, the entire sequence repeats. The pattern was intended to convey the sense of movement along the forearm, toward the wrist. Interestingly, one more step is required for conveying a sense of motion; without it, the pattern in Figure 3-18 simply feels like a repeating sequence of abstract vibration, but not a movement. This lack of clarity was also noticed by [PJ05], in which a similar rising pattern was recognizable at a meager rate of %80 by users that were tested. To generate a sense of movement, there must be a sense of a starting point and an arrival point; this is accomplished simply by allowing the configurations involving a motor nearest to the wrist to last longer than the other configurations, and to have a PWM that drives the motor with a slightly larger amplitude. The illusion of something starting near the elbow and flowing up toward the wrist is then complete, This demonstrates one of the drawbacks of using an overly simple tactile pattern as a benchmark for determining a user's ability to distinguish patterns.

As the musical fragment becomes more energetic and louder, the sensation is altered gradually to convey this. Configurations A, B, and C in Figure 3-18 were transformed into the configuration shown in Figure 3-19.

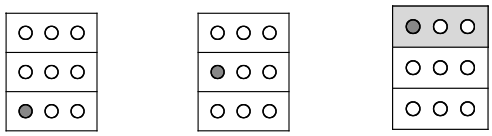


**Figure 3-19: Adding a tactile accent**

## 3.4 A Virtual Control Panel

The purpose of this project was to test the forearm controller's potential as an independent device that made no use of external processing for arm tracking or tactile feedback generation. A simpler method of using the sensors had to be used, as a quaternion-based Unscented Kalman Filter would be difficult to implement on an 8051. The objective was to create an environment in which a user could navigate to an object and select it, but to keep the code simple enough so that the entire system could be hard-coded into the controller, with the exception of the system being controlled. In this case, the latter was a set of sound samples that were played in a Max/MSP patch, which received instructions from the forearm controller via the BlueTooth module. The software diagram is shown in Figure 3-20.

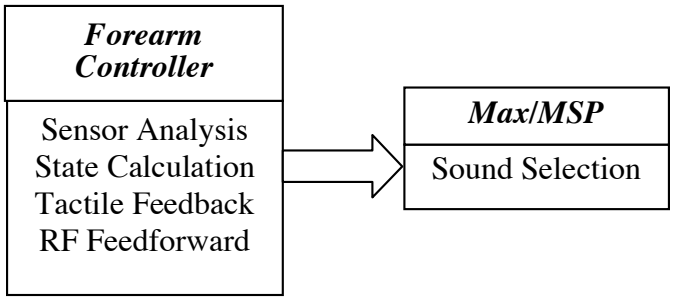



**Figure 3-20: Sound selector/Remote control software diagram**

This project can be described as a virtual control panel with 18 knobs, arranged as in Figure 3-21. The "knobs" can be felt within the environment, even when they are not being triggered.

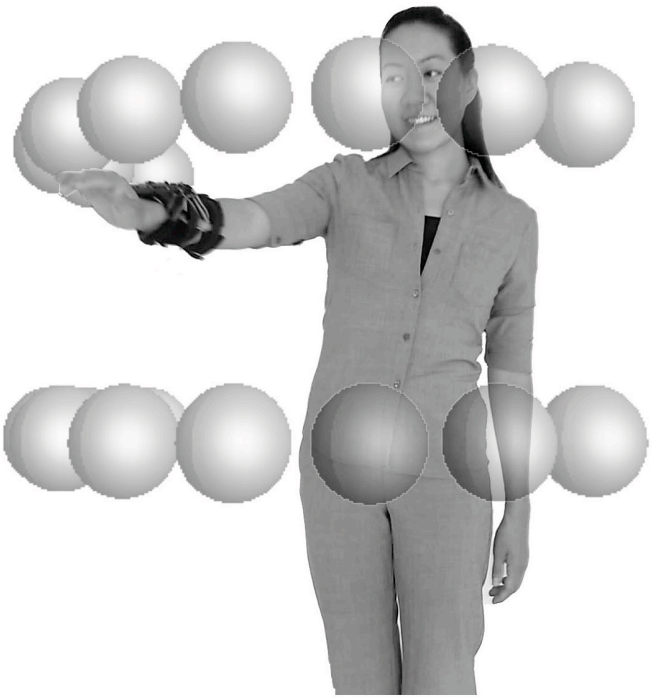


**Figure 3-21: The Control Panel virtual environment**

The sensor filtering had to be relatively simple in this application, because it was being processed directly in the 8051. An assumption was made that the user's palm was facing down; this allowed a single gyroscope to be used to determine the change in horizontal angular position, and a single accelerometer to determine the vertical angular position. A second gyroscope detected rotations along the axis of the forearm which were used to select objects. The three sensors were all low-passed filtered.

To avoid drift due to gyroscope noise, the system was designed only to respond when the gyroscope measurements passed positive or negative thresholds that indicated that the device was moving. Although this stabilized the system against gyroscope drift, it meant that a user could move the controller slowly enough that the system wouldn't respond. Therefore, all the horizontal angular positions in the system were relative. The system was designed to imitate a telephone keypad with numbers from 1-9, corresponding to the array of vibrating motors. To select an object, the user would first

"dial" to a certain number with a horizontal angular movement, and then selected it with a quick rolling movement about the long axis of the forearm. The dialing gesture and its accompanying tactile feedback are shown in Figure 3-22, depicted as they would be seen from above.

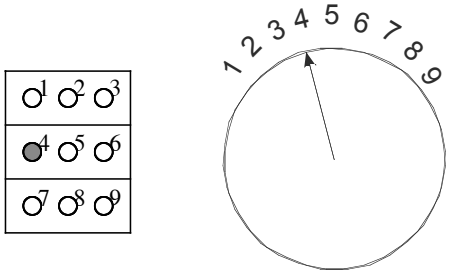



**Figure 3-22: Dialing gesture**

Two sets of nine objects were positioned, one 30° above the horizontal plane and one 30° below it. When the controller was horizontal, the set of objects being manipulated were the set previously chosen; that is, the angular section within 60° of the horizontal plane was designated as a hysteresis area to prevent unwanted vibratory feedback. The accelerometer along the long axis of the forearm was used to determine which set was being selected from, and a transient vibratory jolt was used to indicate that the level had been switched.

In addition, the system was designed such that objects in the upper level feel different from objects in the lower level, a necessary piece of information, given that a user with a horizontally oriented arm could be controlling either level, depending on the last selected state. This feedback was implemented by having the motor corresponding to the chosen object pulse instead of simply vibrating continuously. The type of pulse depended on the level chosen. If the upper set had been chosen, the pulse would "fade in" repeatedly, using pulse width modulation; if the lower set had been chosen, it would "fade out", as can be seen in Figure 3-23. The diagram shows two pulses for each example; the top graph shows the fading out pattern, the bottom graph shows the fading in pattern:

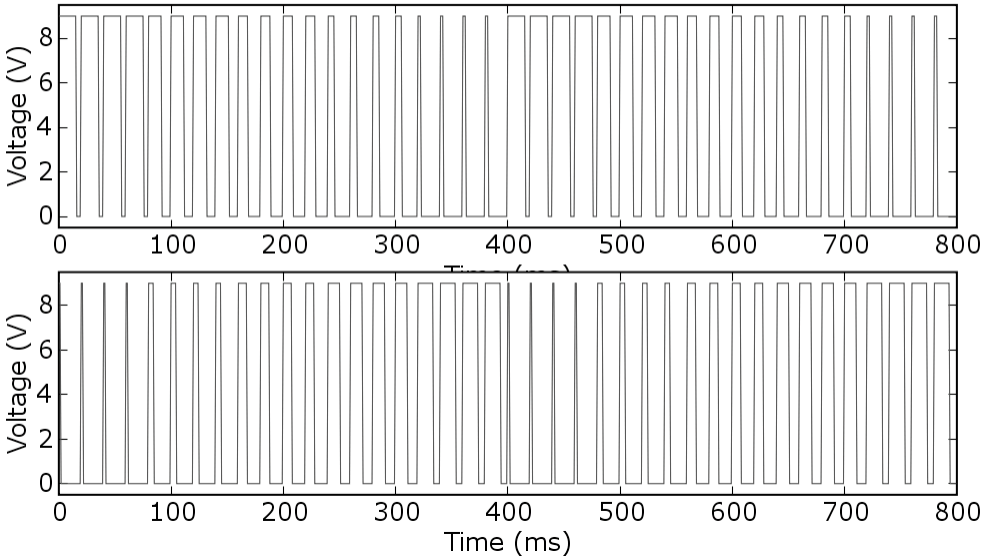



**Figure 3-23: Fading out/fading in PWM**

Finally, arm rolling gestures were detected using the gyroscope along the long axis of the forearm, and were used to select objects. Note that all of these methods are less sensitive and less reliable then the methods used in the applications that follow, and mathematically described in detail in the section entitled “Extracting Useful Information from Quaternions”. The simpler methods above were used because the program was constrained to be hard-coded into the 8051.

The final system allowed a user to trigger 18 sound samples from within Max/MSP by navigating to the correct sample by feel, and triggering it with a rolling movement. The entire system, with the exception of the audio samples themselves, functioned without any external processing, and could potentially be used as a discreet wearable remote control of any system that could be triggered by BlueTooth.

## 3.5 A Fretted Theremin

The first system that made full use of all the software tools developed for this thesis was called a "Fretted Theremin". The Theremin, described in more detail Chapter 1, is an electric field sensing open-air instrument designed to control an analog sound. It is extremely hard to play in practice, though there have been some virtuosic Theremin players, most notably Clara Rockmore, who toured and performed often on this instrument [Cha97].

In this project, a Theremin like instrument was created, giving the user a range of about an octave, distributed over the surface of a sphere. Virtual tactile frets were placed in the air, corresponding to an A major scale; this allowed a user to move quickly to a note by feeling the frets as vibratory sensations on their forearm. Simultaneous control over the volume allows the user to find a note and then play it, which is impossible on a true Theremin. The volume control was mapped to the yaw, or horizontal angular position. The instrument itself was simply a sine wave, with the control parameters being frequency and amplitude. This application used a more complex software system, including a Kalman filter implemented in C++, and a network of UDP packets that allowed communication to occur between a Python-based tactile mapping program, an OpenGL visualization, and a Max/MSP audio patch. The software hierarchy is given in Figure 3-24.

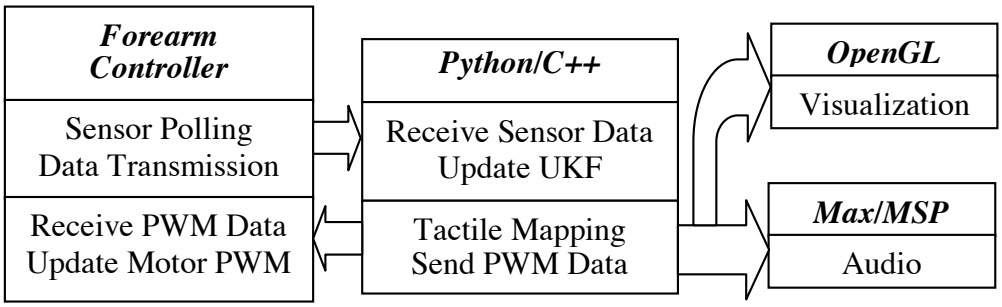



**Figure 3-24: Fretted Theremin software diagram**

The frets were arranged as is shown in Figure 3-25.

**Figure 3-25: Fretted Theremin virtual environment**

### 3.5.1 A Well-Tempered Sine Wave

The angular height, or pitch, of the forearm controller, was used to control the sine wave frequency in this project. This control parameter was given by multiplying a scale factor by the Y component of the orientation vector, determined by quaternion rotation in "Extracting Useful Information from Quaternions". The parameter, when used as a frequency controller, resulted in a linear range of frequencies in which higher pitched members of a traditional chromatic scale are further apart than lower pitched members. Such a mapping can be thought of as the opposite of that in a stringed instrument, in which lower pitched members are further apart than higher pitched members.

Although this mapping could probably be learned by a player, it is difficult to control in this hardware system, because as the pitches get lower and closer together, the sensor noise becomes more and more noticeable, and there is an increase in the danger of producing vibration feedback or jitter as the frets become closer together. To combat this, a logarithmic frequency scale was used, with the equation relating the frequency to the scaled angular elevation being:

$$f = 220(2^{e/12})$$

This allowed the members of the chromatic scale to be evenly spaced within the virtual environment. The frequency chosen as the base frequency was 220Hz, the note 'A' one octave below the note traditionally tuned to by orchestras. The two frequency spacings are shown in Figure 3-26, with the linear spacing to the left, and the more intuitive logarithmic spacing to the right.

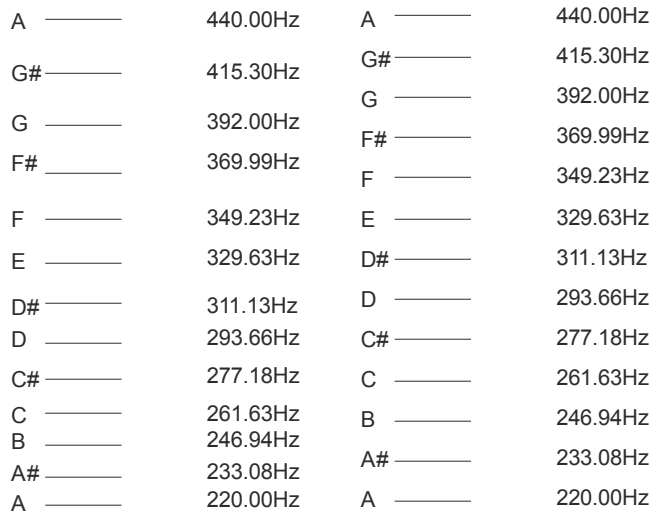



**Figure 3-26: Well-tempered frets**

### 3.5.2 Displaying Frets

The fret mapping was chosen as is shown in Figure 3-27.

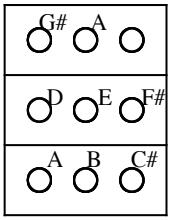



**Figure 3-27: Fret mapping**

The 'A' indicated in the top row is the 'A' at 440Hz, and the one in the bottom row is at 220Hz. The motor in the upper right corner was not used in this application. This mapping was chosen so that when played slowly, each note could be associated with a location on the forearm. At a higher speed, during a glissando that rises in frequency, for example, the exact positions might be ignored, but there would still be a sense of rising, as frets in the bottom row are followed by frets in the middle row, and then frets in the upper row. So as the entire frequency range is moved through quickly, there is an analogous sensation of movement up and down the forearm.

Simply placing frets at the locations where these notes exist would create the possibility of an unwanted vibration jitter. A method of hysteresis was chosen such that a new fret would be felt when a note was reached for the first time, and remained constant until another fret was encountered. For example, if the note "D" were approached from below, the motor labeled "C#" would vibrate until the frequency associated with "D" was reached, and then the motor labeled "D" would vibrate. However, dropping back down below this frequency would not change the fret back to the motor labeled "C#" until the frequency associated with "C#" was reached.

The frequency space between "C#" and "D", or between 277.18Hz and 293.16Hz, could be considered a hysteresis region in which the motor labeled "D" would vibrate if the region had been approached from above, and the motor labeled "C#" would vibrate if

the region had been approached from below. Another way to look at it is that once a note has been played recently, the motor selected won't change until the frequency passes out of the region bordered by the frets on either side of the central note.

### 3.5.3 Adjusting For Movement

Although the method discussed in the previous section should work in theory, in practice it was found that players tended to stop just after passing a fret, and therefore ended up playing a little sharp when approaching a note from below, and a little flat when approaching it from above. This can be attributed to the momentum developed by the arm in the course of moving from one pitch to another. Although this could probably be defeated with practice, it was found that the problem could be alleviated by adjusting the frets so that they were in slightly different places depending on whether the player approached a note from below or from above.

If a frequency f would have corresponded to the location of a fret in the previous section, its final location when approached from below would be given by:

$$f_{final} = f - \frac{f}{100}$$

Similarly, if approached from above, the fret would occur at:

$$f_{final} = f + \frac{f}{100}$$

This caused the vibration to change just before the user struck a note, making it easier to stop on the actual pitch.

## 3.6 A Virtual Crash Cymbal

The previous instrument described was designed to demonstrate a utilitarian approach to tactile feedback, where the vibrating motors were used to display useful information for physically navigating an instrument. This instrument combines that approach with a more abstract one, in which the vibration used is designed to convey a satisfying sensation that corresponds to the sounds being generated. Although the vibration does make it easier to manipulate this instrument in some ways, it serves a greater purpose in adding to the enjoyment and artistic experience of playing the instrument.

The instrument was an abstracted version of a crash cymbal, in which the user can strike the cymbal, dampened it, and brush it, creating a variety of sounds in the process. The tactile and audio design will be described in the following sections. The crash cymbal itself is portrayed visually in Figure 3-28.

**Figure 3-28: Crash cymbal virtual environment**

### 3.6.1 The Tactile Environment

The crash cymbal itself can be thought of existing below the horizontal. To strike it, the user must raise the controller to an elevation above the horizontal, and bring it down sharply. No control or tactile feedback is available above the horizontal, because the controller is not "inside" the instrument in this case. Upon striking the instrument, a large transient jolt is produced in the vibrating motors, providing a sensation of impact.

Touching the surface of the instrument, at the horizontal, will either trigger a crash or dampen a sound currently underway, depending on the speed at which the cymbal has been struck. When dampening the sound, the surface must be touched gently; the sound underway will then fade, and the user will feel an analogous soft fading vibration in each of the motors.

When the controller is “inside” the instrument, below the horizontal, a sort of idling vibration occurs even if there is no movement. However, when moving within the instrument, the fading sound produced by the initial crash can be modulated; this effect is meant to be analogous to the brushing of a real percussion instrument. The movement through the instrument is transmitted as a vibration proportional to the magnitude of the angular velocity, so that a faster movement will generate a more intense sensation. In addition, the motor chosen to provide that sensation depends on the orientation and the direction of movement of the controller.

For example, if the controller is worn on the right forearm, the user’s palm is face down, and the user is making a horizontal angular movement to the left, the thumb is leading the movement, and therefore the motor chosen should be on that side of the forearm. However, if the same movement is executed with the palm facing up, the fifth finger is leading the movement, and the motor chosen should be on that side of the display.

The section entitled “Extracting Useful Information from Quaternions“ describes the mathematics behind choosing the side of the controller on which to indicate movement. In summary, the dot product of the angular velocity vector and vectors normal to the forearm controller are taken, returning the “flux” of the controller through a medium. The result is that, as is desired, a motor on the thumb side will be chosen if the thumb is leading the movement of the arm, regardless of the orientation of the controller relative to the fixed coordinate system. The three diagrams shown in figure 3-29 indicate the location of the motors chosen if the movement is being led by the fifth finger, the palm, or the thumb, assuming the controller is on the left hand:

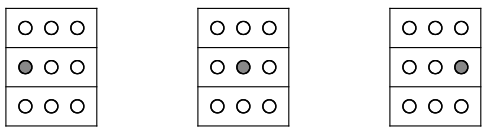


**Figure 3-29: Flux mapping, part 1**

For example, assuming the palm is facing down, if the arm is rotated in a horizontal, or yawing rotation, to the left, the motor chosen would be that in the first diagram shown. If the arm is rotated in a vertical, or pitching motion toward the ground, the motor chosen would be that in the center diagram.

Indicating an upward elevating movement is problematic, because it had been decided at this point that using motors directly under the PCB was too damaging to the filtered orientation estimate. The sensation had to be generated by motors beneath the PCB, but ideally they should provide a sense of pulling rather than pushing; at the least, the sensation should be noticeably different from the other three. The pattern chosen was a quickly repeating effect, in which the two configurations in Figure 3-30 would alternate.

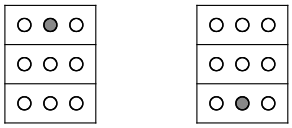


**Figure 3-30: Flux mapping, part 2**

This dull pulsing pattern was intended to imitate the feel of pulling a mechanical actuator against its will, or fighting against a gear train.

### 3.6.2 Sound Design

The set of sounds for this application were intended to convey the feeling of a crash cymbal without making any attempt at imitating a real cymbal in any way. The requirements were a satisfying crash-like noise, an interesting subsiding effect following that, with the option of modulation due to movement within the “cymbal”, and a final dying-away sound as the noise is dampened or simply allowed to fade over time.

#### 3.6.2.1 Deriving Sounds from Xenakis

Once again, “Polytope de Cluny” by Iannis Xenakis [Xen72] was chosen as a reference point for the sounds. This time the piece was distorted and fragmented, rather then simply being chosen as a complete piece to which vibration could be set. The system was designed to extract short fragments from the piece, with about four seconds in duration, and force them into a typical crash-like amplitude envelope, shown in Figure 3-31.

**Figure 3-31: Crash amplitude envelope**

As the soundscape in the Xenakis can be relied upon to be noisy, but not necessarily to have a suitable crash-like attack, these sounds were then added to a set of previously selected Xenakis fragments which were screened to have suitable crash-like qualities. The result of this was something that always sounded like a crash, but was never the same as the previous crash, and retained many noticeable features buried in the original Xenakis, ranging from the low pitched wind-like noise, to the high-pitched glass-like effects.

#### 3.6.2.2 Extending Sounds with FM Synthesis

As the initial crash died away, it was replaced by a longer lasting sound intended to imitate what one might hear when placing a microphone very close to an otherwise inaudibly vibrating cymbal. The sound was intended to have some modulation regardless of the movement by the user, but to have a large modulation added to this when the user moved the controller through the instrument medium.

A range of FM sounds were developed such that they all were appropriately metallic and interesting. The user was then able to alter the modulation index and harmonicity by moving the controller within the instrument [Roa96].

## 3.7 Gesture Based Sound Design

With graduate student colleague Adam Boulanger, a system was implemented for designing sounds ARMadillo. This system will not be described in as much depth as the previous systems, because it is an ongoing project, and because the tactile feedback is currently minimal, but it should be mentioned because some applications of this system take advantage of the design constraints of the forearm controller, particularly its potential for disappearing psychologically.

The Gesture Based Sound Design System allows a user to manipulate parameters of a sound, such as pitch, duration, FM synthesis parameters, reverb, and so on. These manipulations are tracked in the pitch and yaw axis, and a quick, rolling movement selects a desired sound, allowing the user to "zoom in" on the current sample and begin tweaking more parameters. An interesting sound can be designed completely by repeatedly orientating the controller in the direction of the most appealing sound and selecting it. The tactile feedback currently consists simply of a jolt during the selection movement.

Boulanger has proposed using such a system in therapeutic environments. By logging the movement and sound design choices of a participant, it may be possible to produce useful feedback for both physical and psychological therapy. For users who have psychological problems and are uncomfortable with technology, the form factor of a device that can be worn without consequence, and easily forgotten, is particularly appealing.

## 3.8 Conclusion

A suite of simple applications was created that related music to the output of a tactile display and, in some cases, to a virtual tangible environment. While these applications were never subjected to a formal user study, they were explored by many participants during their lifetime, and the reactions of the users were considered in the revisions that followed.

Surprisingly, the most effective application seems to have been "The Rite of Spring", described in the section entitled "Passive Tactile Art". This project was relatively simple in its use of hardware and software, and was by far the simplest of the applications, but had surprising effects on people, who used words like "amazing" and "profound". The impact of music and vibration on an audience clearly needs to be explored, and should not be underestimated; the future of such work and the work of artists like Eric Gunther [Gun01] is viable.

Among the complete environments, the Virtual Crash Cymbal was the most popular. The simplicity of striking an object and making a crashing sound made this application easy to understand and manipulate, while the complexity of the FM sounds and tactile sensations that could be produced added a depth to the environment, and allowed users to make their own discoveries over time.

# 4 Utilitarian Applications: Wearable Braille and Navigation

The bulk of this chapter will focus on a wearable Braille display, and the development of a Braille-like system that will be referred to as Arm Braille; however, the motivation for this application was the desire for a self-contained system that could be used for multiple purposes, including controlling external devices such as those used in home automation systems, receiving text messages from people or locations, and for general urban navigation. While commercial systems do exist for accomplishing such tasks, they typically rely on text-to-speech interfaces rather than haptic ones.

Text-to-speech should not be underestimated as a powerful interface, and is certainly easier to learn then any of the tactile methods presented in this thesis. However, part of the motivation for creating a tactile system was that, when a user is seeing-impaired, the ears become the main window into the outside world. One such person, when interviewed by the author, commented that text-to-speech navigation systems were effective, but felt dangerous [Kes05]. When wearing headphones, and listening carefully to an electronically generated voice, it becomes impossible to tell what is going on in the local environment. A tactile system, despite its other potential flaws, does not impinge on this crucial sense.

Two simple applications will be mentioned as a precursor to the Braille display: a wearable virtual control panel, and a wearable compass. The first will not be described in detail here, as it has already been discussed in the previous chapter as a sound selector; however, it should be remembered here as its alter ego, a wearable remote control that can be used to manipulate anything enabled with BlueTooth. In this case, the 18 knob virtual control panel that the user can feel could be mapped to devices like air conditioners, home entertainment systems, or anything else.

## 4.1 A Wearable Compass

This application was not tested, but will be mentioned in the concluding chapter as a candidate for future research. A simple compass was designed, in which the user was

encouraged to align the thumb of their arm wearing the device in a certain direction. The tactile feedback used to implement this was simple, involving a single pulsing motor. The motor pulsed quicker as the user approached the correct direction, intuitively feeling like a Geiger counter. The motor chosen was either on the right side of the arm or on the left, signifying the direction in which the user's arm was to turn.

During preliminary tests run by the author, it was possible, with this filter and tactile mapping, to stay within about ±5° of the desired direction. This indicates that it might be possible to use ARMadillo as a complete navigation system that guides the user as well as providing text messages related to navigation. Some interesting challenges would have to be addressed before this application could be completed; these will be discussed in the concluding chapter, among other proposals for future research.

## 4.2 Arm Braille

For the sake of simplicity, this research uses Grade 1 Braille, which consists only of the alphabet itself, with no contractions [Brl94]. Although the entire set of Braille characters was implemented, using Grade 1 eliminates certain skills from this application that would otherwise be necessary. For example, in Grade 1 Braille, there is only one character that consists of a single dot, the letter 'A', or ⠁. Having only one character like this in the set means that it is not necessary for the user to be able to determine where on the forearm a single dot character is occurring; it is only necessary to realize that the character has a single dot, and that it must therefore be 'A'.

For reading large quantities of text, Grade 1 would be unacceptable, as the contractions used in Grade 2 are very helpful in shortening the amount of time necessary to read a passage, and help close the gap between the maximum speed possible in visual reading and that in tactile reading. However, this application is strictly for reading simple messages such as signs or short cell phone messages, so the speed gained by using contractions would be negligible.

In the examples that follow, the various methods were first tested on Sile O'Modhrain, a haptics research scientist who is fluent in Braille. She provided consultation on the advantages and disadvantages associated with each method [Mod04]. In the design of Arm Braille, the factors of primary concern were:

1. Legibility, or the ease of distinguishing Braille characters
2. Skill Transfer, or the speed at which a Braille reader can learn the method.
3. Duration, or the minimum length of time required to determine a character.

### 4.2.1 Passive Displays

In the preliminary stages of the Braille display development, a "passive touch" [LL86] version was created which "drew" letters on the skin. The advantage of such a system is that it requires no control at all, and can be used to signal an event without requiring any physical movement from the user. In the first attempt, a character was displayed all at once, simply by vibrating a motor for each corresponding dot. The vibratory pattern for the character N, or ⁘ , is shown in Figure 4-1.

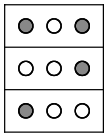


**Figure 4-1: Whole character mapping**

This method failed, mostly due to unwanted vibration conduction through the user's bone, the PCB, and the wires connecting the PCB to the vibrating motors. In general, when more then two or three motors vibrate at once, the resulting pattern is not one of distinct shape, but of a more vague texture.

A second passive method drew the character one dot at a time. For example, the letter "N" would be drawn over the user's skin via sequence in Figure 4-2.

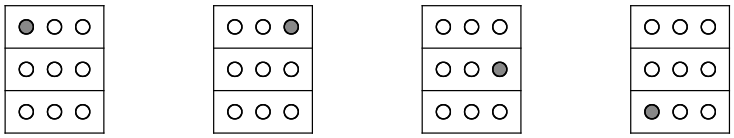


**Figure 4-2: Passive character drawing**

Preliminary tests on O'Modhrain showed that such a method would be legible; however, the system was far to slow for displaying more than a single letter. In the above example, at least four pulses are necessary to convey the letter. The pulses had to be relatively long, because of the problems associated with interpreting passive sensations [LL86]; the characters averaged about three seconds each. The advantage of the system was that the learning curve seemed relatively shallow, and the sensations did seem to "feel" like Braille characters. This method satisfied the legibility and skill transfer constraints, but failed the duration constraint.

Although the passive method was abandoned at this point, the advantages of such a system should be reiterated, and the situations in which it might be successful should be mentioned. For example, the large duration of each character would be acceptable if the messages were only a single character long. For the indoor navigation context described earlier, most of the signs encountered would be similar: "EXIT", "RESTROOMS", "DO NOT ENTER", and so on. Such repetitive signs could be represented by their first letters: "E", "R", "D". A shorthand like this would not be alien to a Braille reader, as the most common representation of literary Braille uses a similar shorthand, in which an individual letter appearing by itself is assumed to represent a word.

A passive method could be used to display a set of previously learned words or phrases with simple sensation patterns. Of course, these patterns need not actually be Braille; but a Braille-based system would take advantage of previously learned skills. The main advantage of a passive system over an active system is that it makes no physical demands on the user.

### 4.2.2 Active Displays

The rest of the Braille systems described in this thesis use “active touch” [LL86], in that they require control from the user, who can scroll through a message at any pace, stop on a character or part of a character, and reverse direction at will. The physical gestures required to control a Braille message, and the algorithms required to interpret these gestures, will be described in a later section. This section focuses on the structure of the characters themselves, and the methods of displaying them that were tried.

The first method that was implemented was a two-step pattern in which the first column was displayed, and then the second, as in Figure 4-3.

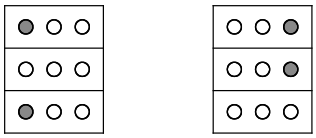


**Figure 4-3: Dividing characters by columns**

Since only one column is delivered at a time, there are essentially only four types of sensations that the user must be able to distinguish:

**Figure 4-4: Grade 1 Braille single column characters, A, B, K, and L**

In addition, the first character has three possible locations, top, middle, and bottom; and the second character has two possible locations, top and bottom. Even with this simple set, letting the motors vibrate in those exact configurations proved to be unhelpful. The three patterns with more than one dot were virtually indistinguishable, and even the single dot character would sometimes set up parasitic vibrations that made it feel like one of the two dot characters.

### 4.2.3 Applying the Concept of the Hamming Distance

The "Hamming Distance", named after the mathematician Richard Hamming, is defined as the number of positions in two strings of equal length in which corresponding elements are different [Opp99]. For example, the following binary strings have a Hamming distance of one:

101000

111000

**Figure 4-5: Hamming distance example**

The above strings represent the Braille characters for K and L, with their dots stretched into a single line such that the first column is given, and then the second. In the context of error correction and detection, it is not desirable to have two valid messages that have a Hamming Distance of one. If a single bit is misinterpreted, the message perceived will be incorrect. A simple method for solving this problem in digital communications is to add a parity bit, which might be zero if the number of ones sent is even, and one if the number of ones sent is odd:

101000 0 (Two ones requires a parity bit of zero)

111000 1 (Three ones requires a parity bit of one)

**Figure 4-6: Parity bits**

In the above example, the strings differ in two places, and therefore have a Hamming Distance of two. The advantage of such a method is that if a single bit is misinterpreted, the resulting string will be invalid. For example, the following string would be known to contain an error somewhere:

111000 0

**Figure 4-7: Invalid string**

The receiver, having detected an error, could then request that the message be resent. These errors might occur due to electrical noise on a wire, or RF noise if the connection is wireless. In the context of Arm Braille text, the noise would be vibration conduction through the wires, the PCB, or the bones in the forearm. The noise could also consist of variations in vibration intensity from one motor to another; that is, given the assumption that these actuators are not all the same, and would be used repeatedly and would be subject to high mechanical, electrical, and thermal battering, it could not be assumed that they would all respond to PWM in the same way. One motor might vibrate with such intensity that, comparatively, it feels as though two motors are vibrating.

In such a context, it is desirable to separate two valid patterns by a Hamming Distance of at least two; that is, they should differ by at least two vibrating motors.

### 4.2.4 Deconstructing the Braille Characters

A second attempt moved away from traditional Braille, and used a representation of four columns that would be easy to distinguish with vibrating motors. These sensations, which made use of the motors on the back of the arm, under the PCB, are shown in Figure 4-8.

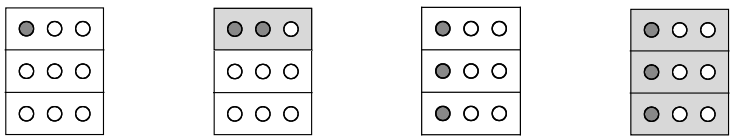


**Figure 4-8: Column sensations for A, B, K, and L**

These four have a Hamming distance of at least two, between the A and the K. The distance between the A and the L is five. The sensations are therefore quite different, and relatively easy to distinguish; and could probably be learned by a dedicated user. This method passes the "duration" test, as a given character requires only two quick

pulses. It passes the "legibility" test, as the patterns can be easily distinguished, even by a user not fluent in Braille. However, it does poorly on the "skill transfer" test. O'Modhrain found that the resulting system was so different from true Braille that it had to be learned from scratch. That is, it didn't "feel" like Braille. A completely new vibratory language, while interesting, was not the object of this application.

### 4.2.5 The Final Braille Method

A compromise was reached; a method similar to the previous one was used, but the characters were displayed as sets of rows rather than sets of columns. The three possibilities for an upper row, ⠁, ⠈, and ⠉, became:

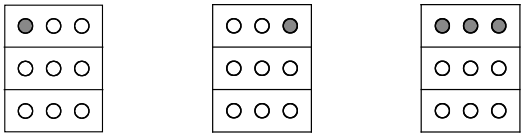
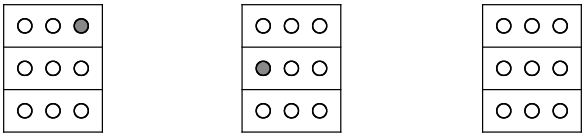


**Figure 4-9: Row sensations**

The resulting system was slower than the previous one, because three rows had to be displayed rather than two columns; however, the final method was conceptually simple and felt more like true Braille. Note that the motors in the center column are used as parity bits, causing the Hamming Distance between any two of the three sensations to be two. In this final method, the word "IN", or ⠊⠝ would be read as a series of sensations that would be scrolled through by the user in the following sequence:



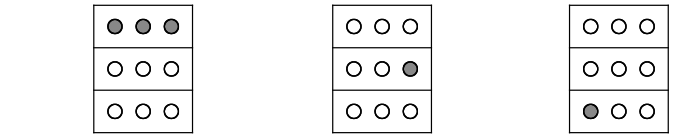



**Figure 4-10: The word "IN"**

### 4.2.6 Tactile Serifs

A “tactile serif” is an intuitive way of thinking about the application of the Hamming distance to Arm Braille. Visually, a serif is a small line or curl that is intended to differentiate one letter from another. The curl at the bottom of a “j”, for example, makes it easily distinguishable from an ‘i’. Similarly, by adding the center row of motors to the display of Arm Braille, characters with two dots in a row are made to be easily distinguishable from characters with one dot in a row.

One can imagine stretching this concept with the use of a multimodal tactile display that adds sensations of pressure and heat, for example. Tactile serifs could be used to speed up character and word recognition without altering the fundamental Braille-like qualities of the characters themselves. Some characters could press harder against the skin, while others could vibrate at higher frequencies then the norm. Still others could feel hotter, if a thermal component is present. Separating the characters in these various dimensions is akin to increasing the Hamming distance between them.

## 4.3 Controlling the Text with Arm Movements

The disadvantage of using an active Braille display is that it requires some form of input from the user. To allow such a device to blend as seamlessly as possible with the user’s other movements, the movements associated with reading characters should be subtle and should be easy to perform in the posture that would be most often adopted while using the device.

In this case, it is assumed that the user is navigating, and would probably be standing up with the arm controlling the display hanging down and close to vertical. The movement chosen to perform the reading action, shown in Figure 4-11, is a rolling gesture along the axis of the forearm about the Y axis.

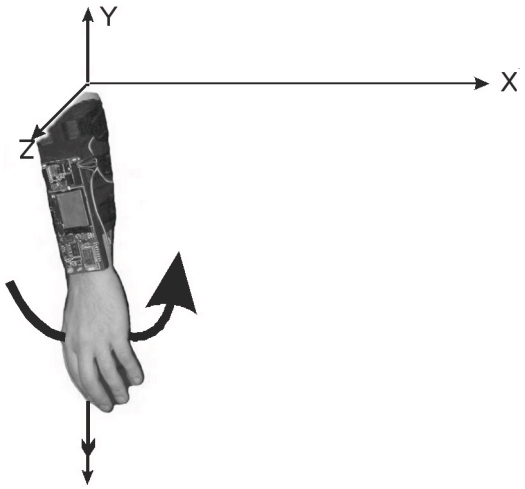



**Figure 4-11: Reading motion**

Such a gesture could be performed while walking, and while holding an object. In a scenario where a user is trying to locate a certain door among many, the doors could announce themselves to the user through the forearm display while the user holds a cane in the right hand and a set of keys in the left hand. In another scenario, a user might be shopping, and could be guided to an exit while holding a shopping bag.

### 4.3.1 Scrolling

To give the impression of scrolling through text, it is important that the text not change when the user's arm is motionless. It is also important that the text advance an amount proportional to the angle through which the user has rotated. However, there should be some flexibility beyond this. Allowing for variations in the movements

required for reading makes the device less tiring to use. In Figure 4-12, two sequences of movements are shown from above, in with the arrow represents the orientation vector of the normal vector to the forearm controller in the X-Z, or horizontal plane. The user's arm is assumed to be hanging vertically downwards in these diagrams, so the arrow can be thought of as the direction that the thumb is pointing in. These gestures would have identical effects of scrolling through six text events.

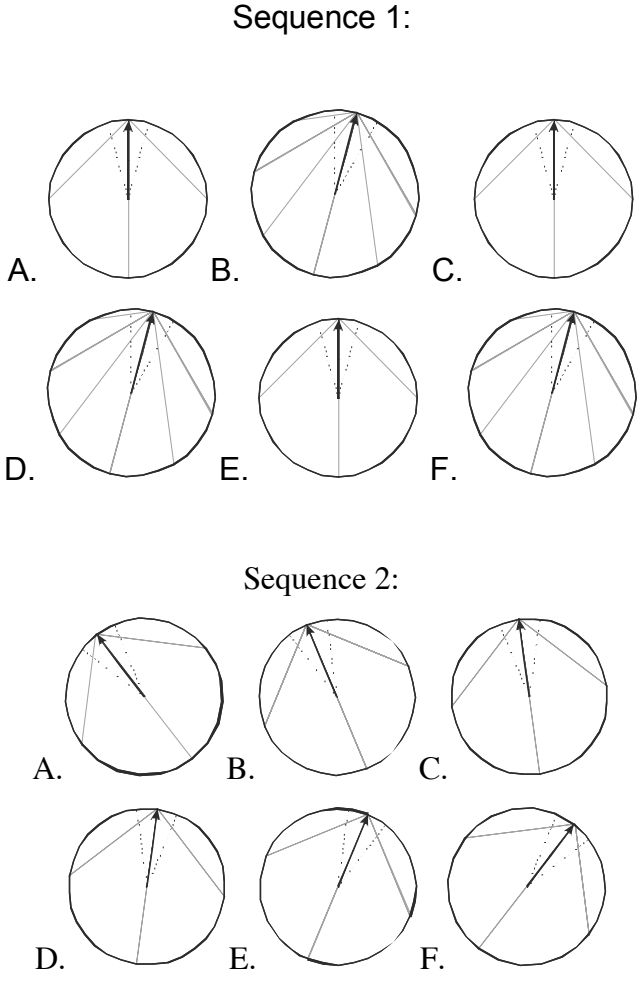



**Figure 4-12: Two valid scrolling gestures**

As the characters are being represented by their rows, either of the above sequences could be used to read two characters. In the first, the user rotates back and forth across a small angle to trigger each vibration pattern. In the second, the user uses

larger sweeping movements to scroll across entire characters or words. Either method can be used in this application, or any combination of the two types of movements; the only requirement for advancing text is that a 15° threshold be passed in either direction. Once one of these thresholds has been passed, two new thresholds are set at 15° on either side of the forearm controller's orientation.

For the sake of simplicity, the application forces the user to hold the display down vertically by ignoring input if the angle between the orientation vector of the display and the vertical, or negative Y axis, exceeds 30°. This allows the use of a simplified algorithm for detecting roll, by simply taking the projection of one of the normal vectors in the X-Z plane, and comparing its angle with magnetic north. It is important that position be tracked directly in this manner, rather than integrating angular velocity. Though the latter method could be used easily to implement the reading mechanism directly from a single gyroscope, and is attractive because it works regardless of the absolute orientation of the forearm controller, the inevitable drift resulting from integrating noise would result in the occasional advancing of text without input from the user. Such events are startling in a virtual reality setting, and destroy the illusion of an object being explored by the user. In this context, these events usually break the user's concentration and require the rereading of a few characters.

Feedback from the vibrating motors can also cause the text to advance unexpectedly. This happens because the sudden jolt generated by a new set of motors being driven causes a transient in the accelerometer data that can sometimes be interpreted by the Kalman filter as another movement, though it should ideally be filtered out. A simple way to avoid this is to place a time delay after each text advancement, allowing these transients to subside.

### 4.3.2 Reversing Direction

To give the user complete control over the message, the ability to reverse direction is crucial. Without this, missing a single character can be frustrating, or even end any chance of successfully reading the message. One method of reading in reverse

might have involved scrolling through the rows backwards, so that a character is represented by its rows in the order 3-2-1 instead of 1-2-3; however, it was determined that this would essentially require the user to learn a second set of characters for reading in reverse. After all, with a little bit of experience, the user should be perceiving characters as units, even if they are transmitted one row at a time. Ideally, the three rows of a character moving across the forearm should be experienced as a single compound sensation; reversing the rows would require a second compound sensation to be learned.

Instead, the rows were sent in their original order, but the characters were reversed. The word "HELLO" would be transmitted as "OLLEH" when being read in reverse, but the rows themselves would not change their orders within individual characters. The disadvantage of such a method is that there is not necessarily any way to determine whether text is being read forwards or backwards. A palindrome, for example, would consist of an identical set of sensations when being read forwards or backwards.

To remove any risk of ambiguity, it was decided that the arm position used when reading backwards would be different from the position used when reading forwards. This makes changing direction as trivial as moving from one position to the next. In this application, reading in reverse is possible when holding the controller within 30° of horizontal. Any orientation not within 30° of either vertical or horizontal is an invalid orientation, in which the text will not advance. In addition, a simple passive tactile sensation is displayed on the arm when the direction is changed. Figure 4-13 shows a user reversing direction, and Figure 4-14 shows the user reading backwards.

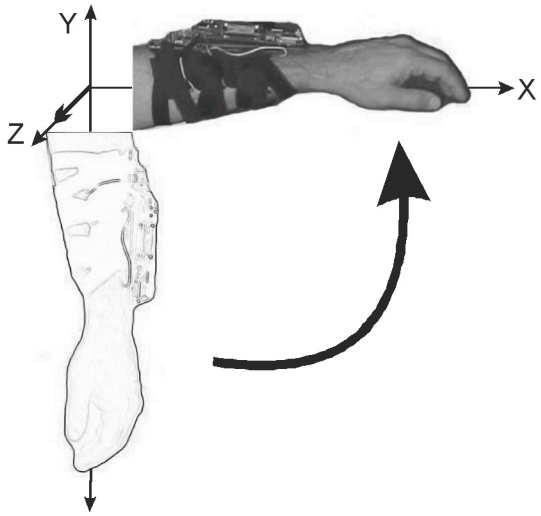



**Figure 4-13: Reversing Direction**

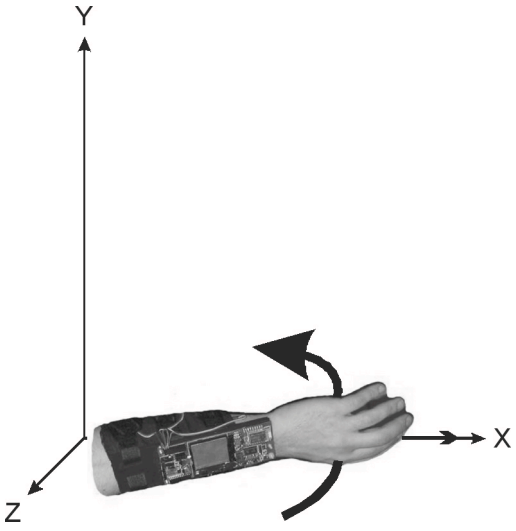



**Figure 4-14: Reading backwards**

## 4.4 Testing the Braille Display

Preliminary tests were done on two sight-impaired subjects, not including Sile O'Modhrain, in which they were trained to identify tactile patterns, Braille characters, and finally Braille words. Both subjects were able to reliably read individual characters after about 25 minutes, with a success rate of 95%, and were reading words slowly after about an hour. One subject is shown modeling the wearable Braille display below:

**Figure 4-15: A forearm controller test driver**

### 4.4.1 The Testing Method

The Arm Braille method that was developed was specifically intended to leverage previous Braille skills; therefore, it should not have been necessary to teach the subjects a mapping of tactile patterns to characters, because this mapping should have been close enough to Braille for the characters to be recognizable. The main skill that had to be taught was a mental mapping of the forearm itself, which was necessary for interpreting patterns displayed there. This skill was introduced gradually by first asking the user to identify a randomly chosen character that was either a left-side character, a right-side character, or a character using both sides. The system randomly selected between ⠇ , ⠸ , and ⠿ , and the user was asked to say “left”, “right”, or “both”.

As control over the character was a necessary precursor to reading, the users were then asked to control the display of a series of ⠇ , and were asked to stop on various rows and hold the character there. It became clear that the control algorithm would have to be improved in order for faster reading to occur, because users had some trouble achieving this skill. Nevertheless, both were able to master it within the first 25 minutes.

The next step was distinguishing between characters that had exactly one dot per row. In Grade 1 Braille, these characters are ⠂ , ⠐ , and ⠁ . The main skill required for this

exercise is the ability to detect when a character has begun; as all Grade 1 characters include at least one dot in the top row, this skill can be reduced to the ability to sense when a pulse has occurred at the wrist. All other attributes of these characters can be derived from this.

The user was then asked to distinguish between the characters ⠁, ⠃, ⠉, and ⠙. This exercise was designed to isolate the user's ability to detect spaces within the character. It was found that ⠃ and ⠉ were especially hard to tell apart, so for both users it was necessary to drill those two characters at random.

The rest of the alphabet was introduced in bulk as a single string, ⠑⠋⠛⠓⠊⠚⠅⠇⠍⠝⠕⠏⠟⠗⠎⠞⠥⠧⠺⠭⠽⠵, in which the user was asked to read through forwards and backwards, and to find specific letters. This allowed the user to learn to change direction and "scrub" a word by moving forwards and backwards within it, and to get used to the rest of the alphabet.

Random character recognition was tested next, followed by word recognition. The words used were chosen randomly from a list of 50 words commonly found on signs, such as "ENTRANCE", "EXIT", "MEN", "WOMEN", and so on.

### 4.4.2 Evaluation

The purpose of this test was to see how quickly a user could become able to use the device independently. For this reason, reading speed was not considered an important criteria; the main parameter being examined was the training time necessary for a user to be able to recognize characters reliably, and the training time necessary for a user to be able to read words. In the case of the latter, users were allowed to stick with a word as long as necessary, "scrubbing" it by reading it backwards and forwards. The entire testing period was about 1 hour for each user; to measure the potential maximum reading speed for Arm Braille, many follow up sessions would be necessary.

Both users were able to achieve a satisfactory character recognition of above 95% within 25 minutes, demonstrating that the device was successfully leveraging the Braille skills that had already been developed. This time compares favorably to non-Braille

based tactile text systems such as Vibratese, which users mastered in about 12 hours [Gel60], and the Tactuator, which users mastered in about 20 to 27 hours [Tan96]. In addition, it is assumed that these training periods were not executed all at once; that is, users had time to digest the material they had learned in one session, and revive it at the next. Arm Braille character recognition was mastered in 25 minutes in one sitting.

Word recognition was achieved by the end of the hour, although the process was slow and often laborious. Reading speed comparisons with Vibratese and its highest observed speed of 38 words per minute (assuming an average of five characters per word) would require many follow up training sessions, and the Tactuator was never tested with English words to the knowledge of the author [Gel60], [Tan96]. However, the criteria being focused on was the fast acquisition of independence, and that was met in one hour. For an aid like ARMadillo to be useful, a user should be able to learn to use it as quickly as possible, so that future "training sessions" can be experienced in the real world.

Although word recognition was achieved in an hour, because of the criteria for developing independence, users were allowed to stick with a word until they successfully read it, with no time limits. As their forearm skin perception skills were still rudimentary and less than an hour old, some of these early words took quite a long time.

It was interesting to observe the relationship word reading had with the user's prediction of the word from the first few characters. This predicting was consciously encouraged by the author in the choice of words commonly used in signs. One user correctly predicted the words "ENTRANCE" and "EXIT" after the first two letters, and was able to read the rest of the word very quickly as a result.

This predicting had its consequences, when this same user required almost 10 minutes to read the word "CAFETERIA"; this was due largely to an initial incorrect prediction of the word. After reading "CAFE" the user announced that the word was "café', and was startled to read the "t" that followed. Misreading it as an "s", an understandable mistake considering their respective Braille characters, ⠎ and ⠞, the user announced that the word was "cafés", and was again startled to read the "e" that followed. This downhill spiral culminated with the user being able to spell out the entire word correctly from memory, while still being unable to identify it as a word. It is assumed that such problems would fade with practice and increased reading speed.

# 5 Conclusion

A device called ARMadillo was made that allowed users to interact with a virtual environment using arm movements, while perceiving the environment through a tactile display. The device satisfied its design constraints of simplicity and convenience with respect to its potential use during everyday movement and interaction. In addition to the hardware and software necessary to implement the tools for developing virtual environments, a variety of applications were tested on the device, demonstrating its potential for being both interesting and useful.

## 5.1 The Future of the Tracking System

Although the orientation tracking system was sufficient for developing a set of interesting applications, the addition of position tracking would uncover a much larger space of environments. In the case of the music applications, this position tracking should be on a very fine scale; for example, an ultrasound system like that used in the Lady's Glove [Cha97] would allow a complete rectilinear environment to be developed, rather than the spherical surface that was used in this thesis. However, although this system would still be wearable, it would suffer in that it would break free of the simple and convenient form factor of the forearm controller, which involves only a single device that can be easily attached and removed.

More interesting, perhaps, would be a system that maintained the form factor, but accomplished the goal of position measurement. Such a system might use some combination of video, ultrasound, and radar to image the body of the user, and perhaps the floor, walls, and ceiling of the space being used. Such a system would require a complex filter and hardware, but might be able to take position measurements without introducing undesirable base stations.

Another parameter that would be interesting to add to the filter is the location of the user. For navigation or for location on a performance stage, measuring the position of the person wearing the controller with an external system could have interesting

consequences. In addition, the inertial sensors already on board could be used to dead-reckon between external position measurements, providing a more accurate estimate, such as that produced by GPS/INS systems. GPS itself is probably not appropriate for this research, however, as it tends not to function well indoors or near urban areas. If it did, simply adding a CompactFlash GPS card to the system could accomplish this task. A more interesting approach for indoor use might be the combination of the BlueTooth module and a WiFi CompactFlash card to record signal strengths from such sources, and triangulate position from these measurements.

Finally, the decision to leave the fingers unfettered had consequences in terms of available movement for input in both the musical applications and the Braille system. Using a glove would solve this problem, but at the expense of the design constraints of the forearm controller. A more interesting approach would be to attempt to measure finger movement by imaging the tendons in the forearm.

## 5.2 The Future of the Virtual Musical Instruments

The music applications presented in this thesis were proofs of concept, and not intended for use in a performance. A logical next step would be to follow the design methods used in this thesis and to create an instrument that is worthy of being presented in front of an audience. In addition, adding a second forearm controller to the other arm would allow a performer to interact with the environment with both arms, and introduces the possibility measuring the distance between the two controllers, as in Waisvisz' "Hands" [Cha97]. The additional sensing methods previously described would introduce interesting possibilities of capturing finger movements without gloves, and the movements of the performer relative to a stage. These could be effectively incorporated into an instrument.

Because the device leaves the fingers free, one interesting possibility that should be explored further is the use of the device with a physical system. Such a system could include an acoustic instrument, such as a piano, or an electronic instrument specifically

designed to complement ARMadillo. They could interact in many interesting ways. For example, if an acoustic piano were used, a performer could play a few notes on the keyboard, which would be simultaneously analyzed by a real-time listening computer system. The performer could then move to a virtual instrument in the space above the keyboard and manipulate synthesized sounds based on the previous piano sounds. Such a system would be similar to that of an organ or a harpsichord, which have multiple layered keyboards; in this example, one would be real, and others would be virtual. In addition, virtual instruments introduce the possibility of instruments that change shape in real-time as the performer plays them.

## 5.3 The Future of Arm Braille and a Navigation System

Many interesting user studies await the Arm Braille system, even without any further technological improvement. Follow-up studies with more users would allow the maximum reading speed of the device to be tested. In addition, it might be interesting to test the device on sighted users, while asking them to learn to read traditional Braille as a control. For simple navigation requirements, many of the signs encountered would be similar; such a system lends itself to the development of a shorthand using only one or two characters. The user's tendencies to predict "ENTRANCE" from "EN" and "EXIT" from "EX" could be taken advantage of in this case, and shortening the signs to single or double characters would decrease the training time from the 1 hour for rudimentary word reading skills to the 25 minutes necessary for reading individual characters. Such a system could also be augmented with an intuitive set of abstract tactile sensations that are not related to Braille.

For real-time navigation, the simple compass system presented earlier could be used to guide a user from one place to another, while interrupting periodically with useful text messages. This system was not tested, in part, because many filtering challenges lie in its path. In addition to the requirement of combining the inertial sensing measurements with external position measurements for the best accuracy, guiding the user with heading

information from the controller is not trivial. Assuming that the heading is also being compared to one found by taking the derivative of the position measurements of some urban GPS-like system, the magnetic heading on the controller would suffer from a set of daunting errors, including magnetic offsets due to local ferrous material, and uncertainties in the way in which the controller has been strapped to the arm, and the difference between the direction the user's thumb points and the direction the user is actually walking. A hardware system that simplifies these problems is ActiveBelt, which locks the magnetic sensors to the user's hips [TY04]. However, such a system does not allow for the control input that an arm-based device does, and does not include support for Braille text.

For the forearm controller, using a learning algorithm and a more complex filter, one might be able to minimize the heading errors, and using a constantly updating feedback control, the errors could be accounted for. In addition, peak detection could be used to analyze the natural swinging of the arm, and a learning algorithm could use this information to further minimize the heading errors.

## 5.4 Braille Based Music Applications?

Although the utilitarian applications and the artistic applications were never intended to cross paths, it is interesting that subjects on which the Arm Braille was tested seemed more interested in the virtual musical instruments. Braille based music has existed since Louis Braille himself devised the first system in the 1820s. Combining the two systems could have startling results; for example, the forearm controller could function as an aid for "sight reading" on an acoustic instrument.

## 5.5 Beyond ARMadillo

While the forearm controller succeeded in its goal of being an interesting prototype of an intimate, versatile, tactile virtual reality environment, the applications that

would be possible on a future version with a more advanced technology are quite compelling. One can imagine a small, simple wristwatch-like device that opens up an entire tactile world that mixes reality with an imaginary layer of being, one that can only be felt through the device. Such an intimate and discreet sixth sense has already been shown in its primitive form to be capable of the fine control of a musical system, and the empowering support of a text-based guidance system.

With more precise tracking, a truly convincing multimodal tactile display, and increased processing power, this virtual environment could break free of the spherical shell that was used in this thesis. A tactile world could be built around the real world inhabited by the user, in which invisible control panels can be found near any objects that can be manipulated, guiding paths can be felt whenever real paths are at foot, and an endless supply of entertaining and useful applications can be learned and explored, allowing the controller to transcend the discontinuous set of applications presented here, and truly become a method of sensing the world.

# Appendix: C++ Code for Unscented Kalman Filtering

```
/*
**********************************************
Header file for UnscentedKalmanFilter.h
David Sachs
Compiles on Microsoft Visual C++ 6.0

Implements a class containing an Unscented Kalman Filter,
a Square Root Unscented Kalman Filter, and a Quaternion
Based Square Root Unscented Kalman Filter.
**********************************************
*/

#include <iostream>
#include "gsl/gsl_matrix.h"
#include "gsl/gsl_cblas.h"
#include "gsl/gsl_blas.h"
#include "gsl/gsl_linalg.h"
#include "stdlib.h"
#include "windows.h"
#include "time.h"
#include <cmath>

#define STATELENGTH 9
#define MEASUREMENTLENGTH 9
#define MOMENTARM 0.3

class UKF {
public:

//Constructor
UKF();
//Destructor
```

```
~UKF();


/*
************************************************************
User Accessible Filter components
************************************************************
*/


//Filter initialization in which most of the memory used is allocated.
//This should be called once before the filtering begins.
void initFilter();

//Times the quaternion based square-root unscented Kalman filter
double timeFilter(int numTrials);


/*
************************************************************************
These functions make it easier to interface with this code from Python
************************************************************************
*/

/*
Allows a python script to set the measurement vector, the magnetic dip angle,
and the time since the filter was last called, and then iterate the quaternion
based square root filter.
*/
void callFilter(double measurementData1, double measurementData2, double measurementData3,double
measurementData4, double measurementData5, double measurementData6, double measurementData7, double
measurementData8, double measurementData9, double timeCalled, double magx,double magy, double magz);

//Sets an element of the orientation quaternion
void setQuatStateElement(int elementNumber, double value);

void printNoise();
```

```
//Retrieves an element of the state vector
double getStateElement(int elementNumber);

//Retrieves an element of the orientation quaternion
double getQuatStateElement(int elementNumber);

void setMeasurementNoise(int elementNumber, double value);

void setProcessNoise(int elementNumber, double value);


/*
************************************************
Unscented Kalman Filters
************************************************
*/


//UKF
void ukf();

//square root UKF
void srukf();

//quaternion based square root UKF
void qsrukf();


private:

//Initialize SRUKF parameters
double lambda, gamma, covWeightZero, covWeightOne, sqrtCovWeightZero, sqrtCovWeightOne;
double alpha;// = 1;//0.01;
double beta;// = 2.0;
double kappa;// = 0;
int numberOfSigmaPoints;
double newTime;// = 0;
```

```
//Declare buffers to be used in calculations (minimizing memory allocation
//at runtime

//Declare matrix buffers
gsl_matrix *stateBuf;
gsl_matrix *sigBuf;
gsl_matrix *obsSigBuf;
gsl_matrix *quatBuf;
gsl_matrix *quatErrBuf;
gsl_matrix *wObsSigBuf;
gsl_matrix *updStateBuf;
gsl_matrix *wUpdStateBuf;
gsl_matrix *mWeights;
gsl_matrix *qWWeights;
gsl_matrix *obsMWeights;
gsl_matrix *QRBuf;
gsl_matrix *obsQRBuf;
gsl_matrix *processNoise;
gsl_matrix *measurementNoise;
gsl_matrix *vErrBuf;
gsl_matrix *cov;
gsl_matrix *tempCov;
gsl_matrix *uBuf;
gsl_matrix *inverted;
gsl_matrix *invertedTrans;
gsl_matrix *makeUpperTriangle;
gsl_matrix *obsMakeUpperTriangle;
gsl_matrix *rBuf;
gsl_matrix *obsRBuf;
gsl_matrix *obsCov;
gsl_matrix *obsCovTemp;
gsl_matrix *dchudTestMat;
gsl_matrix *PXY;
gsl_matrix *PXYTemp;
gsl_matrix *kalmanGain;
gsl_matrix *sigBufLong;
```

```
gsl_matrix *obsSigBufLong;
gsl_matrix *sigBufShort;
gsl_matrix *obsSigBufShort;
gsl_matrix *sigBufLongTemp;
gsl_matrix *obsSigBufLongTemp;
gsl_matrix *sigBufShortTemp;
gsl_matrix *obsSigBufShortTemp;
gsl_matrix *stateMeans;
gsl_matrix *obsMeans;
gsl_matrix *meanTest;

//Declare Vector Buffers
gsl_vector *stateVector;
gsl_vector *measurementVector;
gsl_vector *state;
gsl_vector *qState;
gsl_vector *stateMean;
gsl_vector *sqCWMean;
gsl_vector *obsSqCWMean;
gsl_vector *obsPredMean;
gsl_vector *tempVec;
gsl_vector *rowVector;
gsl_vector *obsTempVec;
gsl_vector *sVec;
gsl_vector *cVec;
gsl_vector *obsSVec;
gsl_vector *obsCVec;
gsl_vector *dchudVec;
gsl_vector *dchddVec;
gsl_vector *zeroVec;
gsl_vector *measVec;
gsl_vector *rotError;

//Declare quaternion buffers
gsl_vector *posQuatInv;
gsl_vector *errQuat;
gsl_vector *magQuat;
gsl_vector *angVQuat;
```

```
gsl_vector *tempQuat;
gsl_vector *gravQuat;
gsl_vector *tempGravQuat;
gsl_vector *tempMagQuat;
gsl_vector *gravNewQuat;
gsl_vector *magNewQuat;
gsl_vector *tempAngMomQuat;
gsl_vector *angMomNewQuat;
gsl_vector *centAccel;
gsl_vector *tempPosQuat;
gsl_vector *posQuat;
gsl_vector *momentArm;

//Declare permutation
gsl_permutation *perm;


/*
*******************************************************
Vector and matrix display utilities
*******************************************************
*/

//Display a matrix on screen (for debugging only)
void matPrint(gsl_matrix *m);
//Display a vector on screen (for debugging only)
void vecPrint(gsl_vector *v);


/*
*************************************************************
Basic Mathematical functions
*************************************************************
*/

//Perform quaternion multiplication on qLeft and qRight, leaving the result in qLeft
void quatMult(gsl_vector *quatLeft, gsl_vector *quatRight);
```

```
//Invert qInitial, and leave the result in qInverse
//(only works if the quaternion is normalized)
void quatInverse(gsl_vector *quatInverse, gsl_vector *qInitial);
//Normalize a quaternion
void quatNormalize(gsl_vector *quat);

//Perform a cholesky factor downdate
//Ported from the LINPACK FORTRAN function DCHDD
int dchdd(int p, gsl_vector *x, gsl_vector *c, gsl_vector *s, gsl_matrix *r);
a
//Perform a cholesky factor update
//Ported from the LINPACK FORTRAN function DCHUD
void dchud(int p, gsl_vector *x, gsl_vector *c, gsl_vector *s, gsl_matrix *r);



/*
*****************************************************
Models used in filter
*****************************************************
*/


//Update the state vector in a quaterion square root UKF
void stateFunction();
//Update the observation vector for a quaternion based square root UKF
void observationFunction();



void updateQuat();
void generateSigmaPoints();
void findMeanState();
void findMeanStateQuat();a
void updateCovarianceState();
void findMeanObservation();
void findMeanObservationQuat();
void updateCovarianceObservation();a
```

```
void findKalmanGain();
void finalMean();
void finalCovariance();

};

/*
*******************************************************
UnscentedKalmanFilter.cpp
David Sachs
Compiles on Microsoft Visual C++ 6.0

Implements a class containing an Unscented Kalman Filter,
a Square Root Unscented Kalman Filter, and a Quaternion
Based Square Root Unscented Kalman Filter. GNU GSL is
used QR Decomposition and matrix inversion, and the
functions DCHDD and DCHUD were ported from LINPACK's
FORTRAN library, to provide cholesky factor downdating
and updating.

The state model updates angular velocity and quaternion
orientation, and includes a gradient descent convergence
algorithm for estimating the mean of a set of quaternions.

The observation model takes the state and predicts a set
of sensor values for a three-axis accelerometer, a
three-axis magnetometer, and a three-axis gyroscope.

Wrapper functions are also provided for accessing the
filter as a Python module.
*******************************************************
*/
#include <iostream>
#include "UnscentedKalmanFilter.h"
#include "gsl/gsl_matrix.h"
#include "gsl/gsl_cblas.h"
#include "gsl/gsl_blas.h"
#include "gsl/gsl_linalg.h"
```

```
#include "stdlib.h"
#include "windows.h"
#include "time.h"
#include <cmath>

#define PI 3.1415926535897931

using namespace std;

//Constructor
UKF::UKF() {}
//Destructor
UKF::~UKF() {}


/*
**********************************************************
User Accessible Filter components
**********************************************************
*/


//Filter initialization in which most of the memory used is allocated.
//This should be called once before the filtering begins.
void UKF::initFilter() {
      int i, j;
      alpha = 1;//0.01;
      beta = 2.0;
      kappa = 0;
      double processNoiseArray[STATELENGTH][STATELENGTH] =
      {
            {0.00007, 0, 0, 0, 0, 0, 0, 0, 0},//MUST BE TWEAKED!!!
            {0, 0.00007, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0.00007, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0.00012, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0.00012, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0.00012, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 0.00025, 0, 0},
```

```
        {0, 0, 0, 0, 0, 0, 0, 0.00025, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0.00025},

    };
    double measurementNoiseArray[MEASUREMENTLENGTH][MEASUREMENTLENGTH] =
    {

        {0.00042, 0, 0, 0, 0, 0, 0, 0, 0}, // MUST BE TWEAKED!!!
        {0, 0.00042, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0.00042, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0.00017, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0.00017, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0.00017, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0.00038, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0.00038, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0.00038},
    };
    double covarianceArray[STATELENGTH][STATELENGTH] =
    {
    {1.7, 0, 0, 0, 0, 0, 0, 0, 0},//MUST BE TWEAKED!!!
      {0, 1.7, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 1.7, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0.1, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0.1, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0.1, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0.0001, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0.0001, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0.0001},
    };

    numberOfSigmaPoints = 2*STATELENGTH+1; //Initialize UKF parameters
  lambda = (alpha*alpha)*(STATELENGTH+kappa)-STATELENGTH;
    lambda = -3;      ;
  gamma = sqrt(lambda+STATELENGTH);

    //Initialize covariance weights
    covWeightZero =(lambda/(STATELENGTH+lambda))+(1-(alpha*alpha)+beta);
    covWeightOne = 1.0/(2*(STATELENGTH+lambda));
```

```
  if (covWeightOne<0)
        sqrtCovWeightOne = -sqrt(-covWeightOne);
else
        sqrtCovWeightOne = sqrt(covWeightOne);
  if (covWeightZero < 0)
        sqrtCovWeightZero = -sqrt(-covWeightZero);
else
        sqrtCovWeightZero = sqrt(covWeightZero);

  //Initialize all the matrix buffers
  sigBuf = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
  obsSigBuf = gsl_matrix_calloc(MEASUREMENTLENGTH,numberOfSigmaPoints);
  wObsSigBuf = gsl_matrix_calloc(MEASUREMENTLENGTH,numberOfSigmaPoints);
  stateBuf = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
  updStateBuf = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
  wUpdStateBuf = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
  mWeights = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
  obsMWeights = gsl_matrix_calloc(MEASUREMENTLENGTH,numberOfSigmaPoints);
  QRBuf = gsl_matrix_calloc(3*STATELENGTH,STATELENGTH);//??????
  obsQRBuf = gsl_matrix_calloc(2*STATELENGTH + MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  processNoise = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
  quatBuf = gsl_matrix_calloc(4, numberOfSigmaPoints);
  quatErrBuf = gsl_matrix_calloc(4, numberOfSigmaPoints);
  measurementNoise = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  cov = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
  tempCov = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
  uBuf = gsl_matrix_calloc(STATELENGTH,MEASUREMENTLENGTH);
  makeUpperTriangle = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
  obsMakeUpperTriangle = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  inverted = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  invertedTrans = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  rBuf = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
  obsRBuf = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  obsCov = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  obsCovTemp = gsl_matrix_calloc(MEASUREMENTLENGTH,MEASUREMENTLENGTH);
  PXY = gsl_matrix_calloc(STATELENGTH, MEASUREMENTLENGTH);
  PXYTemp = gsl_matrix_calloc(STATELENGTH, MEASUREMENTLENGTH);
  kalmanGain = gsl_matrix_calloc(STATELENGTH,MEASUREMENTLENGTH);
```

```
sigBufLong = gsl_matrix_calloc(STATELENGTH, 2*STATELENGTH);
obsSigBufLong = gsl_matrix_calloc(MEASUREMENTLENGTH, 2*STATELENGTH);
sigBufShort = gsl_matrix_calloc(STATELENGTH, 1);
obsSigBufShort = gsl_matrix_calloc(MEASUREMENTLENGTH, 1);
sigBufLongTemp = gsl_matrix_calloc(STATELENGTH, 2*STATELENGTH);
obsSigBufLongTemp = gsl_matrix_calloc(MEASUREMENTLENGTH, 2*STATELENGTH);
sigBufShortTemp = gsl_matrix_calloc(STATELENGTH, 1);
obsSigBufShortTemp = gsl_matrix_calloc(MEASUREMENTLENGTH, 1);
stateMeans = gsl_matrix_calloc(STATELENGTH,numberOfSigmaPoints);
obsMeans = gsl_matrix_calloc(MEASUREMENTLENGTH,numberOfSigmaPoints);

//Initialize all vector buffers
rowVector = gsl_vector_calloc(numberOfSigmaPoints);
state = gsl_vector_calloc(STATELENGTH);
sqCWMean = gsl_vector_calloc(STATELENGTH);
obsSqCWMean = gsl_vector_calloc(MEASUREMENTLENGTH);
obsPredMean = gsl_vector_calloc(MEASUREMENTLENGTH);
tempVec = gsl_vector_calloc(STATELENGTH);
obsTempVec = gsl_vector_calloc(MEASUREMENTLENGTH);
sVec = gsl_vector_calloc(STATELENGTH);
cVec = gsl_vector_calloc(STATELENGTH);
obsSVec = gsl_vector_calloc(MEASUREMENTLENGTH);
obsCVec = gsl_vector_calloc(MEASUREMENTLENGTH);
dchudTestMat = gsl_matrix_calloc(STATELENGTH,STATELENGTH);
dchudVec = gsl_vector_calloc(STATELENGTH);
dchddVec = gsl_vector_calloc(STATELENGTH);
zeroVec = gsl_vector_calloc(STATELENGTH);
measVec = gsl_vector_calloc(MEASUREMENTLENGTH);
stateMean = gsl_vector_calloc(STATELENGTH);
rotError = gsl_vector_calloc(3);

//Initialize permutaion
perm = gsl_permutation_calloc(MEASUREMENTLENGTH);

//Initialize quaternion buffers
magQuat = gsl_vector_calloc(4);
tempAngMomQuat = gsl_vector_calloc(4);
angMomNewQuat = gsl_vector_calloc(4);
```

```
tempGravQuat = gsl_vector_calloc(4);
tempMagQuat = gsl_vector_calloc(4);
gravNewQuat = gsl_vector_calloc(4);
magNewQuat = gsl_vector_calloc(4);
posQuatInv = gsl_vector_calloc(4);
errQuat = gsl_vector_calloc(4);
centAccel = gsl_vector_calloc(4);
tempPosQuat = gsl_vector_calloc(4);
angVQuat = gsl_vector_calloc(4);
tempQuat = gsl_vector_calloc(4);
posQuat = gsl_vector_calloc(4);
gravQuat = gsl_vector_calloc(4);

//Initialize the gravity quaternion
gsl_vector_set(gravQuat, 2, 1.0);


//Initialize the state vector
gsl_vector_memcpy(stateMean, state);
gsl_vector_set(stateMean, 3, 0.0);
gsl_vector_set(stateMean, 4, 0.0);
gsl_vector_set(stateMean, 5, 0.0);

//Initialize the orientation quaternion
gsl_vector_set(posQuat, 0, 1);//cos(ang/2));
gsl_vector_set(posQuat, 1, 0);
gsl_vector_set(posQuat, 2, 0);//sin(ang/2));
gsl_vector_set(posQuat, 3, 0);

//Transfer the initial covariance arrays to GSL matrices
for (i=0; i<STATELENGTH; i++) {
      for (j=0; j<STATELENGTH; j++) {
            gsl_matrix_set(processNoise, i, j, processNoiseArray[i][j]);
            gsl_matrix_set(cov, i, j, covarianceArray[i][j]);
      }

      //Make a matrix of all ones in the upper triangle
      for (j=STATELENGTH-1; j>=i; j--) {
```

```
                  gsl_matrix_set(makeUpperTriangle, i, j, 1.0);
            }
      }

      //Transfer the initial covariance arrays to GSL matrices
      for (i=0; i<MEASUREMENTLENGTH; i++) {
            for (j=0; j<MEASUREMENTLENGTH; j++) {
                  gsl_matrix_set(measurementNoise, i, j, measurementNoiseArray[i][j]);
            }
      //matPrint(processNoise);
      //matPrint(measurementNoise);
            //Make a matrix of all ones in the upper triangle
            for (j=MEASUREMENTLENGTH-1; j>=i; j--) {

                  gsl_matrix_set(obsMakeUpperTriangle, i, j, 1.0);
            }
      }

      //Generate UKF weights
      lambda=0;
      gsl_vector_set_all(tempVec, lambda*1.0/(STATELENGTH+lambda));
      gsl_vector_set_all(obsTempVec, lambda/(STATELENGTH+lambda));

      gsl_matrix_set_col(mWeights, 0, tempVec);
      gsl_matrix_set_col(obsMWeights, 0, obsTempVec);
      gsl_vector_set_all(tempVec, 1.0/(2*(STATELENGTH+lambda)));
      gsl_vector_set_all(obsTempVec, 1.0/(2*(STATELENGTH+lambda)));
      for (i=1; i<numberOfSigmaPoints; i++) {
            gsl_matrix_set_col(mWeights, i, tempVec);
            gsl_matrix_set_col(obsMWeights, i, obsTempVec);
      }
}


//Times the quaternion based square-root unscented Kalman filter
double UKF::timeFilter(int numTrials) {
      double freq, t, tsum=0;
      numTrials+=100;
```

```
	LARGE_INTEGER tickspersecond, tick1, tick2;
	QueryPerformanceFrequency(&tickspersecond); //Initialize timer
	freq = (double)tickspersecond.QuadPart;
	for (int i=0; i<numTrials; i++) {
		QueryPerformanceCounter(&tick1); //Record start time
		callFilter(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1); //Call filter
		QueryPerformanceCounter(&tick2); //Record end time
		t = (double)(tick2.QuadPart-tick1.QuadPart);
		if (i>100) { //Sum up the iteration times, skipping the first 100
			tsum += (t/freq);
		}
	}
	return tsum/(numTrials-100); //Return the average iteration time


}

/*
************************************************
Unscented Kalman Filters
************************************************
*/


//UKF
void UKF::ukf() {
	generateSigmaPoints();
	stateFunction();
	findMeanState();
	updateCovarianceState();
	generateSigmaPoints();
	observationFunction();
	findMeanObservation();
	updateCovarianceObservation();
	findKalmanGain();
	finalMean();
	finalCovariance();
}
```

```
//Square root UKF
void UKF::srukf() {
	generateSigmaPoints();
	stateFunction();
	findMeanState();
	updateCovarianceState();
	generateSigmaPoints();
	observationFunction();
	findMeanObservation();
	updateCovarianceObservation();
	findKalmanGain();
	finalMean();
	finalCovariance();
}


//Quaternion based square root UKF
void UKF::qsrukf() {
	generateSigmaPoints();
	stateFunction();
	findMeanStateQuat();
	updateCovarianceState();
	generateSigmaPoints();
	observationFunction();
	findMeanObservationQuat();
	updateCovarianceObservation();
	findKalmanGain();
	finalMean();
	updateQuat();
	finalCovariance();
}


/*
*************************************************************************
These functions make it easier to interface with this code from Python
```

```
**************************************************************************
*/

/*
Allows a python script to set the measurement vector, the magnetic dip angle,
and the time since the filter was last called, and then iterate the quaternion
based square root filter.
*/
void UKF::callFilter(double measurementData1, double measurementData2, double measurementData3,double
measurementData4, double measurementData5, double measurementData6, double measurementData7, double
measurementData8, double measurementData9, double timeCalled, double magx, double magy, double magz) {//,
double callTime) {
      gsl_vector_set(measVec, 0, measurementData1);
      gsl_vector_set(measVec, 1, measurementData2);
      gsl_vector_set(measVec, 2, measurementData3);
      gsl_vector_set(measVec, 3, measurementData4);
      gsl_vector_set(measVec, 4, measurementData5);
      gsl_vector_set(measVec, 5, measurementData6);
      gsl_vector_set(measVec, 6, measurementData7);
      gsl_vector_set(measVec, 7, measurementData8);
      gsl_vector_set(measVec, 8, measurementData9);
      gsl_vector_set(magQuat, 1, magx);
      gsl_vector_set(magQuat, 2, magy);
      gsl_vector_set(magQuat, 3, magz);
      newTime = timeCalled;
      qsrukf();
}


//Retrieves an element of the state vector
double UKF::getStateElement(int elementNumber) {

      return gsl_vector_get(stateMean, elementNumber);
}

//Retrieves an element of the orientation quaternion
double UKF::getQuatStateElement(int elementNumber) {
```

```
	return gsl_vector_get(posQuat, elementNumber);
}

//Prints the noise matrices
void UKF::printNoise() {
	matPrint(measurementNoise);
	matPrint(processNoise);
}
//Sets an element of the orientation quaternion
void UKF::setQuatStateElement(int elementNumber, double value) {
	gsl_vector_set(posQuat, elementNumber, value);
}

//Sets an element of the measurement noise matrix
void UKF::setMeasurementNoise(int elementNumber, double value) {
	gsl_matrix_set(measurementNoise, elementNumber, elementNumber, value);
}

//Sets an element of the process noise matrix
void UKF::setProcessNoise(int elementNumber, double value) {
	gsl_matrix_set(processNoise, elementNumber, elementNumber, value);
}


/*
*****************************************************
Vector and matrix display utilities
*****************************************************
*/


//Display a matrix on screen (for debugging only)
void UKF::matPrint(gsl_matrix *m) {
	int i;
	for (i=0; i<m->size1; i++) {
		for (int j=0; j<m->size2; j++) {
			cout << gsl_matrix_get(m, i, j) << ' ';
		}
```

```
            cout << '\n';
      }
      cout << '\n';
}

//Display a vector on screen (for debugging only)
void UKF::vecPrint(gsl_vector *v) {
      int i;
      for (i=0; i<v->size; i++) {
            cout << gsl_vector_get(v, i) << ' ';
      }
      cout << '\n';
}


/*
************************************************************
Basic Mathematical functions
************************************************************
*/

//Perform quaternion multiplication on qLeft and qRight, leaving the result in qLeft
void UKF::quatMult(gsl_vector *qLeft, gsl_vector *qRight)
{
      double qLeftW, qLeftX, qLeftY, qLeftZ;
      double qRightW, qRightX, qRightY, qRightZ;
      qLeftW=gsl_vector_get(qLeft, 0);
      qLeftX=gsl_vector_get(qLeft, 1);
      qLeftY=gsl_vector_get(qLeft, 2);
      qLeftZ=gsl_vector_get(qLeft, 3);
      qRightW=gsl_vector_get(qRight, 0);
      qRightX=gsl_vector_get(qRight, 1);
      qRightY=gsl_vector_get(qRight, 2);
      qRightZ=gsl_vector_get(qRight, 3);
    gsl_vector_set(tempQuat, 0, qLeftW*qRightW - qLeftX*qRightX - qLeftY*qRightY - qLeftZ*qRightZ);
      gsl_vector_set(tempQuat, 1, qLeftW*qRightX + qLeftX*qRightW + qLeftY*qRightZ - qLeftZ*qRightY);
      gsl_vector_set(tempQuat, 2, qLeftW*qRightY + qLeftY*qRightW + qLeftZ*qRightX - qLeftX*qRightZ);
      gsl_vector_set(tempQuat, 3, qLeftW*qRightZ + qLeftZ*qRightW + qLeftX*qRightY - qLeftY*qRightX);
```

```
	gsl_vector_memcpy(qLeft, tempQuat);
}

//Invert qInitial, and leave the result in qInverse
//(only works if the quaternion is normalized)
void UKF::quatInverse(gsl_vector *qInverse, gsl_vector *qInitial)
{
	gsl_vector_set(tempQuat, 0, 1);
	gsl_vector_set(tempQuat, 1, -1);
	gsl_vector_set(tempQuat, 2, -1);
	gsl_vector_set(tempQuat, 3, -1);
	gsl_vector_memcpy(qInverse, qInitial);
	gsl_vector_mul(qInverse, tempQuat);
}

//Normalize a quaternion
void UKF::quatNormalize(gsl_vector *quat) {
	double quatW, quatX, quatY, quatZ, norm;
	quatW = gsl_vector_get(quat, 0);
	quatX = gsl_vector_get(quat, 1);
	quatY = gsl_vector_get(quat, 2);
	quatZ = gsl_vector_get(quat, 3);
	norm = 1/sqrt((quatW*quatW)+(quatX*quatX)+(quatY*quatY)+(quatZ*quatZ));
	gsl_vector_scale(quat, norm);
}

//Perform a cholesky factor downdate
//Ported from the LINPACK FORTRAN function DCHDD
int UKF::dchdd(int p, gsl_vector *x, gsl_vector *c, gsl_vector *s, gsl_matrix *r) {
	  int info; //ldr,ldz,nz;
	  int i,ii,j,k;
	double alpha, norm, a; //azeta,dnrm2;
	double t, xx, scale, b; //ddot,zeta;
	  double tempVar;
	  double rvectemp[20];
	  double sVectemp[20];
	  double cVectemp[20];
	  info = 0;
```

```
sVectemp[0] = gsl_vector_get(x,0)/gsl_matrix_get(r, 0, 0);
if (p>=2) {
      for (j=2; j<=p; j++) {
            for (k=0; k<p; k++) {
                  rvectemp[k]=gsl_matrix_get(r,k,j-1);
            }
            sVectemp[j-1] = gsl_vector_get(x, j-1) - cblas_ddot(j-1,rvectemp,1,sVectemp,1);
            sVectemp[j-1] = sVectemp[j-1]/gsl_matrix_get(r, j-1,j-1);
      }
}
for (k=0; k<p; k++) {
}
norm = cblas_dnrm2(p, sVectemp, 1);
if (norm<1.0) {
      alpha = sqrt(1.0-norm*norm);
      for (ii=1; ii<=p; ii++) {
            i = p - ii + 1;
            scale = alpha + abs(sVectemp[i-1]);
            a = alpha/scale;
            b=sVectemp[i-1]/scale;
            norm=sqrt(a*a+b*b);
            cVectemp[i-1] = a/norm;
            sVectemp[i-1] = b/norm;
            alpha = scale*norm;
      }
      for (j=1; j<=p; j++) {
            xx = 0;
            for (ii=1; ii<=j; ii++) {
                  i = j-ii+1;
                  t=cVectemp[i-1]*xx+sVectemp[i-1]*gsl_matrix_get(r, i-1, j-1);
                  tempVar=cVectemp[i-1]*gsl_matrix_get(r,i-1,j-1)-sVectemp[i-1]*xx;
                  gsl_matrix_set(r, i-1, j-1, tempVar);
                  xx=t;
            }
      }

}
else info = -1;
```

```
        for (k=0; k<p; k++) {
              gsl_vector_set(s,k,sVectemp[k]);
              gsl_vector_set(c,k, cVectemp[k]);
        }
        return info;

}

//Perform a cholesky factor update
//Ported from the LINPACK FORTRAN function DCHUD
void UKF::dchud(int p, gsl_vector *x, gsl_vector *c, gsl_vector *s, gsl_matrix *r) {
    int j, i, jm1;
    double t, xj, rtmp, ctmp, stmp;
    for (j=1; j<=p; j++) {
            xj = gsl_vector_get(x, j-1);
        jm1 = j-1;
            if (jm1>=1) {
                  for (i=1; i<=jm1; i++) {
                  t = gsl_vector_get(c, (i-1))*gsl_matrix_get(r,i-1,j-1) + gsl_vector_get(s,(i-1))*xj;
            xj = gsl_vector_get(c,(i-1))*xj - gsl_vector_get(s,(i-1))*gsl_matrix_get(r,i-1,j-1);
            gsl_matrix_set(r,i-1,j-1,t);
                  }
            }
            rtmp = gsl_matrix_get(r, j-1, j-1);
            ctmp = gsl_vector_get(c, j-1);
            stmp = gsl_vector_get(s, j-1);
            cblas_drotg(&rtmp,&xj,&ctmp,&stmp);
            gsl_matrix_set(r, j-1, j-1, rtmp);
            gsl_vector_set(c, j-1, ctmp);
            gsl_vector_set(s, j-1, stmp);
      }
}


/*
*******************************************************
Models used in filter
*******************************************************
```

```
*/


//Update the state vector in a quaterion square root UKF
void UKF::stateFunction() {
      int i;
      double angV1, angV2, angV3, angMag, axis1, axis2, axis3, deltaAngle;
      double angA1, angA2, angA3;
      double angE1, angE2, angE3;
      //Update all state elements as random walks (for now)
      gsl_matrix_memcpy(updStateBuf, stateBuf);

      //Iterate over all sigma points
      for (i=0; i<numberOfSigmaPoints; i++) {
            angA1 = gsl_matrix_get(stateBuf, 6, i);
            angA2 = gsl_matrix_get(stateBuf, 7, i);
            angA3 = gsl_matrix_get(stateBuf, 8, i);

            //Get angular velocity
            angV1 = gsl_matrix_get(stateBuf, 3, i);
            angV2 = gsl_matrix_get(stateBuf, 4, i);
            angV3 = gsl_matrix_get(stateBuf, 5, i);

            //Combine angular velocity with acceleration
            angV1=angV1+angA1*newTime;
            angV2=angV2+angA2*newTime;
            angV3=angV3+angA3*newTime;

            //Create a perturbation angle
            angE1=angV1*newTime+0.5*angA1*newTime*newTime;
            angE2=angV2*newTime+0.5*angA2*newTime*newTime;
            angE3=angV3*newTime+0.5*angA3*newTime*newTime;

            //Separate into magnitude and axis of rotation of a delta vector
            angMag = sqrt((angE1*angE1)+(angE2*angE2)+(angE3*angE3));
            deltaAngle = angMag;
            if (angMag!=0) {
                  axis1 = angE1/angMag;
```

```
      axis2 = angE2/angMag;
      axis3 = angE3/angMag;
}
else {
      axis1 = 0;
      axis2 = 0;
      axis3 = 0;
      deltaAngle=0;
}

//Translate into a delta quaternion
gsl_vector_set(angVQuat, 0, cos(deltaAngle/2));
gsl_vector_set(angVQuat, 1, sin(deltaAngle/2)*axis1);
gsl_vector_set(angVQuat, 2, sin(deltaAngle/2)*axis2);
gsl_vector_set(angVQuat, 3, sin(deltaAngle/2)*axis3);

//Get the error vector
angV1 = gsl_matrix_get(stateBuf, 0, i);
angV2 = gsl_matrix_get(stateBuf, 1, i);
angV3 = gsl_matrix_get(stateBuf, 2, i);

//Separate into magnitude and axis of rotation
angMag = sqrt((angV1*angV1)+(angV2*angV2)+(angV3*angV3));
deltaAngle = angMag;//*newTime;
if (angMag!=0) {
      axis1 = angV1/angMag;
      axis2 = angV2/angMag;
      axis3 = angV3/angMag;
}
else {
      axis1 = 0;
      axis2 = 0;
      axis3 = 0;
      deltaAngle=0;
}

//Form quaternion
gsl_vector_set(errQuat, 0, cos(deltaAngle/2));
```

```
            gsl_vector_set(errQuat, 1, sin(deltaAngle/2)*axis1);
            gsl_vector_set(errQuat, 2, sin(deltaAngle/2)*axis2);
            gsl_vector_set(errQuat, 3, sin(deltaAngle/2)*axis3);

            //Update position quaternion with error quaternion and change
            //due to angular velocity
            quatMult(angVQuat, errQuat);
            gsl_vector_memcpy(tempQuat, posQuat);
            quatMult(tempQuat, angVQuat);
            gsl_matrix_set_col(quatBuf, i, tempQuat);
      }
}



//Update the observation vector for a quaternion based square root UKF
void UKF::observationFunction() {
      double errorangle;
      double cent, norm;
      int i;

      //Back up posQuat
      gsl_vector_memcpy(tempPosQuat, posQuat);

      //Transfer angular velocity directly to the observation buffer
      for (i=3; i<6; i++) {
            gsl_matrix_get_row(rowVector, stateBuf, i);
            gsl_matrix_set_row(obsSigBuf, i+3, rowVector);
      }

      //Iterate over all sigma points
      for (i=0; i<numberOfSigmaPoints; i++) {

            //Back up reference vectors
            gsl_vector_memcpy(tempPosQuat, posQuat);
            gsl_vector_memcpy(tempMagQuat, magQuat);
            gsl_vector_memcpy(tempGravQuat, gravQuat);
```

```
//Get error vector
gsl_vector_set(rotError, 0, gsl_matrix_get(stateBuf, 0, i));
gsl_vector_set(rotError, 1, gsl_matrix_get(stateBuf, 1, i));
gsl_vector_set(rotError, 2, gsl_matrix_get(stateBuf, 2, i));

//Separate into magnitude and axis of rotation
errorangle = gsl_blas_dnrm2(rotError);
if (errorangle!=0) {
      gsl_vector_scale(rotError, 1.0/errorangle);
}

//Assemble error quaternion
gsl_vector_set(errQuat, 0, cos(errorangle/2));
gsl_vector_set(errQuat, 1, gsl_vector_get(rotError, 0)*sin(errorangle/2));
gsl_vector_set(errQuat, 2, gsl_vector_get(rotError, 1)*sin(errorangle/2));
gsl_vector_set(errQuat, 3, gsl_vector_get(rotError, 2)*sin(errorangle/2));

//Adjust orientation quaternion
quatMult(tempPosQuat, errQuat);
norm = gsl_blas_dnrm2(tempPosQuat);
if (norm!=0) {
      gsl_vector_scale(tempPosQuat, 1.0/norm);
}

//Get inverse of orientation quaternion
quatInverse(posQuatInv, tempPosQuat);

//calculate centripetal acceleration
gsl_vector_set(centAccel, 1, gsl_matrix_get(stateBuf, 3, i));
gsl_vector_set(centAccel, 2, gsl_matrix_get(stateBuf, 4, i));
gsl_vector_set(centAccel, 3, gsl_matrix_get(stateBuf, 5, i));
cent = gsl_blas_dnrm2(centAccel);
if (cent!=0) {
      gsl_vector_scale(centAccel, 1.0/cent);
}
cent = cent*cent;
cent = cent/500.0;
cent = cent/9.8;
```

```
gsl_vector_scale(centAccel, cent);
//cent = gsl_vector_get(centAccel, 2);
gsl_vector_set(centAccel, 2, cent/50.0);
//gsl_vector_scale(centAccel, 0);


//Rotate gravity vector with orientation quaternion
//to predict accelerometer values
quatMult(tempGravQuat, posQuatInv);
gsl_vector_memcpy(gravNewQuat, tempPosQuat);
quatMult(gravNewQuat, tempGravQuat);
//cout << "centaccel\n";
//vecPrint(centAccel);
//gsl_vector_add(gravNewQuat, centAccel);
gsl_matrix_set(obsSigBuf, 0, i, gsl_vector_get(gravNewQuat, 1));
gsl_matrix_set(obsSigBuf, 1, i, gsl_vector_get(gravNewQuat, 2));
gsl_matrix_set(obsSigBuf, 2, i, gsl_vector_get(gravNewQuat, 3));

//Rotate magnetic field vector with orientation quaternion
//to predict magnetometer values
quatMult(tempMagQuat, posQuatInv);
gsl_vector_memcpy(magNewQuat, tempPosQuat);
quatMult(magNewQuat, tempMagQuat);
gsl_matrix_set(obsSigBuf, 3, i, gsl_vector_get(magNewQuat, 1));
gsl_matrix_set(obsSigBuf, 4, i, gsl_vector_get(magNewQuat, 2));
gsl_matrix_set(obsSigBuf, 5, i, gsl_vector_get(magNewQuat, 3));

//Rotate angular momentum vector with orientation quaternion
//to predict gyroscope values
gsl_vector_set(tempAngMomQuat, 0, 0);
gsl_vector_set(tempAngMomQuat, 1, gsl_matrix_get(stateBuf, 3, i));
gsl_vector_set(tempAngMomQuat, 2, gsl_matrix_get(stateBuf, 4, i));
gsl_vector_set(tempAngMomQuat, 3, gsl_matrix_get(stateBuf, 5, i));
quatMult(tempAngMomQuat, posQuatInv);
gsl_vector_memcpy(angMomNewQuat, tempPosQuat);
quatMult(angMomNewQuat, tempAngMomQuat);

gsl_matrix_set(obsSigBuf, 6, i, gsl_vector_get(angMomNewQuat, 1));
```

```
            gsl_matrix_set(obsSigBuf, 7, i, gsl_vector_get(angMomNewQuat, 2));
            gsl_matrix_set(obsSigBuf, 8, i, gsl_vector_get(angMomNewQuat, 3));
      }
}


//Update the orientation quaternion
void UKF::updateQuat() {
      double errorangle, norm;

      //Get the final estimated error
      gsl_vector_set(rotError, 0, gsl_vector_get(stateMean, 0));
      gsl_vector_set(rotError, 1, gsl_vector_get(stateMean, 1));
      gsl_vector_set(rotError, 2, gsl_vector_get(stateMean, 2));
      errorangle = gsl_blas_dnrm2(rotError);
      //while (errorangle>(2*PI) && (errorangle<100)) {
      //    errorangle=errorangle-(2*PI);
      //}
      //double errnorm;
      //errnorm = gsl_blas_dnrm2(errQuat);
      //cout << "errorangle " << errorangle << "\n";
      //errorangle=errorangle*-1;
      if (errorangle!=0) {
            gsl_vector_scale(rotError, 1.0/errorangle);
      }

      //Assemble quaternion
      gsl_vector_set(errQuat, 0, cos(errorangle/2));
      gsl_vector_set(errQuat, 1, gsl_vector_get(rotError, 0)*sin(errorangle/2));
      gsl_vector_set(errQuat, 2, gsl_vector_get(rotError, 1)*sin(errorangle/2));
      gsl_vector_set(errQuat, 3, gsl_vector_get(rotError, 2)*sin(errorangle/2));
      double errnorm;
      errnorm = gsl_blas_dnrm2(errQuat);
      //cout << "errnorm " << errnorm << "\n";
      //Rotate old quaternion to form best guess of new quaternion
      quatMult(posQuat, errQuat);
      norm = gsl_blas_dnrm2(posQuat);
      //cout << "norm " << norm << "\n";
```

```
	if (norm!=0) {
		gsl_vector_scale(posQuat, 1.0/norm);
	}
	gsl_vector_set(stateMean, 0, 0);
	gsl_vector_set(stateMean, 1, 0);
	gsl_vector_set(stateMean, 2, 0);
}


//Generate a set of sigma points from a state vector
void UKF::generateSigmaPoints() {
	int i;

	//Arrange state vectors in columns
	for (i=0; i<numberOfSigmaPoints; i++) {
		gsl_matrix_set_col(stateBuf, i, stateMean);
	}
	gsl_matrix_set_col(sigBuf, 0, zeroVec);

	//Arrange covariance in sigBuf
	for (i=0; i<STATELENGTH; i++) {
		gsl_matrix_get_col(tempVec, cov, i);
		gsl_matrix_set_col(sigBuf, i+1, tempVec);
		gsl_matrix_get_col(tempVec, cov, i);
		gsl_vector_scale(tempVec, -1);
		gsl_matrix_set_col(sigBuf, i+STATELENGTH+1, tempVec);
	}

	//Introduce weights
	gsl_matrix_scale(sigBuf, gamma);

	//Combine buffers to finish making sigma points
	gsl_matrix_add(sigBuf, stateBuf);
	gsl_matrix_memcpy(stateBuf, sigBuf);
}


//Calculate the mean of a set of sigma points
```

```
void UKF::findMeanState() {
	int i;
	gsl_matrix_memcpy(wUpdStateBuf, updStateBuf);

	//Weight sigma point vectors
	gsl_matrix_mul_elements(wUpdStateBuf, mWeights);
	gsl_matrix_get_col(stateMean, wUpdStateBuf, 0);

	//Sum up points
	for (i=1; i<numberOfSigmaPoints; i++) {
		gsl_matrix_get_col(tempVec, wUpdStateBuf, i);
		gsl_vector_add(stateMean, tempVec);
	}
}


//Calculate the mean of a set of sigma point quaternions
void UKF::findMeanStateQuat() {
	int i, j;
	double angV1, angV2, angV3, angMag, axis1, axis2, axis3, deltaAngle, errorangle, norm;

	gsl_vector_memcpy(tempPosQuat, posQuat);
	gsl_matrix_memcpy(quatErrBuf, quatBuf);
	quatInverse(posQuatInv, tempPosQuat);

	//Iterate over sigma points
	for (i=0; i<numberOfSigmaPoints; i++) {
		gsl_matrix_get_col(tempQuat, quatErrBuf, i);
		quatMult(tempQuat, posQuatInv);
		quatNormalize(tempQuat);
		double errang;
		errang = gsl_vector_get(tempQuat, 0);
		errorangle = 2*acos(errang);
		gsl_vector_set(tempQuat, 0, 0);
		norm = gsl_blas_dnrm2(tempQuat);
		if (norm!=0) {
			gsl_vector_scale(tempQuat, 1.0/norm);
		}
```

```
      gsl_matrix_set(updStateBuf, 0, i, gsl_vector_get(tempQuat, 1)*errorangle);
      gsl_matrix_set(updStateBuf, 1, i, gsl_vector_get(tempQuat, 2)*errorangle);
      gsl_matrix_set(updStateBuf, 2, i, gsl_vector_get(tempQuat, 3)*errorangle);
}
gsl_matrix_memcpy(wUpdStateBuf, updStateBuf);

//Alter weights for quaternion use
for (i=0; i<numberOfSigmaPoints; i++) {
      for (j=0; j<3; j++) {
            gsl_matrix_set(mWeights, j, i, 1.0/numberOfSigmaPoints);
      }
}

//Weight error vectors
gsl_matrix_mul_elements(wUpdStateBuf, mWeights);
gsl_matrix_get_col(stateMean, wUpdStateBuf, 0);

//Sum over vectors and angular velocity sigma points
for (i=1; i<numberOfSigmaPoints; i++) {
      gsl_matrix_get_col(tempVec, wUpdStateBuf, i);
      gsl_vector_add(stateMean, tempVec);
}

//Get error vector from stateMean
angV1 = gsl_vector_get(stateMean, 0);
angV2 = gsl_vector_get(stateMean, 1);
angV3 = gsl_vector_get(stateMean, 2);
angMag = sqrt((angV1*angV1)+(angV2*angV2)+(angV3*angV3));
deltaAngle = angMag;
if (angMag!=0) {
      axis1 = angV1/angMag;
      axis2 = angV2/angMag;
      axis3 = angV3/angMag;
}
else {
      axis1 = 0;
      axis2 = 0;
      axis3 = 0;
```

```
      deltaAngle = 0;
}

//Form quaternion
gsl_vector_set(tempQuat, 0, cos(deltaAngle/2));
gsl_vector_set(tempQuat, 1, sin(deltaAngle/2)*axis1);
gsl_vector_set(tempQuat, 2, sin(deltaAngle/2)*axis2);
gsl_vector_set(tempQuat, 3, sin(deltaAngle/2)*axis3);
quatInverse(tempPosQuat, tempQuat);
quatMult(posQuat, tempPosQuat);
quatInverse(tempPosQuat, posQuat);
gsl_vector_memcpy(posQuat, tempPosQuat);
norm = gsl_blas_dnrm2(posQuat);
if (norm!=0) {
      gsl_vector_scale(posQuat, 1.0/norm);
}
quatInverse(posQuatInv, posQuat);

//Iterating over gradient descent algorithm
for (j=0; j<10; j++) {
      gsl_matrix_memcpy(quatErrBuf, quatBuf);

      //Iterating over sigma pionts
      for (i=0; i<numberOfSigmaPoints; i++) {

            //Remove orientation quaternion from perturbed quaternions
            gsl_matrix_get_col(tempQuat, quatErrBuf, i);
            quatMult(tempQuat, posQuatInv);
            quatNormalize(tempQuat);

            //Form error vectors from quaternions
            errorangle = 2*acos(gsl_vector_get(tempQuat, 0));
            gsl_vector_set(tempQuat, 0, 0);
            norm = gsl_blas_dnrm2(tempQuat);

            if (norm!=0) {
                  gsl_vector_scale(tempQuat, 1.0/norm);
            }
```

```
      gsl_vector_scale(tempQuat, errorangle);
      gsl_matrix_set_col(quatErrBuf, i, tempQuat);
}

//Sum error vectors
gsl_matrix_scale(quatErrBuf, 1.0/numberOfSigmaPoints);
gsl_matrix_get_col(tempPosQuat, quatErrBuf, 0);
for (i=1; i<numberOfSigmaPoints; i++) {
      gsl_matrix_get_col(tempQuat, quatErrBuf, i);
      gsl_vector_add(tempPosQuat, tempQuat);
}

//Get error quaternion from final error vector
angV1 = gsl_vector_get(tempPosQuat, 1);
angV2 = gsl_vector_get(tempPosQuat, 2);
angV3 = gsl_vector_get(tempPosQuat, 3);
angMag = sqrt((angV1*angV1)+(angV2*angV2)+(angV3*angV3));
deltaAngle = angMag;//*newTime;
if (angMag!=0) {
      axis1 = angV1/angMag;
      axis2 = angV2/angMag;
      axis3 = angV3/angMag;
}
else {
      axis1 = 0;
      axis2 = 0;
axis3 = 0;
deltaAngle = 0;
}
gsl_vector_set(tempQuat, 0, cos(deltaAngle/2));
gsl_vector_set(tempQuat, 1, sin(deltaAngle/2)*axis1);
gsl_vector_set(tempQuat, 2, sin(deltaAngle/2)*axis2);
gsl_vector_set(tempQuat, 3, sin(deltaAngle/2)*axis3);

//Adjust the mean, and iterative the algorithm again
quatMult(tempQuat, posQuat);
gsl_vector_memcpy(posQuat, tempQuat);
```

```
            quatInverse(posQuatInv, posQuat);
      }

      //Set updated state buffer
      for (i=0; i<3; i++) {
            gsl_matrix_get_row(rowVector, quatErrBuf, i+1);
            gsl_matrix_set_row(updStateBuf, i, rowVector);
      }

      //Set state mean
      gsl_vector_set(stateMean, 0, gsl_vector_get(tempPosQuat,1));
      gsl_vector_set(stateMean, 1, gsl_vector_get(tempPosQuat,2));
      gsl_vector_set(stateMean, 2, gsl_vector_get(tempPosQuat,3));
      gsl_matrix_set_col(updStateBuf, 0, stateMean);
}


//Find the mean observation vector
void UKF::findMeanObservation() {
      int i;
      gsl_matrix_memcpy(wObsSigBuf, obsSigBuf);

      //Weight observation buffer
      gsl_matrix_mul_elements(wObsSigBuf, obsMWeights);
      gsl_matrix_get_col(obsPredMean, wObsSigBuf, 0);

      //Sum over observation sigma points
      for (i=1; i<numberOfSigmaPoints; i++) {
            gsl_matrix_get_col(obsTempVec, wObsSigBuf, i);
            gsl_vector_add(obsPredMean, obsTempVec);
      }
}


//Find the mean observation vector using quaternions (under construction)
void UKF::findMeanObservationQuat() {
      int i;
      gsl_matrix_memcpy(wObsSigBuf, obsSigBuf);
```

```
	//Weight observation buffer
	gsl_matrix_mul_elements(wObsSigBuf, obsMWeights);
	gsl_matrix_get_col(obsPredMean, wObsSigBuf, 0);

	//Sum over observation sigma points
	for (i=1; i<numberOfSigmaPoints; i++) {
		gsl_matrix_get_col(obsTempVec, wObsSigBuf, i);
		gsl_vector_add(obsPredMean, obsTempVec);
	}

	double magmag, magx, magy, magz, accx, accy, accz;

	//Retrieve accelerometer and magnetic field sensor values
	accx=gsl_vector_get(obsPredMean, 0);
	accy=gsl_vector_get(obsPredMean, 1);
	accz=gsl_vector_get(obsPredMean, 2);
	magx=gsl_vector_get(obsPredMean, 3);
	magy=gsl_vector_get(obsPredMean, 4);
	magz=gsl_vector_get(obsPredMean, 5);

	//Normalize magnetic field sensors values
	magmag=1;//sqrt(magx*magx+magy*magy+magz*magz);
	gsl_vector_set(obsPredMean, 0, accx);
	gsl_vector_set(obsPredMean, 1, accy);
	gsl_vector_set(obsPredMean, 2, accz);
	gsl_vector_set(obsPredMean, 3, magx/magmag);
	gsl_vector_set(obsPredMean, 4, magy/magmag);
	gsl_vector_set(obsPredMean, 5, magz/magmag);

}


//Update the covariance of the state vector
void UKF::updateCovarianceState() {
	int i, dchddfail;

	//Prepare for QR Decomposition
```

```
	for (i=0; i<(2*STATELENGTH); i++) {
		gsl_matrix_get_col(tempVec, updStateBuf, i+1);
		gsl_vector_sub(tempVec, stateMean);
		gsl_vector_scale(tempVec, sqrtCovWeightOne);
		gsl_matrix_set_row(QRBuf, i, tempVec);
	}
	for (i=0; i<STATELENGTH; i++) {
		gsl_matrix_get_col(tempVec, processNoise, i);
		gsl_matrix_set_row(QRBuf, i+STATELENGTH*2, tempVec);
	}

	//Perform QR Decomposition
	gsl_linalg_QR_decomp(QRBuf, tempVec);
	for (i=0; i<STATELENGTH; i++) {
		gsl_matrix_get_row(tempVec, QRBuf, i);
		gsl_matrix_set_row(cov, i, tempVec);
	}

	//Prepare for cholesky factor updating/downdating
	gsl_matrix_mul_elements(cov, makeUpperTriangle);
	gsl_vector_memcpy(sqCWMean, stateMean);
	gsl_matrix_get_col(tempVec, updStateBuf, 0);
	gsl_vector_sub(tempVec, sqCWMean);
	gsl_vector_memcpy(sqCWMean, tempVec);
	gsl_vector_scale(sqCWMean, sqrtCovWeightZero);

	//Perform cholesky factor updating/downdating
	if (covWeightZero>=0) dchud(STATELENGTH, sqCWMean, cVec, sVec, cov);
	else {
		dchddfail = dchdd(STATELENGTH,stateMean, cVec, sVec, cov);
		cout << "downdating\n";
		if (dchddfail != 0) cout << "UPDATE FAILED!\n";
	}
}


void UKF::updateCovarianceObservation() {
	int i, dchddfail;
```

```
	//Prepare for QR Decomposition
	for (i=0; i<(2*STATELENGTH); i++) {
		gsl_matrix_get_col(obsTempVec, obsSigBuf, i+1);
		gsl_vector_sub(obsTempVec, obsPredMean);
		gsl_vector_scale(obsTempVec, sqrtCovWeightOne);
		gsl_matrix_set_row(obsQRBuf, i, obsTempVec);
	}
	for (i=0; i<MEASUREMENTLENGTH; i++) {
		gsl_matrix_get_col(obsTempVec, measurementNoise, i);
		gsl_matrix_set_row(obsQRBuf, i+(2*STATELENGTH), obsTempVec);
	}

	//Perform QR Decomposition
	gsl_linalg_QR_decomp(obsQRBuf, obsTempVec);

	for (i=0; i<MEASUREMENTLENGTH; i++) {
		gsl_matrix_get_row(obsTempVec, obsQRBuf, i);
		gsl_matrix_set_row(obsCov, i, obsTempVec);
	}
	//Prepare for cholesky factor updating/downdating

	gsl_matrix_mul_elements(obsCov, obsMakeUpperTriangle);
	gsl_vector_memcpy(obsSqCWMean, obsPredMean);
	gsl_matrix_get_col(obsTempVec, obsSigBuf, 0);
	gsl_vector_sub(obsTempVec, obsSqCWMean);
	gsl_vector_memcpy(obsSqCWMean, obsTempVec);
	gsl_vector_scale(obsSqCWMean, sqrtCovWeightZero);

	//Perform cholesky factor updating/downdating
	if (covWeightZero>=0) dchud(MEASUREMENTLENGTH, obsSqCWMean, obsCVec, obsSVec, obsCov);
	else {
		dchddfail = dchdd(MEASUREMENTLENGTH,obsSqCWMean, obsCVec, obsSVec, obsCov);
		if (dchddfail != 0) cout << "UPDATE FAILED!\n";
	}
}
```

```
//Calculate the Kalman gain
void UKF::findKalmanGain() {
      int i;

      //Prepare state and observation buffers
      for (i=0; i<numberOfSigmaPoints; i++) {
            gsl_matrix_set_col(stateMeans, i, stateMean);
            gsl_matrix_set_col(obsMeans, i, obsPredMean);
      }

      //Prepare error buffers
      gsl_matrix_sub(sigBuf, stateMeans);
      gsl_matrix_sub(obsSigBuf, obsMeans);
      gsl_matrix_get_col(tempVec, sigBuf, 0);
      gsl_matrix_set_col(sigBufShort, 0, tempVec);
      gsl_matrix_get_col(obsTempVec, obsSigBuf, 0);
      gsl_matrix_set_col(obsSigBufShort, 0, obsTempVec);

      for (i=0; i<2*STATELENGTH; i++) {
            gsl_matrix_get_col(tempVec, sigBuf, i+1);
            gsl_matrix_set_col(sigBufLong, i, tempVec);
      }
      for (i=0; i<2*STATELENGTH; i++) {
            gsl_matrix_get_col(obsTempVec, obsSigBuf, i+1);
            gsl_matrix_set_col(obsSigBufLong, i, obsTempVec);
      }

      //Form intermediate covariance matrices
      gsl_blas_dgemm(CblasNoTrans, CblasTrans, 1, sigBufShort, obsSigBufShort, 0, PXYTemp);
      gsl_blas_dgemm(CblasNoTrans, CblasTrans, 1, sigBufLong, obsSigBufLong, 0, PXY);
      gsl_matrix_scale(PXY, covWeightOne);
      gsl_matrix_scale(PXYTemp, covWeightZero);
      gsl_matrix_add(PXY, PXYTemp);

      //Combine covariance matrices to form Kalman gain
      gsl_matrix_memcpy(obsRBuf, obsCov);
      gsl_matrix_transpose(obsRBuf);
```

```
	//Solve matrix right divide using back-substitution
	gsl_blas_dtrsm(CblasRight, CblasUpper, CblasNoTrans, CblasNonUnit, 1, obsCov, PXY);
	gsl_blas_dtrsm(CblasRight, CblasUpper, CblasNoTrans, CblasNonUnit, 1, obsRBuf, PXY);
	gsl_matrix_memcpy(kalmanGain, PXY);

}


//Subtract inovation, scale with Kalman gain, and form the final mean
void UKF::finalMean() {
	gsl_vector_memcpy(obsTempVec, measVec);
	//gsl_vector_sub(measVec, obsPredMean);
	gsl_vector_sub(obsTempVec, obsPredMean);
	//vecPrint(obsTempVec);
	//gsl_blas_dgemv(CblasNoTrans, 1, kalmanGain, measVec, 0, tempVec);
	gsl_blas_dgemv(CblasNoTrans, 1, kalmanGain, obsTempVec, 0, tempVec);
	gsl_vector_add(stateMean, tempVec);

}


//Perform repeated cholesky factor downdating and calculate the final covariance
void UKF::finalCovariance() {
	int i, dchddfail;
	gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1, kalmanGain, obsCov, 0, uBuf);
	for (i=0; i<MEASUREMENTLENGTH; i++) {
		gsl_matrix_get_col(tempVec, uBuf,i);
		dchddfail = dchdd(STATELENGTH,tempVec, cVec, sVec, cov);
//		cout << "DDCHDD=\n" << dchddfail << "\n";
	}
}
```

```
/*
*****************************************************
Test Program for the UKF class
David Sachs
Compiles on Microsoft Visual C++ 6.0


This is not actually used in practice, as the functions
are called directly from Python instead. This is a short
program that simply times the filter.
*****************************************************
*/

#include "UnscentedKalmanFilter.h"
#include <stdio.h>
#include <stdlib.h>
using namespace std;

void main()
{
      UKF ukf;
      ukf.initFilter();
      double result;
      double aKey;

      cout << "Timing filter\n";
      result = ukf.timeFilter(1000);
      cout << "The average iteration time is " << result << "\n";
      cout << "Press a key to continue\n";
      cin >> aKey;

}
```

# Bibliography

[BA02] Stephen Barrass and Matt Adcock. Interactive Granular Synthesis of Haptic Contact Sound. In *Proceedings of the international Conference on Virtual, Synthetic and Entertainment Audio,* Helsinki, 2002.

[Ben98] Ari Yosef Benbasat. An Inertial Measurement Unit for User Interfaces. Master's Thesis, MIT Media Lab, 1998.

[BH04] Joaquin A. Blaya and Hugh Herr. Adaptive Control of a Variable-Impedance Ankle-Foot Orthosis to Assist Drop-Foot Gait. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 12(1), 2004.

[BH97] Robert Grover Brown and Patrick Y. C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*, third edition. Wiley, New York, 1997.

[Blu05] BlueRadios website. www.blueradios.com, 2005.

[BP05] Ari Y. Benbasat, Joe A. Paradiso. A Compact Wireless Sensor Platform. In *Proceedings of the 2005 Symposium on Information Processing in Sensor Networks*, Los Angeles, 2005.

[Brl94] *English Braille: American Edition*, Braille Authority of North America, 1994. http://www.brl.org/ebae/

[Bur91] Jim Burns. Braille: A Birthday Look at it's Past, Present, and Future. Braille Monitor, 34(7), July/August 1991.

[Cad88] Cadoz, Claude. Instrumental Gesture and Musical Composition. In *Proceedings of the International Computer Music Conference*, San Francisco, 1988.

[CCB01] Roger W Cholewiak, Amy A Collins, and J. Christopher Brill.
Spatial Factors in Vibrotactile Pattern Perception. In *Proceedings of Eurohaptics*, 2001.

[Cha02] Angela Chang, Sile O'Modhrain, Rob Jacob, Eric Gunther, and Hiroshi Ishii. ComTouch: Design of a Vibrotactile Communication Device. In *Proceedings of. ACM DIS 2002 Designing Interactive Systems Conference*, ACM Press, 2002.

[Cha97] Joel Chadabe. *Electric Sound: The Past and Promise of Electronic Music*. Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[CM03] Crassidis, J. L. and Markley, F. L. Unscented Filtering for Spacecraft Attitude Estimation, Journal of Guidance, *Journal of Guidance, Control, and Dynamics*, Vol. 26, August 2003, pp. 536–542.

[Deb04] Thomas Debus, Tae-Jeong Jang, Pierre Dupont, and Robert Howe. Multi-Channel Vibrotactile Display for Teleoperated Assembly. *International Journal of Control, Automation, and Systems*, 2(3), September 2004.

[Dum99] Ildeniz Duman. Design, Implementation, and Testing of a Real-Time Software System for a Quaternion-Based Attitude Estimation Filter. Master's Thesis, Naval Postgraduate School, 1999.

[Dur98] Lisa J. K. Durbeck, Nicholas J. Macias, David M. Weinstein, Chris R. Johnson, John M. Hollerbach. SCIRun Haptic Display for Scientific Visualization. *Phantom Users Group Meeting,* Dedham, MA, September 1998.

[Fox96] E. Foxlin. A Complementary Separate-Bias Kalman Filter for Inertial Head-Tracking. In *Proc. IEEE VRAIS 96. IEEE Computer Society Press*, March-April 1996.

[Fra04] Jacob Fraden. *Handbook of Modorn Sensors*. Springer-Verlag, New York, 2004.
[Gel60] Frank A. Geldard. Some Neglected Possibilities of Communication. *Science,* 1960 May 27; 131, (3413): 1583-1588.

[GOS01] Francine Gemperle, Nathan Ota and Dan Siewiorek. Design of a Wearable Tactile Display. In *Proceedings of the Fifth International Symposium on Wearable Computers*, Zürich, Switzerland, Oct. 2001.

[Gun01] Eric Gunther. SkinScape: A Tool for Composition in the Tactile Modality. Master's Thesis, MIT, 2001.

[HAM98] A. J. Healey, E. P. An, D. B. Marco. On Line Compensation of Heading Sensor Bias for Low Cost AUVs. In *Proceedings IEEE AUV98 Workshop on Underwater Navigation*, Draper Laboratories, 1998.

[Has01] Leila Hasan. Visual Frets for a Free-Gesture Musical Interface. Master's Thesis, MIT, 2001.

[Hon05] Honeywell Magnetic Field Sensors website. www.ssec.honeywell.com/magnetic/, 2005.

[Hum05] Humanware website. http://www.humanware.com/, 2005.

[Imm05] Immersion website, www.immersion.com, 2005.

[Ise05] Intersense website, www.isense.com, 2005.

[JB02] Lynette A. Jones and Michal Berris. The Psychophysics of Temperature Perception. In *Proceedings of the 10th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 2002.

[JB03] Lynette A. Jones and Michal Berris. Material Discrimination and Thermal Perception. In *Proceedings of the 10th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 2003.

[Kra03] Edgar Kraft, A Quaternion-based Unscented Kalman Filter for Orientation Tracking, In *Proceedings of the 6th Int. Conf. on Information Fusion*, Cairns, Australia, July 2003,

[Kaj01] Hiroyuki Kajimoto, Naoki Kawakami, Taro Maeda, Susumu Tachi. *Electrocutaneous Display as an Interface to a Virtual Tactile World*. In Proceedings of IEEE-VR, 2001.

[Kaj01a] Hiroyuki Kajimoto, Naoki Kawakami, Taro Maeda, Susumu Tachi. Electro-tactile display with force feedback. In *Proceedings of World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, 2001.

[Kaj03] Hiroyuki Kajimoto, Masahiko Inami, Naoki Kawakami, Susumu Tachi. SmartTouch: Augmentation of skin sensation with electrocutaneous display. In *Proceedings of the 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Los Angeles, 2003.

[Kaj04] Hiroyuki Kajimoto, Masahiko Inami, Naoki Kawakami, Susumu Tachi. SmartTouch: Electric Skin to Touch the Untouchable. *Computer Graphics and Applications*, IEEE, 24(1), Jan-Feb 2004.

[Kaj99] Hiroyuki Kajimoto, Naoki Kawakami, Taro Maeda, Susumu Tachi. Tactile Feeling Display using Functional Electrical Stimulation. In *Proceedings of The Ninth International Conference on Artificial Reality and Telexistence*, 1999.

[Kes05] Alicia "Kestrell" Verlager, Personal Communication, 2005.

[KKT02] Hiroyuki Kajimoto, Naoki Kawakami, Susumu Tachi. Optimal Design Method for Selective Nerve Stimulation and Its Application to Electrocutaneous Display. In *Proceedings of the Tenth Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Orlando, 2002.

[KTB97] Kurt A. Kaczmarek, Mitchell E. Tyler, Paul Bach-y-Rita. Pattern Identification on a Fingertip-Scanned Electrotactile Display. In Proceedings of the 19th Annual Int. Conf. IEEE Eng. Med. Biol. Soc., Chicago, IL, 1997.

[Lav03] Joseph LaViola. A Comparison of Unscented and Extended Kalman Filtering for Estimating Quaternion Motion. In *the Proceedings of the 2003 American Control Conference, IEEE Press, 2435-2440*, June 2003.

[Lin04] Robert W. Lindeman, John L. Sibert, Corinna E. Lathan, Jack M. Vice. The Design and Deployment of a Wearable Vibrotactile Feedback System. In

*Proceedings of the Eighth International Symposium on Wearable Computers*, 2004.

[LL86] Jack M. Loomis and Susan J. Lederman. Tactual Perception. In L. Kaufman and J. Thomas (Eds.) *Handbook of Human Perception and Performance*, Wiley, NY, 1986.

[Mac92] Tod Machover. Hyperinstruments: A Progress Report 1987-1991. Technical report, MIT Media Laboratory, 1992.

[Mac96] Tod Machover. *Brain Opera*, 1996.

[Mar01] João Luís Marins, Xiaoping Yun, Eric R. Bachmann, Robert B. McGhee, and Michael J. Zyda. An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hawaii, 2001.

[Mod00] Sile O'Modhrain. Playing by Feel: Incorporating Haptic Feedback into Computer-Based musical Instruments. PhD Thesis, Stanford University, 2000.

[Mod04] Sile O'Modhrain, 2004. Personal communication.

[MB05] Eduardo Miranda, Andrew Brouse. Toward Direct Brain-Computer Musical Interfaces. In *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression*, Vancouver, BC, Canada , 2005.

[MBA01] Louise Moody, Chris Baber, Theodoros N. Arvanitis. The Role of Haptic Feedback in the Training and Assessment of Surgeons using a Virtual Environment. In *Proceedings of Eurohaptics*, 2001.

[MJ04] R. van der Merwe, E. A. Wan & S. Julier, "Sigma-Point Kalman Filters for Nonlinear Estimation and Sensor-Fusion: Applications to Integrated Navigation", In Proceedings of the *AIAA Guidance, Navigation & Control Conference (GNC)*, Providence, Rhode Island, August 2004.

[Mar00] Teresa Marrin. Inside the conductor's jacket: analysis, interpretation and musical synthesis of expressive gesture. PhD Thesis, MIT, 2000.

[MHK00] Danilo Mandic, Richard Harvey, and Djemal H. Kolonic. On the choice of tactile code. *In Proceedings of the IEEE International Conference on Multimedia and Expo*, New York, 2000.

[MW01] Rudolph van der Merwe and Eric A. Wan. Efficient Derivative-Free Kalman Filters for Online Learning, *In ESANN'2001 proceedings - European Symposium on Artificial Neural Networks*, Belgium, 2001.

[OP02] Roberto Oboe, Giovanni De Poli. Multi-instrument virtual keyboard – The MIKEY project. In *Proceedings of the 2002 Conference on New Instruments for Musical Expression (NIME-02),* Dublin, Ireland, May 24-26, 2002.

[Opp99] A. Oppenheim. *Discrete-time Signal Processing*. Prentice-Hall, 2nd edition, 1999.

[Par03] Joseph A. Paradiso, Laurel S. Pardue, Kai-Yuh Hsiao, Ari Y. Benbasat. Electromagnetic Tagging for Electronic Music Interfaces. *Journal of New Music Research, Special issue on New Musical Interfaces*, Vol. 32, No. 4, December 2003.

[Pas04] Jerome Pasquero, Vincent L´evesque, Vincent Hayward, and Maryse Legault. Display of Virtual Braille Dots by Lateral Skin Deformation: A Pilot Study. In *Proceedings of Eurohaptics*, Munich, Germany, 2004.

[PG97] Joseph A. Paradiso and Neil Gershenfeld. Musical Applications of Electric Field Sensing. *Computer Music Journal*, 21(2), 1997.

[PJ05] Erin Piateski and Lynette Jones. Vibrotactile Pattern Recognition on the Arm and Torso. In *Proceedings of Eurohaptics*, 2005.

[Roa96] Curtis Roads. *Computer Music Tutorial*. MIT Press, Cambridge, Massachusetts, 1996.

[Rob00] J. Roberts, O. Slattery, D. Kardos. Rotating-Wheel Braille Display for Continuous Refreshable Braille. *Society for Information Display Conference in Long Beach, California*, 2000.

[RH00] Joseph Rovan and Vincent Hayward. Typology of Tactile Sounds and their Synthesis in Gesture-Driven Computer Music Performance. In *"Trends in Gestural Control of Music*". Wanderley, M., Battier, M. (eds). IRCAM, Paris, 2000.

[Riz05] Albert Rizzo, Margaret McLaughlin, Younbo Jung, Wei Peng, Shih-Ching Yeh, Weirong Zhu, and the USC/UT Consortium for Interdisciplinary Research. Virtual Therapeutic Environments with Haptics: An Interdisciplinary Approach for Developing Post-Stroke Rehabilitation Systems. In *Proceedings of The 2005 International Conference on Computers for People with Special Needs*, 2005.

[RLV03] Daniel Roetenberg, Henk Luinge and Peter Veltink. Inertial and magnetic sensing of human movement near ferromagnetic materials. *In Proceedings of the Second IEEE and ACM ISMAR*, 2003.

[Sen05] Sensable website. www.sensable.com, 2005.

[Sho94] K. Shoemake. *Quaternions*. University of Pennsylvania, Philadelphia, PA, 1994.

[Str21] Igor Stravinsky. *Rite of Spring*, 1921.

[Tal05] Talking Lights website. www.talking-lights.com, 2005.

[Tan96] Hong Zhang Tan. Information Transmission with a Multi-Finger Tactual Display. Phd Thesis, MIT, 1996.

[TH03] Michael S. Triantafyllou and Franz S. Hover. *Maneuvering and Control of Marine Vehicles*. OpenCourseWare, MIT, 2003.

[Tur98] Michael L. Turner, Daniel H. Gomez, Marc R. Tremblay and Mark R. Cutkosky. Preliminary Tests of an Arm-Grounded Haptic Feedback Device in Telemanipulation. In *Proceedings of the ASME IMECE Haptics Symposium*. Anaheim, CA. Nov. 1998.

[TY04] Koji Tsukada and Michiaki Yasumura. ActiveBelt: Belt-type Wearable Tactile Display for Directional Navigation. In *Proceedings of the International Conference on Ubiquitous Computing,* 2004.

[WM01] E. A. Wan and R. van der Merwe. The Unscented Kalman Filter. In . S. Haykin, editor, *Kalman Filtering and Neural Networks*, Wiley Publishing, 2001.

[Xen72] Iannis Xenakis. *Polytope de Cluny*, 1972.

[ZM05] Paul Zarchan and Howard Musoff. *Fundamentals of Kalman Filtering: A Practical Approach*, Second Edition. American Institute of Aeronautics and Astronautics, Reston, Va, 2005.