An Inertial Measurement Unit for User Interfaces

by

Ari Yosef Benbasat

B.A.Sc., University of British Columbia (1998)

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 2000

Author_____

Program in Media Arts and Sciences
September 8, 2000

Certified by_____

Joseph A. Paradiso
Principal Research Scientist
MIT Media Laboratory
Thesis Supervisor

Accepted by_____

Stephen A. Benton
Chair, Department Committee on Graduate Students
Program in Media Arts and Sciences

An Inertial Measurement Unit for User Interfaces

by

Ari Yosef Benbasat

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on September 8, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

Inertial measurement components, which sense either acceleration or angular rate, are being embedded into common user interface devices more frequently as their cost continues to drop dramatically. These devices hold a number of advantages over other sensing technologies: they measure relevant parameters for human interfaces and can easily be embedded into wireless, mobile platforms. The work in this dissertation demonstrates that inertial measurement can be used to acquire rich data about human gestures, that we can derive efficient algorithms for using this data in gesture recognition, and that the concept of a parameterized atomic gesture recognition has merit. Further we show that a framework combining these three levels of description can be easily used by designers to create robust applications.

A wireless six degree-of-freedom inertial measurement unit (IMU), with a cubical form factor (1.25 inches on a side) was constructed to collect the data, providing updates at 15 ms intervals. This data is analyzed for periods of activity using a windowed variance algorithm, whose thresholds can be set analytically. These segments are then examined by the gesture recognition algorithms, which are applied on an axis-by-axis basis to the data. The recognized gestures are considered atomic (*i.e.* cannot be decomposed) and are parameterized in terms of magnitude and duration. Given these atomic gestures, a simple scripting language is developed to allow designers to combine them into full gestures of interest. It allows matching of recognized atomic gestures to prototypes based on their type, parameters and time of occurrence.

Because our goal is to eventually create stand-alone devices,the algorithms designed for this framework have both low algorithmic complexity and low latency, at the price of a small loss in generality. To demonstrate this system, the gesture recognition portion of `(void*)`: A Cast of Characters, an installation which used a pair of hand-held IMUs to capture gestural inputs, was implemented using this framework. This version ran much faster than the original version (based on Hidden Markov Models), used less processing power, and performed at least as well.

Thesis Supervisor: Joseph A. Paradiso
Title: Principal Research Scientist, MIT Media Laboratory

An Inertial Measurement Unit for User Interfaces

by

Ari Yosef Benbasat

The following people served as readers for this thesis:

Thesis Reader

Bruce M. Blumberg
Assistant Professor of Media Arts and Sciences
Asahi Broadcasting Corporation Career Development Professor
of Media Arts and Sciences

Thesis Reader

Paul A. DeBitetto
Program Manager
Special Operations and Land Robotics Group
Charles Stark Draper Laboratory

Acknowledgments

To **Joe Paradiso**, who brought me to the Lab when I knew nothing, and then provided me with his enthusiastic suggestions, directions and advice to support my growth. If I am one day a good engineer, much of it will be Joe's doing.

To **Bruce Blumberg** and **Paul DeBitteto**, my gracious readers, for their help and willingness to operate under deadline. Also, many thanks to Bruce for asking me to work on `(void*)`.

To all those who helped me along: **Matt Berlin** and **Jesse Gray**, who coded up the scripting system at the last moment and contributed to its structure; **Andy Wilson** for wonderful advice on gesture recognition; **Michael Patrick Johnson** for his patience in helping me with quaternion math; **Marc Downie** for his patience in helping me with virtually everything else; **Ari Adler**, who did some early work on this project; **Jacky Mallett**, my partner in crime in late-summer thesis preparation; and **Pamela Mukerji**, who wrote some of the embedded code and helped hunt down references and figures.

To **Ari Adler** and **Ara Knaian**, wonderful officemates who made the rough times at the Lab bearable and the good times better.

To the **Responsive Environments Group**, my good friends and home away from home.

To the **Synthetic Characters**, for use of their code base and their friendship. It was a joy to work on `(void*)` with them.

To the **Natural Sciences and Engineering Research Council**, for financial support.

And finally, to **my parents**, who have never spared any time, effort or love if it would make my life in any way better.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Inertial measurement components, which sense either translational acceleration or angular rate, are being embedded into common user interface devices more frequently. Examples include VFX1 virtual reality headtracking systems[1], the Gyro Mouse[2] (a wireless 3D pointer), and Microsoft's SideWinder tilt-sensing joystick[3]. Such devices hold a number of advantages over other sensing technologies such as vision systems and magnetic trackers: they are small and robust, and can be made wireless using a lightweight radio-frequency link.

However, in most cases, these inertial systems are put together in a very *ad hoc* fashion, where a small number of sensors are placed on known fixed axes, and the data analysis relies heavily on *a priori* information or fixed constraints. This requires a large amount of custom hardware and software engineering to be done for each application, with little reuse possible.

This dissertation proposes to solve this problem by developing a compact six degree-of-freedom inertial measurement unit (IMU) as well as an analysis and gesture recognition framework. My vision is that this IMU could easily be incorporated into almost any interface or device, and a designer would be able to quickly and easily specify the gestures to be detected and their desired response.

1.1 Historical Methods

1.1.1 Inertial Technologies

Inertial measurement devices have a very rich history[4]. The field began with motion-stabilized gunsights for ships and was later driven by guidance systems for aircraft and missiles (dating to the V2 rocket), providing a large body of work to draw on. Because of the relatively large cost, size, and power and processing requirements of these systems, they were previously not appropriate for human-computer interfaces and consumer applications. However, recent advances in micro-electromechanical systems (MEMS) and other microfabrication techniques have led to lower cost, more compact devices, while at the same time, the processing power of personal computers has been increasing exponentially. Therefore, it is now possible for inertial systems, which previously required large computers and large budgets, to reach end-users. The Intersense[5] inertial-acoustic tracking system is an example of a commercial product exploiting this new market.

Inertial tracking systems such as Intersense's are known as strapdown systems, because the sensors are fixed to the local frame of the instrumented object. Many of the early military applications were closed-loop systems, where the inertial sensors are mounted on a controlled gimballed platform which attempts to remain aligned with the world frame regardless of the motion of the body. Such systems can operate over a much smaller dynamic range and therefore provide higher accuracy, but they also tend to be fairly large and costly. Therefore, for low-cost human interface applications, open-loop strapdown systems are more appropriate.

Recent uses of inertial sensors in major products have tended toward the automotive regime. The first major application of MEMS accelerometers was as a cheap, reliable trigger mechanism for airbag deployment[1] and they have since been applied to active suspension control, as well as other applications[7]. Gyroscopes are most often used to provide turn rate information to four wheel steering systems to help the front and rear tires match speed[8]. They

[1] Interestingly enough, gyroscopes are being explored for the same purpose, this time with roof airbag deployment for vehicle roll[6].

have very recently been used to provide heading information for in-vehicle tracking systems (which obtain position from the speedometer or a Global Positioning System unit).

There are also a number of smaller systems which use inertial components for navigation. Pedometry can be done in an entirely feature-based fashion using accelerometer data and knowledge of the average rate of human foot falls[9]. A much more complex device is a suitcase-sized inertial measurement system, designed at Draper Laboratories, which can be carried through a building, and uses features such as the rate of foot falls, path crossing and others, along with some forward integration of the signals, to reconstruct the route taken[10].

1.1.2 Gesture-Based Systems

Computer vision is the sensing modality used in the vast majority of current gesture-based systems. Academic work done at the MIT Media Laboratory in this field will be described and as well as a commercial non-vision product.

The Pfinder system was designed to find human shapes in a video stream, separate them from the background, and then track the motion of the hands and head[11]. People are found by comparing a view of an empty room to that of an occupied one. Their shapes then are broken into a set of blobs which represent portions of the anatomy (torso, foot, hand, etc.), and are tracked using knowledge of Newtonian mechanics and limitations on human movement. This provides the raw data for gesture recognition, which was usually implemented using Hidden Markov Models[12]. Applications exploiting Pfinder include real-time recognition of American Sign Language[13] and the augmentation of dance performances[14].

Andrew Wilson's Ph.D. dissertation[15] describes a number of interesting vision-based models, culminating in a system built around Bayesian networks[16]. This system can both recognize trained gestures and learn new gestures in real-time, by using contextual information. It again uses a blob based representation of form and separates areas of interest from the background. This work provides the foundation for gestural human-computer interaction

by allowing the computer to learn how a user wishes to interact with it on the fly, rather than forcing the user to conform to a predesignated interface.

In terms of commercial products, one popular non-vision system is the Analogous Gypsy motion capture system[17]. It uses an exoskeleton to tracks absolute human motion in the body frame by measuring joint rotation. The system is very popular in user modeling, and its data can be used in gesture recognition.

Though these systems have proven performance, both in term of raw data and recognition rate, they require a significant infrastructure. The Gypsy is tethered and requires a bulky infrastructure; the vision-based systems need a number of fixed cameras to work, suffer from problems of occlusion, and require a large amount of processing power.

1.2 Prior Art

This project builds on previous projects from the Media Laboratory's Responsive Environments and Physics and Media groups. Its direct lineage can be traced to the Expressive Footwear[18] project, where a small printed circuit card instrumented with inertial sensors (gyroscopes, accelerometers, magnetic compass), among a number of others (sonar, pressure), was mounted on the side of a dance shoe to allow the capture of multi-modal information describing a dancer's movements. This data was then filtered for a number of specific features, such as toe-taps and spins, which were used to generate music on the fly via a computer running a simple musical mapping algorithm. While this system collected the appropriate data for its circumstance, it could not be used in an application with arbitrary orientation because the chosen set of sensors measure only along specific axes (as suggested by the constraints of the shoe and a dancer's movement). Further, the circuit card was too large for many applications. Therefore, it was decided to create a system that would contain the sensors necessary for full six degree-of-freedom inertial measurement. The system had to be compact and wireless, to allow the greatest range of possibilities. A prototype version of the hardware was built by the author and was used in the `(void*)`: A Cast of Characters installation, which was demonstrated at SIGGRAPH '99[19] and is described in section 7.2.

There are currently a number of six-degree-of-freedom systems commercially available, and several of them are targeted at either the high-end user interface market or the motion capture market. The Ascension Technology miniBird 500 [20] magnetic tracker is the smallest available at $10mm \times 5mm \times 5mm$ making it particularly easy to use. However, the closed loop nature of the sensor requires that it be wired, and the base unit is fairly cumbersome. The Intersense IS-900 inertial-acoustic system[21] offers excellent accuracy over a very large range, but requires a fair amount of infrastructure for the sonar grid (used in position tracking). Crossbow Technologies offers the DMU-6X inertial measurement unit[22] which has excellent accuracy, but is quite large ($> 30\,\text{in}^3$). Also, all these systems are fairly expensive and none match our specification in terms of ease of use (small, wireless, low-cost).

There are two major threads in the academic literature on the uses of inertial components as a gestural expression. The first is in the area of musical input. Sawada[23] presents an accelerometer-only system that can recognize ten gestures based on a fairly simple feature set. These gestures are then used to control a MIDI (Musical Instrument Digital Interface) instrument. The Brain Opera, a large-scale interactive musical exhibit produced by the Media Laboratory, included an inertially-instrumented baton as an interface[24], among a number of other interesting examples[25]. The Conductor's Jacket[26] uses a magnetic tracker and biological sensors to distill expressive information from a conductor's movement. In each of these cases, the gesture recognition techniques are very specific to the application at hand and are therefore difficult to generalize.

The second area is the use of inertial sensors as a stand-alone interface for palmtop computers. The Itsy system from Compaq[27] uses accelerometers both as a static input, with the user tilting the device to scroll images, and a dynamic input, where fanning the device zooms the image in or out. Speculative execution is used to distinguish between the two operations since the start of a fan looks like a tilt. Interesting design ideas and concepts for such devices are presented by Small and Ishii[28] and Fitzmaurice[29]. While interesting, the input spaces of devices such as the Itsy tend to be very small as they consider only orientation (either static or dynamic) as an input parameter.

This dissertation aims to greatly increase both the input space of inertial devices and the ease of use thereof, as is described in the next section.

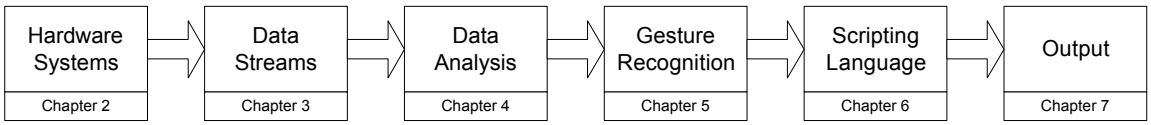

Figure 1-1: System/Document Organization

1.3 Project Goals

The overall goal of this research project is to explore the advantages of inertial sensing in gesture recognition-based applications. The claim made is that inertial sensors will provide a lower-cost, more robust and more flexible sensing modality than those currently in use. As a step towards proving this claim, a framework was designed and built to enable application designers to use inertial sensors with a minimum of knowledge and effort.

The first step in the framework is the inertial measurement unit (IMU), an electronics package that provides three axes each of acceleration and rotation sensing, as well as a microcontroller for data collection and processing, and a wireless link to transmit the data to an external computer. This data is then passed through an analysis algorithm to prepare the data for gesture recognition. This analysis could include, but is certainly not limited to, orientation tracking, data smoothing, and activity detection. The transformed data is then analyzed using a parameterized gesture recognition system, which matches portions of the incoming data stream to a set of predetermined patterns. The recognition system will give both a sense of the quality of a gesture (a straight line, a curve, etc) as well as its parameters (fast, short, etc). The framework provides a scripting language to allow an interface designer to link the occurrence of a gesture (or gestures with certain parameters, two coincident gestures, etc) to output routines of their choosing, effectively attaching meaning to these gestures. The data flow is shown in figure 1-1.

Such a framework would help interface designers bypass much of the difficulty associated with the use of inertial sensors in their devices. Knowledge of analog electronics, microcode, and radio frequency transmission is no longer required. The fragility of *ad hoc* gesture recognition systems and the complexity of data analysis are avoided. The designers simply determine the gestures associated with the movements in which they are interested (software is provided to aid this process) and in turn connect each gesture to output routines.

Systems built with this framework will realize several benefits from inertial technologies. They are open-loop systems, and therefore can be wireless (contrast with magnetic trackers). Their compact size allows for direct instrumentation of the object of interest and therefore provides direct measurement of the quantities of interest (contrast with computer vision). Further, as discussed above, both market pressures and research progress will lead to these sensors becoming increasingly more economical in the short term.

This framework also points towards the long term goals of this project (beyond the scope of this document), which is to realize the sensor package as a stand-alone device, rather than one which sends its data off-board for processing. Given simple (or simplified) algorithms for each of the framework functions described above, it is possible to create stand-alone devices that have a sense of their own movement and the ability to respond to this knowledge in some simple way. Imagine shoes that beep when you are overrotating, golf clubs that yell "Slice!" when you are about to slice, or even juggling balls that can teach you how to juggle[30]. This technology points the way to a world of intelligent unwired devices.

Chapter 2 will discuss the design of the inertial measurement unit (IMU) we created, and a sample data stream therefrom is shown in Chapter 3. Chapter 4 will describe various analysis techniques relevant to inertial measurement and tracking and Chapter 5 presents the single axis gesture recognition algorithms we created for this project, which allow for efficient robust gesture recognition on both full and reduced sets of inertial sensors with few limitations. Chapter 6 presents the simple scripting language we designed to ease the creation of the sample application, described in Chapter 7. Chapter 8 presents the conclusions and future directions.

Chapter 2

Hardware Systems

The hardware systems are the foundation of this project, providing the core sensor data which the analysis and gesture recognition software later interpret. The design of the wireless inertial measurement unit (IMU) itself will be described in some detail, including an examination of component selection and the operation of the unit's microcode. The general operation of the receiver hardware and software will also be explained. For readers not interested in the technical details, a short summary is also provided.

2.1 Summary

The physical design of the wireless inertial measurement unit is a cube 1.25 inches on a side (volume $< 2\,\text{in}^3$) and is shown in figure 2-1. Two sides of the cube contain the inertial sensors. Rotation is detected with three single-axis gyroscopes and acceleration is measured with two two-axis accelerometers . The sensor data are input to a microcontroller (on the remaining side) using a 12-bit analog-to-digital converter (ADC). The raw sensor values are then transmitted wirelessly to a separate basestation, which connects to a data analysis machine via a serial link. In addition, short-range signaling circuitry was provided for use in proximity sensing. The complete system operates at 3 V and draws 26 mA[1] while powered,

[1] Most of the draw is from the processor and the gyroscopes, split evenly between them.

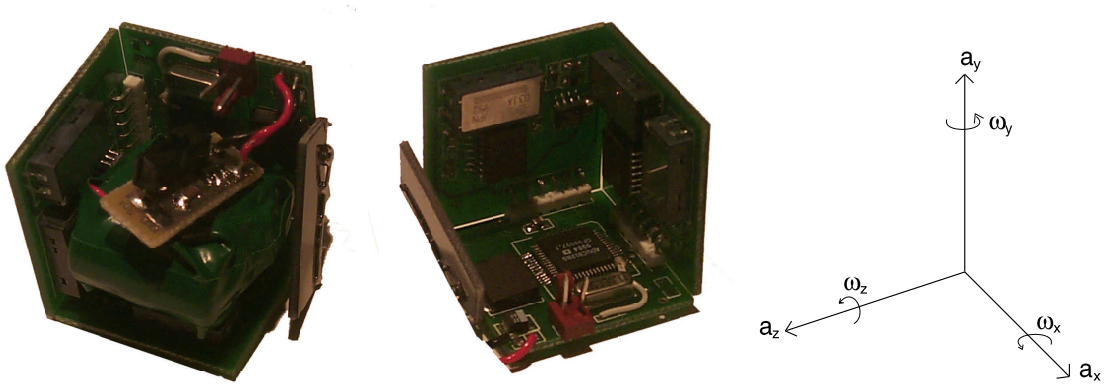

Figure 2-1: Current IMU Hardware and Reference Frame

and runs for about 50 hours on two batteries placed in parallel. These batteries are also small enough to fit inside of the cube formed by the hardware. The total cost of the system, in prototype quantities, is approximately US$300.

2.2 IMU High Level Design

We begin by considering the requirements for overall functionality of the wireless inertial measurement unit; in effect, what are its defining characteristics. Without such an analysis, it is impossible to complete a proper low-level design, much less choose components.

At a minimum, the IMU must provide three axes of both acceleration and rotation sensing, as well as data collection and a wireless link. Subsidiary systems provide capacitive signaling and power supply. Mechanical and structural details that can be vital when using inertial sensors are also discussed.

2.2.1 Core functionality

There are three main subsystems in the core functionality: sensing, processing and communication. Each is examined in turn, and a schematic of high-level data flow is shown in figure 2-2.

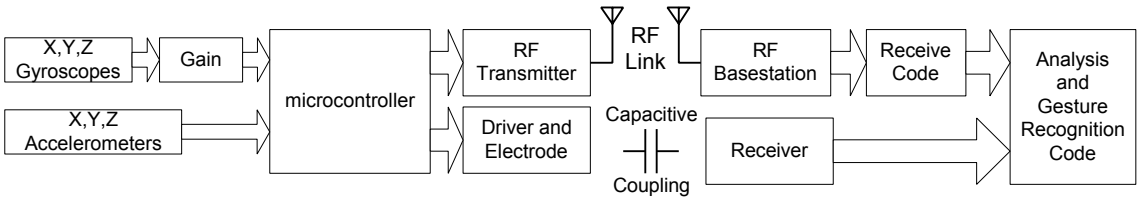

Figure 2-2: High-level Data Flow

The sensing subsystem must detect a full 6 degrees of freedom (DOF) of motion. To achieve this, the IMU will use three axes each of accelerometers and gyroscopes. While it is possible to measure rotation using two (or more) co-axial accelerometers[31], such systems must be completely rigid and tend to be rather large (on the order of an aircraft width) as their resolution is proportional to the separation of the accelerometers. In contrast, gyroscopes provide a more direct measure of rotation in a compact space. Magnetometers are sometimes used to measure orientation relative to the Earth's magnetic field, but only provide two DOF and are hard to use indoors, where the local magnetic field varies wildly.

A sampling rate of approximately 50 to 100 Hz and a signal resolution of 8 to 12 bits are expected to be necessary for adequate recognition latency and accuracy (this assumption is tested in Section 5.4). To match this requirement, the sensors should have a bandwidth of at least 50 Hz (by Nyquist's theorem [32]). To achieve the desired resolution, the signal to noise ratio (SNR) must be greater than ~ 60 dB. Analog processing will be required to low-pass filter the signal, as well as to modify the sensor output ranges to match the input range of the ADC.

The processing subsystem collects this filtered data from the sensors, either by timing pulse-width-encoded digital signals or using an analog to digital converter (ADC) to acquire analog outputs. Some simple signal manipulation is done on board the IMU, consisting primarily of the digital signal timing mentioned above, as well as encoding the data for transmission via a radio frequency (RF) link. Processor speed is relatively unimportant due to the low update rate (relative to usual clock speeds in such devices) and the limited processing being done on board. However, the ADC speed should be fast enough to avoid becoming a bottleneck.

The final subsystem is the communication hardware. To allow the IMU to be used in as wide a variety of situations as possible, a wireless communication system was chosen. The necessary range will be relatively short, on the order of ten meters, and the communication rate should be at least 9600 bps[2]. A wired interface should also be provided for debugging purposes.

At this point, we note that a signal path has been established. The inertial sensors react to movement, their output signals are filtered and then collected by a processing unit. This unit packages up the data and then transmits it via a wireless link to a receiving basestation (described in Section 2.6) which then sends it along to a computer for analysis (Chapter 4).

2.2.2 Mechanical Design

In general, mechanical design tends to be a minor concern for hardware systems. However, since this system uses inertial sensors, matters such as size, strength and rigidity are very important.

The prime consideration, of course, is the placement of the sensors. Since the components chosen (see next section) are virtually certain not to sense along all three axes, circuit cards at right angles to each other are necessary. Specifically, the design uses three square planar circuit panes, at right angles, forming half a cube (see figure 2-1). They are both physically and electrically connected using right angle connectors. Note that while it is difficult to guarantee that the panes will be exactly perpendicular to each other, any misalignment can be dealt with in software (if necessary).

The other consideration is the quality of the electrical connections, since the device will be, by definition, undergoing a fair amount of motion and stress. Therefore, the battery is soldered in place, and any vias must be soldered through to ensure that connectivity is not lost if the circuit boards flex.

[2] Assuming 2 bytes per measurement, 6 DOF x 2 bytes = 12 bytes = 96 bits (disregarding per byte and per packet overhead). Therefore, 9600 bps is the absolute minimum for 100 Hz updates.

2.2.3 Subsidiary systems

The subsidiary systems are those which either provide extra input or outputs, such as the capacitive coupling system, or perform necessary though not defining tasks, such as the power system.

Because this is a wireless system, batteries must be used as the power supply for the hardware. These batteries should be able to power the full system for at least a day to maintain the continuity of user experience, not to mention the usual maintenance and ecological concerns of using large quantities of batteries. The batteries need to be small enough to fit unobtrusively either within the IMU (which would also reinforce the structure) or on an outside face. A voltage regulator and switch circuit will also be necessary.

The only subsidiary output capability in the current design is a capacitive transmit plate, designed to allow the IMU to be detected if it is over a similar receive plate. Varying the transmit frequency (or sending a modulated code) will allow differentiation of devices. Though not implemented, it would be very easy to use the same electrode as either a receiver, so the device could detect when it was over a transmitting plate, or as a load-sensing plate, such that the device could detect if it was being held or touched[33]. A simple input header for auxiliary sensors of various sorts would also be valuable[3], but is not in the current design.

2.3 IMU Component Selection

In selecting components, we consider two sets of goals gleaned from the above design. The first are the functional requirements - 6 DOF sensing, data processing, data transmission. The second are the usability goals - small and wireless, as well as low-cost and low-power. With those in mind, we now examine part selection in four broad categories: the inertial sensors themselves, microprocessors, RF transmitters and receivers, and other components.

[3] *E.g.* the data from a glove measuring finger bend would quite nicely complement the motion information from an IMU mounted on the wrist.

2.3.1 Inertial Sensors

While inertial sensors have a long history, it is only in recent years that their price and size have dropped dramatically, especially with the advent of microfabrication technology. Table 2.1 gives an overview of some of the sensors available on the market. This table is skewed towards lower cost, non-mechanical sensors as per the design criteria.

In the case of the accelerometers, the Analog Devices components are notably superior for our applications. They provide adequate accuracy and bandwidth as well as excellent power drain and price per axis. The ability to switch between the pin compatible 2 g and 10 g version is very useful as well, as is the dual analog and digital outputs. Finally, Analog Devices will soon be distributing a much smaller version of the ADXL202 with a footprint of $0.2 \times 0.2 \times 0.1\,\text{in}^3$[36].

In selecting gyroscopes, the matter is not quite as simple. While the Gyration gyroscope has the best power drain and price, as well as good accuracy, it is simply far too large for this design[4]. While far from perfect, the Murata gyroscopes are small and have reasonable performance and price, and will therefore be used in this design. The Fizoptika gyroscopes' specifications clearly demonstrate that an order of magnitude increase in price will buy an order of magnitude increase in noise performance, these are not appropriate in this case because of their large size, cost and power drain.

The design of the IMU is continuing to evolve. A planned revision will take advantage of both the smaller Analog Devices accelerometers and their promised MEMS gyroscopes[37]. However, the delay of both these releases made this revision impossible within the time frame of this dissertation.

Another possibility for the future lies in ICs or hybrid circuits containing both three degrees of accelerometers and gyroscopes on board; effectively an IMU on a chip. Samsung Electronics is currently developing such a device[38] with dimensions of 20 mm square by 1.5 mm. It would have competitive specifications (see table 2.1) with a price point around $50.

[4] In fact, the module's size is on the order of the desired size of the IMU!

	Part and Type	Voltage	Current per axis	Technology[a]	Axes	Range	Noise	Band-width[b]	Drift	Bits[c]	Size (in^3)	Cost per axis
Accelerometers	Analog Devices ADXL202[d]	$2.7-5.3$ V	0.6 mA	MEMS	2	± 2 g[e]	4.3 mg (50 Hz)	5 kHz	N/A[f]	8.5	0.4x0.4x0.2	$8
	MSI ACH-04-08-05	5 V	Varies[g]	Piezoelectric	3	± 250 g	250 mg[h]	5 kHz	N/A	10	0.3x0.3x0.2	$75
	Silicon Devices 1210	5 V	6 mA	Capacitive	1	± 5 g	2 mg (1 kHz)	400 Hz	N/A	11	0.4x0.4x0.2	$6
Gyroscopes	BEI QRS11	± 5 V	< 80 mA	MEMS	1	$\pm 100°/$ sec	$0.12°/$ sec (100 Hz)	100 Hz	$0.2°/$ sec	9.4	1.6 ⌀x0.6	$1400
	Fizoptika[35] VG941-3	5 V	200 mA	IFOG	1	$\pm 500°/$ sec	$0.04°/$ sec (50 Hz)	1 kHz	$0.02°/$ sec	13	1.1 ⌀x2.2	$1100
	Gyration Microgyro 100	$2.2-5.5$ V	2.7 mA	Vibrating reed	2	$\pm 150°/$ sec	$0.15°/$ sec	10 Hz	$0.12°/$ sec	9.6	0.9x0.9x0.8	$25
	Murata ENC-03J	$2.7-5.5$ V	3.2 mA	Vibrating Reed	1	$\pm 300°/$ sec	$0.5°/$ sec[i] (50 Hz)	50 Hz	$0.5°/$ sec[i]	10.2	0.6x0.3x0.2	$80
IMU	Samsung IMU[j]	$2.7-5.3$ V	3 mA	MEMS	3	$\pm 500°/$ sec	$0.4°/$ sec	50 Hz	$1°/$ sec	10	0.8x0.8x0.1	$10
					3	± 3 g	10 mg	50 Hz	N/A	8		

[a] Details about these technologies can be found in [34].
[b] $3dB$ point
[c] $\log_2(\frac{2.5 \times Noise}{Range})$
[d] Digital and analog outputs.
[e] ± 10 g version also available
[f] Not applicable
[g] Depends on an external bias network.
[h] Based on quantization noise for a 12 bit ADC.
[i] Not given on spec sheet; measured by hand.
[j] Available Q4 2000.

Table 2.1: Overview of available inertial components.

2.3.2 Processors

The next major component to consider is the microcontroller. An overview of parts can be found in table 2.2. It lists only parts from major manufacturers which meet the minimum necessary functionality (ADC, timers) and usability criteria (power, size[5]).

As with the gyroscopes, the selection process is quickly reduced to deciding which design goal is most important. The Analog Devices chip offers excellent ADC features but slower processing, while the Microchip PIC16F877 offers both reasonable ADC performance and processing speed. The Motorola chip, a vintage device, does not stand out, and the Hitachi processor is simply too large and unwieldy for this design. Since our requirements call mostly for data collection rather than data processing, the Analog Devices ADuC812 was chosen. For future systems in which data will be analyzed on board, chips such at the Hitachi SH-1 and the Intel StrongARM series will be more appropriate.

Again, as in the previous discussion, the part which best matches the design considerations, the Microchip PIC16C774, will be available for the next anticipated round of revision.

2.3.3 Radio Frequency Transmitters/Receivers

The last major component to select is the RF chipset, an overview of which is shown in table 2.3. In this case, consideration is limited only to chipsets which require either limited or no external components and which accept a serial stream as input to simplify interfacing with the microcontroller.

At the time the design was done, the RF Monolithics HX/RX1000 series was easily the most appropriate, with a small footprint, reasonable speed, and low current draw. It also provides a large number of different fixed frequencies, allowing multiple devices to be used at once.

For new designs, the TR1000 series transmits at a far higher rate than the HX/RX1000 series for only a small increase in current drain. In the near future, chipsets using short-

[5] Footprint $< 1\,\text{in}^2$

Part	Voltage	Current	Core	Pins	RAM(bytes)/ ROM(words)[a]	ADC Inputs/Bits	Sample Rate	MIPS[a]	Timer Rate	Features	Cost
Analog Devices ADuC812	3 V	12 mA	8 bits CISC	52	256/4k	8/12	200k	0.5	1 MHz	UART, Vref (2.56V)	$8
Hitachi SH-1 7034	3.3 V	60 mA	32 bits RISC	112	4k/16k	8/10	75k	20	20 MHz	UART(x2), Fast Mult.	$35
Microchip PIC16F877	5 V	7 mA	8 bits RISC	40	368/8k	8/10	24k	5	5 MHz	UART	$8
Microchip PIC16C774[b]	5 V	13 mA	8 bits RISC	44	256/4k	10/12	30k	5	5 MHz	UART, Vref (4.096V)	$9
Motorola M68L11E9	3 V	8 mA	8 bits CISC	52	512/6k	8/8	45k	1	2 MHz	UART	$9

[a]For CISC chips, this value is an average.
[b]Limited availability at present.

Table 2.2: Overview of available microcontrollers.

Part	Voltage	TX Current	Data Rate	Receiver Sensitivity[a]	TX Size (in^3)	Frequencies	Cost per set
Linx LC Series	3 V	1.5 mA	5 kbps	-95 dBm	0.5x0.4x0.2	315/418/433 MHz	$15
Radiometrix TX2/RX2	3 V	6 mA	40 kbps	-100 dBm	1.3x0.5x0.2	418/433 MHz	$50
RF Monolithics HX/RX1000[b]	3 V	3.5 mA	19.2 kbps	-71 dBm	0.4x0.4x0.1	303/315/418 433/868/916 MHz	$35
RF Monolithics TR1000 Transceiver[c]	3 V	6 mA	115.2 kbps	-85 dBm	0.4x0.3x0.1	303/315/418 433/868/916 MHz	$50
Ericsson Bluetooth Transceiver[c]	3.3 V	35 mA	1 Mbps[d]	-70 dBm	1.3x0.7x0.2	2.5 GHz	$200

[a] At 10^{-5} BER.
[b] Discontinued April 2001.
[c] Limited availability at present.
[d] Spread Spectrum, 79 channels

Table 2.3: Overview of available RF parts.

range network standards such as Bluetooth[39] will allow reasonable numbers of wireless devices to operate in the same space without interference, but may have to be smaller than current prototypes to be appropriate for applications such as this one. Cambridge Silicon Radio is currently developing a chip combining a Bluetooth radio, baseband DSP and microcontroller, which would be perfect for future embedded applications[40].

2.3.4 Other Components

We now consider parts from the subsidiary and mechanical systems, which, though not as crucial as those described above, still merit some discussion.

The components selected above allow for the IMU to be powered with a single 3 V source. Therefore, a $3.3 - 3.6$ V battery and a regulator will be used. Further, the selections suggest that $\sim 20 - 30$ mA of continuous current will be necessary. Coin cells would be the first choice for use with our cubical design; the cell could simply sit on the inside of one of the faces. A product search found no coin cells with a high enough continuous load rating. Therefore, a short cylindrical cell was chosen which matched our requirements. The Tadiran Battery TL-5902[41] has a 3.6 V nominal voltage, a capacity of 650 mA hours at 10 mA load, and dimensions of $0.57 \varnothing \times 1.0\,\text{in}^3$. Two of these batteries were run in parallel and sit within the cube formed by the electronics.

To create the cubical form factor, connectors between the sides were necessary. The simplest method of achieving this was to use right angle headers as connectors. The AMP 644457 right angle connector, which required 0.2 in on each side, suited our purposes nicely, with a plastic base to add some support. For smaller designs, the new Molex 5550 right angle connector requires only 0.1 in clearance on each side and has pins which do not extend between the base, providing greater stability.

Finally, a Splatch planar antenna from Linx Technologies was used for transmission. While not achieving the same performance as a quarter-wave whip, they do provide transmit powers within 5 dB and require far less space - the Splatch measures $1.1 \times 0.6 \times 0.1\,\text{in}^3$ and

should be able to fit easily within the electronics; a 900 MHz quarter-wave whip antenna is on the order of 3 inches long and will extend beyond the core circuitry.

The remaining parts used in building the IMU hardware are considered to be generic and are not discussed here.

2.4 IMU Low Level Design

With a high level design completed and components selected, we turn our attention to the low level design issues surrounding the IMU. Specifically, we now consider the implementation of the necessary functionality and usability goals.

2.4.1 Core functionality

Our discussion of the implementation of the core functionality will follow the signal path, beginning with the inertial sensors, moving through the processor and then on to the output systems. The schematics for the IMU are found in figure C-1. Part designators from the schematics are referred to throughout this section.

The Murata ENC03J gyroscopes' output signal is differential, floating around a central voltage that is nominally given by a reference voltage pin (V_{ref}). The output is amplified using an inverting amplifier configuration, centered on op amps U1-2 (see figure C-1), with bias of V_{ref} on the non-inverting input. The amplifier has a gain of 1.5 so that its output range will match the full $0 - 2.56$ V input range of the ADuC812 ADC. This gain can be adjusted depending on the expected range of motion of the device. A feedback capacitor (C2) provides a low-pass filter with a roll-off frequency of 66 Hz. The three filtered gyroscope signals are wired into the ADC inputs.

The ADXL202 accelerometer has a configurable bandwidth and period for the pulse-width encoded outputs. In this case, the smallest available period, 1 ms, was chosen to achieve the highest update rate. This was done by setting R12-13 to 125k Ω. The bandwidth is set to

50Hz by letting C18-21 be 47 nF. The digital outputs are wired directly into digital inputs on the ADuC812. All four accelerometers are connected, though only three are necessary for full functionality. While these outputs in theory are at the full accuracy of the device, they require many more processor cycles to measure[6] than the analog outputs, and the accuracy of the measurement will depend on the granularity of the internal timer. The digital outputs were used in this design solely because the other ADC inputs were being used to collect data from two magnetic sensors[7], which have since been removed. It is recommended that the analog protocol be used in any future revisions.

The setup of the ADuC812 itself is fairly straight-forward. Beside the usual voltage inputs (and bypass capacitors), there is an 11.0592 MHz crystal to provide an oscillator and a pull up on the $\overline{EA}$ pin to switch to the internal memory space. There are also push-button switches on the reset and program pins to allow the user to put the processor into in-circuit programming mode.

The data is transmitted using the serial UART on the ADuC812 at 19.2 kbps, which is the maximum rate for the HX1000 transmitter. The same serial stream is also routed to a header to allow hardwired connections for debugging purposes. A kHz-range square wave is output to a separate pin for use with the capacitive signaling hardware.

2.4.2 Mechanical Design

The mechanical structure of the IMU consists of three printed circuit board (PCB) panes at right angles, connected mechanically and electrically using the right angle headers described above. The three panes contain:

- Pane 1: The processor and transmitter
- Pane 2: One accelerometer and one gyroscope
- Pane 3: One accelerometer and two gyroscopes

Each pane is 1.25 in on a side and the cube itself has a volume of less than 2 in^3.

[6] On low-end processors such as the one in this design. Chips with pulse-width modulation inputs can time such signals in the background.

[7] These were included to provide information about the proximity of two devices.

The connectors provide a fair amount of rigidity but are far from perfect. It is hard to align the panes properly, and there is enough play in the system to allow the panes to flex slightly. The battery alleviates this problem to a fair extent, but it is still well worth considering using the other connectors mentioned earlier or providing mechanical rigidity through other means. The simplest solution is to epoxy the panes to a hard shell, while more complicated solutions[42] have embedded the sensors in an acrylic block.

A note on connectors: any external connection to the IMU must either be soldered in place or connected via a screw terminal block. While this level of precaution is unnecessary for most hardware systems, the nature of the IMU, especially in our applications, is that it will be subjected to a large amount of movement, which tends to cause intermittent connection loss that can be very hard to diagnose.

Beyond greater rigidity, another important point to consider in a revision of the mechanical system is the form factor of the device, as a cube is not the most appropriate shape for all application. This system would be more generally useful if it had a planar shape, with only a small extension at right angles to provide the third axis of sensing. It is possible to produce such a structure which could be folded into a cube by moving one of the panes.

2.4.3 Subsidiary systems

The capacitive signaling system is designed to buffer the signal running from the processor to the transmitting electrode and to round the sharp edges of the square wave train (to ensure FCC compliance). The signal is amplified using a common-emitter NPN amplifier and is then passed through an RC low pass filter to the output electrode.

The power regulation system is trivial. The battery runs through a switch, into a low-dropout 3 V voltage series regulator (Toko AM TK11630), and then out to the IMU. The IMU draws a continuous current of 26 mA, giving a pair of the selected batteries a 50 hour lifespan.

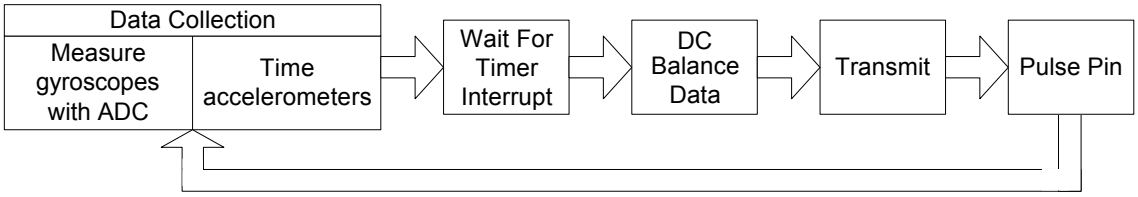

Figure 2-3: Data Flow in the Embedded Code

2.5 Microcontroller Embedded Code

While the ADuC812 microcontroller embedded code performs no complex algorithms, there are still a number of notable implementation issues. The major functions of the code are considered: data collection, RF transmission, and capacitive signaling. The complete code is found in Appendix D and figure 2-3 shows the data flow.

2.5.1 Data Collection

The data collection system works at a fixed rate, with an interrupt designating the start of each time step. Each cycle, the data from the previous round is transmitted and new data is collected, and then the device waits for the next interrupt. If the processor should hang and has not completed collecting data by the start of the next time step, the data is marked as dirty and is not transmitted that round. This should allow the device enough time to reacquire the sequence. The current interrupt period is 15 ms, giving an update rate of 66.6 Hz.

The gyroscope data is collected at each time step using the built-in ADC on the microcontroller. Since the ADC operates at 200,000 samples per second, there is little to gain by executing other instructions in parallel with this collection. The accelerometer data is input to the microcontroller in the form of a duty-cycle modulated square wave, as shown in figure 2-4 below:

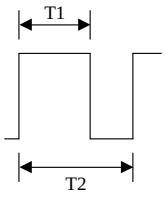

Figure 2-4: ADXL202 Digital Output

The acceleration sensed is $(T1/(T1+T2)-50\%)\times 1\,\mathrm{g}/12.5\%$. Since $(T1+T2)$ is a constant set by the external resistor values (see above), we need only measure either $T1$ or $T2$ at each time step. In this case, we measured $T1$, simply by waiting for a rising edge and starting the timer, then waiting for the falling edge and stopping the timer.

There are some problems with this scheme, however. We note that it takes on average a full cycle to complete a measurement. Further, since the ADXL202 pulse output tends to hang high or low with out of range accelerations (roughly, $|a| > 5\,\mathrm{g}$), it is prudent to include a timeout counter inside the timing loop to watch for this condition. However, this increases the grain of the timing measurement to unacceptable levels. For the above reasons, the next revision of the design will most likely measure the analog output of the accelerometers. It requires more signal processing, but given our constraints provides much better accuracy and shorter measurement times.

2.5.2 RF Transmission

Once collected, the sensor data stream must be DC-balanced[43] before transmission to ensure good receiver tracking of the threshold point. Because the measurements being transmitted are 12 bit quantities, a 6-bit to 8-bit DC balancing scheme was used. Since there are 64 6-bit values and 70 balanced 8-bit values, 6 of the byte values were not used. These were the 5 values with runs of 4 consecutive highs, which can cause tracking problems, as well as 55_x (where subscript x indicates a hexadecimal number), which is reserved for other purposes. The remaining values were mapped in order (i.e. $0 \rightarrow 17_x$, $1 \rightarrow 1B_x$, etc.), and placed in a lookup table.

The strength of this scheme is in its compactness. It quickly and with little memory use (64 bytes for the look-up table) allows the conversion of a 12 bit quantity to two bytes, the atomic units of serial transmission. It guarantees that each byte will be DC-balanced, and that there will not be runs of greater than six. Single bit errors will always be detected.

The data transmission itself is done using the serial port on the ADuC812 with packets constructed as follows. A two-byte header is sent. The first byte is FF_x if the on-board power monitor shows a voltage above 2.93 V, FE_x for between 2.93 V and 2.63 V and FD_x otherwise. The second byte is always 01_x, to balance the header. Then the measurements are sent as two byte quantities, the high byte being the balanced version of the high 6 bits of the ADC reading, the low byte being the balanced version of the low 6 bits. The gyroscope readings are sent in order (x,y,z), followed by the accelerometer timing results in the same order.

While data is being collected, the processor attempts to keep the transmitter active and continually sending the value 55_x. Since this value alternates high and low, it aids greatly in keeping the receiver locked on to the proper threshold value. This transmission is accomplished by scattering non-blocking[8] writes throughout the code.

2.5.3 Capacitive Signaling

The processor strobes a pin for 1ms at the end of each collection cycle, at either 50 kHz or 20 kHz (though any value is possible). This is designed to be used to transmit a signature via a simple mesh electrode, for the purpose of capacitive signaling. An example use of this capability is described in the section on the `(void*)` application (section 7.2).

2.6 Receiver Hardware

The receiver basestation is designed to simply receive the RF transmissions from the IMU and shift them to RS-232 levels for transmission to a computer. The schematic and PCB

[8] Only writes to the port if it is free

Figure 2-5: Current Basestation in Shielded Case

layouts for this board can be found in Appendix C. A description of the functionality follows.

The main IC on the basestation is a RF Monolithics receiver chip (U1) that matches the transmitter on the IMU. The RF signal is input through a BNC jack, to which an appropriate length whip antenna is attached. The digital output data from the receiver is passed through an op amp in unity gain configuration (U7, as the receiver could not drive the serial line driver) and then is shifted to appropriate RS-232 levels[9] using a serial line driver (U3). A serial cable to the computer can be attached to the female DB9 jack provided (J1). Note that a jumper (LK1), allows the user to select between the RF signal and direct cable connection (JP1, from the IMU, presumably) for debugging purposes.

Power is supplied via a standard DIN-5 jack (J8). 3 V power is generated using the TK11630 voltage regulator. Also, debugging LEDs are provided to indicate data reception (L6), as well as 5 V (L3) and 3 V (L4) power.

This receiver should be enclosed in a metal case to provide RF shielding. The drawings for an appropriate case can be found in Appendix C and a picture appears in figure 2-5.

[9] 0 V $\rightarrow$ +12 V, 5 V $\rightarrow$ −12 V.

2.7 Receiver Software

We complete this chapter with a short description of the receiver code for use in a personal computer connected to the basestation. This code is designed to find valid data packets in the incoming data stream, decode them, and then send them along for further analysis.

The receiver code works in two stages. In the collection stage, the code reads 24 bytes from the serial port, 14 bytes for a full data packet plus 10 bytes worth of padding. This guarantees that at least one packet header should be in the buffer. The code scans along this buffer looking for a valid header start byte (FD_x to FF_x) followed by the valid second byte (01_x). It then copies the next 12 bytes (reading more data from the serial port if necessary) to a processing buffer. Bytes are then read in pairs from the buffer and put through the inverse of the DC-balancing mapping to that used in the microcontroller. Any bytes that do not have a valid decoded value are mapped to FF_x, which serves as an error code. Note that all the header bytes as well as the padding byte (55_x) fall into this category, as would any byte with a single bit error. Any invalid byte fails the whole packet. Assuming no invalid bytes, the 12 bit values are reconstructed and placed, together with a timestamp, into an instantiation of a data record class, which is then read by the analysis and gesture recognition code. Since data collection in the embedded processor begins right after the transmission and data analysis does not take place until the whole packet is collected, there is a net latency of 15 ms in the hardware.

This code was made part of the Synthetic Characters[44] code base, a collection of Java classes constructed for the purpose of building virtual creatures who inhabit 3D computer graphical worlds. The code base contains graphics systems, behavior systems, interface mechanisms, and a host of mathematical and prototyping tools, greatly simplifying the process of application creation. While the receiver software is usable separately, modifications will be required to remove various extraneous class references. The author is currently developing stand-alone versions of all the code discussed in this document.

Chapter 3

Sample Data Stream

At this point, we present an example data stream to get a sense of the information being gathered by the hardware. This stream will be used throughout the document. It shows the data associated with a number of simple human arm gestures on one accelerometer axis and two gyroscope axes, one of which is about the accelerometer axis. The streams are shown in figure 3-1 and a list of gestures follows in table 3.1.

This data was gathered a single subject (the author) performing the listed gestures in order. The IMU was contained with a teapot held in the right hand. No effort was made to ensure that parameters such as long or short was reproduced exactly, since this will not be the case in unconstrained human gesture.

Note that while the algorithms developed in this dissertation will be as general as possible, their application will often be in recognizing human gesture, and therefore both this data stream and the subsequent discussion will be skewed toward that use.

The data was collected using a PC-based data acquisition card (National Instruments PCI-6024) at a rate of 2000 samples per second, each with an accuracy of 12 bits. The analog outputs from the IMU were amplified to match the full range of the ADC, and were low-pass filtered to roll-off frequencies outside the range of interest.

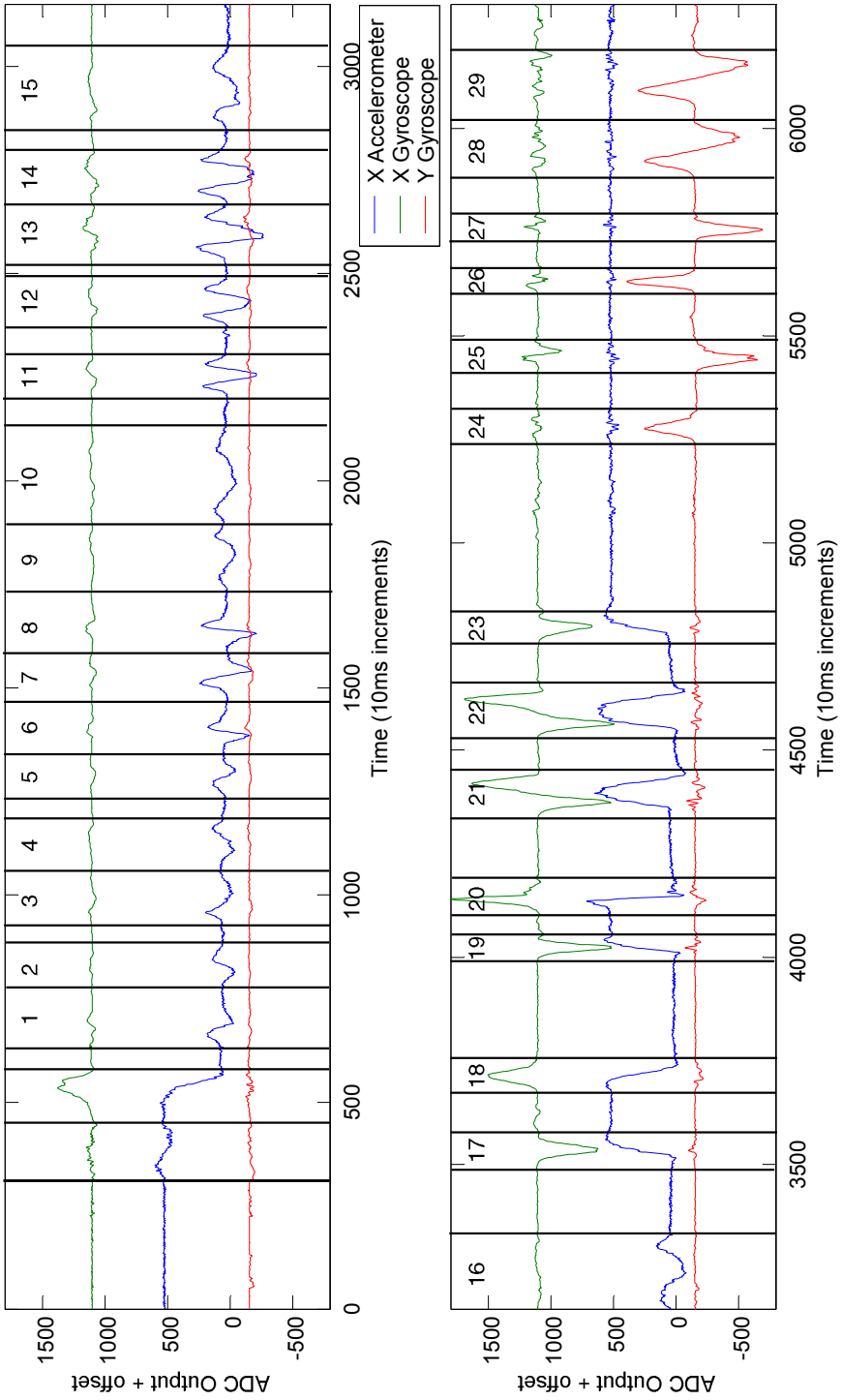

Figure 3-1: Sample Data Stream

1. Short straight line
2. and back
3. Short straight line
4. and back
5. Short fast straight line
6. and back
7. Long fast straight line
8. and back
9. Short Loopback
10. Repeat
11. Long loopback
12. Repeat
13. Long fast loopback
14. Repeat
15. Short fast loopback
16. Repeat
17. Twist
18. and back
19. Fast twist
20. and back
21. Twist and return
22. Repeat
23. Twist
24. Twist on y
25. and back
26. Fast twist on y
27. and back
28. Twist and return on y
29. Repeat

Table 3.1: Gestures in Sample Data Stream

This data stream is fairly comprehensive and includes a wide range of gestures. It will therefore be used to tune the algorithms we develop in the following chapter and to illustrate their performance. Note that in the sample application presented in Chapter 7, the algorithms use real-time input data, not that from this set.

For further analysis, the data will be downsampled to a variety of rates; this allows us to evaluate our performance with different rates and accuracies of data. This will be done using the `decimate` command in MATLAB[45], which includes an anti-aliasing operation. The data is then rounded to the nearest integer to maintain the accuracy[1]. The data shown in figure 3-1 is 12 bits, downsampled to 100 Hz, similar to that obtained from the hardware described in the previous chapter. This resolution and sampling rate will be used in the examples throughout the document. Note that the baseline for the first gyroscope channel (on axis) has been modified for clarity.

We quickly note a few key features of the data. All of the gestures present themselves as a sequence of peaks of alternating polarity, which is consistent with smooth motions. Also, there are no single peaked accelerometer gestures, as this would imply a change in velocity, since each gesture began and ended at zero velocity. Finally, we note that either the gyroscopes' acceleration sensitivity is quite high, or that slight twists are being introduced

[1] Or nearest multiple of 2^n, to reduce the accuracy by n bits.

into straight-line gestures. In either case, these will have to be filtered out.

Without launching into a detailed discussion, it is worth noting that the streams shown are consistent with those in the physiological literature (e.g. [46, 47]). Further reference to this literature will be made when appropriate.

Chapter 4

Analysis

The analysis system acts as an intermediary between the raw data from the IMU and the gesture recognition system. Its purpose is to transform the data in such a way as to make the recognition stage easier and more accurate. The method chosen should have low latency and, with an eye towards the future, require limited processing power. Three algorithms to achieve this goal are examined. Kalman filtering can be used to transform the IMU values from its local coordinate frame to a fixed world frame, which is necessary if the instrumented object is not the sole item of interest and some kind of position tracking is needed. Filtering the data in frequency space can be used to distill the essential information from the data by removing the low-frequency baseline shifts and extraneous high-frequency noise and shocks. Finally, this chapter will show the derivation of an activity detection algorithm, designed to highlight sections of the data stream where there is a significant departure from a noisy baseline. This algorithm was chosen as the input (along with the raw data) to the gesture recognition stage, described in Chapter 5.

4.1 Kalman Filtering

Kalman filtering is the main analysis technique for inertial data and is used almost exclusively for inertial tracking, the determination of position and orientation from inertial

readings and an initial state. As inertial components have become more common in user interfaces, particularly for head-tracking[48], a number of variations on the original algorithms have been created to ensure robustness and accuracy when working with low-cost components and processing hardware[1]. These extensions will be examined, as will the limits placed on this tracking over time by sensor errors.

Note that this section will not present the details of the Kalman filtering algorithm (which is extremely well developed in the literature) or the derivation thereof. For readings on this topic, the user is directed to [50, 51, 52].

4.1.1 Principles and Algorithms

Kalman filtering (KF) is a state-based recursive algorithm, which works in two stages. The first is the prediction stage, where given the (possibly incorrect) current state of the system and a mapping of how the system progresses with time, a prediction of the state at the next time step is calculated. The second is the correction stage, where given a noisy observation (not necessarily full or direct) of the state at the new time step, the predicted state and the measured values are combined to give a new, more accurate, state for the system. Given certain assumptions described below, Kalman filtering is the least-squares optimal linear method of estimating system state.

The strength of Kalman filtering lies in its optimality, its minimal memory requirements for state (only a single time step), and its ability to include a model of the system dynamics. There are a number of limitations which must be considered as well. The most troublesome is that it is the *linear* optimal solution, while most physical systems, including tracking, are non-linear. In such circumstances the Extended Kalman Filter (EKF) can be used, which, while it can no longer be proven optimal (or even to converge), is still very successful in practical applications. The next problem (for either type of filter) lies in the process noise[2] models, which are assumed to be white and Gaussian. To see the implications of

[1] As compared with military systems, which are still the primary users of inertial tracking. Strategic grade IMUs cost in the hundreds of thousands, if not millions, of dollars[49].

[2] Any dynamics not contained by the model

this, consider a state vector containing position and velocity. Since the acceleration is not contained in the state, the best model of the velocity dynamics is $v_{t+1} = v_t$, which treats the (unknown) acceleration as white noise, which it most certainly is not. It is possible to bootstrap by using the velocity to calculate an approximation of the acceleration and then feed that back to the velocity (which actually works because of physical limits on the jerk), but this points to a final problem. The most time consuming operation in the execution of the Kalman Filter is a $n \times n$ matrix inversion in the correction step, where n is the length of the state vector. Since matrix inversion is an $\mathcal{O}(n^3)$ operation, this can become very costly if the state vector becomes large, as it does in the case of inertial tracking, with a usual length of eighteen[3].

A number of techniques have been used to alleviate some of the above problems. Foxlin[53] demonstrates two interesting methods. By using a Complementary Kalman Filter which estimates the state error rather than the state itself, the latency associated with the inversion operation can be reduced. The smaller magnitudes in the state vector also make linear approximation more accurate, allowing for the use of the KF instead of the EKF, and the state vector itself is reduced to six members. By exploiting certain aspects of the measurement structure, the equations can in fact be rewritten with a single 3×3 inversion. Welch and Bishop[54] extend this concept to the logical extreme with Single Constraint at a Time (SCAAT) tracking, where each observation is used in a correction step as it is collected, thereby minimizing the inversion size and reducing latency as well (since we no longer have to wait for a full set of measurements).

While these techniques are strong, they ultimately fail the context of this project for two reasons. The first is that the above systems use sensors other than accelerometers and gyroscopes (magnetometers for Foxlin, acoustic sensors for Welch) which allow for their simplifications, while in our case the systems are not as separable since both the accelerometers (via tilt angle) and gyroscopes are used to update the orientation, conflating those two portions of the state vector. The second, and more fundamental problem, is that the sensor update rates and accuracy are not high enough to track for any reasonable periods

[3]Three degrees each of the position, velocity, acceleration, orientation, angular velocity, and the gyroscope biases.

of time, as will be discussed in the next section.

4.1.2 Limits of Inertial Tracking

We now consider the cumulative position and orientation error over time for inertial tracking. This problem will be examined for the case of a single axis, and taking only gyroscope errors into account (we will use the parameter for the Murata ENC-03J from table 2.1). While fairly simplified, this model will give a lower bound on the error, which should be illustrative. More detailed discussions can be found in [55, 56].

There are three main sources of gyroscope error. Axis misalignment error occurs if the sensors are not exactly perpendicular to each other and can be corrected in software up to one half of the accuracy of the accelerometers. Therefore, since the accelerometers have 8.5 bits of accuracy, there will be a misalignment error of $M = \sin(0.5 \times 1/2^{8.5}) = 0.0014°$. The next source of error is random walk error caused by integration of noisy data. This error is proportional to the sensor noise and to the square root of time, giving an error term of $\sigma\sqrt{t} = 0.5\sqrt{t}$. Finally, we have gyro bias drift, which is based on the inability to track the gyroscope output baseline exactly. This error appears as a fixed rotation, adding a term of $\beta t = 0.5t$. Combining these, we have an error of:

$$\Delta\theta(t) = 0.5t + 0.5\sqrt{t} + 0.0014° \tag{4.1}$$

At any point in time, this gives a cross-axis acceleration error of:

$$\Delta a(t) = a(t)\sin(\Delta\theta(t)) \tag{4.2}$$

where $a(t)$ is the acceleration at time t. To calculate the net error, this equation must be double integrated. Therefore, we need to select an acceleration profile. We choose a

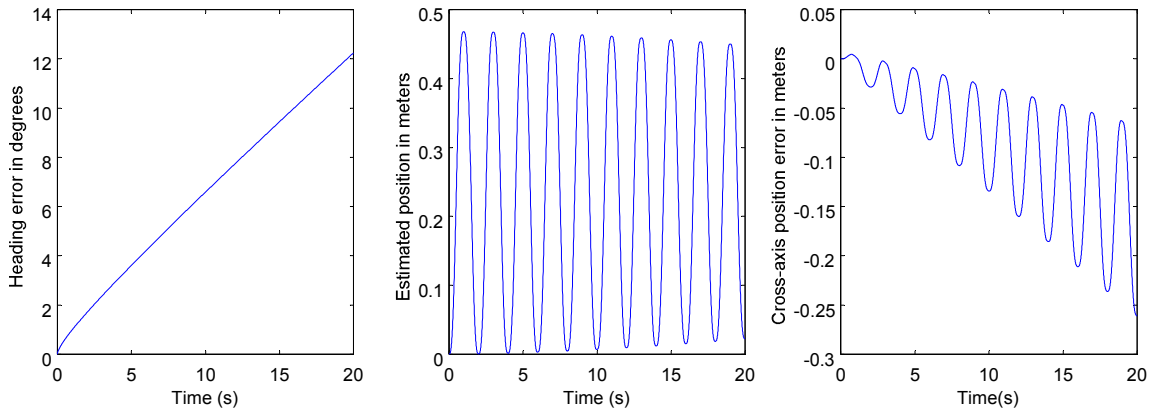

Figure 4-1: Error Growth over Time for Inertial Tracking of Gesture

function which approximates human gesture, letting:

$$a(t) = \begin{cases} +(0.3\,\mathrm{g})\sin(2\pi t), & \text{if } \lfloor t \rfloor \text{ is even (forwards motion)} \\ -(0.3\,\mathrm{g})\sin(2\pi t), & \text{if } \lfloor t \rfloor \text{ is odd (backwards motion)} \end{cases} \tag{4.3}$$

where $\lfloor t \rfloor$ denotes the greatest integer less than or equal to t, giving a 1 Hz sine wave which reverse polarity every cycle and has a magnitude of 0.3 g. This approximates a person performing a straight line gesture away from the body and back again, using the sample data stream of Chapter 3 as a guide.

Figure 4-1 shows the error growth over twenty seconds. We note that at the end of this time frame, the gesture is only 2 cm from the expected position on the axis of movement, but the heading error is over 12° and the cross-axis error is over 25 cm, which is completely unacceptable. Such an error could make a huge difference in the interpretation of a gesture, and this value will continue to compound over time. Therefore, inertial tracking is simply inaccurate given the quality of sensors chosen.

Note that if there is an external source of information about position and heading available, it can be used to correct the system. For example, if there is a slot in which the device must be placed at regular intervals, then the tracking error can be zeroed. Therefore, the result does not mean that we cannot track for more than five seconds, only that we cannot track for more than five seconds without external information.

4.2 Frequency Space Filtering

While the sampling bandwidth of the ADC is between 0 and 60 Hz, that is not necessarily the frequency range over which the data contains the information of interest. In this section we examine the benefits of reducing the frequency band considered using both high- and low-pass filters.

4.2.1 Low Pass

The purpose of a low-pass filter is to reduce the effect on the system of noise in the bandwidth of immediate interest. The maximum frequency of interest for human arm gestures (the example application) is considered to be approximately 10 Hz[57], though quantitative analysis of the sample data stream suggests that most of the gestures in which we are interested have a maximum frequency in the $3 - 5$ Hz range.

Simple low-pass filtering can be done using digital signal processing techniques[58], using either finite-impulse response (FIR) or infinite-impulse response (IIR) techniques. FIR filters use only correlations with the incoming data stream, and their length is inversely proportional to the pole frequency. FIR filters introduce a constant group delay equal to half the filter length. IIR techniques feedback the filtered results of the preceding points as well, and therefore can achieve the same filter parameters in a shorter length. However, they do not have constant group delay, which causes shifts in the relative location of low and high frequency components (fast and slow motion), which is not acceptable.

Another interesting low-pass filter is the Savitzky-Golay[59, p. 650] smoothing filter. Originally designed for use in resolving frequency spectra, it reduces noise while keeping the n^{th} moment unchanged, where n is at least 2. Therefore, peak widths and heights are left unchanged, as that is the characteristic of interest in spectral analysis.

The utility of these filters eventually depends on the gesture recognition algorithm chosen. If it examines either the first or second moment (magnitude or derivative), then low-pass filtering will prove quite useful. If it instead looks at the zeroth moment (integral), then

any noise will be canceled by the summing operation, and low-pass filtering is irrelevant. This point will be revisited as the discussion progresses.

4.2.2 High Pass

High-pass filtering can be used to remove constant and slowly changing values from the signals. In this case, this would allow us to remove the gyroscope offset and bias drift and the accelerometer offset (due both to the 50% bias on the pulse-width modulated signal and the current tilt angle), which can be very useful if thresholding of the signals is desired. Again, quantitative analysis of the sample data stream suggests that a single gesture can take as long as 1.5 seconds, therefore, the high frequency cut-off will need to be approximately 0.5 Hz.

However, two problems occur in this case. The low frequency pole location results in IIR filters which can go unstable and FIR filters which are quite long. More importantly, a high-pass filter can only react on the order of its pole frequency (by causality) and therefore can generate artifacts if the IMU changes orientation too quickly. In the worst case, these artifacts can create new gestures as shown in figure 4-2. Because of these artifacts and the latency caused by the filter, these techniques are unacceptable.

4.3 Activity Detection

The final algorithm considered is an activity detection scheme we developed for this application. The purpose of this algorithm is to determine portions of the data stream that have a high probability of containing interesting motion.

To do this, it is first necessary to have a piecewise model of the data stream. We consider the lowest order model possible, with a constant for sections with no activity and a straight line for those with activity. Given this definition, the most obvious algorithm to use would

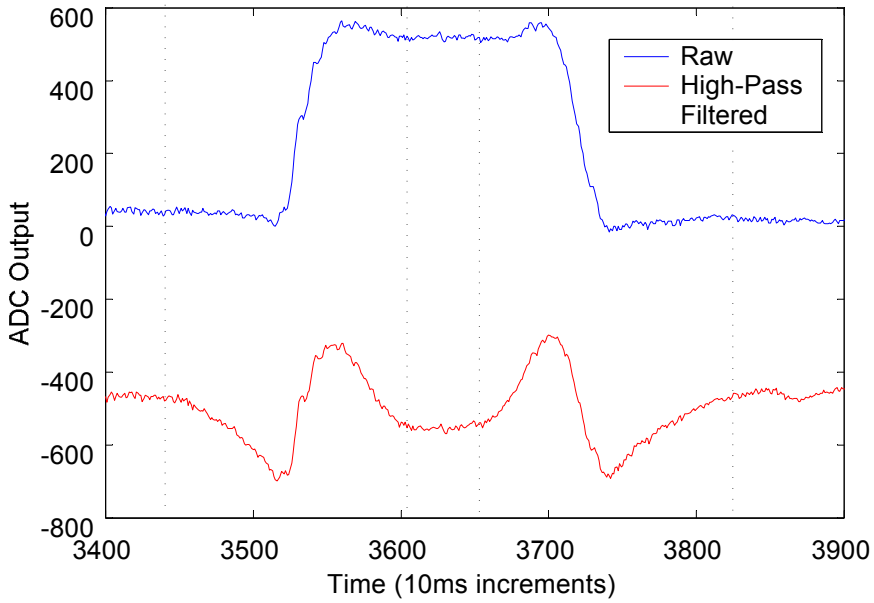

Figure 4-2: Gesture Artifacts Caused by High-pass Filtering

be a linear correlation in time:

$$r = \frac{\sum_i (y - \overline{y})(t - \overline{t})}{\sqrt{\sum_i (y - \overline{y})^2}\sqrt{\sum_i (t - \overline{t})^2}} \tag{4.4}$$

where y is the data stream of interest and t is the time index. Not only will this detect straight lines (by definition), but it is slope-independent, which is consistent with the parameterized nature of the system. This algorithm has reasonable latency (100 ms for the example shown), though the structure of linear correlations does not allow for caching of calculations[4].

However, as can be seen in figure 4-3 (which runs this algorithm on an indicative section from the sample accelerometer of figure 3-1), this technique does not give good results at the local extrema of the gestures, where the linear approximation breaks down. This suggests that the model should be expanded and another algorithm found.

We next constructed an algorithm based on windowed variances. To understand how this

[4]The kernel cannot be precomputed at each point because of the presence of the means.

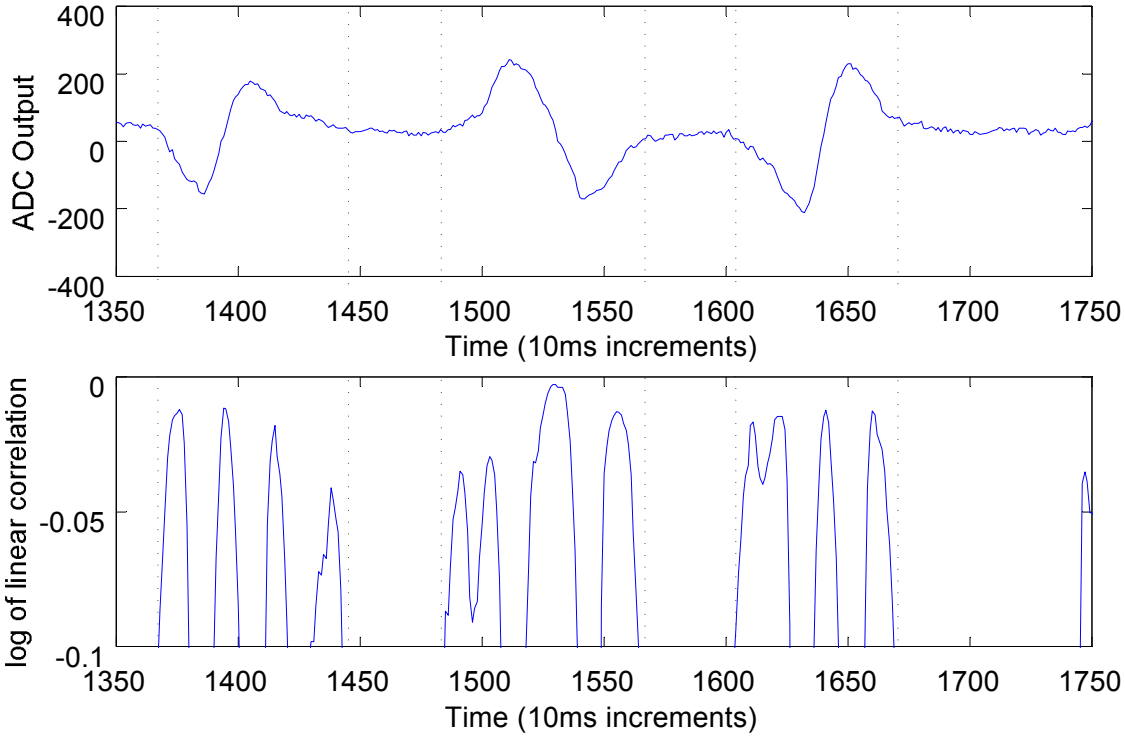

Figure 4-3: Linear Correlation as Activity Detection Algorithm for Accelerometer Data

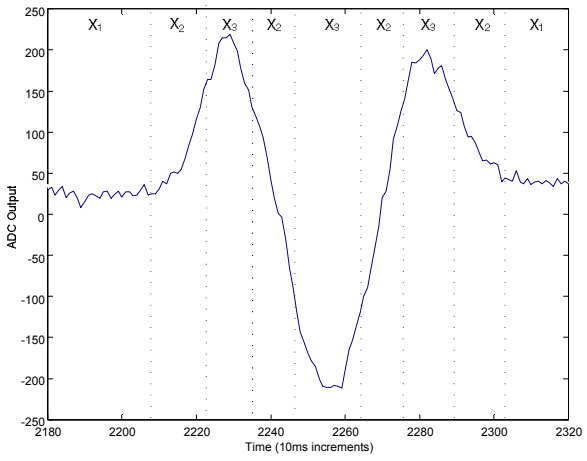

Figure 4-4: The Piecewise Model Used in the Activity Detection Algorithm

will work, we consider three piecewise models to make up our data stream: constant, linear, and quadratic, each with additive white Gaussian noise (again see figure 4-4). In the constant case, the calculation is trivial:

$$x_1 = \mathcal{N}(0, \sigma) \tag{4.5}$$

$$\Rightarrow \sigma_{x_1}^2 = \sigma^2 \tag{4.6}$$

where σ is the sensor noise and $\mathcal{N}(0, \sigma)$ is a zero-mean Gaussian distribution with variance σ^2. In the linear case, we have:

$$x_2 = m_2 i + \mathcal{N}(0, \sigma) \tag{4.7}$$

where m_2 is the slope and $i \in \mathbb{Z}$, $i \in [-n/2, n/2]$, n *even*, without loss of generality and $n+1$ is the window size. This gives:

$$\overline{x_2} = 0 \tag{4.8}$$

$$\begin{aligned} \Rightarrow \sigma_{x_2}^2 &= \overline{x_2^2} \\ &= \frac{1}{n} \sum_{i=-n/2}^{n/2} (m_2 i)^2 + \mathcal{N}^2(0, \sigma) + (mi)\mathcal{N}(0, \sigma) \\ &= \sigma^2 + 2m_2^2 \sum_{i=0}^{n/2} i^2 \end{aligned}$$

$$\Rightarrow \sigma_{x_2}^2 = \sigma^2 + \frac{m_2^2 (n+2)(n+1)}{12} \tag{4.9}$$

where we now see that it is possible to differentiate between these two portions of the data stream. For the quadratic model, using the same n as the linear model, we have:

$$x_3 = m_3 i^2 + \mathcal{N}(0, \sigma) \tag{4.10}$$

where m_3 is the quadratic coefficient. This gives:

$$\overline{x_3} = \frac{m_3^2 n(n+2)}{12} \tag{4.11}$$

$$\sigma_{x_3}^2 = \frac{1}{n} \sum_{i=-n/2}^{n/2} (x_3^2 - \overline{x_3})$$

$$\Rightarrow \sigma_{x_3}^2 = \sigma^2 + \frac{m_3^2}{180}(n^4 + 5n^3 + 5n^2 - 5n - 6) \tag{4.12}$$

	Constant		Linear			Quadratic		
n	$min(\sigma_1^2)$	$\max(\sigma_1^2)$	$min(\sigma_2^2)$	$\max(\sigma_2^2)$	$min(m_2)$	$min(\sigma_3^2)$	$\max(\sigma_3^2)$	$min(m_2)$
5	4.62	61.67	5.67	75.70	9.58	4.63	61.78	6.56
10	9.61	48.88	16.47	83.82	3.11	9.81	49.91	1.11
15	12.20	43.99	30.18	108.79	1.73	13.35	48.11	0.42
20	13.84	41.25	48.49	144.47	1.16	17.67	52.64	0.21
30	15.88	38.15	101.19	243.17	0.66	36.49	87.69	0.08
40	17.13	36.38	176.91	375.73	0.45	84.78	180.06	0.04

Table 4.1: Expected Variance Ranges for Various Window Sizes for Accelerometer Data

where the equation 4.12 is found using the MATLAB symbolic solver. This piecewise model is illustrated in figure 4-4, which shows data from a typical gesture divided into the regions described above. The designations at the top of the graph indicate the most appropriate model for that portion of the data.

It follows from observation of equations 4.9 and 4.12 that for small n the variance of a 2nd order section will be less than that of a 1st order section. Therefore, activity can be found using a simple thresholding scheme with a high start threshold and a low stop threshold. The start threshold should be slightly less than the minimum expected value of variance in a linear section, and the stop threshold should be slightly smaller than the minimum expected variance in a quadratic section. This algorithm will work as long the maximum expected value of the variance in the constant section is smaller than the two thresholds.

We consider this algorithm for the case of the accelerometer sample data stream described. For this stream, which is composed of typical human arm gestures, we found $\sigma^2 \cong 26$, $min(m_2) \cong 1.3$, and $min(m_3) \cong 0.08$. We calculated the 90% confidence interval for $\sigma^2_{1,2,3}$[60] for a number of different window sizes (n), using the minimum m (model coefficients) given above. The minimum value of m for which there is no overlap of the constant case (x_1) is also shown for each model. This data can be found in table 4.1. A similar calculation was done for the gyroscopes, but is not shown.

A value of $n = 30$ was chosen, which appropriately balances the latency caused by the filter ($n/2$ or 15 ms) against the minimum value of m which can be detected. A start threshold of 100, far above the expected variance in constant portions, and a stop threshold of 50,

just slightly above the expected constant model variance, will be used. This value of n is not perfect, as the quadratic case overlaps just slightly with the constant case. This may cause some stuttering, where the period of detected activity missed a portion of a gesture, though it is not of great concern and is easily solved in higher-level software. The sample accelerometer data stream segmented using these values can be found in figure 4-5.

As can be seen in the figure, this algorithm was fairly successful. It identified at least the start and end of all areas of activity, though it missed the center of some. The false positives are unavoidable, as they are either caused by a number of random errors forming a linear section, or by a shift in baseline, which, while not a gesture (on that axis), is certainly activity. This algorithm has a reasonable latency and can be executed very quickly by caching data streams of cumulative data point sums and square sums, and using the identity $\sigma^2 = \overline{x^2} - \overline{x}^2$. MATLAB code showing this algorithm can be found in Appendix E in the function `findgests`.

There are some potential definitional problems which stem from this algorithm. They arise from the fact that we define constant acceleration or angular velocity as the start and end condition of the gestures. Therefore, constant accelerations and angular rates are effectively ignored, and the instrumented object must meet this condition before parsing can begin. For human motion, which for a given path is usually executed such that the jerk is minimized[61], this condition would imply that the velocity or angular velocity is in fact nil, which would make it difficult to recognize cyclic gestures or a quick transition between two gestures. With this algorithm, we may have no choice but to consider what could possibly be classified as two (or more) gestures as a single one and attempt to solve this issue as part of the gesture recognition routines.

Despite these concerns, windowed variance will be used as the input (along with the raw data) to the gesture recognition stage. The algorithms for recognition, including the separation of false positives from actual gestures, and the combining of stuttering gestures[5] into full ones, are described in the next chapter.

[5] *E.g.* time index 2850-3050 in figure 4-5

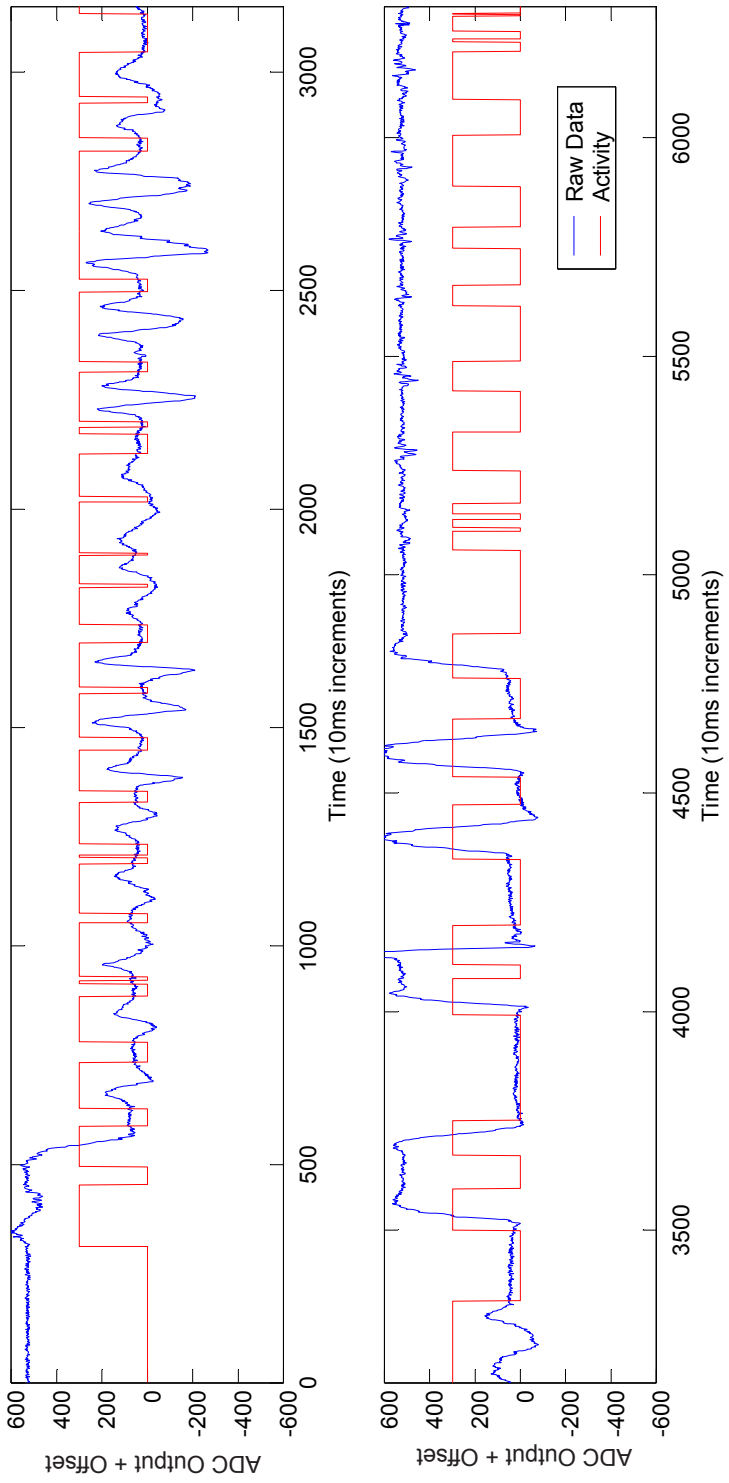

Figure 4-5: Windowed Variance as Activity Detection Algorithm for Accelerometer Data

Chapter 5

Gesture Recognition

The concept behind the gesture recognition system is fairly straight-forward. The author claims that it is possible to decompose any arbitrary gesture (within reasonable limits) into one or more *atomic* gestures. These gestures should be fairly simple and easy to recognize via algorithmic means. Magnitude and duration will be treated as parameters of these atomic gestures, rather than being considered to be fundamental to them. This chapter details the development of such a generalized parameterized gesture recognition system. Our concept of a gesture will be defined and the parameterization will be described. State-based gesture recognition techniques, which have had great success in computer vision applications, will be examined, with an eye towards their suitability to this project. The main portion of this chapter will give the details of a single-axis gesture recognition system, including comments on the interpretation of gestures. Finally, the minimum data sampling rate and accuracy for recognition will be examined.

5.1 Key Definitions

To be able to construct a parameterized gesture recognition system we must obviously first consider what is meant by a gesture, and how we wish to parameterize it. This section gives those definitions, and comments on the strengths and limitations of the approaches taken.

5.1.1 Gesture

In the context of this thesis, a gesture is considered to be any space-time curve. This definition serves two goals. First, it highlights that a gesture cannot be defined solely by its value at a small number of instants, as thresholding schemes do, but is instead a path having an extent in time, which allows for more complex structures. Second, and more importantly, it suggests gesture as a way of thinking of more than simply human motion, though that is the sample application in this case. The same curves in space could be made by a human forearm, a dog's torso, a juggling ball, or a car. The breadth of our definition will rest on how general the analysis and gesture recognition algorithms are.

Consider the limitations that have already been placed on the gestures. We have enforced a constant acceleration or angular rate as the start/stop condition of the gesture, with the implication that the net acceleration or angular rate is in fact nil, and the data simply reflects a bias, either inherent in the sensors or from a tilt of the object. This is built on the assumption that a human motion cannot produce such values otherwise (which the sample data stream supports). How does this effect our ability to measure non-human gesture? Take the example of a vehicle swerving. Prior to the event, the vehicle will have zero acceleration as it is maintaining a constant speed. The swerve itself will have a noticeable tangential jerk, which will be detected, and the vehicle will then return to zero velocity, making recognition possible.

We now examine the concept of *atomic* gestures. By definition, such gestures cannot be decomposed into smaller meaningful gestures, and it must be possible compose them into larger gestures. The value of this definition is fairly obvious: it is only necessary to recognize a small set of atomic gestures which span the space of (human) gestures; thereafter, any new gesture of interest can be recognized simply by discovering the atomic gestures it is composed of. By contrast, conventional gesture recognition algorithms usually require that each gesture be trained individually. This can lead to possible overlaps in the state space and makes the system more difficult to use for those who do not understand the complexities of modern pattern recognition algorithms. The value of this scheme will lie in how well it can

approximate the key strength of conventional systems, which is the ability (given enough examples) to be trained to recognize any gesture.

5.1.2 The Parameterization

The gesture recognition algorithms for this framework will be parameterized as an aid to the application designer. Humans tend to divorce the quality of a gesture from the quantity; *e.g.* a long linear motion (without curvature) is thought of as a specific version of a line gesture, rather than a gesture in and of itself. The recognition system will reflect that notion by recognizing lines as atomic gestures, with their length given by the parameters as described below.

We now consider a prototypical gyroscope gesture $g_1(t)$ where we let:

$$g_1(t) = \begin{cases} c_1, & t \leq 0 \\ \omega_x(t), & 0 \leq t \leq t_0 \\ c_2, & t \geq t_0 \end{cases} \tag{5.1}$$

where $\omega_x(t)$ is the measured angular velocity (from the gyroscope) about the x-axis, t_0 is the duration of the gesture and $c_1, c_2 \in \mathbb{Z}$. We further enforce the boundary condition (as per the activity detection algorithm) of $\dot{g_1}(0) = \dot{g_1}(t_0) = 0$ (*i.e.* the angular velocity is constant).

We wish to consider two possible parameterizations of this gesture: those which travel the same path in different time, and those which travel an expanded or compressed version of the path in the same amount of time. We will place limits on the mappings allowed between our prototype $g_1(t)$ and some variation thereof $g_2(t)$. Equivalence is defined by the space-time trajectory of the object of interest being the same up to two scale factors, one in time, the other in space:

$$\theta_1 \equiv \theta_2 \text{ iff } \exists\ \alpha, \beta \in \mathbb{R} \text{ s.t. } \theta_1(t) - \theta_1(0) = \alpha(\theta_2(\beta t) - \theta_2(0)), \forall t.$$

Taking the derivative and using the definition above, we get:

$$g_1(t) \equiv g_2(t) \text{ iff } \exists\ \alpha, \beta \in \mathbb{R} \text{ s.t. } g_1(t) = \alpha\beta g_2(\beta t), \forall t. \tag{5.2}$$

Our boundary conditions denote the start and end of the gesture by requiring zero derivative at those points. Boundaries based on zero magnitude would require filtering to remove the baseline, as shown in the previous chapter.

Note that the parameterizations for all gyroscope measurements can be derived in this fashion.

We now consider the case of analogous accelerometer gesture $g_1(t)$ defined as above:

$$g_1(t) = \begin{cases} c_1 & t \leq 0 \\ a_x(t) & 0 \leq t \leq t_0 \\ c_2 & t \geq t_0 \end{cases} \tag{5.3}$$

where $a_x(t)$ is the measured acceleration along the x-axis, t_0 is the duration of the gesture, and $c_1, c_2 \in \mathbb{R}$. Similar to the previous equivalence relation, we have:

$$x_1 \equiv x_2 \text{ iff } \exists\ \alpha, \beta \in \mathbb{R} \text{ s.t. } x_1(t) = \alpha x_2(\beta t), \forall t.$$

setting $x_1(0) = x_2(0) = 0$ without loss of generality. Taking two derivatives and using the above definition gives:

$$g_1(t) \equiv g_2(t) \text{ iff } \exists\ \alpha, \beta \in \mathbb{R} \text{ s.t. } g_1(t) = \alpha\beta^2 g_2(\beta t), \forall t. \tag{5.4}$$

We again use the constant derivative boundary constraint on the gesture to specify start/stop.

The literature supports such a parameterization. Atkeson and Hollerbach[62] show that regardless of speed, distance and load, arm movement velocity profiles can all be scaled to the same general shape (they further use a similar scaling to ours[47]. Plamondon's very

detailed model of such systems[63] also allows for such scalings.

5.2 State-Based Approaches

State-based approaches, notably Hidden Markov Models (HMMs) [12], have had great success in the area of human gesture recognition. These techniques are based on tracking the internal state of a system and the transitions between them. Each state is associated with an output probability density for the observables of the system, and a transition probability to other states. Given enough examples of the observables of a gesture of interest over time, it is possible to infer the state transition and output probabilities[64].

In the context of gesture recognition, the states are usually the actual position and orientation of the object of interest, and the output probability density is a Gaussian bubble (representing sensor noise) around that value. The next position/orientation in the sequence is reflected in the transition probabilities. Since it is possible to calculate the probability with which an observation stream came from a model of a specific gesture, this system can be used for gesture recognition.

One intriguing state-based approach, suggested by Wilson[65], would be to predetermine a state space to represent the expected states through which the data from the IMU might pass during a gesture. The output means and covariances (of the Gaussian bubbles) would be determined based on knowledge of the expected values and noise levels of the sensors used. Training could then be accomplished simply by recording the states that a single example gesture travels through.

However, there are two key difficulties in using these techniques. The first stems from the fact that the parameters of the states themselves will usually be the output from the inertial sensors. Therefore, there is no way to achieve parameter-independent recognition of a gesture - each version of the gesture will travel a completely different path in the state space. Techniques have been developed to allow parametric extensions of HMMs[66], but they tend to be very processor intensive, which would not support the goal of stand-alone

devices. The second (and greater) problem is that while state-based approaches are very strong with absolute data, they tend to have problems with derivative data, as is available in this case. The trouble lies in the fact that gestures in this data space are cyclic - they start and end in the same state, and often pass through a given state more than once. This leads to degenerate parses in which a stationary device is believed to be in the stationary state, then progresses through the other states as quickly as possible, and returns to the stationary state again. Therefore, with reasonable probability, a full gesture has now been traversed, even though the device has not moved at all. This behavior is simply unacceptable and leads to difficulty in setting the probability threshold on gestures, thereby making the system less accurate. Because of these problems with HMMs, and the fact that they are fairly processor intensive, it was decided to seek another algorithm.

5.3 Single-Axis Gesture Recognition

This section describes the single-axis gesture recognition scheme created for this project. This scheme breaks down the recognition problem to the smallest possible unit: atomic gestures on a single data axis. The detected atoms can then be combined together into composite gestures using the scripting language described in Chapter 6. Three majors points are considered. The first is an examination of the pros and cons of this approach compared to looking for gestures in the full space, the second is the algorithms which were developed to find the gestures, and the third is a brief attempt at assigning some meaning to the atomic gestures chosen.

5.3.1 Pros/Cons

The general idea behind considering gestures on an axis-by-axis basis is that very few gestures in fact exist in the higher dimensional spaces considered by most algorithms. The author proposes that most gestures can easily be decomposed without loss in meaning. An arbitrary straight line gesture, for example, can always be projected onto (at most) three axes of space, and therefore can be recognized.

The benefits of such a scheme are readily apparent. Recognition algorithms in one dimension can be far less complex than those in higher dimensional spaces and still perform adequately. There is a further gain to be achieved through parallelization, since an identical set of algorithms can be run on each axis, recognizing the same gestures on each. Multi-dimensional schemes treat not only straight lines on different axes as different gestures, but also those at every angle in between. Single-axis algorithms will therefore require less processor time and less memory space, making them perfect for embedded applications. Furthermore, this system will work on reduced sets of inertial sensors, with gestures scripted for the full 6 DOF still applicable as long as the axes of interest remain instrumented.

The major concern in using single-axis gesture recognition is that a granularity is imposed on the gesture space that did not previously exist. Gestures are now broken down into a lexicon that may not be complete. The scripting system will have to impose a notion of simultaneity that will further reduce the space of allowable gestures. And while logical AND and logical OR operations are straight-forward to implement, logical NOT creates a host of problems (some described in section 6.1.2), meaning that not all combinations of gestures possible under a full dimensional scheme can be constructed here.

Nonetheless, there is little to suggest that these limitations will be fatal. The space of recognizable gestures and combinations thereof is still large, and it is possible that unrecognizable gestures could prove to be impossible for a human to execute, rendering the point moot. A full examination of this is a topic for future research.

5.3.2 Algorithms

We now consider the gesture recognition algorithms themselves. These algorithms take the raw data streams and use the activity detection algorithms' output (as described in the previous chapter) as a starting point to finding gestures in the stream. The recognition algorithm for the accelerometer data will be examined first, followed by comments on the slight differences for the case of the gyroscope data.

In both cases, groups of continuous peaks were considered the atomic unit. Therefore, peaks were never split in two, and groups of peaks were only split under limited conditions. The number of peaks was used to designate the type of the atomic gesture, as described in the next section.

Accelerometers

We begin by considering the constraints that have already been placed on the problem to determine whether they can now be exploited. Based on the assumption that the velocity is constant on both ends of an atomic gesture, we note that the integral of the acceleration over that range should be zero, once a baseline has been subtracted. In the case where the constant accelerations on either side of the gesture are not the same ($c_1 \neq c_2$), the baseline representing the change in velocity is approximated as a straight-line interpolation from one end to the other.

This provides the basics of the gesture recognition. Beginning with an area of suspected activity, the baseline is subtracted and three values are tracked across the gesture. They are the total absolute area under the peaks of the gesture, the net area under each peak, and the number of peaks. The consistency check for a gesture is simply the ratio of the net area to the absolute area under the peaks. If this ratio is below a fixed threshold, the gesture is recognized and passed, along with its parameters, to the scripting system. Single peaked gestures are, by this definition, ignored. Note that simply matching the height or widths of the peaks will not work, because asymmetries therein are common in precise motions[67].

While under ideal conditions, the net integral would be zero, that assumption cannot be made here for two reasons. The first is that the integral is effectively a tracking operation and therefore suffers from random walk noise. The second is that noise will also be introduced into the system if a baseline shift from the start to the end of a gesture is not linear, and this will also be reflected in the net to absolute integral ratio.

More needs to be said about the integral and peak parsing procedure. Firstly, since this is a noisy signal, it is necessary to ensure that it does not dither back and forth across the

zero point, thereby causing spurious peaks. This is done by enforcing a window after a zero crossings in which further crossing are ignored. Also, any peaks with a net integral below a fixed threshold are subsumed into the previous peak. This takes care of both spurious peaks cause by an incorrect baseline and the small peaks (used for correction) common at the end of fast human gesture[68].

The parameters for each atom are as follows. The end-time of the gesture as well as its width (β) are used to provide temporal information. The number of peaks determines the type of the gesture, and the polarity of the initial peak determines the direction. Integrating equation 5.4, we note that the net integral divided by β gives α.

We next consider two cases stemming from incorrect parsing by the activity detection algorithm. Note that these algorithms are used after the initial peak finding algorithm. The first is the case where two gestures are incorrectly pieced together as one. These gestures can be separated by considering the polarity of adjacent peaks, which cannot be identical without the device having come to a stop first. If two peaks of identical polarity are found adjacent to each other, the system splits the peaks into two gestures at that point and checks the consistency of each separately.

The second problem is the case where a single gesture is broken up into several separate activity sections, with a gap in between. This is solved by recording the location of the last segment to fail the consistency check since the most recent good gesture. If a failing segment is found and there is a previous failing segment with which it could be combined, the following algorithm is run. The start of the most recent segment is compared to the end of the previous one to ascertain that they are within a reasonable distance of each other. If so, the peak finding and integration algorithm is run on the segment reaching from the start of the first failed segment to the end of the second. If this new segment is consistent enough to be termed a gesture, it is passed along to the scripting systems. Otherwise, the combined segment is stored once again, hopefully to be combined with another segment in the future.

The algorithms described occupy approximately 200 lines of code, and can be written using solely integer quantities. This makes it highly suited to embedded systems.

Start	Stop	Peaks	α	β	Gesture
314	458	2	33.45	144	Not in Script
497	595	2	63.38	98	False Positive
630	739	2	45.70	109	Short Line
782	887	2	46.17	105	Short Line
932	1063	2	43.75	131	Short Line
1077	1215	2	38.71	138	Short Line
1235	1331	2	45.28	96	Fast Short Line
1356	1451	2	66.72	95	Fast Short Line
1479	1587	2	82.62	108	Fast Long Line
1594	1700	2	80.54	106	Fast Long Line
1738	2141	5	40.90	403	Joined Gestures
2202	2317	3	90.83	115	Long Loopback
2339	2501	3	76.28	162	Long Loopback
2528	2680	3	110.71	152	Long Fast Loopback
2681	2826	3	112.22	145	Long Fast Loopback
2852	3058	3	58.43	206	Short Fast Loopback
3135	3340	3	58.31	205	Short Fast Loopback
3672	3765	2	98.92	93	False Positive

Table 5.1: Recognized Accelerometer Gestures

A parsed data stream is shown in figure 5-1 and the recognized gestures are listed in table 5.1. In this case, the parameters (which were found empirically and are given in the units of the ADC, where appropriate) were:

- Consistency < 0.3
- Distance between segments to attempt combination < 30
- Net integral of a valid peak > 500

In this figure, the solid line (boxcar) indicates the good gestures and the dotted line indicates the bad gestures. We note that all the gestures were recognized and that the system properly pieced together the stuttering gestures (time indices 2800-3300) and split a single activity block into two gestures ($t = 2500 - 2800$). There are false positives in the segments where the baseline of the accelerometer shifted radically because of a change in orientation ($t = 550$ and 3700), but there is little that can be done in such cases, short of either tracking the orientation, which is quite difficult (as shown earlier), or disregarding all accelerometer gestures when there is a gyroscope gesture on the same axis, which would reduce our recognition set greatly. The small correction at the start and end of those segments, which

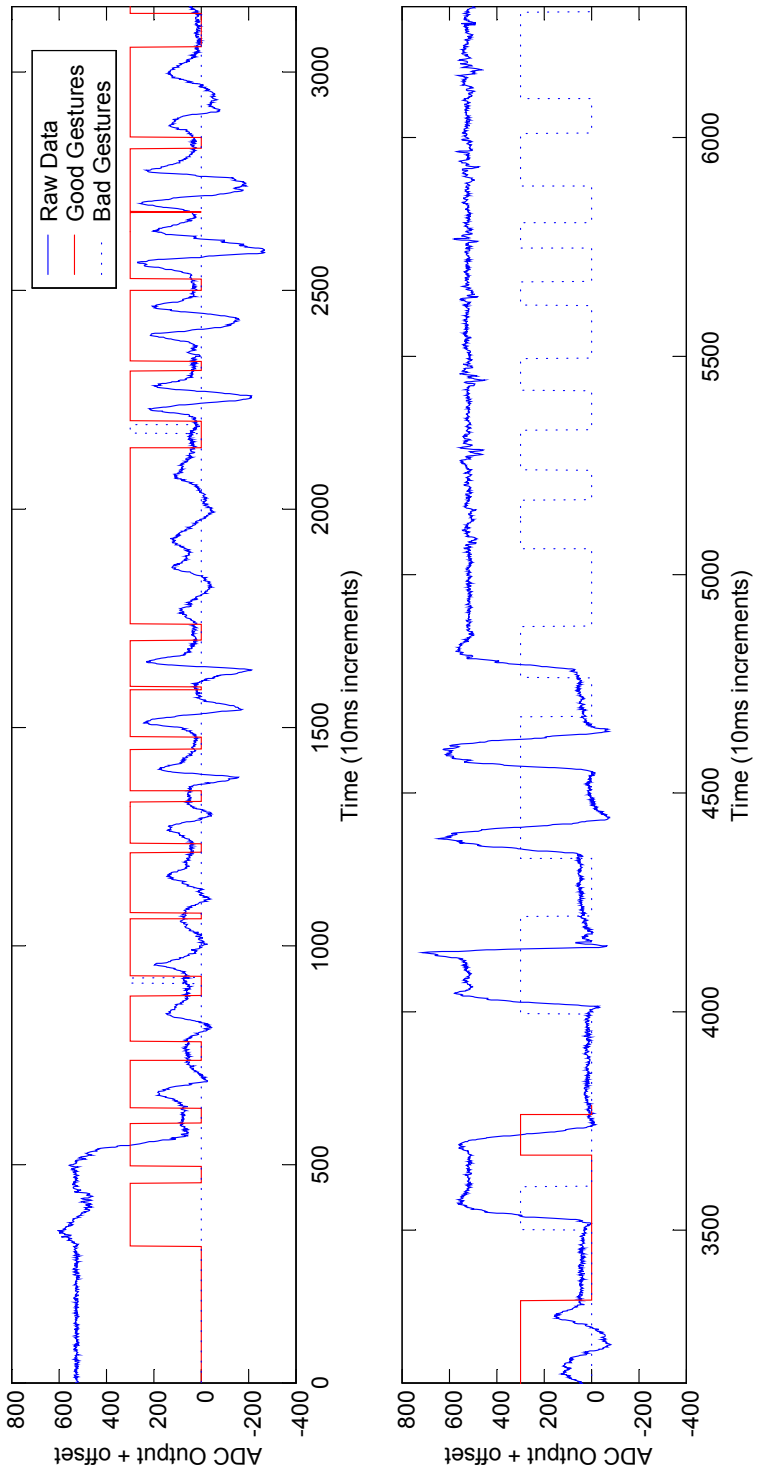

Figure 5-1: A Parsed Accelerometer Data Stream

are misunderstood as part of a gesture, are in fact fundamental to fast human motion[68]. A final concern is the case where two gestures were pieced together as one ($t = 2000$), even though the activity detection system recognized them as separable. In this case, a shift in the baseline and a lack of deliberateness in the gestures are the likely the source of error.

The results of the parameterization can be considered reasonable in this case, given the variability of human gestures. All the short lines have similar values for α, and the fast versions have a slightly longer duration. The long lines have approximately double the average α of the short lines. In the case of the loopbacks, the values of α for the long loopbacks are consistent, though the β values are hard to interpret. The short loopback has a much smaller α, as would be expected. Some of the difficulty in getting accurate parameters comes from the inconsistency in start and end points for the gestures, with some very accurate and others less so. Since α is dependant on this value, the only solution would seem to be a better activity detection algorithm.

MATLAB code for the accelerometer gesture recognition algorithm can be found in Appendix E in the functions `parsegests` and `gestcons` (helper function).

Gyroscopes

We now consider the differences between the accelerometer and gyroscope algorithms. The main difference is that the consistency check must be replaced as the zero velocity condition it enforces is no longer applicable. Instead, a peak size threshold is used, to ignore gestures that are caused either by the acceleration sensitivity of the gyroscopes, or by misalignment of the sensors. Otherwise, the only difference is that α is now the absolute integral itself.

A parsed data stream is shown in figure 5-2 and a list of recognized gestures is given in table 5.2. In this case, the parameters (which were found empirically and are given in the units of the ADC, where appropriate) were:

- Absolute sum for valid gesture > 7500
- Distance between segments to attempt combination < 30
- Net integral of a valid peak > 1000

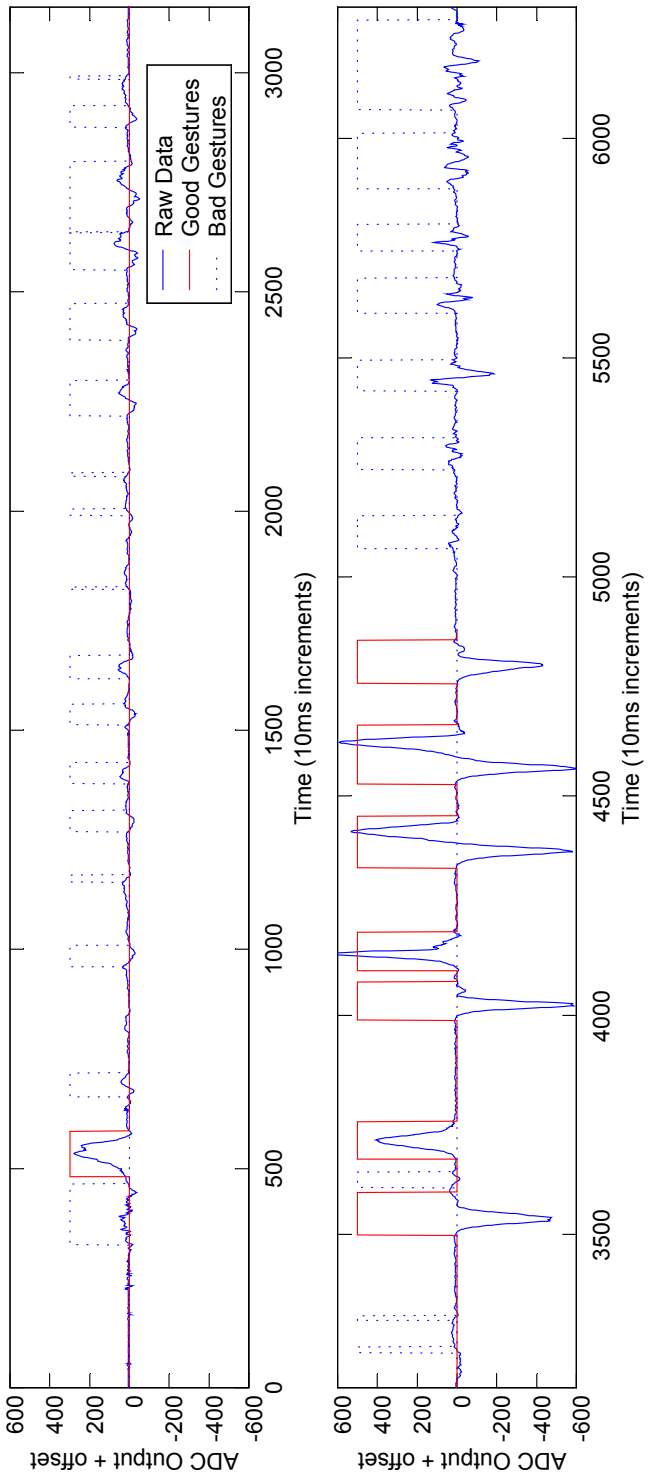

Figure 5-2: A Parsed Gyroscope Data Stream

Start	Stop	Peaks	α	β	Gesture
482	586	1	11976	104	Not in script
3498	3597	1	14753	99	Slow Twist
3672	3758	1	12330	86	Slow Twist
3989	4077	1	12350	88	Fast Twist
4102	4190	1	12978	88	Fast Twist
4336	4455	2	25909	119	Twist and Return
4527	4663	2	27948	136	Twist and Return
4757	4857	1	11536	100	Twist

Table 5.2: Recognized Gyroscope Gestures

In this figure, the solid line indicates the good gestures and the dotted line indicates the bad gestures. This case is far simpler than the previous one, with all gestures successfully recognized and no false positives. The parameterization was quite effective in this parse as well. Each of the single twist gestures, regardless of speed, had an α between 12000 and 15000, and the twist and return gestures were between 26000 and 28000, falling well within double the first range.

MATLAB code for the gyroscope gesture recognition algorithm can be found in Appendix E in the functions `parseggests` and `gestmass` (helper function).

5.3.3 Interpretation

The current interpretation of the gestures follows from the recognition scheme. Accelerometer and gyroscope gestures are classified in terms of the number of peaks. A two-peaked accelerometer gesture is a straight-line, three peaks represents a loopback, etc, with each new peak adding another line segment in the path. The gyroscope gestures follow the same pattern, where a single peak is a twist, two peaks is a twist and return, etc. We do not claim that these gestures provide a basis for the space of (human) gesture, merely that they are one logical way of decomposing the problem given the structure of the data.

Note that these descriptions are somewhat arbitrary and are based on observations of human arm gestures. Overall, any attempt to assign meaning to these atoms will fail in one

	33 Hz	50 Hz	100 Hz	200 Hz	500 Hz
8 bits	1/2/0	0/3/0	1/2/1	1/3/0	2/1/0
12 bits	1/0/0	2/0/2	1/0/1	1/0/1	1/0/1

False Positives/False Negatives/Joined Gestures

Table 5.3: Gesture Recognition Rate for Various Sampling Rates and Accuracies

context or another. In general, what the atomic gestures mean is irrelevant, as their key characteristic is that they can be combined to form larger gestures, as shown in the next chapter.

5.4 Entropic Limits

In this section, we consider the minimum data sampling rate and accuracy necessary to achieve reasonable recognition. This is done by starting with the raw accelerometer data stream and decimating it in time and accuracy to produce lower entropy[1] data streams. The activity detection and gesture recognition algorithms were run on these streams, and the number of false positives, false negative and joined gestures (two or more gestures grouped as one) are recorded in table 5.3 (out of 29 total). The parameters used by the activity detection and gesture recognition algorithm were chosen in the same fashion as described in Section 4.3 and 5.3.2, respectively. Note that the goal here is not to prove the optimality of our algorithms or to compare them to others, but merely to get a sense of how the success rate changes with entropy.

Data reduced to six bits of accuracy was also analyzed, but our current algorithms proved inappropriate. Visual inspection suggests that it would be possible to collect interesting information from that data stream, certainly the presence of gestures and the number of peaks. However, it seems there is not enough entropy in the stream for our current algorithm, which produced meaningless output. A simpler scheme looking at pairwise differences between data points could be successful in this case and is left as possible future work.

[1]Total information content, comprising both number of bits per sample and number of samples per second.

The only conclusion that can be drawn from the table is that there is no statistically significant difference between any of the tested data streams. The few differences which can be seen are based on differing parses of a few segments. Specifically, referring to figure 3-1, the two first loopbacks (9 & 10) are usually joined together, the change in baseline before the first gesture is often considered a gesture, and the change in baseline during gesture 20 is sometimes classified as a gesture.

However, the lack of significance in and of itself is relevant, as it suggests that running at lower sampling rates and accuracies will not result in a significant loss of recognition. However, latencies at lower sampling rates were slightly higher, and the accuracy of the parameters will not be as good.

Chapter 6

Output Scripting

The recognition portion of the framework produces parameterized atomic gestures on an axis-by-axis basis. For this to be useful to interface designers, there must be a way to combine these gestures into larger units and to interpret those units in some fashion. An output scripting system was designed to fill this role. It allows users to combine gestures based on their type, any or all aspects of the parameterization, and their temporal relation. Output routines can be tied to the occurrence of these combinations. A sample script is provided for reference.

6.1 Scripting System

The output scripting system extends the generalized nature of the overall framework. It allows for any number of atomic gestures to be assembled into full prototype gestures, with matching functions on both the individual atoms and the temporal combinations thereof. These full gestures are matched against the recognized atomic gesture stream. If the gesture is found, a specified output routine is executed. The general flow of data in the scripting system is shown in figure 6-1 and is detailed below.

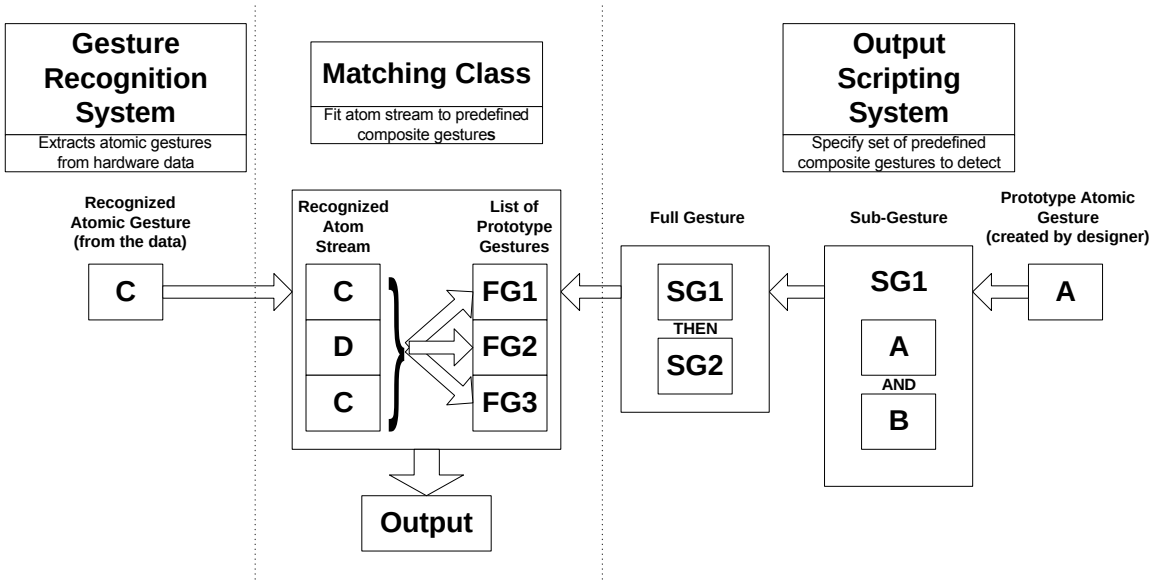

Figure 6-1: Flow of Data in the Scripting System

6.1.1 High-Level Structure

In general, designers will put together the full gestures of interest in a bottom-up fashion. The first step is to construct the prototype atomic gestures that are relevant to the application of interest, specifying the type (line, twist, etc.) - axis and number of peaks - and the desired range (if any) that the parameters should match. One or more atoms which should occur in a single timeframe are combined together into a sub-gesture. The designer can specify logical AND, OR and NOT existence operations between the atoms[1]. Finally, these sub-gestures are combined, in chronological order, into a full prototype gesture.

These gestures are held by a top level matching class, which receives atomic gestures from the recognition software as they occur. This matcher examines each prototype gesture to determine whether it appears in the stream of received atoms. If so, it will execute an output function which the user has associated with the full gesture. No limits are placed on the actions of this function, including accessing parameters from the received atoms. Examples range from printing text messages to the screen to providing commands to artificial intelligence-based creatures.

[1] *I.e.* Atomic gesture A AND Atomic gesture B occur simultaneously, either A OR B occurs, etc.

Note that while the gesture set is fairly straight-forward, how an individual gesture will be broken down, and the range of interest of the parameters, is not immediately clear. To make the construction of prototypes easier, a short helper script is provided that will output the atomic gestures as they occur. Automatic training systems which can learn scripts like the one described below are considered future work.

The code for the output scripting system is written in JPython[69], a variant of the Python scripting language designed to interface with Java. JPython provides a large amount of graphical user interface and networking code, simplifying the process of writing output functions.

A short block of example code, showing the instantiation of prototype gestures and simple output functions, can be found in Section 6.2 and figure 6-2.

6.1.2 Low-Level Code

The low-level code of the scripting system was built using a fairly simple class hierarchy and a number of generalized matching functions. The lowest class is the atomic gesture class itself. Each prototype gesture is specified with an index corresponding to an axis, the number of peaks specifying the gesture, and the parameters from the recognition system: α, β and direction. The prototype further takes a generalized matching function written by the designer. Equivalence of a prototype atom to one found by the recognition system is defined as follows: the axis and number of peaks must be identical, and the matching function must return true. A sample that does matches parameters within a fixed range is provided.

The prototype atoms are combined together to form sub-gesture detectors, which contain a list of atoms and a generalized matching function designating the necessary logical relation among the existence functions of the atoms. Logical AND, OR, and NOT are provided. Atoms in a sub-gesture must happen simultaneously, which is defined as any time overlap

between them[2]. The matching function examines a number of recognized atoms equal to the maximum number of atoms necessary for a match (*e.g.* 1 for OR, 2 for AND). If the atoms tested are equivalent to a set of prototype atoms in the sub-gesture which fulfill the matching function, then a match has been found.

A list of sub-gestures in chronological order forms a full gesture. The limitation on the time frame in which a sub-gesture may fall is two-fold: first, it cannot overlap a previous sub-gesture; second, it must start less than the width of the longest atom after the end of the previous sub-gesture.

A master matching class contains all of the user-defined gestures and controls the matching process. Whenever it receives a new atomic gesture from the recognition system, it prompts each gesture to look for a match in a vector containing the twenty most recent atoms (in all axes). If a gesture is found, the function associated with it within the matcher is executed. The master class also receives a tick command from the recognition code at a fixed interval to use as a time reference (if necessary).

The scripting system is general enough to allow virtually any combination of gestures to be detected, though there are a few limitations in the current instantiation. Because the system does not have knowledge of the gestures currently taking place, but only those completed, a delay (of a gesture width) is necessary to ensure that no other atom overlaps the atom(s) of interest. Also, the sub-gesture matching algorithm has difficulty with spurious atoms interspersed among those it seeks. While this case is fairly unlikely (it can only happen if an atom is concurrent with those in the sub-gesture), the only way to currently accommodate for it is to define a number of extra sub-gestures to account for that case. While this is algorithmically correct, it violates the spirit of the scripting language, which is ease of use. The ability to filter the packets before inclusion in the master matching array will be added in the next revision and should aid in resolving these difficulties.

[2]The NOT function enforces a time window after the last matched atom in which its parameter cannot occur.

6.2 Sample Script

To render the above discussion more concrete, a sample script is provided in figure 6-2 and will be discussed below. This script is designed to demonstrate a few simple gestures and to give the reader a sense of the syntax of JPython and the ease of use of the scripting system.

Lines 1 through 11 in the script define simple comparison functions to be used by the atomic gestures defined below. In this case, these functions test whether a gesture is within a reasonable range of the prototype in both α and width (β), as well as testing that they are in the same direction (in the case of the `CompDir` function). Lines 15 through 19 define four gestures. `TwistY` is a single-peaked gyro gesture, while the lines (`LineX`,`Y`,`Z`,`Up`) are double-peaked accelerometer gestures. Note that the only difference between `LineZ` and `LineUp` is the comparison function.

Next, sub-gestures are constructed (lines 21-25). `dAnyLine` will respond to any straight line, `dSweep` to a combination of an upwards movement and a twist, and `LineX` and `LineY` to movements in those planes. These are then combined to form full gestures (lines 27-31). Note that `g2` and `g3` differ only in order. Lines 34-39 demonstrate some simple output functions.

Finally, the full gestures above are added to the matching system, and are paired with the appropriate output function (line 42-46). Here the logic of the definitions of `g2` and `g3` becomes apparent. Since they are both simple L-shaped gestures, they both have the same output code, allowing the system to respond identically to all four such gestures.

```
## A few comparators
dAlpha = 20.0
dDuration = 20.0
def CompDir(myAlpha, myDuration, myDirection, theirAlpha, theirDuration, theirDirection):
  return (myDirection == theirDirection) and \
    (abs(myAlpha-theirAlpha)<dAlpha) and \
    (abs(myDuration-theirDuration)<dDuration)

def CompNoDir(myAlpha, myDuration, myDirection, theirAlpha, theirDuration, theirDirection):
  return (abs(myAlpha-theirAlpha)<dAlpha) and \
    (abs(myDuration-theirDuration)<dDuration)

## Atomic Gestures
## (axis, number of peaks, alpha, timestamp, width, direction, matching function)
TwistY = Subgesture(1, 1, 80, 0, 100, 1, CompNoDir)
LineX  = Subgesture(3, 2, 70, 0, 100, 1, CompNoDir)
LineY  = Subgesture(4, 2, 70, 0, 100, 1, CompNoDir)
LineZ  = Subgesture(5, 2, 70, 0, 100, 1, CompNoDir)
LineUp = Subgesture(5, 2, 70, 0, 100, 1, CompDir)

## Subgestures (List of atoms, number for a match, matching function)
dAnyLine = SubgestureDetector([LineX, LineY, LineZ], 1, gOR)
dSweep = SubgestureDetector([LineUp, TwistY], 2, gAND)
dLineX = SubgestureDetector([LineX], 1, gOR)
dLineY = SubgestureDetector([LineY], 1, gOR)

## Full gestures
g0 = Gesture([dAnyLine])
g1 = Gesture([dSweep])
g2 = Gesture([dLineX, dLineY])
g3 = Gesture([dLineY, dLineX])

## Output functions
def f0():
  print "Found a straight line"
def f1():
  print "Found a sweep"
def f2():
  print "Found an L"

## Constructing the matching system (full gesture, output function)
grs = GestureRecognitionSystem()
grs.addGesture(g0, f0)
grs.addGesture(g1, f1)
grs.addGesture(g2, f2)
grs.addGesture(g3, f2)
```

Figure 6-2: Sample Output Script

Chapter 7

Sample Application

As the easy creation of applications is a core feature of this project, a sample application will be examined in detail in this chapter. We begin with a discussion of the form gestures of interest must take to accommodate the limitations of the framework described in earlier sections. We then present the `(void*)` user interface, concentrating on the differing implementations of gesture recognition in the system shown at SIGGRAPH '99 and the revised system using the framework described herein.

7.1 Solvable Problems

While we attempted to create as general a system as possible, various features of both inertial sensing generally and this framework specifically impose certain restrictions on the gestural applications that can be implemented. These are detailed below.

The most important constraint to keep in mind is the lack of an absolute reference frame. As seen in Section 4.1.2, it is not possible, given the class of sensors used, to track orientation relative to a fixed frame for longer than five seconds. Therefore, it is necessary for the body reference frame to have some meaning associated to it, such that the application designer will be able to construct output functions which are appropriate regardless of the instrumented object's orientation in the world frame.

The second constraint is that the system cannot track multi-dimensional gestures, except for those which are separable in space and time. An arbitrary straight line can always be decomposed, and therefore can be recognized, while a rotating object tracing the same linear path would not be recognizable because the local axis of acceleration changes with time, making the decomposition impossible. However, the physical constraints of the gestural system often prevent such movements, especially in the case of human movement.

The final set of constraints are those imposed by the algorithms used for analysis and gesture recognition. Gestures must come to a complete stop to be found by the activity detection algorithm, making fluid motions and transitions hard to pick up. Note that as long as movement on one axis comes to a stop, the recognition of at least part of the gesture can be attempted. Also, the limited baseline tracking of the gesture recognition algorithm makes movements which combine translation and rotation difficult to measure. Research shows that subjects using 6 DOF controls in docking experiments[1] tend to control the rotational and translation DOFs separately[70], though the extension of this data to human hand gestures is not clear.

7.2 `(void*)`: A Cast of Characters

Consider a diner, surrounded by the void, on a hot starless night. Three haphazard fools at the counter, alone and tired. The trucker is the strong, silent type. A salesman sits fretting, half-expecting some unknown plague. A hipster awaits excitement, but seems unwilling to provide it himself. A Spirit descends. Suddenly the salesman twitches, looks at his legs in amazement and is dragged feet first away from the bar. The others eye their feet warily. In the center of the floor, the salesman tentatively begins to dance.

This is the setting for `(void*)`: A Cast of Characters.

Figure 7-1: Buns and Forks Interface to `(void*)`

7.2.1 Summary of Installation

Created by the Synthetic Characters Group at the MIT Media Laboratory, `(void*)` is an interactive installation which transports the user to a world of intelligent creatures which interact both with one another and the user, and have the ability to learn from those interactions. These characters are semi-autonomous in that they will obey external commands, but may make assumptions about the user's intentions based on their current context and will color how they execute the commands with their own desires. The reader is directed to [71, 72] for a detailed discussion of such creatures.

Drawing inspiration from Charlie Chaplin's famous 'buns and forks' scene in The Gold Rush[73], we created for this installation an input device whose outer casing is two bread rolls, each with a fork stuck near the end, thereby mimicking a pair of legs (see figure 7-1). By performing gestures with the interface, the user controls the lower half of the body of one of the three characters. However, their upper body movements and the quality of the lower body movement remains under the characters' autonomous control. The user can choose to control a specific character by placing the buns and forks on one of the three plates (left hand image in figure 7-1), where they are detected through the capacitive signalling system.

The buns and forks are an abstracted representation of feet and legs. This is well suited

[1]The task in a docking experiment is to manipulate a randomly situated object (in 3D) such that it overlaps an identical fixed object exactly.

Gesture	Description
Kick Forward (L/R/B)	Rotate bun forward
Kick Sideways (L/R/B)	Rotate bun outward
Twirl (L/R)	Rotate bun about fork
Twist (L/R)	Sweep a circle in XY plane
Knee Lift (L/R/B)	Raise and lower bun
Crossover	Move one bun across the other
Walk	Move buns forward and back, half-cycle out of phase

L=left bun, R=right bun, B=both

Table 7.1: Gesture Set for `(void*)`

to the installation - an interface that closely resembles a lower torso would have suggested greater control, while an even more abstracted interface such as a mouse would have been too distant. We note that our IMU is the best way to sense the motion of such objects, given that it is small enough to be embedded within the bread rolls used (which are $3 \times 2 \times 2\,\text{in}^3$) and can operate wirelessly. There is a separate IMU in each roll.

The target set of gestures to be recognized, as derived from the Chaplin scene, is listed in table 7.1. They include both one- and two-handed dynamic gestures. Notice that there are no gestures that imply navigation - absolute position and orientation are not used. We now consider two implementations of the gesture recognition for this set: an HMM-based system shown at SIGGRAPH '99 and a system based on the algorithms developed in this dissertation.

7.2.2 Then

The original gesture recognition for `(void*)` was done using an HMM-based system designed principally by Andrew Wilson and Marc Downie[74]. Each individual gesture was implemented using a separate model (or two, depending on the number of different acceptable interpretations of each gesture), with 5 to 10 training examples necessary. A null gesture was also provided. At any given time step, the user was assumed to be performing the gesture whose HMM had the highest log-likelihood after the subtraction of a fixed offset. Because of this winner-takes-all thresholding algorithm, the system could respond to

a gesture even before its peak in likelihood. The gesture recognition software was written in Java and, running on a single 400MHz Pentium computer could recognize gestures from 21 HMMs at 30Hz. Note that the time complexity of this system is linear with the number of models, and that the addition of a new gesture generally required that all the HMM threshold offsets be altered.

A few features of this system merit note. The first is that the time progression through the HMM states was done using a technique known as linear time warping. In this algorithm, the HMM is assumed to have spent an equal amount of time in each state and a range of total durations is tested to find the one with the highest probability. This technique helps avoid degenerate parses, but requires a fair amount of processing time. The second is an activity detection algorithm implemented by creating an HMM with a single zero mean, low covariance state. This model would drop below a certain probability whenever the device was moved. Using this information, it was possible to alleviate user frustration with the latency in recognition (since the gesture must be completed to be recognized) by having the characters look at their feet if the buns were moved at all.

This HMM-based gesture recognition system was used with reasonable success both at SIGGRAPH '99 and in demonstrations at the Media Laboratory. The overall recognition rate was on the order of 70%[75]. The main source of error was the overconstraint of the HMM parameters, leading to a lack of generality not only between users, but also between slight variations in the IMU position over time. Adjusting the offsets had some limited success in remedying this situation.

7.2.3 Now

The gesture recognition for `(void*)` was recently redone, with help from researchers in the Synthetic Characters Group, using the algorithms described in this dissertation and can be found in Appendix F. This system exploits the similarity between the gestures in the set to simplify the implementation procedure. The atomic gestures that make up a 'lift left' are exactly the same as those that make up a 'lift right', except on the other IMU, and a 'lift

both' is (by definition) the combination of the two. The atoms that each gesture caused was determined using the helper script mentioned earlier. The full script for this application can be found in Appendix F. This system was written in Java and JPython and ran at full rate (66 Hz) on a dual Pentium 933 MHz, using only 5% of the processor's capability. The complexity of the system is linear in the number of axes, and constant otherwise.

Note that because the IMU must come to a complete stop at the end of a gesture, the movement for walk was altered. Instead of continuously moving the buns back and forth a half-cycle out of phase, only a single cycle is completed. While this gesture is easily recognized, it is somewhat less than intuitive.

The most interesting feature to consider in this implementation is mutual exclusion. It turns out that the kick gestures contain the Y-accelerometer loopback atom as part of their structure. Unfortunately, this atom alone corresponds to a lift. Therefore, to avoid spurious triggers, the prototypes for the lift gestures uses the NOT function with a null operand to require that only the atom specified take place for a trigger to occur. This results in a slight extra latency, but that is unavoidable. Also, the researchers implementing the script found it quite easy to add a subgesture combination function, allowing lift both (for example) to be specified as a combination of lift right and lift left.

It should also be noted that the `(void*)` installation does have a subsystem that can make use of the parameterized information from the recognition system. The motor system used in this installation has the ability to blend between prototypical animations[76] - currently, it blends between happy and sad versions based on the characters state. This technique could use the α parameter (for example) to provide the position in a blend between weak and forceful versions of the movements.

An informal study was done with six people who were familiar with the `(void*)` gesture set. Each subject was reminded of the new gesture set, and then was asked to perform a number of the gestures following a script. The results are shown in table 7.2, which lists the gesture, the number of times it was correctly performed (out of two), and the reason for missed gestures. An overall recognition rate of 87% was found. There is not enough

Gesture	User 1	User 2	User 3	User 4	User 5	User 6
Kick Forward Right	2	2	2	2	2	2
Kick Forward Left	2	2	2	2	2	2
Kick Forward Both	2	2	2	2	2	2
Knee Lift Right	2	2	2	2	2	2
Knee Lift Left	2	2	1[a]	2	2	2
Knee Lift Both	2	2	2	1[b]	2	2
Kick Sideways Right	2	2	2	2	2	2
Kick Sideways Left	2	2	2	2	2	2
Kick Sideways Both	2	2	1[b]	2	2	2
Twist Left	2	2	2	2	2	2
Twist Right	2	2	2	2	2	1[c]
Twirl Left	0[d]	2	2	2	1[d]	2
Twirl Right	2	2	2	1[d]	2	1[d]
Walk (Version 1)	2	1[d]	2	1[d]	2	1[d]
Walk (Version 2)	2	2	2	1[d]	2	2
Crossover	2	2	2	2	2	2

2 trials per gesture per user

Table 7.2: Results of Informal Testing of New Algorithms

[a]Resulted in no gestures.
[b]RF transmission problem. Not used in calculation.
[c]Resulted in kick forward.
[d]2-peaked rather than three peaked gesture.

data to claim that the new system performed better than the previous version, though it is reasonable to conclude that it performed at least as well.

Most of the missed gestures were the result of a lack of deliberateness in the motion - weak or off-axis gestures are hard to detect with this scheme. This is the cause of the large number of kick forward gestures found in error, and the few times when nothing was found. The 2-peaked gestures seen in place of 3-peaked gestures were likely caused by the limit on minimum peak size. Altering the thresholds on the various algorithms can help solve these problems, but will result in more spurious gestures being detected.

7.2.4 Comparison

The strength of the new system lies, by design, in its speed and ease of use. It occupies few processor cycles, freeing them up for more complex output tasks. The script took less than two hours to write, and will remain valid even after changes to the underlying gesture recognition scheme (within reasonable limits). In contrast, the HMM-based system was not able to accept data at even half the update rate, and needed to be retrained after even cursory change to the algorithm. Further, the new system proved more robust in admittedly less than complete testing, which is the expected result of using an algorithm that exploits fundamental *a priori* knowledge about the data.

The generality of HMMs does provide two notable benefits in this case. Firstly, they can achieve continuous recognition, while our algorithm requires that gestures be distinct. Secondly, complex relations between the axes are handled automatically in the (high-dimensional) gesture state space, rather than manually through in the scripting system. However, these advantages are not enough to overwhelm the obvious benefits of the new generalized system.

Chapter 8

Conclusions

In this chapter, the entirety of the project is considered. We begin with a summary of this dissertation; the work to date. Comments on the short term improvements to be made to the current framework and on the long-term potential of self-contained inertial gesture recognition are given.

Overall, the work in this dissertation demonstrated that inertial measurement can be used to acquire rich data about human movement, that we can devise efficient algorithms for using this data in gesture recognition, and that the concept of a parameterized atomic gesture recognition has merit. Further we show that a framework combining these three can be easily used by designers to create robust applications.

8.1 Summary

The framework we constructed uses an inertial measurement unit to collect data from the object of interest and wirelessly transmit it to a personal computer. The data is then analyzed with a windowed variance algorithm to find periods of activity, and the generalized gesture recognition algorithms are applied to those periods. These gestures are recognized in an atomic form on an axis-by-axis basis using a number of physically based constraints, and

those atoms can be combined into more complicated gestures using an output scripting system. This system was designed for use by application designers and allows output functions to be linked to specific gesture inputs. Each portion of the framework is considered below.

The gesture sensing for these applications is accomplished using a compact inertial measurement unit. The device is a cube 1.25 inches on a side, and measures a full six degrees-of-freedom in motion using three orthogonal accelerometers and three orthogonal gyroscopes. Data is collected at 15 ms intervals, and is transmitted wirelessly, to allow for the greatest possible range of applications. The hardware runs on a pair of batteries with a 50 hour life, and currently costs approximately US$300, in prototype quantities.

A thresholded windowed variance algorithm for activity detection was developed. This routine has low latency and is fairly effective at finding areas of interest in the data, though it imposes the constraint that each gesture must begin and end with an area of constant acceleration or rotational rate. Two other analysis schemes were considered. Kalman filtering, which is commonly used in inertial tracking applications, was deemed ineffective over the relatively long time scales of interest, and frequency space filtering did not provide any useful information.

Next, the parameterized gesture recognition algorithm considers the data on an axis-by-axis basis, and is further simplified by considering only atomic gestures - those which cannot reasonably be decomposed into simpler gestures. Therefore, only a very small number of gestures need to be recognized on each spatial axis, and the same algorithm can be applied to each. The recognition method for accelerometers is fairly straight-forward. Using the assumption that constant acceleration implies zero velocity (in the case of human gesture), we note that once a baseline has been removed the net integral across any accelerometer gesture should be nil (within a reasonable threshold). This condition does not hold for gyroscope data, and therefore gestures are separated from incoherent activity by a threshold on their absolute integral. Width, length, number of peaks, and direction are the parameters collected for each gesture. This system was used rather than more conventional state-based methods such as Hidden Markov Models, which are processor intensive and have difficulties analyzing dynamic (rather than absolute) data.

Given these atomic gestures, a designer must have some method of piecing them together into full gestures. An output scripting language is provided for this purpose. The designer can create prototypes of various atomic gestures of interest. These are then combined into subgestures of one or more atoms that should occur simultaneously. An ordered sequence of subgestures produces a full prototype gesture and the occurrence of a full gesture in the stream of recognized atoms can be associated with an arbitrary output function. Recognized atoms can be matched to prototypes based on any or all of their parameters, and they can be combined together into subgestures using logical operands.

Because our goal is to eventually create stand-alone devices with this functionality, the design constraints were different than those in vision-based and tethered gesture systems. The algorithms used in this framework were chosen for their low algorithmic complexity and low latency. While not as general as those commonly used in gesture recognition, they provide similar performance at a much lower cost in terms of processor cycles. Further, it was shown that the algorithms developed can be used with much lower data rates and accuracies. The gesture recognition portion of `(void*)`: A Cast of Characters was reimplemented using this scripting language. The new version ran much faster than the previous version, used less processing power, and performed at least as well.

8.2 Future Work

As the intent of this dissertation work was to provide a proof of principle, there are a number of possible improvements to both the hardware and the software that are worth noting. Further, there are fundamental design questions to be answered regarding stand-alone devices, and they are discussed here as well.

Possibilities for the next revision of the inertial hardware are described in Chapter 2 and the three areas for improvement will just be touched on here. Sensor size should be reduced, hopefully moving to smaller accelerometers and MEMS gyroscopes in the near future. The device should also be more flexible in shape, to allow for a greater range of applications. Finally, the non-inertial input capability should be extended by providing other modes of

capacitive sensing (useful for hand-held applications) and an external header for additional wired sensor inputs.

In the analysis stage, there is the possibility of combining both activity detection and tracking using a Kalman filter. We noted that accurate tracking can be achieved only with a high enough rate of external update. Using the assumption that constant acceleration implies zero velocity, the rate of increase of position error can be greatly slowed, and magnitude of the constant acceleration (assumed to be from tilt) can be used to correct two degrees of freedom in the orientation. Further, any zero crossing in the velocity will demark the start or end of a gesture in the world frame. However, with no heading information available, the angular error in the plane perpendicular to gravity will tend to increase quickly.

Turning to gesture recognition, the need to remove the baseline is the greatest concern with our algorithm. It requires that atomic gestures be completed before any recognition can begin, and leads to missed gestures and false positives in areas of orientation change. The latency could be overcome by doing a speculative integration from the start of a period of activity, assuming a constant baseline at the current value. The integration ends when the sum returns to zero, indicating a successful parse, or appears to be increasing without bounds, indicating a failure due to change in baseline. For gestures parsed after completion, it should be possible to remove the baseline more accurately by using information from the associated gyroscope channel. Specifically, by checking for overlapping gestures between the channels, the start and end point of a change in orientation can be determined more accurately. This algorithm was attempted using a linear interpolation between the start and stop points of a gyroscope gesture, as long as those points were within an accelerometer gesture, but proved ineffective. It will be necessary to consider changes in orientation beyond the time extent the acceleration gesture and higher-order interpolation routines for this technique to succeed.

The main area for future expansion of this work is in the design of stand-alone devices. Improvements in the hardware and algorithms will be necessary, particularly to provide greater processing power and storage. However, larger questions lie in the uses of such a device. Will the gestural lexicon created here be rich enough to describe not only interesting

motion, but interesting variations therein? How can effective output be accomplished in a compact low-power inexpensive device? Can the device itself learn not just the motion of interests, but the most effective feedback mechanism and the contexts in which to offer (or withhold) comment?

Once accomplished, the benefits of such a system would be enormous. Otherwise ordinary objects could have a sense of their own motion and the ability to communicate this knowledge completely *in situ*, with no wires or infrastructure. We conclude now with a short discussion of such an object.

8.3 Future Applications

To comment on the future possibilities of inertial measurement as not just a user interface, but a learning tool, we consider the somewhat whimsical example involving a general inertial unit that can be attached to any ordinary object. It will sense 6 degrees-of-freedom, recognize gestures using a generalized scheme, and have a small wireless link with which to receive commands.

On your favorite cooking show, the chef is baking a chocolate cake, and is demonstrating how to properly fold the batter. You follow along with the program and prepare the cake in question, only to have it come out flat. But why? It turns out that the batter was folded incorrectly.

Of course, you could prepare the cake again, but it would be sheer happenstance if you did so correctly, because two key portions of the learning process are missing. The feedback received has only a weak temporal coupling to the action, which makes learning more difficult[77]. Further, the feedback received is based on the outcome, rather than the quality of your action, which would be far more valuable[78]. By placing the generalized IMU on the spatula, this situation can be improved. Transmitted along with the cooking program will be a script of atomic gestures which represent proper folding technique. These gestures can be sent to the IMU via a low bandwidth link because of the abstraction in the gesture

recognition progress. Then, as you fold, the IMU will produce pleasing tones if you do so correctly, and cacophonous notes otherwise. These sounds could further give some sense of the mistake being made. Not only will this cake turn out well, but so will those you produce in the future.

This is just one possible example: other applications include physical therapy systems where a user is alerted when they make awkward movements, and interfaces which can train users on how to use them via feedback. Such systems would color our everyday life, changing the way we learn by allowing perception and expertise to be contained in the device itself.

Appendix A

Abbreviations and Symbols

∅	Diameter
ADC	Analog to digital converter.
DOF	Degree of freedom.
EKF	Extended Kalman Filter.
FIR	Finite impulse response (digital filter).
HMM	Hidden Markov Model.
IC	Integrated circuit (chip).
IIR	Infinite impulse response (digital filter).
IMU	Inertial measurement unit.
KF	Kalman Filter.
MEMS	Micro-electromechanical systems.
PCB	Printed circuit board.
RF	Radio frequency.
RS-232	Serial transmission protocol.
SNR	Signal to noise ratio.

Appendix B

Glossary

Activity Detection The finding of areas of (possible) interest in a data stream.

dB Power ratio expressed as $10\log(\cdot)$.

DC Balancing The processing of data to ensure that it contains an equal number of ones and zeros, and sometimes also to limit runs. Usually done prior to RF transmission to ensure that the receiver maintains the proper threshold level between high and low bits.

Embedded Code Software which is executed on a microcontroller. It is usually restricted in terms of its use of memory and floating-point operations, and is often executed at low speeds.

Gesture The space-time curve associated with a motion of interest.
Atomic gestures are the fundamental unit of the gesture recognition system and cannot be further decomposed.
Sub-gestures are a combination of simultaneous atomic gestures.
Full gestures are a combination of consecutive sub-gestures.

Inertial Sensors Sensors which measure their own motion through inertial reatction, sensitive to either acceleration or rotational rate.

IMU Inertial Measurement Unit; any systems with three axes of accelerometers and three axes of gyroscopes. Minimum set of sensors necessary to completely describe motion in three dimensions.

Microcontroller Small microprocessor, usually featuring additional integrated features such as an analog to digital converter and/or a serial communication hardware (UART).

Reference Frame The (arbitrary) axes which form a coordinate system. The world reference frame is considered fixed to a certain location, while the local (or body) reference frame moves with an object.

Appendix C

Schematics, PCB Layouts and Drawings

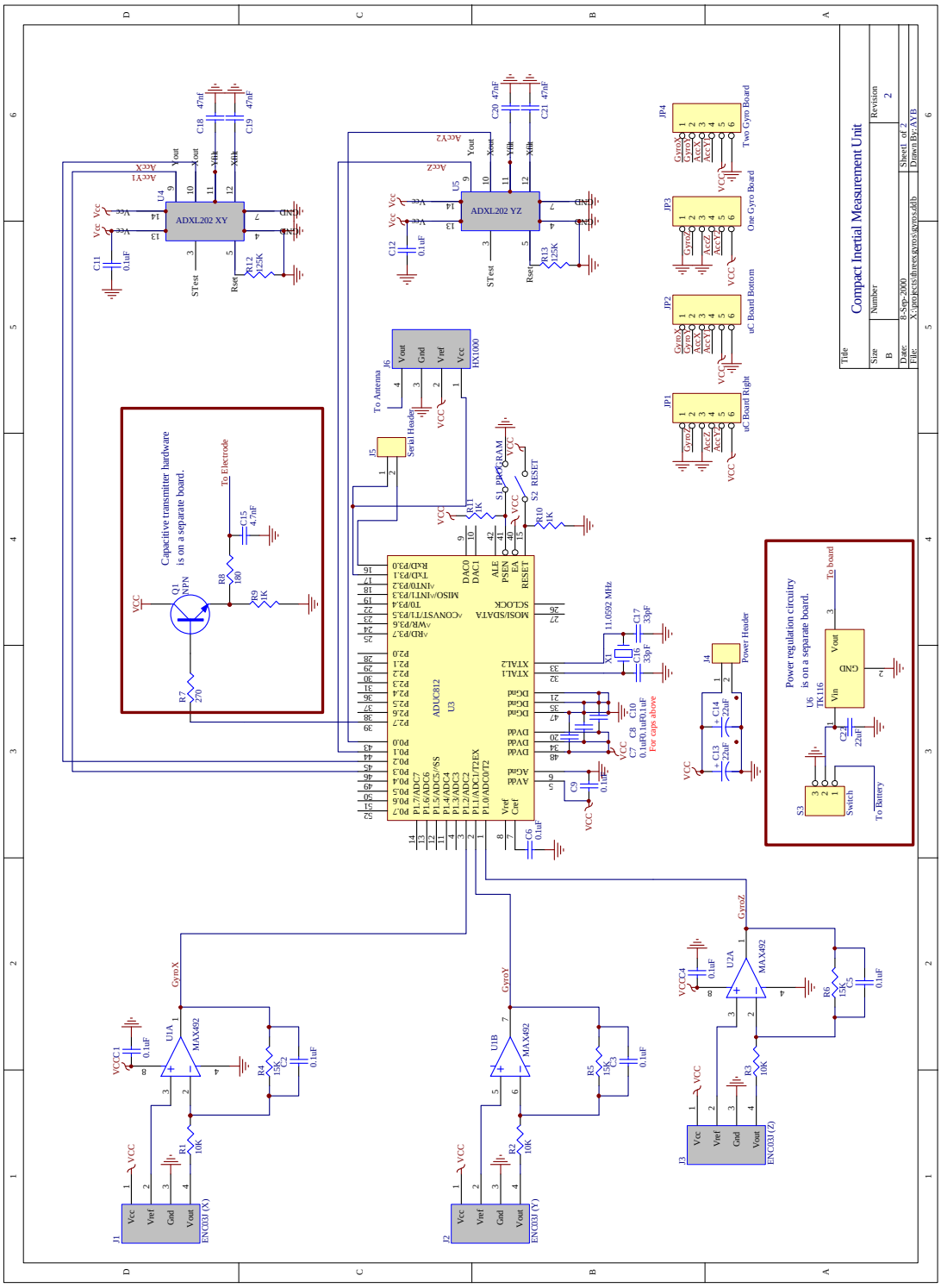

Figure C-1: Main IMU Schematic

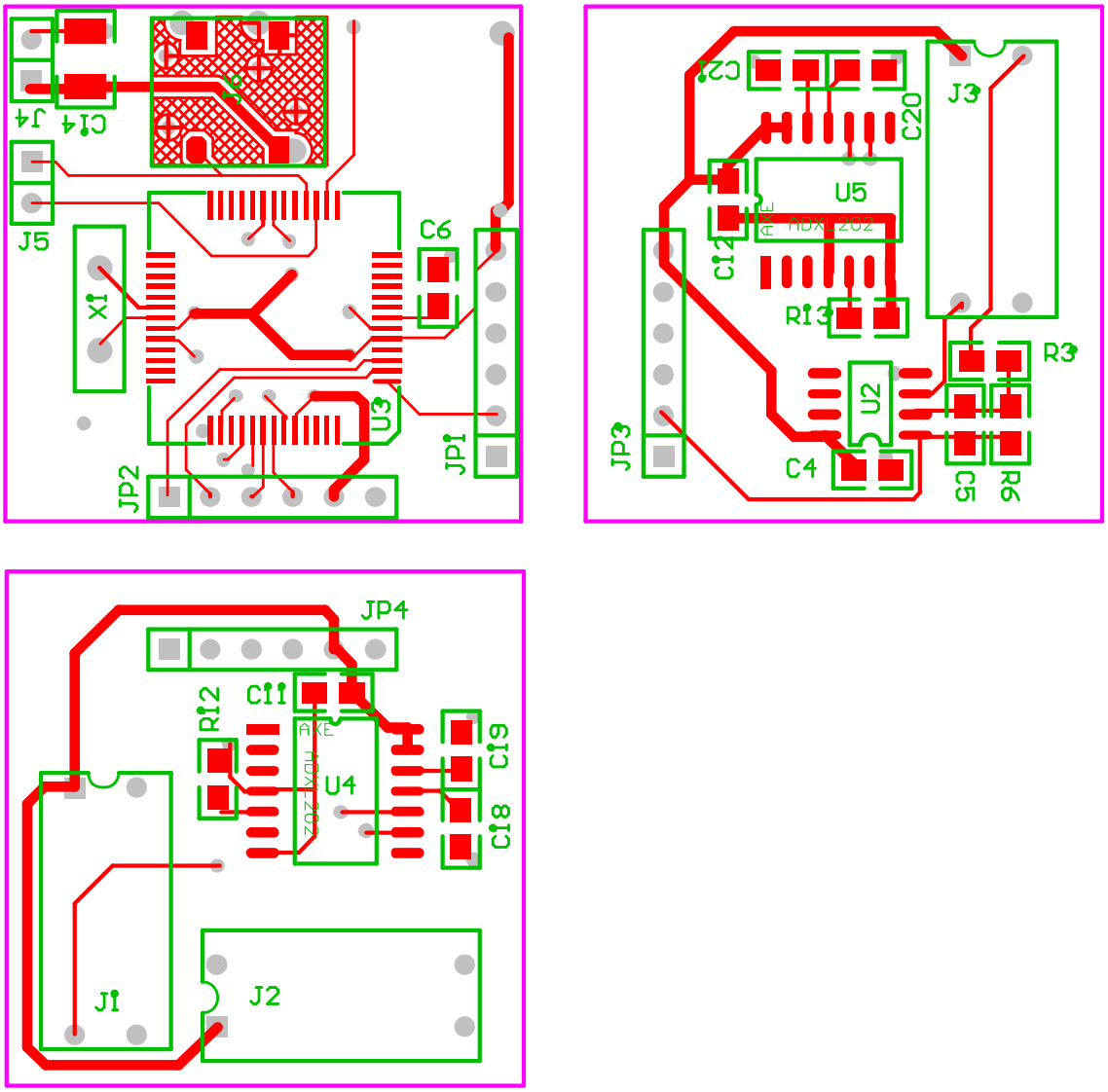

Figure C-2: Main IMU PCB - Top

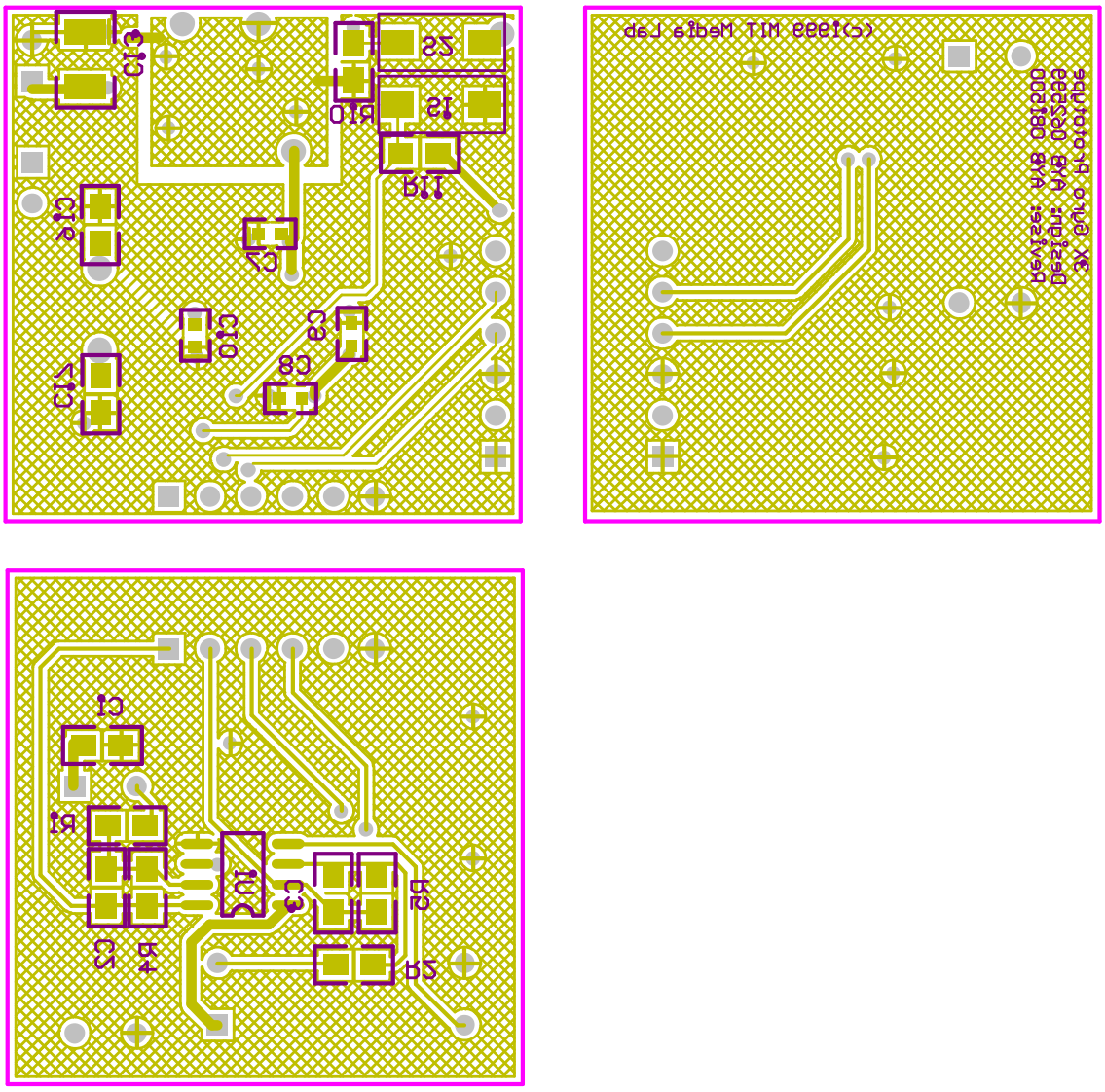

Figure C-3: Main IMU PCB - Bottom

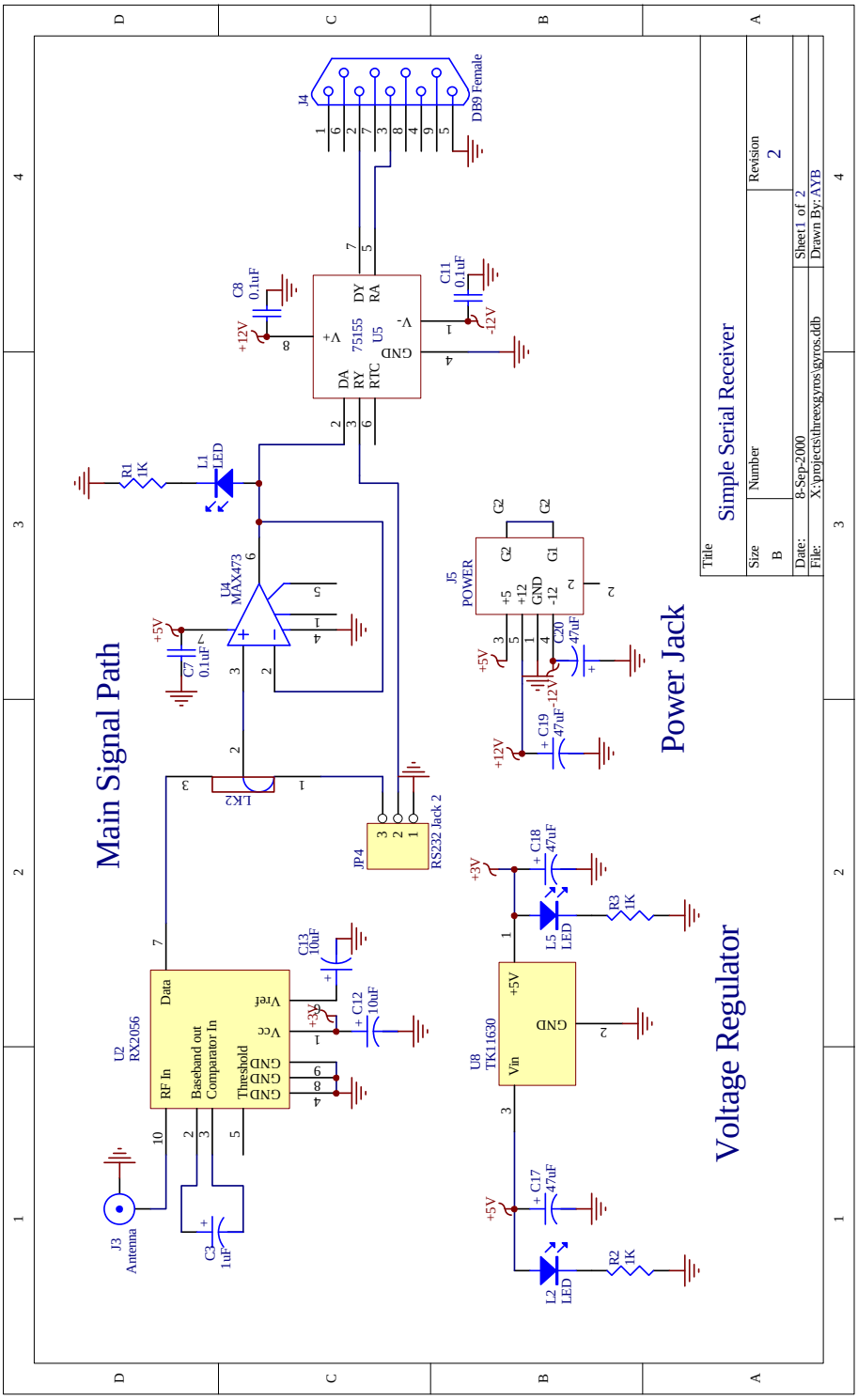

Figure C-4: Receiver Schematic

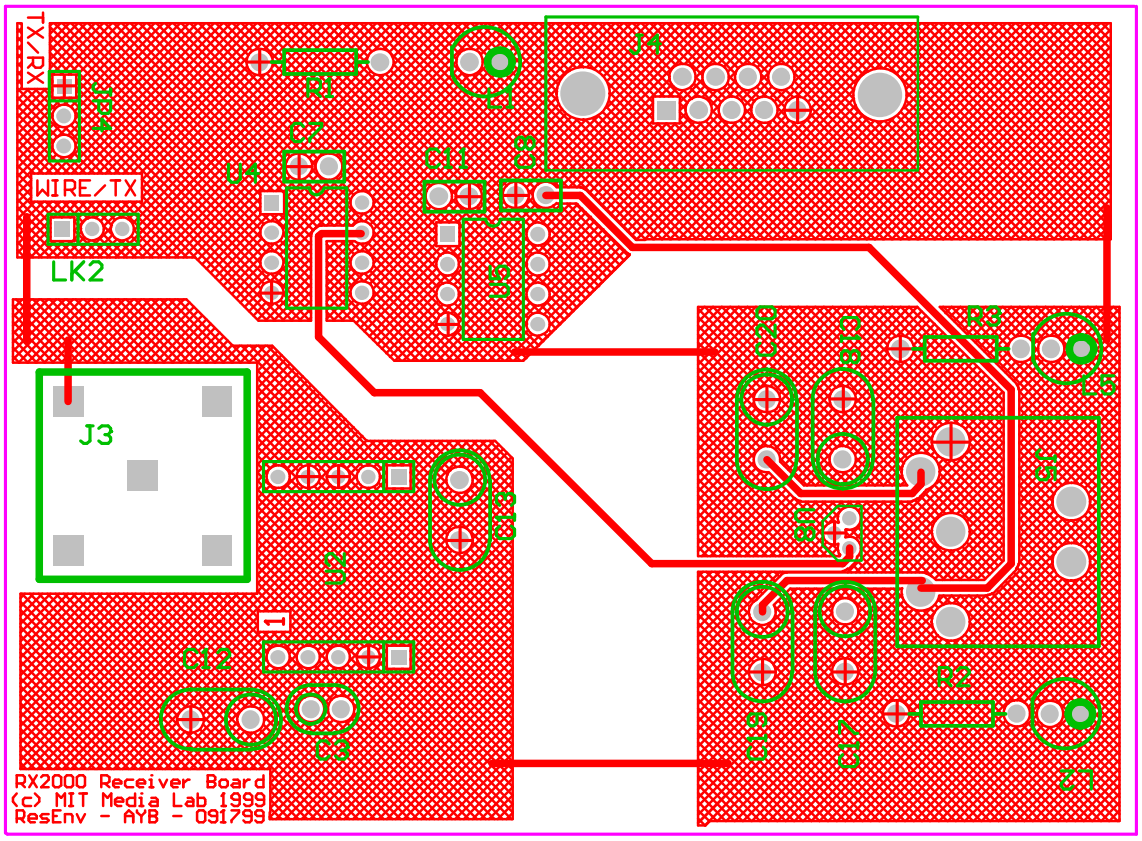

Figure C-5: Receiver PCB - Top layers

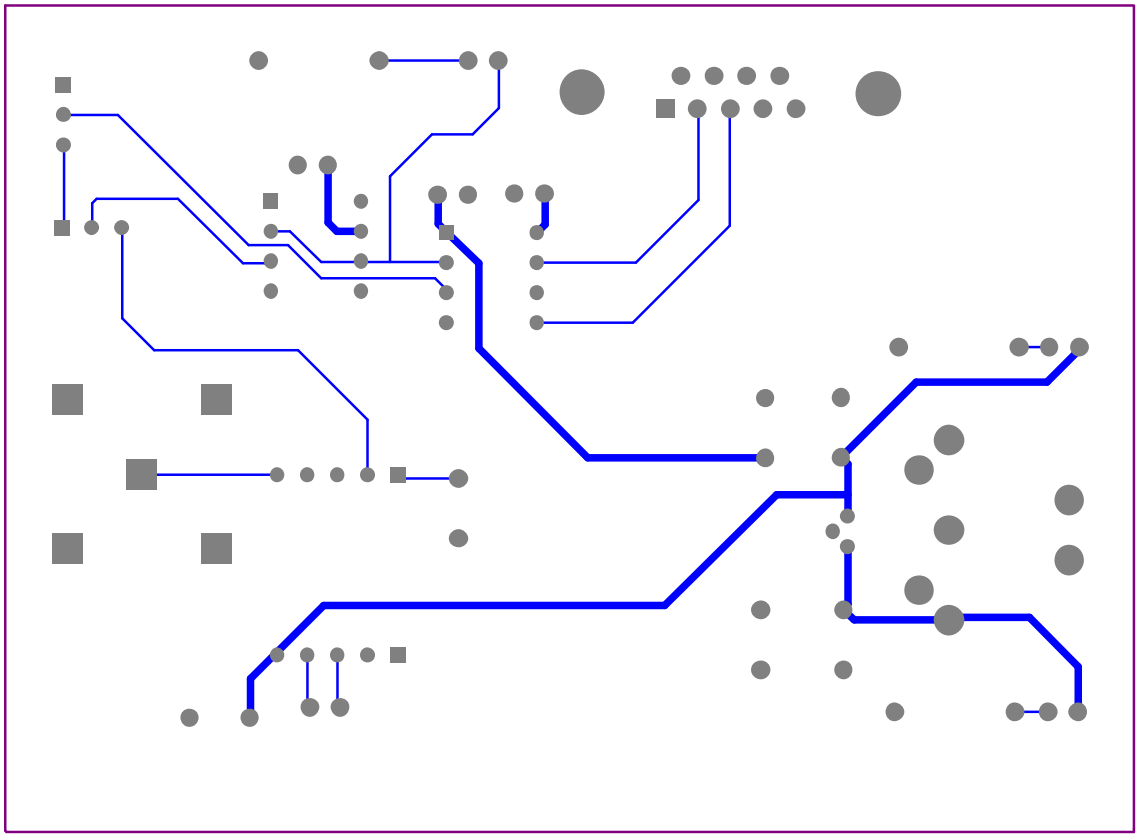

Figure C-6: Receiver PCB - Bottom layers

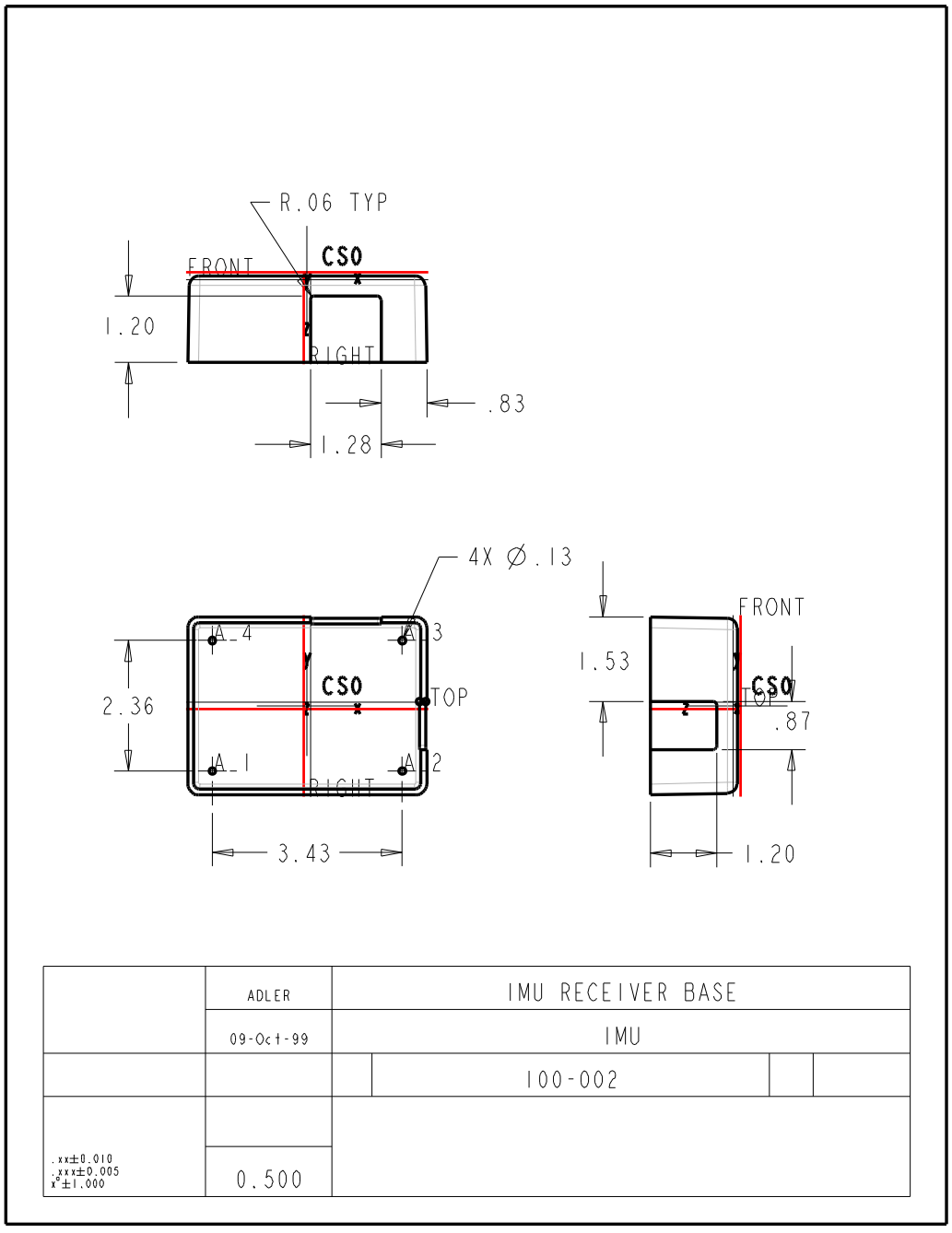

Figure C-7: Receiver Base Drawing

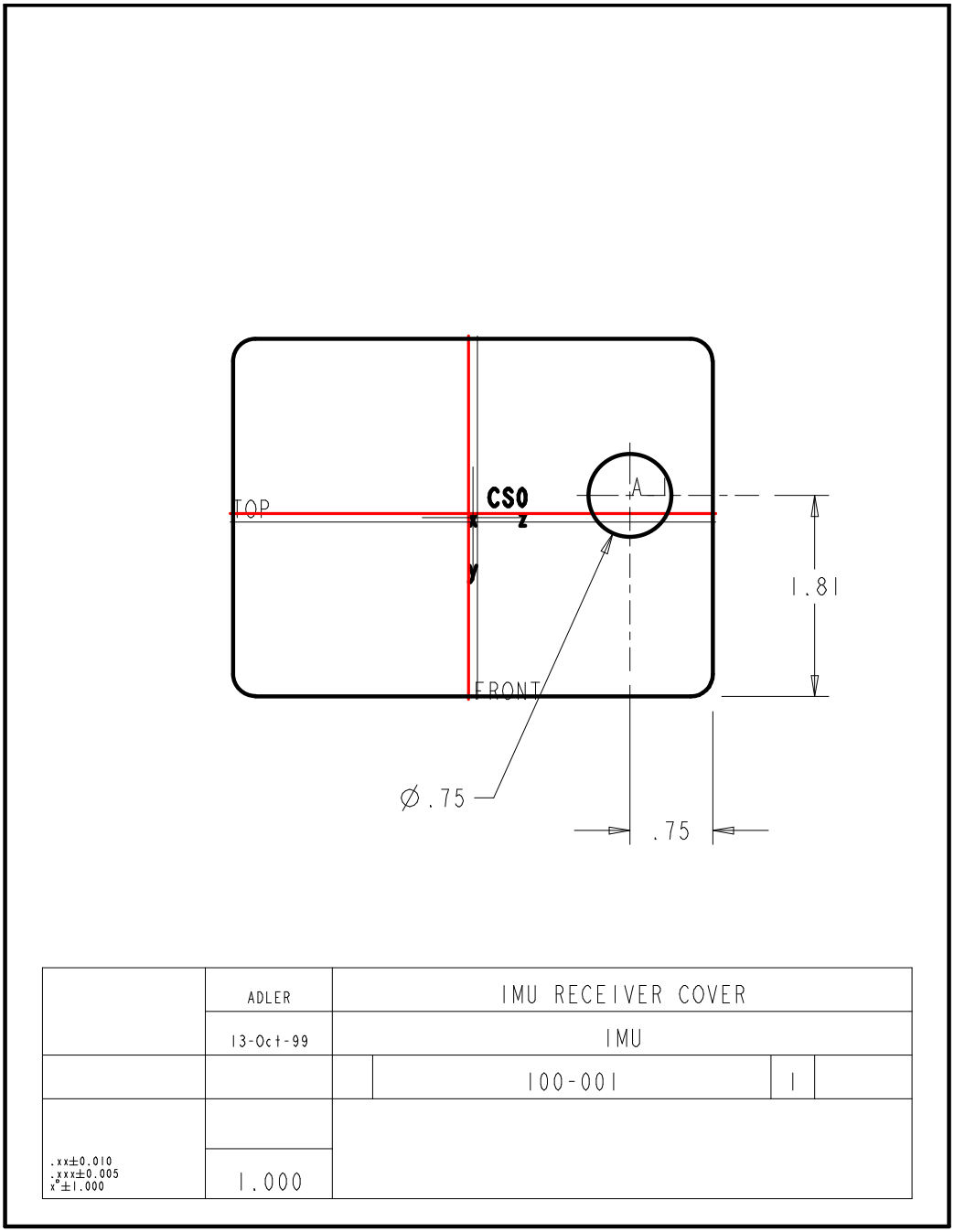

Figure C-8: Receiver Cover Drawing

Appendix D

ADuC812 Embedded Code

```
/* thesis.c — Final version of IMU microcontroller code. */

#pragma CODE /* pragma lines can contain state C51 */
#include <stdio.h>
#include <ADuC812.h>
#include <intrins.h>
#include "812.h"

//Are we using ONEY accelerometer value or TWOY values?
#define ONEY
//Are we using the PAN? (Uncomment first line and the frequency of choice)
//#define PAN
//#define PAN50k
//#define PAN20k

//Forward declerations
void init(void);
void collectAnalog(void);
void collectAccels(void);
void checkPower(void);
void transmitData(void);
void trans3Nibbles(unsigned char,unsigned char);
void transByte(unsigned char);
void pulsePAN(void);

// Declare ports
sbit AccelX   = 0x82; //Pin P0.2
sbit AccelY1  = 0x83; //Pin P0.3
sbit AccelY2  = 0x80; //Pin P0.0
sbit AccelZ   = 0x81; //Pin P0.1
sbit PANpin   = 0xA7; //Pin P2.7

// Plug in lookup table
idata unsigned char lookup[64] = {23,27,29,39,43,45,46,51,53,54,57,
                                  58,71,75,77,78,83,86,89,90,92,99,
                                  101,102,105,106,108,113,114,116,
                                  135,139,141,142,147,149,150,153,
                                  154,156,163,165,166,169,170,172,
                                  177,178,180,184,195,197,198,201,202,
                                  204,209,210,212,216,225,226,228,232};

// Declare ADC Channels
#define GYROXCHAN   SCONV2
#define GYROYCHAN   SCONV1
#define GYROZCHAN   SCONV0
#define TEMPCHAN    SCONVTEMP

// New Header Versions
#define POWER_GOOD  0xFF
#define POWER_MED   0xFE
#define POWER_SUCKS 0xFD

// Time out thresholds
// For rise to fall timer
#define TIMERF  0xFF
// For rise to rise timer
#define TIMERR  0xFF

// Misc defines
#define VERSION 0x0C

#ifdef PAN
  #ifdef PAN50k
  // Offset is 256-number of instruction cycles for one period
  #define PAN_OFFSET  0xF7
  // Number of periods for 1 ms
  #define PAN_ITER    100
  #endif
  #ifdef PAN20k
  // Offset is 256-number of instruction cycles for one period
  #define PAN_OFFSET  0xE9
  // Number of periods for 1 ms
  #define PAN_ITER    40
  #endif
#endif

// Variables
lu gyrox;
lu gyroy;
lu gyroz;
lu XT1;
lu Y1T1;
lu Y2T1;
lu ZT1;
bit under293,under263;
bit in_cycle,pause,dirty;

// Setup function
void init(void)
{
  // Setup serial comm (using timer 1)
  SCON = UART8BIT; // Setup serial port
  TMOD = T1TIMER | T18BITRELOAD; // Setup up serial timer
  SET_TIMER(1,0xFD,0x00); // 9600 Baud timeout for 11.0592 MHz crystal
  PCON = DOUBLEBAUD; // 19.2Kbps
  TI = 1; // Ready to send
  START_TIMER(1);

  // Setup ADC
  ADCCON1 = ADCNORMAL | MCLKDIV4 | AQIN1;
  ADCCON3 = ADCCON3CLR;

  // Setup power low interrupt
  under293 = 0;
  under263 = 0;
  PSMCON = TRIP293;

  //Setup accelerometer timer 0
  TMOD |= T0TIMER | T016BIT;

  //Setup timed interrupt
  T2CON = T2TIMER | T2RELOAD;   // Reloading timer
  SET_TIMER(2,0xC9,0xFF);       // Cycle start value is 0xC9FF since each tick
  SET_TIMER2_RELOAD(0xC9,0xFF); // is 1.085 us and (0x10000-0xC9FF)x1.085us = 15ms

  //Setup interrupts
  IE = T2OVERFLOW;
  EA=1;         // Make sure global interrupts are on

  //Blank variables
  gyrox.word = gyroy.word = gyroz.word = 0;
  XT1.word = Y1T1.word = Y2T1.word = ZT1.word = 0;
```

```
  pause=in_cycle=dirty=0;
}

void collectAnalog(void)
{
  //Collect data
  ADCCON2 = GYROXCHAN;
  while(ADCCON3&0x80); // Wait for operation to finish
  STORE_ADC(gyrox);

  ADCCON2 = GYROYCHAN;
  while(ADCCON3&0x80);
  STORE_ADC(gyroy);

  ADCCON2 = GYROZCHAN;
  while(ADCCON3&0x80);
  STORE_ADC(gyroz);
}

void collectAccels(void)
{
//  unsigned char timeout; // Needed for timer macros

  // Time the X pulse
  TIMER_RISE_TO_FALL_NOTIMEOUT(0,AccelX,TIMERF);
  STORE_TIMER(0,XT1);
  // Time the Y1 pulse
  TIMER_RISE_TO_FALL_NOTIMEOUT(0,AccelY1,TIMERF);
  STORE_TIMER(0,Y1T1);
#ifdef TWOY
  // Time the Y2 pulse
  TIMER_RISE_TO_FALL_NOTIMEOUT(0,AccelY2,TIMERF);
  STORE_TIMER(0,Y2T1);
#endif
  // Time the Z pulse
  TIMER_RISE_TO_FALL_NOTIMEOUT(0,AccelZ,TIMERF);
  STORE_TIMER(0,ZT1);
}

void checkPower(void)
{
  if (!(PSMCON & POWERSTATMASK))
  {
    if(under293)
    {
      // Set flag and turn power monitor off
      under263 = 1;
      PSMCON = PSMOFF;
    }
    else
    {
      // Set flag and new trip point
      under293 = 1;
      PSMCON = TRIP263;
    }
  }
}

void trans3Nibbles(unsigned char high, unsigned char low)
{
  // DC Balance each six bit block
  unsigned char in,out;

  in = low&0x3F; //first six bits
  out = lookup[in];
  SEND(out);
  in = low>>6 | high<<2;
  out = lookup[in];
  SEND(out);
}

void transmitData(void)
{
  // Power/header byte
  if(under263)
  {
    SEND(POWER_SUCKS);
  }
  else if(under293)
  {
    SEND(POWER_MED);
  }
  else
  {
    SEND(POWER_GOOD);
  }
  SEND(0x01); // Generic second header byte
  //Send analog data
  trans3Nibbles(gyrox.byte[0],gyrox.byte[1]);
  trans3Nibbles(gyroy.byte[0],gyroy.byte[1]);
  trans3Nibbles(gyroz.byte[0],gyroz.byte[1]);
  //Send accel data
  trans3Nibbles(XT1.byte[0],XT1.byte[1]);
  trans3Nibbles(Y1T1.byte[0],Y1T1.byte[1]);
#ifdef TWOY
  trans3Nibbles(Y2T1.byte[0],Y2T1.byte[1]);
#endif
  trans3Nibbles(ZT1.byte[0],ZT1.byte[1]);
}

#ifdef PAN
void pulsePAN()
// This cannot run much faster than about 50kHz, because of loop instruction count
// Assume that timer0 may be running on entry, but that value is not important
{
  unsigned char oldTMOD,i;

  // Setup timer
  STOP_TIMER(0);
  oldTMOD = TMOD; //Store sfr
  TMOD = (TMOD & 0xFC) | T08BITRELOAD; // Blank timer 0 control register,
                                       // reset it as 8 bit auto-reload timer
  SET_TIMER(0,PAN_OFFSET,0x00); // Set timer0 so that it resets at PAN interval

  //Do it
  for(i=0;i<PAN_ITER;i++)
  {
    TF0=0; // reset flag
    START_TIMER(0); //Redundant after first loop, but who cares?
    while(!TF0); // wait for overflow
    PANpin = ~PANpin;//flip PAN pin
```

```
    }
    PANpin = 0; //Set PAN pin low

    // Clean up
    STOP_TIMER(0);
    TMOD = oldTMOD; //Restore sfr
}
#endif

void time_int() interrupt 5
{
  // Timer 2 interrupt routine. Clear flag. Clear pause. Set dirty is in cycle
  TF2=0;
  pause=0;
  dirty=in_cycle;
}

void main(void)
{
  init();
  while(1)
  {
      START_TIMER(2); // So that it doesn't interrupt the first cycle
      dirty=0;
      in_cycle=1;
      pause=1;
      // Get accelerometer data
      collectAccels(); // The PUTs are interleaved into the Accel calls (3 per call)
      // Get gyro data
      PUT(0x55);
      collectAnalog();
      // Is the power OK?.
      PUT(0x55);
      checkPower();
      in_cycle=0;
      // Pad data a bit
      while(pause) {PUT(0x55);}
      if(!dirty) {transmitData();}
#ifdef PAN
      pulsePAN();
#endif
  }
}
```

```
/* 812.h -- Stab at a collection of useful macros for the */
/*          ADuC812 microController.  */

/* Include SFR and sbit definitions */
#include "812SFR.h"

// Structures and unions
// Combination of two bytes and a word. Helpful for math.
typedef union {
  int  word;
  unsigned char byte[2];
} lu;

// RS232
/* Blocking send. Check to make sure that the previous send isn't still
   going, Reset TI bit, send. */
#define SEND(x) while(!TI);TI=0;SBUF=x;
/* Non-blocking send. Check if previous send is done, if so, send */
#define PUT(x)  if(TI) {TI=0;SBUF=x;}


// Basic Timer Stuff
#define START_TIMER(num)  TR##num=1;
#define STOP_TIMER(num)   TR##num=0;
#define SET_TIMER(num,high,low) TL##num=low;TH##num=high;
#define STORE_TIMER(num,un) un.byte[0]=TH##num;un.byte[1]=TL##num;
#define SET_TIMER2_RELOAD(high,low) RCAP2H=high;RCAP2L=low;

// More timer stuff
// This is not really the place for the PUTs, but there is no other way.
#define TIMER_RISE_TO_FALL(num,bitname,time) \
    timeout=time; \
    STOP_TIMER(num); \
    SET_TIMER(num,0x00,0x00); \
    PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}     \
    if(timeout!=0) {timeout=time;} \
    PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==1){break;}}     \
    START_TIMER(num);   \
    if(timeout!=0) {timeout=time;} \
    PUT(0x55); \
    while(1) { timeout-=1; if(timeout==0 || bitname==0){break;}}     \
    STOP_TIMER(num); \
    if(timeout==0) {SET_TIMER(num,0x00,0x00);}
// Same as above, but without timeouts.
#define TIMER_RISE_TO_FALL_NOTIMEOUT(num,bitname,time) \
    STOP_TIMER(num); \
    SET_TIMER(num,0x00,0x00); \
    PUT(0x55); \
    while(bitname==1);     \
    PUT(0x55); \
    while(bitname==0);     \
    START_TIMER(num);   \
    PUT(0x55); \
    while(bitname==1);     \
    STOP_TIMER(num);
// ADC storage stuff
#define STORE_ADC(un)   un.byte[0]=ADCDATAH&0x0F;un.byte[1]=ADCDATAL;
```

Appendix E

MATLAB Code

```
function [gest,vars] = findgests3(vec,half,high,low,vars)

%findgests.m
%
% Finds the gestures in a data set. Half is half the window size.
% This method uses a high and low threshold on a windowed variance.
% Gestures are returned in an array as their start and stop points.
%

usenew = ~exist('vars');
len = 2*half;
sumsq = cumsum(vec.^2);
summa = cumsum(vec);
numgests=1;
inGest = 0;

if(usenew)
   vars = zeros(size(vec));
end

for i=half+1:length(vec)-half
   if(~inGest)
      %Find start
      if(usenew)
         vars(i) = (sumsq(i+half)-sumsq(i-half))/(len-1)
                     - (summa(i+half)-summa(i-half))^2/(len*(len-1));
      end
    if(vars(i) > high)
         gest(numgests,1) = i;
         inGest = 1;
    end
   else
      %Find stop
      if(usenew)
         vars(i) = (sumsq(i+half)-sumsq(i-half))/(len-1)
                     - (summa(i+half)-summa(i-half))^2/(len*(len-1));
      end
    if(vars(i) < low)
         gest(numgests,2) = i;
         inGest = 0;
         numgests=numgests+1;
     end
   end
end
if(inGest) % If still in a gesture
   gest(numgests,2)=length(vec)-half+1;
end
```

```
function [goodgests,badgests]= parsegests3(vec,gests);
%
% parsegests.m (For use with accels)
%
% Takes in an array of gesture start - stop points and spits out a parse of those
% potential gestures as an array of parameters of good gestures and an array of
% parameters of bad gestures.
%

% Constants
consthresh = 0.30; % How much slop is allowed?
ditherthresh = 50; % When is it dithering and not a second gesture?

% Misc
numgests = length(gests);
numgood = 1;
numbad = 1;
lastbad = 0;

for i=1:length(gests)
   % Seperate out the gesture
   gstr = floor(gests(i,1));
   gstp = floor(gests(i,2));
   gestvec = vec(gstr:gstp);
   % Check the consistency
   [newgood good newbad bad] = gestcons3(gestvec,consthresh);
   % Add to lists
   for i=1:newgood
      goodgests(numgood,1) = gstr + good(i,1);
      goodgests(numgood,2) = gstr + good(i,2);
      goodgests(numgood,3:5) = good(i,3:5);
      numgood = numgood+1;
   end
   for i=1:newbad
      badgests(numbad,1) = gstr + bad(i,1);
      badgests(numbad,2) = gstr + bad(i,2);
      badgests(numbad,3:5) = bad(i,3:5);
      numbad = numbad+1;
   end
    % Check for dither
    if(newbad==1 & numbad>2 & ((badgests(numbad-1,1) - badgests(numbad-2,2))<ditherthresh))
      % Possible combo gest, do it all again
      newstr = badgests(numbad-2,1);
      newstp = badgests(numbad-1,2);
      gestvec = vec(newstr:newstp);
      numbad = numbad - 2;
      % Check the consistency
    [newgood good newbad bad] = gestcons3(gestvec,consthresh);
    % Add to lists
       for i=1:newgood
       goodgests(numgood,1) = newstr + good(i,1);
        goodgests(numgood,2) = newstr + good(i,2);
          goodgests(numgood,3:5) = good(i,3:5);
       numgood = numgood+1;
       end
    for i=1:newbad
        badgests(numbad,1) = newstr + bad(i,1);
          badgests(numbad,2) = newstr + bad(i,2);
       badgests(numbad,3:5) = bad(i,3:5);
        numbad = numbad+1;
                                                                    end
                                                                end
                                                            end
```

```
function [numgood,good,numbad,bad] = gestcons3(gestvec,constresh)

%
% gestcons.m (For use with accel data)
%
% Takes in a gestures and checks the consistency.
% Takes an array that contains a potential gestures and the consistency threshold.
% Returns the number of good gestures, an array with their parameters
% and the same for bad gestures.
%

% Misc constants
numpeaks= 1;
winsize = floor(10);
sumthresh = 500;
minSize = 3000;

% Remove the baseline
numpoints = length(gestvec);
start = gestvec(1);
stop = gestvec(numpoints);
inc = (stop-start)/(numpoints-1);
if(inc~=0)
   gestvec = gestvec - (start:inc:stop)';
else
   gestvec = gestvec - start;
end

% Integrate and find peaks
summa(1) = gestvec(1);
absum(1) = abs(gestvec(1));
totsum = 0;
totabs = 0;
window = winsize;

% Calculate the integrals
for i=2:numpoints
   summa(i) = summa(i-1) + gestvec(i);
   abssum(i) = abssum(i-1) + abs(gestvec(i));
   if((window==0) & ((winsize+i)<numpoints) & sign(gestvec(i))~=sign(gestvec(i-1)))
      if((absum(i) - totabs)>sumthresh)
        % End of good-sizedpeak
          sums(numpeaks) = summa(i) - totsum;
       sumabs(numpeaks) = abssum(i) - totabs;
        ends(numpeaks) = i;
          totsum = summa(i);
       totabs = abssum(i);
        numpeaks = numpeaks + 1;
          window = winsize;
      else
         %Combine it will previous peak if there is one
         if(numpeaks>1)
            sums(numpeaks-1) = sums(numpeaks-1) + summa(i) - totsum;
           sumabs(numpeaks-1) = sumabs(numpeaks-1) + abssum(i) - totabs;
        ends(numpeaks-1) = i;
           totsum = summa(i);
        totabs = abssum(i);
         end
      end
   else
      % Decrease the window counter.
      if(window~=0)
         window = window-1;
      end
   end
end
% Deal with last peak
if((abssum(numpoints)-totabs)>sumthresh | numpeaks == 1)
   sums(numpeaks) = summa(numpoints) - totsum;
   sumabs(numpeaks) = abssum(numpoints) - totabs;
   ends(numpeaks) = numpoints;
else
   numpeaks = numpeaks-1;
   sums(numpeaks) = sums(numpeaks)+summa(numpoints)-totsum;
   sumabs(numpeaks) = sumabs(numpeaks)+abssum(numpoints)-totabs;
   ends(numpeaks)= numpoints-1;
end
% Add dummy
sums(numpeaks+1) = sums(numpeaks);

% Check whole gesture.
if(totabs==0)
   consistency = 1;
else
   consistency = abs(summa(numpoints)/abssum(numpoints));
end
if(consistency  < constresh & abssum(numpoints) > minSize)
   % If it is a good gesture, then see if you can break it down
   % Makes sure it is a single movement (peaks flip polarity)
   numgood = 0;
   good = [];
   numbad = 0;
   bad = [];
   lastbreak = 1;
   for i=2:numpeaks+1
      if(sign(sums(i))==sign(sums(i-1))) % If same polarity, break gestures
         totsum = sum(sums(lastbreak:i-1));
         totabs = sum(sumabs(lastbreak:i-1));
         %Check consistency
         if(totabs==0)
            consistency = 1;
         else
            consistency  = abs(totsum/totabs);
         end
         %%%
         %if(lastbreak~=1)
         %  len= ends(i-1) - ends(lastbreak-1) + 2;  % Start of gest
         %else
         %  len = ends(i-1);
         %end
         %consistency = abs(totsum/sqrt(len));
         %%%
         if (consistency < constresh  & (i-lastbreak)>1)
            numgood = numgood+1;
            if(lastbreak~=1)
               good(numgood,1) = ends(lastbreak-1)+1;  % Start of gest
            else
               good(numgood,1) = 1;
            end
            good(numgood,2) = ends(i-1);  % end of gest
            good(numgood,3) = i-lastbreak; % number of peaks
            good(numgood,4) = totabs; % total mass
```

```
            good(numgood,5) = consistency; % consistency
         else
            % If it is no good, leave it
            numbad = numbad+1;
            if(lastbreak~=1)
               bad(numbad,1) = ends(lastbreak-1)+1;  % Start of gest
            else
               bad(numbad,1) = 1;
            end
            bad(numbad,2) = ends(i-1);  % end of gest
            bad(numbad,3) = i-lastbreak; % number of peaks
            bad(numbad,4) = totabs; % total mass
            bad(numbad,5) = consistency; % consistency
         end
         % Deal with remaining stuff
         lastbreak = i;
      end
   end
else
   % If it is no good, leave it
   numgood = 0;
   good = [];
   numbad = 1;
   bad(numbad,1) = 1;
   bad(numbad,2) = numpoints;
   bad(numbad,3) = numpeaks; % number of peaks
   bad(numbad,4) = abssum(numpoints); % total mass
   bad(numbad,5) = consistency; % consistency
end
return;
```

```
function [goodgests,badgests]= parseggests(vec,gests);
%
% parseggests.m (For use with gyros)
%
% Takes in an array of gesture start - stop points and spits out a parse of those
% potential gestures as an array of parameters of good gestures and an array of
% parameters of bad gestures.
%

% Constants
massthresh = 7500; % When is it noise and when is it a real gesture?
ditherthresh = 30; % When is it dithering and not a second gesture?

% Misc
numgests = length(gests);
numgood = 1;
numbad = 1;
lastbad = 0;

for i=1:length(gests)
   % Seperate out the gesture
   gstr = floor(gests(i,1));
   gstp = floor(gests(i,2));
   gestvec = vec(gstr:gstp);
   % Check the consistency
   [newgood good newbad bad] = gestmass(gestvec,massthresh);
   % Add to lists
   for i=1:newgood
      goodgests(numgood,1) = gstr + good(i,1);
      goodgests(numgood,2) = gstr + good(i,2);
      goodgests(numgood,3:5) = good(i,3:5);
      numgood = numgood+1;
   end
   for i=1:newbad
      badgests(numbad,1) = gstr + bad(i,1);
      badgests(numbad,2) = gstr + bad(i,2);
      badgests(numbad,3:5) = bad(i,3:5);
      numbad = numbad+1;
   end
    % Check for dither
    if(newbad==1 & numbad>2 & ((badgests(numbad-1,1) - badgests(numbad-2,2))<ditherthresh))
      % Possible combo gest, do it all again
      newstr = badgests(numbad-2,1);
      newstp = badgests(numbad-1,2);
      gestvec = vec(newstr:newstp);
      numbad = numbad - 2;
      % Check the consistency
    [newgood good newbad bad] = gestmass(gestvec,massthresh);
    % Add to lists
       for i=1:newgood
       goodgests(numgood,1) = newstr + good(i,1);
        goodgests(numgood,2) = newstr + good(i,2);
          goodgests(numgood,3:5) = good(i,3:5);
       numgood = numgood+1;
       end
    for i=1:newbad
        badgests(numbad,1) = newstr + bad(i,1);
          badgests(numbad,2) = newstr + bad(i,2);
       badgests(numbad,3:5) = bad(i,3:5);
        numbad = numbad+1;
                                                                    end
                                                                end
                                                            end
```

```
function [numgood,good,numbad,bad] = gestmass(gestvec,massthresh)

%
% gestmass.m (For use with gyro data)
%
% Takes in a gestures and checks the mass.
% Takes in an array which could contain a gesture and the mass threshold.
% Returns the number of good gestures, an array with their parameters
% and the same for bad gestures.
%

% Misc constants
numpeaks= 1;
winsize = 10;
sumthresh = 1000;

% Remove the baseline
numpoints = length(gestvec);
start = gestvec(1);
stop = gestvec(numpoints);
inc = (stop-start)/(numpoints-1);
if(inc~=0)
   gestvec = gestvec - (start:inc:stop)';
end

% Integrate and find peaks
summa(1) = gestvec(1);
absum(1) = abs(gestvec(1));
totsum = 0;
totabs = 0;
window = winsize;

% Calculate the integrals
for i=2:numpoints
   summa(i) = summa(i-1) + gestvec(i);
   abssum(i) = abssum(i-1) + abs(gestvec(i));
   if((window==0) & ((winsize+i)<numpoints) & sign(gestvec(i))~=sign(gestvec(i-1)))
      if((abssum(i) - totabs)>sumthresh)
        % End of good-sizedpeak
          sums(numpeaks) = summa(i) - totsum;
       sumabs(numpeaks) = abssum(i) - totabs;
        ends(numpeaks) = i;
          totsum = summa(i);
       totabs = abssum(i);
        numpeaks = numpeaks + 1;
          window = winsize;
      else
         %Combine it will previous peak if there is one
         if(numpeaks>1)
            sums(numpeaks-1) = sums(numpeaks-1) + summa(i) - totsum;
           sumabs(numpeaks-1) = sumabs(numpeaks-1) + abssum(i) - totabs;
        ends(numpeaks-1) = i;
           totsum = summa(i);
        totabs = abssum(i);
         end
      end
   else
      % Decrease the window counter.
      if(window~=0)
         window = window-1;
      end
   end
end
% Deal with last peak
if((abssum(numpoints)-totabs)>sumthresh | numpeaks == 1)
   sums(numpeaks) = summa(numpoints) - totsum;
   sumabs(numpeaks) = abssum(numpoints) - totabs;
   ends(numpeaks) = numpoints;
else
   numpeaks = numpeaks-1;
   sums(numpeaks) = sums(numpeaks)+summa(numpoints)-totsum;
   sumabs(numpeaks) = sumabs(numpeaks)+abssum(numpoints)-totabs;
   ends(numpeaks)= numpoints-1;
end
% Add dummy
sums(numpeaks+1) = sums(numpeaks);

% Check whole gesture
if(abssum(numpoints) > massthresh)
   % If it is a good gesture, then see if you can break it down
   % Makes sure it is a single movement (peaks flip polarity)
   numgood = 0;
   good = [];
   numbad = 0;
   bad = [];
   lastbreak = 1;
   for i=2:numpeaks+1
      if(sign(sums(i))==sign(sums(i-1))) % If same polarity, break gestures
         totsum = sum(sums(lastbreak:i-1));
         totabs = sum(sumabs(lastbreak:i-1));
         %Get consistency
         if(totabs==0)
            consistency = 1;
         else
            consistency  = abs(totsum/totabs);
         end
         if (totabs > massthresh)
            numgood = numgood+1;
            if(lastbreak~=1)
               good(numgood,1) = ends(lastbreak-1)+1;  % Start of gest
            else
               good(numgood,1) = 1;
            end
            good(numgood,2) = ends(i-1);  % end of gest
            good(numgood,3) = i-lastbreak; % number of peaks
            good(numgood,4) = totabs; % total mass
            good(numgood,5) = consistency; % consistency
         else
            % If it is no good, leave it
            numbad = numbad+1;
            if(lastbreak~=1)
               bad(numbad,1) = ends(lastbreak-1)+1;  % Start of gest
            else
               bad(numbad,1) = 1;
            end
            bad(numbad,2) = ends(i-1);  % end of gest
            bad(numbad,3) = i-lastbreak; % number of peaks
            bad(numbad,4) = totabs; % total mass
            bad(numbad,5) = consistency; % consistency
         end
         % Deal with remaining stuff
         lastbreak = i;
```

```
      end
   end
else
   % If it is no good, leave it
   numgood = 0;
   good = [];
   numbad = 1;
   bad(numbad,1) = 1;
   bad(numbad,2) = numpoints;
   bad(numbad,3) = numpeaks; % number of peaks
   bad(numbad,4) = abssum(numpoints); % total mass
   bad(numbad,5) = 1; % dummy consistency
end
return;
```

Appendix F

`(void*)` Recognition Script

The following script was implemented by Matt Berlin, Jesse Gray and Ari Benbasat, and is designed to recognize the `(void*)` gesture set:

```
from GestureRecognitionSystem import *
from Gesture import *
from Subgesture import *
from SubgestureDetector import *
from conjunctiveFunctions import *
from java.util import Random
from research.ayb.threeX.gesture import SubgestureDataConduit


## the default subgesture comparator
dAlpha = 1.1
dDuration = 10.0
def defaultComparator(myAlpha, myDuration, myDirection, \
                      theirAlpha, theirDuration, theirDirection):
    return (myDirection == theirDirection) and \
        (abs(myAlpha-theirAlpha)<dAlpha) and \
        (abs(myDuration-theirDuration)<dDuration)

def directionComparator(myAlpha, myDuration, myDirection, \
                        theirAlpha, theirDuration, theirDirection):
    return (myDirection == theirDirection)


##### subgestures follow the naming convention sxyz, where
#####   x indicates the device - l(eft) or r(ight)
#####   y indicates the sensor type - g(yro) or a(ccelerometer)
#####   z indicates the positive direction - f(orward), b(ackward), u(p),
#####                                        d(own), l(eft), or r(ight)
##### a number following the subgesture name indicates the number of peaks

### left bun subgestures
slgf = Subgesture(0, 0, 2, 0, 0, 1, directionComparator)
slgu = Subgesture(1, 0, 2, 0, 0, 1, directionComparator)
slgl = Subgesture(2, 0, 2, 0, 0, 1, directionComparator)
slab = Subgesture(3, 0, 3, 0, 0, 1, directionComparator)
slad = Subgesture(4, 0, 3, 0, 0, 1, directionComparator)
slal = Subgesture(5, 0, 3, 0, 0, 1, directionComparator)
slab4 = Subgesture(3, 0, 4, 0, 0, 1, directionComparator)
slad4 = Subgesture(4, 0, 4, 0, 0, 1, directionComparator)
slal4 = Subgesture(5, 0, 4, 0, 0, 1, directionComparator)
slab2 = Subgesture(3, 0, 2, 0, 0, 1, directionComparator)
slad2 = Subgesture(4, 0, 2, 0, 0, 1, directionComparator)
slal2 = Subgesture(5, 0, 2, 0, 0, 1, directionComparator)


slgb = Subgesture(0, 0, 2, 0, 0, 0, directionComparator)
slgd = Subgesture(1, 0, 2, 0, 0, 0, directionComparator)
slgr = Subgesture(2, 0, 2, 0, 0, 0, directionComparator)
slaf = Subgesture(3, 0, 3, 0, 0, 0, directionComparator)
slau = Subgesture(4, 0, 3, 0, 0, 0, directionComparator)
slar = Subgesture(5, 0, 3, 0, 0, 0, directionComparator)
slaf4 = Subgesture(3, 0, 4, 0, 0, 0, directionComparator)
slau4 = Subgesture(4, 0, 4, 0, 0, 0, directionComparator)
slar4 = Subgesture(5, 0, 4, 0, 0, 0, directionComparator)
slaf2 = Subgesture(3, 0, 2, 0, 0, 0, directionComparator)
slau2 = Subgesture(4, 0, 2, 0, 0, 0, directionComparator)
slar2 = Subgesture(5, 0, 2, 0, 0, 0, directionComparator)


### right bun subgestures
srgf = Subgesture(6, 0, 2, 0, 0, 1, directionComparator)
srgu = Subgesture(7, 0, 2, 0, 0, 1, directionComparator)
srgl = Subgesture(8, 0, 2, 0, 0, 1, directionComparator)
srab = Subgesture(9, 0, 3, 0, 0, 1, directionComparator)
srad = Subgesture(10, 0, 3, 0, 0, 1, directionComparator)
sral = Subgesture(11, 0, 3, 0, 0, 1, directionComparator)
srab4 = Subgesture(9, 0, 4, 0, 0, 1, directionComparator)
srad4 = Subgesture(10, 0, 4, 0, 0, 1, directionComparator)
sral4 = Subgesture(11, 0, 4, 0, 0, 1, directionComparator)
srab2 = Subgesture(9, 0, 2, 0, 0, 1, directionComparator)
srad2 = Subgesture(10, 0, 2, 0, 0, 1, directionComparator)
sral2 = Subgesture(11, 0, 2, 0, 0, 1, directionComparator)

srgb = Subgesture(6, 0, 2, 0, 0, 0, directionComparator)
srgd = Subgesture(7, 0, 2, 0, 0, 0, directionComparator)
srgr = Subgesture(8, 0, 2, 0, 0, 0, directionComparator)
sraf = Subgesture(9, 0, 3, 0, 0, 0, directionComparator)
srau = Subgesture(10, 0, 3, 0, 0, 0, directionComparator)
srar = Subgesture(11, 0, 3, 0, 0, 0, directionComparator)
sraf4 = Subgesture(9, 0, 4, 0, 0, 0, directionComparator)
srau4 = Subgesture(10, 0, 4, 0, 0, 0, directionComparator)
srar4 = Subgesture(11, 0, 4, 0, 0, 0, directionComparator)
sraf2 = Subgesture(9, 0, 2, 0, 0, 0, directionComparator)
srau2 = Subgesture(10, 0, 2, 0, 0, 0, directionComparator)
srar2 = Subgesture(11, 0, 2, 0, 0, 0, directionComparator)


## dNothing detects the absence of subgestures
dNothing = SubgestureDetector([],1,gNOT)

dLiftLeft = SubgestureDetector([slau], 1, gOR)
dLiftRight = SubgestureDetector([srau], 1, gOR)
dLiftBoth = SubgestureDetector([slau, srau], 2, gAND)

dKickForwardLeft = SubgestureDetector([slgr], 1, gOR)
dKickForwardRight = SubgestureDetector([srgr], 1, gOR)
dKickForwardBoth = SubgestureDetector([slgr, srgr], 2, gAND)

dWalk1 = SubgestureDetector([slgr, srgl], 2, gAND)
dWalk2 = SubgestureDetector([srgr, slgl], 2, gAND)

dKickSidewaysLeft = SubgestureDetector([slgf], 1, gOR)
dKickSidewaysRight = SubgestureDetector([srgb], 1, gOR)
dKickSidewaysBoth = SubgestureDetector([slgf, srgb], 2, gAND)
dCrossover = SubgestureDetector([slgb, srgf], 2, gAND)

dTwistLeft = SubgestureDetector([slgu, slgd], 1, gOR)
dTwistRight = SubgestureDetector([srgu, srgd], 1, gOR)
## detectorORANDOR returns the logical AND of two single-argument gOR detectors
dTwistBoth = SubgestureDetector([dTwistLeft, dTwistRight], 2, detectorORANDOR)

## a twirl consists of a 2, 3, or 4-peaked accelerometer gesture on two axes simultaneously
dTwirlLeft1 = SubgestureDetector([slal, slal4, slal2], 1, gOR)
dTwirlLeft2 = SubgestureDetector([slaf, slaf4, slaf2], 1, gOR)
dTwirlLeft = SubgestureDetector([dTwirlLeft1, dTwirlLeft2], 2, detectorORANDOR)
dTwirlRight1 = SubgestureDetector([srar, srar4, srar2], 1, gOR)
dTwirlRight2 = SubgestureDetector([sraf, sraf4, sraf2], 1, gOR)
dTwirlRight = SubgestureDetector([dTwirlRight1, dTwirlRight2], 2, detectorORANDOR)


def fLiftLeft():
    print "Gesture recognized!---lift left"
```

```
    def fLiftRight():
        print "Gesture recognized!---lift right"
    def fLiftBoth():
        print "Gesture recognized!---lift both"

    def fKickForwardLeft():
        print "Gesture recognized!---kick forward left"
    def fKickForwardRight():
        print "Gesture recognized!---kick forward right"
    def fKickForwardBoth():
        print "Gesture recognized!---kick forward both"

    def fWalk():
        print "Gesture recognized!---walk"

    def fKickSidewaysLeft():
        print "Gesture recognized!---kick sideways left"
    def fKickSidewaysRight():
        print "Gesture recognized!---kick sideways right"
    def fKickSidewaysBoth():
        print "Gesture recognized!---kick sideways both"
    def fCrossover():
        print "Gesture recognized!---crossover"

    def fTwistLeft():
        print "Gesture recognized!---twist left"
    def fTwistRight():
        print "Gesture recognized!---twist right"
    def fTwistBoth():
        print "Gesture recognized!---twist both"

    def fTwirlLeft():
        print "Gesture recognized!---twirl left"
    def fTwirlRight():
        print "Gesture recognized!---twirl right"


    ## a lift is a lift subgesture with no following subgesture
    gLiftLeft = Gesture([dLiftLeft, dNothing])
    gLiftRight = Gesture([dLiftRight, dNothing])
    gLiftBoth = Gesture([dLiftBoth])

    gKickForwardLeft = Gesture([dKickForwardLeft])
    gKickForwardRight = Gesture([dKickForwardRight])
    gKickForwardBoth = Gesture([dKickForwardBoth])

    gWalk1 = Gesture([dWalk1])
    gWalk2 = Gesture([dWalk2])

    gKickSidewaysLeft = Gesture([dKickSidewaysLeft])
    gKickSidewaysRight = Gesture([dKickSidewaysRight])
    gKickSidewaysBoth = Gesture([dKickSidewaysBoth])
    gCrossover = Gesture([dCrossover])

    gTwistLeft = Gesture([dTwistLeft])
    gTwistRight = Gesture([dTwistRight])
    gTwistBoth = Gesture([dTwistBoth])

    gTwirlLeft = Gesture([dTwirlLeft])
    gTwirlLeft2 = Gesture([dTwirlLeft2])
    gTwirlRight = Gesture([dTwirlRight])
    gTwirlRight2 = Gesture([dTwirlRight2])


    grs = GestureRecognitionSystem()
    grs.addGesture(gLiftLeft, fLiftLeft)
    grs.addGesture(gLiftRight, fLiftRight)
    grs.addGesture(gLiftBoth, fLiftBoth)

    grs.addGesture(gKickForwardLeft, fKickForwardLeft)
    grs.addGesture(gKickForwardRight, fKickForwardRight)
    grs.addGesture(gKickForwardBoth, fKickForwardBoth)

    ## the two versions of walk both call the same output function
    grs.addGesture(gWalk1, fWalk)
    grs.addGesture(gWalk2, fWalk)

    grs.addGesture(gKickSidewaysLeft, fKickSidewaysLeft)
    grs.addGesture(gKickSidewaysRight, fKickSidewaysRight)
    grs.addGesture(gKickSidewaysBoth, fKickSidewaysBoth)
    grs.addGesture(gCrossover, fCrossover)

    grs.addGesture(gTwistLeft, fTwistLeft)
    grs.addGesture(gTwistRight, fTwistRight)
    grs.addGesture(gTwistBoth, fTwistBoth)

    grs.addGesture(gTwirlLeft, fTwirlLeft)
    grs.addGesture(gTwirlLeft2, fTwirlLeft)
    grs.addGesture(gTwirlRight, fTwirlRight)
    grs.addGesture(gTwirlRight2, fTwirlRight)
```

Bibliography

[1] Interactive Imaging Systems.
http://www.iisvr.com/99iis/Products/vfx3d/VFX3D.htm. 17

[2] Gyro Point, Inc. http://www.gyropoint.com. 17

[3] Analog Devices, Inc.
http://content.analog.com/pressrelease/prprintdisplay/0,1628,47,00.html. 17

[4] D. Mackenzie. *Inventing Accuracy*. MIT Press, 1990. 18

[5] Intersense, Inc. http://www.isense.com. 18

[6] Royal Academy of Engineering. http://www.raeng.org.uk/awards/robert/base.html. 18

[7] R.E. Bicking. Automotive accerometers for vehicle ride and comfort. In Nwagboso [80], pages 125–140. 18

[8] S. Re Fiorentin. Sensors in automobile applications. In Nwagboso [80], pages 27–44. 18

[9] J. Bachiochi. Just one more mile. *Circuit Cellar INK*, (75):54–57, 1996. 19

[10] J. Elwell. Inertial navigation for the urban warrior. In *Digitization of the Battlespace IV, SPIE Proceedings Vol. 3709*, pages 196–204, 1999. 19

[11] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):780–785, 1997. 19

[12] L. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–86, 1989. 19, 67

[13] T. Starner and A. Pentland. Real-time American Sign Language recognition from video using Hidden Markov Models. In *Proceedings of the International Symposium of Computer Vision*, 1995. 19

[14] F. Sparacino, C. Wren, G. Davenport, and A. Pentland. Augmented performance in dance and theater. In *International Dance and Technology 99 (IDAT99)*, pages 80–89. FullHouse Publishing, 1999. 19

[15] A. Wilson. *Adaptive Models for the Recognition of Human Gesture*. PhD thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, September 2000. 19

[16] F. Jensen. *An Introduction to Bayesian Networks*. Springer, 1996. 19

[17] Analogous Corporation. http://www.analogus.com/index1.html. 20

[18] J. A. Paradiso, K. Hsiao, A. Y. Benbasat, and Z. Teegarden. Design and implementation of expressive footwear. *IBM Systems Journal*, 39(3&4):511–529, 2000. 20

[19] B. Blumberg et al. `(void*)`: A cast of characters. In *Conference Abstracts and Applications, SIGGRAPH '99* [79], pages 169–170. 20

[20] Ascension Technology Corp. http://www.ascension-tech.com/products/miniBird/minibird.htm. 21

[21] Intersense, inc. http://www.isense.com/products/prec/is900/index.htm. 21

[22] Crossbow Technology, Inc. http://www.xbow.com/html/gyros/dmu6x.htm. 21

[23] H. Sawada and S. Hashimoto. Gesture recognition using an accelerometer sensor and its application to musical performance control. *Electronics and Communications in Japan, Part 3*, 80(5):9–17, 1997. 21

[24] T. Marrin and J. Paradiso. The digital baton: a versatile performance instrument. In *Proceedings of the International Computer Music Conference*, pages 313–316. Computer Music Association, 1997. 21

[25] J. A. Paradiso. The Brain Opera technology: New instruments and gestural sensors for musical interaction and performance. *Journal of New Music Research*, 28(2):130–149, 1999. 21

[26] Teresa Marrin Nakra. *Inside the Conductor's Jacket: Analysis, Interpretation and Musical Synthesis of Expressive Gesture*. PhD thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology, February 2000. 21

[27] J.F. Bartlett. Rock 'n' scroll is here to stay. *IEEE Computer Graphics and Applications*, 20(3):40–45, May/June 2000. 21

[28] D. Small and H. Ishii. Design of spatially aware graspable displays. In *Proceedings of CHI '97*, pages 367–368. ACM Press, 1997. 21

[29] G. W. Fitzmaurice. Situated information spaces and spatially aware palmtop computers. *Communications of the ACM*, 36(7):38–49, July 1993. 21

[30] Seymour Papert. Things that Think Meeting, October 1995. 23

[31] A. A. Santiago. Extended Kalman filtering applied to a full accelerometer strapdown inertial measurement unit. Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 1992. 27

[32] B.P. Lathi. *Modern Digital and Analog Communications Systems*. Oxford University Press, 1995. 27

[33] L. Baxter. *Capacitive Sensors: Design and Applications*. IEEE Press, 1997. 29

[34] A. Lawrence. *Modern Inertial Technology*. Springer, 1998. 31

[35] Fizoptika Co. http://www.fizoptika.ru/. 31

[36] Analog Devices, Inc. http://www.analog.com/industry/iMEMS/products/ADXL202.html. 30

[37] J. Geen. Minimizing micromachined gyros, January 2000. MIT MTL VLSI Seminar Series. 30

[38] J. Kang, Samsung Electronics. Personal Communication. 30

[39] J. Haartsen. The bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, February 2000. 35

[40] Cambridge Silicon Radio. http://www.cambridgesiliconradio.com/bluecore.htm. 35

[41] Tadiran U.S. Battery Division. http://www.tadiranbat.com/specs6/tl_5902.htm. 35

[42] E. Foxlin. Inertial head tracking. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1993. 38

[43] J. Anthes. Unique considerations for data radio UARTS. http://www.rfm.com/corp/appdata/AN43.pdf. 40

[44] Synthetic Character Group, MIT Media Lab. http://characters.www.media.mit.edu/groups/characters/. 43

[45] MathWorks, Inc. http://www.mathworks.com/products/matlab/. 47

[46] K. Novak, L. Miller, and J. Houk. Kinematic properties of rapid hand movements in a knob turning task. *Experimental Brain Research*, 132:419–433, 2000. 48

[47] J. Hollerbach and T. Flash. Dynamic interactions between limb segments during planar arm movement. *Biological Cybernetics*, 44:67–77, 1982. 48, 66

[48] S. Emura and S. Tachi. Multisensor integrated prediction for virtual reality. *Prescence*, 7(4):410–422, 1998. 50

[49] N. Barbour and G. Schmidt. Inertial technology trends. In *Proceedings of the 1998 Workshop on Autonomous Underwater Vehicles*, pages 55–63. IEEE Press, 1998. 50

[50] D. Catlin. *Estimation, Control and the Discrete Kalman Filter*. Springer Verlag, 1989. 50

[51] A. Gelb, editor. *Applied Optimal Estimation*. MIT Press, 1974. 50

[52] R. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. Wiley, 3rd edition, 1997. 50

[53] E. Foxlin. Inertial head-tracker sensor fusion by a complementary separate-bias Kalman filter. In *Proceedings of the IEEE VRAIS 96*, pages 185–195. IEEE Computer Society, 1996. 51

[54] G. Welch and G. Bishop. Single-Constraint-at-a-Time tracking. In *SIGGRAPH '97 Conference Proceedings*, pages 333–344. ACM Press, 1997. 51

[55] D. Titterton and J. Weston. *Strapdown Inertial Navigation Technology*. IEE, 1997. 52

[56] K. Britting. *Inertial Navigation Systems Analysis*. Wiley, 1971. 52

[57] C. Verplaetse. Inertial proprioceptive devices: Self-motion-sensing toys and tools. *IBM Systems Journal*, 35(3&4), 1996. 54

[58] A. Oppenheim. *Discrete-time Signal Processing*. Prentice-Hall, 2nd edition, 1999. 54

[59] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992. 54

[60] R. Hogg and J. Ledolter. *Applied Statistics for Engineers and Physical Scientists*. Macmillan, 1992. 59

[61] T. Flash and N. Hogan. The coordination of arm movements: An experimentally confirmed mathematical model. *Journal of Neuroscience*, 5:1688–1703, 1985. 60

[62] C. Atkeson and J. Hollerbach. Kinematic features of unrestrained vertical arm movements. *Journal of Neuroscience*, 5(9):2318–2330, 1985. 66

[63] R. Plamondon. A kinematic theory of rapid human movements. part i: Movement representation and generation. *Biological Cybernetics*, 72:295–307, 1995. 67

[64] L. Rabiner and B. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993. 67

[65] A. Wilson, MIT Media Lab. Personal Communication. 67

[66] A. Wilson and A. Bobick. Parametric Hidden Markov Models for gesture recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9):884–900, 1999. 67

[67] R. Marteniuk, C. MacKenzie, M. Jeannerod, S. Athenes, and C. Dugas. Constraints on human arm movement trajectories. *Canadian Journal of Psychology*, 41(3):365–378, 1987. 70

[68] R. Plamondon. A kinematic theory of rapid human movements. part ii: Movement time and control. *Biological Cybernetics*, 72:309–320, 1995. 71, 74

[69] M. Lutz. *Programming Python*. O'Reilly, 1996. 81

[70] M. Masliah and P. Milgram. Measuring the allocation of control in a 6 degree-of-freedom docking experiment. In *CHI 2000 Conference Proceedings*, pages 25–32. ACM Press, 2000. 86

[71] C. Kline and B. Blumberg. The art and science of synthetic character design. In *Proceedings of the AISB1999 Symposium on AI and Creativity in Entertainment and Visual Art, Edinburgh, Scotland*, 1999. 87

[72] M.P. Johnson, A. Wilson, B. Blumberg, C. Kline, and A. Bobick. Sympathetic interfaces: using a plush toy to direct synthetic characters. In *CHI 1999 Conference Proceedings*, pages 152–158. ACM Press, 1999. 87

[73] C. Chaplin, director. The Gold Rush, 1925. 100 mins. 87

[74] M. Downie, MIT Media Lab. Personal Communication. 88

[75] M. Berlin, MIT Media Lab. Personal Communication. 89

[76] M.P. Johnson. Multi-dimensional quaternion interpolation. In *Conference Abstracts and Applications, SIGGRAPH '99* [79], page 258. 90

[77] J. Anderson. Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, 94(2):192–210, 1987. 97

[78] W. Balzer, M. Doherty, and R. O'Connor, Jr. Effects of cognitive feedback on performance. *Psychological Bulletin*, 106(3):410–433, 1989. 97

[79] *Conference Abstracts and Applications, SIGGRAPH '99*. ACM Press, 1999. 132, 135

[80] C. Nwagboso, editor. *Automotive Sensory Systems*. Chapman and Hall, 1993. 131