Gesture analysis through a computer's audio interface: The Audio-Input Drum

by

Ben Nevile
B.Sc.(Hons), University of Manitoba, 1996

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF APPLIED SCIENCE (INTD)

in the Department of Interdisciplinary Studies

Examiners:

Dr. Peter Driessen, Co-Supervisor (Department of Electrical and Computer Engineering)

Dr. W. Andrew Schloss, Co-Supervisor (School of Music)

Dr. George Tzanetakis, Outside Member (Department of Computer Science)

Dr. Mantis Cheng, External Examiner (Department of Computer Science)

ABSTRACT

When people first started to use digital technology to generate music, they were thrilled with their new ability to create novel and different sounds; accordingly, much research effort has been directed towards rich and complex methods of sound synthesis. Unfortunately the deep physical connection that exists between a musician and an instrument has not received as much attention, and so although we have machines capable of synthesizing fantastic new sounds, we don't have the ability to use these sounds with any immediate finesse, of developing virtuosity with our new instruments. The work presented in this thesis is an exciting step towards a more dynamic future for computer-based musical performance. The Radio Drum was designed in 1987 at AT&T labs as a prototype of a three-dimensional mouse. Max Mathews later repurposed the apparatus as a musical instrument known as the Radio Baton, which in its most modern form outputs data using the MIDI protocol. The result of this work, a new system called the **Audio-Input Drum**, provides high-resolution gesture capture, a simplified apparatus, and access to the extremely flexible Max/MSP/Jitter real-time software signal processing environment.

Table of Contents

List of Figures

List of Symbols

Acknowledgement

Thanks to everyone for their patience.

Chapter 1

Introduction - Motivation

> ”The stone age was marked by man’s clever use of crude tools; the information age, to date, has been marked by man’s crude use of clever tools.” - Unknown

Computers have created a revolution in the world of music. When people first started to use digital technology to generate sound, they were enamoured with their new ability to create dramatically novel and different tones. No longer were they tied to the acoustic properties of real objects in the physical world: digital signal processing provided them with the tools to explore new sonic possibilities.

This was liberating for composers, but performers were for the most part left out of the loop: the magical interaction between a musician and an instrument has been largely set aside in computer music. Modern computing engines can be made to behave as predictably as physical systems, so it is therefore theoretically possible to develop a level of virtuosity similar to that which a performer might achieve through hours of practice with an acoustic instrument. The difficulty is one of interface; while a lot of attention has been devoted to making computers capable of capturing, generating, and outputting high-resolution sound data, not nearly as much energy has been devoted to the problem of creating a high-resolution input for a performer’s gestures. It is the position of the author that to fully appreciate the performance potential of a computer equipped with a modern audio interface, the nature of the real-time input systems available to control musical processes must be at least as resolved as the musical output. As David Zicarelli wrote in his Computer Music Journal article, ”Communicating with Meaningless Numbers,”

> There has ... been little discussion of the computational architecture and resources necessary to support a situation in which the audio output might actually represent a reduction in the amount of data transmitted when compared with the gestural input.

The work presented in this thesis aims towards this ”resolution gap” between input and output technology.

The idea of gesture sensing using capacitive moments was exploited at Bell Labs by R. Boie to create a prototype for a three-dimensional mouse [1]. The apparatus has gone through many generations of development and many different forms, mostly under the direction of Max Mathews. [2] Some versions of this apparatus are known as the Radio Drum, and others as the Radio Baton. Chapter 2 includes a brief outline of work related to digital musical interfaces, and describes the Radio Drum in more detail.

This work was motivated by the desire to improve several characteristics of the Radio Drum and Radio Baton’s performance. The mechanism by which data is transmitted to the computer, the MIDI protocol, is a serial communication scheme with a maximum data rate of 31.25kbits/sec [3, 4]. The amount of information that can be sent over a MIDI connection is therefore small compared to the amount of data produced by a standard 16-bit, 44.1 kHz monoaural audio output channel. This limitation inherently limits the **resolution** of the interface, both in terms of the **sensitivity** and **timing** of gestures. A lack of sensitivity means that a performer using the Radio Drum has a limited range of dynamic expression. Timing problems manifest themselves through a **latency** between the action of hitting the surface of the drum and the resulting synthesis of sound and **quantization** in the possible locations of events in time. In addition, a variable delay we’ll call **synthesis jitter** is a function of the serial queue in which MIDI messages are processed by the computer. Our new Audio-Input Drum system is more resolved than the original Radio Drum, and the mechanism by which modern operating systems process audio is more immediate and reliable than the mechanism by which serial or MIDI messages are processed, resulting in less synthesis jitter. Because of the amount of computational power necessary to carry out the

signal processing in software, this technique has really only become practical within the last few years. In addition to these technical improvements, the apparatus of the Audio-Input Drum is more compact, portable, and less expensive than the Radio Drum. Chapter 3 discusses the design of the Audio-Input Drum.

Another major drawback to the Radio Drum system is that the algorithms that process the signals received by the antennae are implemented in hardware, which makes them very difficult to modify. Moving this signal processing into the realm of computer software simplifies the process of implementing new algorithms by an order of magnitude. Using the flexible, rapid prototyping environment of Max/MSP/Jitter [5] to implement algorithms for hit detection helped us to fix problems with velocity reporting that plagued the Radio Drum. Performers using the Radio Drum frequently experienced **duffing**, a phenomenon where a forceful drum hit would result in a small velocity being reported. By being able to visualize and analyze the signals in software, a hypothesis was formed for the set of circumstances that resulted in duffing and a more robust hit-detection algorithm was designed and implemented. Chapter 4 discusses the design of the gesture analysis algorithms used in the Audio-Input Drum system.

The same flexibility that helps us to improve what the Radio Drum and Radio Baton could already do also helps us to implement new modes of operation. For example, it was easy to modify the software to simultaneously use both data created through hit detection and continuous positional data. It was impossible to use these at the same time with previous generations of the interface. Chapter 5 discusses different potential modes of operation for the Audio-Input Drum, as well as potential improvements to the interface and system itself in terms of resolution and performance. The future potential of the Audio-Input gesture system is the most exciting part of this project.

Chapter 2

Related and Previous Work

In this chapter we briefly discuss some contributions to computer music interface design. We then introduce the Radio Drum and review its various stages of development, as well as highlight some artistic output that would not have been possible without this powerful interface. But first, no summary of the history of this interface could be complete without some discussion of the life of Léon Theremin, the grandfather of capacitive sensing.

2.1 Leon Theremin and Capacitive Sensing

Léon Theremin was a Russian inventor from Saint Petersburg most famous for his invention of the **Theremin** in 1919, one of the first electronic musical instruments [6]. He toured Europe, performed with the New York Philharmonic, and patented his invention in 1929. In a laboratory he set up in New York in the 1930s he experimented with other electronic instruments, including the Rhythmicon [7]. In 1938 Theremin was seized by Soviet KGB agents and forced to return to the Soviet Union. After a stint of forced hard labour, Theremin returned to inventing an impressive array of devices, including the first covert listening device to use passive electromagnetic induction to transmit an audio signal.

The theremin was unusual in that it was the first musical instrument designed to be played without being touched. The instrument was originally the product of Russian government-sponsored research into proximity sensors. Because of the timing of the commercial release immediately prior to the Great Depression, the instrument was not a com-

mercial success. However, dedication to the theremin persisted among enthuiasts such as Robert Moog [8].

In performance a musician stands in front of the theremin and moves his hands in the proximity of two metal antennas. The distance from one antenna determines the frequency of the generated waveform, and the distance from the other controls its amplitude. Typically the right hand controls the pitch and the left controls the volume, although some performers reverse this arrangement. The theremin uses the mixing of two higher frequency signals to generate a signal in the audio bandwidth. The instrument's circuitry includes two radio frequency oscillators, one which operates at a fixed frequency and one whose frequency is modulated by the performer's hand, which acts as the ground plate of a variable capacitor in a tuned circuit. The difference between these two frequencies generates a beat frequency in the audio bandwidth. The amplitude of the output frequency is controlled in a similar manner, with the distance between the performer's hand and the volume control antenna determining a variable capacitor's impedance, which regulates the theremin's volume.

2.2 Relevant electronic drum research

Much of the crucial research in electronic percussion has been conducted by Max Mathews. [2] The **Sequential Drum** was a rectangular drum surface that sent four electrical signals to a computer-synthesizer [9]. The four signals represented a trigger pulse that occurred when the drum was hit, an amplitude signal proportional to how hard the drum was hit, and x and y signals to encode the position of the hit. The drum got its name from the synthesis engine's method of advancing to the next element of a score every time the drum is hit. Ostensibly this allowed the "musician to focus specifically on aspects of interpretation rather than on a technically correct realization, which is handled automatically." [9]

Later Mathews introduced the **Daton**, a rigid plate about 14 inches square and an inch thick supported at its four corners by strain gauges [10]. These gauges generated electric pulses when the plate was struck, and circuitry analyzed the amplitude of these pulses to

compute both where and how hard the surface was hit.

Both the Sequential Drum and the Daton only sent information to the computer in response to a drum hit. The Daton was introduced in the context of a conductor program that simulates an orchestra following the baton of a conductor. To control continuous factors such as the variation of amplitude or vibrato, a large joystick had to be added to the interface. When the conductor program was adapted to use the **Radio Baton**, a version of the instrument that is the subject of this thesis, the continuous control could be provided by the three-dimensional tracking of the sticks, so the joystick was no longer needed. [11]

Many companies manufacture electronic drum kits based on piezoelectric technology [12, 13]; piezo components are placed in contact with physical drum surfaces and produce electrical signals in response to the vibration of the drum heads. In 1994 Korg introduced the Wavedrum to the music market [14]. It is special in that it includes a synthesis engine that uses physical modeling algorithms to produce sound based on the input of three contact microphones that sit underneath the drumhead. The drum can output MIDI, but the synthesis algorithms cannot be triggered by MIDI because they require more information than the MIDI protocol can diirectly support.

In 1999, Emmanuel Fléty built a custom ultrasound-based solution for the tracking of a percussionist's gestures for Ronald Auzet [15, 16]. Motions within a cube of 70cm per side could be tracked with a 5.5mm resolution at a sampling rate somewhere between 50 and 100 Hz, and data was communicated to synthesis software via the MIDI protocol.

The Marimba Lumina, designed by Don Buchla, is a marimba-like controller that uses short range capacitive sensing to detect when and where different zones on the controller's surface have been hit [17]. It has an internal synthesizer, but also can use MIDI to communicate the event information to external synthesizers.

Figure 2.1. *Max Mathews playing the Radio Baton in 1992. Photo by Patte Wood.*

2.3 The Radio Drum and Radio Baton

The idea of gesture sensing using capacitance was exploited at Bell Labs by R. Boie in 1987 to create a prototype for a three-dimensional mouse [1]. The apparatus was later repurposed as a musical instrument by Max Mathews [18]. The apparatus has been known both as the **Radio Drum** and the **Radio Baton**, depending on the generation of the design and who you are talking to [11, 19, 20]. Drum sticks with wire coiled tightly around the tips are driven at ”radio frequencies” - in the original apparatus the two carrier frequencies were both in the range between 55 and 60 kHz. This driving potential induces a sympathetic response in antennae contained in the flat surface of the drum.

The geometry of the antennae imprinted on the drum plane changed from generation to generation of the interface. Figure 2.2 shows one version of the Radio Baton that featured five different regions, four of which corresponded to edges of the drum surface and one large square surface in the middle. Figure 2.8 shows the newest version of the Radio Baton surface. Figure 2.9 illustrates the geometry of the four separate antennae on this newest generation of the Baton.

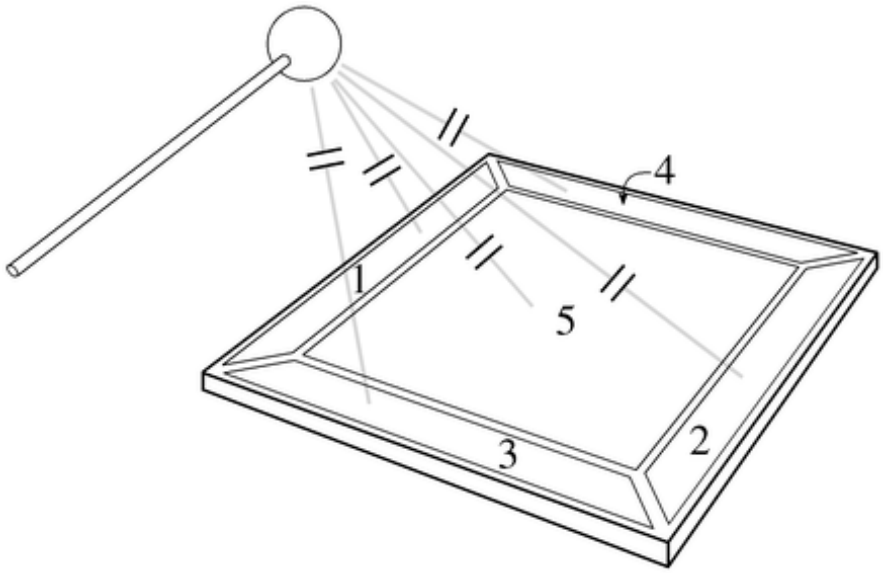

Figure 2.2. *A diagram of an older version of the Radio Baton apparatus with five distinct antenna surfaces.*

For all geometries the main principle of the interface is consistent: the magnitude of the signal induced is different for each region depending on the position of the transmitting drum stick above the surface. These relative magnitudes are input into a formula based on the geometry of the antenna, and can in this way be used to estimate the position of the stick.

A data flow diagram of the apparatus for the previous generation of the Radio Drum is presented in Figure 2.5. This version of apparatus processed the eight signals (four for each stick) with a special hardware component that we called the **Black Box**. This was in reference to the color of the hardware's casing, but also was appropriate because it was more or less a hardware device whose innards were not easily examined or altered. The Black Box produced the driving signals for the sticks, took the antenna signals as input, processed the information, and reported events out via a serial interface.

As illustrated in Figure 2.7, a simple two-plane thresholding algorithm was used by the Black Box to detect when the surface of the drum had been hit: when the estimated z-

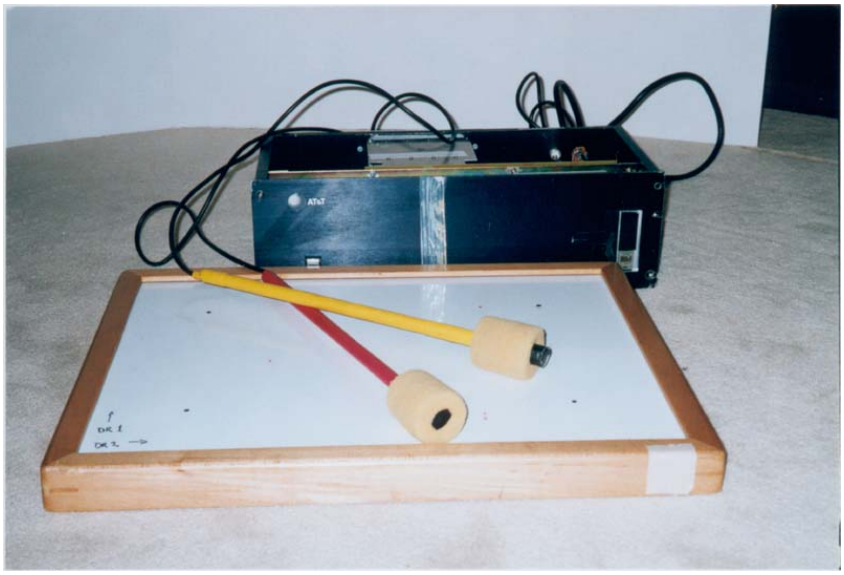

Figure 2.3. *The previous generation of Radio Drum apparatus.*

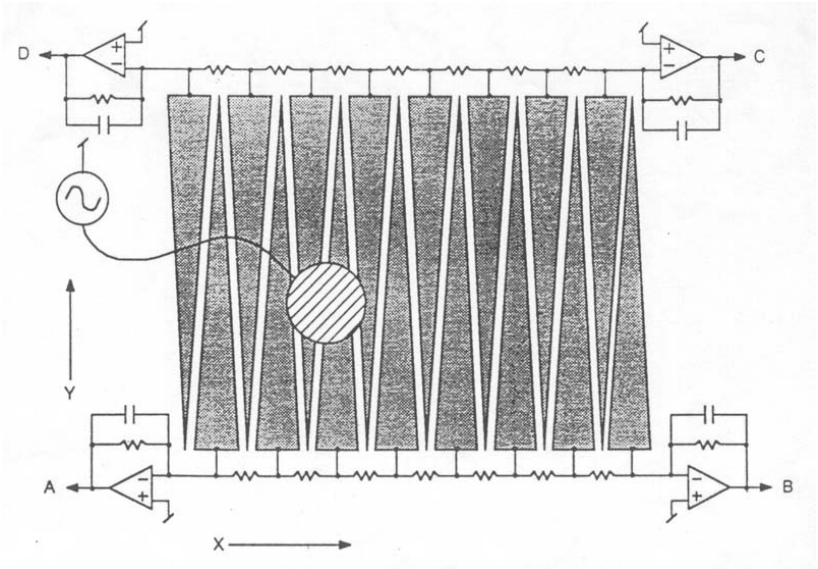

Figure 2.4. *Diagram of antenna surface of previous generation of Radio Drum apparatus (from [1]).*

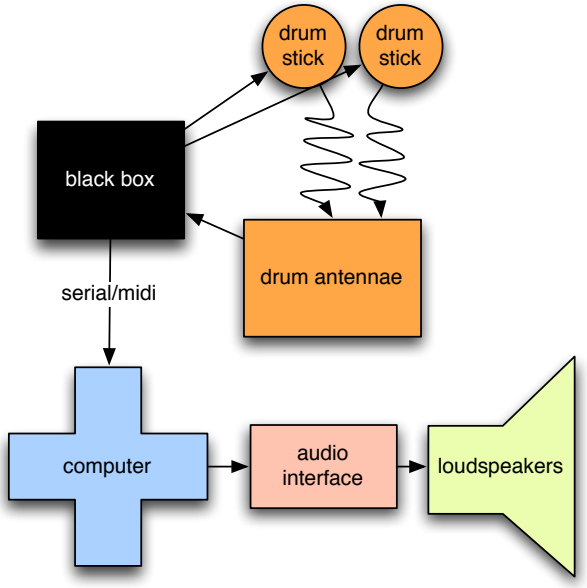

Figure 2.5. *A data flow diagram of the apparatus for the previous generation of the Radio Drum.*

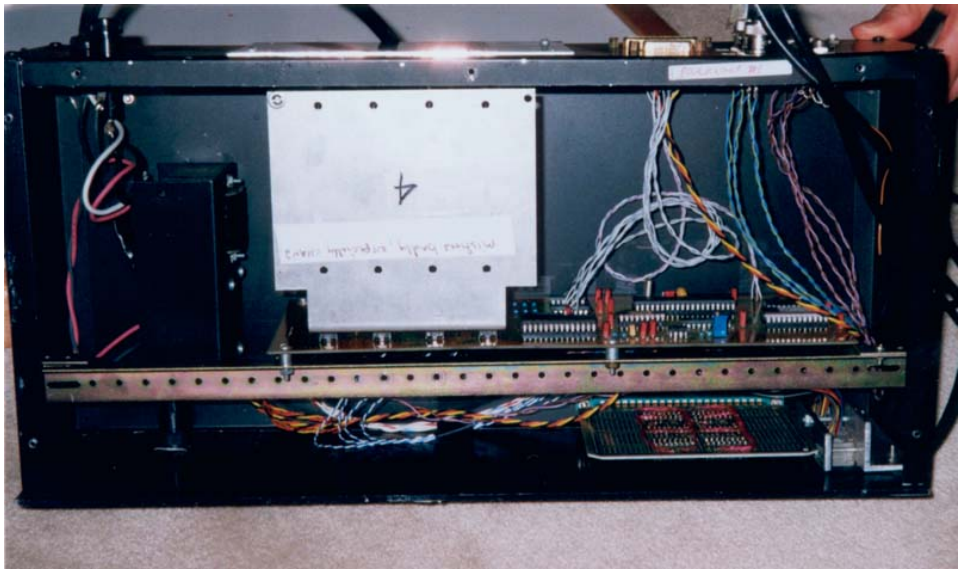

Figure 2.6. *The inside of the Black Box.*

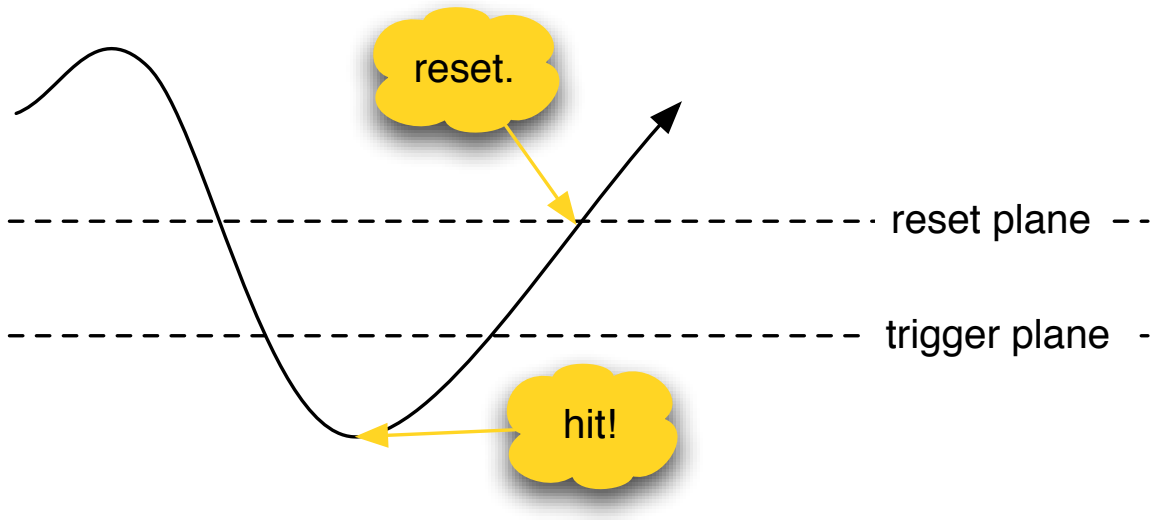

Figure 2.7. *Illustration of the trigger and reset planes used by the Black Box to detect hits.*

position of a drum stick fell below a **trigger plane**, the hardware would transmit a message that the pad had been hit once it had detected that a minimum of the z-position was reached. The Black Box would not output another hit until the drum stick rose above a **reset plane**; this small hysteresis served as protection against multiple triggers due to a noisy signal. The estimated force or **velocity** of the strike was estimated as a function of the depth of this minimum z-position below the surface threshold.

Data from the Black Box could be transmitted in one of two modes: in **continuous** mode the three-dimensional estimate of the position of each stick was sent to the computer at a rate of 20Hz. In **whack** mode data was transmitted to the computer only when the Black Box's algorithm detected that one of the sticks has whacked the surface.

As we'll see in Chapter 4, this previous generation of the Radio Drum had problems estimating the forcefulness of a drum hit. It was felt that access to the raw signals coming from the antennae could improve the sensitivity, resolution, latency characteristics, and reliability of the interface, and unlock the potential of this fascinating instrument.

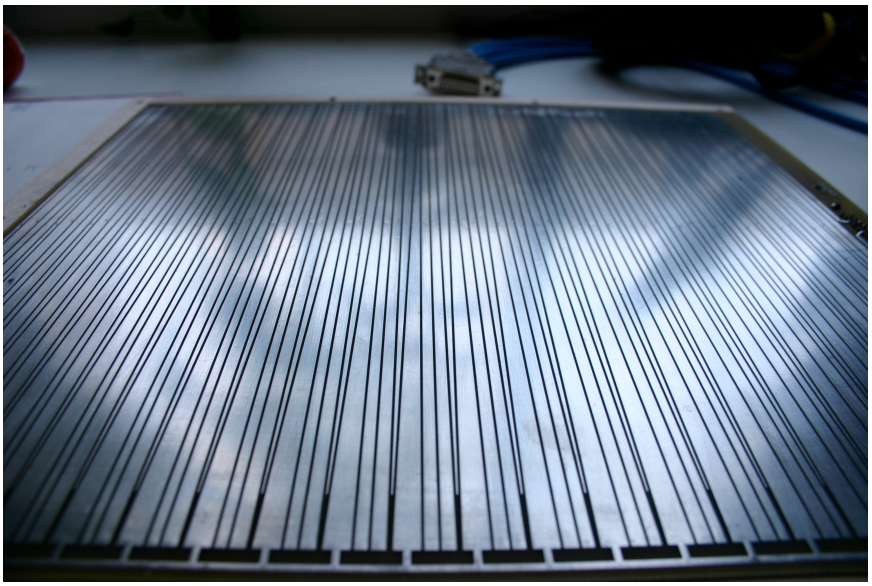

Figure 2.8. *Top of the antenna surface in the most recent version of the Radio Baton. There are four distinct antenna surfaces, two of which are triangular "backgammon" patterns and two of which are simple rectangular strips.*

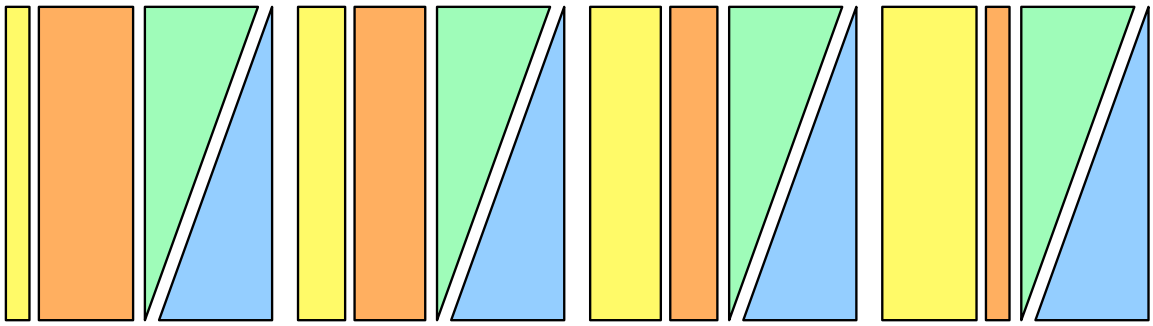

Figure 2.9. *Illustration of the geometry of the four antennae in the newest version of the Radio Baton. Horizontally the two rectangular areas vary in width inversely from one another.*

Figure 2.10. *W. Andrew Schloss performing with the Radio Drum and a set of foot pedals.*

2.4 The Success of the Radio Drum

The Radio Drum has enjoyed significant success as performance interfaces. Andrew Schloss and David Jaffe have worked together on several successful musical projects involving the Radio Drum which have been performed all over the world. "Wildlife", an interactive duo for Mathew/Boie Radio Drum and Zeta Violin, is available on CD from Centaur Records [21]. In a fascinating demonstration of the interface's potential to perform sound not usually associated with a drum, Schloss and Jaffe created "The Seven Wonders of the Ancient World", a concerto that coupled the Yamaha Disklavier Grand piano with the Radio Drum [22]. With its revving engines, "Racing Against Time", written by Jaffe and featuring Schloss as the soloist, continued work on non-standard sounds controlled by drum sticks. Through their experience with this decoupled instrument, Schloss and Jaffe have many interesting ideas related to the relationship between performer, pre-recorded music, and computer [20].

Schloss has produced work independently of Jaffe as well. In 2000, Schloss and Randall Jones produced "UNI", an interactive music and video performance that had its world

Figure 2.11. *Promotional photo for David A. Jaffe and W. Andrew Schloss's performance of Wildlife.*

premiere in Cuba [23]. In 2005 "Blind Data," an improvisational duet performed with the estimable Cuban pianist Hilario Durán, was recorded at the NIME conference in Vancouver [24].

Richard Boulanger has written and performed many pieces for the Radio Baton. **Radio Sonata** uses Max Mathews' conductor program and a Yamaha TG77 synthesizer [25]. **Grainstorm**, a piece produced a couple of years later, uses a similar set of equipment and also features a performer working with real-time video sequencing [26]. In 2003's **Stardust** the interface used by the video performer is dubbed the "Video Baton" [27]. Many more of his compositions are written about and available for download at Boulanger's website [28].

Other contemporary uses for the apparatus continue to be explored. The Radio Baton has been adapted for use as a DJ controller, using thimbles placed on the fingertips of the performer rather than sticks [29]. The author of this thesis has performed with the result of this thesis's work, the Audio-Input Drum, at numerous festivals in several different capacities: as a controller for video synthesis to accompany music; as a live sampling and playback interface in a duo with wind player Andrew Pask; and as a loop capture and playback interface with numerous DJs playing vinyl records.

Chapter 3

The Audio-Input Radio Drum

3.1 Introduction

Consider how humans have historically communicated musical performance data. Music notation involves the placement of **events** - notes and rests - in a sequence. Each event has a start-time, a duration, and if applicable, a pitch. The sequence is decorated with important but subjective meta-data such as tempo and dynamic changes.

We can feed such data into a computer connected to a synthesis module, and the computer can play the sequence back ”perfectly”. Indeed, a simple computer program will produce the exact same sequence of sounds every time it is given the same musical score. A human performer, however, will never give exactly the same performance twice. The gestures that make up a performance are broadly controlled by the musical score, but what makes each performance unique and alive is exactly what is **not** encoded in the music notation - the thousands of minute details on top of and in between the events written in the score.

In other words, the performer-instrument communication is better characterized by a continuous engagement of control rather than by a few discrete events. Much work has been done thinking about ways to classify, characterize, and parametrize musical gestures [30]. Perhaps the most basic way to define a gesture is to think of it as a form of communication that extends beyond what can be accomplished verbally. We spontaneously gesture to one another as we talk - with our hands, our eyes, our posture - and we do it naturally and

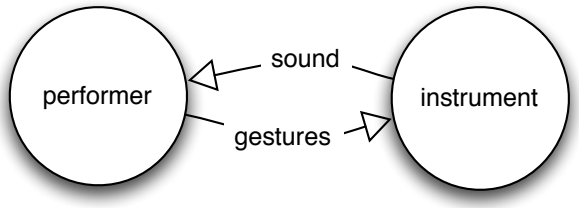

Figure 3.1. *The flow of information in developing virtuosity*

without conscious effort. The conversational gestural vocabulary may differ from culture to culture, but its importance is universal.

Gestures that a musician makes when interacting with an instrument can be thought of as a communication between performer and instrument: the performer interacts physically with an interface, and the instrument responds with some output directly triggered by the performer. Unlike the gestures made in person-to-person communication, which are decoded by subjective interpretation, artistic gestures are precisely interpreted by the state and physics of the instrument. The reliable feedback loop that an artist is engaged in with an interface allows the development of extremely fine gestural control; we can call the development of such skill **virtuosity**. Figure 3.1 illustrates the flow of information necessary to reinforce the creation of virtuosic skill.

Like a physical system, a computing engine behaves predictably, so it is therefore theoretically possible to develop virtuosity with a computer-enhanced instrument. Much research has been devoted to what a computer can output: modern machines and algorithms are capable of synthesizing sound and imagery that is rich and complex. However, the nature of the real-time input that one can use to control these processes is much less resolved in space and time than the output produced. As discussed in Chapter 1, we want to aim for a situation where the audio output is actually a **reduced** amount of data compared to the input that drives its synthesis. The work presented in this thesis aims towards this ”resolution gap” between input and output technology. The Audio-Input Drum that’s presented in this chapter is an adaptation of an older instrument with an improved, high-data rate gesture

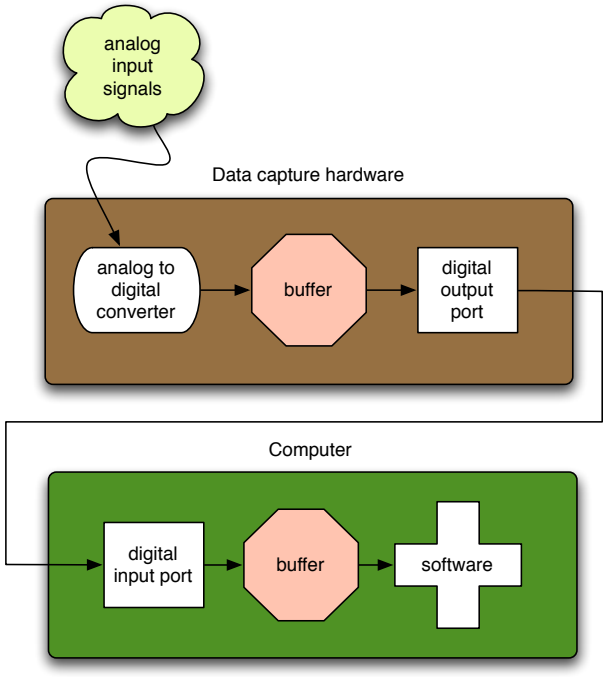

Figure 3.2. *The flow of signals that control software running in a computer*

acquisition system.

3.2 Data Input in Computers

At the start of the project, the central question was how best to input the gesture data into the computer. Modern computers have many possible ports of entry: ethernet, 802.11 wireless, IEEE 1394 firewire, USB, serial, and analog audio inputs come standard on many commercially available machines.

Figure 3.2 shows the network of data flow that is used in the distribution of input signals to software. Some sort of analog signal is translated into digital information using an analog-to-digital converter. The mechanism here will be different for every device: a computer keyboard is a bank of switches whose analog signal will either be open or closed, and older mice contained a ball that when rolled would in turn rotate two perpendicular cylinders whose motion was translated into electrical signals. Since we live in a physical

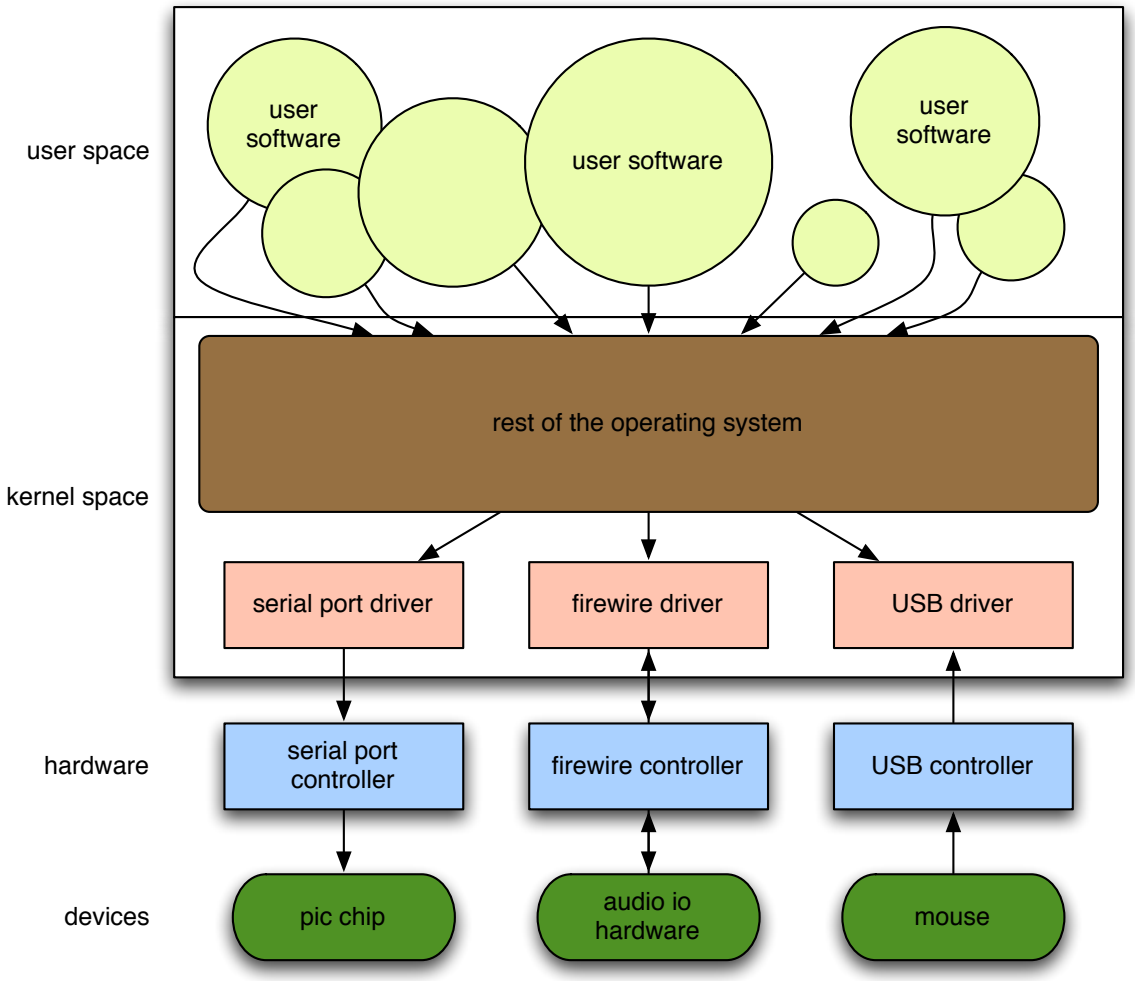

Figure 3.3. *Logical positioning of device drivers. Adapted from [31].*

world, our input will necessarily start out as an analog signal of some sort.

After the analog input signal has been translated into digital information, it is stored in a buffer. At some time in the future it will be sent out via a digital output port into the computer's digital input port. The subsystem of the computer that controls the input port will store the data into another buffer, and at some point in the future software will access that buffer to retrieve the information.

Figure 3.3 may help explain why the buffers in Figure 3.2 are necessary. Modern operating systems separate user processes from the kernel processes. Input devices are handled

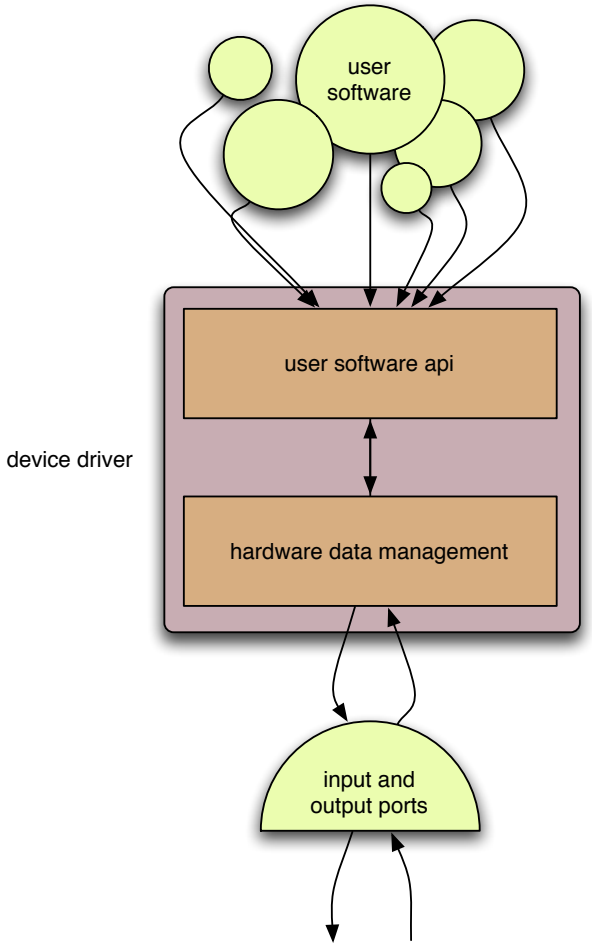

Figure 3.4. *A device driver has two components: the lower level that interacts with hardware, and the API that is exposed to user software.*

through **device drivers** that are loaded and executed by the operating system kernel. These drivers interact with the hardware controllers on their own schedule, independent of whatever user software may be running. User software has access to the device drivers only through calls made through the operating system kernel. User software may use a kernel-provided mechanism to poll a device driver periodically to retrieve any new data that has arrived. The kernel therefore has to keep incoming data in a buffer, ready to be given to the user process when it is requested.

We can think of device drivers as having two components, as illustrated in Figure 3.4.

The lower level interacts with the hardware: it understands the particular language that the device speaks, and manages data coming in from and out to the device in buffers. The upper level is the Application Programming Interface (API) that is presented to programmers of user software. This API may include methods to get and send data from and to the device, as well as methods to set and query the state of any attributes the device may have.

3.3 The audio subsystem

Figure 3.5 illustrates the flow of data when using the computer as part of an audio processing network. The analog-to-digital converter in the audio hardware converts the analog input signals to digital signals and places the results in a local buffer. The data is transported to the computer using a digital protocol (firewire in this case), and once again buffered. The user software accesses the data from this buffer, processes it, and places it in an output buffer. The data is transported back to the same audio interface, and buffered one more time before a digital-to-analog converter converts the data back to the analog domain.

Each of the buffering stages in Figure 3.5 adds to the total time it takes for an analog signal to pass through the entire processing chain from input to output. We call this input-to-output delay the **round-trip latency** of the audio-processing network. Because there are many musical applications for which reducing this latency is highly desirable, much engineering effort has been made by many independent companies developing audio hardware, device drivers, and operating system schemes that optimize this data flow.

Unfortunately, like most things in engineering, there is a tradeoff to consider; in this case the tradeoff is between efficiency and latency. To optimize the processing of audio inside the host software, audio drivers pass in blocks of audio data rather than single samples. A typical audio **vector** might be 64 samples long; some drivers allow the size of the audio vector to be set as low as 2 samples and as high as 4096 samples. The larger the block, the more computationally efficient the audio processing will be. To understand why, we need to examine the innards of an audio processing method.

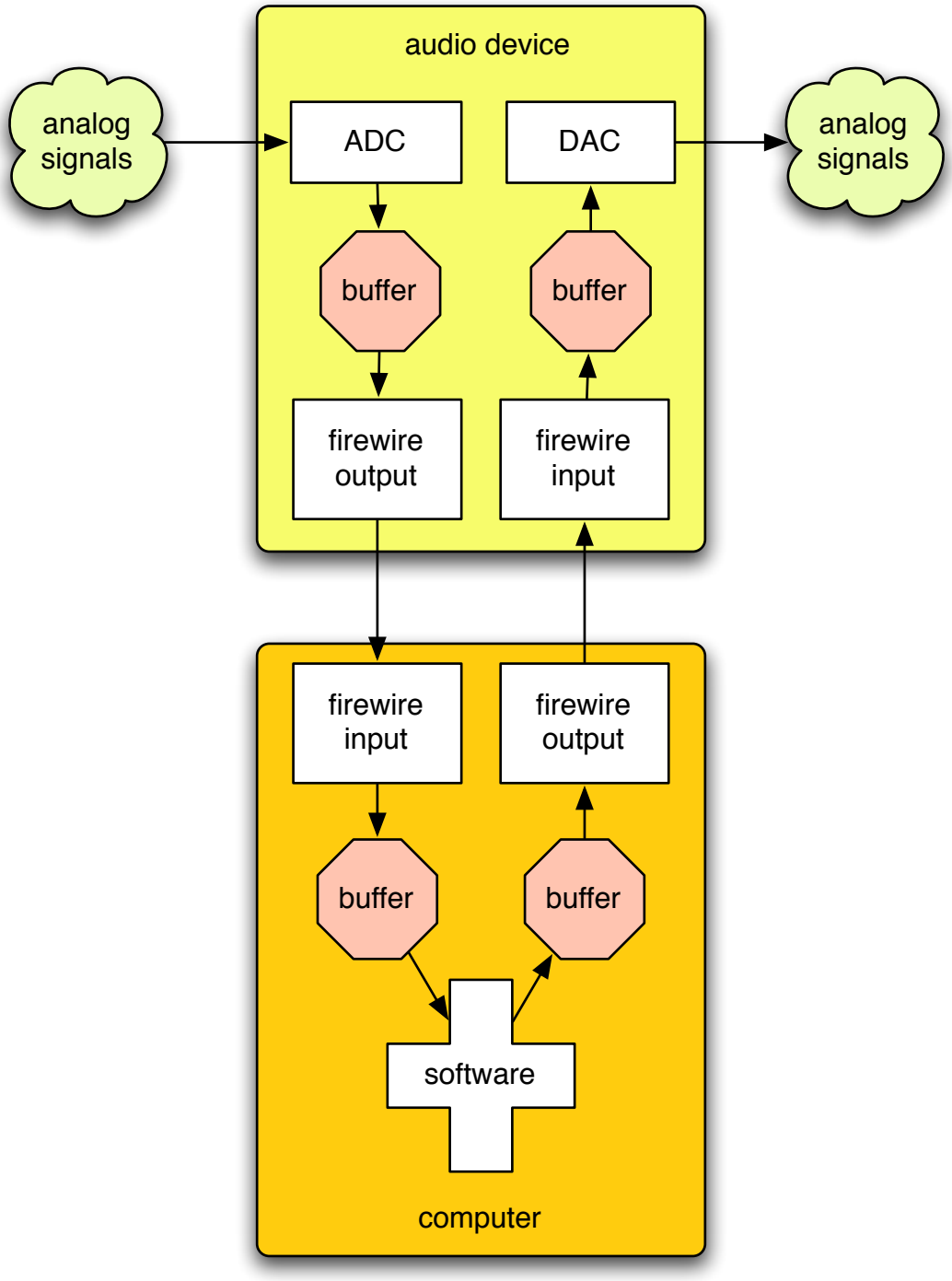

Figure 3.5. *The flow of audio processing data.*

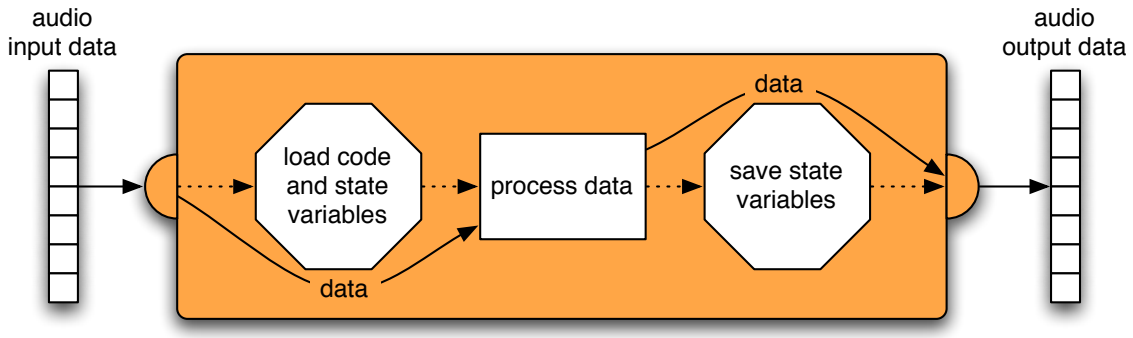

Figure 3.6. *Illustration of the computational flow of an audio perform method.*

Figure 3.6 illustrates a simplified execution of what is called a **perform** method in an audio DSP calculation. The process has three phases: first, any state variables associated with the processing of the audio must be loaded from memory. For example, a method performing biquadratic filtering on the audio would have to load the coefficients of the filter and the feedforward and feedback sample values. We also conceptually include the loading of all necessary machine instructions for the perform routine in this first phase, although in reality the code will be loaded from RAM dynamically on an as-needed basis. Once all state variables have been loaded, the actual processing of the data can take place: this tight inner loop will iterate over all n samples of the input audio vector and place the results into the output audio vector. Finally, the necessary state variables are saved. At this point the output vector can be sent to the audio hardware for conversion to analog output, or if the DSP calculating chain continues, it can be used as input for a subsequent perform method.

The time taken to execute the loading and saving portions of the perform method is independent of the audio vector size n - only the actual processing of the data varies with n. We can model the execution time per sample Δ_{sample} for a perform method with the following formula:

$$\Delta_{sample} = \frac{\Delta_{RAM} + \Delta_{process}(n)}{n} = \frac{\Delta_{RAM}}{n} + \frac{\Delta_{process}(n)}{n} \tag{3.1}$$

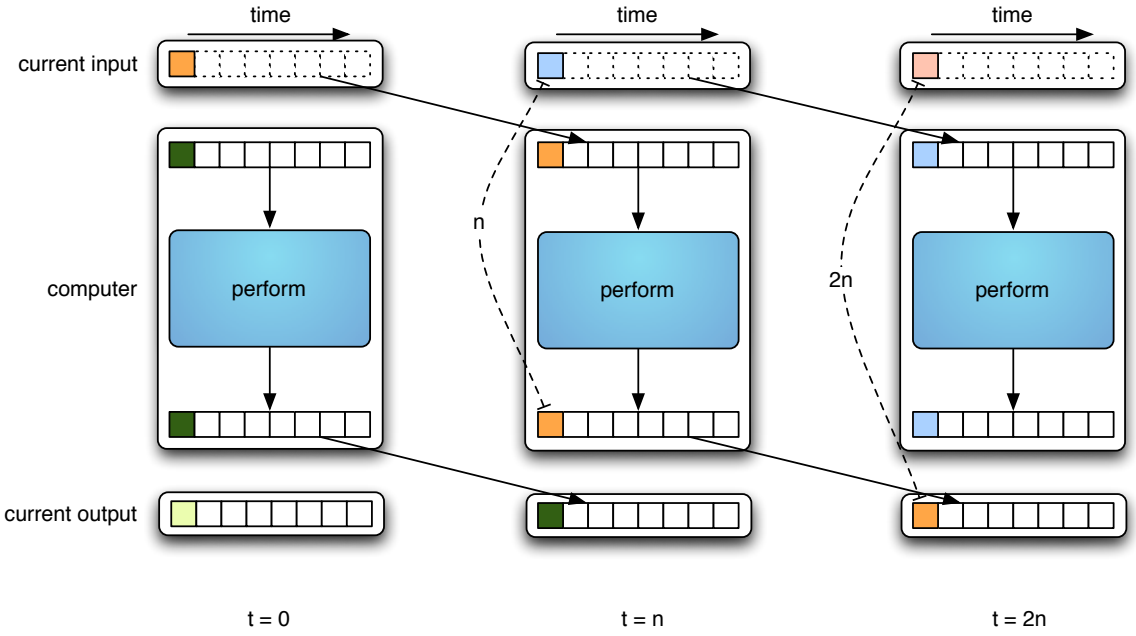

Figure 3.7. *An illustration of the $2n$ latency created by the input and output buffering of audio vectors.*

$\Delta_{process}(n)$ is a function of n that will be different for every signal processing routine. Most simple DSP operations are linear with n; some exceptions, such as FFT processing which varies as $n \log n$, do exist, but typically the size of the analysis vector for such operations is set independently of the audio vector size. Δ_{RAM} in the first term of equation 3.1, however, is constant, so a larger n mortgages its cost over a larger number of samples and decreases the execution time per sample Δ_{sample}.

The tradeoff to this more efficient calculation per sample is that a larger audio vector size n increases the latency of the system. Figure 3.7 illustrates this latency: when it is input into the audio perform method in the center figure, the orange sample is already n samples delayed from the current (blue) input sample because the audio software retrieves a chunk of n samples. After the orange sample is processed and output in the right figure, it is $2n$ samples behind the current (pink) input sample. Therefore the minimum latency of the audio processing chain is $2n$ sample periods.

Modern operating systems such as Mac OS X or Windows XP have special optimized

callback methods for audio processing. Because reliability and timing are of critical importance, audio processing callbacks are executed in a high-priority thread. This is the same importance we want to give our input gesture data; indeed, since our gesture data will be intimately linked with resulting sound, it is prudent to leverage the same optimized mechanism designed for importing sound data to import gesture data. Keeping the processing of the data and the resulting synthesis in the same computational thread can be of critical importance in terms of timing and reliability.

3.4 Signal Input Approaches

The environment of choice for computer musicians is the Max/MSP/Jitter software package, a visual programming language focussed on real-time work. Max/MSP's power lies in the considerable functionality afforded by its vast graphical object-oriented framework that facilitates the creation of art-generating algorithms. A computer-savvy, digital composer is therefore no longer limited to the tones and physical performer-instrument practicalities of traditional acoustic instruments, and can design virtually any sequence of sounds imaginable. A visual artist has a largely unexplored world at his fingertips. A broad overview and details of the various scheduling mechanisms that operate in Max/MSP/Jitter are discussed in Appendix A.

As we discussed in the previous section, for reasons of latency and reliability it is desirable to acquire and process the antenna signals using the audio subsystem of MSP. There is much commerically available hardware designed to import audio signals, but our problem is complicated by the fact that gesture signals have very different frequency content than audio signals. Figure 3.8 illustrates this difference: gesture signals are in the baseband and have a reasonably low upper bound; we can assume that they have an upper limit of 450 Hz, the effective upper bound on the frequency of muscular contraction[32]. Audio signals, on the other hand, do not have any content below a low frequency cutoff somewhere between 20 and 30 Hz. Most audio interfaces use a high-pass filter to help protect against DC offset

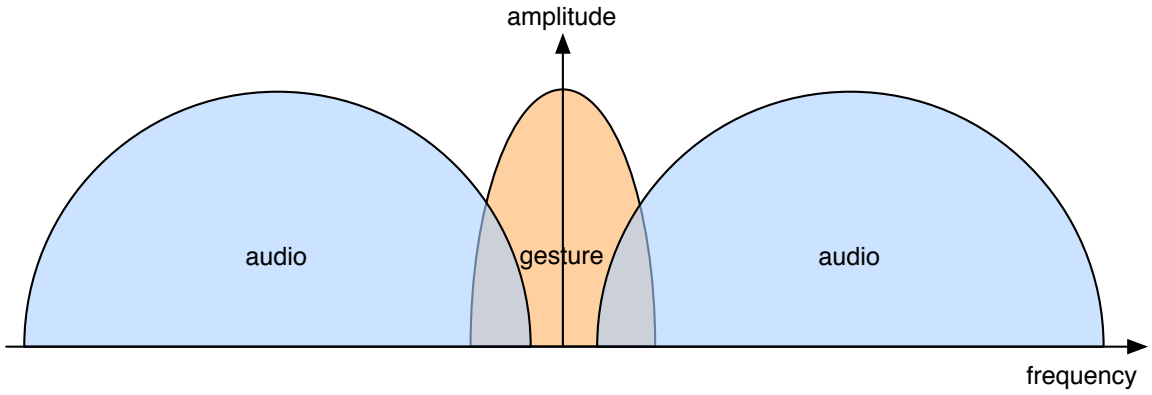

Figure 3.8. *Audio spectrum vs. gesture spectrum*

in the signal path; this is a problem for inputting the gesture data of a performer, who may sometimes be motionless (ie, generate a signal at 0Hz).

Several solutions to this problem were tested; a brief outline of each follows. These approaches are not presented in the chronological order in which they were tried, but rather in decreasing order of implementation difficulty.

3.4.1 The Rimas Box

The most direct way to solve the problem of existing audio hardware not having the proper specifications is to create your own audio hardware. A group of researchers at the Center for New Music and Audio Technologies at the University of California Berkeley built the Scalable Connectivity Processor (or **Rimas Box**) [33]. Along with AC-coupled audio inputs, the Rimas Box has DC-coupled gesture inputs incorporated into its design. These gesture inputs operate at one eighth the sample rate of the audio inputs.

Figure 3.9 illustrates the data flow of the radio drum when using the Rimas Box as the mechanism to import gesture data. The black box provides driving signals to the sticks. A signal is produced on the antennae via capacitive coupling, and fed back into the black box. The black box demodulates the signals. The demodulated gesture signals are intercepted from the multiplexing chip and input into the Rimas Box. The gesture data is received by

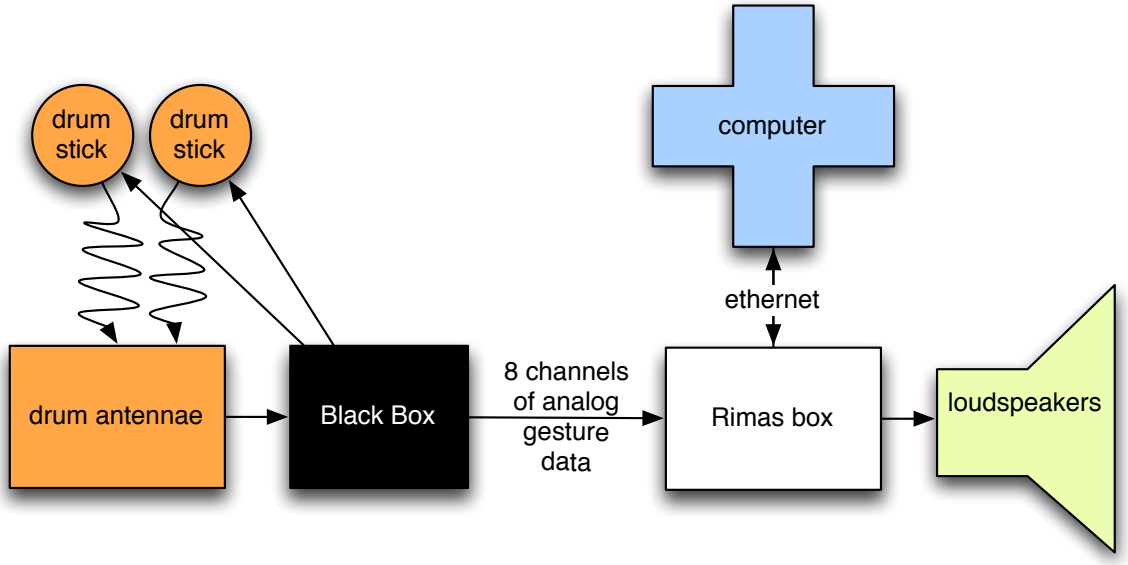

Figure 3.9. *Data flow diagram of the apparatus using the Rimas Box.*

the software over ethernet, which interprets it and synthesizes sound. The software sends the sound data to the audio interface, which translates from the digital to analog domains and sends signals to the loudspeakers

Although the gesture signals were transmitted successfully, some problems with the Rimas Box were identified. Most notably, the gesture inputs do not have proper anti-aliased filters incorporated into their design. This has important negative consequences in terms of the analysis of noise in the signal: without filtering out the frequencies above the Nyquist cutoff, high-frequency noise aliases and contaminates the baseband.

The Rimas Box communicates with the computer using an ethernet connection. A minimum round-trip latency of more than 10 ms was measured; this is significantly longer than is possible with modern commercially-available audio devices. Using the Rimas Box requires a special kernel extension to be installed for Mac OS X; Windows and other operating systems are not supported. Future updates to OS X could require updates to this kernel extension. Modern operating system theory discourages the use of kernel extensions since buggy software in the kernel can cause serious stability problems.

Figure 3.10. *The Teabox.*

Most importantly perhaps, the construction quality of the Rimas Box is lacking in terms of durability. This is to be expected, for it is of course an experimental audio device and not a commercial product ready for the market. However, it is worth mentioning because it highlights an important drawback to using custom-built hardware. A system designed around the Rimas Box will rely on fragile hardware and an up-to-date and bug-free kernel extension.

3.4.2 The Teabox

The Teabox, built by Electrotap, ”piggybacks” gesture data through a regular sound card[34]. This small hardware box features multiple analog gesture inputs and multiplexes these into a single digital stream of data in the S/PDIF format. The S/PDIF format is supported by most manufacturers of professional sound cards, either with a coaxial or an optical input. The multiplexed data from the Teabox is efficiently demultiplexed in software by a custom external object for the Max/MSP environment. The sampling rate of the gesture inputs is one tenth the CD 44.1kHz sampling rate.

Figure 3.11 illustrates the data flow of the apparatus using the Teabox. The Black Box provides driving signals to the sticks. A signal is produced on the antennae via capacitive

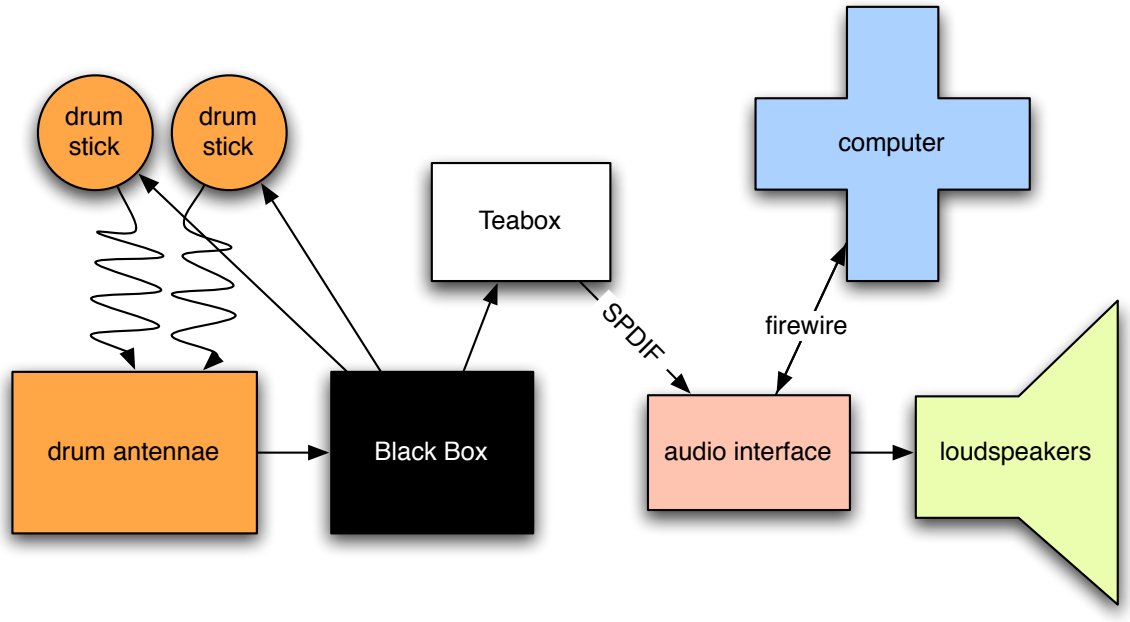

Figure 3.11. *Data flow diagram of the apparatus using the Teabox.*

coupling, and fed back into the black box. The black box demodulates the signals. The signals are intercepted from the multiplexing chip and input into the Teabox. The gesture data is multiplexed into a single channel and sent to a SPDIF digital output. The audio interface receives the SPDIF signal and sends the data to the computer. The gesture data is received by the software, which interprets it and synthesizes sound. The software sends the sound data to the audio interface, which translates from the digital to analog domains and sends signals to the loudspeakers.

Although we have heard reports of other musicians are using this device successfully, when attached to our apparatus we were never able to use the Teabox without intermittent digital noise. The cause of these brief discontinuities in the signal was not determined.

Because the Teabox is not widely available, the same issues with custom hardware discussed in section 3.4.1 exists, although in this case the device is available on the market and is relatively inexpensive (about $350 USD), so it would be feasible to keep a second unit handy in case of problems. The construction quality of the device is admirable.

3.4.3 The Chopper

The Chopper is a circuit that built at the University of Victoria to multiplex multiple gesture signals in the analog domain. Each of the eight input gesture signals are amplitude modulated, and the sum of the eight is passed in one channel of a analog audio input. In the other channel of the analog input the eight unmodulated carrier frequencies are passed. Once inside the computer individual bandpass filters separate the eight signals in each channel from one another. These filters were designed to precise specifications of width suitable to pass the gesture bandwidth and strong, 90dB rejection outside this width, using iterative methods [35]. After the filters have separated the signals, each modulated signal is demodulated using the standard synchronous technique of multiplying it with its corresponding unmodulated carrier wave and then lowpass filtering [36].

Figure 3.12 illustrates the data flow of the Radio Drum using the chopper multiplexing circuit. The Black Box provides driving signals to the sticks. A signal is produced on the antennae via capacitive coupling, and fed back into the Black Box. The Black Box demodulates the signals. The signals are intercepted from the multiplexing chip and input into the chopper. The chopper modulates the gesture signals at audio rates and multiplexes them into two channels, the modulated signals and the carriers, and outputs them via a stereo audio output. The audio interface receives the stereo signal and sends the data to the computer. The multiplexed data is received by the software, which demodulates it, interprets it and synthesizes sound. The software sends the sound data to the audio interface, which translates from the digital to analog domains and sends signals to the loudspeakers.

This novel technique provided for multiple channels of low-frequency gesture data to be passed through an inexpensive stereo sound input, the kind that comes standard with virtually every computer sold today. The separation filtering in the demodulation software adds several milliseconds of latency to the system. Multiple design iterations produced a small, stable circuit board, but a sturdy enclosure was never built. Being custom hardware, the same high-risk ”replaceability” issues that we talked about in section 3.4.1 exist here as well. Additionally, the computational cost of the demodulation is not insignificant.

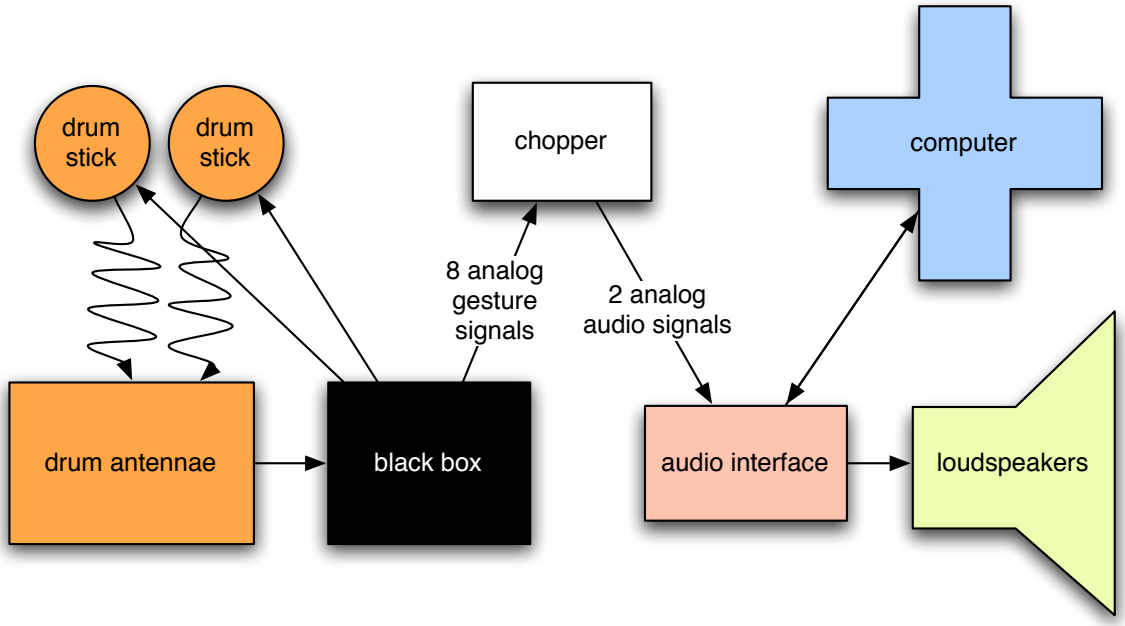

Figure 3.12. *Data flow diagram of the Radio Drum using the chopper multiplexing circuit.*

However, since computational power is a resource that is boundlessly growing, efficiency should not be todays primary consideration.

3.5 The Audio-Input Drum

One afternoon, the author was assembling the chopper-based apparatus. In a ”eureka!” moment, we realized that the Black Box was completely unnecessary - the computer could both generate the driving signals, and listen directly to the sympathetic response from the antennae. This idea simplified the apparatus tremendously.

Figure 3.13 shows the basic outline of a transmitting and receiving system where the audio interface provides a driving signal to a transmitting device. The potential caused by this signal capacitively induces movement of charge on the surface of the antenna. The antenna is connected to the input of the audio interface, so the computer ”listens” to the output of the antenna. The output of the antenna is an amplitude modulated sinusoidal wave

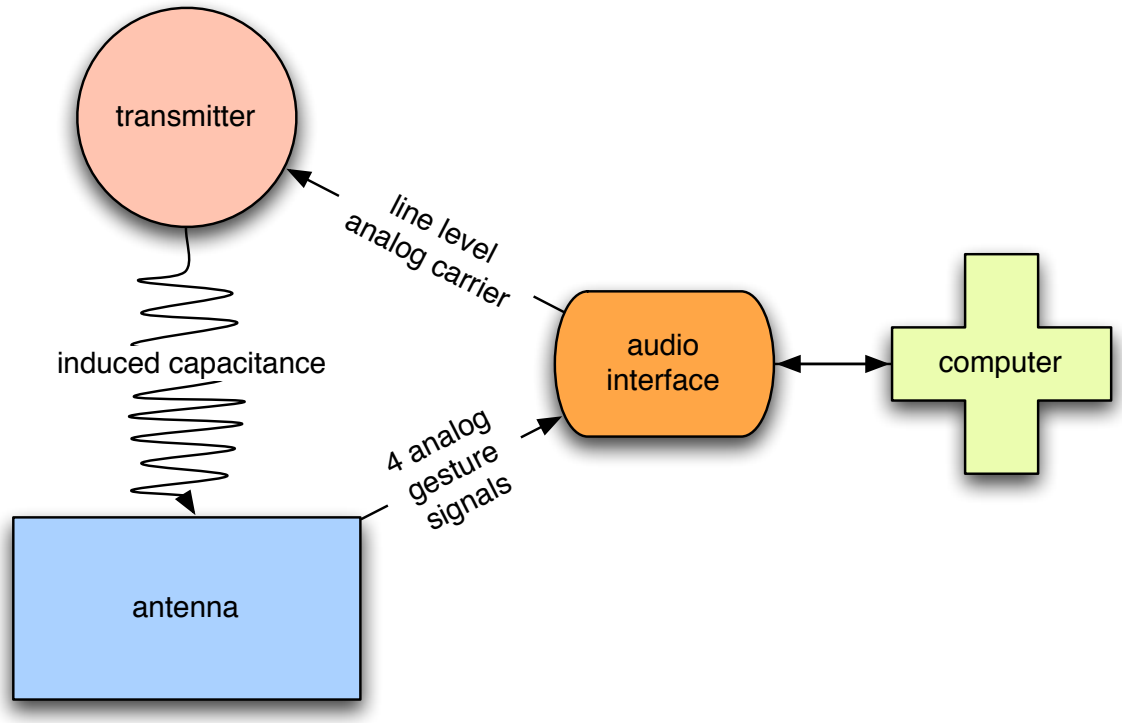

Figure 3.13. *The transmitter and antenna system.*

with frequency of the driving signal plus noise. To extract the gesture signal it's necessary to demodulate the amplitude information from the carrier, and then filter to reduce the noise.

In the case of the radio drum, each stick is driven with a different carrier frequency signal and the signals from the four antenna channels are input into four separate audio channels. The antenna signals are weak, so they are amplified before being digitized. Regular microphone preamplifiers that are built into some sound cards have proved suitable for this task - the Fireface 800 used by the author provides 60dB of gain on each of its four microphone preamplifiers [37].

The two driving carrier frequencies must be spaced far enough apart so that the sidebands created by the movement of the sticks do not spectrally overlap; Figure 3.14 illustrates this idea. The spectral width of each sideband is dependent on the physical nature of the interface. It is problematic to assume that the effective upper bound of the frequency of

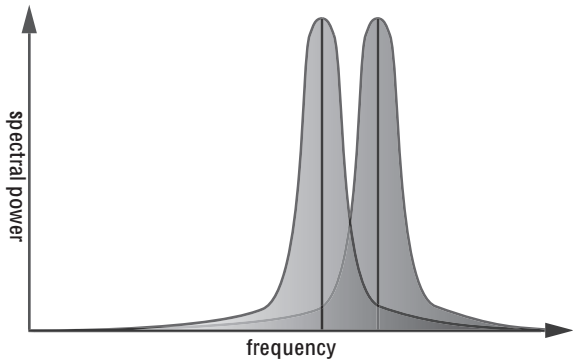

Figure 3.14. *Illustration of the input spectrum. Two carrier frequencies must be chosen far enough apart so that their bandwidths do not significantly overlap. Bandpass filters are used to isolate the carriers from one another.*

muscular contractions is the upper bound of a gesture signal's frequency content, since the physical nature of the interface may extend the frequency content above what the body's muscles can do on their own. For instance, a broadband spike would be the result if a perfectly rigid drum stick were to hit a perfectly rigid surface and instantaneously stop. The upper bound for the frequency content of the gesture signal must therefore be chosen after careful observation of the characteristics of the physical system; in the case of our interface, 450Hz has proven suitable as the upper bound.

We are of course limited to frequencies less than half of the audio output device's maximum sampling rate, which on a modern audio interface can be as high as 192kHz. Figure 3.15 shows a plot of the antenna's relative gain as a function of the driving frequency. The vertical axis of the plot is in dB units and and is normalized relative to the maximum receiving power around 60kHz.

Another factor in choosing carrier frequencies is that operating a signal-processing environment at high sampling rate is computationally demanding. Perhaps counter-intuitively, this is not due to the actual calculating of the DSP chain, but is primarily due to the transmission of data through the computer's IO subsystems [31]. The author has enjoyed efficient and reliable performance when operating the audio device with a sampling rate of

64kHz and generating carrier frequencies of 26 and 30kHz on modern hardware.

3.5.1 Demodulation

Since the carrier frequencies are generated internally in software, our implementation could make use of any synchronous or asynchronous demodulation scheme to extract the amplitude envelope from the modulated waveforms [36]. Figure 3.16 illustrates the computationally efficient method employed to move the amplitude envelope into the baseband. In this scheme a naive downsampling of the signal is performed at a rate that exactly matches an integer number of periods of the carrier signal in order to subsample on the carrier sine wave peaks. This downsampling folds the carrier frequency to 0 Hz, and the bandpass filters that separated the carriers act as lowpass filters around the baseband. When operating at a sample rate of f_r and downsampling by a factor of n, the carrier frequencies must be $k(f_r/n)$ where integer k satisfies $1 <= k <= (n/2) - 1$. So for example at a sampling rate of 64kHz and downsampling by a factor of 32, we can use carrier of $2k$ kHz, where $1 <= k <= 15$. In Figure 3.16, $k = 5$. Carriers of 30kHz and 26kHz ($k = 15$ and 13) have worked well for us. In Figure 3.16, exactly five periods of the sine wave fit within thirty-two samples at the main sampling rate. The dots indicate the points that the sine wave is sampled at the main sampling rate, and the larger circles indicate the points at which the downsampling mechanism chooses a sample.

Each of the four input antenna signals contains an amplitude modulated sinusoidal carrier for each drum stick. The first stage in demodulation is to separate the two carriers using bandpass filters. The ideal box filter for a given carrier would be centered at the frequency of that carrier, and would be just wide enough to accommodate all of the gesture-generated sidelobes without attenuation. The current version of the Audio-Input Drum patch uses cascaded biquadratic bandpasses designed using MSP's filtergraph object, and Figures 3.17 and 3.18 show the attenuation and gain, respectively, of the 26kHz and 30kHz filters.

The amplitude of the demodulated information is dependent on the phase of the input carrier wave. Since the driving frequencies are generated in software, it's possible to adjust

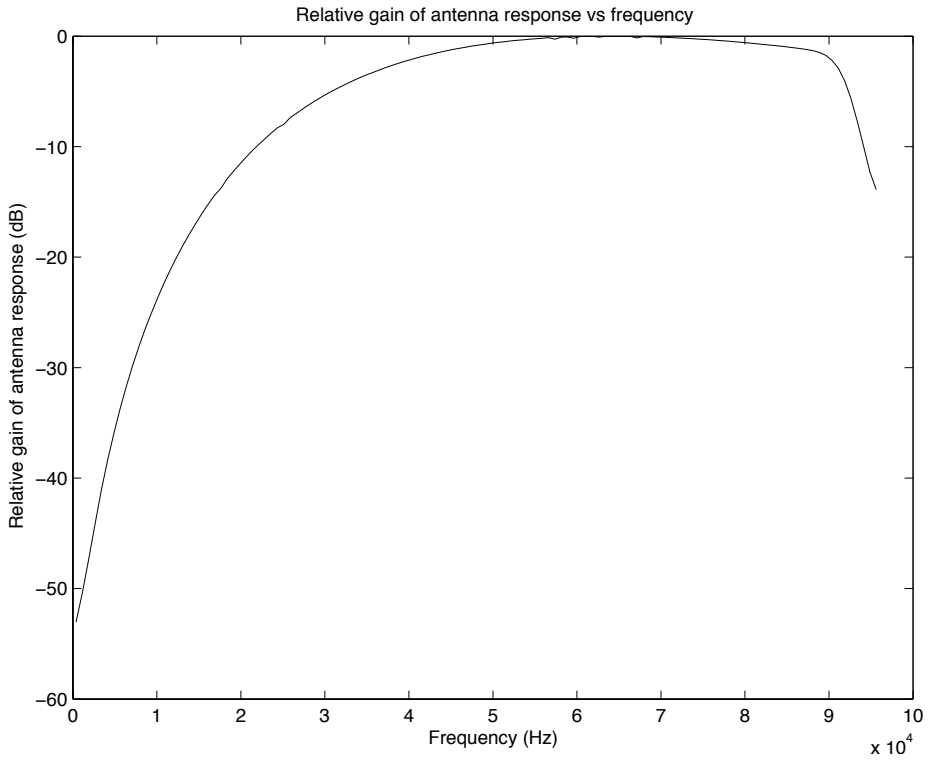

Figure 3.15. *Gain of the antenna (relative to maximum gain) versus the frequency of the carrier.*

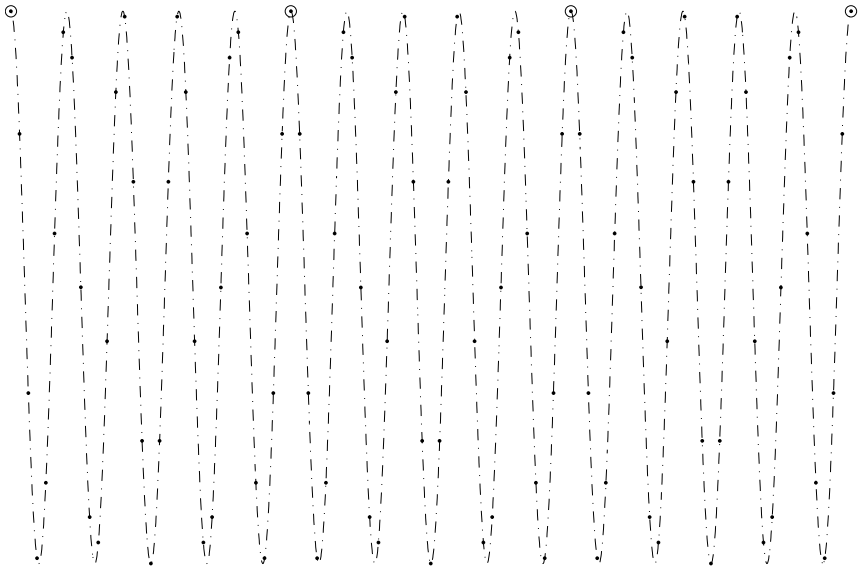

Figure 3.16. *Illustration of downsampling technique used for the Audio-Input Drum.*

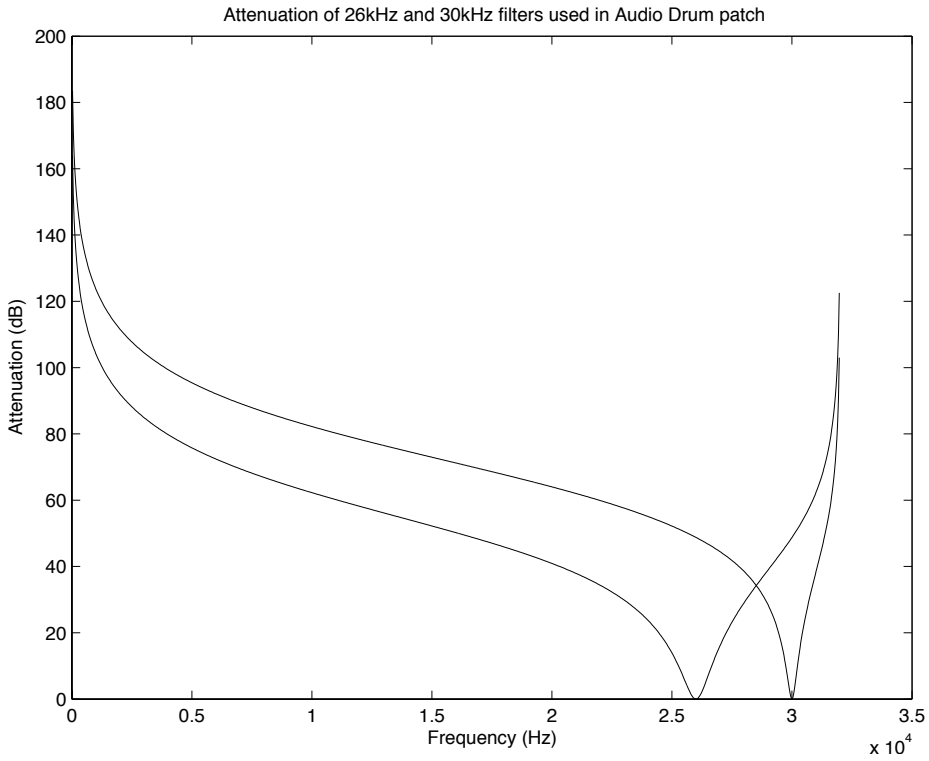

Figure 3.17. *Attenuation of the 26kHz and 30kHz bandpass filters implemented in the Audio-Input Drum patch.*

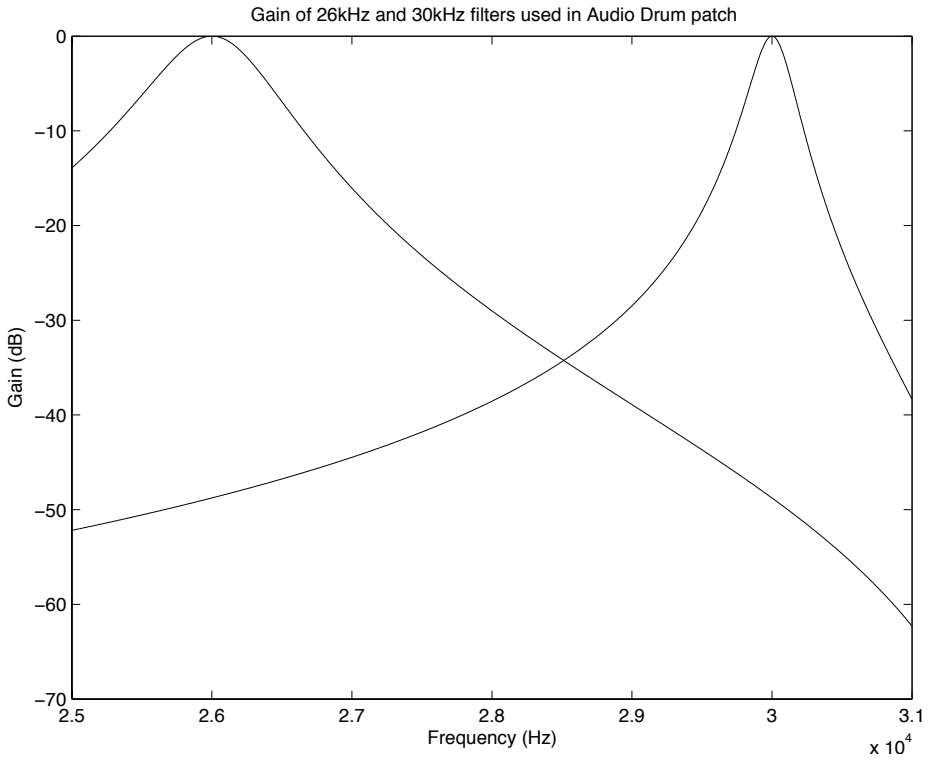

Figure 3.18. *Gain of the 26kHz and 30kHz bandpass filters implemented in the Audio-Input Drum patch.*

their phase so that the downsampled data is acquired at the zeniths of the sinusoidal waves to maximize the signal-to-noise ratio of our instrument. In our implementation this is done once by the user for each carrier and saved as an attribute of the patch.

In a noiseless environment the factors that limit the signal-to-noise ratio of the instrument are the bit depth of our audio interface and the signal strength transmitted by the sticks - currently a little more than 6 volts peak-to-peak. Of course we don't live in a noiseless environment; in addition to broadband white noise, the antenna is sensitive to electromagnetic interference from common sources such as 60Hz hum and radiating computer peripherals, and less expected sources such as refrigerators and light bulbs. Identifying and dealing with such sources of noise is important for achieving reliable operation with the interface.

Chapter 4

Analyzing Gesture Data

4.1 Introduction

Since the birth of the computer, the most often discussed factor in terms of accelerating our culture has been the rapid increase in the speed of hardware. However, the most dramatic increases in productivity have been because of shifts in the world of software. Indeed, being able to store a program that provides automated execution of an arbitrary chain of commands increased the productivity of the first computers by some thousands of percent [38].

With the previous generations of Radio Drum and Radio Baton, reprogramming the hardware in its native machine code was difficult enough that it was never done, even though there have been important problems with the whack velocity estimation algorithm. It was found that even with the original engineer's full cooperation, rewriting the code and burning a new PROM for the controller every time we wanted to try a new idea was extremely cumbersome and limited our ability to develop the system. In a 1991 paper "Communicating with Meaningless Numbers", David Zicarelli proposed that gesture analysis algorithms be liberated from hardware [39]:

> As an illustration of why gesture analysis is best considered an element of the overall control exercised by the CPU, it should be noted that Max Mathews didn't devise his notion of the reset plane and trigger plane for extracting hits from the radio drum until he bypassed the microcontroller that turned the

> drum's signals into serial data and digitized the analog signals directly on his PC host computer. He then used the analog-to-digital converter card's DMA capability to transfer the signals directly into the PC's memory, where he had complete control over how they should be interpreted. Given the direct access to the continuous gesture, and a scheme fast enough to use the data for rudimentary analysis, other people could invent their own schemes for extracting features such as hits from the drum.

The Audio-Input Drum realizes Zicarelli's vision of having the gesture signals processed with algorithms implemented in high-level software. The barrier to entry for reprogramming the system is much lower, changes are able to be made rapidly, and the result is a more capable system that can more easily adapt to present and future needs.

At the end of the day we wish to respond to the gestures with some sort of synthesis - sound, image, or some other form of output. To respond with a wide-ranging and consistent synthesis, it's important to accurately **estimate** the relevant parameters of our input signal. Estimation is made complicated by the inevitable presence of noise in the signal.

Beyond estimating parameters of the gestures, we may want to be able to extract gestural **events** from the continuous flow of data. This chapter discusses the definition of events within the context of the gestural parameters, the designing of tests to detect such events, and the statistical determination of the likelihood of test errors. When analyzing such errors in an engineering context we can consider the classic probabilities of a **miss** and a **false alarm**. We'll do our best to reduce both of these probabilities, although it's impossible to simultaneously minimize both. A related idea is the fact that we must pick a point in the tradeoff between **latency** and **sensitivity**, which is also linked to the probabilities of failure. The nature of statistical analysis intrinsically links all these test qualities. The final section of the chapter discussed some details related to the practicalities of triggering synthesis from signal events in the Max/MSP environment, including the creation of the new **event~** object to make convenient the transition between the signal and message worlds.

4.2 Parameter Estimation

With the capture of analog signals comes the inevitable fact of life that the data that we import into our application will always be corrupted by noise. We can model a stationary input signal as

$$x_j = s_j + n_j \tag{4.1}$$

where s_j and n_j are the signal and noise for sample j, respectively.

4.2.1 Analysis of Noise

We can characterize this noise by examining an estimate of its autocorrelation function. Data was recorded with both sticks sitting motionless on the surface of the drum. Figure 4.1 contains eight plots, each of which is an autocorrelation estimate for one of the eight signals, with DC offset removed, using the following formula:

$$R_N(k) = \frac{1}{N} \sum_{i=0}^{N-|k|-1} x_i x_{i+|k|} \tag{4.2}$$

where the lag k in these estimates ranges from 0 to 200 (0 to 100 ms at the sampling rate of 64kHz downsampled by a factor of 32: 2kHz). This sample autocorrelation function is asymptotically unbiased when N, the number of samples, is large compared to the lag k [40]. In the data that generated these plots, $N = 33801$. A white signal has $R(k) = 0$ for $k \neq 0$; none of these plots matches that description precisely, but in general they have "whiteish" behaviour, with a large spike around $k = 0$ and tapering off towards zero as $|k|$ gets larger.

The cross-correlation estimates shown in Figure 4.2 are helpful for evaluating the independence of the noise in each channel. Since there are eight channels, $7+6+5+4+3+2 = 27$ total cross-correlations could be examined. In Figure 4.2 we show only the 7 that involve channel 1, as well as its own autocorrelation. It's clear that the noise in a given channel will have more in common with some channels than others; the plot for channels 1 and 3 shows

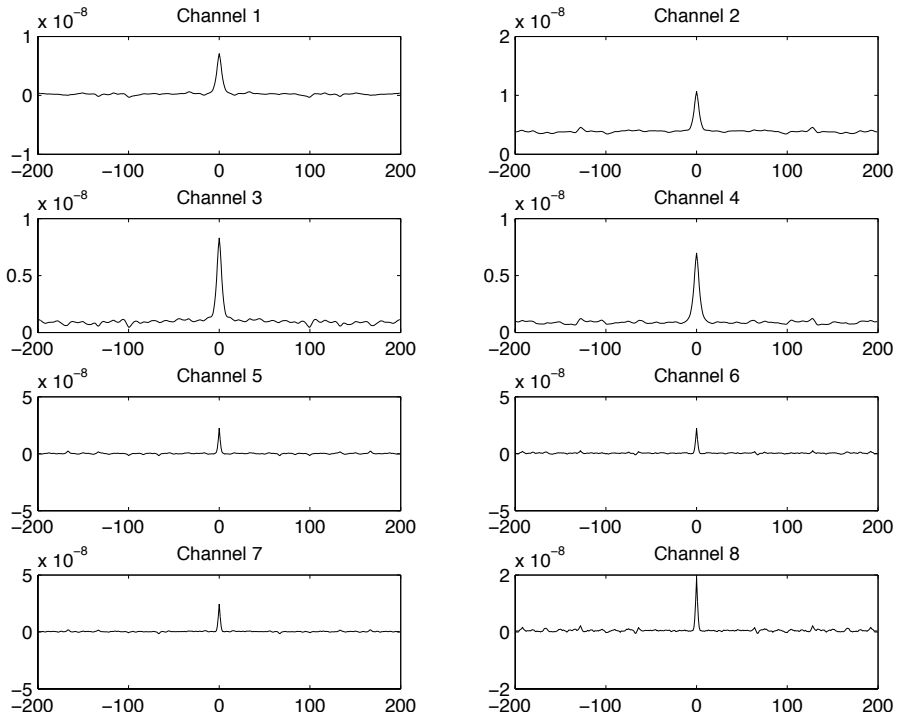

Figure 4.1. *Autocorrelation estimates for the noise in the eight signals.*

a strong connection at the $R_{1,3}(0)$ point, whereas the plot for channels 1 and 4 shows less of a bond, and the plots involving channels 5 through 8 even less still.

Strictly speaking, Figures 4.1 and 4.2 reveal that the noise is not zero-mean gaussian noise, and that the noises are not strictly independent from one another. Future research may be able to better characterize the noise and use it to improve each signal's estimate. In the meantime, for the purposes of our analysis we will still choose to make the simplifying assumptions of independent, zero-mean gaussian noises.

4.2.2 Averaging Filters

One way to reduce the effect of noise in the estimate of a signal is to use multiple values. Averaging n successive samples of a stationary signal corrupted with zero-mean gaussian noise reduces the variance in an estimate by a factor of n. However, we cannot reasonably assume that our signal is stationary. It could be reasonable to think of the signal as locally stationary if the sample rate of the gesture signal is significantly higher than the maximum gesture frequency. In the case of our implementation of the Audio-Input Drum, downsampling a 64 kHz signal by a factor of 32 leaves us with a 2 kHz data rate. This is only a little more than four times our maximum assumed gesture signal frequency of 450 Hz, so an assumption of stationarity is questionable.

A more visible way to think about the effect of such an averaging is to think of the averaging process as a filter. Figure 4.3 contains plots of the gain of filters that result from averaging 2, 3, 4, and 5 samples. As expected the frequency of the first lobe of the filter decreases with increasing n. However, the sidelobes, which reach as high as -10 dB in the case of $n = 3$, make the rejection characteristics poor. Compare Figure 4.3 with Figure 3.17 and it's obvious that the averaging filters are crude compared even with the filters that are possible even with simple biquadratic forms. We could achieve a more optimal noise-power reduction by designing a special IIR lowpass filter. Furthermore, because of the frequency shift that happens in the synchronous downsampling, this lowpass filtering can effectively be accomplished in our bandpass filtering of the original 64 kHz amplitude

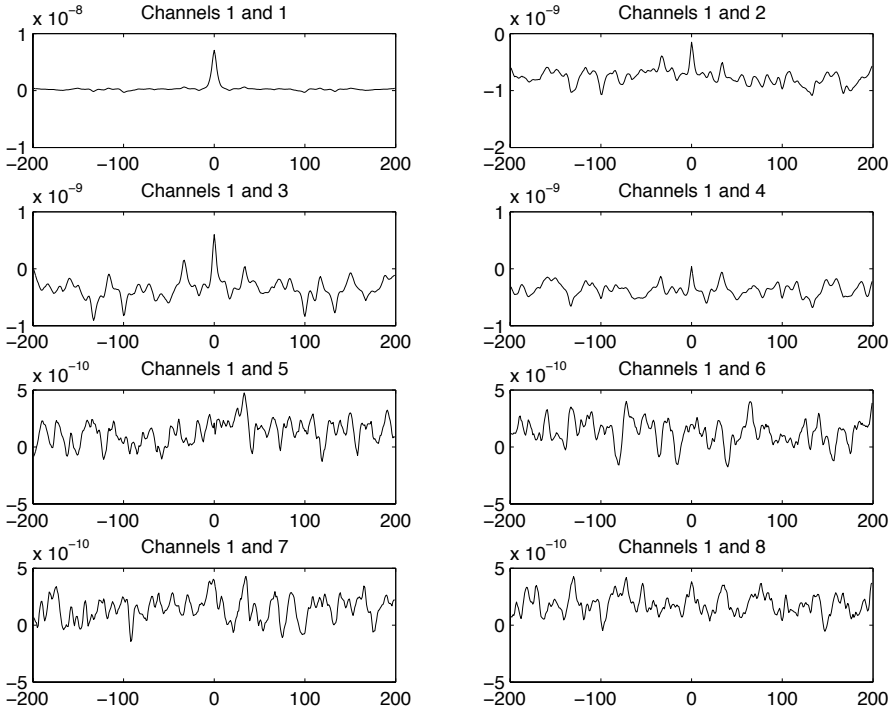

Figure 4.2. *Cross-correlation estimates for the noise in each of the eight signals with the first signal (channel 1).*

modulated waveforms.

4.2.3 Height Estimation

The printed circuit board that serves as the antenna for the Audio-Input Drum is actually made up of four separate regions; section 4.2.4 discusses the geometry of these regions in detail. These four regions are useful primarily for estimating position in the two planar dimensions, but for estimating the vertical height of the stick above the surface we can treat the surface as a single planar antenna by adding the contributions of the four separate antennae together. That is,

$$s = s_A + s_B + s_C + s_D \tag{4.3}$$

where s_A, s_B, s_C and s_D are the signal values from the four individual antennae.

When a stick is close to the drum surface we can make the simplifying assumption that the antenna plane stretches infinitely in all directions. Consider a frame of reference with the stick's x and y position being the origin, the $z = 0$ plane as the antenna surface, and the transmitting stick a height z above the antenna, as shown in Figure 4.4. The induced charge density on the antenna surface as a function of x and y is

$$\sigma(x, y) = \frac{-qz}{2\pi(x^2 + y^2 + z^2)^{3/2}} \tag{4.4}$$

where we are treating the stick as a point charge of magnitude q [41]. Figure 4.5 is a plot of the received signal strength versus the height of the drum stick above the center of the small 10 inch by 11.5 inch antenna surface. Figure 4.6 is the same plot with both axes logarithmically scaled. We might expect an inverse-square relationship from the form of equation 4.4, and we can test for this by using the collected data in the logarithmic scale. If the relationship between the signal strength s and the height z is

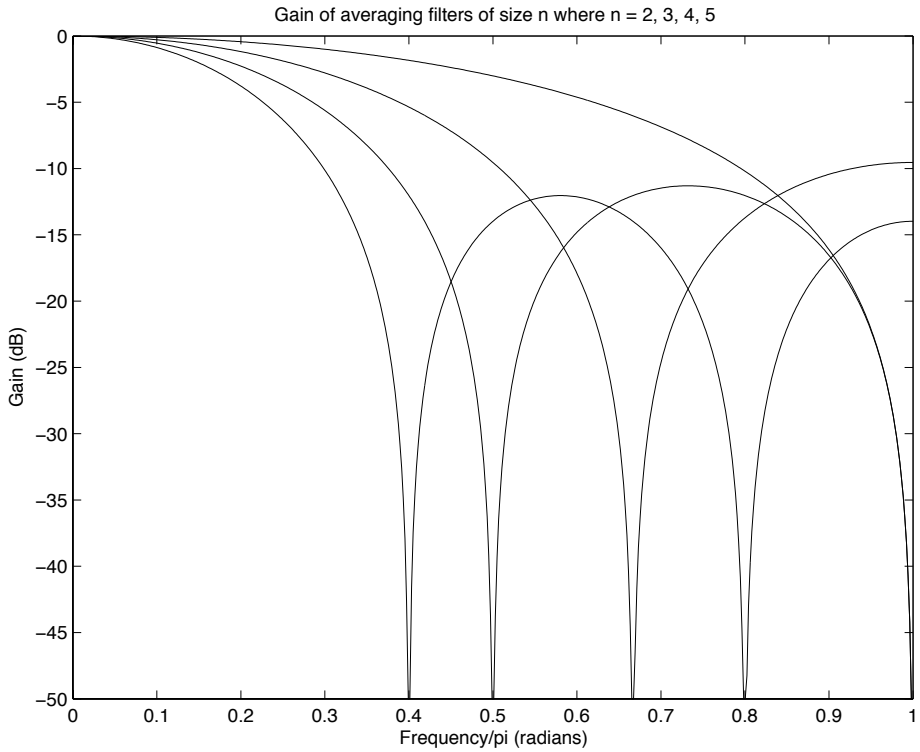

Figure 4.3. *Gain of averaging filters of size n where n = 2, 3, 4, 5.*

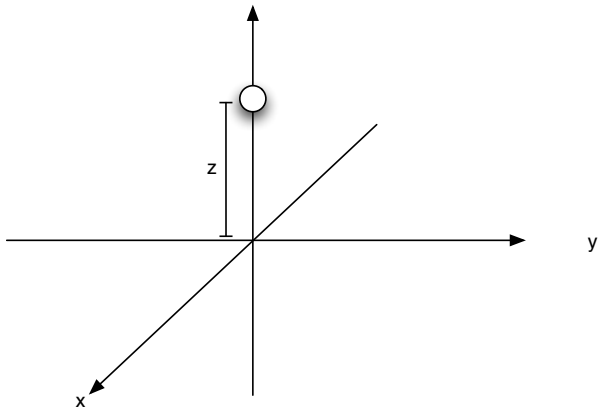

Figure 4.4. *A point charge a height h above the surface of an infinite plane.*

$$s = c_1 z^{c_2} \tag{4.5}$$

where c_1 and c_2 are unknown constants, then taking the logarithm of both sides yields

$$\log s = \log c_1 + c_2 \log z \tag{4.6}$$

This expresses a linear relationship between the two variables $\log s$ and $\log z$. An examination of the log-scale plot in Figure 4.6 shows that the behaviour of this logarithmically-scaled signal strength is not linear when the stick is close to the surface. However, the straight line in Figure 4.6 is the result of a linear fit with the last twelve points of data taken, which range from 21.2 cm to 100.9 cm. The slope of this line is $c_2 = -2.049$, which is consistent with our expectation of an inverse-square relationship.

The fact that the inverse-square relationship does not hold at small distances makes precise vertical tracking difficult. Future research could aim at modeling the field-space of the rectangular antenna more accurately and perhaps tracking the position of the sticks precisely, but there are many complicating factors, including the asymmetry of the transmitting sticks, influence of one stick's electric field on the other's, and the influence of other conductors in the area, such as the body of the performer. For the purposes of our work it is not necessary to precisely estimate the cartesian position of the stick. It is only

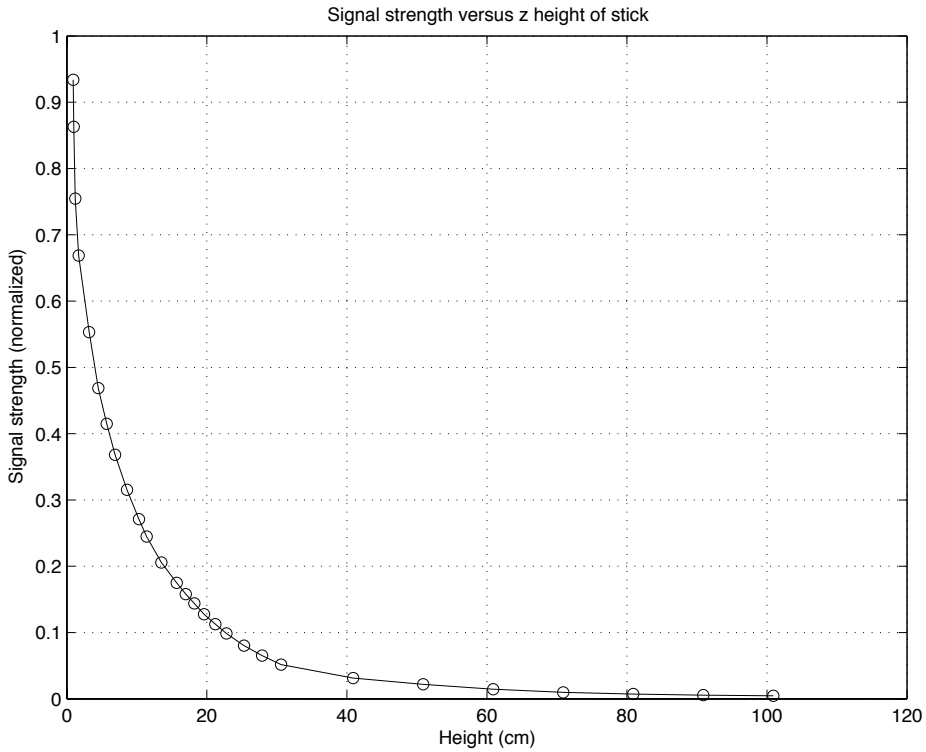

Figure 4.5. *The signal strength as a function of the stick's z height above the surface.*

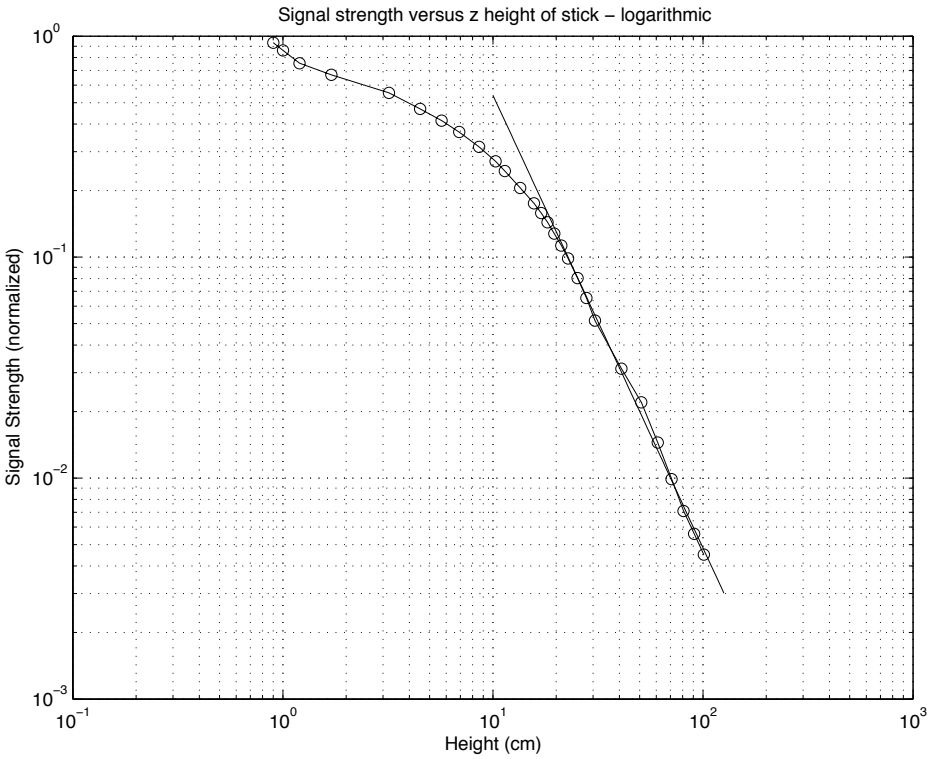

Figure 4.6. *The signal strength as a function of the stick's z height above the surface, plotted logarithmically. The straight line fit through the last twelve points has a slope of -2.049, indicating the inverse square relationship we expect.*

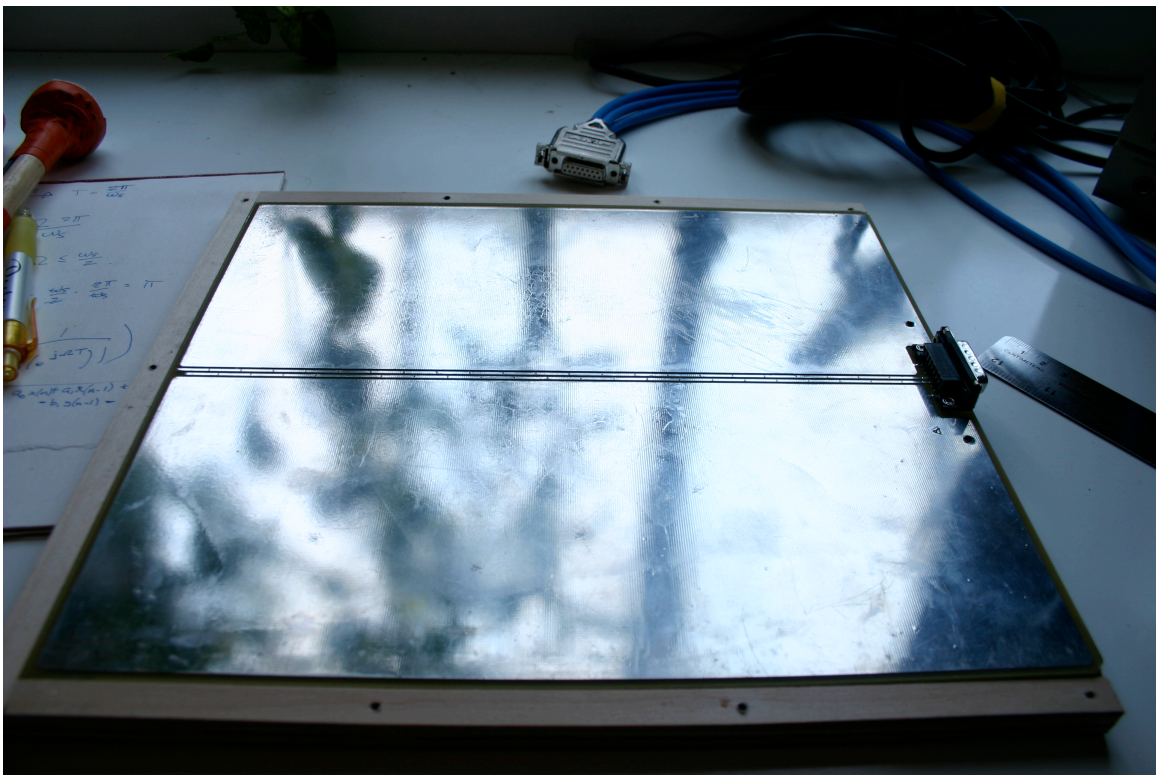

Figure 4.7. *Bottom of the drum antenna surface*

important that the strength of the received signal monotonically increases as the distance from the stick to the antenna decreases. We can therefore track the height of a stick simply with the sum of the four antennae values. ie,

$$\hat{z} = s_A + s_B + s_C + s_D \tag{4.7}$$

If each of s_A, s_B, s_C, and s_D is an independent random variable with gaussian noise, the sum $s_A + s_B + s_C + s_D$ is a random variable with gaussian noise. If the four signals have identical variance σ_s^2, the variance of this sum will be $\sigma_z^2 = 4\sigma_s^2$.

4.2.4 Planar Estimation

Figures 2.8 and 4.7 show the top and bottom, respectively, of the circuit board that serves as the Audio-Input Drum's antenna. This is the same antenna that is used in the most

recent version of the Mathews Radio Baton. Figure 2.9 illustrates the four distinct regions of the antenna that are intended to operate in pairs: two of the regions are used to estimate position in the x dimension, and the other two to estimate position in the y dimension. The x estimating pair is made up of vertical bars of varying thickness: the thickness of the bars for one antenna increases with increasing x, whereas for the other antenna the thickness decreases with increasing x. The y estimating pair are made up of long, thin triangular wedges that sit with their bases on opposite edges of the circuit board. The effect of these positioning of these regions is that the received signal strength for each of the antennae is strongly associated with a different edge of the circuit board. Previous versions of the Radio Drum and Radio Baton have featured different geometries for these antenna regions; the original Radio Drum, for instance, featured a strictly "backgammon" pattern with four regions that had signal strengths associated to the four corners of the antenna surface.

Although we do not need to know precisely the z-height of a stick to make use of it for musical purposes, it is more important that we have an accurate way to detect the x and y position of a drum stick. A common performance mode is to map drum hits at different parts on the drum's surface to different musical processes. Figures 4.8 and 4.9 show the signal strength as a function of motion in the x and y dimensions, respectively. For both the z height was less than a centimeter. In each case, two of the signals remain roughly constant, while the other two linearly increase and decrease, crossing somewhere in the middle. These figures show that each of the four signals corresponds to a different edge of the drum, and that near the center of the antenna the change in the signal strength is linear with the position.

We can therefore estimate the x and y position of a stick as follows: let s_A and s_B be the signals induced on the pair of antennae responsible for the estimation of either the x or the y transmitter position. The relative position p_{AB} between the A and B edges can be estimated with the following formula:

$$p_{AB} = \frac{s_A}{s_A + s_B} \tag{4.8}$$

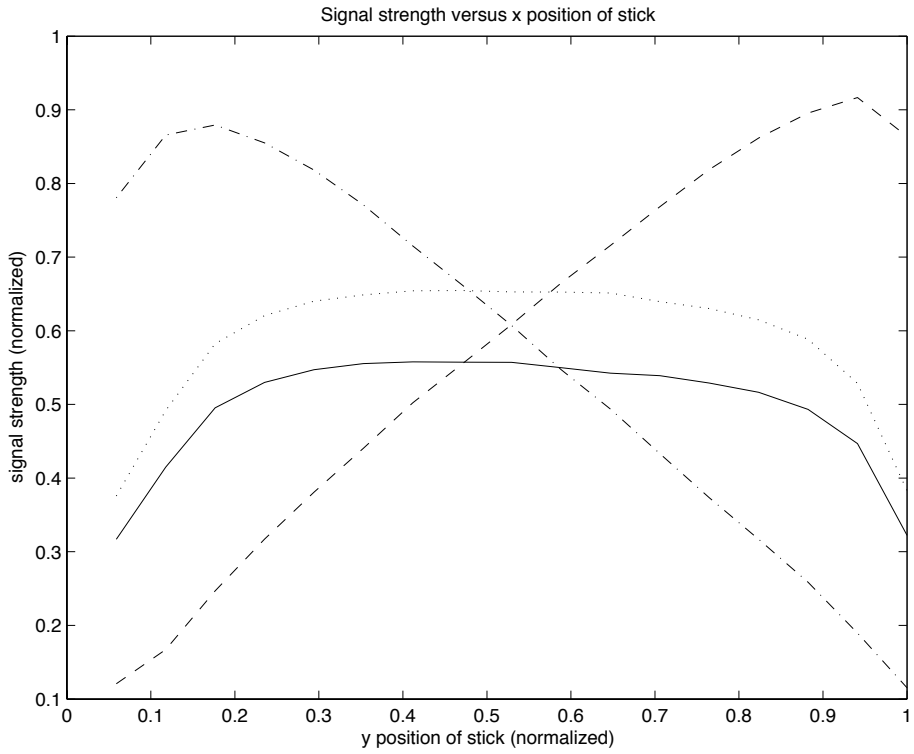

Figure 4.8. *The signal strength for each of the four antennae as a function of the stick's x position above the surface.*

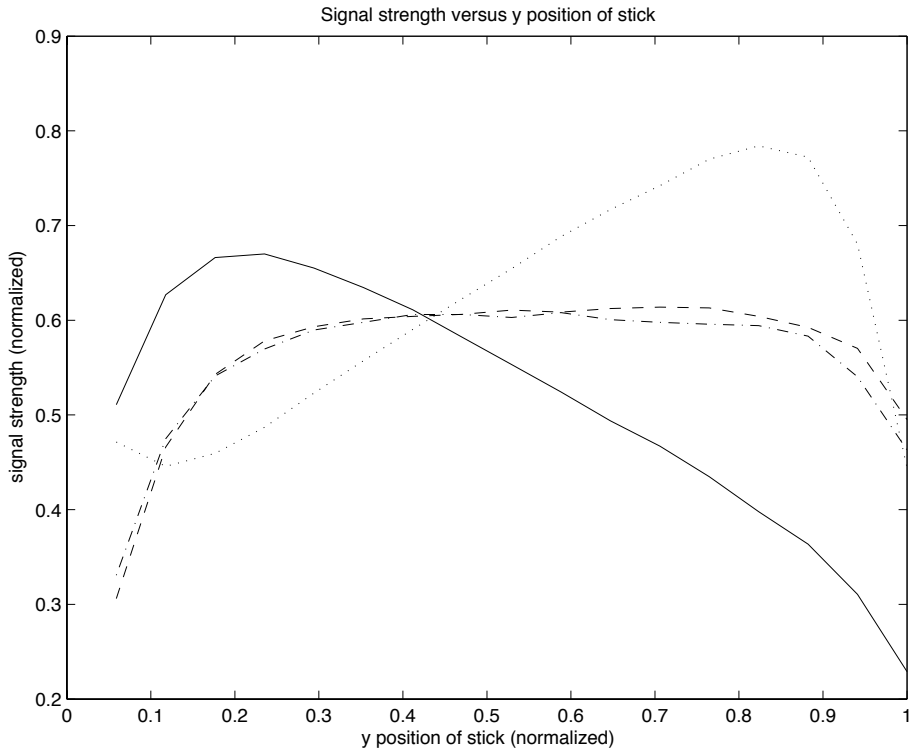

Figure 4.9. *The signal strength for each of the four antennae as a function of the stick's y position above the surface.*

This formula uses the linear nature of the two opposing signals, and its ratio formulation allows for a change in vertical position that will gain each antenna's response equally. To use the estimate effectively we subsequently rescale the output of this ratio to the desired range, eg 0 to 1 or -1 to 1. As is obvious from Figures 4.8 and 4.9, this simple model is unusable near the edges of the drum surface.

A random variable formed from the ratio of two independent gaussian random variables is a Cauchy random variable. In 1979, Hodgson wrote a paper about what has since come to be known as Hodgson's paradox [42]. The paradox is that the ratio of two normally distributed random variables has neither a mean nor a variance, and therefore no well-defined expectation of behaviour. In real life no random variable is exactly Gaussian, so the paradox is never truly realized. In our case, s_A and $s_A + s_B$ are clearly not independent variables, so the situation is further complicated. The further away the sticks are from the surface, the larger the components of noise relative to the signals in s_A and s_B, and so the x and y estimation quickly becomes useless. Luckily for our primary application, deciding where on the surface the drum has been hit, the signal-to-noise ratio is at its zenith.

Figure 4.10 shows a three-dimensional plot that renders in real time while the Audio-Input Drum software is running. The blue plane represents the drum surface and the two "snakes" trace out the position of the sticks as they are moved. Every estimated position that the sticks pass through is rendered, which makes this display enormously helpful in some debugging situations.

4.3 Event Detection

4.3.1 Introduction

The concept of an event is a powerful artistic tool. Consider the western system for notating music: each note is an event, and the horizontal dimension represents time so that a sequence of events is defined by the horizontal orientation of notes arranged on the mu-

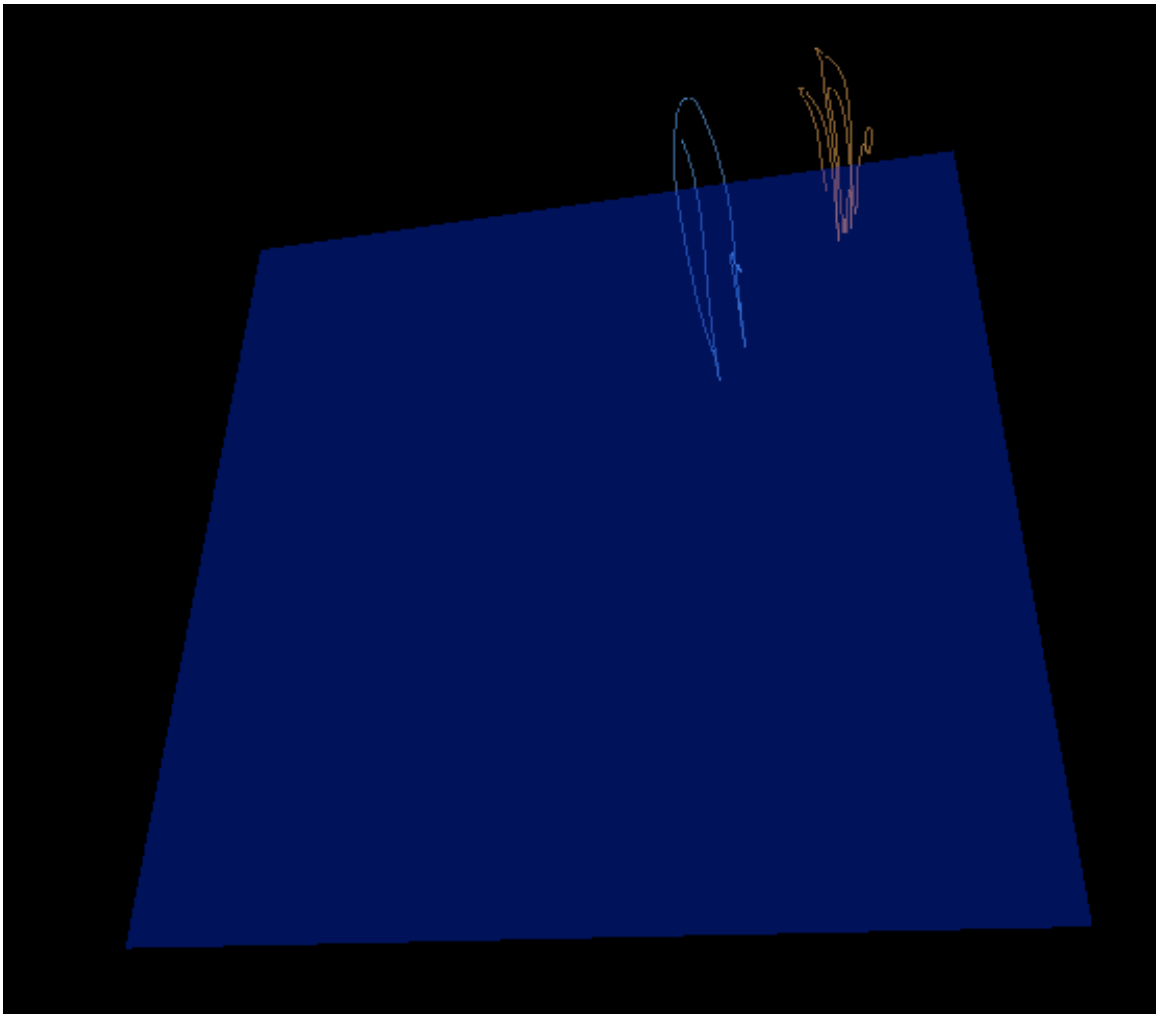

Figure 4.10. *A realtime three dimensional plot of the x, y, and z position estimates of the two sticks. The blue plane represents the drum surface and the two "snakes" trace out the position of the sticks as they are moved. This display has been enormously helpful in some debugging situations.*

sical staff. Data for the pitch and duration of the note is encoded in the vertical position and shape of the note, respectively. The intended character of the notes can be further influenced by expression and dynamics markings.

The MIDI file format takes a slightly different approach to representing most of this same information. Instead of a note having a start time and a duration, in MIDI there is only the concept of ”note on” and ”note off”. This encoding works fairly well when recording the information from a performer playing a piano keyboard. Unlike musical notation, in which the dynamics of the notes are implied by superannotations and left up to the interpretation of the performer, a MIDI file also associates a 7-bit ”velocity” value with each note to represent the force with which the note is played.

While a MIDI file can do a reasonable job of representing a piano keyboard’s possibilities, for many other instruments the MIDI protocol could not do an adequate job of capturing all the nuances of performance [43]. Consider the violin, whose pitch and amplitude components of performance are both continuous variables. Although we could logically consider the entire motion of fingering a steady pitch and bowing steadily across a string as an event, to encode the intricate complexities of the violin’s interface with any simple, concise event-based grammar would be difficult. It is in this way that our highly resolved gesture signals will help us create rich instruments: rather than try to represent an interface via the paradigm of a simplified piano as the MIDI protocol does, we instead capture as many of the subtle nuances of a continuous gesture as we can inside the constraints of the sample rate and bit depth of our data signal.

So in the context of the above discussion it is tempting to propose that we do away with the notion of a musical ”event” entirely. But an event is still a useful concept in terms of organizing the output synthesis. For instance, if we were attempting to emulate a violin, one type of event we would want to detect would be the contact of the bow with the strings. The end of this gestural event could be defined as the time at which the bow and the strings separate. In the case of the Audio-Input Drum, we of course would like to know when the surface of the drum has been hit. However, more complicated analysis is possible: for

instance, what if the drum stick is depressed into the surface and held firmly in place? A performer might use this gesture to dampen a resonating sound, or perhaps to modulate the pitch of the resonance. Indeed, the stages of an event can be defined in any way, so therefore for events of interest the problem becomes one of identifying the conditions in the gesture signals that are true when the event you want to detect has occurred.

4.3.2 Detecting a hit

The ”surface” of the Audio-Input Drum is a plane of constant z height somewhere above the actual antenna pad's surface. A layer of spongy foam placed on top of the antenna can provide a near-silent surface to strike with the drum sticks, but care should be taken to ensure that anything placed on the surface does not impact the electric field so much that the monotonic nature of the estimation of the stick's vertical position is compromised. Some types of foam that we tried are susceptible to the buildup of electrostatic charge. The eventual sudden discharge of this excess charge would manifest itself as sudden large discontinuities in the gesture signals. The near-silence of the interface is a musically important detail, since it helps to effectively decouple the striking of the interface and the resulting sound.

Figure 4.11 illustrates the hit detection technique used by the black box of the old Radio Drum. When the estimated z-position of a stick would fall below a certain z-threshold value, a hit would be triggered when a local minimum was reached. The **depth** of the dip - that is, the distance between the z-threshold and the minimum - was used as the sole determinant of the estimated **velocity** of the hit. The drum's hardware would then communicate the x, y, and velocity estimates for the hit via the serial connection to the computer. A type of hysterisis was implemented so that the drum could not be re-triggered until the z-estimate rose back above the z-threshold.

The algorithms in the original drum had problems with velocity estimation. Performers noted that frequently the reported velocity value of a strike would be wrong - a forceful hit that was reported with small velocity was nicknamed a ”duff”. Without access to the

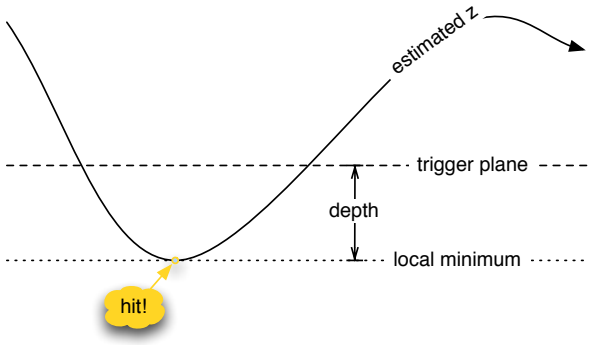

Figure 4.11. *Illustration of the hit detection technique used by the original Radio Drum. If the trigger plane was crossed, the minimum of the estimated signal would trigger a hit. The depth of the hit (distance between the threshold and the minimum) determined the estimated velocity of the hit.*

demodulated signals and hit-detection algorithms in hardware, determining what caused this problem was difficult, but it may have been triggered by a signal estimate as depicted in Figure 4.12. The hypothesis presented is that the signal drops below the z-threshold for a strike, and then experiences a small reversal of direction somewhere on its way down to the real minimum of its path. The reason for this reversal might have been something physical, or perhaps just noise in the signal. The mechanism in the Radio Drum hardware that determines the minimum of the estimated z-position signal is fooled by this small reversal, and hence reports a smaller velocity than would have resulted if the real minimum had been properly tracked.

Figure 4.13 illustrates the scheme used by the Audio-Input Drum to detect hits. Rather than use the minimum of the estimated z-position as the trigger point for a hit, the new scheme uses the maximum negative point of the estimated velocity signal. This maximum negative velocity value is also used as the estimate for the velocity of the hit. The velocity can be estimated from the estimated z-position samples $\hat{z}_i$ as follows:

$$\hat{v}_i = \hat{z}_i - \hat{z}_{i-1} \tag{4.9}$$

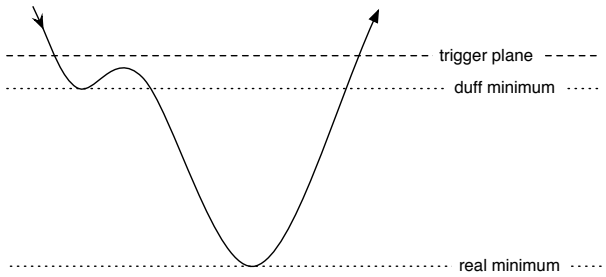

Figure 4.12. *Diagram of the hypothesis for the estimated z-position over time for the situation where a "duff" was reported by the original Radio Drum apparatus.*

The maximum negative velocity point corresponds to the time when the acceleration crosses from negative to positive. Detecting when the acceleration crosses from negative to positive therefore becomes the central problem for detecting when the drum has been hit. The acceleration can be estimated as follows:

$$\hat{a}_i = \hat{v}_i - \hat{v}_{i-1} \tag{4.10}$$

$$= \hat{z}_i - 2\hat{z}_{i-1} + \hat{z}_{i-2} \tag{4.11}$$

Figure 4.14 illustrates the process of detecting whether a signal's value is above or below a threshold. The solid area behind the curved signal represents the output of this ideal threshold detector: it flips upwards and downwards when the signal is above and below the threshold. Figure 4.15 shows a Max patch that features an MSP network that executes a simple test and uses Jitter to graph the results. Noise is added to a sine wave to imitate a noisy signal, and the signal is tested to see if it's greater than zero. True and false results are indicated by the background grey bar graph. As it approaches the zero threshold, the noise around the signal rapidly moves the combined value above and below the zero threshold, causing the result of the test to flip-flop many times. To make decisions in the presence of noise obviously will require methods more sophisticated than this naive approach.

Figure 4.16 illustrates one possible solution to this problem. Rather than use a single

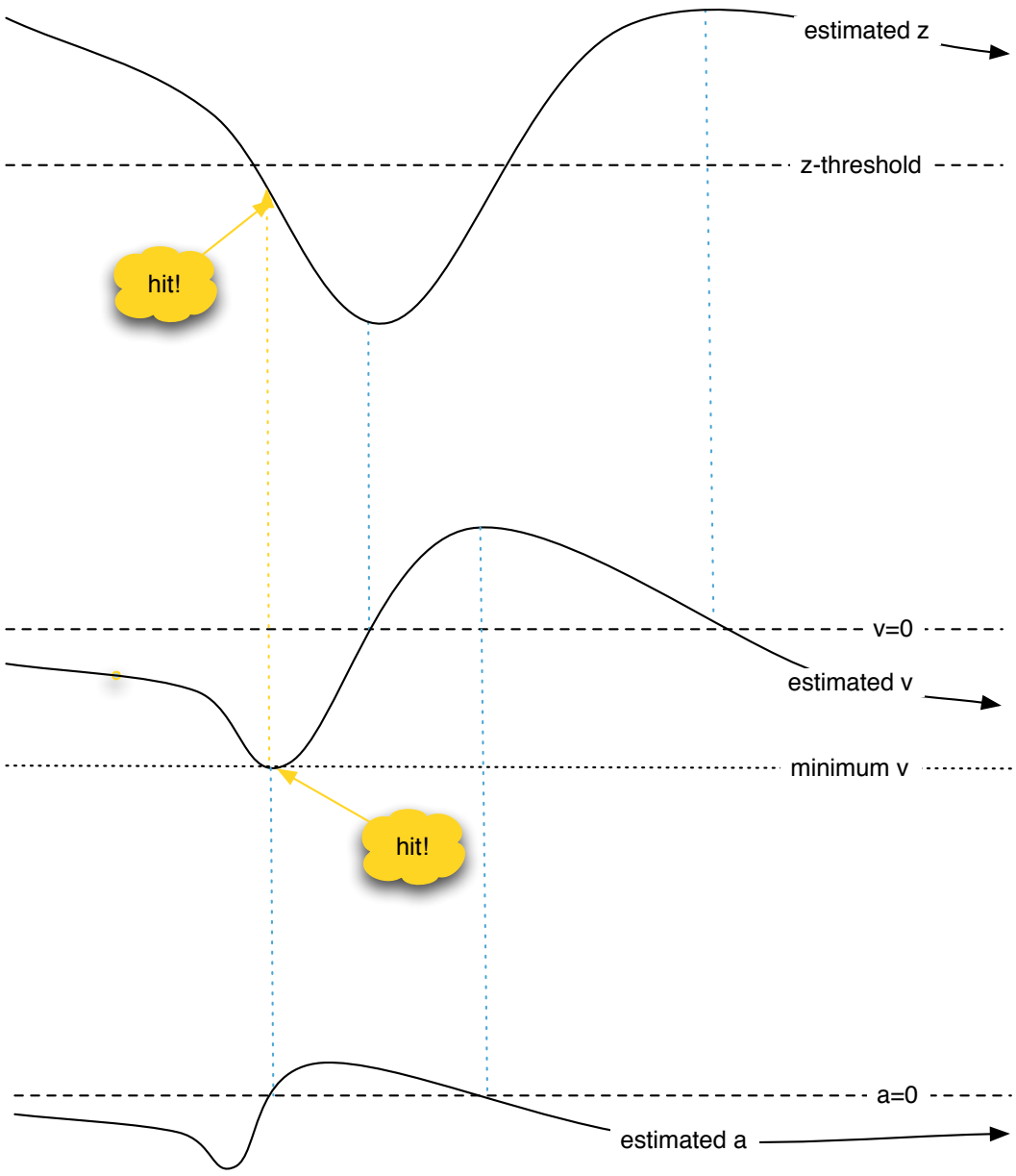

Figure 4.13. *Illustration of the hit detection technique used by the Audio-Input Drum. The maximum negative velocity is used to trigger a hit, but only if the signal is below the z-threshold. The maximum negative velocity corresponds to the estimated acceleration crossing above* $a = 0$*. The blue dotted vertical lines highlight the minimum of a signal, and the corresponding crossing of the zero line by its derivative signal.*

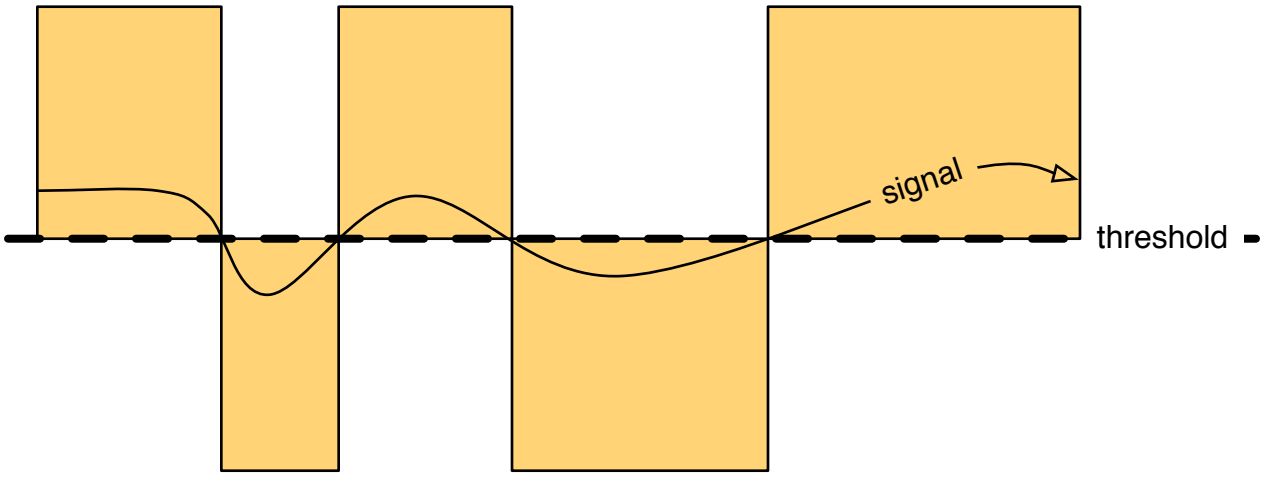

Figure 4.14. *An illustration of the input and output of a naive threshold test. The solid area behind the curved signal flips upwards or downwards depending on whether the signal is above or below the threshold.*

threshold, the test in this case uses distinct upper and lower thresholds for the transition from below to above and above to below their central value, respectively. This test is implemented in the thresh~ object, as illustrated in Figure 4.17. If the input signal moves from below to above the object's upper limit parameter, the output signal is set to 1. If the input signal drops from above to below the object's lower limit parameter, the output signal is set to 0.

The limitation of the threshold test presented in Figure 4.16 is illustrated in Figure 4.18, where the signal is above the central value of the test, but below the upper threshold level. This example exposes the cost that this dual thresholding-scheme imposes in terms of the resolution of the detector. Figure 4.19 shows a way to get around this problem of resolution by **integrating** the values of a signal. Small signal values above or below the central threshold can be detected if multiple samples of the signal are accumulated to help identify the overall trend. However, using multiple samples to detect state is only valid if the signal is **stationary**, and as we know from the discussion in Section 4.2.2, in our case our latitude to use multiple samples is small.

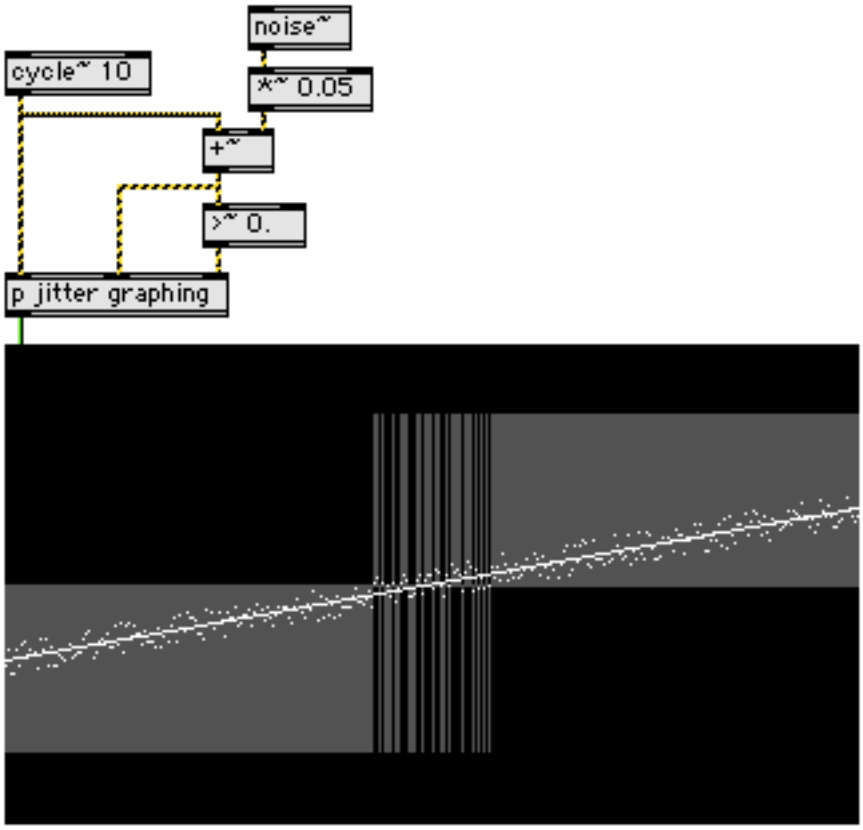

Figure 4.15. *Max patch featuring an MSP network that executes a naive threshold test and uses Jitter to graph the results. Noise is added to a sine wave to imitate a noisy signal. This signal is tested to see if it's greater than zero. True and false results are indicated by the background grey bar graph.*

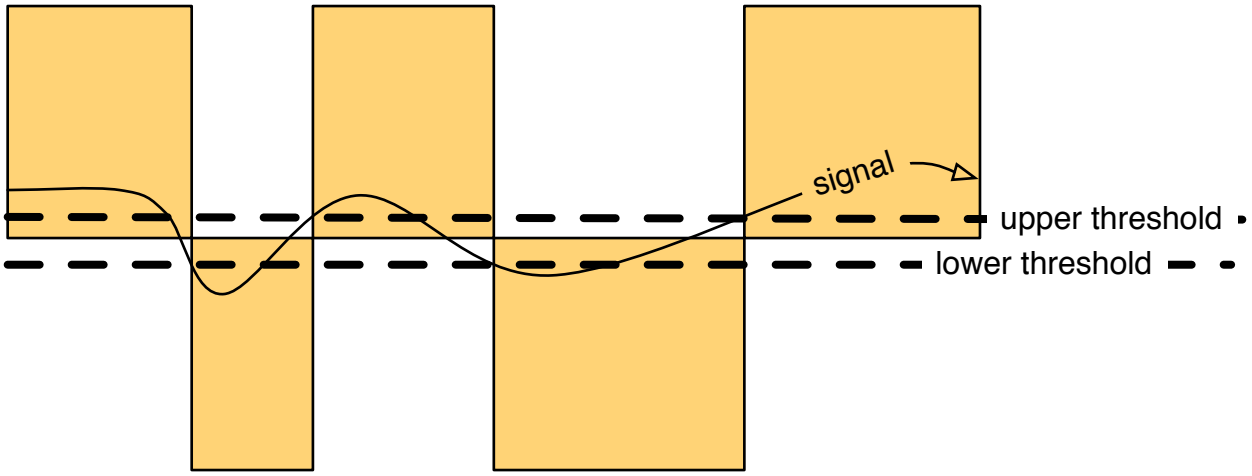

Figure 4.16. *An illustration of the input and output of a practical threshold test with hysterisis thresholds above and below the threshold. The solid area behind the curved signal flips upwards or downwards when the signal has crossed the above or below threshold, respectively.*

4.3.3 Sequential Analysis

The theory of **sequential analysis** allows us to design a test that adaptively scales the number of samples used based on the signal strength [44]. An introduction to the basic concepts behind this branch of statistical decision theory is presented in Appendix C. A Max/MSP object called **wald~**, named in honor of sequential analysis's inventor, Abraham Wald, was created to track the status of the acceleration signal. The code for this object is listed in Appendix D.

Figure 4.20 illustrates the flow of logic in the sequential analysis test. Input samples are accumulated, and if the value of the sum is either bigger than a rejection threshold or less than an acceptance threshold, the test is terminated. If the test is not terminated, the test continues by adjusting the thresholds and adding another input sample to the accumulation.

With reference to Figure 4.21, we'll examine the application of sequential analysis to the problem of determining whether a signal's true value θ is less than some fixed value v: in other words, the null hypothesis for our test is that $\theta < v$. In his theory, Wald defines

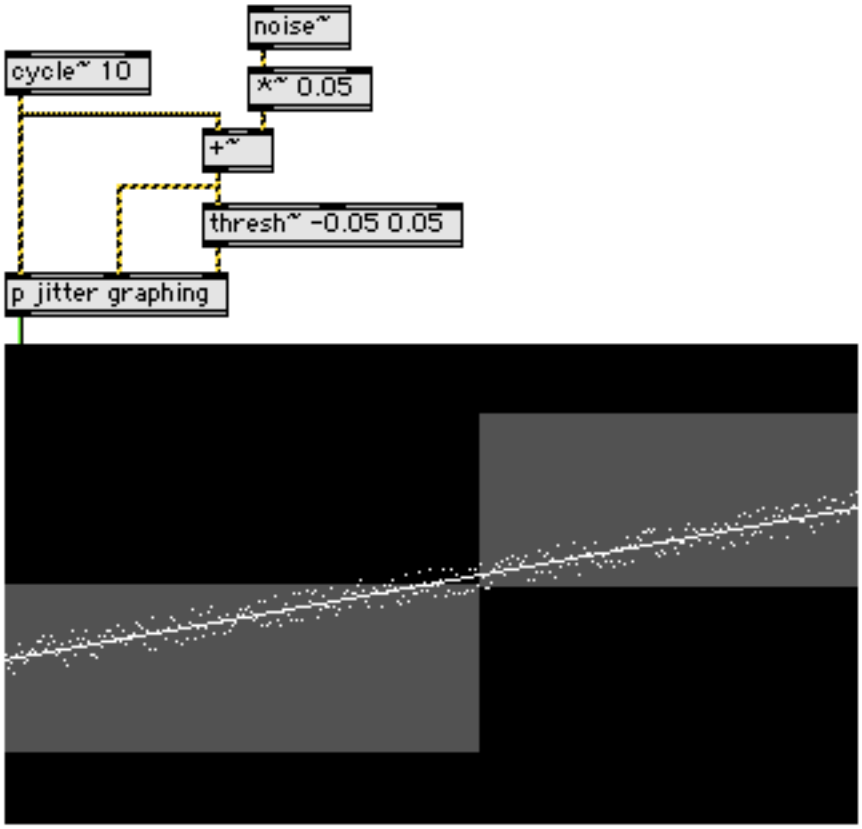

Figure 4.17. *The same network as in figure 4.15 but with a thresh~ object instead of a naive >~ object.*

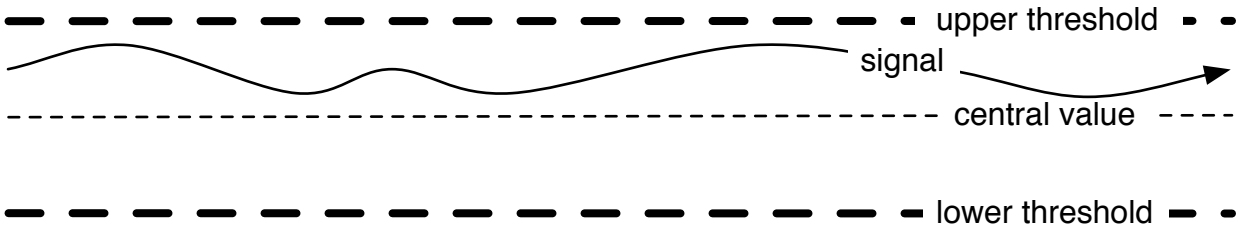

Figure 4.18. *Illustration of the limitation of the practical threshold test presented in Figure 4.16. The resolution of the detector is limited to values outside the thresholds; although they may be uniformly on one side of the central threshold, values in between the two outer thresholds do not trigger the threshold detector one way or the other.*

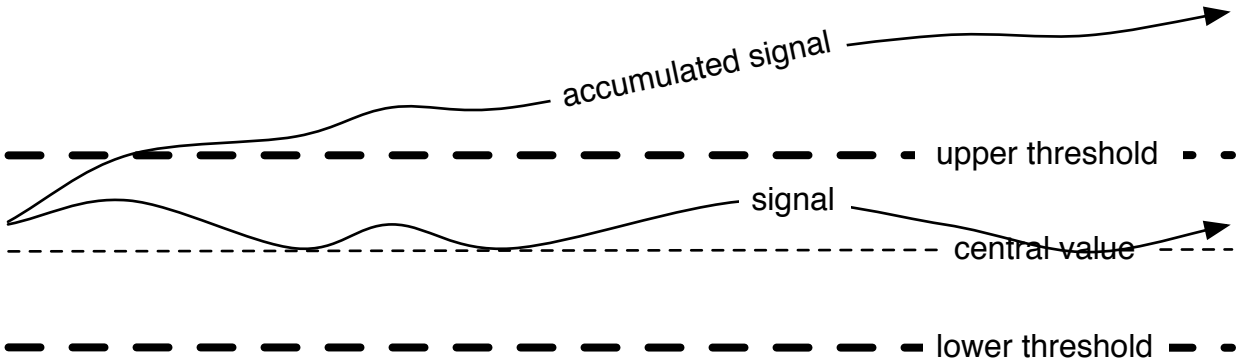

Figure 4.19. *Illustration of accumulating a signal to detect the overall trend of a signal value.*

what he calls a **zone of indifference** between two values θ_0 and θ_1. The idea is that if the true value θ is in between these limits, the results of the test will be indeterminate. However, if the true value θ is above θ_1 we should reject the null hypothesis as being false, and if the true value is below θ_0 we should accept the null hypothesis as being true.

Furthermore, we can adjust the thresholds to give us the error probabilities we desire. We let α represent the desired probability of rejecting the hypothesis when in fact the signal is in the zone of preference for acceptance, and β represent the desired probability of accepting the hypothesis when the signal is in the zone of preference for rejection; these are sometimes known as the **miss** and **false alarm** errors, respectively. Then if σ^2 is the variance in the signal, and m is the index of the sample in the test, starting with $m = 0$, the threshold of acceptance $thresh_a$ and the threshold of rejection $thresh_r$ can be expressed as follows:

$$thresh_a(m) = \frac{\sigma^2}{\theta_1 - \theta_0} \log \frac{\beta}{1-\alpha} + m\frac{\theta_0 + \theta_1}{2} \tag{4.12}$$

$$thresh_r(m) = \frac{\sigma^2}{\theta_1 - \theta_0} \log \frac{1-\beta}{\alpha} + m\frac{\theta_0 + \theta_1}{2} \tag{4.13}$$

Each of the thresholds in the above equations has a static **intercept** component that's dependent on σ^2, α, β, θ_0 and θ_1, and a **slope** $(\theta_1+\theta_0)/2$ that is added to the thresholds with

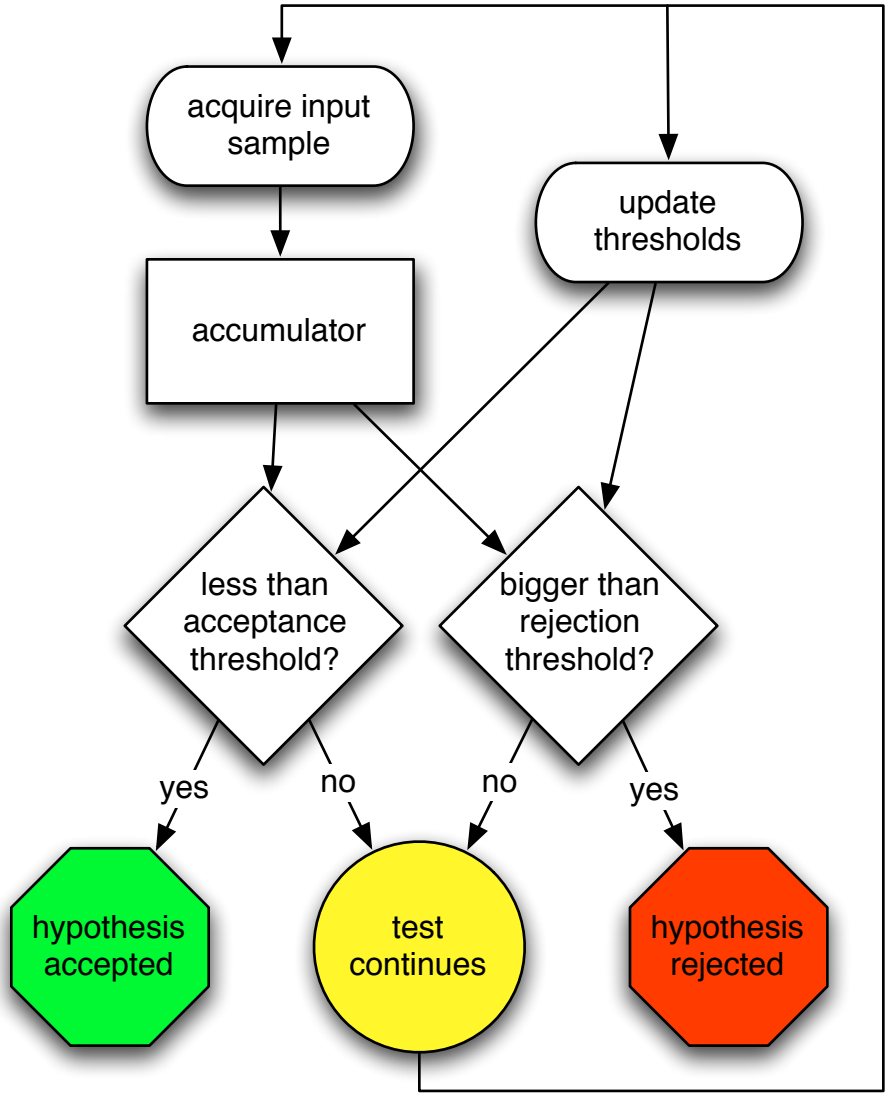

Figure 4.20. *Flow of data in the sequential test of whether the input signal is below a certain value. If the accumulated value of the input samples is either bigger than the rejection threshold or less than the acceptance threshold, the test is terminated. If the accumulated value lies in between these thresholds the test continues.*

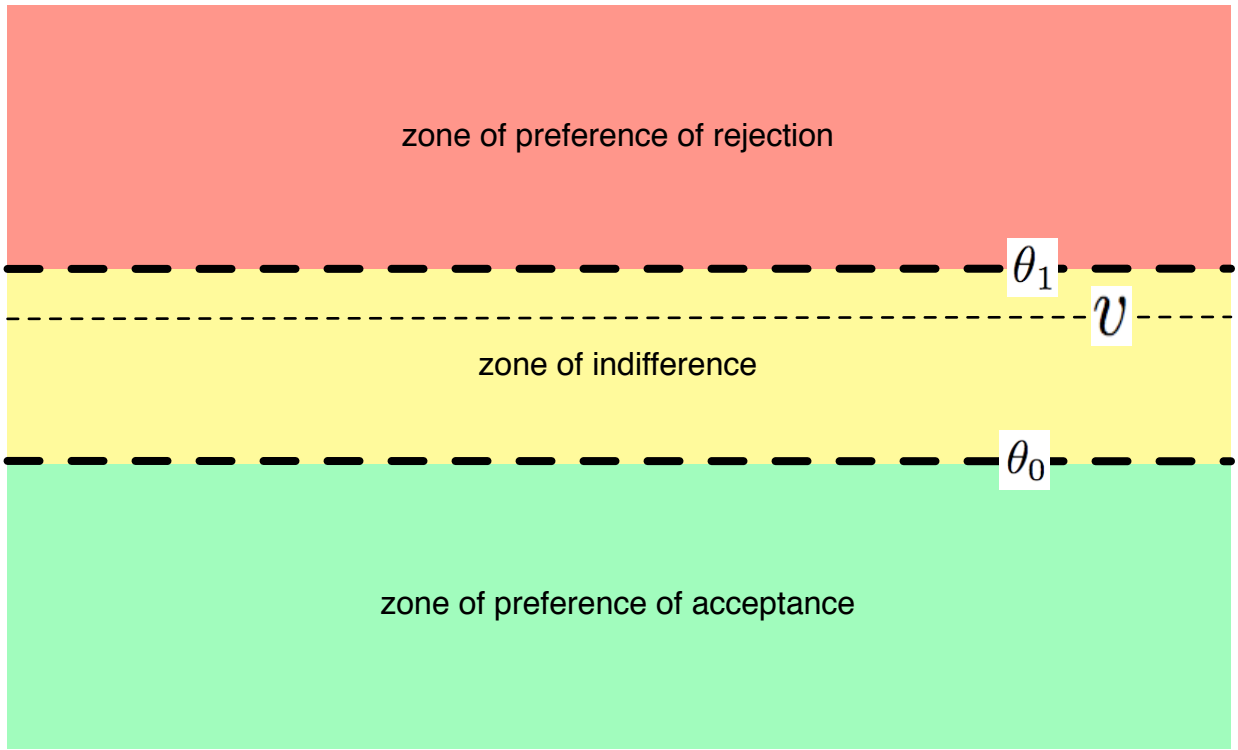

Figure 4.21. *Illustration of the various zones of the wald test.*

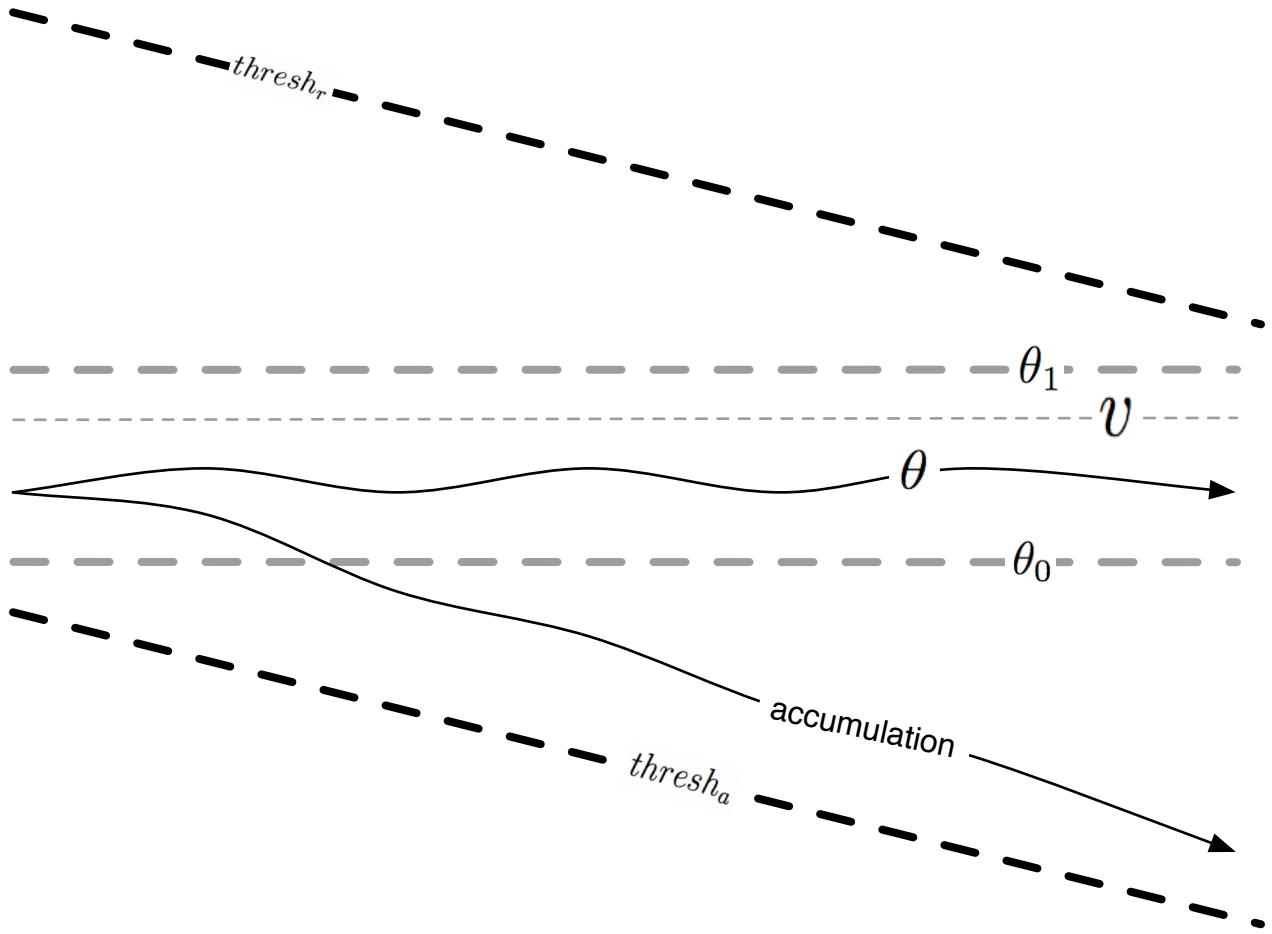

Figure 4.22. *An illustration of the sequential test procedure when testing if the mean of a normal random variable is below a certain value. In this plot* $\theta_0 + \theta_1 < 0$*, so the slope of the two threshold lines is negative. If the sum of successive samples exceeds the upper threshold the hypothesis is rejected; if the sum falls below the lower threshold the hypothesis is accepted; while the sum stays in between the two thresholds the test continues.*

each increasing sample index m. If $\theta_1 + \theta_0 = 0$ the acceptance and rejection thresholds $thresh_a$ and $thresh_r$ are simply horizontal lines with a slope of 0; on the other hand, Figure 4.22 illustrates the situation where $\theta_0 + \theta_1 < 0$ so that the thresholds decrease linearly as the sample index m increases.

We use the wald˜ external to track the sign of the acceleration signal - that is, whether $a > 0$ or $a < 0$. Ideally we would like $\theta_0 = \theta_1 = 0$, but the theory can't provide for this since the above threshold formulas have $\theta_1 - \theta_0$ in the denominator of a fraction in the intercept, so the very small value of $\theta_1 - \theta_0$ that we desire will result in thresholds very far away from the $a = 0$ line. This might not be a problem if our signal were stationary and we could collect samples forever, but unfortunately the accelerations we want to detect last only a short time.

The variable nature of our signal makes using sequential analysis problematic, since it makes no provision for data that is not stationary. Given a true signal value of θ, the average amount of samples required for our test to reach a decision, $E_\theta(n)$, is given by the following formula [44]:

$$E_\theta(n) = \frac{h_1 + L(\theta)(h_0 - h_1)}{\theta - s} \tag{4.14}$$

where h_1 and h_0 are the intercepts of the rejection and acceptance thresholds, respectively, $s = \frac{\theta_0 + \theta_1}{2}$ is the slope of the thresholds and $L(\theta)$ is the **operating characteristic** of the test, the probability that the sequential test will lead to the acceptance of the hypothesis when θ is the true signal value. This function is plotted in Figure 4.23. The application of our probability of error constraints requires that $L(\theta_1) = \beta$ and $L(\theta_0) = 1 - \alpha$. So when the true mean $\theta = \theta_0$,

$$E_{\theta_0}(n) = \frac{h_1 + (1 - \alpha)(h_0 - h_1)}{\theta_0 - s} \tag{4.15}$$

or,

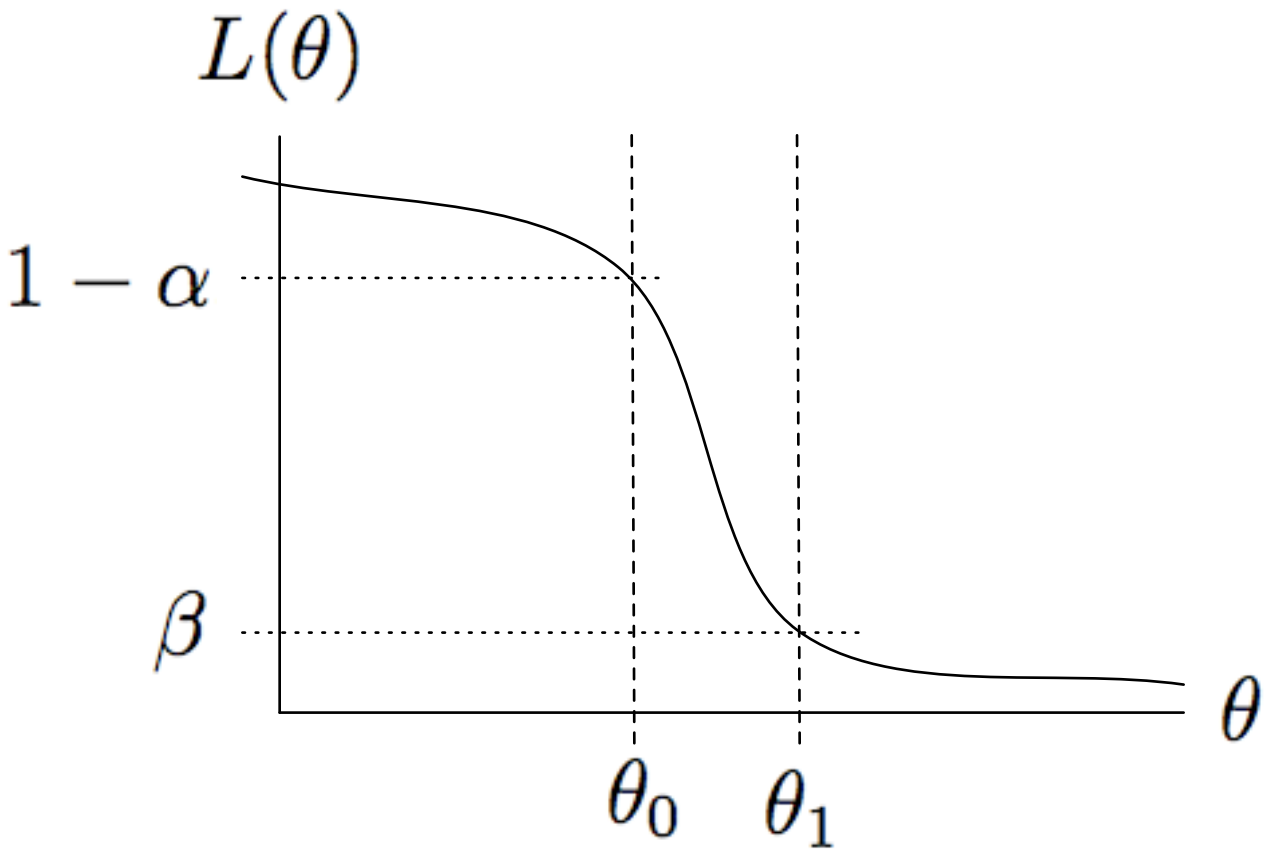

Figure 4.23. *The operating characteristic of our Wald test.*

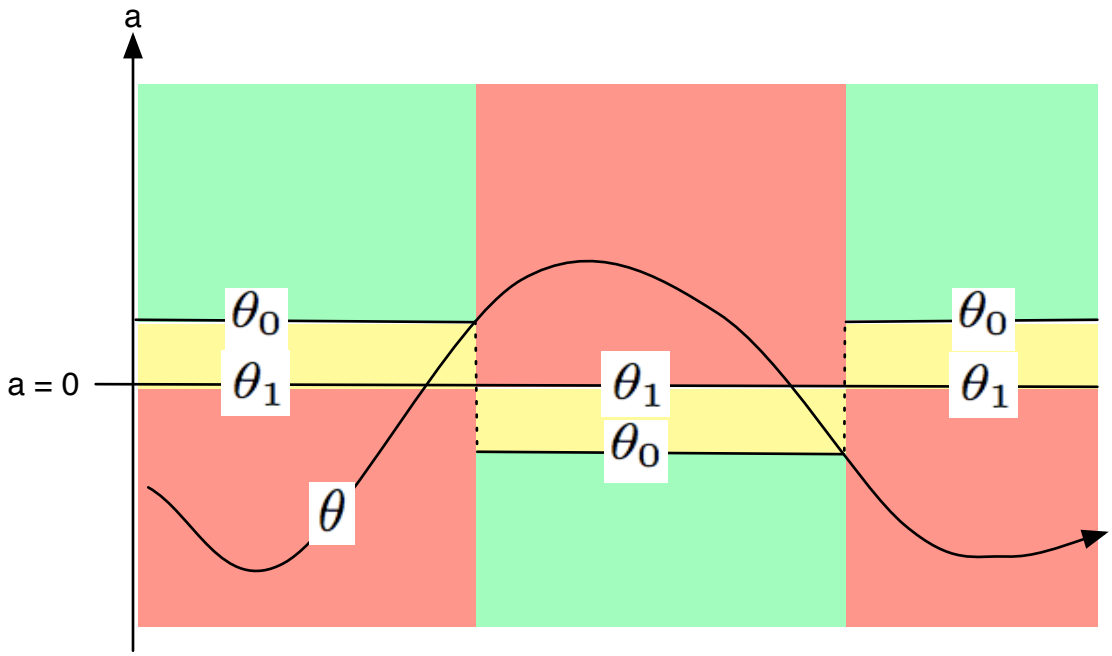

Figure 4.24. *Hysteresis involved in bidirectional acceleration tracker. When the signal transitions from positive to negative or vice versa, the θ_0 acceptance threshold flips. $\theta_1 = 0$ always.*

$$E_{\theta_0}(n) = \frac{h_1 + (1-\alpha)(h_0 - h_1)}{\theta_0 - \frac{\theta_0+\theta_1}{2}} \tag{4.16}$$

$$= \frac{\frac{\sigma^2}{\theta_1-\theta_0}\left[\log\frac{1-\beta}{\alpha} + (1-\alpha)(\log\frac{\beta}{1-\alpha} - \log\frac{1-\beta}{\alpha})\right]}{\frac{\theta_0-\theta_1}{2}} \tag{4.17}$$

4.3.4 wald˜ object implementation

To implement our acceleration sign-tracker, we're most interested in the transitions - that is, if the acceleration is currently negative we want to know when it transitions to positive. Furthermore, we strongly do not want the test to indicate that a transition from negative to positive has occurred if the true value is negative. We therefore set $\theta_1 = 0$ and offset θ_0 in the direction of the transition we want to detect as depicted in Figure 4.24; in the case of

searching for a negative to positive transition this means $\theta_0 > 0$, and when searching for a positive to negative transition this means $\theta_0 < 0$. Then solving for θ_0 we get

$$\theta_0^2 = \frac{-2\sigma^2}{E_{\theta_0}(n)}\Big[\log\frac{1-\beta}{\alpha} + (1-\alpha)(\log\frac{\beta}{1-\alpha} - \log\frac{1-\beta}{\alpha})\Big] \tag{4.18}$$

The above equation allows us to set α, β, and $E_{\theta_0}(n)$, the expected number of samples needed to terminate the test at the θ_0 acceptance point, calculate the value of σ^2 from some sample data, and from all of this, calculate the value of θ_0 needed to achieve the desired behaviour.

After fixing the desired α and β failure rates, the relationship can be expressed simply as

$$\theta_0 \propto \frac{\sigma}{\sqrt{E_{\theta_0}(n)}} \tag{4.19}$$

This formula is useful in crystallizing the three-way relationship between latency, represented by $E_{\theta_0}(n)$, noise, represented by σ, and sensitivity, represented by θ_0. With reference to Section 4.2.2, and specifically the plot of Figure 4.3, we can see that at a sampling rate of 2000 Hz, an averaging filter 5 samples long has a 6db cutoff point at about 250Hz. We would not want to go any lower than that, so setting $E_{\theta_0}(n) = 5$ is probably as high as we would want to go.

The wald~ object maintains a running sum $\sum a_m$ of the signal that's input, and updates the acceptance and rejection $thresh_a(m)$ and $thresh_r(m)$ thresholds with each sample value a_m. The implementation includes a real-time plot of the accumulation, the thresholds, and the current estimate of the sign of the signal. Figure 4.25 shows this plot in the case where a hit has been detected. The solid yellow bars are the accumulated input acceleration signal $\sum a_m$. The acceleration samples a_m, rendered as white single pixels, are most visible where the hit takes place on the right-hand side of the graph. The blue pixels above and below the yellow accumulated signal represent the rejection and acceptance thresholds. When the object is searching for a crossing from negative to positive, as it is for most of

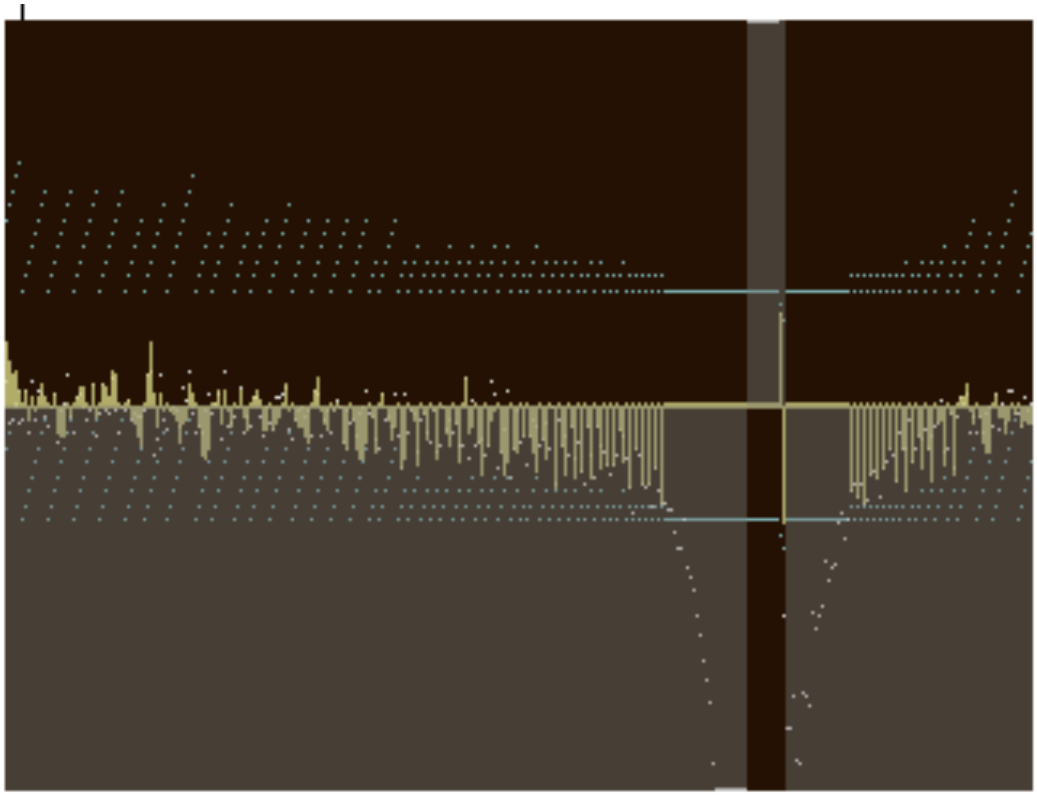

Figure 4.25. *A plot of the wald~ object data where a hit is detected.*

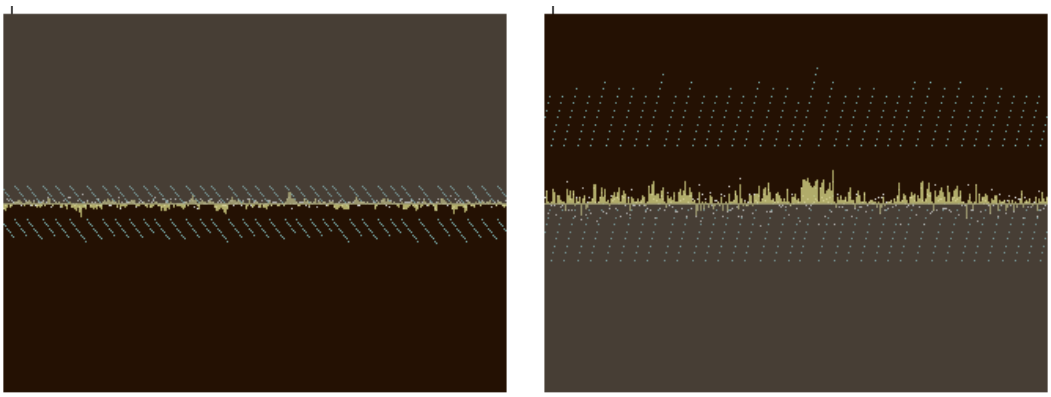

Figure 4.26. *A plot of the wald~ object data where the sticks are motionless.* $\theta_1 = 0$ *in this plot. The rejection threshold is quickly reached over and over.*

the horizontal space in this plot, the upper boundary is the threshold of acceptance and the lower is boundary of rejection. With each sample that passes, if neither threshold has been reached the position of the two thresholds shifts an amount $(\theta_0 + \theta_1)/2$. The accumulated signal will eventually fall outside the thresholds. If the threshold of rejection is exceeded, the sum is set to zero, the thresholds are reset to their initial values, and the test continues. This is what happens over and over in the left hand side of Figure 4.25.

The translucent grey foreground indicates the current estimate of the sign of the signal. This is negative on the whole left-hand side of Figure 4.25. There is a sudden downward spike in the input signal on the middle right of the graph. In the middle of this plot, the translucent grey flips to the upper half of the plot, meaning that the sum of the input signal has crossed the acceptance threshold. When this happens the estimated sign flips from negative to positive. Since the object is now looking for a transition from positive to negative, the threshold of rejection becomes the upper blue line and the threshold of acceptance the lower blue line. The sign estimate quickly flips back to negative after the stick has finished its rapid reversal of direction and begins the deceleration of its upwards arc.

Figure 4.26 shows plots for both the left and right sticks. The wald~ object is using the input acceleration samples a_m to track the sign of the acceleration signal. On the left hand side the wald~ object is in an $a_m > 0$ state, whereas on the right hand side it is in

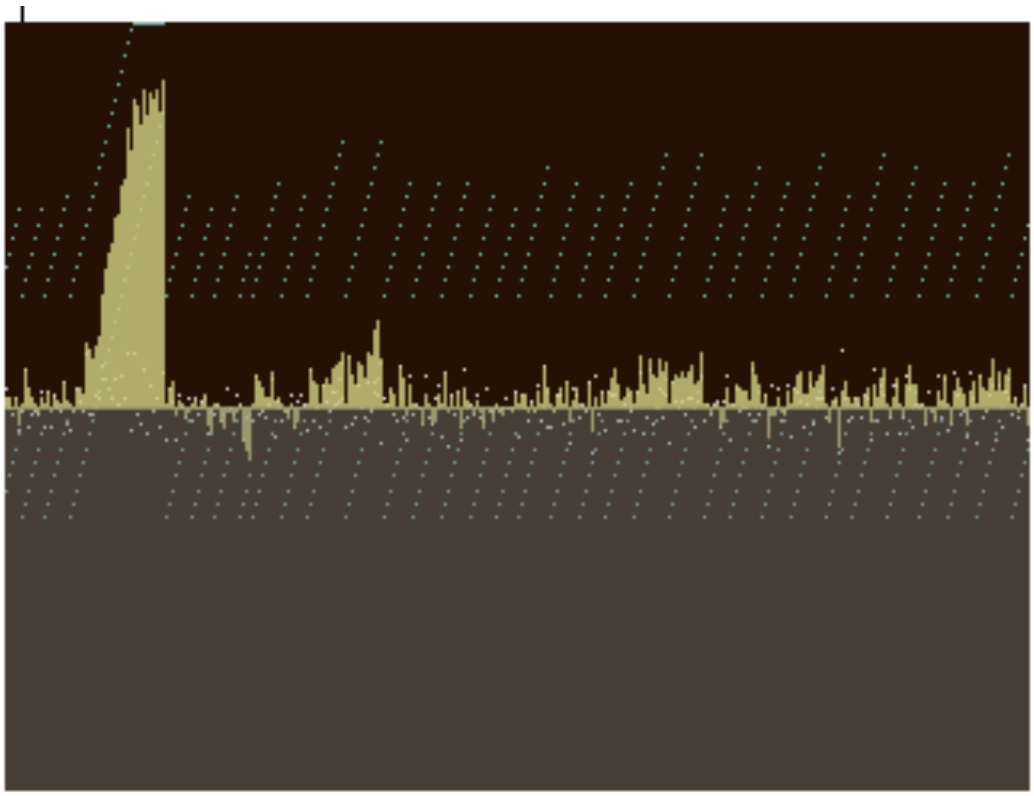

Figure 4.27. *A plot of the wald~ object data where a hit is missed. The large upwards acceleration in the left part of the plot is missed because the incoming samples a_m are about the same magnitude as the slope of the acceptance threshold $thresh_a(m)$. With $\theta_1 = 0$, this slope is a good measure of the **sensitivity** of the interface.*

an $a_m < 0$ state. Note that in the left hand plot the acceptance and rejection thresholds slope downwards, whereas in the right hand plot the thresholds slope upwards. The sticks were not moving during the formation of these plots, so the rejection threshold is quickly reached over and over, and the test re-started. The two plots are scaled identically with $a = 0$ in the vertical center of the plot. Note that the larger amount of noise in the right hand graph causes the thresholds to be farther from the $a = 0$ line and have a more steep slope.

Figure 4.27 is an example of the wald~ detector **missing** a hit. The large upwards spike on the far left is growing at about the same rate as the threshold is sloping upwards. If the

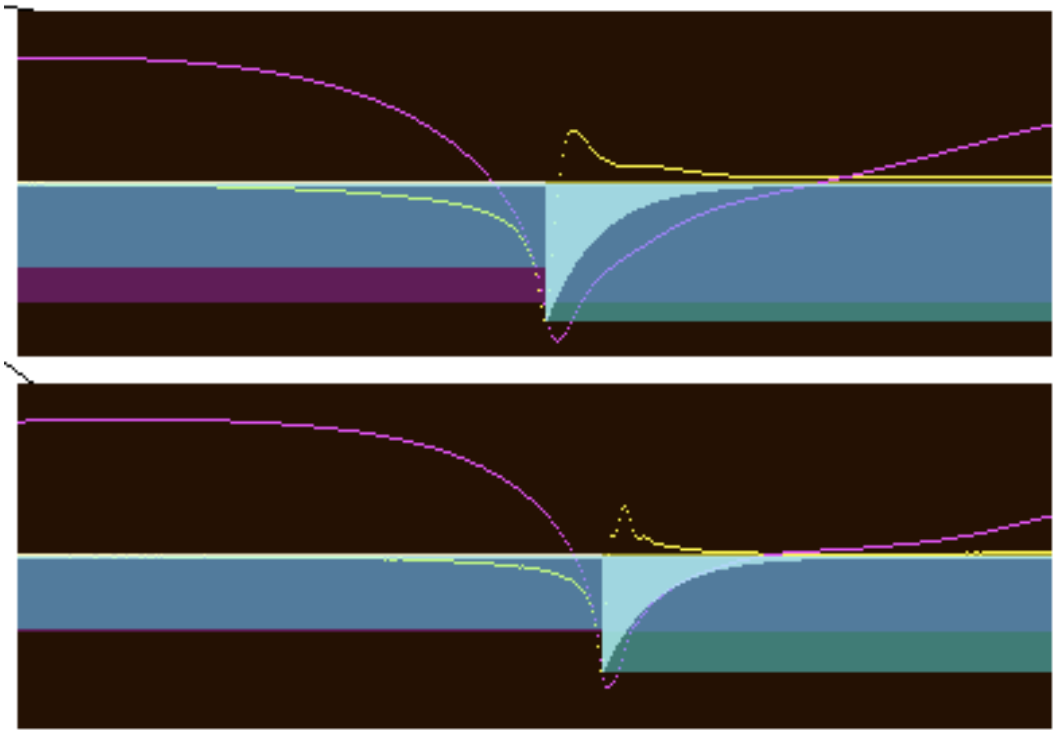

Figure 4.28. *A plot of the z and velocity estimates. In this figure the pad has been hit by the two sticks at roughly the same time. Note that the two plots are not synchronized.*

input signal were a little larger the sum would grow a little faster and this spike might have exceeded the threshold. What happens instead is that eventually the growth of the spike stops as the stick stops accelerating upwards, and the lower rejection threshold eventually surpasses the input sum. This shows how the slope of the thresholds, $(\theta_0 + \theta_1)/2 = \theta_0/2$, can be thought of as the **sensitivity** of the detector.

Using our assumption of gaussian noise, if σ_z^2 is the variance in the z estimate, and σ_v^2 and σ_a^2 are the variances in the velocity and acceleration of the z estimate, respectively, then $\sigma_v^2 = 2\sigma_z^2$ and $\sigma_a^2 = 4\sigma_z^2$. We would expect then a better rate of success at detecting the minimum of the z estimate rather than the minimum of the v estimate, but detecting an event using this minimum of velocity has the advantage that a hit is detected *before* the minimum of the z estimate is reached. This can be seen in Figure 4.28, which is a snapshot of the velocity-graph implemented as a part of the Audio-Input Drum patch. Note that the

white decaying spikes, which represent detected hits, come at the minimum of the yellow velocity estimate, and **before** the purple z estimate line reaches its minimum. Using the maximum negative velocity as the trigger point makes sense from a physical standpoint as well: when a drum stick comes into contact with the surface of the drum, the restoring force of the drum surface immediately begins to accelerate the stick upwards. The striking of a drum head is an **excitation gesture** [30]; resulting synthesis should begin as soon as the drum head and stick are in contact.

4.3.5 The full hit detection algorithm

Figure 4.29 is a flow chart of the operations in the hit detection algorithm. A change from negative to positive acceleration is not the only characteristic of a hit. We must also ensure that the stick is actually on the surface of the drum and not simply waving around in the air. We implement this positional test simply by comparing the z estimate with a threshold value that represents the value of the z estimate when the stick is at the surface of the drum. The z estimate at the time of a potential hit must be below this threshold. This z threshold is measured during a step in the calibration procedure where the sticks remain motionless on the surface of the drum. During this procedure the average z estimate is recorded. Since the z estimate is bowl-shaped over the area of the drum surface that we may want to use, a parameter of the patch allows the user to shift this threshold upwards or downwards, scaled by the estimated standard deviation of this z estimate signal. A negative value for this parameter allows more of the edges of the drum surface to be involved in hit detection, but also creates the possibility of hits being detected above the surface of the drum.

Let's imagine that a stick is sitting motionless on the surface of the drum. Let's also imagine that this stick is well below the z-threshold, as might be the case in the current implementation if the stick is sitting in the center of the drum due to the bowl-like curvature of the z estimate over the surface of the drum. Let's assume then that the z threshold test will return a true value, so the only thing preventing the hit-detection algorithm from a false alarm is the wald~ external's detection of a transition from negative to positive acceleration.

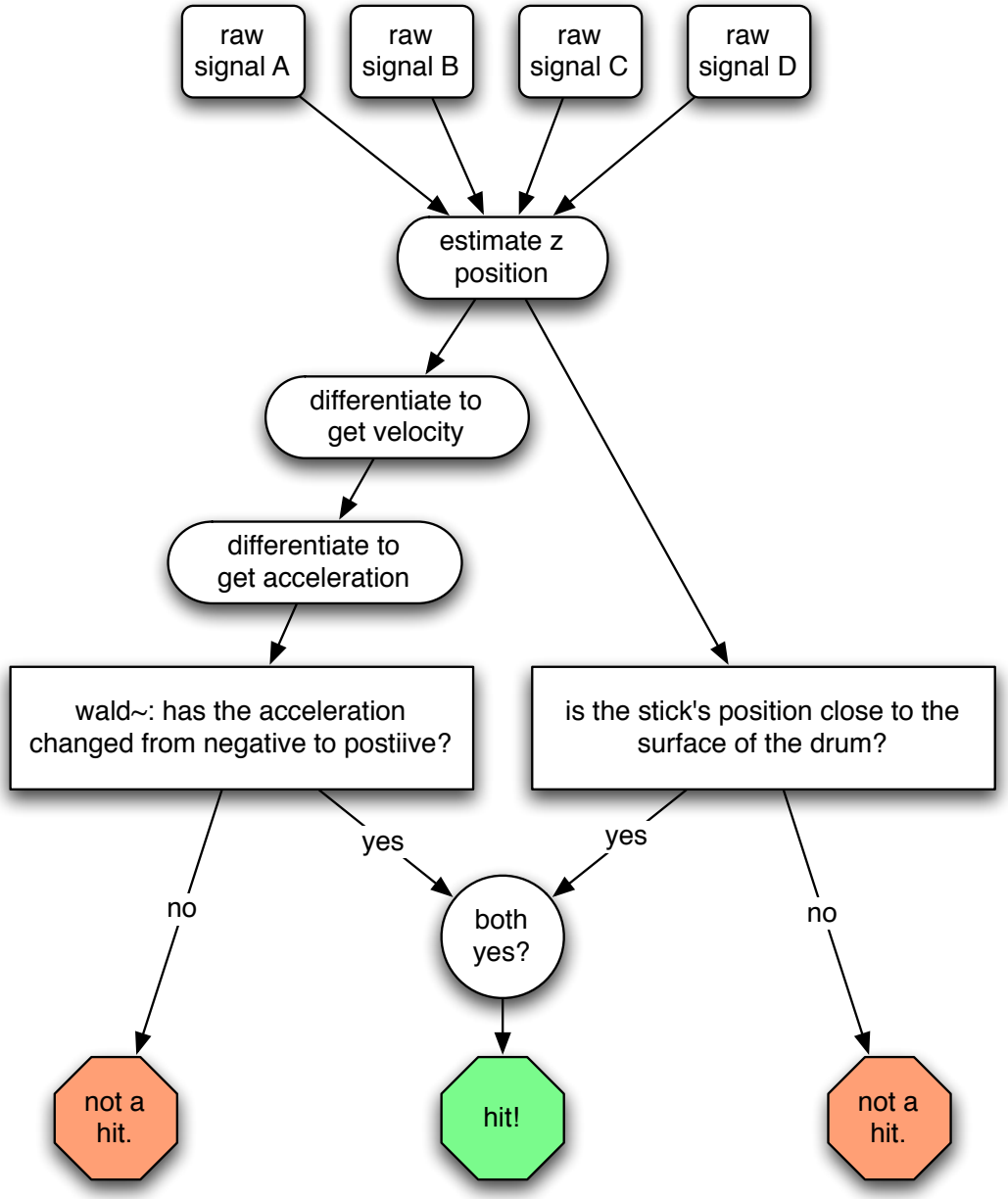

Figure 4.29. *Logical flow chart of operations in the hit detection algorithm.*

In this situation the signal really is stationary, so the sequential detection theory behind the wald~ external is valid without caveat, and our false alarm rate of β will determine the rate of false alarms in the hit detection algorithm.

If a hit is detected, the velocity of the hit is estimated to be the maximum negative value that the velocity estimate reached in the time since the last hit. Because the signal is so much stronger close to the surface of the drum, this maximum negative velocity is effectively the minimum of the signal in the few milliseconds leading up to the detection of the hit. Using the maximum negative velocity as the estimate of the force of a hit intuitively makes more sense than using the maximum negative position, which can vary depending on the physical characteristics of the foam that is being hit.

A calibration procedure was designed to quickly set all the parameters for both the scaling of the x, y and z estimates and to calculate statistical parameters to be used in the hit detectors. The procedure and its implementation code are detailed in Appendix E.

4.4 Responding to Signal Triggers in Max/MSP

We now should consider the output of a signal-processing algorithm that has been devised to detect an event from an incoming set of signals. It can help conceptually to abstract the event-detection algorithm into its own Max object box, as is shown in Figure 4.30; here we see the three possibilities of an algorithm that produces event data, signal data, or some combination of the two.

Making an educated choice for output data format requires a solid understanding of the advantages and disadvantages of signal and event-based processing in the Max/MSP/Jitter environment; details about the Max/MSP/Jitter runtime architecture can be found in Appendix A. The choice of signal or message may be influenced by the expected frequency of the event to be detected, and the desired response. If the event is expected to occur infrequently and the programmed response is computationally expensive, event-based output will likely be the more efficient choice. However, if response to one or more of the

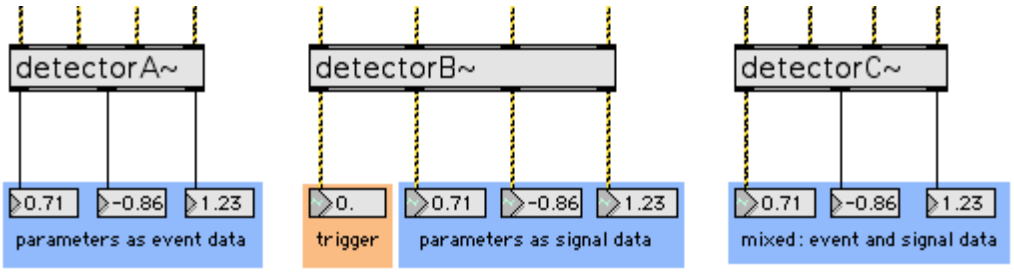

Figure 4.30. *Possible output from a signal-based event-detection algortihm: output parameters as an event, as multiple signals, or as some combination of event and signal data.*

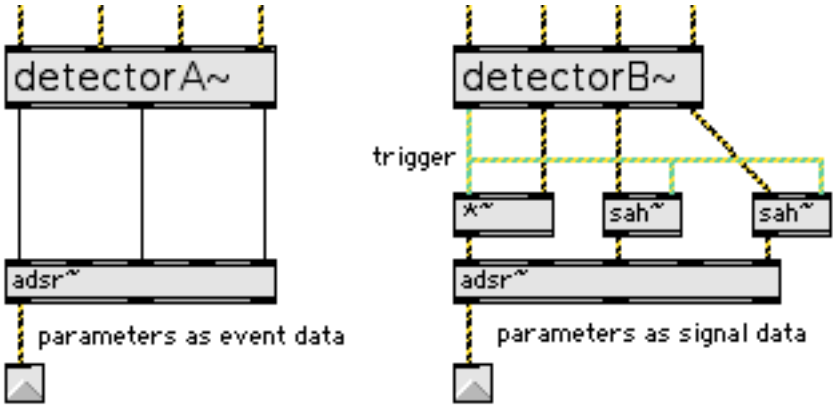

Figure 4.31. *Triggering the adsr~ object with event or signal data.*

parameters requires sub-vector sample-accuracy, signals will be required.

To illustrate the kind of inefficiency introduced by processing temporally local data in the signal domain, consider the example of Figure 4.31, which shows two different ways to trigger the adsr~ object, an MSP generator written by the author of this thesis. Each of the adsr~ object's inputs will accept signal or event data. In the construction on the left, the input to the object is entirely in the event domain, whereas in the right-hand construction a functionally identical network is replicated in the signal domain.

In the event-domain scheme, each gestural event results in a single set of data being output. The adsr~ object will generate an ADSR envelope after this data set has been received. In the signal-domain scheme, a trigger signal is necessary to define the moment in

time at which the event has been detected. The trigger signal will be zero at all times when a new event has not been detected, and 1 for the single sample at the time when an event is thought to have occurred. Figure 4.31 illustrates two ways in which the trigger signal can be used to propagate the type of static signal values that are suitable for event-based calculations. The leftmost inlet of the adsr~ object is that which controls the amplitude of the generated envelope. The trigger signal from the detectorB~ abstraction is multiplied by its second output, and the resulting signal sent into the adsr~ object's first inlet. This signal will be zero at all times when a new event has not been detected, and will be the value of the signal coming from the second outlet when a new event is thought to have occurred. The adsr~ object will respond by generating an ADSR envelope beginning at the same sample frame in which it received the non-zero signal value.

The other two signal parameters are sent into the adsr~ object with the sah~ (sample and hold) object. This object uses the trigger signal to determine when to *s*ample the incoming parameter signal value. Its output signal will remain at this sampled value until it is triggered to sample the incoming signal again. Using the sah~ object allows us to freeze a signal parameter at a certain value in between events.

The advantage of the signal-based method is that the ADSR envelope will be triggered with sample-accuracy - that is, the envelope will begin in the same sample frame that the event was detected. In the event-based model, on the other hand, the triggering message will not be sent to the synthesizing objects until the next division between audio vectors. This concept is illustrated in Figure 4.32. The individual blocks in the figure represent single samples; these are grouped in long, rectangular audio vectors. With a signal-based trigger, the output synthesis begins in the same sample that the event was detected. With a message-based trigger a message is placed in the queue when the event is detected. When the message queue is next serviced this message will be sent to the synthesizing objects. If the **scheduler in audio interrupt** option is enabled, a message-based trigger will begin synthesis before the next audio vector begins calculating. Since the trigger can occur anywhere within an audio vector, but the synthesis won't be realized until the start of the next

vector, there will therefore be a random latency added to the system, and the distribution of this latency is uniform across the size of the audio vector. If the software is operating with a small vector size, this potential jitter in the response time may be acceptable. At 64000 Hz with a vector size of 64 the maximum latency introduced in this random process is 1 millisecond. If the scheduler in audio interrupt option is not enabled, the event-processing is services in a separate thread and there's no guarantee that the synthesis will begin at the start of the next vector, so jitter could be worse.

The drawback to using signal-based triggers is computational efficiency. Given an event that occurs on average one time per second and a sampling rate of 64000 Hz, 63999 false comparisons will be made for every object that is monitoring the event trigger signal. If parameters must be frozen as in Figure 4.31, the sah~ objects will also waste many cycles waiting for the command to re-sample their inputs.

Figure 4.33 shows another example of a message-based network and its signal-based equivalent. To keep the response to an event in the signal domain, all residual calculations must be done without resorting to messages. In this example a simple random choice between two values is being made. As in Figure 4.32, many more cycles will be used by the signal-based network. Numeric processes such as that illustrated in Figure 4.33 will usually have a signal-based translation, but for many complex procedures it's not possible to replicate a message-based network in the signal domain without resorting to creating your own custom C or Java external.

4.4.1 The event~ object

Even if one is satisfied with a message-based synthesis trigger, one still has to deal with transferring parameter values from the signal to the message domain. Figure 4.34 shows one way to accomplish this transfer: when a trigger is received the sah~ objects sample the value of their parameters. The next time the scheduler is serviced the edge~ object sends out a bang and the snapshot~ objects send out the sampled parameter values as messages. Although this technique works in the vast majority of cases, it has a serious problem if two

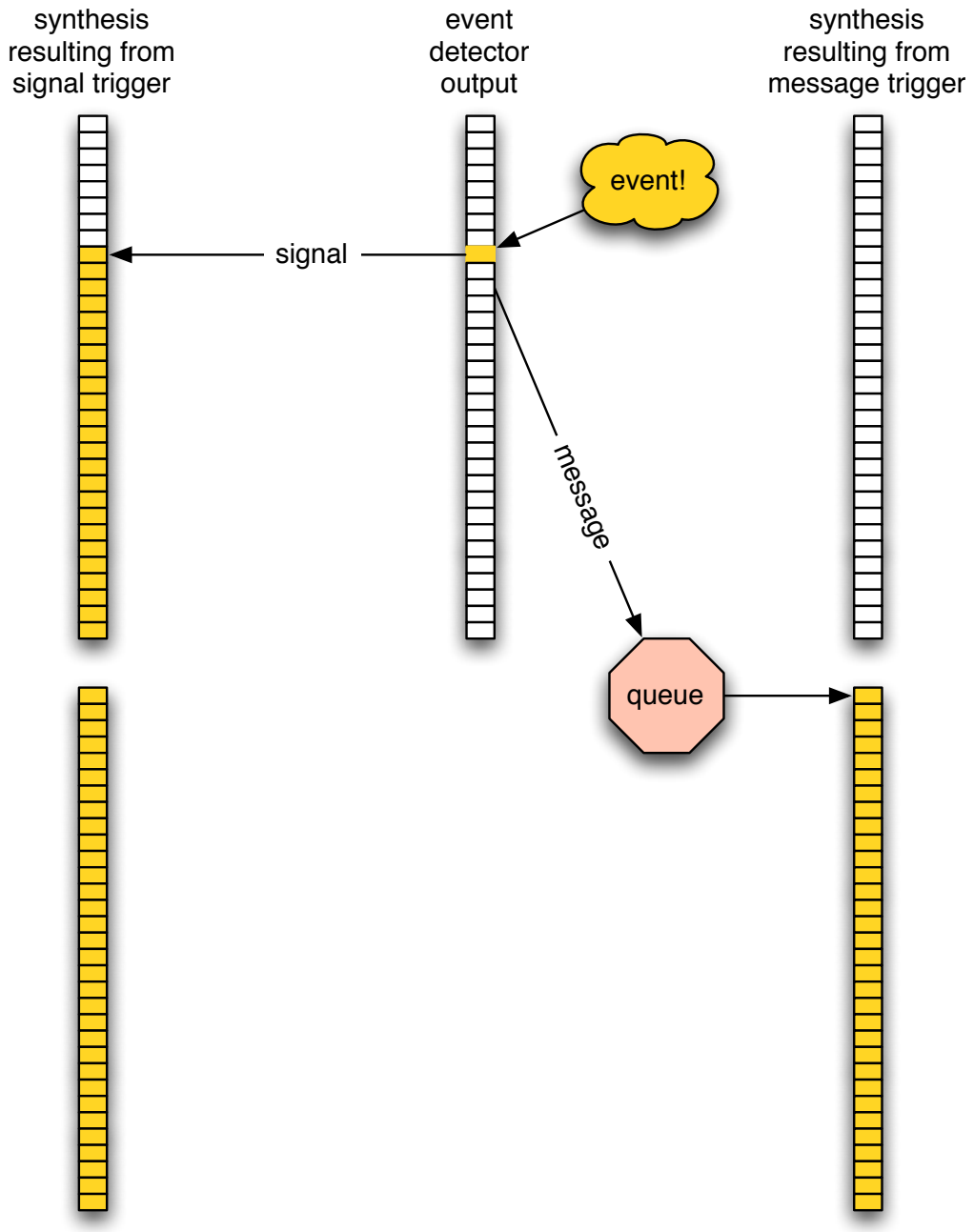

Figure 4.32. *The difference between triggering a synthesis event with a signal and a message. The individual blocks are single samples, vertically grouped in audio vectors. With a signal-based trigger, the output synthesis begins synchronously with the event detection. With a message-based trigger a message is placed in the queue, and output synthesis begins when the queue is next serviced.*

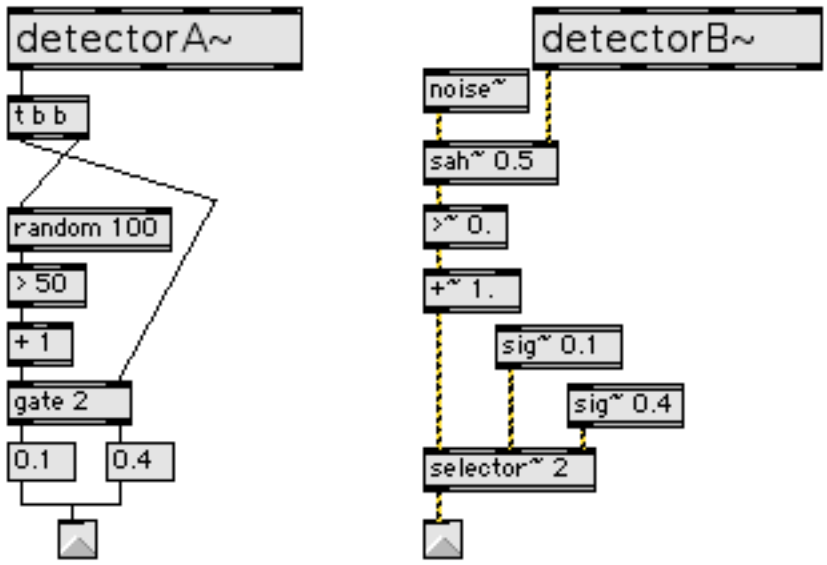

Figure 4.33. *Making a random decision in the event and signal domains.*

or more events are triggered within a single audio vector. In this case the edge~ object will send out a single bang, and the sah~ objects will contain the parameter values for the last event; any events detected prior to the last event will be lost.

To solve this problem, the event~ object was created. The code for this object is listed in Appendix F. Figure 4.35 recreates the object network of Figure 4.34. The first argument to the event~ object sets the number of parameter inlets to create and sample. When it receives a trigger the current values of the parameters are sampled and placed in an output queue. When the scheduler is next serviced, all events in the queue are output as lists in the order they were recorded.

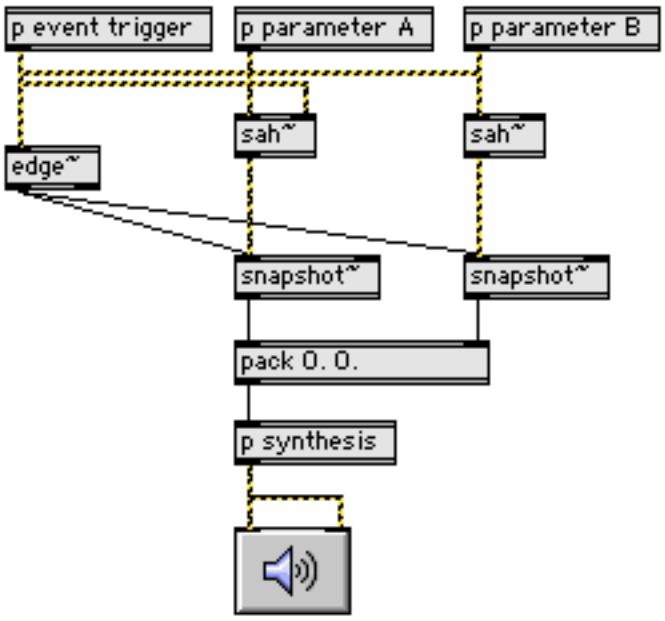

Figure 4.34. *Transcoding from the signal domain to the message domain. When a trigger is received the sah~ objects sample the value of their parameters. The next time the scheduler is serviced the edge~ object sends out a bang and the snapshot~ objects send out the sampled parameter values as messages.*

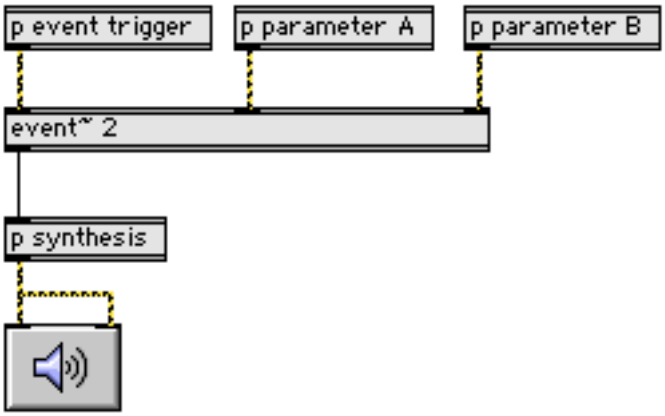

Figure 4.35. *The event~ object. This graph is very nearly functionally equivalent to Figure 4.34, except that the event~ object can deal with multiple events in a single audio vector, whereas the sah~-based solution of Figure 4.34 will only output the last event in a vector.*

Chapter 5

Future Audio-Input Gesture Work

Chapters 3 and 4 detailed various ways in which the work presented in this thesis has improved the Radio Drum and Radio Baton interface: the interface is more sensitive, more responsive, and more reliable. But perhaps the most important contribution from this work is how it creates opportunity for further development. Liberated from the constraints of custom hardware, the barrier to entry for gesture analysis will continue to drop as audio-input hardware and powerful personal computers continue to become less expensive. The analysis techniques implemented to mimic the behaviour of the Radio Drum are relatively simplistic; some of the numerous opportunities for further advancement of the interface are discussed below.

5.1 Signal-to-noise

The potential signal-to-noise ratio of the interface can be improved either by increasing the amount of received signal or reducing the amount of noise in the signal. The former could be accomplished by amplifying the carrier signals. Currently the output of the audio interface is sent directly to the transmitter at the end of the drum stick. An amplification of this signal by a factor of a would result directly in an equivalent improvement of a factor of a in the signal-to-noise ratio.

Some thought would be required to decide how much voltage was safe to send to the end of the rapidly moving drum sticks. It might be safer to reverse the signal flow in the audio

loop so that what are currently the antennae become the transmitters and the sticks become the antennae. The modular design of the current patch makes the software engineering required for this kind of fundamental change minor. With the current apparatus, rather than have two carrier waves there would have to be four, and instead of four incoming signals each being split with two bandpass filters there would be two incoming signals each split with four bandpass filters.

The other method of improving the signal-to-noise ratio, reducing the noise, involves improving the filters that operate on the raw input data. The filters currently being used to demodulate the input signals are basic biquadratic filters designed in conformance to the RBJ cookbook of audio filter design [45]. It is clear from Figure 3.17 that the 30kHz filter is providing more attenuation than the 26kHz filter. Accordingly, the post-demodulation noise of the input channels using the 26kHz carrier signal are worse than those using the 30kHz carrier wave. After the calibration procedure that measures this noise and sets the hit-detecting thresholds accordingly, the effect of the difference in noise between the sticks is that the 30kHz stick is more sensitive than the 26kHz stick. Since it is desirable for both sticks to have the same sensitivity, either the two filters should be designed with similar attenuation characteristics, or the event-detection thresholds of the better-performing stick have to be limited to the performance of the worse-performing stick. Designing filters with prescribed specifications can be accomplished through various iterative algorithms [35]. Another possibility would be to use the same carrier frequency for both sticks, but to have the two carriers be 90 degrees out-of-phase. This was experimented with, but was phased out in order to keep the design as simple as possible in this first iteration.

A large potential improvement in the signal-to-noise ratio might be realized by implementing an estimator based on a Kalman filter [46]. The Kalman filter is an adaptive noise-removing system that has had great success in navigation applications, among others. The design of a Kalman filter would use the cross-correlations between antennae to more efficiently separate signal from noise. Furthermore, the three dimensional estimates would be made as a vector rather than independently - for example, the estimate of z and y could

inform the estimate of x.

5.2 Antennae Geometry and Positional Estimation

The flat surface of the current antenna is excellent for detecting changes in vertical direction in close proximity to its surface, but the complex shape of the electric field that results from a finite rectangular plane makes three-dimensional position estimation difficult. I propose experimenting with a new antenna geometry based on **transmitting balls**. The carrier signals from the audio interface could be amplified and sent to several small metallic spheres. The spheres could be arranged in any position, and the sticks act as antennae. The mathematics of positional estimation is simplified significantly by the spherical symmetry of the transmitters. The flexible positioning of the spheres would allow spheres to be placed in locations of interest - for instance, underneath each surface in a traditional trap drum kit - while still allowing reasonable three-dimensional positional estimation. Software would have to be cognizant of the positions of the transmitting spheres and flexible enough to accommodate a variable number of spheres.

Any accurate positional estimator would have to deal with the fact that the two sticks interact with one another electrically. It is clear from using this interface that there is a strong correlation between the proximity of the two sticks and the induced signals. The field theory behind their interaction needs further exploration, and if the effect can be modeled some multi-dimensional estimator could perhaps deal with the six-dimensional problem (three for each stick) as a single state.

5.3 Visual feedback

Theory is all well and good, but when you're ready to step on stage it's the small details that count. Practical performance considerations and further pushing of the envelope in terms of visualization technology and interface design are important areas to explore to truly push

the boundaries of the interface as a musical instrument. One fundamental difficulty that still exists when setting up a complicated performance patch with the Audio-Input Drum is that there is no visual feedback system on the drum itself. To interact visually with the program, even just to confirm that it is working, requires looking at a computer screen. It would be exciting to use a video projector to project directly onto the surface of the drum. This would allow for visual interfaces that adapt in real time to the position of the sticks. For instance, imagine a performance patch that interacted with other musicians, allowing the user of the Audio-Input Drum to sample in real-time audio from the other performers, and assign the sample to a physical location on the drum surface. A projection system could render a waveform of the sampled audio, or some other representation of the sound, onto the desired location of the drum, giving the performer a visual cue to know where she should hit the drum to play back the sound. With the spherical antenna geometry discussed above it might even be possible to rear-project onto and strike a thin but taut elastic membrane.

5.4 New Modes

Providing an extended set of tools to facilitate the creation of new interactive pieces is another important area ripe for exploration. Having access to the raw signals of the gesture affords us the opportunity to create new modes of interaction beyond the simple duality of the ”whack” and ”continuous” modes offered by the Radio Drum.

First, there is the opportunity to simplify what might be called **compound modes** made up of various stages of whack and continuous reporting. For instance, imagine a mode that might be called ”whack and hold”: after the surface is hit, the mode could report continuous data until a vertical threshold were reached. One potential mapping of this mode would be as a synthesizer controller: the initial whack triggers a note on, and the vertical position controls the amplitude of the note until the upper vertical threshold is exceeded, at which point a note off is sent.

Beyond compound modes, algorithms could be developed to detect entirely new ges-

tures. For instance, a slight horizontal ”flick” of the wrist might be a gesture that conveniently augments regular whack and continuous gestures. A flick would have a three-dimensional position, an amplitude, and possibly a two-dimensional orientation to describe the direction of the flick. If the gesture can be defined parametrically, an algorithm can be developed to track it. Now that the analysis is software-based, it will be exciting to see what kinds of gestures people find valuable to detect and parametrize.

5.5 Non-artistic applications

The strict demands of the performance paradigm make the musical instrument an outstanding application for refinement of a gesture-capture system. Now that an acceptable system has been developed, it could be valuable to work towards using the technology for other applications, perhaps with broader purposes. For instance, rather than forcing a disabled person to adapt to the interfaces available, an open-ended and precise system such as this could be trained to track gestures suitable to the individual.

Bibliography

[1] R. Boie, L. W. Ruedisueli, and E. R. Wagner, "Gesture Sensing via Capacitive Moments," internal report for AT&T Bell Labs.

[2] M. Mathews, "Three dimensional baton and gesture sensor," Patent 4 980 519.

[3] V. Lennard, *MIDI Survival Guide*. San Jose: PC Publishing, 1993.

[4] J. Rona, *The MIDI Companion*. New York: Hal Leonard Corp, 1992.

[5] [Online]. Available: http://www.cycling74.com

[6] [Online]. Available: http://en.wikipedia.org/wiki/L%C3%A9on_Theremin

[7] [Online]. Available: http://en.wikipedia.org/wiki/Rhythmicon

[8] [Online]. Available: http://en.wikipedia.org/wiki/Theremin

[9] M. V. Mathews and C. Abbott, "The Sequential Drum," *Computer Music Journal 4(4)*, 1980.

[10] M. Mathews, "The Conductor Program and the Mechanical Baton," 1989.

[11] M. V. Mathews, "The Radio Baton and the Conductor Program, or: Pitch, the Most Important and Least Expressive Part of Music," *Computer Music Journal 15(4)*, 1991.

[12] [Online]. Available: http://en.wikipedia.org/wiki/Piezoelectric

[13] A. R. Tindale, A. Kapur, G. Tzanetakis, P. Driessen, and A. Schloss, "A Comparison of Sensor Strategies for Capturing Percussive Gestures," *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression (NIME05), Vancouver, BC, Canada*, 2005.

[14] [Online]. Available: http://www.soundonsound.com/sos/1994_articles/nov94/korgwavedrum.html

[15] E. Fléty, "3D Gesture Acquisition Using Ultrasonic Sensors," *Trends in Gestural Control of Music*, 2000.

[16] R. Auzet, "Gesture-following Devices for Percussionists," *Trends in Gestural Control of Music*, 2000.

[17] [Online]. Available: http://www.buchla.com/mlumina/index.html

[18] R. Boie, M. Mathews, and W. A. Schloss, "The Radio Drum as a Synthesizer Controller," *Proceedings of the International Computer Music Conference*, 1989.

[19] R. Boulanger and M. Mathews, "The 1997 Mathews Radio-Baton and Improvisation Modes," *Proceedings of the International Computer Music Conference*, 1997.

[20] D. Jaffe and A. Schloss, "The Computer-Extended Ensemble," *Computer Music Journal 18:2*, 1994.

[21] ——, "Wildlife," *The Virtuoso in the Computer Age*, 1994, an interactive duo for Mathew/Boie Radio Drum and Zeta Violin.

[22] ——, "A Virtual Piano Concerto The coupling of the Mathews/Boie Radio Drum and Yamaha Disklavier Grand Piano in 'The Seven Wonders of the Ancient World'," *Proceedings of the International Computer Music Conference*, 1994.

[23] R. Jones and A. Schloss, "UNI," *Iota Center*, 2000, interactive music and video performance.

[24] A. Schloss and H. Durán, "Blind Data," *DVD of the 2005 NIME conference, Vancouver, BC*, 2006, recording of a live performance.

[25] R. Boulanger, "Radio Sonata," 1990, for Radio Baton, Conductor Program and Yamaha TG77.

[26] ——, "GrainStorm," 1993, for Radio Baton, Conductor Program and Yamaha TG77. Features real-time video sequencing by Greg Thompson.

[27] ——, "StarDust," 2003, for Radio Baton, Video Baton and Max/MSP/Jitter.

[28] [Online]. Available: http://www.csounds.com/boulanger/

[29] R. Bresin, K. F. Hansen, and S. Dahl, "The Radio Baton as configurable musical instrument and controller," *Proceedings of the Stockholm Music Acoustics Conference*, 2003.

[30] C. Cadoz and M. M. Wanderley, "Gesture - Music," *Trends in Gestural Control of Music*, 2000.

[31] A. S. Tanenbaum, *Modern Operating Systems, 2nd Ed.* Upper Saddle River, New Jersey 07458: Prentice-Hall, Inc., 2001.

[32] T. M. Nakra, "Searching for Meaning in Gestural Data," *Trends in Gestural Control of Music*, 2000.

[33] R. Avizienis, A. Freed, T. Suzuki, and D. Wessel, "Scalable Connectivity Processor for Computer Music Performance Systems," 2000.

[34] J. T. Allison and T. A. Place, "Teabox: A Sensor Data Interface System," *Proceedings of the 2004 International Computer Music Conference*, 2004.

[35] A. Antoniou, *Digital Filters: Analysis, Design and Applications, 2nd ed.* Dubuque, Iowa: McGraw Hill, 2000.

[36] S. Haykin, *Communication Systems, 3rd ed.* United States: John Wiley and Sons, Inc., 1994.

[37] [Online]. Available: http://www.rme-audio.com/

[38] N. MacRae, *John von Neumann*. Toronto: Random House, 1992.

[39] D. Zicarelli, "Communicating with Meaningless Numbers," *Computer Music Journal*, vol. 15, no. 4, pp. 74–77, 1991.

[40] M. Schwarz and L. Shaw, *Signal Processing: Discrete Spectral Analysis, Detection and Estimation*. Dubuque, Iowa: McGraw Hill, 1975.

[41] D. J. Griffiths, *Introduction to Electrodynamics, 2nd ed.* Englewood Cliffs, New Jersey 07458: Prentice-Hall, Inc., 1989.

[42] R. T. Hodgson, "The problem of being a normal deviate," 1979.

[43] F. R. Moore, "The Dysfunctions of MIDI," *Computer Music Journal*, 1988.

[44] A. Wald, *Sequential Analysis*. New York: John Wiley & Sons, Inc., 1947.

[45] [Online]. Available: http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt

[46] P. Y. H. Robert Grover Brown, *Introduction to Random Signals and Applied Kalman Filtering*. New York: John Wiley & Sons, Inc., 1992.

[47] M. Puckette, "The Patcher," *Proceedings, International Computer Music Conference*, pp. 420–429, 1988.

[48] D. Zicarelli, "An Extensible Real-Time Signal Processing Environment for Max," *Proceedings of the 1998 International Computer Music Conference*, pp. 463–466, 1998.

Appendix A

The Architecture of Max/MSP/Jitter

Object-oriented programming encourages a mode of thinking that segments a programming problem into smaller, localized pieces that together combine to solve a problem. The importance of modularity is stressed; rather than operating as a collection of functions, a program written in an object-oriented language may be seen as a collection of cooperating **objects**, each of which is capable of sending and receiving messages, processing data, and sending messages to other objects.

Some object-oriented languages carry a performance penalty when compared to a native piece of code that performs the same function. Still, for some applications the tradeoff of having a stable environment built on higher-level, independent pieces of code is worth the loss in efficiency. Nowhere is this more evident than with the latest generation of modular audio processing tools, such as Cycling '74's Max/MSP/Jitter[47, 48]. The visual programming environment of Max aggregates the ability to build component objects in several different high-level languages - C, Java, JavaScript, GLSL, and Cg are officially supported, and extensions for many other languages have been contributed by independent developers.

The graphical nature of Max is a stumbler for many programmers used to scripted, text-based languages. Programs, which start out as empty, blank pages, are known as *patches*. In Max/MSP/Jitter objects are represented by boxes that can be placed anywhere on the screen. The boxes have inlets, which accept data, and outlets, which send data. Data can be in the form of messages (zero-dimensional), signal vectors (one-dimensional), or matrices,

which could probably more properly be called tensors (many-dimensional). This section gives a description of each of the data types, and then describes the threading model in which execution occurs.

A.1 Objects

The basic building block of Max programming is called an *object*. Like other object-priented programming languages, Max is made up of many *classes* of objects, each of which has a unique behaviour. Different objects can be used together in different ways to accomplish the desired programming task. At the time of writing, objects for the Max environment can be programmed in C, Java, or JavaScript. Jitter users can also program shader components in GLSL and Cg. Third-party extensions provide support for PERL, Python, and some other less well-known scripted languages.

For example, have a look at Figure A.2. What you might call a program in another language is called a *patch* in Max: this simple patch contains three objects, each of which is visually represented by a grey box. The class of a Max object is defined by the text contained within its box. The *notein* class is one of several classes designed to receive MIDI input from the outside world. Max is able to receive MIDI input through a standard MIDI interface, the kind that one could use with a commercial sequencing package such as EMagic's Logic or Steinberg's Cubase.

Notice that this notein object is connected to the other objects via some straight black lines. These connections are called *patch cords*, and like the similarly named cables that connect different components of a modular synthesizer together, the patch cords in Max define the path of data flow and allow objects to act together. Data emanates outwards from the bottom of a object into the top of another object; the notein object in this example has three outlets which represent, from right to left, the MIDI channel, velocity, and pitch of a note event.

The + *12* object takes the integer input from the pitch outlet of the notein object, adds

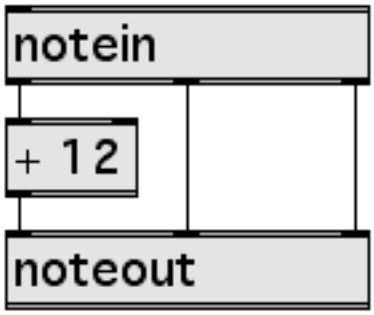

Figure A.1. *Max patch that adds twelve semitones (an octave) to all incoming MIDI data*

12, and outputs the result. This is then passed to the *noteout* object, which will output the altered MIDI information out the interface, presumably to a sound generator of some sort. The function of this simple patch, therefore, is to shift incoming note events up an octave in pitch.

All of the data flowing in Figure A.2 is integer data. As primitives in its message passing architecture Max is able to pass 32-bit integers, 32-bit floating-point values, and symbols. Additionally, a message can consist of any combination of these data types combined into a multi-element list. Since a 32-bit value can be used to reference an address in memory, some objects pass pointers which they dereference internally to operate on more complicated data structures.

Programming a Max object consists of defining a set of responses to incoming messages. For instance, the + object in Figure A.2 has been designed to respond to incoming integers by adding the addend argument, in this case 12, to the input and sending the result out the outlet. Methods can be defined to respond generally to any of the primitive types, or specifically to a named message. Appendix **??** is a good overview of what a Max/MSP external really is.

A.2 Messages

MIDI was proposed in 1982 and standardized in 1983 by a consortium of synthesizer manufacturers[3, 4]. The idea was to create a standard for the communication of musical performance data between synthesizers and sequencers of different makes and models. The protocol was popular with companies because it allowed their products to interface with thos efrom better-known manufacturers such as Roland and Sequential Circuits, who spearheaded the MIDI effort. The protocol was also popular with composers who wanted to connect the various synths in their studios to a single master sequencer. When computers became more powerful software MIDI sequencers appeared, and MIDI was used as the backbone of increasingly-complex scores. MIDI continues to be popular today amongst composers working with electronic tools.

Miller Puckette's original version of Max was written primarily for manipulation of the MIDI protocol. Max provided composers working with MIDI equipment a graphical environment in which to rapidly create maps of data flow: MIDI could be input into the computer or generated internally, operated upon by some Max programming, and output again to generate noise on an external synthesizer. Although the programming model has changed subtly and a lot of functionality has been added since those original days, much of what was available then to the Max programmer remains in the language.

The original version of Max, written in 198x by Miller Puckette while working at IRCAM, was written primarily for manipulation of the MIDI protocol. Max provided composers working with MIDI equipment a graphical environment in which to rapidly create maps of data flow: MIDI could be input into the computer or generated internally, operated upon by some Max programming, and output again to generate noise on an external synthesizer. Although the programming model has changed subtly and a lot of functionality has been added since those original days, much of what was available then to the Max programmer remains in the language.

The basic building block of Max programming is called an *object*. Like many object-

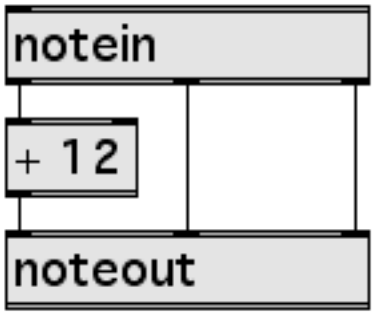

Figure A.2. *Max patch that pitches all incoming MIDI data up an octave*

priented programming languages, Max is made up of many *classes* of objects, each of which has a unique behaviour. Different objects can be used together in different ways to accomplish the desired programming task. At the time of writing, objects for the Max environment can be programmed in C, Java, or JavaScript. Third-party objects provide support for PERL, Python, and some other less well-known scripted languages.

For example, have a look at Figure A.2. What you might call a program in another language is called a *patch* in Max: this simple patch contains three objects, each of which is visually represented by a grey box. The class of a Max object is defined by the text contained within its box. The *notein* class is one of several classes designed to receive MIDI input from the outside world. Max is able to receive MIDI input through a standard MIDI interface, the kind that one could use with a commercial sequencing package such as EMagic's Logic or Steinberg's Cubase.

Notice that this notein object is connected to the other objects via some straight black lines. These connections are called *patch cords*, and like the similarly named cables that connect different components of a modular synthesizer together, the patch cords in Max define the path of data flow and allow objects to act together. Data emanates outwards from the bottom of a object into the top of another object; the notein object in this example has three outlets which represent, from right to left, the MIDI channel, velocity, and pitch of a note event.

The + *12* object takes the integer input from the pitch outlet of the notein object, adds 12, and outputs the result. This is then passed to the *noteout* object, which will output the altered MIDI information out the interface, presumably to a sound generator of some sort. The function of this simple patch, therefore, is to shift incoming note events up an octave in pitch.

All of the data flowing in Figure A.2 is integer data. As primitives in its message passing architecture Max is able to pass 32-bit integers, 32-bit floating-point values, and symbols. Additionally, a message can consist of any combination of these data types combined into a multi-element list. Since a 32-bit value can be used to reference an address in memory, some objects pass pointers which they dereference internally to operate on more complicated data structures.

Programming a Max object consists of defining a set of responses to incoming messages. For instance, the + object in Figure A.2 has been designed to respond to incoming integers by adding the addend argument, in this case 12, to the input and sending the result out the outlet. Methods can be defined to respond generally to any of the primitive types, or specifically to a named message.

A.3 Signals

In 1998, the Max universe was extended with the addition of *MSP*, a set of objects that enables the processing of signals using the Max paradigm of object-oriented programming with boxes and patch cords. Unlike messages, which are not necessarily connected with any notion of time, signals consist of a steady stream of data, each sample equally spaced apart in time. At CD audio rate, a signal contains 44100 *samples* (pieces of data) every second. Software that operates on this stream of signal data must process every one of these samples - in practical terms, this can place considerable demands on the computational resources available.

Furthermore, to ensure that the processing of samples keeps up with the demand for

samples by the digital to analog converter, the *DSP engine* that handles signals in Max/MSP must process samples no slower than the rate specified by the audio sampling rate. At CD audio rate, this means that the DSP engine must process the data no slower than one sample every $1/44100 = 0.0000226757$ seconds. Since the clock rate of a modern OS X or Windows XP CPU is typically much greater than the CD audio sampling rate, one might assume that this rate of calculation is easily met. However, the number of cycles used for a single sample of data can be arbitrarily large, depending on the signal network that the software has implemented. Also, many simultaneous channels of signal data may be active at once - music studios with as many as 64 audio output channels coming from a computer are not uncommon. Finally, modern multitasking operating systems demand that many separate processes be shared between the available computational resources - unlike some older operating systems, such as Mac OS 9 and earlier, it is no longer possible to write a piece of software that monopolizes the computer. All of these factors contribute to an environment that demands efficiency in the processing of signal data. Events, on the other hand, can be generated and processed at a non-uniform rate.

A.4 Matrices

Jitter was introduced in 2002, and with it the matrix was introduced as a new data type that ”piggybacks” onto the Max event model. The ”matrix” message is followed by an integer whose contents are actually a pointer in memory to the location of a large data set. Object operators must peek into this memory location to obtain the dimensions of the data.

A.5 The Threading Model

Obviously it makes more sense to think in terms of signals when the values will be rapidly changing, and more sense to think in terms of events when changes occur less regularly. From a computational perspective, signals are much more demanding. However, in a per-

formance perspective signals have the practical advantage of being sample-synchronous with the signal-data that will result from their analysis.

Some human ears perceive frequencies as high as 20 kHz, whereas our eyes refresh the image that's sent to our brain at a rate as much as three orders of magnitude lower. Correspondingly, MSP operates with very strict timing, and Jitter does not. In fact, Jitter was designed so that its frame rate adapts to the processing power available after audio and high-priority event processing has finished. The operation of this variable frame-rate architecture involves a system of event queues and priorities which is the subject of the remainder of this paper.

The internal Max/MSP "engine" changed a great deal during the transition from the cooperative multitasking world of Mac OS 9 to the pre-emptive multitasking environments of Mac OS X and Windows XP. In version 4.5 of Max/MSP/Jitter there are three primary threads of execution. The first is called the main thread this low priority thread is responsible for bursty, expensive operations, such as the servicing of the user interface objects. When you use the mouse to click on a button object or change the value in a number box, the main thread is where the results of the user interface action are calculated and the screen is redrawn. The high priority scheduler thread operates on time-sensitive data, such as the bytes input through MIDI interfaces, or bangs emanating from a clocked object like metro. The very high priority thread that calculates the MSP signal vectors is called the perform thread.

Although the perform thread is given a higher priority in the operating system's thread scheduling mechanism than the scheduler thread, and the scheduler a higher priority than the main, in a pre-emptive multitasking operating system any thread can interrupt any other at any time. Indeed, on a computer with more than one CPU, more than one of these threads can be executing simultaneously. This can lead to confusing situations for the novice (or experienced!) Max programmer when a patch has data that flows in different threads, and processes can interrupt one another in the midst of a calculation. The organization of the scheduler thread is further complicated by two options that can be set in Max's DSP Sta-

tus dialog. If overdrive is disabled, all scheduler data is processed in the main thread. If overdrive and scheduler in audio interrupt are both enabled, all scheduler data is processed in the perform thread immediately prior to the calculating of the signal vector. If overdrive is enabled but scheduler in audio interrupt is not enabled, the scheduler thread exists normally. These three configurations allow the Max programmer to tailor the execution of high-priority events to their needs: disabling overdrive removes the special status of scheduler messages but increases efficiency through the elimination of the additional thread, whereas executing the scheduler in the perform thread ensures the most accurate timing but reduces the amount of time the perform thread has available to calculate the signal vector. With the latter option one must be careful to keep the scheduler operations very short. A lengthy scheduler process increases the risk of exceeding the amount of time available to process the signal vector, which may result in audible glitches.

Regardless of the configuration of the scheduler, if the processing resulting from a clocked action is expensive it is usually wise to transfer the execution of the processing to the main thread, where a lengthy operation will only delay the execution of other time-insensitive actions. This can be accomplished using the defer object or jit.qball. Conversely, execution can be transferred from the main thread to the scheduler thread using the delay object with an argument of 0. There are objects that operate in both threads: qmetro, for instance, functions as a metronome internally by using the scheduler thread to clock when bangs should be output, but instead of sending the bangs in the scheduler thread it defers their output to the main thread. This deferral is done using an usurp mechanism: if a message is waiting in the main thread's queue and has not yet been output, any new message coming from the same object will replace the old message in the queue.

Figure A.3 provides an illustration of the usurp mechanism in action. The two counter objects in Figure A.3 are deferred to the main thread counter a uses an usurp mechanism and counter b does not. Figure A.4 illustrates a schedule of executing messages as they are passed from the scheduler thread to the main thread. On the first line counter a has sent out a 1, and this message is placed in the main thread's queue. The second line sees counter

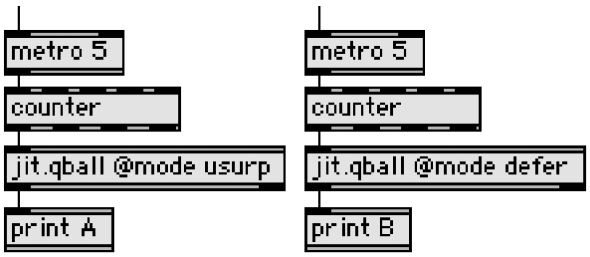

Figure A.3. *An illustration of the usurp mechanism used in a Max patch.*

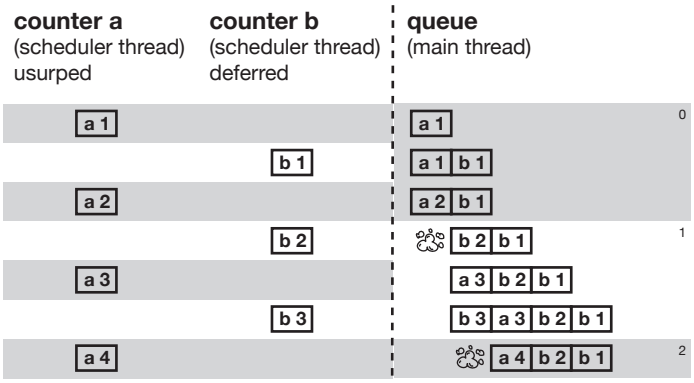

Figure A.4. *Time sequence of messages running through the patch in Figure A.3.*

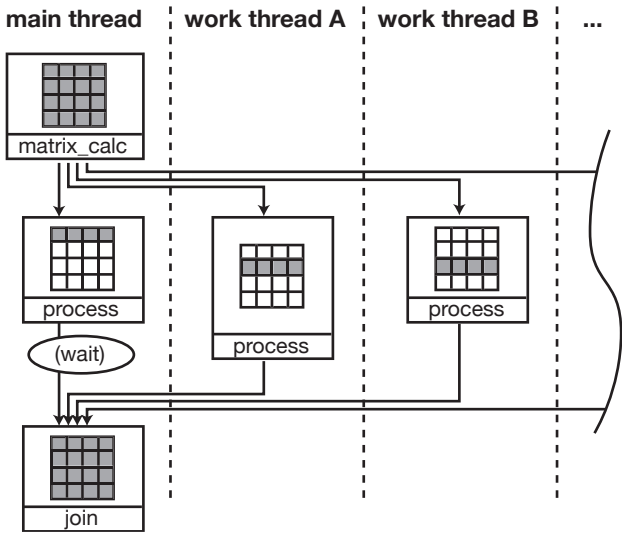

Figure A.5. *An illustration of the execution flow in a Jitter object's matrix processing method.*

b output a 1, which is also placed on the queue. On the third line counter a outputs a 2, which, because of the usurp mechanism, replaces the 1 from counter a that was waiting to be output. The fourth line illustrates the main thread's processing of the first event in the queue, as well as the output of a 2 from counter b. The fifth line shows the output of 3 from counter a, which is added to the front of the queue because no other messages from counter a are waiting to be output. On the sixth line the 3 message from counter b is placed at the front of the queue. The seventh line shows the processing and removal of counter b's 3 message, as well as the replacement of the 3 from counter a with the new output of 4 because of the usurping mechanism.

In the case of qmetro, the usurp mechanism ensures that only a single bang from the qmetro is ever waiting to be output in the main thread's queue. In a situation where a clocked object is connected to a network of Max objects that perform some expensive computations, and bangs are output more quickly than the network can perform the computations, the usurping mechanism prevents stack overflow. Networks of Jitter objects that operate on video matrices iterate over many thousands of pixels. Accordingly, these de-

manding calculations typically take place in the main thread. In fact, on multi-processor machines Jitter maintains a pool of threads for its own use in iterating over large matrices. Figure A.5 shows the execution flow in the matrix_calc method of a multiprocessor-capable Jitter object. The Max programmer need not think about these extra threads, other than to know that they can significantly speed up iteration over large matrices.

It is common to drive a Jitter network with a qmetro object set to a very short period between bangs. Because of the usurping mechanism discussed above, the result is that the Jitter network will calculate new frames as quickly as possible given the computational resources it has available. Because the scheduler thread and perform thread execute concurrently, processing of audio and time-dependent events is not affected. The frame rate of the video output is dependent on the available computational resources. Since a modern operating system typically operates several dozen processes in the background, each of which requires a varying amount of time to be serviced, the available computational resources are constantly fluctuating. Therefore the frame rate of the video output also fluctuates. Luckily, the eye does not have the same stringent periodic requirements as the ear. This variable frame-rate architecture requires a different mind-set from that required by MSP when evaluating the computational load of a patch. Because it has a fixed rate at which it must process samples, with MSP the computational load can be defined as the ratio of the time taken to calculate a single sample to the period of the audio signal. On the other hand, driving a Jitter network with a quickly resetting qmetro as described above effectively means that the Jitter processing will suck up all available processing power after the perform and scheduler threads have done their work. The best way to estimate the different computational loads of different Jitter networks is therefore to compare their frame rates, something that is easily done with the fpsgui object, a graphical object that can be connected anywhere in the network to provide valuable feedback about the frame rate, data type of the matrix, and other information. It is worth noting that the frame rate of the computer's monitor or video projector is a practical upper limit on the rate that output imagery can be updated.

Appendix B

The Jitter Signal-Visualization Objects

Shortly after beginning to experiment with gesture signals it became apparent that a set of tools to visualize the signals was lacking. Unlike Max events, for which there exist numerous user interface objects that one can use to monitor data values as they flow through a patch, the set of objects to monitor MSP signals was limited. The objects that did exist, such as the oscilloscope-emulating scope~, were designed with the expectation that the signals they would be monitoring would oscillate about the zero point like a standard audio signal. Furthermore, synchronously visualizing an arbitrary number of signals in real-time was not possible. So a toolkit was built to provide Max/MSP/Jitter users with improved signal-visualization capabilities; these tools have proven themselves as extremely useful in assessing the performance of and debugging algorithms operating on the gesture data.

The design of the toolkit was kept flexible by implementing a bridge between the signal world of MSP and the matrix world of Jitter. Once in Jitter, not only can the signal data be rendered in any of the various visual forms that Jitter allows, it can be used for non-visualization purposes. One example use is frame-based processing of the signal data by Jitter objects, which is conceptually much simpler - rather than build networks that operate on a sample at a time and maintain complicated index signals, Jitter supports the type of block operations that a frame-based algorithm makes use of.

To translate from the synchronous signal world of MSP to the asynchronous, frame-based Jitter paradigm is done with the jit.catch~ object, which works as follows: in the signal processing thread, incoming vectors of MSP signal data are written into an internal

circular data buffer. When the object receives a bang a new matrix is created and populated with floating-point data from the internal buffer. The format of the data written to the matrix is dependent on an attribute of the object called the *mode*: mode 0 outputs all the data received since the last bang; mode 1 outputs the most recent n samples where n is the value of an attribute called the *framesize*; mode 2 outputs all the samples that have been received since the last bang as a two-dimensional matrix of width n with variable height, with any leftover samples saved for the next bang; and mode 3 outputs the most recent n samples before the point where the trigger channel (set by the *trigchan* attribute) has crossed the trigger threshold (set by the *trigthresh* attribute.)

In Figure B.1 the jit.catch~ object is taking two input signals. jit.catch~ handles multiple signal inputs by multiplexing the data into multiple planes; the two-channel input here means that every time it receives a bang from the qmetro object it will output a 2-plane matrix. The object is in **trigger** mode - note that the slower of the two sine waves is about at the middle of the graph on the right edge. This is the default *trigthresh* value of 0.

The jit.graph object takes a one-dimensional matrix, such as the one that jit.catch~ produces, and renders it as a two-dimensional plot. jit.graph has various drawing modes - points, lines, areas, and bipolar areas can be rendered - and colors for individual signals can be set individually and blended by using the *outname* and *clearit* attributes.

A complimentary object called jit.release~ is the converse to jit.catch~: it takes floating-point matrices as input and streams out signals. Although it is important in some situations, such as when you're using Jitter to do real-time frame processing on signal data, it is not used in the context of the performance instrument that is the topic of this thesis, so we won't discuss it any further.

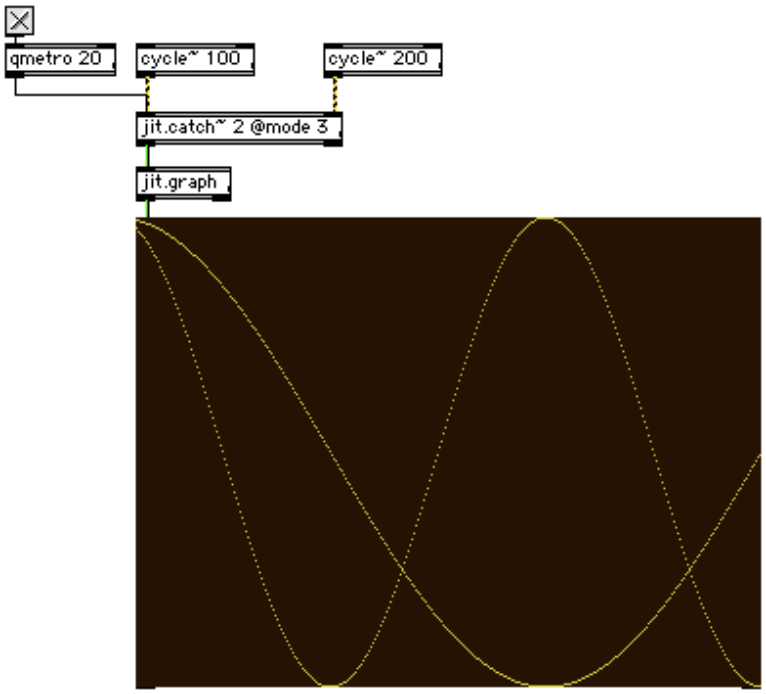

Figure B.1. *An example of the jit.catch~ and jit.graph objects in action.*

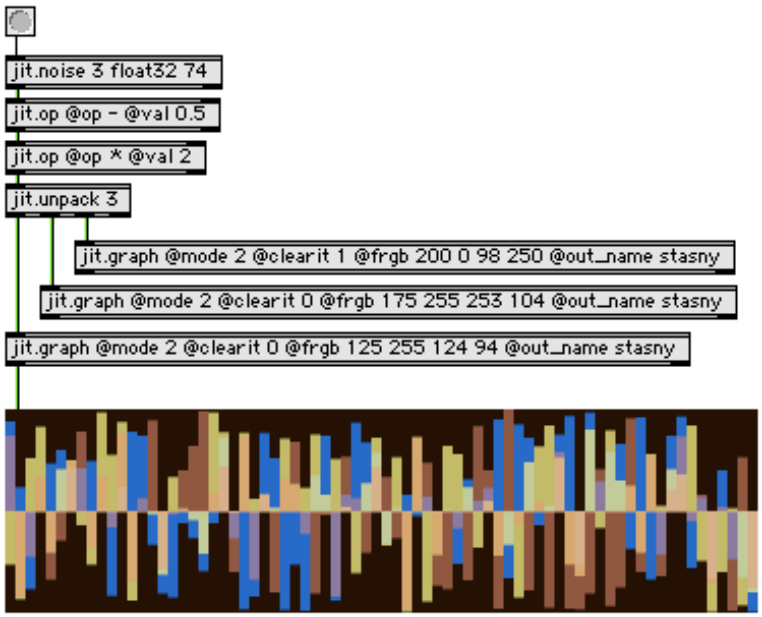

Figure B.2. *An example of using multiple jit.graph objects and the clearit and outname attributes to composite multiple graphs and colors.*

Appendix C

Sequential Analysis

Simply put, sequential analysis is statistical analysis where the sample size is not fixed in advance. The decision to terminate an experiment depends on prior observations. Sequential Analysis can arrive at a decision much sooner and with fewer observations than equally reliable test procedures based on a pre-determined number of observations. The brief description of the theory that follows is a summary of pertinent sections from Wald's classic text on the subject [44].

C.1 Preliminaries

Consider a method of testing a hypothesis H: at each stage of a experiment, a decision has to be made between 1. accepting the hypothesis H, 2. rejecting the hypothesis H, and 3. continuing the experiment by incorporating another observation. At the mth trial our collected data $(x_1, x_2, ..., x_m)$ will be a member of an m-dimensional sample space that can be split into three mutually exclusive parts: a region of acceptance R_m^0, a region of rejection R_m^1, and a region of indecision R_m. The fundamental problem in the theory of sequential tests is to properly define these three sets.

We assume that the distribution of x is known except for the values of a finite number of parameters $\theta = (\theta_1, ..., \theta_k)$. Since the distribution of x is determined by the parameter point θ, the probability of accepting H_0 will be a function of θ; we call this function the **operating charateristic** function and denote it as $L(\theta)$.

Let the random variable of the number of observations required by a sequential test to reach a decision be represented by n. For any given test procedure the expected value of n depends only on the distribution of x, and since the distribution of x depends only on θ, the distribution of n depends only on θ. The expected value of n at a given value of θ can be denoted as $E_\theta(n)$ - this can also be called the **average sample number** function.

Defining a sequential test requires investigation of the preferences for rejection or acceptance of the null hypothesis H_0 over the parameter space of θ. It is necessary to divide this space into three mutually exclusive zones: a zone where acceptance of H_0 is strongly preferred, a zone where rejection of H_0 is strongly preferred, and a zone where neither acceptance nor rejection is strongly preferred. We use these zones when formulating requirements for $L(\theta)$. In the zone of indifference we don't impose any conditions on the behaviour of $L(\theta)$. For any θ in the zone of preference for acceptance the probability of rejecting the hypothesis H_0 should be less than or equal to a value α. For any θ in the zone of preference for rejection the probability of accepting H_0 should be less than or equal to a value β. In other words, in the zone of acceptance $1 - L(\theta) \leq \alpha$, and in the zone of rejection $L(\theta) \leq \beta$.

C.2 The Sequential Probability Ratio Test

Let $f(x, \theta)$ denote the distribution of the random variable x whose successive observations are $x_1, x_2, ...$, let H_0 be the hypothesis that $\theta = \theta_0$, and let H_1 be the hypothesis that $\theta = \theta_1$. For any m the probability that a sample $x_1, ..., x_m$ is obtained is given by

$$p_{1m} = f(x_1, \theta_1)...f(x_m, \theta_1) \tag{C.1}$$

when H_1 is true, and

$$p_{0m} = f(x_1, \theta_0)...f(x_m, \theta_0) \tag{C.2}$$

when H_0 is true. The **sequential probability ratio test** is defined by choosing two positive constants A and B, $B < A$. At the mth trial of the experiment the ratio of $r = p_{1m}/p_{0m}$ is compared to these constants. If $r < B$ the experiment is terminated with acceptance of the hypothesis H_0; if $r > A$ the experiment is terminated with rejection of the hypothesis H_0; if $B < r < A$ the experiment continues.

The constants A and B are determined so that the test will have the prescribed strength (α, β). Define a sample set of type 0 as one that leads to the acceptance of H_0 and a sample set of type 1 as one that leads to the rejection of H_0 and acceptance of H_1. The probability of any sample of type 1 is at least A times as large under hypothesis H_1 as under H_0. Therefore the probability measure of the totality of all samples of type 1 is also at least A times as large under H_1 as under H_0. The probability measure of the totality of all samples of type 1 is the same as the probability that the sequential process will terminate with the acceptance of H_1. But we set this probability to be equal to α when H_0 is true and to $1 = \beta$ when H_1 is true. Thus we obtain

$$1 - \beta \geq A\alpha \tag{C.3}$$

which we can write as

$$A \leq \frac{1 - \beta}{\alpha} \tag{C.4}$$

Thus, $(1 - \beta)/\alpha$ is an upper limit for A. The lower limit for B can be derived in a similar way:

$$B \geq \frac{\beta}{1 - \alpha} \tag{C.5}$$

These inequalities give us upper and lower boundaries for A and B, but not exact values. Simply setting A and B to be equal to the expressions on the right hand sides of equations C.4 and C.5, respectively, does not result in any appreciable increase in the value of either α or β. Furthermore, the increase in the necessary number of observations caused by the

use of these simplified expressions will generally only be slight.

When the data is distributed normally, the probability densities of the samples are given by

$$p_{0m} = \frac{1}{2\pi^{m/2}\sigma^m} e^{-\frac{1}{2\sigma^2}\sum_{\alpha=1}^{m}(x_\alpha-\theta_0)^2} \tag{C.6}$$

if $\theta = \theta_0$ and

$$p_{1m} = \frac{1}{2\pi^{m/2}\sigma^m} e^{-\frac{1}{2\sigma^2}\sum_{\alpha=1}^{m}(x_\alpha-\theta_1)^2} \tag{C.7}$$

if $\theta = \theta_1$. By taking logarithms and simplifying, the test inequality can be written as

$$\log\frac{\beta}{1-\alpha} < \frac{\theta_1-\theta_0}{\sigma^2}\sum_{\alpha=1}^{m} x_\alpha + \frac{m}{2\sigma^2}(\theta_0^2-\theta_1^2) < \log\frac{1-\beta}{\alpha} \tag{C.8}$$

Figure C.1 illustrates the application of this test procedure when testing if the mean of a normal random variable is below a certain value. The vertical axis is the sum of the input random variable, and the horizontal axis represents successive trials of the experiment. If the sum of successive samples exceeds the upper threshold the hypothesis is rejected; if the sum falls below the lower threshold the hypothesis is accepted; while the sum stays in between the two thresholds the test continues.

The operating characteristic function of a sequential probability ratio test with $A = (1-\beta)/\alpha$ and $B = \beta/(1-\alpha)$ can be approximated as

$$L(\theta) \sim \frac{(\frac{1-\beta}{\alpha})^h - 1}{(\frac{1-\beta}{\alpha})^h - (\frac{\beta}{1-\alpha})^h} \tag{C.9}$$

where when x is normally distributed with unknown mean θ and known variance σ^2, the value of h is given by

$$h(\theta) = \frac{\theta_1+\theta_0-2\theta}{\theta_1-\theta_0} \tag{C.10}$$

The expected length of the test can be approximated by

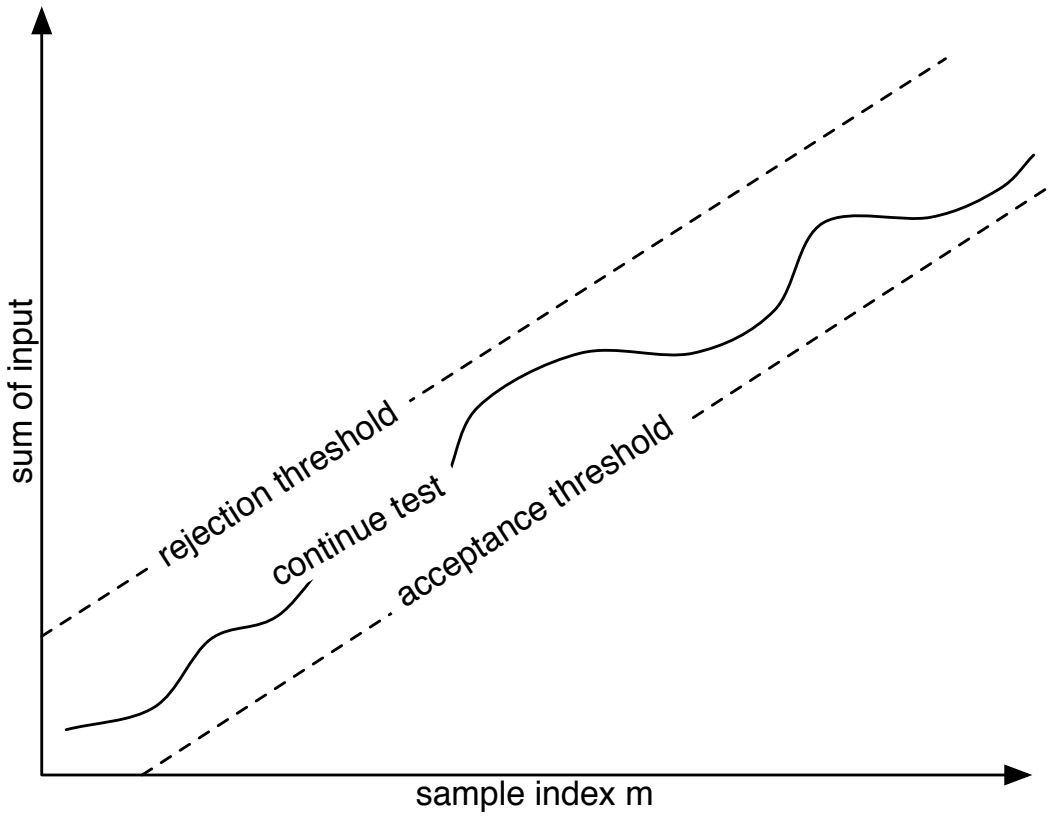

Figure C.1. *An illustration of the sequential test procedure when testing if the mean of a normal random variable is below a certain value. If the sum of successive samples exceeds the upper threshold the hypothesis is rejected; if the sum falls below the lower threshold the hypothesis is accepted; while the sum stays in between the two thresholds the test continues.*

$$E_\theta(n) \sim \frac{L(\theta)\log B + [1 - L(\theta)]\log A}{E_\theta(z)} \tag{C.11}$$

Appendix D

The wald~ external

The wald~ object implements a sign detector that follows the envelope of an incoming signal. The goal is to detect transitions across the zero line. The code is below.

```
/*  wald~
envelope following using the wald's sequential analysis.
*/


#include "ext.h"
#include "math.h"
#include "z_dsp.h"

#define VERSION_STRING "wald~ 2006.10.26  bugs?  bbn@saoul.ca"

void *wald_class;

typedef struct _wald {
t_pxobject l_obj;

float alpha, beta, theta1, theta0, variance, acceptthresh, rejectthresh, n, threshinc, thetafrac,
original_acceptthresh, original_rejectthresh;

float positive,
sum;

void *infoout;
} t_wald;

t_int *wald_perform(t_int *w);
void wald_dsp(t_wald *x, t_signal **sp, short *count);
void wald_assist(t_wald *x, void *box, int msg, int arg, char *dst);
void *wald_new(t_symbol *s, short argc, t_atom *argv);
void wald_alpha(t_wald *x, double alpha);
void wald_beta(t_wald *x, double beta);
void wald_theta(t_wald *x, double theta);
void wald_variance(t_wald *x, double variance);
void wald_n(t_wald *x, double n);
void wald_thetafrac(t_wald *x, double t);
void calc_threshes(t_wald *x);
```

```
void main(void) {
setup((t_messlist **)&wald_class,(method)wald_new, (method)dsp_free,
(short)sizeof(t_wald), 0L, A_GIMME, 0);
addmess((method)wald_dsp, "dsp", A_CANT, 0);
addmess((method)wald_assist, "assist", A_CANT, 0);
addmess((method)wald_alpha,"alpha",A_DEFFLOAT,0);
addmess((method)wald_beta,"beta",A_DEFFLOAT,0);
addmess((method)wald_variance,"variance",A_DEFFLOAT,0);
addmess((method)wald_n, "n", A_DEFFLOAT, 0);
addmess((method)wald_thetafrac, "thetafrac", A_DEFFLOAT, 0);
dsp_initclass();
    post(VERSION_STRING);
}

t_int *wald_perform(t_int *w) {
t_float *sigin  = (t_float *)(w[1]);
t_wald *x = (t_wald *)(w[2]);
int n = (int)(w[3]);
t_float *trigOut =(t_float *)(w[4]);
t_float *sumOut = (t_float *)(w[5]);
t_float *acceptout = (t_float *)(w[6]);
t_float *rejectout = (t_float *)(w[7]);

float  in,
sum = x->sum,
positive = x->positive,
acceptthresh = x->acceptthresh,
rejectthresh = x->rejectthresh,
threshinc = x->threshinc;

while (--n) {
sum += *++sigin;

*++acceptout = acceptthresh;
*++rejectout = rejectthresh;
*++sumOut = sum;

if (((positive > 0.) && (sum < acceptthresh)) || ((positive < 0.) && (sum > acceptthresh)))
{
x->positive *= -1.;
positive = x->positive;
sum = 0.;
calc_threshes(x);
acceptthresh = x->acceptthresh;
rejectthresh = x->rejectthresh;
threshinc = x->threshinc;
}
else if (((positive > 0.) && (sum > rejectthresh)) || ((positive < 0.) && (sum < rejectthresh)))
{
sum = 0.;
acceptthresh = x->original_acceptthresh;
rejectthresh = x->original_rejectthresh;
threshinc = x->threshinc;
}
else
{
acceptthresh += threshinc;
```

```
rejectthresh += threshinc;
}


*++trigOut = positive;

}

x->acceptthresh = acceptthresh;
x->rejectthresh = rejectthresh;
x->sum = sum;
return (w+8);
}

void wald_alpha(t_wald *x, double alpha)
{
if (alpha > 0.)
x->alpha = alpha;
calc_threshes(x);
}

void wald_beta(t_wald *x, double beta)
{
if (beta > 0.)
x->beta = beta;
calc_threshes(x);
}

void wald_variance(t_wald *x, double variance)
{
if (variance > 0.)
x->variance = variance;
calc_threshes(x);
}

void wald_n(t_wald *x, double n)
{
if (n > 0.)
x->n = n;
calc_threshes(x);
}

void wald_thetafrac(t_wald *x, double t)
{
if ((t >= 0.) &&(t <= 1.0))
x->thetafrac = t;
calc_threshes(x);
}

void calc_threshes(t_wald *x)
{
x->theta0 = (-x->positive) * sqrt( -2*x->variance / x->n * (log((1.0-x->beta)/x->alpha) + (1-x->alpha)*(log(x->beta/(1-x->alpha)) - log((1-x-

x->theta1 = (-x->theta0) * (1. - x->thetafrac);
x->theta0 = x->theta0 * x->thetafrac;
```

```
x->acceptthresh = x->variance / (x->theta1 - x->theta0) * log(x->beta / (1 - x->alpha));
x->original_acceptthresh = x->acceptthresh;
x->rejectthresh = x->variance / (x->theta1 - x->theta0) * log((1 - x->beta) / x->alpha);
x->original_rejectthresh = x->rejectthresh;
x->threshinc = (x->theta0 + x->theta1) / 2.;
}


void wald_dsp(t_wald *x, t_signal **sp, short *count) {
dsp_add(wald_perform, 7, sp[0]->s_vec-1, x, sp[0]->s_n+1, sp[1]->s_vec-1, sp[2]->s_vec-1, sp[3]->s_vec-1, sp[4]->s_vec-1);
}

void *wald_new(t_symbol *s, short argc, t_atom *argv) {
    t_wald *x = (t_wald *)newobject(wald_class);

    dsp_setup((t_pxobject *)x,1);

x->infoout = outlet_new((t_object *)x, "anything");
    outlet_new((t_object *)x, "signal");
outlet_new((t_object *)x, "signal");
    outlet_new((t_object *)x, "signal");
outlet_new((t_object *)x, "signal");
x->positive = 1.0;
x->alpha = 0.0001;
x->beta = 0.0001;
x->variance = 0.00001;
x->sum = 0.;
x->n = 10;
x->threshinc = 0.;
x->thetafrac = 1.;
calc_threshes(x);

return (x);
}

void wald_assist(t_wald *x, void *box, int msg, int arg, char *dst)
{
if (msg==1) { //inlets
switch (arg) {
case 0: strcpy(dst,"(signal) data"); break; }
} else if (msg==2) { // outlet
switch (arg) {
case 0: strcpy(dst,"(signal) current sign"); break;
case 1: strcpy(dst,"(signal) running sum"); break;
case 2: strcpy(dst,"(signal) acceptance threshold"); break;
case 3: strcpy(dst,"(signal) rejection threshold"); break;
}
}
};
```

Appendix E

Java Calibration Code

The calibration has three stages: in the first stage, both sticks are moved through their entire vertical range of motion, in the second stage both sticks are rubbed over the surface of the drum, and in the third stage the sticks are motionless on the surface of the drum. For each of the stages the data is collected into a buffer and the radioDrumCalibrator class analyzes the results and sends the proper parameters out to the MSP objects that implement the scaling and event detection. Parameters for α, β, and n can be modified at any time from the top level of the patch. Calibration data can be saved and recalled through a parameter management scheme.

The radioDrumCalibrator mxj Java object handles the logic of the calibration. The code is reproduced below.

```
import com.cycling74.max.*;
import com.cycling74.msp.*;

/*
radioDrumCalibrator
ben nevile, march 30 2005

the idea is that a 4-channel buffer~ of calibration data is available.  the buffer~
only contains the data for one stick, but of course in the patch the calibration data
will be collected for both sticks simultaneously.
in this buffer~, the stick will start at the z-threshold -- that is,
the user will begin the calibration procedure with the sticks lying on
the surface of the drum for at least one second.
then they will move the stick all around to what they think will be the
maximum and minimum points of signal strength.  they will also strike the
surface of the drum with the stick with  maximum and minimum strength.
then they will signal that the calibration is over.

this java code will analyze the buffer~ and determine the multiplicative and
additive constants necessary to scale the x,y and z signals within a friendly range,
```

```
as well as the thresholds for the peak-picking algorithm..

April 18, 2006

Breaking the calibration into three separate routines,
triggered by the following messages:

"vertical" - Performer should move sticks from as far away as possible
to as close as possible.  The performer should hit the surface as hard as possible.
Calibrator will calculate the minimum and maximum Z plus the maximum V.

"horizontal" - With sticks on surface, performer should move them horizontally
around the surface, making sure to get to all parts of the surface that will be hit.
Calibrator will calculate the average Z position (surface threshold) and the ranges
of the X and Y params.

"noise" - Stick should remain motionless.  Calibrator will quantify the noise.

*/


public class radioDrumCalibrator extends MaxObject
{
private static final int numChans = 4;
private static final int drumBottom = 2;
private static final int drumTop = 3;
private static final int drumRight = 1;
private static final int drumLeft = 0;

private String bufferName = null;
private float zstddev = 3.0f;
private float vstddev = 3.0f;
private float vrevstddev = 2.0f;
// eventually this last one will be called aThreshStandardDeviations
private float zThreshAverage = 0.0f;
private float zThreshStdDev = 0.0f;
private float vThreshStdDev = 0.0f;

/**
 * minmaxSignal will always contain the minimum and maximum values for each signal.
 * All calibration params will be recalculated if minmaxSignal is awry.
 * we initialize it by default to null.
 */
private float minmaxSignal[][] = null;
private static float BIG_NUMBER_THAT_SIGNAL_SHOULD_ALWAYS_BE_SMALLER_THAN = 99999.f;
private static float SMALL_NUMBER_THAT_SIGNAL_SHOULD_ALWAYS_BE_BIGGER_THAN = -99999.f;

/**
 * same deal for minmaxZ.
 * if noise calibration is done before minmaxZ has been initialized, noise calib
 * just stores the data and waits for vertical calib before it will output data.
 */
private float minmaxZ[] = null;
private float minmaxVZ[] = null;
/**
 * We will maintain all three of these buffers so that we can recalculate
 * if we need to change the minmaxSignal numbers.
```

```
 * for instance, say you do a vertical calibration and then do a horizontal calibration,
 * but the horizontal results in larger maxmin values than the vertical.  the vertical
 * would need to be recalculated.
 */
private float[][] verticalData = null;
private float[][] horizontalData = null;
private float[][] noiseData = null;


private void recalc(float[][] data)
{
//initialize if necessary
if (minmaxSignal == null)
{
minmaxSignal = new float[numChans][2];
for (int i=0;i<numChans;i++)
{
minmaxSignal[i][0] = BIG_NUMBER_THAT_SIGNAL_SHOULD_ALWAYS_BE_SMALLER_THAN;
minmaxSignal[i][1] = SMALL_NUMBER_THAT_SIGNAL_SHOULD_ALWAYS_BE_BIGGER_THAN;
}
}

//determine the minimum and maximum values for each channel
float tempMinMaxSignal[][] = new float[numChans][];
for (int i=0;i<data.length;i++) {
tempMinMaxSignal[i] = minmax(data[i]);
}

//update vals if necessary
for (int i=0;i<numChans;i++)
{
if (tempMinMaxSignal[i][0] < minmaxSignal[i][0])
minmaxSignal[i][0] = tempMinMaxSignal[i][0];
if (tempMinMaxSignal[i][1] > minmaxSignal[i][1])
minmaxSignal[i][1] = tempMinMaxSignal[i][1];
}

//output new values to Max patch
for (int i=0;i<numChans;i++)
{
outlet(0, "minmaxSignal",
new Atom[] {Atom.newAtom(i),
Atom.newAtom(minmaxSignal[i][0]),
Atom.newAtom(minmaxSignal[i][1])});
}
//update other components of calibration that depend on this data
recalcVertical();
recalcHorizontal();
recalcNoise();
}

public radioDrumCalibrator(Atom[] args)
{
declareIO(1,2);
declareAttribute("bufferName");
declareAttribute("zstddev",null,"setzstddev");
declareAttribute("vstddev",null,"setvstddev");
```

```
declareAttribute("vrevstddev",null,"setvrevstddev");
}

public void setzstddev(Atom a[])
{
zstddev = a[0].toFloat();
outputThresholds();
}

public void setvstddev(Atom a[])
{
vstddev = a[0].toFloat();
outputThresholds();
}

public void setvrevstddev(Atom a[])
{
vrevstddev = a[0].toFloat();
outputThresholds();
}

private void outputThresholds()
{
outlet(0, "zThresh", zThreshAverage-zstddev*zThreshStdDev);
outlet(0, "vThresh", 0.0f-vstddev*vThreshStdDev);
outlet(0, "vRevThresh", vrevstddev*vThreshStdDev);
}

public void analyze()
{
error("The analyze method has been deprecated. "+
"You are using an old version of the patch with a newer version of the "+
"radioDrumCalibrator.class file.");
}

private float[][] getData(String bufferName, int numberOfSamples)
{
//move data from buffer into java float arrays
float data[][] = new float[numChans][];
for (int i=0;i<numChans;i++) {
data[i] = MSPBuffer.peek(bufferName, i+1, 0, numberOfSamples);
if (data[i].length != numberOfSamples)
{
error("radioDrumCalibrator: "+
"there is a problem with the number of samples.");
return null;
}
}
return data;
}

private void copyData(float[][] A, float[][] B)
{
for (int c=0;c<A.length;c++)
for (int i=0;i<A[0].length;i++)
{
B[c][i] = A[c][i];
```

```
}
}

private void rescaleData(float[][] A)
{
for (int i=0;i<numChans;i++) {
float range = minmaxSignal[i][1] - minmaxSignal[i][0];
for (int j=0;j<A[i].length;j++)
A[i][j] = (A[i][j] - minmaxSignal[i][0])/range;
}
}


public void vertical(int numberOfSamples)
{
//check for user error
if (bufferName == null) {
error("radioDrumCalibrator: "+
"I need a buffer name");
return;
}

verticalData = getData(bufferName, numberOfSamples);
recalc(verticalData);
}


private void recalcVertical()
{
if (verticalData == null) return;

int numberOfSamples = verticalData[0].length;

//copy verticalData to local array
float data[][] = new float[numChans][numberOfSamples];
copyData(verticalData, data);

//rescale the input signal arrays between 0 and 1
rescaleData(data);

//find the minimum and maximum values z
float z[] = new float[numberOfSamples];
for (int i=0;i<numberOfSamples;i++) {
z[i] = calcZ(data[drumLeft][i], data[drumRight][i],
 data[drumBottom][i], data[drumTop][i]);
}
minmaxZ = minmax(z);

//swap these two, because up is down and down is up
float ztemp = minmaxZ[0];
minmaxZ[0] = minmaxZ[1];
minmaxZ[1] = ztemp;

//output range to Max patch
outlet(0, "minmaxZ", new Atom[] {Atom.newAtom(minmaxZ[0]),
Atom.newAtom(minmaxZ[1])});
```

```
//rescale z
for (int i=0;i<numberOfSamples;i++)
z[i] = (z[i] - minmaxZ[0])/(minmaxZ[1]-minmaxZ[0]);

//calculate a z-velocity array and the minmax of it
float vz[] = new float[numberOfSamples-1];
for (int i=0;i<numberOfSamples-1;i++)
vz[i] = z[i+1]-z[i];
minmaxVZ = minmax(vz);

//output to the patch
outlet(0, "minmaxVZ", new Atom[] {Atom.newAtom(minmaxVZ[0]),
Atom.newAtom(minmaxVZ[1])});

}

public void horizontal(int numberOfSamples)
{
//check for user error
if (bufferName == null) {
error("radioDrumCalibrator: "+
"I need a buffer name");
return;
}

horizontalData = getData(bufferName, numberOfSamples);
recalc(horizontalData);
}

private void recalcHorizontal()
{
if (horizontalData == null) return;

int numberOfSamples = horizontalData[0].length;

//copy horizontalData to local array
float data[][] = new float[numChans][numberOfSamples];
copyData(horizontalData, data);

//rescale the input signal arrays between 0 and 1
rescaleData(data);

float x[] = new float[numberOfSamples];
float y[] = new float[numberOfSamples];
for (int i=0;i<numberOfSamples;i++) {
x[i] = calcXY(data[drumLeft][i], data[drumRight][i]);
y[i] = calcXY(data[drumBottom][i], data[drumTop][i]);
}
float minmaxX[] = minmax(x);
outlet(0, "minmaxX", new Atom[] {Atom.newAtom(minmaxX[0]),
Atom.newAtom(minmaxX[1])});
float minmaxY[] = minmax(y);
outlet(0, "minmaxY", new Atom[] {Atom.newAtom(minmaxY[0]),
Atom.newAtom(minmaxY[1])});
}
```

```
public void noise(int numberOfSamples)
{
//check for user error
if (bufferName == null) {
error("radioDrumCalibrator: "+
"I need a buffer name");
return;
}

noiseData = getData(bufferName, numberOfSamples);
recalc(noiseData);
}


private void recalcNoise()
{
if (noiseData == null) return;

int numberOfSamples = noiseData[0].length;
post("number of samples in use for noise calc:" + numberOfSamples);

//copy noiseData to local array
float data[][] = new float[numChans][numberOfSamples];
copyData(noiseData, data);

//rescale the input signal arrays between 0 and 1
     rescaleData(data);

float z[] = new float[numberOfSamples];
for (int i=0;i<numberOfSamples;i++)
{
z[i] = calcZ(data[drumLeft][i], data[drumRight][i],
 data[drumBottom][i], data[drumTop][i]);
}

// if we have no data for minmaxZ there's no point calculating the
// noise stats now.
if (minmaxZ == null) return;

//rescale z
for (int i=0;i<numberOfSamples;i++)
z[i] = (z[i] - minmaxZ[0])/(minmaxZ[1]-minmaxZ[0]);

//calculate a z-velocity array and the minmax of it
float vz[] = new float[numberOfSamples-1];
for (int i=0;i<numberOfSamples-1;i++)
vz[i] = z[i+1]-z[i];

//rescale v so that negative range goes from 0 to -1
for (int i=0;i<numberOfSamples-1;i++)
vz[i] = (vz[i] - minmaxVZ[0])/(0.0f-minmaxVZ[0]);


//calc a z-acceleration array
float az[] = new float[numberOfSamples-2];
for (int i=0;i<numberOfSamples-2;i++)
az[i]= vz[i+1]-vz[i];
```

```
float azAverage= calcAverage(az);
float aStdDev = calcStdDev(az, azAverage);
Atom azmsg[] = new Atom[3];
azmsg[0] = Atom.newAtom("wald");
azmsg[1] = Atom.newAtom("variance");
azmsg[2] = Atom.newAtom(aStdDev*aStdDev);
outlet(0, azmsg);
azmsg[1] = Atom.newAtom("theta");
azmsg[2] = Atom.newAtom(aStdDev/10.);
outlet(0, azmsg);

zThreshAverage = calcAverage(z);
outlet(0, "zThreshAverage", zThreshAverage);
zThreshStdDev = calcStdDev(z, zThreshAverage);
outlet(0, "zThreshStdDev", zThreshStdDev);
float vThreshData[] = new float[vz.length];
for (int i=0;i<vThreshData.length;i++)
vThreshData[i] = vz[i];
float vThreshAverage = calcAverage(vThreshData);
vThreshStdDev = calcStdDev(vThreshData, vThreshAverage);
outlet(0, "vThreshStdDev", vThreshStdDev);

post("z SNR: ("
+1.0+")/("+zThreshStdDev+") -> "
+(20 * Math.log(1.0/zThreshStdDev)/Math.log(10)));
post("v SNR: ("
+(1.0)+")/("+vThreshStdDev+") -> "
+(20 * Math.log(1.0/vThreshStdDev)/Math.log(10)));
outputThresholds();
}


float calcAverage(float data[]) {
double total = 0.0;
for (int i=0;i<data.length;i++)
total += (double)data[i];
return (float)(total / ((double)data.length));
}

float calcStdDev(float data[], double average) {
double total = 0.0;
for (int i=0;i<data.length;i++) {
double diff = (double)data[i] - average;
total += diff*diff;
}
return (float)Math.sqrt(total/(double)(data.length-1));
}

//interpolates between 0 and 1 based on two signal strengths
float calcXY(float zeroSig, float oneSig) {
if (zeroSig + oneSig == 0.0f)
return 0.5f;
else
return oneSig/(zeroSig + oneSig);
}
//calculates Z value based on four sigs
```

```
float calcZ(float sigA, float sigB, float sigC, float sigD) {
return (sigA+sigB+sigC+sigD);
}


//returns (min, max) of the input array
private float[] minmax(float f[])
{
float max = f[0], min = f[0];
for (int i=0;i<f.length;i++)
if (f[i] > max) max = f[i];
else if (f[i] < min) min = f[i];
return new float[] {min, max};
}

}
```

Appendix F

The event~ object

```
#include "ext.h"
#include "ext_obex.h"
#include "z_dsp.h"
#include <math.h>

// sample accurate event generator.
// like sample-and-hold, except that it outputs the sampled
// signals as float elements of a list.

#define EVENT_MAX_SIGNALS 32

void *event_class;

typedef struct _event
{
    t_pxobject s_obj;
    float s_prev;
    float s_thresh;
    void *float_outlet;
    int numsignals;
void *clock;
t_float *eventbuffer;
long eventreadindex, eventwriteindex, eventbuffersize;
} t_event;

t_int *event_perform(t_int *w);
void event_dsp(t_event *x, t_signal **sp, short *count);
void event_thresh(t_event *x, double f);
void *event_new(t_symbol *s, short argc, t_atom *argv);
void event_assist(t_event *x, void *b, long m, long a, char *s);
void event_new_output(t_event *x, long numsigs, t_float **sig);
void event_tick(t_event *x);

void main(void)
{
setup((t_messlist **)&event_class,
(method)event_new,
(method)dsp_free,
(short)sizeof(t_event),
0L, A_GIMME, 0);
addmess((method)event_dsp, "dsp", A_CANT, 0);
addfloat((method)event_thresh);
```

```
dsp_initclass();
addmess((method)event_assist, "assist", A_CANT, 0);
}

void event_assist(t_event *x, void *b, long m, long a, char *s)
{
if (m==2) {
sprintf(s, "(float/list) signal values at time of trigger");
}
else if (m==1) {
switch (a) {
case 0: sprintf(s,"(signal) trigger input"); break;
default: sprintf(s, "(signal) value to sample"); break;
}
}

}

void event_dsp(t_event *x, t_signal **sp, short *count)
{
long *vector;
int i;
long size = (3 + x->numsignals) * sizeof(long);
long sigvs = (long)(sp[0]->s_n);

if (x->eventbuffer != NULL)
sysmem_freeptr(x->eventbuffer);
x->eventbuffersize = x->numsignals * (sigvs/2) + 1;
x->eventbuffer = (t_float *)sysmem_newptr(sizeof(t_float) * x->eventbuffersize);
x->eventreadindex = x->eventwriteindex = 0;
/* rationale for the above eventbuffer size:
the maximum number of events that can be detected in one signal vector is sigvs/2
with audio in interrupt on this size should be safe.
Need the +1 in there because with a vector size of 2, nothing ever gets output!
(see event_tick)
*/
vector = (long *)t_getbytes(size);
vector[0] = (long)x;
vector[1] = sigvs+1;
vector[2] = (long)(sp[0]->s_vec-1); //trigger
for (i=0;i<x->numsignals;i++)
vector[i+3] = (long)(sp[i+1]->s_vec-1);
dsp_addv(event_perform, 3 + x->numsignals, (void **)vector);
t_freebytes(vector,size);
}

t_int *event_perform(t_int *w)
{
t_event *x = (t_event *)(w[1]);
int n = (int)(w[2]);
t_float *trig = (t_float *)(w[3]);
t_float *in[EVENT_MAX_SIGNALS];

float prev = x->s_prev;
float thresh = x->s_thresh;
float current;
int numsignals = x->numsignals;
```

```
int i;

if (x->s_obj.z_disabled)
goto out;

for (i=0;i<numsignals;i++)
in[i] = (t_float *)(w[4+i]);
while (--n) {
    current = *++trig;
for (i=0;i<numsignals;i++)
++(in[i]);
    if (prev <= thresh && current > thresh)
event_new_output(x, numsignals, in);
     prev = current;
}
x->s_prev = prev;
out:
return (w+4+numsignals);
}

void event_new_output(t_event *x, long numsigs, t_float **sig)
{
long i;
for (i=0;i<numsigs;i++)
{
x->eventbuffer[x->eventwriteindex] = *sig[i];
if (++(x->eventwriteindex) >= x->eventbuffersize)
x->eventwriteindex = 0;
}
clock_delay(x->clock, 0.);
}

void event_tick(t_event *x)
{
long i;
t_atom *outputfloats = (t_atom *)sysmem_newptr(sizeof(t_atom)*x->numsignals);
while (x->eventreadindex != x->eventwriteindex)
{
for (i=0;i<x->numsignals;i++) {
SETFLOAT(outputfloats+i, x->eventbuffer[x->eventreadindex]);
if (++(x->eventreadindex) >= x->eventbuffersize)
x->eventreadindex = 0;
}
outlet_list(x->float_outlet, 0L, x->numsignals, outputfloats);
}
sysmem_freeptr(outputfloats);
}

void event_thresh(t_event *x, double f)
{
x->s_thresh = f;
}

void *event_new(t_symbol *s, short argc, t_atom *argv)
{
    t_event *x = (t_event *)newobject(event_class);
long numsignals = 1;
```

```
double thresh = 0.5;
int i;
if (argc > 0)
numsignals = atom_getlong(argv);
if (argc > 1)
thresh = atom_getfloat(argv+1);
    if ((numsignals > 0)&&(numsignals <= EVENT_MAX_SIGNALS))
     x->numsignals = numsignals;
    else
     x->numsignals = 1;
    dsp_setup((t_pxobject *)x, x->numsignals + 1);
    x->float_outlet = outlet_new((t_object *)x, 0L);
    x->s_thresh = thresh;
    x->s_prev = -1./0.0000000001;
x->clock = clock_new(x, (method)event_tick);
x->eventreadindex = 0;
x->eventwriteindex = 0;
x->eventbuffersize = 0;
x->eventbuffer = NULL;
    return (x);
}
```

VITA

Surname: Nevile ***Given Names:*** Benjamin Bowles

Place of Birth: Vancouver, British Columbia, Canada

Educational Institutions Attended

University of Manitoba	1992 to 1996

Degrees Awarded

B.Sc.(Hons)	University of Manitoba	1996

Honors and Awards

Harold R. Coish Memorial Scholarship	93-94
St. John's College Alumni Scholarship	94-95

Journal Publications

1. Randy Jones and Ben Nevile, "Creating Visual Music in Jitter: Approaches and Techniques", *Computer Music Journal*, Fall 2005.

Conference Publications

1. Ben Nevile, Peter Driessen, W. Andrew Schloss, "Radio drum gesture detection system using only sticks, antenna and computer with audio interface", *International Computer Music Conference,*, Fall 2006.

UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE

Title of Thesis:

Gesture analysis through a computer's audio interface: The Audio-Input Drum

Author: _____

BEN NEVILE
December 15, 2006

THESIS WITHHOLDING FORM

At our request, the commencement of the period for which the partial licence shall operate shall be delayed from December 15, 2006 for a period of at least six months.

_____

(Supervisor)

_____

(Department Chairman)

_____

(Dean of Graduate Studies)

_____

(Signature of Author)

_____

(Date)

Date Submitted to the Dean's Office: _____