THE ELECTRONIC TABLA

By
Ajay Kapur

A thesis submitted in partial fulfillment of the requirements for the degree of

Computer Science B.S.E.

Princeton University

May 6th, 2002

Princeton University

THE ELECTRONIC TABLA

By Ajay Kapur

I pledge my honor I did not violate the Honor Code in writing my senior thesis.

Thesis Advisor:	Perry R. Cook Department of Computer Science and Music
Second Reader:	Ben Shedd Department of Computer Science
Thesis Instructor:	Randy Wang Department of Computer Science

THE ELECTRONIC TABLA

Abstract

This paper describes the design of an Electronic Tabla controller. The Electronic Tabla (ETabla) triggers both sound and graphics simultaneously. It allows for a variety of traditional Tabla strokes and new performance techniques. Graphical feedback allows for artistical display and pedagogical feedback. This paper will describe the background of the Tabla, explain key concepts of digital signal processing and electronic music, and outline in detail the process of creating the Electronic Tabla.

Table of Contents

Acknowledgments

I would first like to thank my mentor, Professor Perry Cook, who introduced me to the field of Electronic Music, and showed me how to integrate my two passions (computers science and music) together into one. I would also like to thank Philip Davidson designing the graphic feedback system for the ETabla, and Georg Essl for designing the physical model for this project. Many thanks to Tae Hong Park who explained the details of the Fast Fourier Transform over and over again. Special thanks to Brad Alexander for helping design custom wood pieces for the ETabla. Special thanks to my aunt in Bombay, Asha Vohra, who organized the purchase of four sets of Tablas.

This project would have been impossible without Guru Rakesh Kumar Parihast and Manjul Bhargava, who both tutored me in traditional Tabla theory. I would also like to thank all the artists who participated in the world premiere concert of the ETabla.

Many thanks to all those who helped fund this project including: Chair Professor Scott Burnham of the Music Department, Chair David Dopkin of the Computer Science Department, Dean Peter Bogucki of the Engineering School, President Harold T. Shapiro, President Shirley M. Tilghman, Vice President of Student Life Janet S. Dickerson, Professor Perry Cook, and my parents.

I would also like to thank Stuart Rosse for encouraging me to pursue my musical inclinations and endeavors, making my “duty my desire”. I would like to thank my two roommates Adam Nemett and David Hittson for coming to my rescue through out this year. A special thanks to my sister, Asha Kapur, for listening as I shared my excitement, conflicts and frustrations. Finally, a loving thanks to my parents, Arun and Meera Kapur, who’s upbringing allowed me to stretch myself to follow through on my dreams.

Chapter

1

Introduction

Overview of the Electronic Tabla Project

Tablas are a pair of hand drums traditionally used to accompany North Indian vocal and instrumental music. The silver, larger drum (shown in Figure 1.1) is known as the *Bayan*. The smaller wooden drum is known as the *Dahina.*[1] The pitch can be tuned by manipulating the tension on the *pudi* (drumhead). The Bayan is tuned by adjusting the tightness of the top rim. The Dahina can be tuned similarly, as well as by adjusting the position of the cylindrical wooden pieces on the body of the drum. Tabla is unique because the drumheads have weights at the center made of a paste of iron oxide, charcoal, starch, and gum (round, black spots shown in the Figure 1.1).[2] Also, the Tabla makes a myriad of different sounds by the many different ways it is stroked. These strokes follow a tradition which has been passed on from generation to generation, from *guru* (teacher, master) to *shikshak* (student) in the country of India. The combination of the "weighting" of the drum-head, and the variety of strokes by which the Tabla can be played, gives the drum a complexity that makes it a challenging controller to create, as well as a challenging sound to simulate.

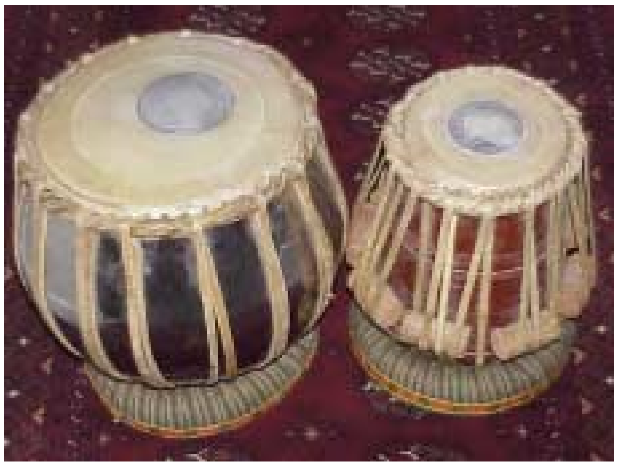


Figure 1.1 Picture showing North Indian Tabla. The *Bayan* is the silver drum on the left. The *Dahina* is the wooden drum on the right.

Project Description

The purpose of this project is to use technology to create a real-time instrument that models the Tabla. This Electronic Tabla (known as the ETabla) has digitizing sensors, custom positioned to traditional Tabla technique, which converts finger strikes and hand slaps to binary code which computers can understand. These signals are then used to trigger real-time sound and graphics.

Project Goals

The motivations and goals for creating the Electronic Tabla are to:

1. Develop a controller which can simulate traditional North Indian Tabla strokes
2. Facilitate the fusion between North Indian Classical Music and modern electronic music
3. Expedite the learning process for beginner Tabla players by easing the execution of basic Tabla sounds and rhythms
4. Widen the number of sounds available within the repertoire of expert Tabla players
5. Make it easier to tune the Tabla to the desired pitch
6. Create audio and visual experiences that express the feelings of the performer and enamors the audience
7. Increase the popularity of the North Indian Drum

Project Overview

In this report, I will present:

- An overview of the Tabla, including its evolution with technology, and how it is traditionally played.
- An overview of the technology used to create the Electronic Tabla controller
- The creation process and details of the MIDI Tabla Controller
- The sound analysis of the traditional Tabla
- The various models used to simulate the sound of the Tabla
- The creation and explanation of the graphic feedback system
- The concert which introduces the Electronic Tabla to the campus
- Other applications of the technology used to produce the Electronic Tabla

The ETabla Team

The Electronic Tabla team consists of five key players who have helped make the instrument a success. Professor Perry Cook of the Computer Science and Music department has brought his past experience of making electronic instruments. Philip Davidson, who is an undergraduate student in the Computer Science program, has brought his expertise in creating real-time graphic feedback. Georg Essl, a PhD Computer Science student has brought his knowledge in physically modeling different sounds. Brad Alexander is a millwork foreman at County Cabinet Shop, Inc., and helped design custom wood pieces for our project. I, Ajay Kapur, am a musician getting a degree in Computer Science, who has led and coordinated all these talents together to create the Electronic Tabla as my senior thesis project.

Chapter

2

The Tabla

Origin, Evolution, & Tradition

Evolution of the Tabla with Technology

There are a few accounts for the origin of the Tabla. A mythological account reads:

> *"Once, a long time ago, during the transitional period between two Ages… people took to uncivilized ways … ruled by lust and greed [as they] behaved in angry and jealous ways, [while] demons, [and] evil spirits… swarmed the earth. Seeing this plight, Indra* (The Hindu God of thunder and storms) *and other Gods approached God Brahma* (God of creation) *and requested him to give the people a Krindaniyaka* (toy) *… which could not only be seen, but heard, … [to create] a diversion, so that people would give up their bad ways."*[1]

One of the Krindaniyakas, which Brahma gave to humans was the Tabla. Other legends state that the Tabla was created in the 18th Century by Sidhar Khan Dhari, a famous *Pakhawaj* player. Pakhawaj is a genre of Indian drum defined by a barrel with drumheads on either side. The *Mrindangam*, shown in Figure 2.1, is one drum in this family of drums. It was said that Sidhar Khan provoked an angry dispute after losing a music contest and his

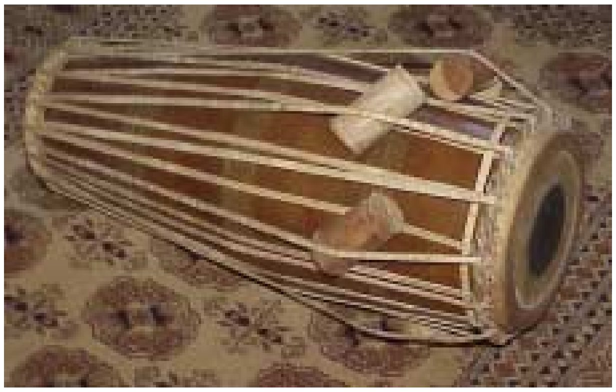


Figure 2.1 Picture showing a Mridangam, a drum of the Pakhawaj family.

Pakhawaj was chopped in half by a sword. Thus, the first Tabla was created accidentally.[3]

Some Tablas were created out of clay, others out of wood. As technology for producing metal alloys evolved, the Bayan started to be molded out of brass and steel.[4]

As the popularity of the Tabla spread to the western hemisphere, nearly coincident with emergence of the personal computer, scientists began to combine the Tabla with computers. In 1992, James Kippen created software which allowed a user to input a traditional Tabla rhythmic pattern, which the computer would then use to synthesize an improvised pattern that followed traditional rules for variation.[5] In 1998, Mathew Wright and David Wessel of University of California Berkeley, aimed to achieve a similar goal, with a real time interface and unique data structure. They successfully created software that generated *"free and unconstrained"* music material, which could fit into a given traditional rhythmic structure.[6] Meanwhile, Talvin Singh created a direct input from his Tabla to computer effects, achieving sound manipulations in an invention he calls "Tablatronics". [7] [8]

Now, our team has created a Tabla controller that is modeled to the playing style of North Indian classical traditions, and which outputs computer-generated sound and graphics.

Traditional Tabla Strokes

It is important to understand the traditional playing style of the Tabla to see how our controller models its hand movement. Below, in Figure 2.2, is a picture explaining the names of the different parts of the Tabla *pudi* (drum head).

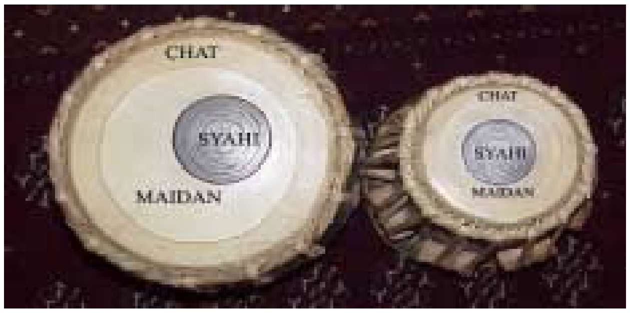


Figure 2.2 Picture showing parts of the Tabla pudi.

Bayan Strokes

There are two basic strokes played on the Bayan. The *Ka* stroke is executed by slapping the flat left hand down on the Bayan as shown in Figure 2.3 (a). Notice the tips of the fingers extend from the *maidan* through to the *chat* and over the edge of the drum. The slapping hand remains on the drum after it is struck to kill all resonance, before it is released away. The *Ga* stroke, shown in Figure 2.3 (b), is executed by striking the *maidan* directly above the *syahi* with the middle and index fingers of the left hand. When the fingers strike, they immediately release away from the drum, to let the Bayan resonate with sound. The heel of the left hand controls the pitch of the *Ga* stroke, as shown in Figure 2.3 (c). It controls the pitch at the attack of the stroke, and can also bend the pitch while the drum is resonating. Pitch is controlled by two variables of the heel of the hand: force on to the *pudi*, and the position on the *pudi* from the edge of the *maidan* and *syahi* to the center of the *syahi*. The greater the force on the *pudi*, the higher the pitch. The closer to the center of the *syahi*, the higher the pitch. [1]

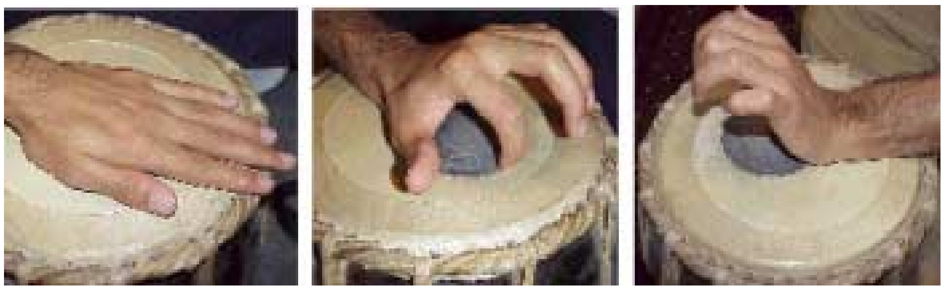
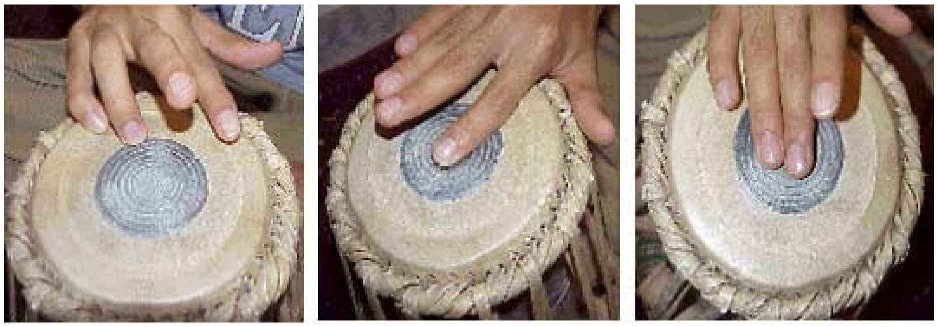

(a) (b) (c)

Figure 2.3 Pictures showing traditional strokes played on Bayan

Dahina Strokes

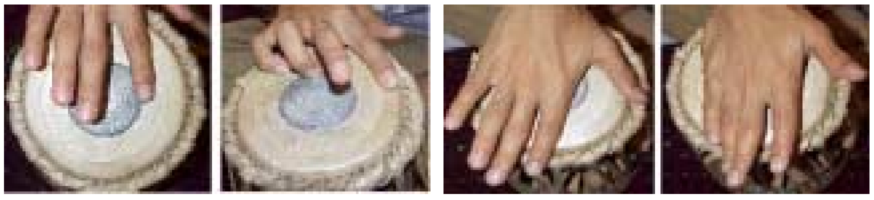
There are six basic strokes played on the Dahina. The *Na* stroke, shown in Figure 2.4 (a), is executed by lightly pressing down the pinky finger of the right hand between the *chat* and the *maidan*, and lightly pressing the ring finger down between the *syahi* and the *maidan,* in order to mute the sound of the drum. Then one strikes the chat with the index finger and quickly releases it so the sound of the drum resonates. The *Ta* stroke is executed by striking the index finger of the right hand at the center of the *syahi*, as shown in Figure 2.4 (b). The finger is held there before release so there is no resonance, creating a damped sound. The *Ti* stroke, shown in Figure 2.4 (c), is similar to *Ta* except the middle and ring finger of the right hand strike the center of the *syahi*. This stroke does not resonate and creates a damped sound.



(a) **(b)** **(c)**

Figure 2.4 Picture showing *Na*, *Ta*, & *Ti* strokes played on the Dahina.

The *Tu* stroke is executed by striking the *maidan* with the index finger of the right hand and quickly releasing, as shown in Figure 2.5 (a). This stroke resonates the most because the pinky and ring fingers are not muting the *pudi*.[1] The *Tit* stroke, shown in Figure 2.5 (b), is executed similar to *Na*, by lightly pressing the pinky finger of the right hand down between the *chat* and the *maidan*, and lightly pressing the ring finger down between the *syahi* and the *maidan*. The index finger now strikes the *chat*, quickly releasing to let it resonate. The index finger strike on the *chat* is further away from the pinky and ring finger than it is on the *Na* stroke. *Tira* is a combination of two strokes on the Dahina, which explains the two syllables of the stroke. This stroke is shown in Figure 2.5 (c) and Figure 2.5 (d). It is executed by shifting the entire right hand from one side of the drum to the other. It creates a damped sound at each strike.[9]



(a) **(b)** **(c)** **(d)**

Figure 2.5 Picture showing *Tu*, *Tit*, & *Tira* strokes played on the Dahina.

Combination Strokes

Specific names are given to strokes that are produced by both hands simultaneously on the Dahina and Bayan. *Dha* refers to a *Ga* stroke combined with a *Na* stroke. *Dhin* refers to a *Ga* stroke combined with a resonating *Ta* stroke.[1] *Tin* refers to a *Ka* stroke combined with a *Na* stroke.[9]

Traditional use of the Tabla in Indian Music

Music is a central component of many functions throughout India, such as birth, engagements, weddings, and funerals. Indian Music consists of four main styles: folk, tribal, pop, and classical. The Tabla is used in all of these forms of music. Indian folk and tribal music are both played in villages all around India. Cheaper versions of Tablas are created out of commonly available materials.[10] Music created for films is the most popular in India, similar to music in America which is broadcasted on the "top 40" and "MTV". The Tabla is used in a myriad of songs for these three-hour films which are similar to American musicals. There are two systems of Indian classical music: Hindusthani music from the North, and Carnatic music for the south. The Tabla outlines the rhythmic structure in Hindusthani music, while the mridangam, shown in Figure 2.1, outlines the rhythmic structure of Carnatic music.[11] [12]

Hindustani Tabla Theory

Musical enhancement is the major role of the Tabla in North Indian classical music. *Theka*, which literally means "support", is the Indian word for simple accompaniment performed by a Tabla player. The importance of the *theka* underscores the role of the Tabla player as timekeeper. An even more specific definition of *theka* is the conventionally accepted pattern of *bols* which define a *tal*. The word *tal* literally means clap, for the clapping of hands is one of the oldest forms of rhythmic accompaniment.[1]

The most fundamental unit of this rhythmic system is the *matra*, which translates to "beat". In many cases the *matra* is just a single stroke. Just as sixteenth, or eighth notes maybe strung together to make a single beat, so too may several strokes of Tabla be strung together to have the value of one *matra*. The next higher level of structure is *vibhag*, which translates to “measure” or “bar”. These measures may be as short as one beat or longer than five. Usually, however, there are two, three, or four *matras* in length. These *vibhags* are described in waves or claps. A *vibhag* which is signified by a clap of the hands is said to be *bhari* or *tali.* Conversely, a *vibhag* which is signified by a wave of the hand is said to be *khali.*[1]

In the common *tal* known as *Tin Taal (*which translates to “three claps”), there are 16 *matras*, divided into four *vibhags.* Its clapping arrangement is arranged:

Clap, 2, 3, 4,

Clap, 2, 3, 4,

Wave, 2, 3, 4,

Clap, 2, 3, 4,

The third line is a *khali vibhag,* where as the other three lines are *bhari vibhags.*

In performance, the cycle of sixteen beats is repeated over and over again. This cycle, known as *avartan,* refers to the highest level of conceptual rhythmic structure. The repetition of the cycle gives special significance to the first beat. This beat, known as *sam,* is a point of convergence between the Tabla player and the other musicians. Whenever a cadence is indicated it will usually end on the *sam*. This means that the *sam* may be thought of as both the beginning of some structures as well as the ending of others.[1]

The mnemonic syllables, known as *bol*, represent the various strokes of the Tabla, which are described earlier in this chapter. The cycle of 16 *bols* that create *Tin Taal* is written below:

Dha, Dhin, Dhin, Dha

Dha, Dhin, Dhin, Dha

Dha, Tin, Tin, Ta

Ta, Dhin, Dhin, Dha [13]

Bols are useful for two reasons; First, the *bol* allows the musician to remember complicated fixed compositions. Second, the musician uses the *bol* to create the mental permutations of a *theka* into an improvised passage.

If a musician were to play a basic unadorned *theka,* it would be excruciatingly dull. Advanced Tabla players improvise with dynamics, modulation, and ornamentation to add beauty and life to the *theka*. Adding different embellishments and variations to the music is a concept defined as *prakar*.[1]

These concepts and techniques of traditional Hindustani music are considered in some compositions using the ETabla discussed in Chapter 8, as well as the design of the ETabla discussed in Chapter 4.

Chapter

3

The Technology

Force Sensing Resistors, Basic Stamp, & MIDI

Music Controller

This chapter describes the technology used to create a musical controller. A controller is a device made of different sensors which measure human interaction and convert instances into the digital realm. A mouse is a controller which uses an infrared LED and sensor to convert hand movement into *x* and *y* coordinates on a computer screen.[14] A musical controller takes input from a musician, such as rhythm and pitch, to trigger recorded or physically modeled sound using a computer. This process is displayed in Figure 3.1. The ETabla is primarily concerned with capturing rhythmic impulse from the performers finger taps, as well as pitch and type of stroke.

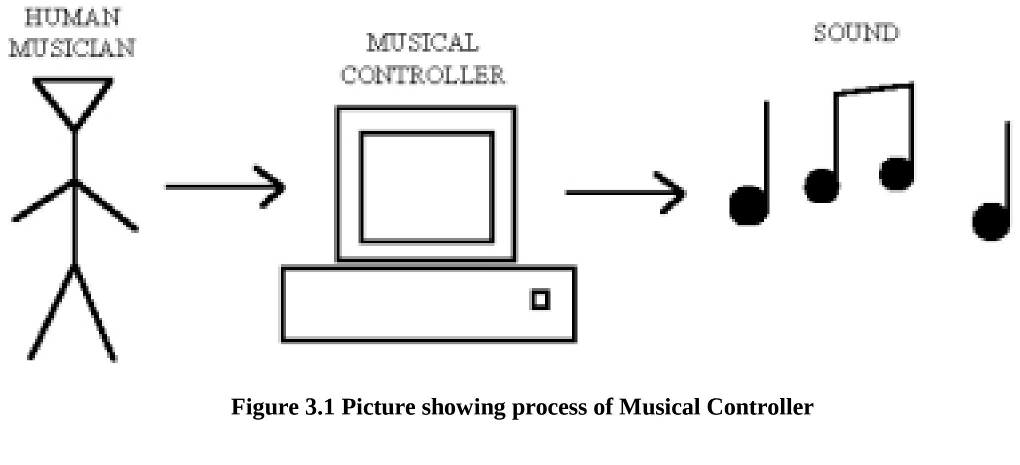


Figure 3.1 Picture showing process of Musical Controller

Force Sensing Resistors

Force sensing resistors (FSRs) are used to digitize the taps of the performer. FSRs are manufactured by Interlink Electronics and can be purchased at their online store.[15] These sensors use the electrical property of resistance to measure the force (or pressure) exerted by a user. They essentially are force to resistance transducers. The more pressure exerted, the lower the resistance drops. FSRs are made of two main parts: a resistive material applied to a piece of film, and a set of digitizing contacts applied to another film. This configuration is shown in Figure 3.2. The resistive material creates an electrical path between a set of two conductors. When force is applied, conductivity increases as the connection between the conductors is improved.[16] Experiments with FSRs, explained in detail in Appendix A, show that conductivity is a linear function of force.

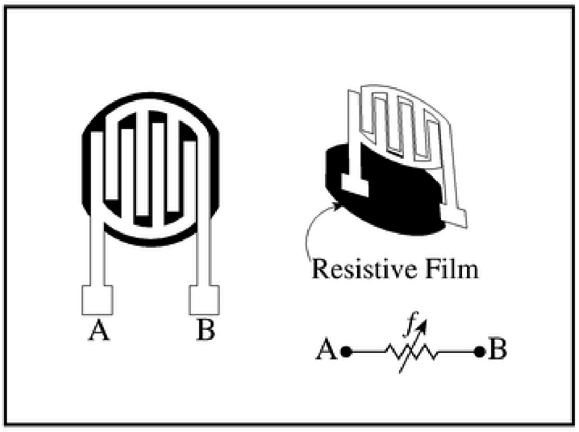
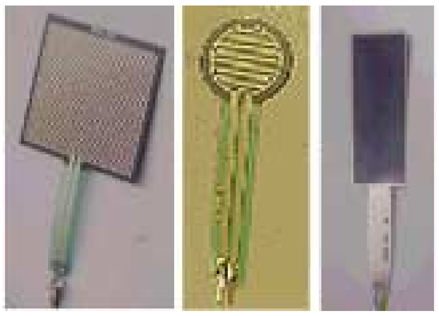


Figure 3.2 Diagram showing configuration of Force Sensing Resistors[16]

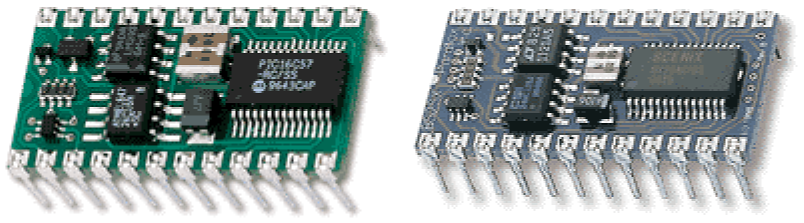
The ETabla uses three types of FSRs. Square FSRs, shown in Figure 3.3 (a), and small FSRs, shown in Figure 3.3 (b), measure only force. Long FSRs, shown in Figure 3.3 (c), measure force as well as position on the vertical axis. Force measurements will be used to control velocity (volume). Position measurements will be used to control pitch and resonance of different finger strikes. Look at Appendix A for more details on FSRs.



(a) (b) (c)
Figure 3.3 Pictures of the three types of FSRs used to create ETabla

Basic Stamp

The Basic Stamp is a programmable micro controller, developed by Parallax, Inc. There are currently five types of BASIC Stamps: BASIC Stamp I, BASIC Stamp II, BASIC Stamp IIe, BASIC Stamp IIsx, and BASIC Stamp IIp. The ETabla was first developed using the BASIC Stamp II (Shown in Figure 3.4 (a)), and then upgraded to the BASIC Stamp IIsx (Shown in Figure 3.4(b)), which has a faster processing speed.



(a) (b)

Figure 3.4 Pictures showing BASIC Stamp II and BASIC Stamp IIsx [17]

The ETabla uses the BASIC Stamp II Carrier Board, which can accommodate the BASIC Stamp II, and the BASIC Stamp IIsx. FSRs, LEDs, resistors, capacitors, and other gizmos are wired together onto this carrier board. Detailed design schematics and circuitry of the ETabla will be discussed in Chapter 4.

The BASIC Stamp is programmed by software provided by Parallax, for Windows. The programming language is PBASIC (Parallax BASIC) which is based off the BASIC programming language. There are several versions of PBASIC. The ETabla used versions PBASIC2 and PBASIC2sx for programming the BASIC Stamp II and BASIC Stamp IIsx respectively.

Code is transferred from the computer to the powered BASIC Stamp via a serial port on the carrier board. The code is stored in the EEPROM memory after being tokenized. Programming elements, such as constants, comments, and variable names, are not stored in the BASIC Stamp, so descriptive names and comments are included in PBASIC code for the ETabla.

The BASIC Stamp II only has room for about 500 lines of code, executed at 4000 instructions per second, whereas the BASIC Stamp IIsx has room for 4000 lines of code, executed at 10,000 instructions per second. Thus the BASIC Stamp IIsx executes 2.5 times as fast to time sensitive commands. This is why an upgrade was made.[17]

Pin Descriptions:

PIN	NAME	DESCRIPTION
1	SOUT	Serial Output
2	SIN	Serial Input
3	ATN	Attention
4	VSS	System Ground
5-20	P0-P15	General Purpose I/O
21	VDD	5 volt Input/Output
22	RES	Reset
23	VSS	System Ground
24	VIN	Unregulated Power

Table 3.1 Table describing pins of the BASIC Stamp II and BASIC Stamp IIsx [17]

Both the BASIC Stamp II and the BASIC Stamp IIsx have 16 I/O pins, and two dedicated serial port pins. The serial input is the SIN pin and the serial output is the SOUT pin. (Shown in Table 3.1)

The BASIC Stamp runs on 5 to 15 volts DC. It has a feature of a 5-volt regulator, which converts input from 6 to 15 volts (at the VIN pin) down to 5 volts to run the components. +5 volts are available to use on the VDD pin. The VSS pin is the ground pin. The ETabla uses a 9-volt battery for power which is directly connected to the VIN and VSS pins.

The RES pin is the internal reset pin, which is normally high (+5 volts) when BASIC Stamp is running its program. It turns low, when power supply drops below 4 volts, to sleep the BASIC Stamp safely. When re-powered, the BASIC Stamp starts at the first of its stored program. The ATN pin has an inverse relationship with the RES pin. It normally is low when the RES pin is high, and the BASIC Stamp is running properly. When the ATN pin is driven high, it forces the RES low, putting the BASIC Stamp to sleep safely.[17]

<u>Memory:</u>

The BASIC Stamp II has 2048 bytes of program storage, while the BASIC Stamp IIsx, have 16,384 bytes, separated into 8 pages of 2048 bytes. Each command written in PBASIC takes a variable amount of space. Most commands take 2 to 4 bytes of memory space. Tens of bytes or more are taken up by commands such as SERIN, SEROUT, LOOKUP and LOOKDOWN, which have many arguments. While editing code on a Windows machine, CTRL-M shows a memory map of how space in the EEPROM is used.[17]

MIDI

MIDI is short for Musical Instrument Digital Interface. It is a communication protocol, which allows electronic instruments (such as keyboards, synthesizers, and the ETabla) to connect and interact with each other. Thus taps on the ETabla controller can trigger sounds on a keyboard remotely. In this case, the ETabla would take information about a musical note, such as pitch, volume, start time, stop time, and convert it to MIDI. The protocol would then be sent out to a keyboard which has a changeable bank of sounds, and the MIDI information is opened to create a noise in real-time. This allows one controller to generate the sounds of hundreds of instruments! [18]

Starting in 1983, MIDI was developed in cooperation with the major electronic instrument companies such as Roland, Yamaha, and Korg. The companies created a standard interface, to try to generate more sales. Since than, the protocol has evolved to fit the needs of professional musicians, as larger amounts of controllers and sounds were created.

MIDI is transmitted at 31,250 bits per second. Each message has one start bit, eight data bits, and one end bit, which means the maximum transmission rate would be 3215 bytes per second. When the first bit is set to 1, the byte is a status byte. Status byte denotes MIDI commands such as NOTE ON, NOTE OFF, and CONTROL CHANGE, and communicates which channel (0-15) to send information. The status byte also determines the length of the message, which are generally one, two, or three bytes in length. An example of a common message is illustrated below:

10010000	00111100	01000000
Note On / Channel 0	Note #60	Velocity = 64

The NOTE ON command will trigger a MIDI device to turn on a sound. The pitch byte will tell the device to play Note 60, which is middle C on a piano sound bank. The velocity byte will tell the device how loud to play the note.[19]

On a standard MIDI device there are three five-pin ports (IN, OUT, THRU) that transmit and receive MIDI information. The IN port receives and processes MIDI commands, while the OUT transmits it. The THRU port, receives and processes MIDI information and transmits the same message through the OUT port.[18] The ETabla's BASIC Stamp converts taps on the force sensing resistors to MIDI protocol which is sent through a MIDI OUT port. Specific MIDI messages that are triggered will be discussed in the detailed design schematic in Chapter 4.

Chapter

4

The MIDI Tabla Controller

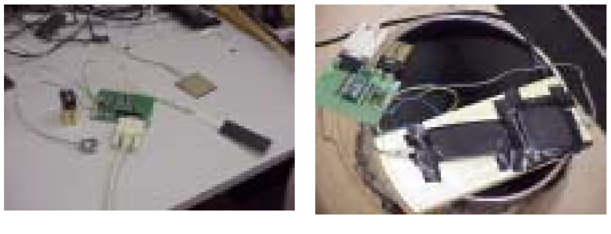
Chronological Design Schematics

This chapter describes a detailed design schematic for the ETabla. The chapter is organized by time, showing progress at key milestones of accomplishment. Thus, it outlines the process of creating the MIDI Tabla controller. There are two controllers which make up the ETabla: EDahina and EBayan. Each section will describe the progress for both of these controllers.

November 29th, 2001

EBayan 1.0:

The EBayan was born on a piece of wood. Chapter 2 explained how the Bayan has two main strokes: *Ga* and *Ka*. The left hand's middle and index finger tap out *Ga* strokes, while the heel of the hand changes the pitch by force and position. Chapter 3 explained how square FSRs measure force, and how long FSRs measure force and position. Thus, we used one square FSR and one long FSR to try and simulate the *Ga* stroke, as shown in Figure 4.1 (a). Figure 4.1 (b) shows how these FSRs were positioned on a slab of wood, as the beginning of the EBayan design. The square FSR measures velocity of the middle and index finger



(a) (b)

Figure 4.1 Pictures showing EBayan's birth on a slab of wood.

striking, while the long FSR measures force and position of the heel of the left hand, to determine pitch. When the square FSR was tapped once and released, a MIDI message was encoded with a *Ga* NOTE ON, with the pitch byte determined by the long FSR and the velocity determined by the force measured by the square FSR. To simulate a *Ka* stroke, it was simply decided that when the square FSR was held down by the entire slap of the hand, then a *Ka* NOTE ON would be triggered, with the velocity determined by force on the square FSR.

EBayan's Initial BASIC Stamp Carrier Board:

Figure 4.2 and Figure 4.3 describe the circuit designs for the long FSR and the square FSR respectively, showing connections to the different pins of the BASIC Stamp and to the analog-to-digital converter (ADC). The ADC used is Linear Technology's LTC 1298 12-bit ADC Chip which comes in Parallax's AppKit.[20] The circuit design used to wire the ADC is shown in Figure 4.4 (a). Figure 4.4 (b) shows the connections made to create the MIDI OUT port.

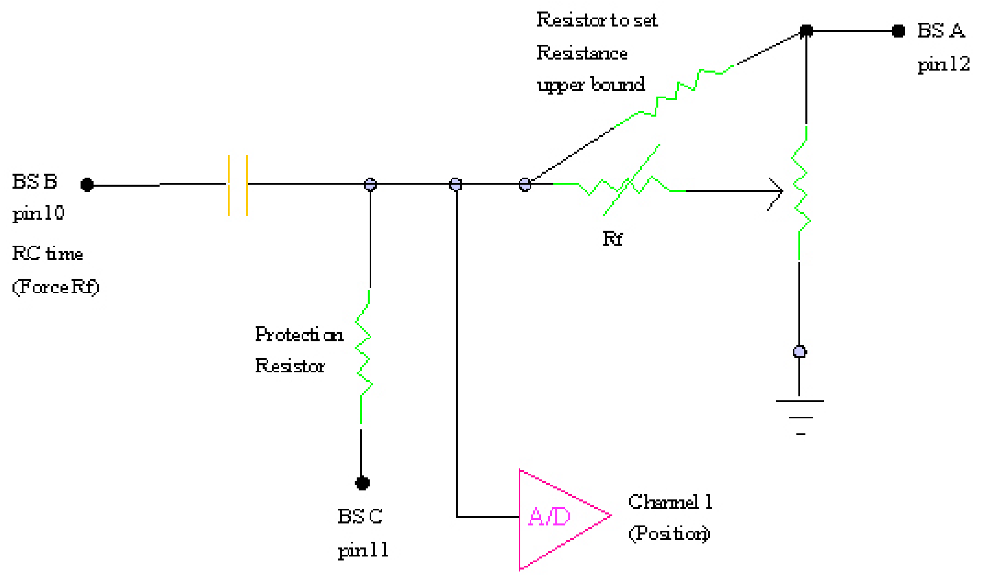


Figure 4.2 Drawings showing circuitry to connect the long FSR on the EBayan

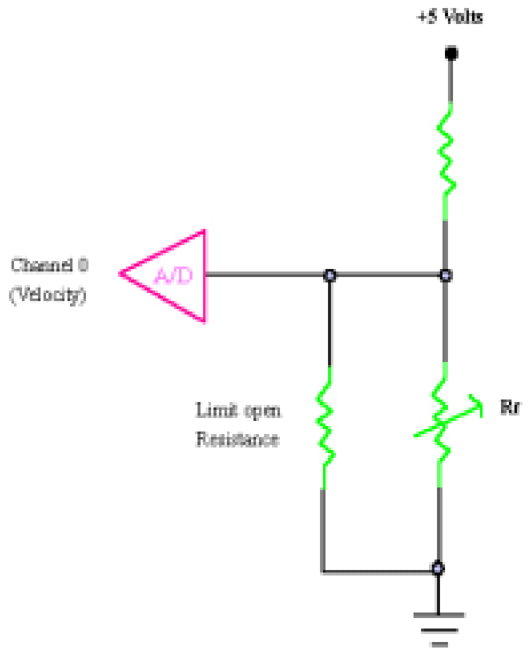
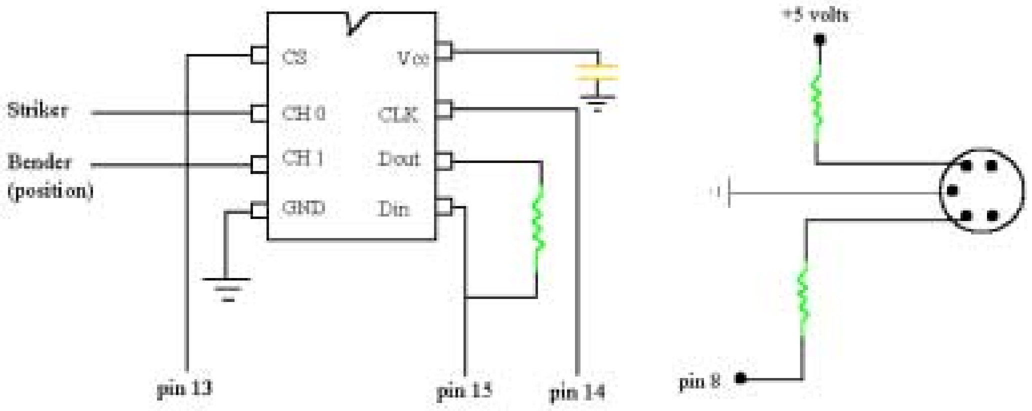


Figure 4.3 Drawing showing circuitry of first square FSR on the EBayan



(a) **(b)**

Figure 4.4 Drawing (a) shows the circuitry design of A-to-D Converter for the EBayan. Drawing (b) shows the circuitry for the MIDI OUT port on the EBayan.

<u>EBayan 1.0 Design Problems:</u>

The problem with this schematic was that the square FSR could not distinguish between a hand slapping *Ka* and fingers striking *Ga*. The thought was that when the square FSR was held down a *Ka* command would be sent through the MIDI OUT port. However, this did not work out, because fast finger strikes ended up sending *Ka* commands, and a *Ga* command would be sent out at the beginning of every *Ka* stroke. Figure 4.5 shows the EBayan at this point.

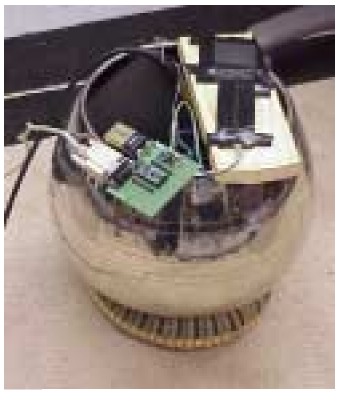


Figure 4.5 EBayan 1.0

December 15th, 2001

<u>EBayan 2.0: Addition of a Square FSR</u>

A solution to the problem described above was to add another square FSR above the existing square FSR that can only be reached by fingers when the left hand slaps the *Ka* stroke. Figure 4.6 shows a layout of EBayan 2.0. The top square FSR (referred to as Slapper) is used to capture *Ka* stroke events, when a player slaps down with their left hand. If it receives a signal, then the other two FSRs are ignored. The square FSR in the middle (referred to as Striker), captures *Ga* stroke events, when struck by the middle and index finger of the left hand. The long FSR (referred to as Bender) controls the pitch of the *Ga* stroke events as it did in EBayan 1.0.

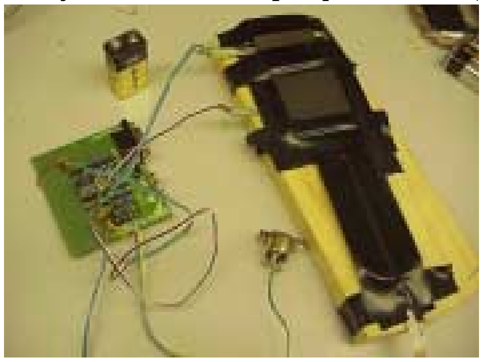


Figure 4.6 Picture showing EBayan 2.0 FSR layout

The Slapper FSR is different than the Striker FSR, because it uses an RC time circuit to trigger events, rather than going through a channel on the ADC. The force on the Bender FSR is being captured by a RC time circuit, while the position is going through channel 1 of the ADC. The Slapper FSR is attached to pin 7 as shown in Figure 4.7. Figure 4.8 shows the picture of the BASIC Stamp Carrier Board for the EBayan at this stage.

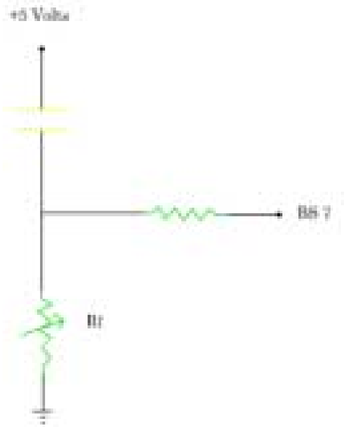


Diagram showing RC Time circuit of the Slapper FSR

Figure 4.8: Picture showing BASIC Stamp Carrier Board for EBayan 2.0

January 15th, 2002

EDahina 1.0:

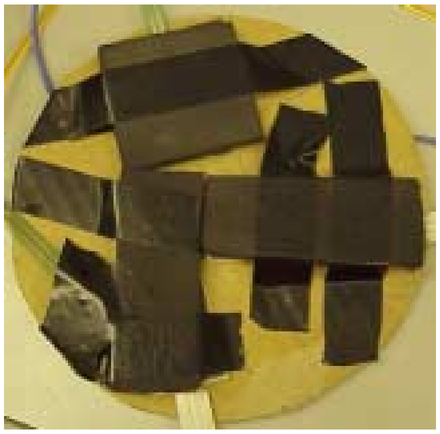


Figure 4.9 Picture showing EDahina FSR layout

To implement the EDahina, four FSRs were used: two long FSRs, one square FSR, and one small FSR attached to a circular piece cardboard. Figure 4.9 shows a layout of these FSRs. The small FSR triggers a *Tit* stroke event. It measures the velocity of the index finger's strike. The square FSR triggers a *Tira* stroke event. It measures the velocity of the hand slapping the top of the drum. If the *Tira* FSR is struck, all other FSRs are ignored. If the *Tit* FSR is struck, both long FSRs are ignored. The rightmost long FSR in Figure 4.9, is the ring finger FSR, and the leftmost long FSR is the index finger FSR. If there is a little force on the ring finger FSR (modeling a mute), and the index finger FSR is struck at the edge of circle, a *Na* stroke is triggered. If the index finger FSR is struck near the center of the circle, a *Ta* stroke is triggered. If there is no force on the ring finger FSR, and the index finger FSR is stuck, then a *Tu* stroke is triggered. When the ring finger FSR is struck with enough force, and not held down, then a *Ti* stroke is triggered. Thus the modeling was completed for every stroke that was discussed in Chapter 2.

For this drum, two analog-to-digital converters were used. Each ADC had one square FSR going into channel 0, and position data coming in from one long FSR going into channel 1. Table 4.1 describes how the variables are collected.

FSR	VARIABLE	COLLECTION
Ring Finger (long)	Velocity	RC Time
Index Finger (long)	Velocity	RC Time
Tira (square)	Velocity	Channel 0 on ADC A
Ring Finger (long)	Position	Channel 1 on ADC A
Tit (square)	Velocity	Channel 0 on ADC B
Index Finger (long)	Position	Channel 1 on ADC B

Table 4.1: Table describing how variables are collected for EBayan 1.0

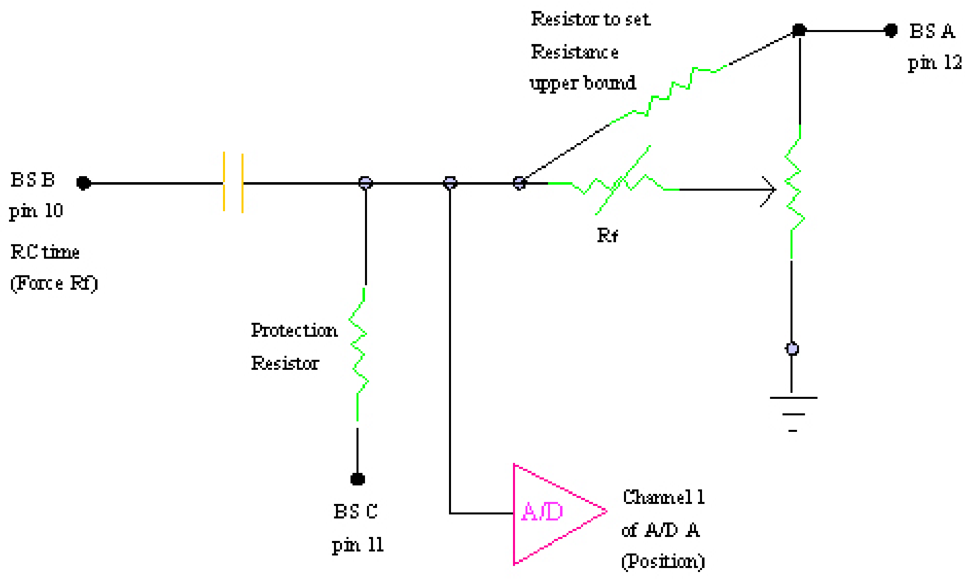


Figure 4.10: Drawing describing circuit design of the EDahina's Ring Finger FSR

Figure 4.10 shows design circuitry used to obtain velocity and position of the Ring Finger FSR. Figure 4.11 shows design circuitry to obtain velocity and position of the Index Finger FSR. Figure 4.12 (a) shows the circuit design to obtain velocity of Tira FSR, and Figure 4.12 (b) shows circuit design to obtain velocity of Tit FSR. Details about ADC A and ADC B are shown in Figure 4.13 (a) and (b) respectively. The MIDI OUT port on the EDahina is wired in the same way as the EBayan. Figure 4.14 shows a picture of the BASIC Stamp Carrier Board for the EDahina 1.0.

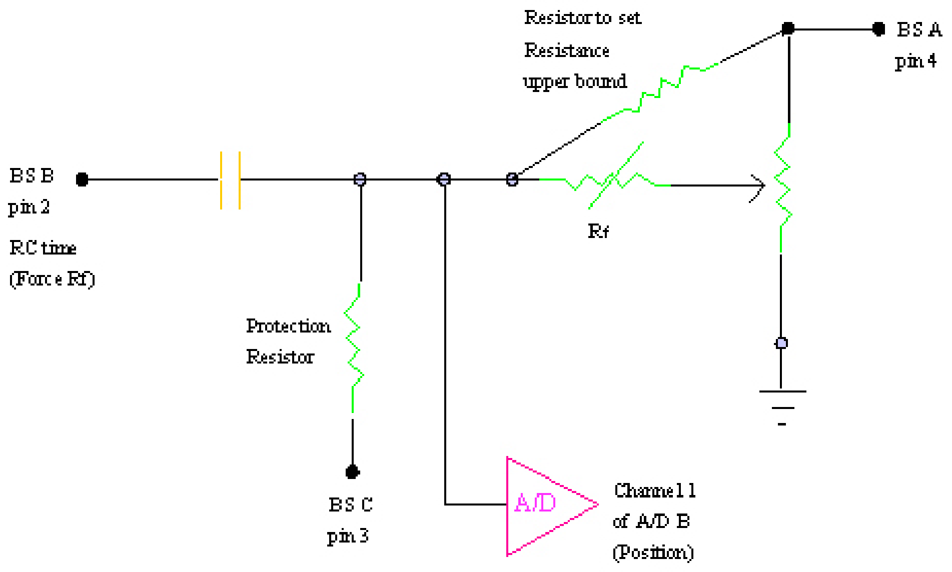


Figure 4.11: Drawing describing circuit design of the EDahina's Index Finger FSR

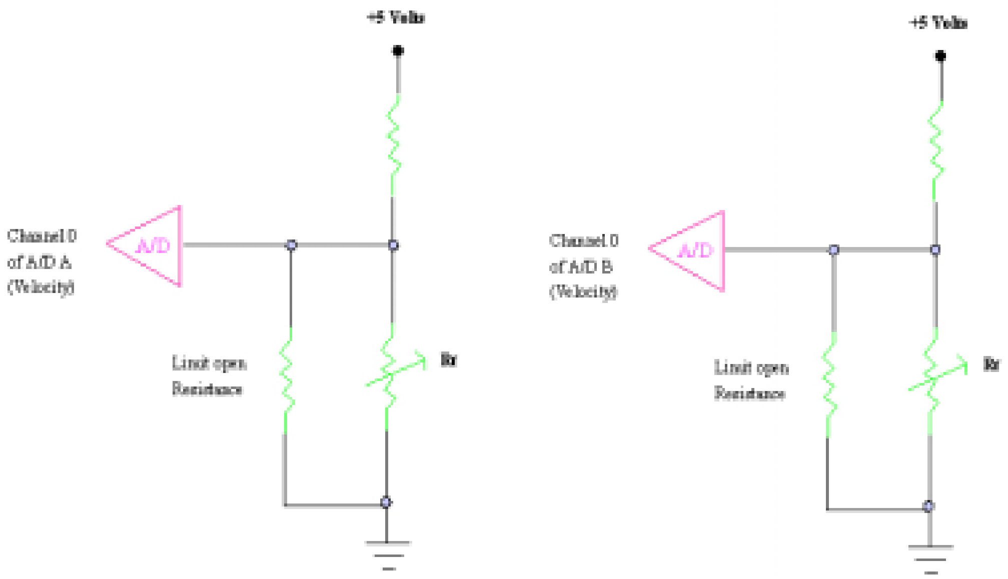
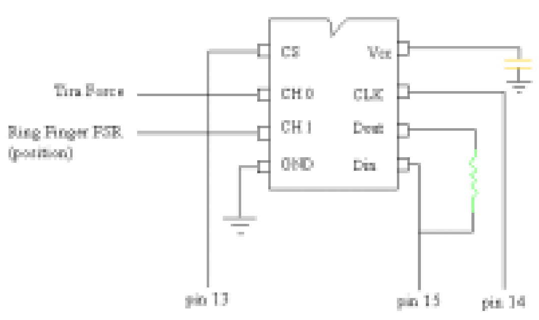
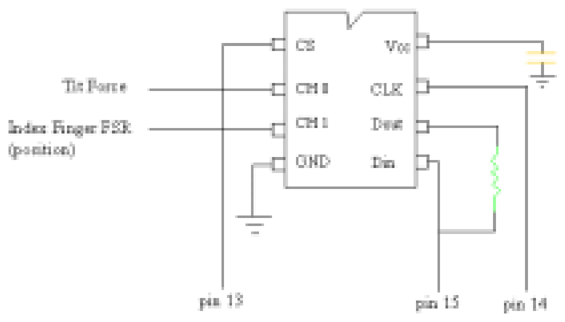


Figure 4.12: Drawing (a) shows the circuitry design of Tira FSR for the EDahina. Drawing (b) shows the circuitry design of the Tit FSR for the EDahina.





(a) **(b)**

Figure 4.13: Drawing (a) shows the circuitry design of A-to-D Converter A for the EDahina. Drawing (b) shows the circuitry design of A-to-D Converter B for the EDahina.

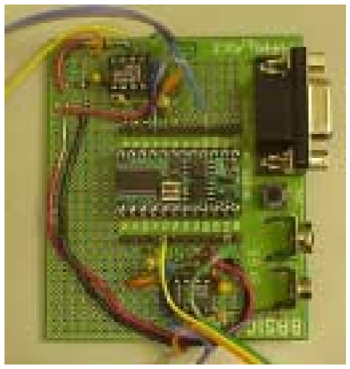


Figure 4.14: Picture showing BASIC Stamp Carrier Board fo EDahina 1.0

ETabla 1.0: First User Test

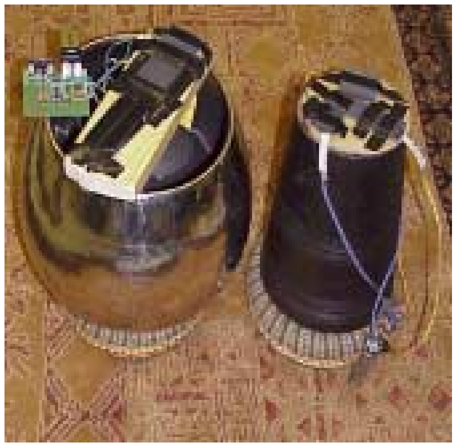


Figure 4.15: The Electronic Tabla Controller

The ETabla 1.0 is the combination of the EBayan 2.0 and EDahina 1.0. Figure 4.15 shows a picture of the controllers in their constructed encasements for the first time. Manjul Bhargava, a musician who has been playing Tabla for 10 years, tested the ETabla at this point. He successfully was able to trigger all the traditional Tabla strokes discussed in Chapter 2, but with a margin of error. He hypothesized that the errors occurred because the ETabla 1.0 was not stable, as the slab of wood and piece of cardboard were simply sitting on top of a Tabla shell, without any support.

March 7th, 2002

ETabla 2.0: Constructing the ETabla Encasement

One of the goals of the project is to make the ETabla's encasement look and feel like a real Tabla. To achieve this, a professional millwork foreman, named Brad Alexander was hired to help create custom wood pieces for the ETabla. Brad runs County Cabinet Shop, Inc. in Princeton, New Jersey.

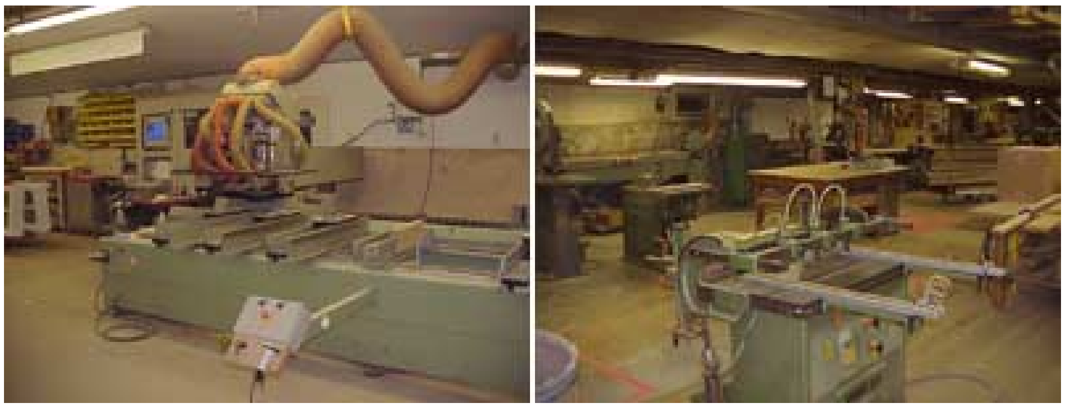
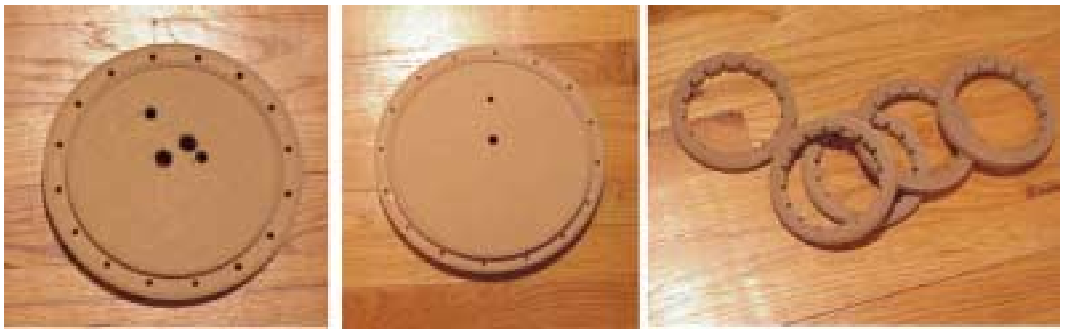
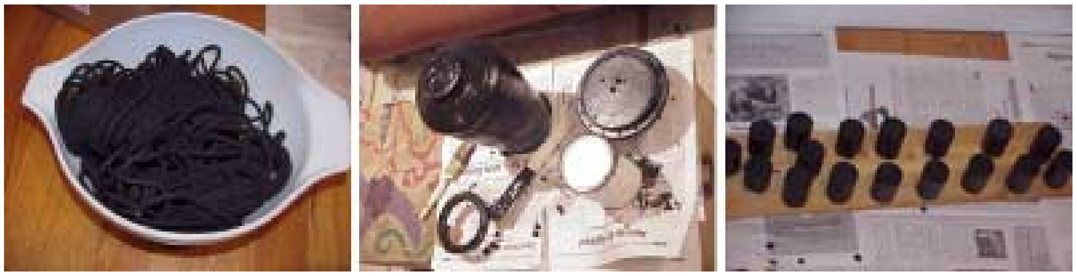


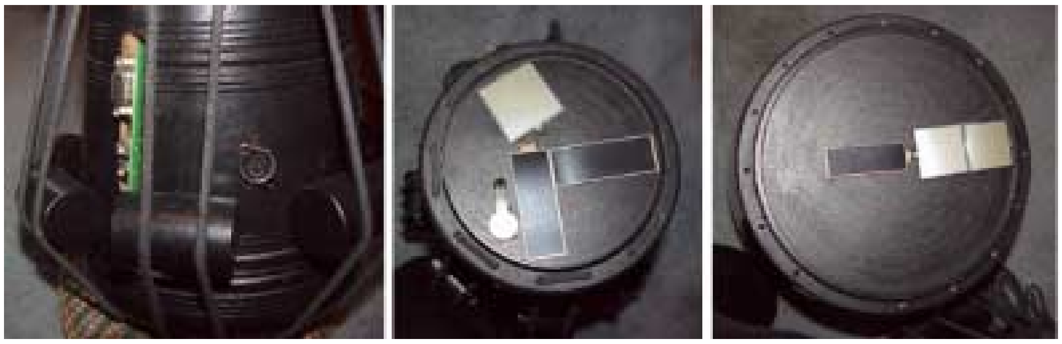
Figure 4.16: Pictures showing professional Wood Working Machinery at County Cabinet Shop, Inc.

The starting point was to draw designs for the custom pieces on AutoCad software. The AutoCad drawing would then be converted to Gcode which machinery would cut out of wood. This point-to-point computer controlled machine is shown on the left of Figure 4.16. Three pieces were designed. The first was the top cover for the EDahina, shown in Figure 4.17 (a). This piece snuggly plugged into the hole on the EDahina and had 16 holes for roping and holes to fit the FSRs in the desired schematic. The second piece was a larger version of the EDahina's top cover, which would fit the EBayan, shown in Figure 4.17 (b). The third piece was a bottom ring which could fit both the EDahina and EBayan to help rope the drums together. Figure 4.17 (c) shows a group of these rings.


(a) (b) (c)
Figure 4.17: Pictures showing custom wooden pieces created at County Cabinet Shop, Inc.

Long, cylindrical pieces of wood were cut for the EDahina to make it look like a real Tabla. Rope was purchased that could fit through the holes that were drilled in all the pieces. This rope was dyed black as shown in Figure 4.18 (a). The top covers, the bottom rings and the cylindrical pieces of wood were all painted black as shown in Figure 4.18 (b) and (c).


(a) (b) (c)
Figure 4.18: Pictures showing painting process of the ETabla 2.0.


(a) (b) (c)
Figure 4.19: Pictures showing Technology integrated into encasement of the ETabla 2.0.

Holes were drilled into the EDahina so that the BASIC Stamp Carrier Board and the MIDI OUT port could be accessed, as shown in Figure 4.19 (a). The FSRs were mounted onto the top covers as shown in Figure 4.19 (b) and (c). The drums were roped and put together.

The PBASIC code was modified to work through the Roland HandSonic, which is a professionally made drum controller which has hundreds of different drum settings in its sound bank. The HandSonic's sound bank has three settings for Tabla. The ETabla was set up to trigger the correct sound based on which FSR was struck. PBASIC code for the ETabla is included in Appendix B (*Dahina2HS.bs2* and *Bayan2HS.bs2*).

<u>ETabla 2.0: Design Problems:</u>

The ETabla 2.0 is shown in Figure 4.20. At this point, there were a few problems which needed to be addressed. The BASIC Stamp Carrier Board and the MIDI OUT port on the EDahina kept slipping inside the encasement. Nuts and bolts were added to fix this problem. Also LEDs were required to help debugging. It would be nice to know whether the BASIC Stamp is receiving power and if it is sending messages. Also, the FSRs needed to be protected by some covering which still shows their location.

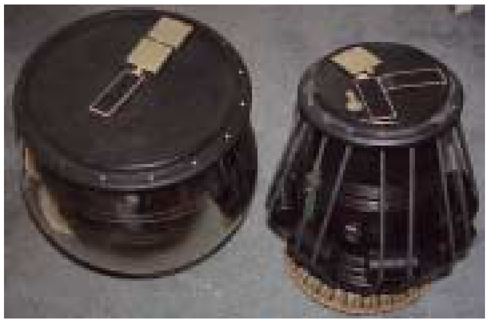


Figure 4.20: Picture showing ETabla 2.0

March 23rd, 2002

<u>ETabla 2.0: User Testing:</u>

The response time of the EDahina was illustrated by user testing. A metronome was used to measure the rate at which one can strike a particular FSR before it becomes unreliable. To design this user test, I took the test myself. I played the EDahina through the Roland HandSonic. Below is a chart showing the response times of the EDahina by stroke. There is one strike per metronome click tested for each stroke. A * denotes if the sound response is immediate with no problems.

Clicks per minute:	**60**	**70**	**80**	**900**	**100**	**110**	**120**	**130**	**140**	**150**	**160**	**170**
Tira Stroke	*	*	*	*	*	*	*	*	*	*	*	*
Tit Stroke	*	*	*	*	*	*	*	*	*	*	*	*
Ring Finger FSR (Ta stroke)	*											
Index Finger FSR(Na Stroke)	*	*	*	*	*	*	*	*	*			
Index Finger FSR (Ta Stroke)	*	*	*	*	*	*	*	*	*	*		
Index Finger FSR (Tu Stroke)	*	*	*	*	*	*	*	*	*	*	*	

From this user test, it was clear that the two FSRs which only measure position were responding well. However, the long FSRs were running slow. This could be because the force variable for the long FSR is captured through RC time rather than on a channel on the ADC. RC time is slower. I also felt that the Ring Finger FSR was not calibrated correctly in the PBASIC code and thus finger strike responses were difficult to pick up.

To solve these time problems, an upgrade from the BASIC Stamp II to the BASIC Stamp IIsx was required for reasons described in Chapter 3.

April 3rd, 2002

<u>ETabla 3.0: Upgrading to the BASIC Stamp IIsx:</u>

The new micro controller was upgraded on both the EDahina and EBayan. The chips should now run 2.5 times faster. There were three modifications needed to upgrade from the BASIC Stamp II to the BASIC Stamp IIsx. First, new software which can compile PBASICIIsx code needed to be installed. Second, variables which are captured by RC time were double in value, and thus the PBASIC code needed to be recalibrated. Third, the *serout* command needed to be modified from:

serout 8. ***12****. 1. [144, 70. RfA]*

to:

serout 8. ***60****. 1. [144, 70. RfA]*

This variable is the timing variable of the MIDI message and is thus effected by the change in speed of the micro controller. Code for the PBASICIIsx code is included in Appendix B.

ETabla 3.0: User Testing:

I now requested Manjul Bhargava to play Tin Taal (described in Chapter 2) and tracked how fast he could play each stroke. Manjul successfully played a recognizable Tin Taal using the ETabla 3.0 at a moderate tempo.

Manjul then tested the response time of the EDahina. Below is a chart showing the results of his tests. There is one strike per metronome click. A * denotes if the sound response is immediate with no problems.

Clicks per minute:	**140**	**160**	**180**	**200**	**220**	**240**	**260**	**280**	**300**	**320**	**340**	**360**
Tira Stroke	*	*	*	*	*	*	*	*	*	*	*	
Tit Stroke	*	*	*	*	*	*	*	*	*	*		
Ring Finger FSR (Ta stroke)	*	*	*	*	*							
Index Finger FSR(Na Stroke)	*	*	*	*								
Index Finger FSR (Ta Stroke)	*	*	*	*								
Index Finger FSR (Tu Stroke)	*	*	*	*								

The Ta stroke on the Ring Finger FSR was the slowest for the ETabla 2.0 user test, only being able to be hit at 60 beats per minute. With the new upgrade, the ETabla 3.0 could now do the same strike 3.5 times faster at 220 beats per minute! This was a major improvement. The Tira and Tit strokes were very fast and were acceptable for performance. The next goal was to raise every stroke close to this level.

Manjul complained that the two long FSRs were generally difficult to strike and get a response. This could be fixed with recalibration. He also recommended that the edge of the Index Finger FSR should always play a Na stroke and the center should always play a Tu stroke. This improvement would make the response time faster.

April 6th, 2002

<u>ETabla 4.0: Optimization of the PBASIC code:</u>

To try increasing the response time of the ETabla, the PBASIC code was optimized to run faster. It was confirmed that no mathematical manipulation of variables occurred unless they needed to for that particular event. This involved moving some lines of code into and out of different *if* conditions. Then all divide operations were converted to shift right operations to save instruction time. Thus “divide by 4” was replaced by “shift right 2”. Next, the position variable for the Ring Finger FSR was eliminated, as it is not used in triggering the HandSonic. Adjustments were also made to the Index Finger FSR, as Manjul recommended.

April 23rd, 2002

<u>ETabla 5.0: Triggering the STK Toolkit:</u>

The ETabla was modified to run two types programs: one to trigger the HandSonic, and the other to trigger the STK Toolkit Tabla sounds, discussed in Chapter 6. For MIDI messages, the STK Toolkit separated the Bayan onto channel 0 and the Dahina onto channel 1. It also did not require any NOTE OFF messages, so all were removed. The linear variables on the EDahina, now had the functionality of changing resonance, which the Handsonic could not do. The edge of the EDahina was programmed to be most resonant, while the center of the drum was programmed to be least resonant. The Long FSR on the EBayan sent POLY PRESSURE MIDI messages to change the pitch of the *Ga* strokes real-time. Thus pitch bending was achieved and functional. PBASICIIsx code that triggers the STK Toolkit is included in Appendix B.

<u>ETabla 5.0: Final Touches:</u>

Final adjustments to the ETabla were made to ensure it could be used in performance on April 25th, 2002 in Taplin Auditorium. This concert is described in Chapter 8 in detail. A protective substance covering the FSRs was created, and each drum was assembled using rope. A final picture of the ETabla is shown in Figure 4.21.

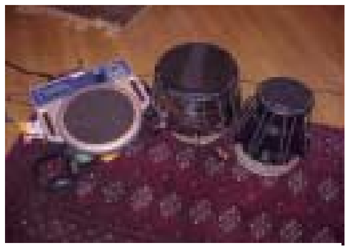


Figure 4.21: Picture showing the ETabla in its final form

Chapter

5

Sound Analysis of the Tabla

"The Musical Drum"[21]

This chapter describes modal analysis of the Tabla. It defines modal analysis and other digital signal processing terms used to describe experimentation of the sound of the Tabla. Then it presents experiments performed in 1919, by scientist C.V. Raman who coined the name for the Tabla: "The Musical Drum"[21]. It finally documents the experiments I have done using MATLAB programming to analyze the sounds of the Tabla, and compare the data obtained with the results of C.V. Raman. The information learned from this process will be used to create a physical model of the sound of the Tabla, which is described in Chapter 6.

What is Modal Analysis?

Sound is vibration that propagates through air, created by the oscillations of objects such as vocal chords, musical instruments, and speakers. These vibrations are converted to the realm of digital audio by recording the sound using a microphone, which converts the varying air pressure into varying voltage. An analog-to-digital converter measures the voltage at regular intervals of time. For all recordings in this analysis, there are 44,100 samples per second. This is known as the sampling rate (SR). The data the computer stores after the analog-to-digital conversion is the sound as a function of time.[22] Figure 5.1 is a graph of a sound of a Bayan as a function of time.

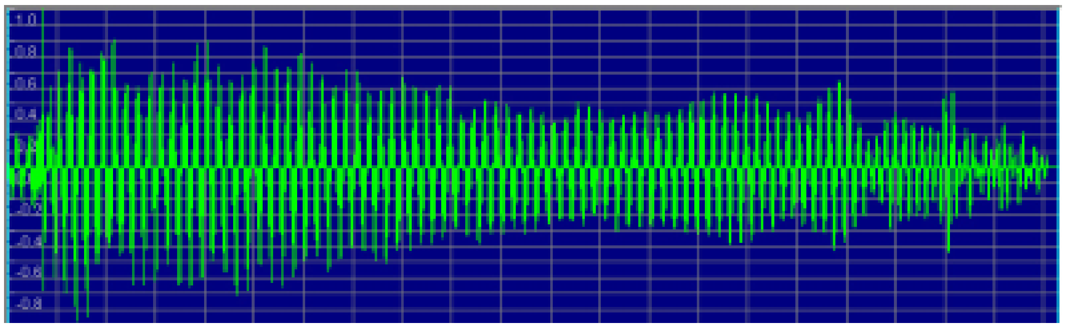


Figure 5.1: Graph of sound of the Bayan as a function of time.

When one plucks a string or blows air through a tube, it begins a repeating pattern of movement, known as oscillation. If a sound has a repeating pattern of movement it has a tone and pitch (harmonic), which distinguishes it from noise (inharmonic). The tone and pitch of the sound can be determined by a sine wave with a particular frequency.[22] The cochlea, an organ in the inner ear enables humans to detect these frequencies. The cochlea is a spiral shaped sum of tissue, with thousands of miniscule hairs that vary in size. The shorter hairs resonate with higher frequencies, while the longer hairs resonate with lower frequencies. So the cochlea converts the air pressure to frequency information, which the brain can use to classify sounds.[23] The Fourier Transform is a mathematical technique that does this exact process. It converts sounds represented in the time domain to sound represented in the frequency domain.[22]

Fourier analysis is based on the important mathematical theorem formulated by Joseph Fourier (1768-1830): "Any periodic vibration, however complicated, can be built up from a series of simple vibrations, whose frequencies are harmonics of a fundamental frequency, by choosing the proper amplitudes and phases of these harmonics"[24]. The Fourier Transform takes a periodic function of time F(t) and turns it into a summation of cosine and sine waves. A periodic function is transformed into the Fourier Series by the equation below:

$$F(t) = \sum_{m=1}^{\infty} \left(a_m \cos(2\pi k f_o t) + b_m \sin(2\pi k f_o t) \right)$$

The term a_m is the average waveform. Coefficients b_m and c_m are the weights of the cosine and sine terms, which describe different frequencies.[22]

With the Fast Fourier Transform (FFT), one can find the peaks of a sound, in the frequency domain. These peaks are known as modes. In this chapter the modes of the Tabla will be analyzed.

C.V. Raman's Tabla Sound Analysis

Nobel Prize winning scientist C.V. Raman published his works on the acoustic characteristics of the Tabla in 1934.[21] He explains that the drums have two features which enable strikes to emit harmonic tones which sustain, distinguishing the Tabla from any other drum in the world. First is the heavy wooden shell on which the *pudi* (drum-head) is stretched upon. Second is the weight added to the *pudi* of iron-filings, rice, charcoal, and gum, which is precisely applied to match the sustaining tone that the acoustic of the shell prescribe. Raman further explains that the duration of the tone is a function of the width of the ring of leather which holds the *pudi* in place. A narrow ring emits a prolonged, bright tone, whereas a broad ring emits a dull, short-lived tone.

The first nine normal modes of the membrane can be split into five harmonic tones because of the unique construction of the Tabla. The normal modes of a harmonic circular membrane described by Raman, are shown in Figure 5.2. The dotted lines denote nodal lines.

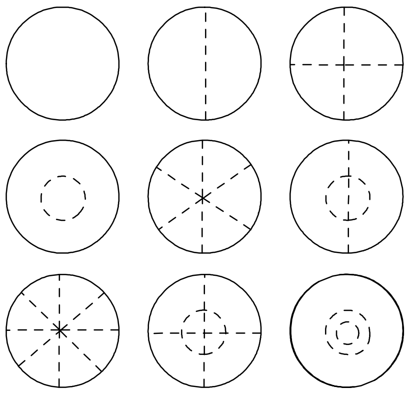


Figure 5.2: Figure showing Normal modes of a Harmonic Membrane [21]

The first mode of vibration occurs when the flat fingertips slap the center of the drum and quickly releases. This mode is without any interior nodal lines. This is the fundamental mode. The second harmonic occurs when the membrane vibrates in two separate parts, divided by a nodal diameter. This mode is excited when the flat palm slaps the edge of the *pudi* while a finger gently lays upon a diameter.

The third harmonic mode occurs when the *pudi's* vibration is separated by two parallel nodal lines. This mode is excited by the *Na* stroke, which makes the *pudi* vibrate in three circular regions. The fourth harmonic occurs when the *pudi's* vibration is separated into three parallel regions. This could be exerted by adding the middle finger as a damper in the *Na* stroke, dividing the *pudi* into four regions. The fifth harmonic is excited by splitting the *pudi* into five different parts, separated by four nodal lines. This is generally very difficult to achieve on most Tabla. [21]

Modal Analysis using MATLAB

This section will describe modal analysis of different Tabla strokes using MATLAB programming. I am trying to observe if in fact the Tabla emits a harmonic tone, validating it as a "musical drum" which C.V. Raman observed. I will outline the evolution and development of software that can be used to obtain the modes of any sound file. All MATLAB code described in this section can be found in Appendix D.

The sound files used in this analysis can be found on the CD at the back of this report. They are taken from Zakir Hussain's, Remember Shakti, CD 2, Track 1.[25] The following chart describes how I will refer to these sound files:

TRACK	Sound File Name	Drum	Stroke
1	*new-B.wav*	Bayan	*Ga*
2	*D-na.wav*	Dahina	*Na*
3	*D-tu.wav*	Dahina	*Tu*
4	*D-ta.wav*	Dahina	*Ta*

FFT:

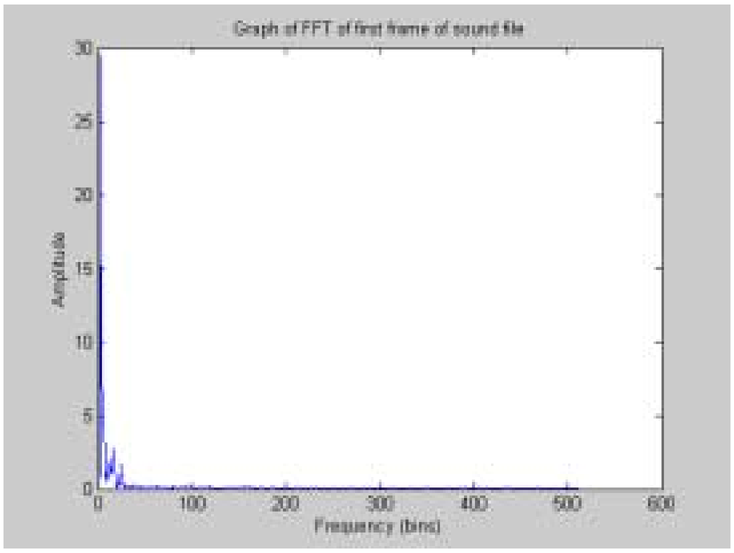
The first program I wrote was a function that calculates and graphs the FFT of the first frame of a given sound file. I did this using Matlab, as it has functions that support sound files. This function is called myFFT.m, and takes in the digital array containing the sound file and the *FFTSize*. The FFT size is the number of bins that the frequency domain is broken into. To convert the frequency in hertz given the *bin*, one uses the following equation:

$$Frequency(Hz) = \frac{(SamplingRate)(bin)}{(FFTSize)}$$

I graphed the first frame of sound file *new-B.wav*, with the following Matlab command:

array = myFFT('new-B.wav', 44100);

The program output the following graph:



From this graph, one can determine that the first frame of the *Ga* stroke has peaks at low frequency bins. However, one cannot determine how these frequencies change over time. This is implemented in the next program.

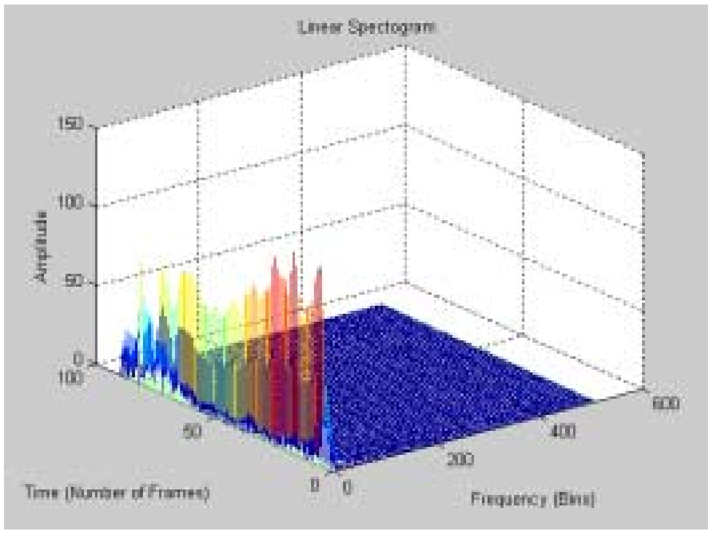
***Short-Time Fourier Transform (STFT)*:**

Now I need to calculate the FFT for every frame in the sound file. This is known as a STFT. I implemented this in *Spectrogram1.m*. This program takes in the sound file, the sampling rate (which is always 44,100 Hz for this analysis), and whether one wants a linear FFT or a logarithmic FFT. In a logarithmic graph, one can see more peaks then in a linear graph, as the amplitudes are put through a logarithm function. This decreases the distance between very high peaks and lower peaks, for easier visual analysis.

I first graphed the linear version of sound file *new-B.wav*, with the following Matlab command:

array = Spectrogram1('new-B.wav', 44100, 'Lin');

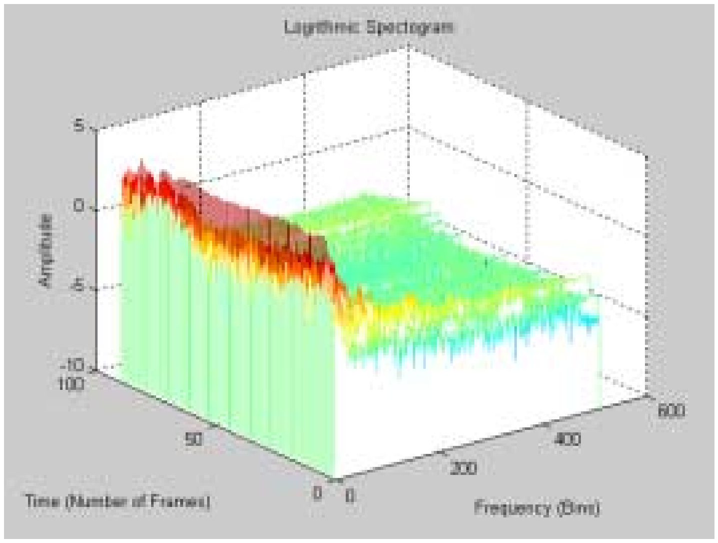
The program output is the graph below:



I then graphed the logarithmic version of the same sound file *new-B.wav*, with the following Matlab command:

array = Spectrogram1('new-B.wav', 44100, 'Log');

The program output is the graph on the next page:



As one can see, the logarithmic version of the Spectrogram portrays more detail to the human eye, for easier analysis. We can see from these graphs that the Bayan stroke *Ga* has many peaks at lower frequencies. However these graphs do not help determine exactly what frequencies these peaks occur at. They just give a general sense of what the sound file's frequencies are over time.

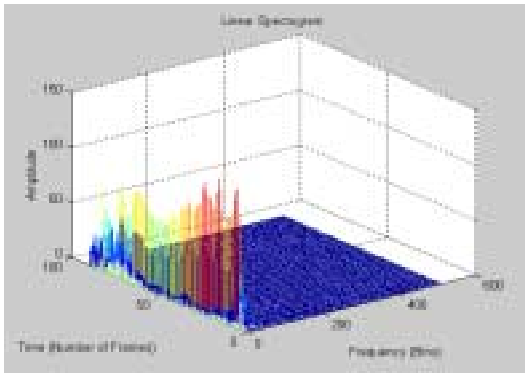
***<u>BREAKING UP THE STFT</u>*:**

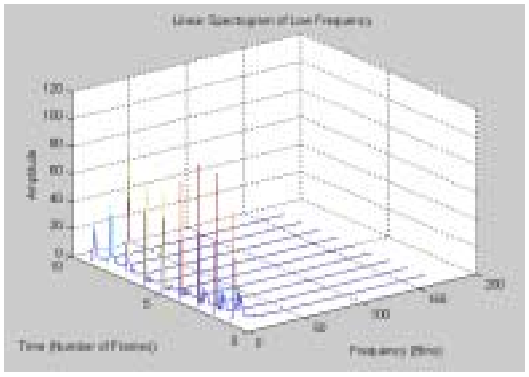
Now I am going to create a program that splits the STFT into three parts: high frequency, middle frequency, and low frequency, for sharper visual analysis. I am also going to average every ten frames together for easier analysis. This program is implemented in *Spectrogram2.m* . This program takes in the sound file, the sampling rate, and whether one wants a linear FFT or a logarithmic FFT.

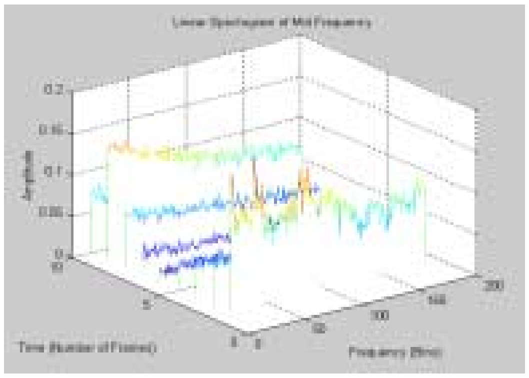
I first graphed the linear version of sound file *new-B.wav*, with the following Matlab command:

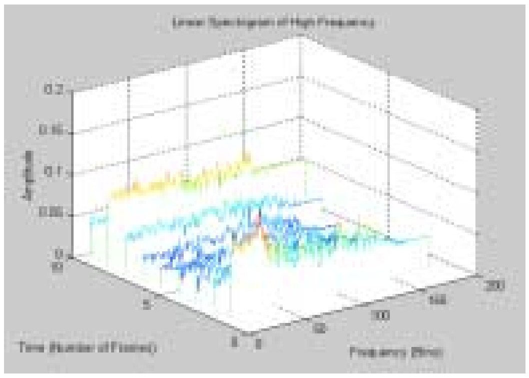
```
array = Spectrogram2('new-B.wav', 44100, 'Lin');
```

The programs output the following four graphs:





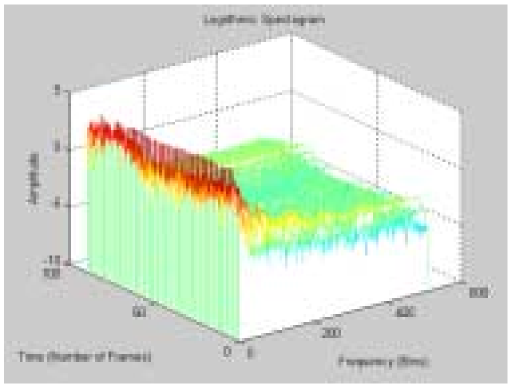


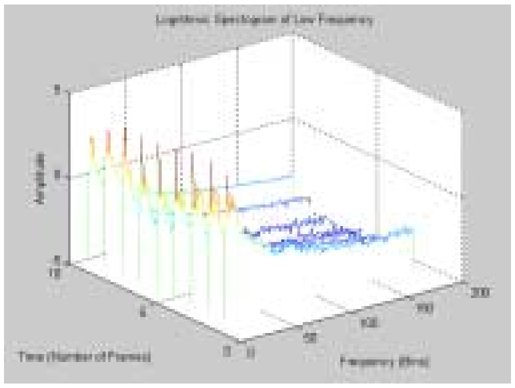


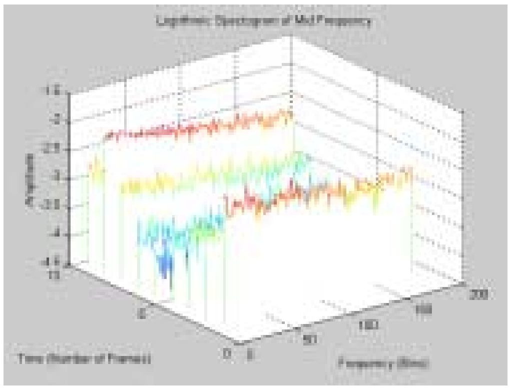
I then graphed the logarithmic version of sound file *new-B.wav*, with the following Matlab command:

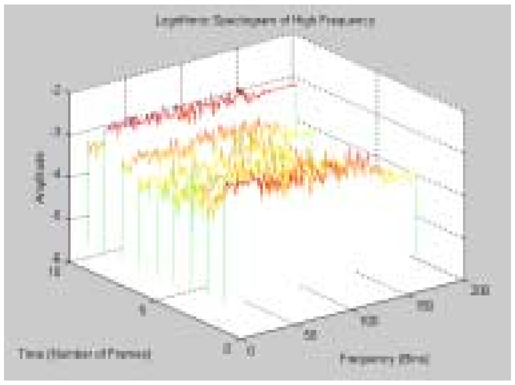
```
array = Spectrogram2('new-B.wav', 44100, 'Log');
```

The program output the following four graphs, shown on the next page:









From these low frequency graphs, one can see that the highest peaks are below bin 50. From the high and middle frequency graphs, one can see that the *Ga* stroke has higher and middle frequencies in the initial attack, which then die down fairly quickly. However, it is interesting that towards the end of the duration of the sound, the high and middle frequencies return to their initial amplitudes. One can attribute this quality to force exerted by the heel of the hand on the head of the drum, which creates a pitch bending effect. The more force the higher the tone. This explains why the graphs look the way they do. This is a key finding in our analysis!

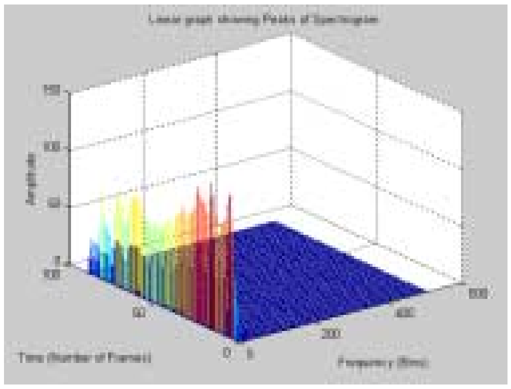
PEAK FINDING TRIAL 1:

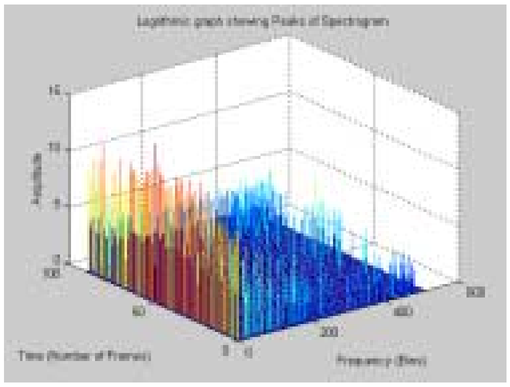
Next, I created a program that finds the peaks of the FFT, and how they vary over time. This program graphs the peaks on a 3D axis. It prints out both the linear version of the graph and logarithmic version. I implemented this program in *Spectrogram3.m*. It takes in the sound file, the sampling rate, and an accuracy number, which determines how precise the peak search is. The higher the accuracy, the more number of peaks it will find. This program uses a function called *HillClimbing.m (written by Tae Hong Park)*, which finds the maximum peaks by determining change in slope.

I first graphed the sound file *new-B.wav*, with an accuracy of 40, with the following Matlab command:

array = Spectrogram3('new-B.wav', 44100, 40);

It created the following two graphs:

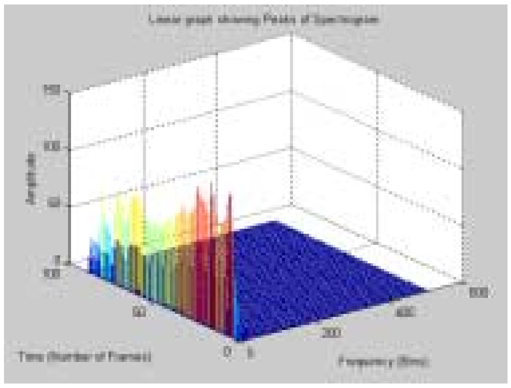


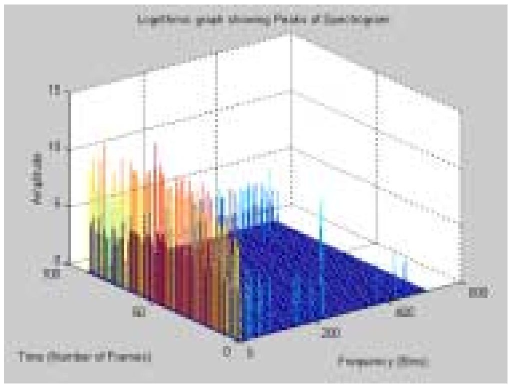


I then graphed the sound file *new-B.wav*, with an accuracy of 20, with the following Matlab command:

array = Spectrogram3('new-B.wav', 44100, 40);

It created the following two graphs:





An accuracy of 40 and an accuracy of 20 gave the same linear graph. However an accuracy of 40 gave a myriad of peaks, where as 20 gave a more manageable amount. From the way I find the maximum peaks in this implementation, I do not have a consistent number of peaks per time frame. Also I do not have a way of determining what the peaks are exactly. This will have to be changed in the next implementation.

PEAK FINDING TRIAL 2:

The next program finds the six highest peaks of the FFT for each frame of the sound file. It then converts the bins to hertz, by the equation:

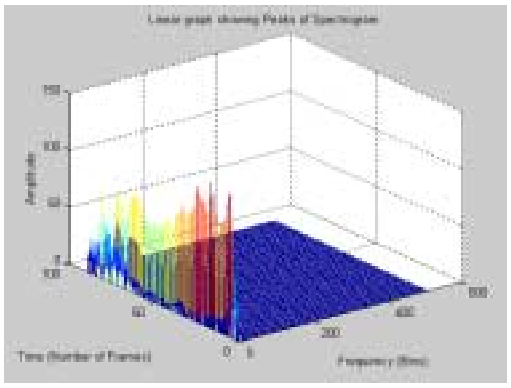
$$Frequency(Hz) = \frac{(SamplingRate)(bin)}{(FFTSize)}$$

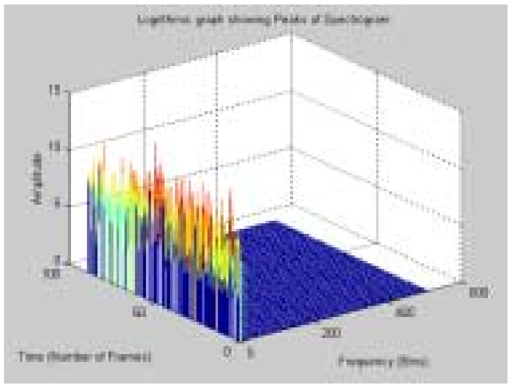
These values are printed out for the linear version of the FFT. This program is implemented in *Spectrogram4.m*. It takes in the sound file and the sampling rate as arguments. This program calls a function called *sixpeaks.m*, to find the maximum peaks of the FFT of each frame. This function determines the peaks by first finding the highest point of the entire FFT, storing it, and then zeroing it out to the minimum value of the FFT. It then zeros out all the points of that hill until the slope changes on either side. It then goes and finds the next maximum value. It repeats this process for six peaks.

I graphed the sound file *new-B.wav*, with the following Matlab command:

array = Spectrogram4('new-B.wav', 44100);

It created the following two graphs, followed by the peaks for every frame of the linear FFT (In order of descending magnitude):





```
129.199219   172.265625   215.332031   258.398438   689.062500   301.464844
129.199219   344.531250   215.332031   387.597656   430.664063   473.730469
129.199219   387.597656   430.664063   215.332031   172.265625   301.464844
129.199219   172.265625   387.597656   215.332031   301.464844   258.398438
129.199219   172.265625   301.464844   215.332031   258.398438   344.531250
129.199219   301.464844   387.597656   430.664063   172.265625   258.398438
129.199219   301.464844   387.597656   430.664063   215.332031   344.531250
129.199219   301.464844   387.597656   430.664063   344.531250   215.332031
129.199219   301.464844   387.597656   430.664063   473.730469   215.332031
129.199219   301.464844   473.730469   430.664063   215.332031   387.597656
129.199219   301.464844   387.597656   430.664063   215.332031   473.730469
129.199219   301.464844   387.597656   430.664063   473.730469   215.332031
129.199219   301.464844   387.597656   473.730469   430.664063   215.332031
```

129.199219	301.464844	473.730469	430.664063	387.597656	172.265625
129.199219	301.464844	387.597656	430.664063	473.730469	344.531250
129.199219	301.464844	430.664063	172.265625	387.597656	473.730469
129.199219	387.597656	301.464844	473.730469	215.332031	430.664063
129.199219	473.730469	301.464844	172.265625	430.664063	387.597656
129.199219	301.464844	387.597656	430.664063	473.730469	215.332031
129.199219	301.464844	387.597656	430.664063	473.730469	215.332031
129.199219	301.464844	387.597656	473.730469	215.332031	430.664063
129.199219	301.464844	473.730469	430.664063	172.265625	215.332031
129.199219	301.464844	387.597656	215.332031	430.664063	473.730469
129.199219	301.464844	387.597656	430.664063	215.332031	473.730469
129.199219	387.597656	301.464844	473.730469	430.664063	215.332031
129.199219	387.597656	473.730469	172.265625	301.464844	430.664063
129.199219	430.664063	301.464844	387.597656	172.265625	215.332031
129.199219	301.464844	387.597656	430.664063	344.531250	86.132813
129.199219	301.464844	387.597656	473.730469	172.265625	430.664063
129.199219	301.464844	430.664063	215.332031	387.597656	172.265625
129.199219	301.464844	430.664063	387.597656	215.332031	258.398438
129.199219	387.597656	301.464844	430.664063	215.332031	473.730469
129.199219	301.464844	387.597656	430.664063	172.265625	473.730469
129.199219	301.464844	387.597656	430.664063	258.398438	215.332031
129.199219	301.464844	215.332031	430.664063	387.597656	172.265625
129.199219	215.332031	387.597656	301.464844	172.265625	430.664063
129.199219	215.332031	387.597656	301.464844	430.664063	473.730469
129.199219	387.597656	215.332031	301.464844	172.265625	430.664063
129.199219	258.398438	172.265625	387.597656	301.464844	215.332031
129.199219	430.664063	215.332031	387.597656	301.464844	172.265625
129.199219	387.597656	215.332031	430.664063	172.265625	86.132813
129.199219	215.332031	258.398438	301.464844	430.664063	387.597656
129.199219	301.464844	387.597656	430.664063	215.332031	258.398438
129.199219	387.597656	301.464844	430.664063	172.265625	86.132813
129.199219	258.398438	430.664063	387.597656	301.464844	172.265625
129.199219	215.332031	387.597656	301.464844	430.664063	258.398438
129.199219	301.464844	387.597656	172.265625	258.398438	215.332031
129.199219	301.464844	172.265625	430.664063	387.597656	344.531250
129.199219	215.332031	172.265625	387.597656	301.464844	258.398438
129.199219	215.332031	387.597656	430.664063	86.132813	301.464844
129.199219	215.332031	172.265625	301.464844	258.398438	387.597656
129.199219	215.332031	301.464844	258.398438	344.531250	387.597656
129.199219	215.332031	301.464844	172.265625	258.398438	387.597656
129.199219	215.332031	301.464844	387.597656	172.265625	258.398438
129.199219	301.464844	215.332031	387.597656	258.398438	172.265625
129.199219	301.464844	215.332031	258.398438	387.597656	172.265625
129.199219	301.464844	215.332031	344.531250	86.132813	387.597656
129.199219	301.464844	215.332031	387.597656	344.531250	172.265625
129.199219	172.265625	215.332031	301.464844	387.597656	344.531250
129.199219	215.332031	301.464844	172.265625	344.531250	86.132813
129.199219	215.332031	301.464844	172.265625	258.398438	86.132813
129.199219	301.464844	215.332031	172.265625	86.132813	344.531250
129.199219	172.265625	301.464844	86.132813	43.066406	215.332031
129.199219	86.132813	172.265625	43.066406	344.531250	301.464844
129.199219	86.132813	172.265625	43.066406	215.332031	344.531250
129.199219	172.265625	86.132813	215.332031	43.066406	344.531250
129.199219	172.265625	215.332031	86.132813	301.464844	258.398438
129.199219	172.265625	344.531250	215.332031	86.132813	301.464844
129.199219	172.265625	86.132813	43.066406	344.531250	215.332031
129.199219	86.132813	172.265625	43.066406	215.332031	258.398438
129.199219	172.265625	86.132813	43.066406	258.398438	215.332031
129.199219	172.265625	215.332031	86.132813	258.398438	387.597656
129.199219	172.265625	86.132813	215.332031	43.066406	344.531250
129.199219	172.265625	86.132813	43.066406	387.597656	215.332031
172.265625	129.199219	258.398438	215.332031	86.132813	344.531250
172.265625	129.199219	86.132813	43.066406	258.398438	215.332031
129.199219	172.265625	215.332031	86.132813	301.464844	387.597656
129.199219	172.265625	86.132813	215.332031	43.066406	258.398438
172.265625	129.199219	86.132813	215.332031	43.066406	258.398438
172.265625	129.199219	215.332031	258.398438	301.464844	86.132813
172.265625	129.199219	215.332031	86.132813	43.066406	258.398438
172.265625	129.199219	215.332031	86.132813	258.398438	387.597656
172.265625	129.199219	215.332031	86.132813	43.066406	258.398438
172.265625	215.332031	129.199219	258.398438	344.531250	387.597656
215.332031	129.199219	258.398438	301.464844	86.132813	430.664063
215.332031	129.199219	172.265625	258.398438	301.464844	86.132813
172.265625	215.332031	129.199219	258.398438	344.531250	301.464844
215.332031	129.199219	258.398438	301.464844	86.132813	344.531250
215.332031	129.199219	172.265625	258.398438	301.464844	344.531250
172.265625	215.332031	129.199219	301.464844	258.398438	344.531250

From the graphs one can see that six highest peaks are in the lower frequencies, for both the logarithmic version and the linear version. From the frequencies printed out one can see that peaks range from 86.1328 Hz to 430.6640 Hz. However, there are too many numbers printed out to do a proper analysis. In the next implementation I will fix this.

PEAK FINDING TRIAL 3:

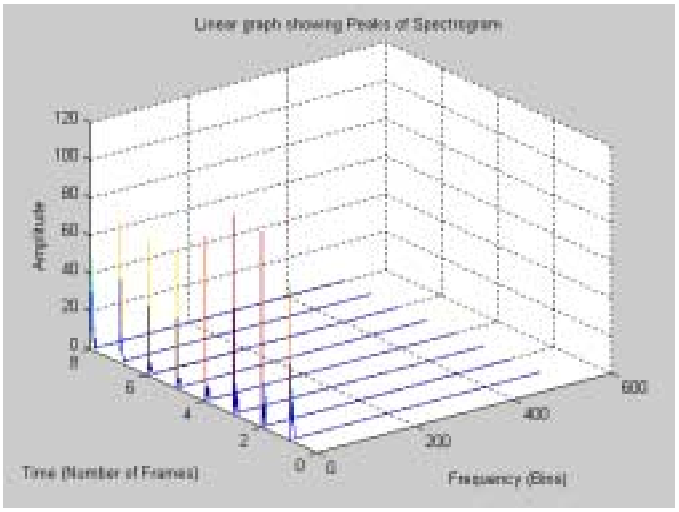
In the next program, a variable number peaks, determined by the user, can be found for an averaged number of frames. The number of averaged frames is a variable that can also be determined by the user. This program is implemented in *Spectrogram5.m*. It takes in the sound file, the sampling rate, whether one wants a linear FFT or a logarithmic FFT, *k* number of frames to be averaged, the number of peaks to find, the FFT size, and whether one wants the peaks printed out in order of ascending magnitude or ascending frequency. This program calls a function *peaks.m* that finds the maximum peaks for each *k* averaged FFT frames. This function uses the same algorithm as *sixpeaks.m* discussed in Peak Finding Trial 2, except that it has a variable number of peaks.

Analysis of *Ga* stroke*:*

I graphed the sound file *new-B.wav*, with the following Matlab command:

```
array = Spectrogram5('new-B.wav', 44100, 'Lin', 10, 8, 1024, 'Bin');
```

It created the following graph, followed by the frequency in hertz as they change over time, in ascending frequency order:



```
129.199219   172.265625   215.332031   258.398438   301.464844   344.531250   387.597656   430.664063
129.199219   172.265625   215.332031   301.464844   344.531250   387.597656   430.664063   473.730469
129.199219   172.265625   215.332031   301.464844   344.531250   387.597656   430.664063   473.730469
129.199219   172.265625   215.332031   258.398438   301.464844   387.597656   430.664063   473.730469
86.132813    129.199219   172.265625   215.332031   258.398438   301.464844   387.597656   430.664063
86.132813    129.199219   172.265625   215.332031   258.398438   301.464844   344.531250   387.597656
43.066406      86.132813    129.199219   172.265625   215.332031   258.398438   301.464844   344.531250
43.066406      86.132813    129.199219   172.265625   215.332031   258.398438   301.464844   387.597656
```

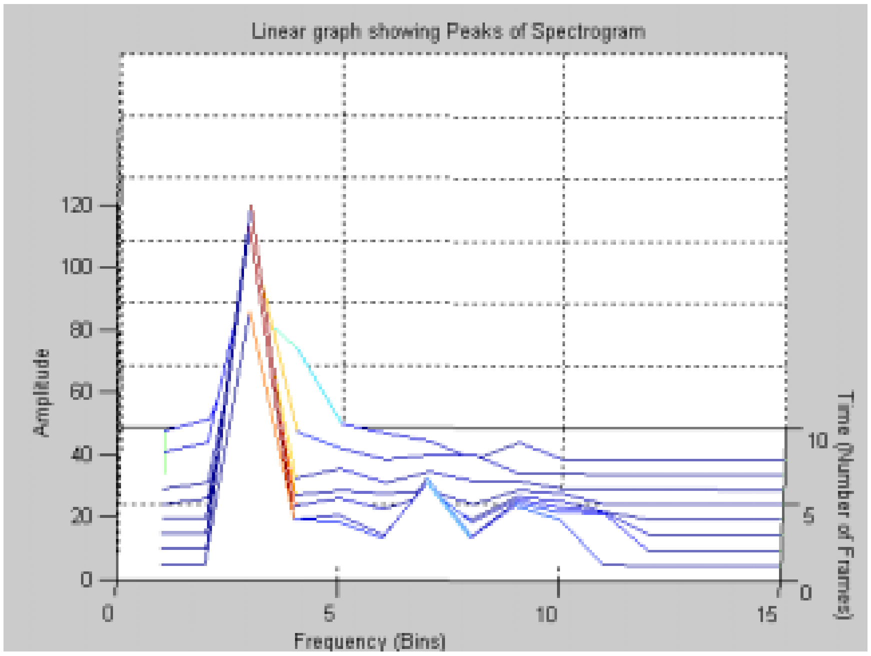
I then wanted to see which peaks were the highest in order of Magnitude, so I typed in the following Matlab command:

array = Spectrogram5('new-B.wav', 44100, 'Lin', 10, 8, 1024, 'Mag');

It gave the same graph as above with these numbers:

```
129.199219   301.464844   387.597656   172.265625   430.664063   215.332031   344.531250   258.398438
129.199219   301.464844   387.597656   430.664063   473.730469   215.332031   172.265625   344.531250
129.199219   301.464844   387.597656   430.664063   473.730469   215.332031   172.265625   344.531250
129.199219   301.464844   387.597656   215.332031   430.664063   172.265625   258.398438   473.730469
129.199219   387.597656   215.332031   301.464844   430.664063   258.398438   172.265625   86.132813
129.199219   215.332031   301.464844   172.265625   387.597656   344.531250   258.398438   86.132813
129.199219   172.265625   86.132813    215.332031   43.066406    301.464844   344.531250   258.398438
129.199219   172.265625   86.132813    215.332031   43.066406    258.398438   301.464844   387.597656
```

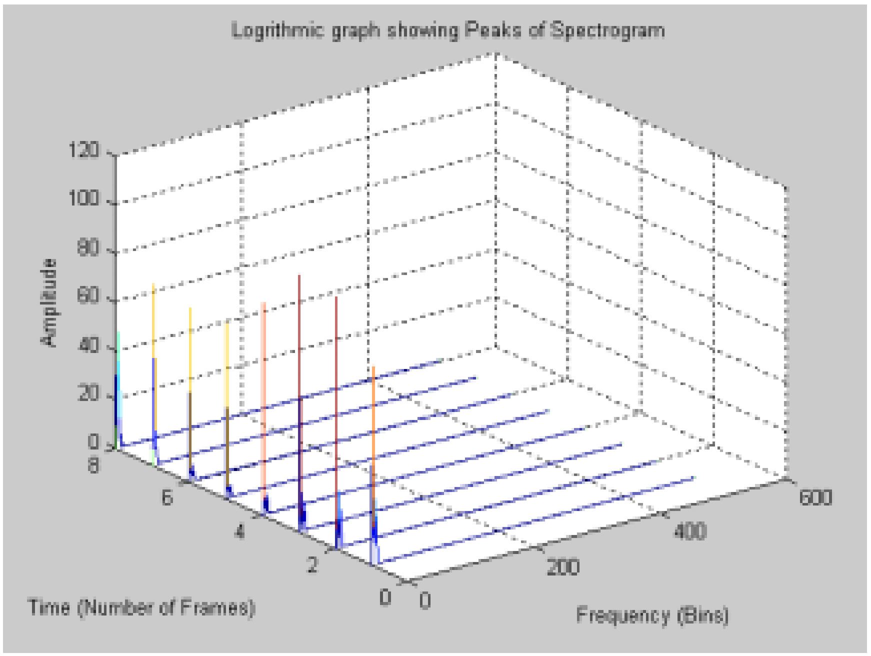
I zoomed in on the important part of the linear version of the graph for the *Ga* stroke, which is shown on the next page:



I then looked at the logarithmic version of this sound file, by typing the following Matlab command:

array = Spectrogram5('new-B.wav', 44100, 'Log', 10, 8 , 1024, 'Bin');

It created the following graph, followed by the frequency in hertz as they change over time, in ascending frequency order:



129.199219	172.265625	215.332031	258.398438	301.464844	344.531250	387.597656	430.664063
129.199219	172.265625	215.332031	301.464844	344.531250	387.597656	430.664063	473.730469
129.199219	172.265625	215.332031	301.464844	344.531250	387.597656	430.664063	473.730469

129.199219	172.265625	215.332031	258.398438	301.464844	387.597656	430.664063	473.730469
86.132813	129.199219	172.265625	215.332031	258.398438	301.464844	387.597656	430.664063
86.132813	129.199219	172.265625	215.332031	258.398438	301.464844	344.531250	387.597656
43.066406	86.132813	129.199219	172.265625	215.332031	258.398438	301.464844	344.531250
43.066406	86.132813	129.199219	172.265625	215.332031	258.398438	301.464844	387.597656

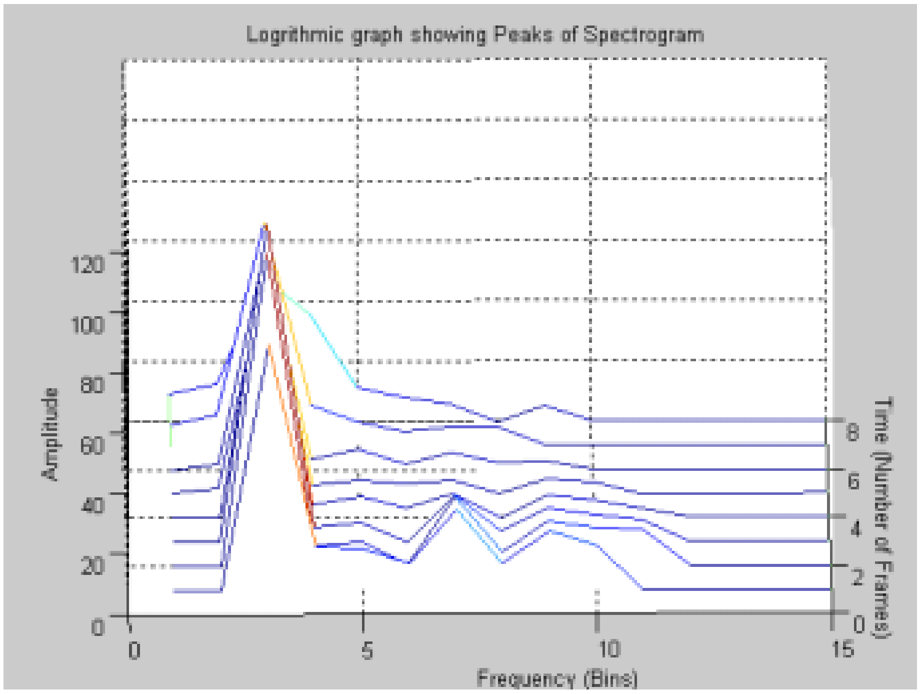
I then wanted to see which peaks were the highest in order of Magnitude, so I typed in the following Matlab command:

array = Spectrogram5('new-B.wav', 44100, ‘Log’, 10, 8, 1024, 'Mag');

It gave the same graph as above with these numbers:

129.199219	301.464844	387.597656	172.265625	430.664063	215.332031	344.531250	258.398438
129.199219	301.464844	387.597656	430.664063	473.730469	215.332031	172.265625	344.531250
129.199219	301.464844	387.597656	430.664063	473.730469	215.332031	172.265625	344.531250
129.199219	301.464844	387.597656	215.332031	430.664063	172.265625	258.398438	473.730469
129.199219	387.597656	215.332031	301.464844	430.664063	258.398438	172.265625	86.132813
129.199219	215.332031	301.464844	172.265625	387.597656	344.531250	258.398438	86.132813
129.199219	172.265625	86.132813	215.332031	43.066406	301.464844	344.531250	258.398438
129.199219	172.265625	86.132813	215.332031	43.066406	258.398438	301.464844	387.597656

I zoomed in on the important part of the logarithmic version of the graph for the *Ga* stroke, which is shown below:



One can first notice that the logarithmic and linear versions of the graphs and data are close to identical in terms of frequency information, which is expected. Thus, for other sound file analysis, we will only look at the logarithmic version of the sound.

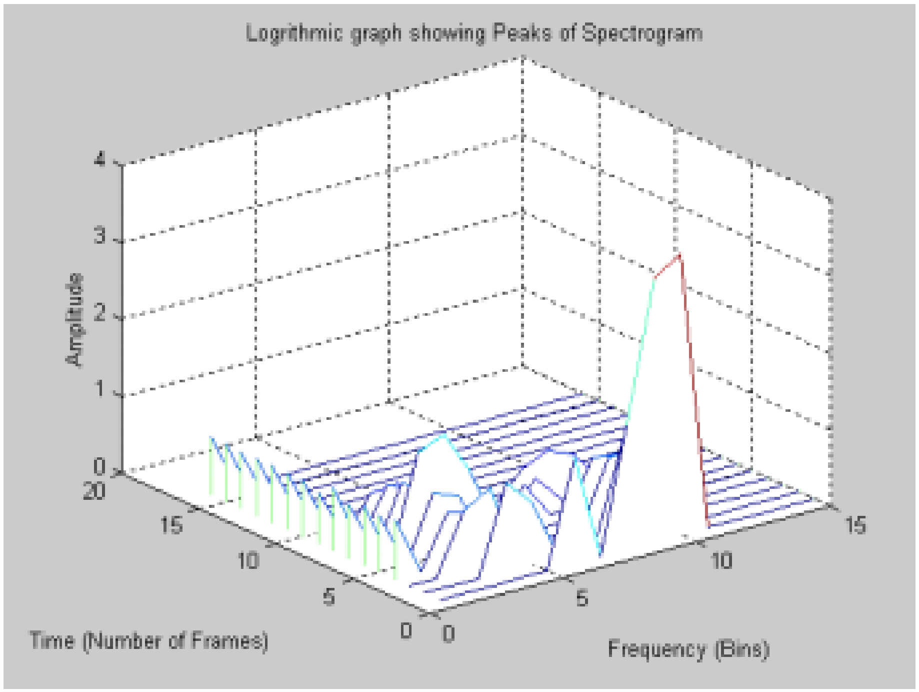
From the data, one can determine that the *Ga* stroke has a fundamental frequency of around 129 Hz, which is present throughout the sound file. It is also clear that 258 Hz (which is 2 * 129 Hz) is a mode and 387 Hz (which is 3 * 129 Hz) is a mode. Notice that these 2 modes are multiples of 2 and 3 of the fundamental pitch. This means that the Bayan creates a harmonic tone.[21] One can also notice from the graphs that the tones lower frequency peaks that are not the fundamental, decay in amplitude over time.

Analysis of *Na* stroke*:*

I graphed the sound file *D-na.wav*, with the following Matlab command:

array = Spectrogram5('D-na.wav', 44100, 'Log', 7, 6, 1024, 'Bin');

It created the following graph, followed by the frequency in hertz as they change over time, in ascending frequency order:



258.398438	344.531250	387.597656	430.664063	1033.593750	1378.125000
172.265625	215.332031	387.597656	689.062500	732.128906	1033.593750
129.199219	172.265625	387.597656	430.664063	689.062500	1033.593750
43.066406	129.199219	172.265625	387.597656	430.664063	689.062500
43.066406	129.199219	172.265625	387.597656	430.664063	689.062500
43.066406	129.199219	172.265625	215.332031	301.464844	689.062500
43.066406	129.199219	172.265625	215.332031	344.531250	689.062500
43.066406	129.199219	172.265625	344.531250	387.597656	430.664063
43.066406	129.199219	172.265625	258.398438	387.597656	689.062500
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

I then wanted to see which peaks were the highest in order of Magnitude, so I typed in the following Matlab command:

array = Spectrogram5('D-na.wav', 44100, 'Log', 7, 6, 1024, 'Mag');

It gave the same graph as above with these numbers:

1033.593750	430.664063	387.597656	344.531250	1378.125000	258.398438
1033.593750	689.062500	172.265625	387.597656	215.332031	732.128906
1033.593750	387.597656	689.062500	172.265625	430.664063	129.199219
689.062500	430.664063	387.597656	43.066406	129.199219	172.265625
129.199219	689.062500	387.597656	43.066406	172.265625	430.664063
172.265625	129.199219	215.332031	43.066406	689.062500	301.464844
129.199219	43.066406	172.265625	689.062500	344.531250	215.332031
43.066406	129.199219	387.597656	172.265625	430.664063	344.531250
43.066406	172.265625	129.199219	387.597656	258.398438	689.062500
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

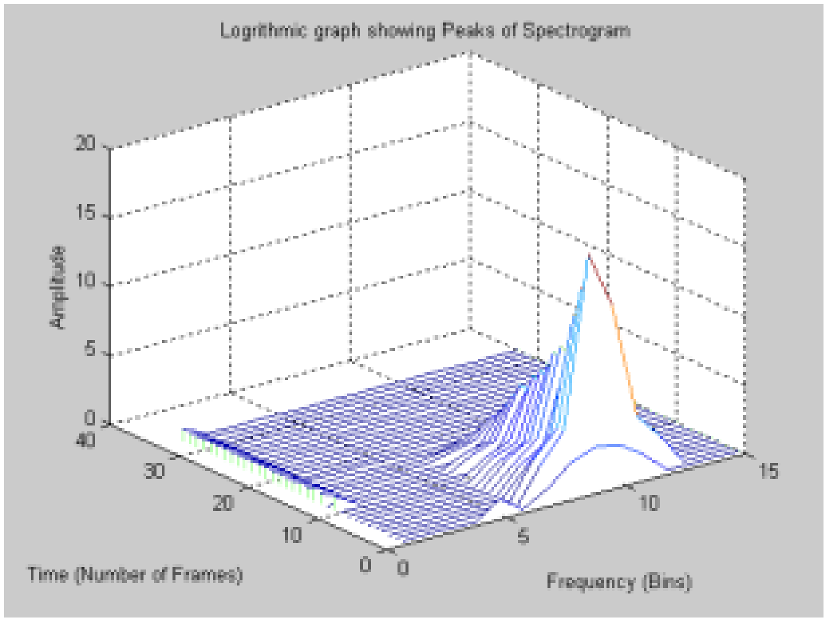
From the data, one can determine that the *Na* stroke has a fundamental frequency of around 172 Hz. It is also clear that 344 Hz (which is 2*172 Hz) is a mode, 689 Hz (which is around 4 * 129 Hz) is a mode, 1033 Hz (which is around 6*172 Hz), and 1378 Hz (which is around 8 * 172 Hz). Notice that these four modes are multiples of 2, 4, 6, and 8 of the fundamental pitch. This means that the Dahina creates a harmonic tone.[21] One can also notice from the graphs that the modes decay in amplitude over time. The 43 Hz which shows up in the data, is the end of the sound file (due to the hum of the recording), not the sound of the drum.

Analysis of *Tu* stroke:

I graphed the sound file *D-tu.wav*, with the following Matlab command:

array = Spectrogram5('D-tu.wav', 44100, 'Log', 3, 6, 1024, 'Bin');

It created the following graph, followed by the frequency in hertz as they change over time, in ascending frequency order:



215.332031	301.464844	344.531250	387.597656	430.664063	473.730469
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
43.066406	344.531250	387.597656	430.664063	473.730469	516.796875
301.464844	344.531250	387.597656	430.664063	473.730469	516.796875
43.066406	301.464844	344.531250	387.597656	430.664063	473.730469
43.066406	344.531250	387.597656	430.664063	473.730469	516.796875
43.066406	301.464844	344.531250	387.597656	430.664063	473.730469
43.066406	301.464844	344.531250	387.597656	430.664063	473.730469
43.066406	344.531250	387.597656	430.664063	473.730469	516.796875
43.066406	86.132813	129.199219	172.265625	215.332031	473.730469
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

I then wanted to see which peaks were the highest in order of Magnitude, so I typed in the following Matlab command:

array = Spectrogram5('D-tu.wav', 44100, 'Log', 3, 6, 1024, 'Mag');

It gave the same graph as above with these numbers:

387.597656	430.664063	344.531250	473.730469	215.332031	301.464844
387.597656	430.664063	344.531250	473.730469	301.464844	516.796875
387.597656	430.664063	344.531250	473.730469	301.464844	516.796875
387.597656	430.664063	344.531250	473.730469	516.796875	301.464844
387.597656	430.664063	344.531250	473.730469	516.796875	301.464844
387.597656	430.664063	344.531250	473.730469	301.464844	516.796875
387.597656	430.664063	344.531250	473.730469	301.464844	516.796875
387.597656	430.664063	473.730469	344.531250	516.796875	301.464844

```
387.597656   430.664063   344.531250   473.730469   301.464844   516.796875
387.597656   430.664063   344.531250   473.730469   301.464844   516.796875
387.597656   430.664063   344.531250   473.730469   43.066406    516.796875
387.597656   430.664063   344.531250   473.730469   301.464844   516.796875
 387.597656   430.664063   344.531250   473.730469   301.464844   43.066406
387.597656   430.664063   344.531250   43.066406    473.730469   516.796875
387.597656   430.664063   344.531250   473.730469   43.066406    301.464844
387.597656   430.664063   43.066406    344.531250   473.730469   301.464844
387.597656   430.664063   43.066406    344.531250   473.730469   516.796875
43.066406    86.132813    129.199219   172.265625   215.332031   473.730469
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
 43.066406    43.066406    43.066406    43.066406    43.066406    43.066406
```

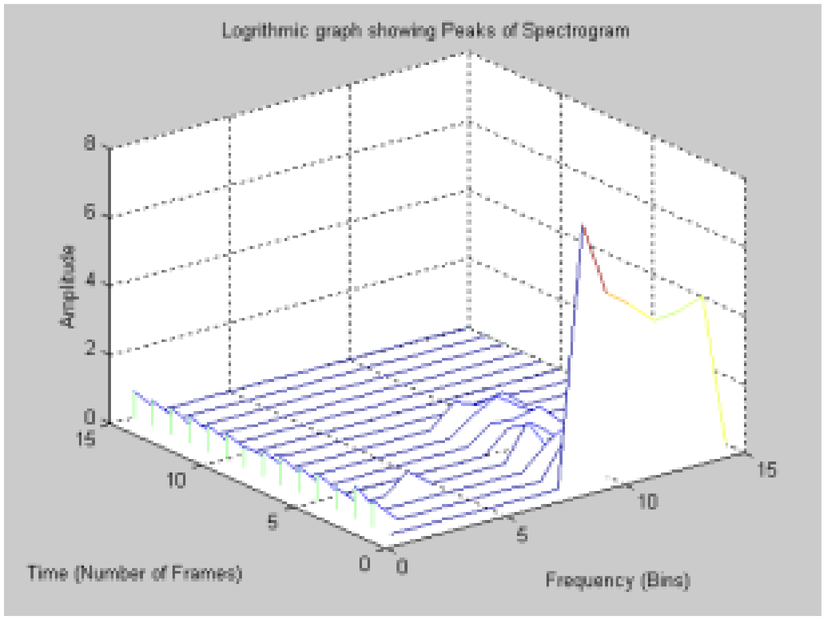
From the data, one can determine that the *Tu* stroke has the same fundamental frequency of around 172 Hz as the *Na* stroke. It is also clear that 344 Hz (which is 2* 172 Hz) is a mode. Notice that this mode is a multiple of 2 of the fundamental pitch. It can also be determined that 215 Hz and 430 Hz (2 * 215 Hz) are present. However, it makes sense that the *Tu* stroke has the same fundamental as the *Na* stroke. Either way, this means that the Dahina creates a harmonic tone.[21] One can also notice from the graphs that the modes decay in amplitude over time. Once again, the 43 Hz which shows up in the data, is the end of the sound file (due to the hum of the recording), not the sound of the drum.

Analysis of *Ta* stroke*:*

I graphed the sound file *D-ta.wav*, with the following Matlab command:

array = Spectrogram5('D-ta.wav', 44100, 'Log', 4, 6, 1024, 'Bin');

It created the following graph, followed by the frequency in hertz as they change over time, in ascending frequency order:



387.597656	430.664063	473.730469	516.796875	559.863281	602.929688
43.066406	387.597656	430.664063	473.730469	516.796875	689.062500
43.066406	387.597656	430.664063	516.796875	559.863281	645.996094
43.066406	172.265625	387.597656	473.730469	645.996094	689.062500
43.066406	387.597656	430.664063	516.796875	559.863281	689.062500
43.066406	387.597656	430.664063	473.730469	559.863281	645.996094
43.066406	387.597656	430.664063	473.730469	602.929688	689.062500
43.066406	387.597656	430.664063	473.730469	645.996094	689.062500
43.066406	86.132813	473.730469	516.796875	689.062500	732.128906
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

I then wanted to see which peaks were the highest in order of Magnitude, so I typed in the following Matlab command:

array = Spectrogram5('D-ta.wav', 44100, 'Log', 4, 6, 1024, 'Mag');

It gave the same graph as above with these numbers:

387.597656	430.664063	473.730469	602.929688	559.863281	516.796875
473.730469	387.597656	430.664063	689.062500	43.066406	516.796875
430.664063	43.066406	387.597656	645.996094	516.796875	559.863281
387.597656	689.062500	43.066406	473.730469	645.996094	172.265625
43.066406	689.062500	387.597656	559.863281	430.664063	516.796875
43.066406	473.730469	559.863281	387.597656	430.664063	645.996094
430.664063	43.066406	387.597656	602.929688	689.062500	473.730469
43.066406	387.597656	473.730469	689.062500	645.996094	430.664063
43.066406	86.132813	473.730469	732.128906	516.796875	689.062500
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406
43.066406	43.066406	43.066406	43.066406	43.066406	43.066406

From the data, one can determine that the *Na* stroke has a fundamental frequency of around 172 Hz. It makes sense that the fundamental is the same for all three Dahina strokes we have analyzed. It is also clear that 344 Hz (which is 2* 172 Hz) is a mode, and 689 Hz (which is around 4 * 129 Hz) is a mode. Notice that these 2 modes are multiples of 2 and 4 of the fundamental pitch. This verifies once again that the Dahina creates a harmonic tone. [21] One can also notice from the graphs that the modes decay in amplitude over time. Once again, the 43 Hz which shows up in the data, is the end of the sound file (due to the hum of the recording), not the sound of the drum.

Sound Analysis Conclusion

From this analysis one can determine that the Bayan and the Dahina studied have fundamentals of 129 Hz and 172 Hz respectively. Both drums have modes above the fundamental that are integer multiples of the fundamental frequency, making both the Bayan and the Dahina harmonic instruments. This concurs with analysis done by C.V. Raman in 1919. However, my analysis must have been much easier to determine this result compared to Raman's experiments, because of the power of the computer as a computational tool.

Chapter

6

Sound Simulation

Using MATLAB and STK Toolkit

This chapter describes the process of creating computer generated Tabla sounds, by programming physical models which emulate the acoustic nature of the drums. It first describes attempts to generate sound using MATLAB programming of the Plucked String Model and different types of filters. It then proceeds to describe how the Tabla sound is programmed in STK Toolkit, a software developed at the Center for Computer Research in Music and Acoustics at Stanford University, designed by Perry Cook and Gary Scavone. This implementation uses banded-waveguides to physically model the sound, designed by Georg Essl, a Ph.D. student at Princeton University.

MATLAB Simulated Sound

Attempt 1:

I started out by using the Plucked String Model to Simulate the Bayan. This means the filter equation was:

$$y[n] = x[n] + .5*(y[n-N] + y[n-(N+1)])$$

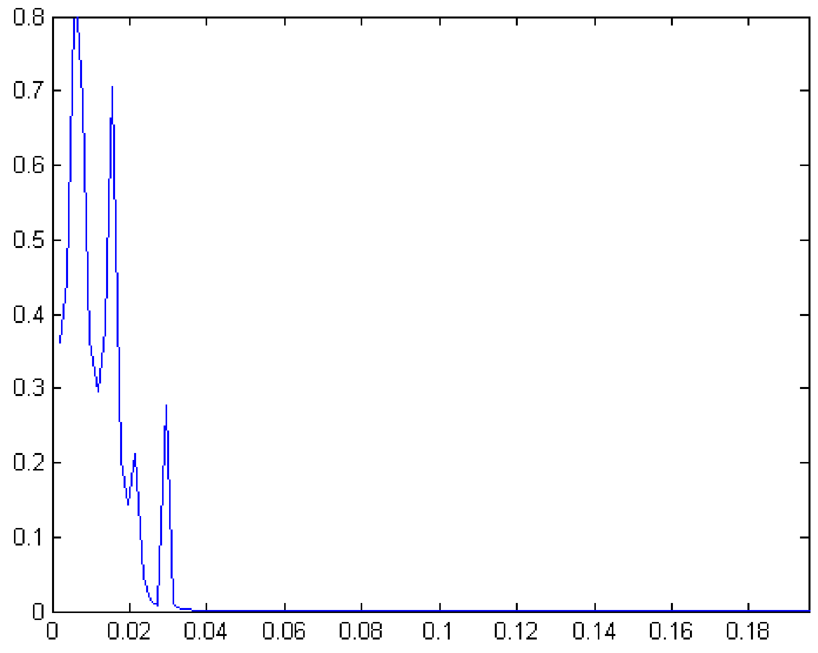
I implemented this in the Matlab program called *BayanSimulator2.m*, found in Appendix E. One gives this program a start frequency and end frequency, and the program will morph between the two. This program runs through the above filter, and then through an ADSR (Attack, Decay, Sustain, Release) envelope. The simulated sound is recorded on Track 5 of the enclosed CD. The problem with it is that it sounds "too stringy" and does not sound like a drum. However I learned the details about the Plucked String Model by actually getting to program it (or rather hack with it).

Attempt 2:

Next, I created a *CoefficientFinder.m* program, which creates 2 arrays, *a* and *b* to hold the coefficients for the equation:

$$y[n] = a1*x[n] + a2*x[n-1] + a3*x[n-2] + ... + b1*y[n-1] + b2*y[n-2] + b3*y[n-3]...$$

I then put these two arrays into the MATLAB function *filter* to get my filtered equation. A graph of the filter I used is shown in the Figure below. It is based on the discoveries of the harmonic nature of the Tabla discussed in Chapter 5. A sample of the sound generated is Track 6 on the CD enclosed. The problem with this program is that it does not allow me to change the filter with time, as the delay line is lost once variable *i* does not equal 1 (see code for *BayanSimulator3.m* in Appendix E). I now have achieved modal synthesis!



Graph of Filter Used

Attempt 3:

Now I created a program that had modal synthesis and that could change the filter over time. I could not use the MATLAB filter() function to do this, because the delay line would get lost if the coefficients were changed. So I had to model a filter similar to the way I did it for the Plucked String Model. Here is the filter equation I used:

$$y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+a(4)*x(n-3)+a(5)*x(n-4)+a(6)*x(n-5) + b(1)*y(n-1)+b(2)*y(n-2)+b(3)*y(n-3)+b(4)*y(n-4)+b(5)*y(n-5)+b(6)*y(n-6)$$

I had to remember to update the delay lines for each iteration. I changed the filter by using a *for* loop which at every iteration had the high frequency amplitude in array *m* get bigger and bigger, as the low frequencies get smaller and smaller. I also moved the highest frequency poles higher and higher at each iteration. This program is called *BayanSimulator.m*, and can be found in Appendix E. This did not produce the desired sound I was hoping.

Physical Model using STK Toolkit

(This section is written by Georg Essl who was in charge of developing Tabla sound on STK Toolkit)

The electronic Tabla controller signals can be used with any standard MIDI device to produce sound. However, the typical synthesis methods do not properly mimic the dynamics of the Tabla drums and hence the performance sound in relation to strokes is not well captured. Physical modeling is known to allow for direct physical interactions and hence the control values produced by the Tabla controller can be directly used as inputs rather than first finding a mapping that relates controller-output to synthesis-relevant parameters. We use the “banded waveguides” which were originally introduced for one-dimensional structures like bar percussion instruments [26] but has recently been generalized to higher-dimensional structures [27].

Banded waveguides are a generalization of digital waveguide filters which accommodate complex material behavior and higher dimensions by modeling the traveling waves for each model frequency separately as is depicted in Figure 6.1.

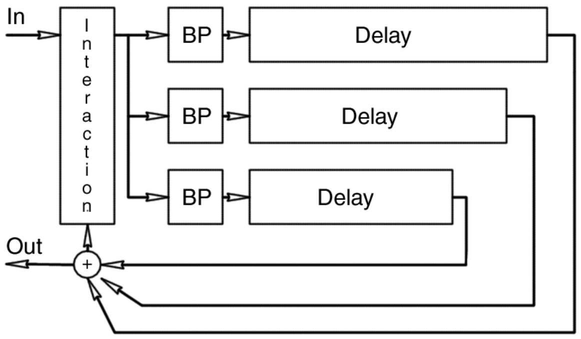


Figure 6.1: Figure showing banded waveguide schematic.

These collection of banded wavepaths which build the full system have, however, a geometric correspondence which allows to find the interactions points. Modes come about as standing waves, which is equivalent to the condition that traveling waves close onto themselves. Hence the task of finding geometric positions from modes corresponds to finding paths that close onto themselves and finding the matching mode for that path. This problem has been studied by Keller and Rubinow [28] and the construction of finding these paths on a circular membrane is depicted in Figure 6.2.

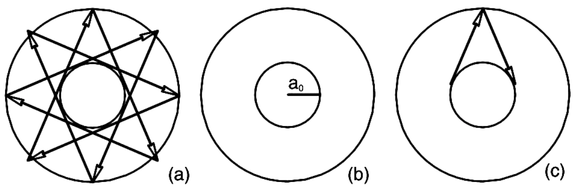


Figure 6.2: Figures showing construction of paths that close onto themselves

Tabla strokes correspond to feeding strike-velocities at the right positions into the delay-lines. A particularly interesting performance stroke is the *Ga* stroke performed on the Bayan. It includes a pitch bend which is achieved by modifying the vibrating area due to pushing forward. This can be viewed as a moving boundary, which in case of banded waveguide corresponds to a shortening of the closed wavepaths, which in turn corresponds to a shortening of the delay-lines of the banded waveguide model. A comparison of a recorded and a simulated *Ga* stroke can be seen in Figure 6.3.

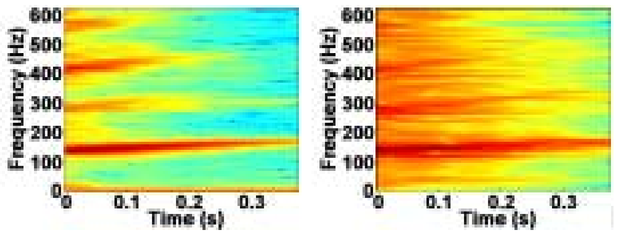


Figure 11. Sonograms comparing recorded (left) and simulated (right) *Ga* strike.

Chapter

7

Graphical Feedback

by Philip Davidson

The Visual System

The visual system for the electronic Tabla is designed to augment the experience of the electronic Tabla by providing player and audience with a visual display that dynamically responds to the drums in parallel to the audio response. Since the audio synthesis requires most of processing power of the audio machine, graphics processing occurs on a second machine, with controller messages routed to both systems. We will describe the response of the system to Bayan strikes.

Our concept for the graphics system began as a combination of geometric form with fluid motion. To respond to the percussive energy of Tabla music, the visualization we developed is based on a particle system in which strikes made by the player appear as patterns composed of small shapes which are the basic visual elements of the display. As the player makes *Ka* and *Ga* strikes on the Bayan controller, particles are rearranged into lines, circles, cardioids, and other shapes depending on the type and quality of each strike. The velocity and pitch are mapped to the size, color, complexity and physical characteristics of the patterns we create.

Once particles have been placed, their continuing motion is controlled by a vector field which imposes forces on each particle. After a strike places a form on the screen, the form will break apart and returns to the background motion. The vector field can also be configured to respond to the movement and positions of particles. The behavior of the field is governed by a distribution of ‘cells’ which determine the forces that will be exerted in their local area, based on the number and distribution of particles in their domain. Through this feedback of cell-particle dynamics, we obtain behaviors which can mimic real-world systems. By altering both the physical characteristics of particles and the specifics of cell-response behaviors, we can use the same system to produce a variety of effects (see Figure 7.1) .

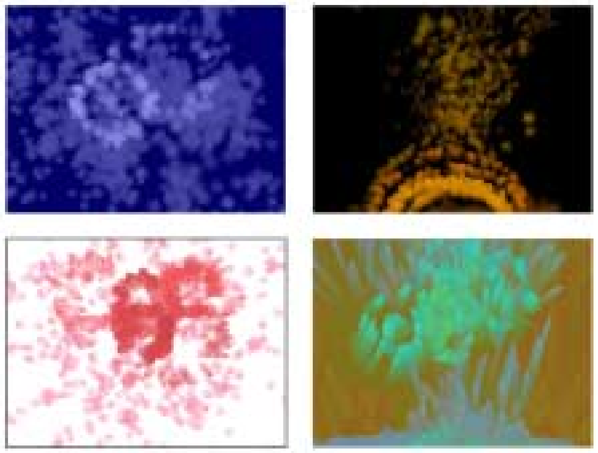


Figure 7.1: Different modes of visual feedback.

In addition to effects produced by the different strikes, it is also helpful to provide a response to the state of the drum itself. Since the Bayan is responsive to palm pressure on the head of the drum, we visually impart a sense of increase or decrease in tension on the drumhead through corresponding compressive or decompressive forces to the particle systems. [29]

The Controls

Table 7.1 describes the controls for the visual system. This graphic system can be used during performance as another instrument. The one who controls the visualization can react real-time to changes in mood, tempo and style as the ETabla is being performed.

KEY	ACTION
Spacebar	Full screen mode
P	Regular window
W	Display MIDI History
E	Show cell history (for Debugging)
R	Change particle type – triangle, cone, spark, blur, petal
T	Change field type – water, fire, snow, flower, off
Y	Change Ga shape
U	Change Ka shape
A	Trails (doesn't clear screen)
S	Randomize colors

Table 7.1: Showing controls of Visual System

Chapter

8

Music Created with the ETabla

Thesis Performance

One of the goals of this project was to make an instrument which can actually be used to create an audio and visual experience that expresses the feelings of the performer and enamors the audience. A performance was held on April 25th, 2002 in Taplin Auditorium, Princeton University, to premiere the Electronic Tabla to the world. Princeton undergraduate and graduate students joined faculty members and alumni in a spectacular performance mixing music from India, Africa and modern America, with electronic grooves and beats.

The ETabla premiered in a traditional North Indian classical song playing a Tin Taal which is described in Chapter 2. The ETabla was also featured in a song with an artist playing the Roland GrooveBox, an instrument that uses a metronome to keep time. It was a major accomplishment that the ETabla could keep up with such a rhythmically precise machine. Another highlight of the concert was the “Dissonance Ritual”, where the ETabla created atmospheric sound-scapes, triggering long lasting electronic samples.

From this thesis performance, the ETabla successfully created a variety of styles of music, with a many types of musicians. Program notes are included in Appendix F. A CD of the concert is also included.

Chapter

9

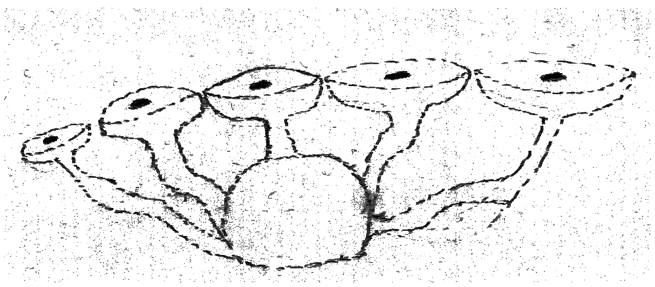
Other Projects using the ETabla

New Instrument Designs

With the technology learned from creating the ETabla, I have developed an array of novel ideas for new controllers for musical expression. I need to learn how to use two new types of sensors: Piezo and Accelerometers. Piezo is used to measure changes in pressure, while accelerometers measure change in rotation. This chapter will outline the design schematics for these new instruments.

New Instrument Design Schematics

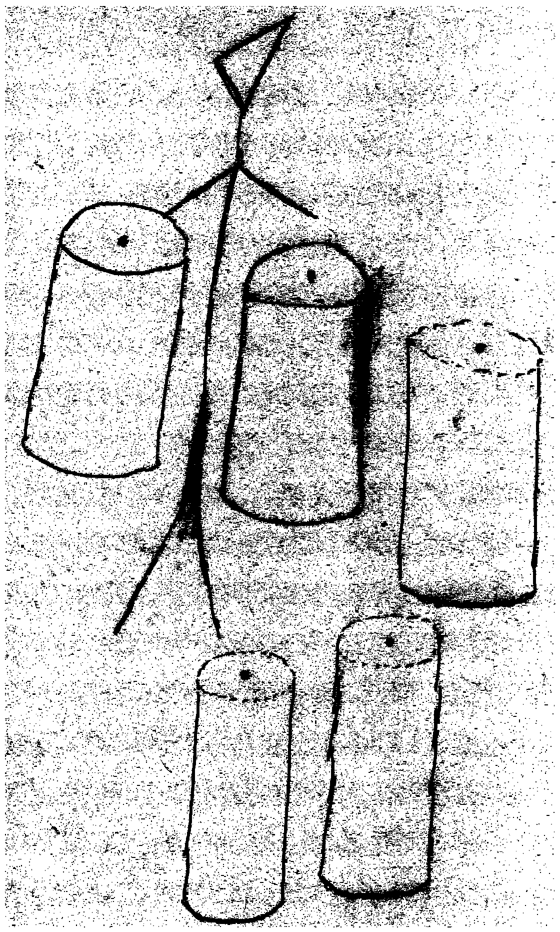
The Electronic Pentatonic Hand Drums ("Pentrix"):



- This instrument has five drums which can play a pentatonic scale
- It is played by a musician who will sit on the floor
- The center circle will collect all the wires and send all outputs

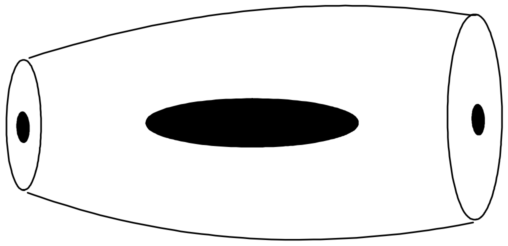
- Each drum should be able to fit a parallax circuit board
- Each drum should be designed to have an easily removable drum head so getting to the inside of the drum is possible
- Technology: Square FSRS, piezo, (maybe long FSRs)

The Electronic "BoomBaZ":



- This set of five drums will be played by a musician who is standing – thus they should all be around waist high
- Each drum from bottom to top will get bigger and bigger
- These Drums will be used to play the lower tones, acting like a bass
- Music Note: These drums have the option of playing a pentatonic scale, or having all 12 tones
- Each drum should be designed to have an easily removable drum head so getting to the inside of the drum is possible
- Technology: Square FSRS, piezo, (maybe long FSRs)

<u>The Electronic Dolak:</u>



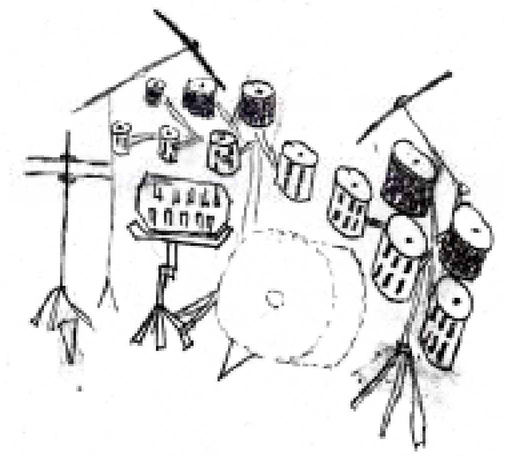
- Two musicians who sit on the floor play this drum - one person will strike both sides of the drum (setting the rhythm), and the other will strike the black square on the top of the drum (setting the tempo, and warping the sounds).

- One end of the drum is bigger than the other side of the drum, to represent a low tone and a high tone relatively

- Each drum should be designed to have an easily removable drum head so getting to the inside of the drum is possible

- <u>Technology</u>: Square FSRS, piezo, long FSRs

<u>The Electric "12toneKit" :</u>

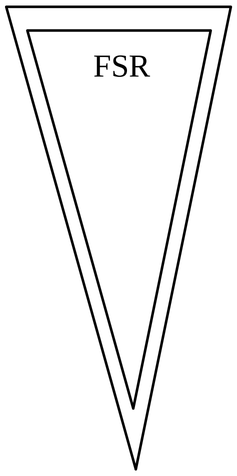


- Has 2 foot pedals and a snare drum to achieve drum kit functionality

- Has 12 tom toms in configuration of a piano with white drums and black drums, to signify the accidentals of the key of C major

- <u>Music Note</u>: The configuration shown starts on F and ends on F (F Lydian). This makes it easy for a drummer to hit the *tonic* with their right hand, while the left hand and feet play a *groove* (beat).

- The foot pedals for bass drum and high hat are going to be on the floor (different than a real drum set where the foot triggers a spring action which strikes the bass drum with a pedal)



- Each tom must be compatible with a existing drum hardware. However new ideas for structures to hold toms are encouraged.

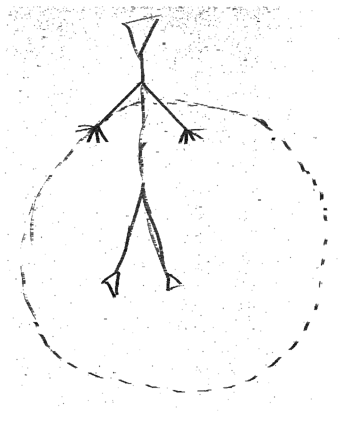
- Each drum should be designed to have an easily removable drum head so getting to the inside of the drum is possible

- The Bass drum may act as a Hub for all the wires and inter-connects of all the drums. The MIDI outputs will also be located here

- Technology: Square FSRS, piezo, (maybe long FSRs)

The Electronic "DanceMachine":

- This a large circular pad which can sense position and velocity of the feet

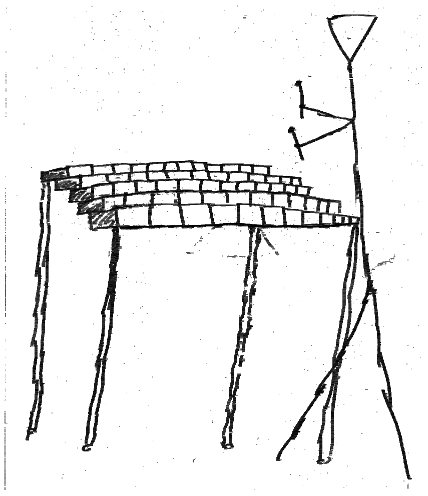


- The structure must have the ability to be transported

- A durable material must be used as humans will jump up and down with their feet

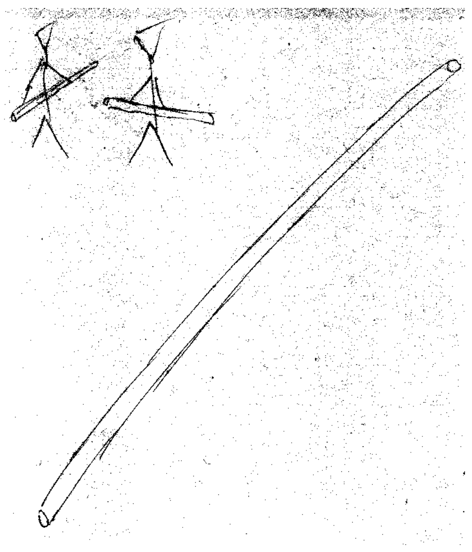
- Technology: Square FSRs and long FSRs

<u>The Electronic Matrix:</u>



- This is a series 5 rows of 12 blocks, based on an instrument of Harry Partch
- Blocks can be all the same size
- Rack for blocks should organize rows with ascending height
- <u>Music Note</u>: Can map 12 tones scales with octaves stacked on each other or map according to Guitar or Violin neck's frets
- <u>Technology</u>: Square FSRs, piezo (maybe long FSRs)

<u>The Electronic Bow:</u>



- This instrument will make sounds when comes in contact with the ground or another Electronic Bow
- Two people can play this instrument together while dancing
- Sticks must encase a circuit board
- Sticks must also be able to have sensors next to outer edge but with protection from strikes
- <u>Technology</u>: piezo, or Square FSRs, maybe Accelerometers for rotation , Wireless

Chapter

10

Conclusion

“Musical performance with entirely new types of computer instruments is now commonplace, as a result of the availability of inexpensive computing hardware, of new sensors for measuring physical parameters such as force and position, and of new software for real-time sound synthesis and manipulation. Musical interfaces that we construct are influenced greatly by the type of music we like, the music we set out to make, the instruments we already know how to play, and the artists we choose to work with, as well as the available sensors, computers, networks, etc. But the music we create and enable with our new instruments can be even more greatly influenced by our initial design decisions and techniques.”[30]

I have outlined the initial design schematics as well as the process of creating a new musical controller. I feel the process has defined the end product, and how the new instrument can be used in a musical context.

Our team successfully created a real-time device for Tabla performance. The ETabla controller augments the traditional interactions in various ways. The performer can now choose the sound production, independent of the physical interaction. Automated teaching feedback has also become possible. We illustrate this ability by providing performance-dependent visual feedback. The ETabla was successfully used to entertain 200 people in an audio-visual extravaganza which demonstrated the power of computers.

However, this project is not over. There still needs to be improvements in sound generation and processing speed of the ETabla controller. Other gizmos such as LEDs to denote pitch and switches to control drones would also be fun to add. As I begin to create new electronic instruments and learn more about the emerging technology, the ETabla will continue to evolve. This Senior Thesis project is just the beginning of the work of my life.

Appendix

A

Experiments with Force Sensing Resistors

Digitizing Force and Position
Human Computer Interface Experiment 1

Experiments on Measuring Position Using an FSR

Materials

- A computer that runs Microsoft Windows
- Programs available at: http://www.CS.Princeton.EDU/courses/cs436/Lab2/Lab2Code/
- A Long FSR
- A Battery Powered Circuit Board with four-pin connector
- A National Instruments input block
- Two 9-volt Batteries

Procedure

1. Connect the four-pin connector of the long FSR to the circuit making sure to match the black dots.
2. Connect the output wires of the circuit to the National Instruments input block by connecting the black wire (ground) to pin 67 and the red wire (signal) to pin 68
3. Connect the two 9-volt batteries to bias the circuit.
4. Load the program called scope.prj in LabWindows CVI and run it.

Results & Conclusions

1. **What happens as you press on the FSR in various locations?**
   The higher we press on the FSR, the higher the voltage. The lower we press on the FSR, the lower the voltage

2. **Is the output a function of how hard you press?**
   No. The FSR is a function of position. However, it can be rewired to measure both position and force.

3. **What is the voltage range of the sensor?**
   The voltage goes from 0 volts to 9 volts

4. **How fast can you tap the sensor and see the effects?**
   We set the sampling rate at 44100 Hz. We could tap the sensor approximately 2 times per second and see our results. Once we went to 3 times a second the output signal was unreadable.

5. **What is the effect of changing the sampling rate?**
   When we lowered the sampling rate, the output response got much slower with less precision. When the sampling rate was raised, the output response got much faster and more precise.

6. **What kind of signal conditioning circuit is this?**
   Non-Inverting Amplifier

7. **What are the pros and cons of using this circuit?**
   The advantage of this the non-inverting amplifier is that there is no conversion necessary to obtain a voltage. The disadvantage of this signal conditioning circuit is that we are especially sensitive to the direct output of the sensor so that if physical properties of that sensor change, or a different sensor is used, we will have to recalibrate the system.

Experiments on Measuring Force Using an FSR

Materials

- A computer that runs Microsoft Windows
- Matlab
- Programs available at: http://www.CS.Princeton.EDU/courses/cs436/Lab2/Lab2Code/
- A Square FSR
- A Battery Powered Circuit Board with four-pin connector
- A National Instruments input block
- Two 9-volt Batteries

Procedure

1. Remove the long FSR and connect the four-pin connector of the square FSR to the circuit making sure to match the black dots. Answer questions 1 through 4 below.
2. Stop running scope.prj.
3. Launch MATLAB.
4. Load the program called daqstart.prj in LabWindows CVI and run it.
5. Set the number of samples to 1000.
6. Set the sampling rate to 500.
7. While slowly increasing the pressure on the sensor, start the data collection. Try to linearly increase pressure across the data collection window, but DO NOT look at the trace while doing it! Have one lab partner say "go" while another pushes the FSR.
8. Click on the Matlab button. This sends the acquired samples to MATLAB. In Matlab, execute the plot(cvi_data) command to see the acquired data. Answer Questions 5 and 6 below.
9. Go back to CVI and acquire 1000 more samples with the sensor at rest on the table.
10. As before, plot the result in MATLAB.
11. Repeat steps 11 and 12 while simply holding the sensor in your hand, but applying no pressure.
12. Remove all of the batteries from the signal conditioning circuit.
13. Disconnect the FSR circuit.

Results & Conclusions

1. **Is the output a function of how hard you press?**
   The harder you press, the higher the voltage. The weaker you press the lower the voltage.

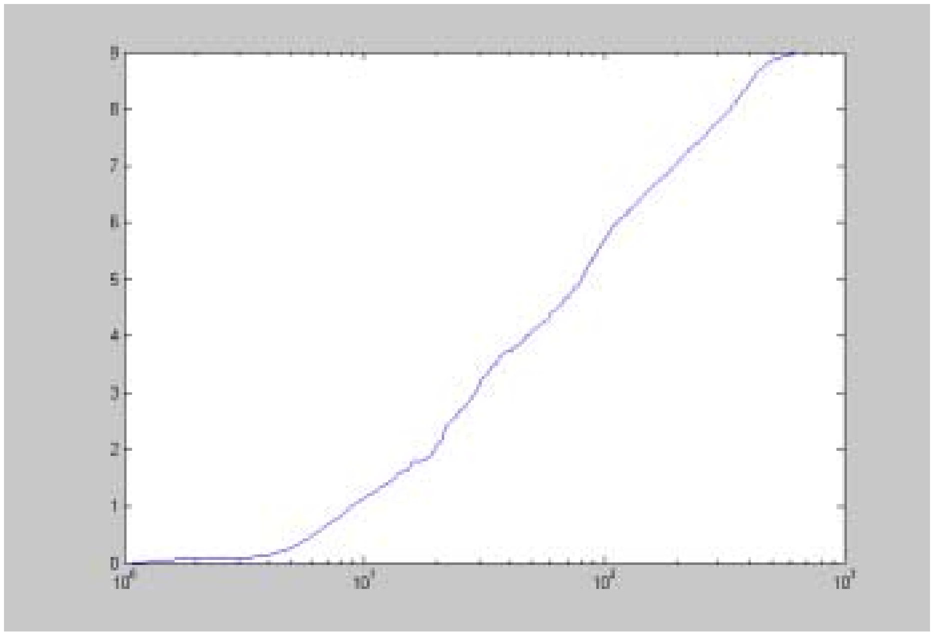
2. **What is the voltage range of the sensor?**
   The voltage ranges from 0 volts to 9 volts.

3. **How fast can you tap the sensor and see the effects?**
   We set the sampling rate at 44100 Hz. We could tap the sensor approximately 8 times per second and see our results.

4. **What is the effect of changing the sampling rate?**
   When we lowered the sampling rate, the output response got much slower with less precision. When the sampling rate was raised, the output response got much faster and more precise.

5. **What is the relationship between pressure on the sensor and the voltage output of the circuit?**
   The relationship of pressure to voltage is not linear. However, the greater the pressure the greater the voltage output.

6. **Write a function in MATLAB to linearize the relationship between pressure on the sensor and the plotted result.**

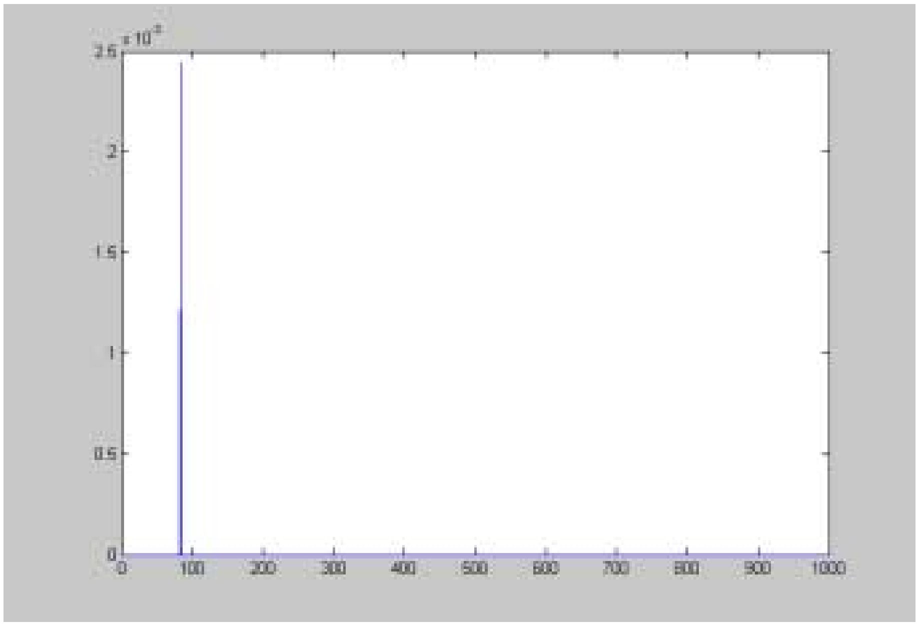


Square FSR - voltage vs log (pressure)

Since we are assuming that force increased linearly across the window, if we plot voltage vs force on a log scale, we obtain a linear relationship.

Formula Voltage $= b\, e^{(\text{pressure})}$, where b is a constant

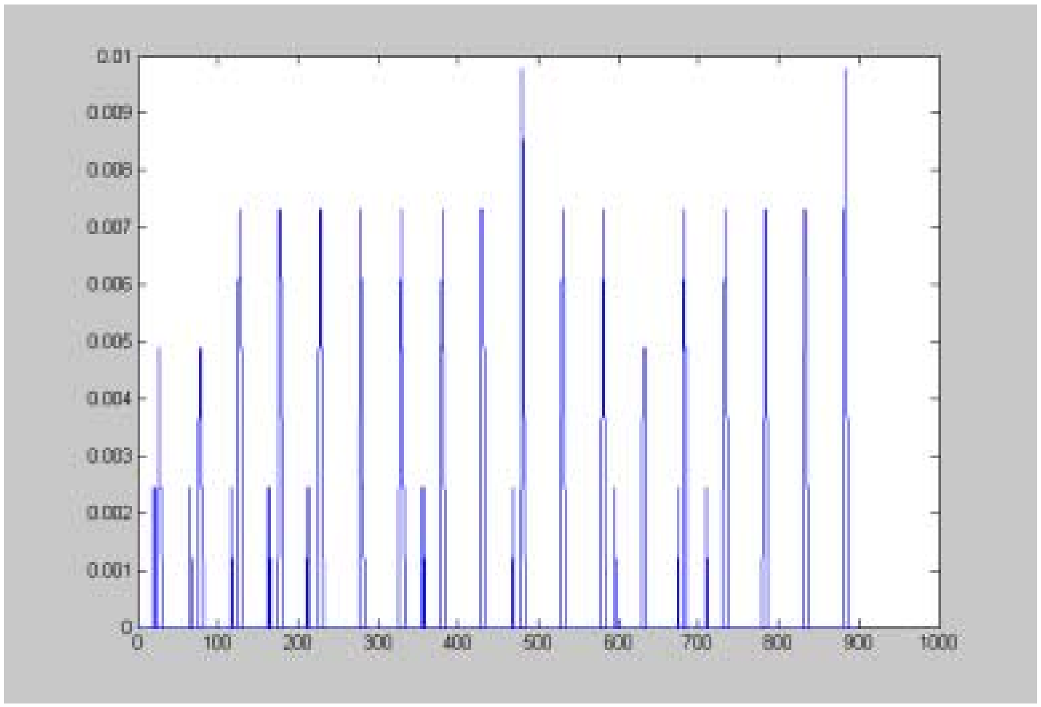
7. What do you see? How does this relate to the quantization of the analog to digital converter?



FSR left still on table

The noise is masked by the FSR in experiments before. Now the pressure exerted by the computers in the room and other sources showed up in our graph in digital format. Our graph shows that there is a sharp peak at 90. It does not show the real analog curve from start to finish.

8. Now, what do you see? Why is this different?



FSR held 'still' in palm of hand

There are many more peaks on this graph. This is probably the sound of our heartbeat.

9. Could choosing another signal conditioning circuit eliminate this?
Yes, use a Single Pole Low Pass Filter.

10. The FSRs plug into a small battery powered circuit board. What is the purpose of that board?
To achieve signal conditioning circuitry.

11. The large square FSR has a resistor attached to it. Why is this necessary?
The resistor on the large square FSR acts as a reference voltage. The long FSR has two resistive films interweaved so it does not need the extra resistor.

Appendix

B

PBASIC Code

ETabla Basic Stamp Code

Bayan2HS.bs2

```
Rf var word
Rpos var word
strikeDamp var bit
slapHold var bit
temp var byte
velocity var byte
CS con 13
CLK con 14
DIO_n con 15
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
AD0 var word
AD1 var word

BSA con 12        '  High side of FSR Fixed Resistor
BSB con 10  '  Capacitor Pin
BSC con 11  '  Force Wiper of FSR, Other side of Capacitor

BSS con 7
SRf  var word ' Slapper Force Variable

loop:

'*** This is for the Slapper
   high BSS
   rctime BSS, 1, SRf
```

```
'   debug ? SRf
    if SRf < 150 then doSlap
    slapHold = 0

'***  This is the linear strip (bender)
'*** Read Force Resistance
    low BSA
    high BSB
    high BSC
    rctime BSC, 1, Rf

'*** Read A/D Channels
    HIGH BSA
    INPUT BSB
    INPUT BSC
    gosub ReadADs

'*** STRIKER variable AD1: The greater the pressure the smaller AD1
'*** AD1 Ranges from 20 to 2100, We will take ones smaller than 1770
if AD1 > 1770 then skipStrike
    '*** Velocity of Striker: 127 loud, 0 soft
     velocity = 127 - (AD1/16)
    'debug ? velocity
    'debug ? strikeDamp
    '*** When hold down striker, strikeDamp = 1, 0 when 1 hit
    if strikeDamp = 1 then doDamp
        '*** Send Controller  (176)
        'serout 8, 12, 1, [176, 16, 127]
        '***  Note on (144) -- good
        serout 8, 12, 1, [144, 67, velocity]
                ' Note Off (128)
        serout 8, 12, 1, [128 ,67, 64]

        'debug "serout 8, 12, 1, [144, 30, "
                'debug ? velocity
        '*** Set strikeDamp = 1 so if holding, wont come back into loop until let go
        strikeDamp = 1
        goto strikeOut
    doDamp:
                AD1 = AD1 / 16
        'serout 8, 12, 1, [176, 1, AD1/2 + 64]
        'debug "serout 8, 12, 1, [176, 1, "
        'debug ? (AD1/2 + 64)
                goto strikeOut
skipStrike:
    '*** strikeDamp = 0 when no Strike, or when let go from hold
    strikeDamp = 0

'*** BENDER: Rf is force, AD0 is position
```

```
strikeOut:

'*** If enough force Rf on bender, then do something
   if Rf > 271 then skipPosition
                '*** AD0 varies from 3250(bottom) to 0 (top)
'               debug DEC AD0, " ", DEC AD1, cr

                AD0 = 127 - (AD0/31)

        Rf = 127 - (Rf/3)
                'debug ? Rf
                '*** Higher AD0 and Rf the higher the pitch
                'debug "Force = ", DEC Rf, "  Position = ", DEC AD0, cr
                if (Rf > AD0) then DomForce
                            temp = ((3*AD0)/4) + (Rf/4)
              'debug ? temp
              '*** Pitch Change (224)
              'serout 8, 12, 1, [233, temp, 30]
                            'Handsonic needs control change (176) for Bender- - good
                            serout 8, 12, 1, [176, 16, temp]
            goto skipPosition
         DomForce:
                            temp = ((3*Rf)/4) + (AD0/4)
              'debug ? temp
              '*** Pitch Change (224)
              'serout 8, 12, 1, [233, temp, 30]
                            'Handsonic needs control change (176) for Bender- - good
                            serout 8, 12, 1, [176, 16, temp]
            goto skipPosition

doSlap:
              if slapHold = 1 then skipPosition
                SRf = 127 - (SRf/2)
                'debug ? SRf
        '*** Send Controller  (176)
        'serout 8, 12, 1, [185, 1, 1]
        '***  Note on (144) Ka (71) -good
        serout 8, 12, 1, [144, 71, SRf]
                ' Note Off (128)
        serout 8, 12, 1, [128 ,71, 64]
                'debug "serout 8, 12, 1, [176, 1, 1]", cr
        'debug "serout 8, 12, 1, [144, 44, "
                'debug ? SRf
        '*** Set slapHold = 1 so if holding, wont come back into loop until let go
        slapHold = 1

skipPosition:

goto loop
```

```
ReadADs:
   high DIO_n
   oddsign = 0
   config = config | %1011
   low CS
   shiftout DIO_n, CLK, lsbfirst, [config\4]
   shiftin DIO_n,CLK,msbpost,[AD0\12]
   high CS
   oddsign = 1
   config = config | %1011
   low CS
   shiftout DIO_n, CLK, lsbfirst, [config\4]
   shiftin DIO_n,CLK,msbpost,[AD1\12]
   high CS
return
```

Dahina2HS.bs2

```
RfA var word '  Force of Ring Linear FSR
RfB var word '  Force of Index Linear FSR
RposA var word
RposB var word
TitHold var bit
RingDamp var bit
DhiraHold var bit
IndexHold var bit
temp var byte
velocity var byte
CSA con 13
CLKA con 14
DIO_nA con 15
CSB con 5
CLKB con 6
DIO_nB con 7
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
ADA0 var word ' Position of Ring Linear FSR
ADA1 var word ' Tira force FSR
ADB0 var word ' Position of Index Linear FSR
ADB1 var word ' Tit force FSR

BSAA con 12      '  High side of FSR Fixed Resistor A
```

```
BSAB con 10  ' Capacitor Pin A
BSAC con 11  ' Force Wiper of FSRA, Other side of Capacitor

BSBA con 4        '  High side of FSR Fixed Resistor B
BSBB con 2  '  Capacitor Pin B
BSBC con 3  '  Force Wiper of FSRA, Other side of Capacitor


loop:

'***  These are the linear FSRs
'*** Read Force Resistance
            low BSAA
            high BSAB
            high BSAC
            rctime BSAC, 1, RfA
            low BSBA
            high BSBB
            high BSBC
            rctime BSBC, 1, RfB
'           debug DEC RfA, " ", DEC RfB, cr

'*** Read A/D Channels
            HIGH BSAA
            INPUT BSAB
            INPUT BSAC
            HIGH BSBA
            INPUT BSBB
            INPUT BSBC
            gosub ReadADs

            'debug DEC ADA0, " ", DEC ADA1, " ", DEC ADB0, " ", DEC ADB1, cr
'           debug DEC ADB0, " ", DEC ADB1, cr
'     debug ? RfA

            '*** DHIRA: If dhira then jump
            if ADA1 < 1000 then doDhira
            DhiraHold = 0

            '*** TIT: If tit then jump
            if ADB1 < 1000 then doTit
            TitHold = 0

'*** RING LINEAR FSR
'*** If enough force RfA on Ring FSR, then do something
  if RfA > 240 then skipRing
             ADA0 = 127 - (ADA0/31)
      RfA = 127 - (RfA/3)
```

```
            'debug DEC ADA0, " ", DEC RfA, cr
            ' '*** Higher ADA0 the more damped the sound
            if (RfA > 100) then DoRingStrike
                    RingDamp = 1
          goto skiptoIndex
       DoRingStrike:
                            if (RingDamp = 1) then skiptoIndex
                            '***  Note on (144)
                            serout 8, 12, 1, [144, 70, RfA]
                                    ' Note Off (128)
                            serout 8, 12, 1, [128 ,70, 64]
                                    'debug "Ti  ", DEC ADA0," ", DEC RfA, cr
                                    RingDamp = 1
                                    goto SkiptoLoop

skipRing:
        RingDamp = 0

skiptoIndex:

'*** INDEX LINEAR FSR
'*** If enough force RfB on Index FSR, then do something
 if RfB > 240 then skipIndex
        if (IndexHold = 1) then SkiptoLoop
                IndexHold = 1
                ADB0 = (ADB0/31)
        RfB = 127 - (RfB/3)
                'debug DEC ADB0, " ", DEC RfB, cr
                ' '*** Higher ADA0 the more damped the sound
           if (RingDamp = 0) then StrikeTu
                        if (ADB0 < 25) then StrikeNa
                                '*** Ta Strike
                                'debug "Ta  ", DEC ADB0, " ", DEC RfB, cr
                        '***  Note on (144)
                        serout 8, 12, 1, [144, 72, RfB]
                                ' Note Off (128)
                        serout 8, 12, 1, [128 ,72, 64]
                goto SkiptoLoop
                        StrikeNa:
                                'debug "Na  ", DEC ADB0, " ", DEC RfB, cr
                                '*** Send Controller  (176)
                        serout 8, 12, 1, [176, 17, 76]
                        '***  Note on (144)
                        serout 8, 12, 1, [144, 74, RfB]
                                ' Note Off (128)
                           serout 8, 12, 1, [128 ,74, 64]
                goto SkiptoLoop

                StrikeTu:
                        'debug "Tu  ", DEC ADB0, " ", DEC RfB, cr
                '***  Note on (144)
                serout 8, 12, 1, [144, 73, RfB]
                        ' Note Off (128)
                serout 8, 12, 1, [128 ,73, 64]
        goto SkiptoLoop

skipIndex:
        IndexHold = 0
        goto skiptoLoop

doTit:
        if TitHold = 1 then skiptoLoop
         'debug ? ADA1
         ADB1 = 127 - (ADB1 / 10)
         'debug ? ADB1
    '*** Send Controller  (176)
    serout 8, 12, 1, [176, 17, 96]
    '***  Note on (144)
    serout 8, 12, 1, [144, 74, ADB1]
         ' Note Off (128)
    serout 8, 12, 1, [128 ,74, 64]
         'debug "Tit  ", DEC ADB1,  cr
    '*** Set TitHold = 1 so if holding, wont come back into loop until let go
    TitHold = 1
         goto skiptoLoop

doDhira:
        if DhiraHold = 1 then skiptoLoop
         'debug ? ADA1
         ADA1 = 127 - (ADA1 / 10)
         'debug ? ADA1
    '***  Note on (144)
    serout 8, 12, 1, [144, 64, ADA1]
    ' Note Off (128)
    serout 8, 12, 1, [128 ,64, 64]
         'debug "Tira  ", DEC ADA1,  cr
    '*** Set DhiraHold = 1 so if holding, wont come back into loop until let go
    DhiraHold = 1

skiptoLoop:

goto loop

ReadADs:
  '*** Read AD A
  high DIO_nA
  oddsign = 0
  config = config | %1011
```

```
    low CSA
    shiftout DIO_nA, CLKA, lsbfirst, [config\4]
    shiftin DIO_nA,CLKA,msbpost,[ADA0\12]
    high CSA
    oddsign = 1
    config = config | %1011
    low CSA
    shiftout DIO_nA, CLKA, lsbfirst, [config\4]
    shiftin DIO_nA,CLKA,msbpost,[ADA1\12]
    high CSA

    '*** Read AD B
    high DIO_nB
    oddsign = 0
    config = config | %1011
    low CSB
    shiftout DIO_nB, CLKB, lsbfirst, [config\4]
    shiftin DIO_nB,CLKB,msbpost,[ADB0\12]
    high CSB
    oddsign = 1
    config = config | %1011
    low CSB
    shiftout DIO_nB, CLKB, lsbfirst, [config\4]
    shiftin DIO_nB,CLKB,msbpost,[ADB1\12]
    high CSB

return
```

Bayan2HS.bsx

```
'{$STAMP BS2sx}
Rf var word
Rpos var word
strikeDamp var bit
slapHold var bit
temp var byte
velocity var byte
CS con 13
CLK con 14
DIO_n con 15
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
AD0 var word
AD1 var word
```

```
BSA con 12        '  High side of FSR Fixed Resistor
BSB con 10  '  Capacitor Pin
BSC con 11  '  Force Wiper of FSR, Other side of Capacitor

BSS con 7
SRf  var word ' Slapper Force Variable

loop:

'*** This is for the Slapper
   high BSS
   rctime BSS, 1, SRf
'   debug ? SRf
   if SRf < 500 then doSlap
   slapHold = 0

'***  This is the linear strip (bender)
'*** Read Force Resistance
   low BSA
   high BSB
   high BSC
   rctime BSC, 1, Rf

'*** Read A/D Channels
   HIGH BSA
   INPUT BSB
   INPUT BSC
   gosub ReadADs

'*** STRIKER variable AD1: The greater the pressure the smaller AD1
'*** AD1 Ranges from 20 to 2100, We will take ones smaller than 1770

if AD1 > 1770 then skipStrike
   'debug ? strikeDamp
   '*** When hold down striker, strikeDamp = 1, 0 when 1 hit
   if strikeDamp = 1 then doDamp
            '*** Velocity of Striker: 127 loud, 0 soft
            velocity = 127 - (AD1>>4) '*** Optimization: used to be divide by 16
      '***  Note on (144) -- good
      serout 8, 60, 1, [144, 67, velocity]
            ' Note Off (128)
      serout 8, 60, 1, [128 ,67, 64]
            'debug ? velocity
      '*** Set strikeDamp = 1 so if holding, wont come back into loop until let go
      strikeDamp = 1
      goto strikeOut
   doDamp:
            AD1 = AD1 / 16
            goto strikeOut
skipStrike:
   '*** strikeDamp = 0 when no Strike, or when let go from hold
   strikeDamp = 0

'*** BENDER: Rf is force, AD0 is position
strikeOut:
'*** If enough force Rf on bender, then do something
   if Rf > 230 then skipPosition
            '*** AD0 varies from 3250(bottom) to 0 (top)
'           debug DEC AD0, " ", DEC AD1, cr

            AD0 = 127 - (AD0>>5) '*** Optimization: used to be Divide by 32

      Rf = 127 - (Rf/3)
            'debug ? Rf
            '*** Higher AD0 and Rf the higher the pitch
            'debug "Force = ", DEC Rf, "  Position = ", DEC AD0, cr
            if (Rf > AD0) then DomForce
                     temp = ((3*AD0)>>2) + (Rf>>2) '*** Optimization: used to be Divide by 4
           'debug ? temp
                     '***  Note on (144) -- good
           'serout 8, 60, 1, [144, 60, velocity]
                     'Handsonic needs polypressure (160) for Bender- - good
                     serout 8, 60, 1, [160, 60, (127-temp)]
         goto skipPosition
      DomForce:
                     temp = ((3*Rf)>>2) + (AD0>>2) '*** Optimization: used to be Divide by 4
           'debug ? temp
               '***  Note on (144) -- good
         'serout 8, 60, 1, [144, 60, velocity]
                     'Handsonic needs polypressure (160) for Bender- - good
                     serout 8, 60, 1, [160, 60, (127-temp)]
         goto skipPosition

doSlap:
           if slapHold = 1 then skipPosition
            SRf = 127 - (SRf>>2) '*** Optimization: used to be Divide by 4
            'debug ? SRf
      '***  Note on (144) Ka (71) -good
      serout 8, 60, 1, [144, 71, SRf]
             ' Note Off (128)
      serout 8, 60, 1, [128 ,71, 64]
      '*** Set slapHold = 1 so if holding, wont come back into loop until let go
      slapHold = 1

skipPosition:

goto loop
```

```
ReadADs:
   high DIO_n
   oddsign = 0
   config = config | %1011
   low CS
   shiftout DIO_n, CLK, lsbfirst, [config\4]
   shiftin DIO_n,CLK,msbpost,[AD0\12]
   high CS
   oddsign = 1
   config = config | %1011
   low CS
   shiftout DIO_n, CLK, lsbfirst, [config\4]
   shiftin DIO_n,CLK,msbpost,[AD1\12]
   high CS
return
```

Dahina2HS.bsx

```
'{$STAMP BS2sx}
RfA var word '  Force of Ring Linear FSR
RfB var word '  Force of Index Linear FSR
RposA var word
RposB var word
TitHold var bit
RingDamp var bit
DhiraHold var bit
IndexHold var bit
temp var byte
velocity var byte
CSA con 13
CLKA con 14
DIO_nA con 15
CSB con 5
CLKB con 6
DIO_nB con 7
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
ADA0 var word ' Position of Ring Linear FSR
ADA1 var word ' Tira force FSR
ADB0 var word ' Position of Index Linear FSR
ADB1 var word ' Tit force FSR
```

```
BSAA con 12      '  High side of FSR Fixed Resistor A
BSAB con 10  ' Capacitor Pin A
BSAC con 11  ' Force Wiper of FSRA, Other side of Capacitor

BSBA con 4       '  High side of FSR Fixed Resistor B
BSBB con 2  '  Capacitor Pin B
BSBC con 3  '  Force Wiper of FSRA, Other side of Capacitor

loop:

'***  These are the linear FSRs
'*** Read Force Resistance
            low BSAA
            high BSAB
            high BSAC
            rctime BSAC, 1, RfA
            low BSBA
            high BSBB
            high BSBC
            rctime BSBC, 1, RfB
'           debug DEC RfA, " ", DEC RfB, cr

'*** Read A / D Channels
            HIGH BSAA
            INPUT BSAB
            INPUT BSAC
            HIGH BSBA
            INPUT BSBB
            INPUT BSBC
            gosub ReadADs

            'debug DEC ADA0, " ", DEC ADA1, " ", DEC ADB0, " ", DEC ADB1, cr
'           debug DEC ADB0, " ", DEC ADB1, cr
'     debug ? RfA

            '*** DHIRA: If dhira then jump
            if ADA1 < 1000 then doDhira
            DhiraHold = 0

            '*** TIT: If tit then jump
            if ADB1 < 1000 then doTit
            TitHold = 0


'*** RING LINEAR FSR
'*** If enough force RfA on Ring FSR, then do something
  if RfA > 500 then skipRing
      RfA = 127 - (RfA>>2) '*** Optimization used to be divide by 4
             'debug ? RfA
```

```
           'debug DEC ADA0, " ", DEC RfA, cr
           ' '*** Higher ADA0 the more damped the sound
           if (RfA > 60) then DoRingStrike
                  RingDamp = 1
         goto skiptoIndex
      DoRingStrike:
                           if (RingDamp = 1) then skiptoIndex
                           '***  Note on (144)
                           serout 8, 60, 1, [144, 70, RfA]
                                  ' Note Off (128)
                           serout 8, 60, 1, [128 ,70, 64]
                                  debug "Ti  ", DEC RfA, cr
                                  RingDamp = 1
                                  goto SkiptoLoop

skipRing:
           RingDamp = 0

skiptoIndex:

'*** INDEX LINEAR FSR
'*** If enough force RfB on Index FSR, then do something

 if RfB > 500 then skipIndex
           if (IndexHold = 1) then SkiptoLoop
                  IndexHold = 1
                  ADB0 = (ADB0>>5) '*** Optimization: was divided by 32
           RfB = 127 - (RfB>>2) '*** Optimization: was divide by 4
                  'debug ? ADB0
                  'debug DEC ADB0, " ", DEC RfB, cr
                  ' '*** Higher ADA0 the more damped the sound
              if (ADB0 > 25) then StrikeTu
                         debug "Na  ", DEC ADB0, " ", DEC RfB, cr
                         '*** Send Controller  (176)
                  serout 8, 60, 1, [176, 17, 96]
                  '***  Note on (144)
                  serout 8, 60, 1, [144, 74, RfB]
                         ' Note Off (128)
                         serout 8, 60, 1, [128 ,74, 64]
           goto SkiptoLoop
                  StrikeTu:
                         debug "Tu  ", DEC ADB0, " ", DEC RfB, cr
                  '***  Note on (144)
                  serout 8, 60, 1, [144, 73, RfB]
                         ' Note Off (128)
                  serout 8, 60, 1, [128 ,73, 64]
           goto SkiptoLoop

skipIndex:
```

```
            IndexHold = 0
            goto skiptoLoop

doTit:
            if TitHold = 1 then skiptoLoop
             'debug ? ADA1
             ADB1 = 127 - (ADB1/10)
             'debug ? ADB1
       '*** Send Controller  (176)
       serout 8, 60, 1, [176, 17, 96]
       '***  Note on (144)
       serout 8, 60, 1, [144, 74, ADB1]
              ' Note Off (128)
       serout 8, 60, 1, [128 ,74, 64]
              debug "Tit  ", DEC ADB1,  cr
       '*** Set TitHold = 1 so if holding, wont come back into loop until let go
       TitHold = 1
              goto skiptoLoop

doDhira:
            if DhiraHold = 1 then skiptoLoop
             'debug ? ADA1
             ADA1 = 127 - (ADA1/10)
             'debug ? ADA1
       '***  Note on (144)
       serout 8, 60, 1, [144, 64, ADA1]
       ' Note Off (128)
       serout 8, 60, 1, [128 ,64, 64]
              debug "Tira  ", DEC ADA1,  cr
       '*** Set DhiraHold = 1 so if holding, wont come back into loop until let go
       DhiraHold = 1

skiptoLoop:

goto loop


ReadADs:
   '*** Read AD A
   high DIO_nA
   oddsign = 0
   config = config | %1011
   low CSA
   shiftout DIO_nA, CLKA, lsbfirst, [config\4]
   shiftin DIO_nA,CLKA,msbpost,[ADA0\12]
   high CSA
   oddsign = 1
   config = config | %1011
   low CSA
   shiftout DIO_nA, CLKA, lsbfirst, [config\4]
   shiftin DIO_nA,CLKA,msbpost,[ADA1\12]
   high CSA

   '*** Read AD B
   high DIO_nB
   oddsign = 0
   config = config | %1011
   low CSB
   shiftout DIO_nB, CLKB, lsbfirst, [config\4]
   shiftin DIO_nB,CLKB,msbpost,[ADB0\12]
   high CSB
   oddsign = 1
   config = config | %1011
   low CSB
   shiftout DIO_nB, CLKB, lsbfirst, [config\4]
   shiftin DIO_nB,CLKB,msbpost,[ADB1\12]
   high CSB
return
```

Bayan2STK.bsx

```
'{$STAMP BS2sx}
Rf var word
Rpos var word
strikeDamp var bit
slapHold var bit
temp var byte
velocity var byte
CS con 13
CLK con 14
DIO_n con 15
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
AD0 var word
AD1 var word

BSA con 12       '  High side of FSR Fixed Resistor
BSB con 10  '  Capacitor Pin
BSC con 11  '  Force Wiper of FSR, Other side of Capacitor

BSS con 7
SRf  var word ' Slapper Force Variable
```

```
loop:

'*** This is for the Slapper
   high BSS
   rctime BSS, 1, SRf
'   debug ? SRf
   if SRf < 500 then doSlap
   slapHold = 0

'***  This is the linear strip (bender)
'*** Read Force Resistance
   low BSA
   high BSB
   high BSC
   rctime BSC, 1, Rf

'*** Read A / D Channels
   HIGH BSA
   INPUT BSB
   INPUT BSC
   gosub ReadADs

'*** STRIKER variable AD1: The greater the pressure the smaller AD1
'*** AD1 Ranges from 20 to 2100, We will take ones smaller than 1770

if AD1 > 1770 then skipStrike
   'debug ? strikeDamp
   '*** When hold down striker, strikeDamp = 1, 0 when 1 hit
   if strikeDamp = 1 then doDamp
            '*** Velocity of Striker: 127 loud, 0 soft
            velocity = 127 - (AD1>>4) '*** Optimization: used to be divide by 16
      '***  Note on (144) -- good
      serout 8, 60, 0, [144, 40, velocity]

            'debug ? velocity
      '*** Set strikeDamp = 1 so if holding, wont come back into loop until let go
      strikeDamp = 1
      goto strikeOut
   doDamp:
            AD1 = AD1 / 16
            goto strikeOut
skipStrike:
   '*** strikeDamp = 0 when no Strike, or when let go from hold
   strikeDamp = 0

'*** BENDER: Rf is force, AD0 is position
strikeOut:
'*** If enough force Rf on bender, then do something
   if Rf > 230 then skipPosition
```

```
                '*** AD0 varies from 3250(bottom) to 0 (top)
'               debug DEC AD0, " ", DEC AD1, cr

                AD0 = 127 - (AD0>>5) '*** Optimization: used to be Divide by 32

        Rf = 127 - (Rf/3)
                'debug ? Rf
                '*** Higher AD0 and Rf the higher the pitch
                'debug "Force = ", DEC Rf, "  Position = ", DEC AD0, cr
                if (Rf > AD0) then DomForce
                        temp = ((3*AD0)>>2) + (Rf>>2) '*** Optimization: used to be Divide by 4
              'debug ? temp
              '*** Pitch Change (224)
              'serout 8, 60, 1, [224, temp, 30]
                        'Handsonic needs polypressure (160) for Bender- - good
                        serout 8, 60, 0, [160, 40, (127-temp)]
            goto skipPosition
        DomForce:
                        temp = ((3*Rf)>>2) + (AD0>>2) '*** Optimization: used to be Divide by 4
              'debug ? temp
              '*** Pitch Change (224)
              'serout 8, 12, 1, [224, temp, 30]
                        'Handsonic needs polypressure (160) for Bender- - good
                        serout 8, 60, 0, [160, 40, (127-temp)]
            goto skipPosition

doSlap:
              if slapHold = 1 then skipPosition
                SRf = 127 - (SRf>>2) '*** Optimization: used to be Divide by 4
                'debug ? SRf
        '***  Note on (144) Ka
        serout 8, 60, 0, [144, 40, SRf]
                '***  Modulation (11)
        serout 8, 60, 0, [11, 127]

        '*** Set slapHold = 1 so if holding, wont come back into loop until let go
        slapHold = 1

skipPosition:

goto loop


ReadADs:
    high DIO_n
    oddsign = 0
    config = config | %1011
    low CS
    shiftout DIO_n, CLK, lsbfirst, [config\4]
```

```
   shiftin DIO_n,CLK,msbpost,[AD0\12]
   high CS
   oddsign = 1
   config = config | %1011
   low CS
   shiftout DIO_n, CLK, lsbfirst, [config\4]
   shiftin DIO_n,CLK,msbpost,[AD1\12]
   high CS
return
```

Dahina2STK.bsx

```
'{$STAMP BS2sx}
RfA var word '  Force of Ring Linear FSR
RfB var word '  Force of Index Linear FSR
RposA var word
RposB var word
TitHold var bit
RingDamp var bit
DhiraHold var bit
IndexHold var bit
temp var byte
velocity var byte
CSA con 13
CLKA con 14
DIO_nA con 15
CSB con 5
CLKB con 6
DIO_nB con 7
config var nib
startB  var config.bit0
sglDif  var config.bit1
oddSign var config.bit2
msfb    var config.bit3
ADA0 var word ' Position of Ring Linear FSR
ADA1 var word ' Tira force FSR
ADB0 var word ' Position of Index Linear FSR
ADB1 var word ' Tit force FSR

BSAA con 12      '  High side of FSR Fixed Resistor A
BSAB con 10  ' Capacitor Pin A
BSAC con 11  ' Force Wiper of FSRA, Other side of Capacitor

BSBA con 4       '  High side of FSR Fixed Resistor B
BSBB con 2  '  Capacitor Pin B
BSBC con 3  '  Force Wiper of FSRA, Other side of Capacitor
```

```
loop:

'***  These are the linear FSRs
'*** Read Force Resistance
            low BSAA
            high BSAB
            high BSAC
            rctime BSAC, 1, RfA
            low BSBA
            high BSBB
            high BSBC
            rctime BSBC, 1, RfB
'           debug DEC RfA, " ", DEC RfB, cr

'*** Read A / D Channels
            HIGH BSAA
            INPUT BSAB
            INPUT BSAC
            HIGH BSBA
            INPUT BSBB
            INPUT BSBC
            gosub ReadADs

            'debug DEC ADA0, " ", DEC ADA1, " ", DEC ADB0, " ", DEC ADB1, cr
'           debug DEC ADB0, " ", DEC ADB1, cr
'     debug ? RfA

            '*** DHIRA: If dhira then jump
            if ADA1 < 1000 then doDhira
            DhiraHold = 0

            '*** TIT: If tit then jump
            if ADB1 < 1000 then doTit
            TitHold = 0


'*** RING LINEAR FSR
'*** If enough force RfA on Ring FSR, then do something
  if RfA > 500 then skipRing
      RfA = 127 - (RfA>>2) '*** Optimization used to be divide by 4
             'debug ? RfA
             'debug DEC ADA0, " ", DEC RfA, cr
             ' '*** Higher ADA0 the more damped the sound
             if (RfA > 60) then DoRingStrike
                        RingDamp = 1
          goto skiptoIndex
      DoRingStrike:
                              if (RingDamp = 1) then skiptoIndex
                              '*** Send Controller  (176)
                        serout 8, 60, 1, [176, 17, 96]
                        '***  ModWheel (1)
                        serout 8, 60, 1, [1, (127-ADA0)]
                              '***  Note on (144)
                        serout 8, 60, 1, [144, 70, RfA]
                              'debug "Ti  ", DEC RfA, cr
                              RingDamp = 1
                              goto SkiptoLoop

skipRing:
      RingDamp = 0

skiptoIndex:

'*** INDEX LINEAR FSR
'*** If enough force RfB on Index FSR, then do something

 if RfB > 500 then skipIndex
      if (IndexHold = 1) then SkiptoLoop
            IndexHold = 1
            ADB0 = (ADB0>>5) '*** Optimization: was divided by 32
      RfB = 127 - (RfB>>2) '*** Optimization: was divide by 4
            'debug ? ADB0
            'debug DEC ADB0, " ", DEC RfB, cr
            ' '*** Higher ADA0 the more damped the sound
            '*** Send Controller  (176)
      serout 8, 60, 1, [176, 17, 96]
      '***  ModWheel (1)
      serout 8, 60, 1, [1, (127-ADB0)]
debug ? 127 -ADB0
            '***  Note on (144)
      serout 8, 60, 1, [144, 70, RfB]
     goto SkiptoLoop

skipIndex:
      IndexHold = 0
      goto skiptoLoop

doTit:
      if TitHold = 1 then skiptoLoop
       'debug ? ADA1
       ADB1 = 127 - (ADB1 / 10)
       'debug ? ADB1
   '*** Send Controller  (176)
   serout 8, 60, 1, [176, 17, 96]
   '***  Note on (144)
   serout 8, 60, 1, [144, 74, ADB1]
```

```
        ' Note Off (128)
    serout 8, 60, 1, [128 ,74, 64]
        debug "Tit  ", DEC ADB1,  cr
    '*** Set TitHold = 1 so if holding, wont come back into loop until let go
    TitHold = 1
        goto skiptoLoop

doDhira:
       if DhiraHold = 1 then skiptoLoop
        'debug ? ADA1
        ADA1 = 127 - (ADA1/10)
        'debug ? ADA1
    '***  Note on (144)
    serout 8, 60, 1, [144, 64, ADA1]
    ' Note Off (128)
    serout 8, 60, 1, [128 ,64, 64]
        debug "Tira  ", DEC ADA1,  cr
    '*** Set DhiraHold = 1 so if holding, wont come back into loop until let go
    DhiraHold = 1

skiptoLoop:

goto loop


ReadADs:
  '*** Read AD A
  high DIO_nA
  oddsign = 0
  config = config | %1011
  low CSA
  shiftout DIO_nA, CLKA, lsbfirst, [config\4]
  shiftin DIO_nA,CLKA,msbpost,[ADA0\12]
  high CSA
  oddsign = 1
  config = config | %1011
  low CSA
  shiftout DIO_nA, CLKA, lsbfirst, [config\4]
  shiftin DIO_nA,CLKA,msbpost,[ADA1\12]
  high CSA

  '*** Read AD B
  high DIO_nB
  oddsign = 0
  config = config | %1011
  low CSB
  shiftout DIO_nB, CLKB, lsbfirst, [config\4]
  shiftin DIO_nB,CLKB,msbpost,[ADB0\12]
  high CSB
```

```
   oddsign = 1
   config = config | %1011
   low CSB
   shiftout DIO_nB, CLKB, lsbfirst, [config\4]
   shiftin DIO_nB,CLKB,msbpost,[ADB1\12]
   high CSB
return
```

Appendix

C

MATLAB Code used for Sound Analysis

Modal Analysis Programming Software

myFFT.m

```
% This matlab program creates a FFT of frame of a soundfile
%
% myFFT(infileName, fs)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%
%   ex. array = myFFT('new-B.wav', 44100);
%
%   Ajay Kapur,     January 5, 2001

function array = myFFT(infileName, fs)

% Initialize Variables
winsize = 1024;
fftSize = 1024;

soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
```

```
soundfile1 = soundfile1 - temp;
s = soundfile1(1:winsize-1);
s = abs(fft(s, fftSize));   % fft of s

plot(s(1:512,:));
title('Graph of FFT of first frame of sound file');
xlabel('Frequency (bins)');
ylabel('Amplitude');
```

Spectrogram1.m

```
% This matlab program creates a 3D spectogram of a STFT of a given .wav file
%
% Spectrogram1(infileName, fs, type)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%   type        :   'Log' for Logrithmic, 'Lin' for Linear
%
%   ex. array = Spectrogram1('new-B.wav', 44100, 'Log');
%
%   Ajay Kapur,     January 6, 2002

function array = Spectrogram1(infileName, fs, type)

% Initialize Variables
winsize = 1024;
fftSize = 1024;
hopsize = winsize*.5; % set hopsize to 50% of winsize

soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
soundfile1 = soundfile1 - temp;

% pre-initialize variables before loop
pos = 1;
frameIndex = 1;

while (pos+winsize) < soundsize
```

```
    s = soundfile1(pos:pos+winsize-1);
    s = abs(fft(s, fftSize));   % fft of s
    s = s(1:fftSize/2);         % s from 0 to Nyquist frequency
    sLog = log(s);              % take log of s
    if type == 'Log'
        array(frameIndex, :) = sLog';  % put sLog into 3d array with index frameIndex
    else
        array(frameIndex, :) = s';  % put s into 3d array with index frameIndex
    end
    pos = pos + hopsize;        % increment pos
    frameIndex = frameIndex + 1; % increment frameIndex
end

if type == 'Log'  % print out Logrithmic Spectrograms with titiles
    waterfall(array), title('Logrithmic Spectogram'); %create 3d graph
    xlabel('Frequency (Bins)');
    ylabel('Time (Number of Frames)')
    zlabel('Amplitude');
else % print out Linear Spectrograms with titles
    waterfall(array), title('Linear Spectogram'); %create 3d graph
    xlabel('Frequency (Bins)');
    ylabel('Time (Number of Frames)')
    zlabel('Amplitude');
end
```

Spectrogram2.m

```
% This matlab program creates a 3D spectogram of a STFT, and then splits it up
% into 3 other AVERAGE graphs: Low, Mid, High for better analysis
%
% Spectrogram2(infileName, fs, type)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%   type        :   'Log' for Logrithmic, 'Lin' for Linear
%
%   ex. array = Spectrogram2('new-B.wav', 44100, 'Log');
%
%   Ajay Kapur,     January 7, 2001

function array = Spectrogram2(infileName, fs, type)

% Initialize Variables
winsize = 1024;
fftSize = 1024;
hopsize = winsize*.5; % set hopsize to 50% of winsize
```

```
soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
soundfile1 = soundfile1 - temp;

% pre-initialize variables before loop
pos = 1;
frameIndex = 1;

while (pos+winsize) < soundsize
   s = soundfile1(pos:pos+winsize-1);
   s = abs(fft(s, fftSize));   % fft of s
   s = s(1:fftSize/2);        % s from 0 to Nyquist frequency
   sLog = log(s);               % take log of s
   if type == 'Log'
      array(frameIndex, :) = sLog';  % put sLog into 3d array with index frameIndex
   else
      array(frameIndex, :) = s';  % put s into 3d array with index frameIndex
   end
   pos = pos + hopsize;       % increment pos
   frameIndex = frameIndex + 1; % increment frameIndex
end

% Initialize Variables before taking Average and spliting Spectrogram into 3 Arrays
k= 10; % k is the number of frames we are going to average together
frameIndex2 = 1;
pos2 = 1;

while pos2 < frameIndex

   temp = 0;
   for i=1:k % get average for k frames
      temp = temp + array(pos2, :);
      pos2 = pos2+1;
   end
   temp = temp/k;

   arrayL(frameIndex2, :) = temp(1:floor(length(temp)/3)); % low array
   arrayM(frameIndex2, :) = temp(floor(length(temp)/3):2*floor(length(temp)/3)); % mid
array
   arrayH(frameIndex2, :) = temp(2*floor(length(temp)/3):length(temp)); % high array

   frameIndex2 = frameIndex2+1; % increment
```

```
end

if type == 'Log'  % print out Logrithmic Spectrograms with titiles
   waterfall(array), title('Logrithmic Spectogram'); %create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayL), title('Logrithmic Spectogram of Low Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayM), title('Logrithmic Spectogram of Mid Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayH), title('Logrithmic Spectogram of High Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
else % print out Linear Spectrograms with titles
   waterfall(array), title('Linear Spectogram'); %create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayL), title('Linear Spectogram of Low Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayM), title('Linear Spectogram of Mid Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
   figure;
   waterfall(arrayH), title('Linear Spectogram of High Frequency'); % create 3d graph
   xlabel('Frequency (Bins)');
   ylabel('Time (Number of Frames)')
   zlabel('Amplitude');
end
```

Spectrogram3.m

```
% This matlab program creates a 3D PEAK spectogram of a STFT.
%
% Note: This program calls hillclimbing.m which finds the max values of the fft.
%
% Spectrogram3(infileName, fs, accuracy)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%   accuracy    :   determines how precise the peak search is
%
%   ex. array = Spectrogram3('new-B.wav', 44100, 40);
%
%   Ajay Kapur,     January 9, 2001

function array = Spectrogram3(infileName, fs, accuracy)

% Initialize Variables
winsize = 1024;
fftSize = 1024;
hopsize = winsize*.5; % set hopsize to 50% of winsize

soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
soundfile1 = soundfile1 - temp;

% pre-initialize variables before loop
pos = 1;
frameIndex = 1;

while (pos+winsize) < soundsize
   s = soundfile1(pos:pos+winsize-1);
   s = abs(fft(s, fftSize));   % fft of s
   s = s(1:fftSize/2);         % s from 0 to Nyquist frequency
   sLog = log(s);              % take log of s and store in sLog
   array(frameIndex, :) = s';  % put s into 3d array with index frameIndex
   arrayLog(frameIndex, :) = sLog'; % put sLog into 3d array with index frameIndex

   % do peak search %
   % Initialize varaiables before peak search
   negMagThresh = max(array(frameIndex,:)) / accuracy; % inputs into HillClimbing
function
   posMagThresh = max(array(frameIndex,:)) / accuracy; % inputs into HillClimbing
function

   minLogValue = min(arrayLog(frameIndex, :)); % find min value in log data
   arrayLog(frameIndex, :) = arrayLog(frameIndex, :) + abs(minLogValue); % shift by min
value up, and then shift down again
   [peakBinl, peakMagl] = hillClimbing(arrayLog(frameIndex,:), negMagThresh,
posMagThresh);
   arrayLog(frameIndex, :) = arrayLog(frameIndex, :) - abs(minLogValue);  % shift back
down
   LogPeakArray(frameIndex, :) = zeros(1,fftSize / 2); % initialize LogpeakArray

   for (i=1:length(peakBinl))
      LogPeakArray(frameIndex, peakBinl(i)) = peakMagl(i);  % create 3d array of peaks
for log graph
   end
   [peakBin, peakMag] = hillClimbing(array(frameIndex,:), negMagThresh,
posMagThresh);

   PeakArray(frameIndex, :) = zeros(1,fftSize / 2); % initialize peakArray

   for (i=1:length(peakBin))
      PeakArray(frameIndex, peakBin(i)) = peakMag(i); % create 3d array of peaks for
linear graph

   end

   pos = pos + hopsize;       % increment pos
   frameIndex = frameIndex + 1; % increment frameIndex
end

waterfall(PeakArray), title('Linear graph showing Peaks of Spectrogram'); % create 3d
graph
xlabel('Frequency (Bins)');
ylabel('Time (Number of Frames)')
zlabel('Amplitude');
figure;
waterfall(LogPeakArray), title('Logrithmic graph showing Peaks of Spectrogram'); %
create 3d graph
xlabel('Frequency (Bins)');
ylabel('Time (Number of Frames)')
zlabel('Amplitude');
```

hillClimbing.m

```
% This function finds peaks in FFT spectrum
%
% function [peakBin, peakMag] = hillClimbing(x, negMagThresh, posMagThresh)
% [peakBin, peakMag] = hillClimbing(x, negMagThresh, posMagThresh)
%
% returns arrays peakBin[] and peakMag[]
%

function [peakBin, peakMag] = hillClimbing(x, negMagThresh, posMagThresh)

xLen          = length(x);
maxMag        = max(x);
tempPeakMag = min(x); %0
foundPeak      = 0;
peakCount      = 1;

i             = 1;
outOfBound     = 0;
slope          = x(i+1)-x(i);

x(512)
while 1%i < xLen-1
    % positive slope start
    % ---------------------------
    while slope > 0

        i = i+1;
        if i > xLen-1                        % out of bound: > analysis window
            return;
        end

        slope = x(i+1)-x(i);

        if foundPeak == 1
            if x(i) > tempPosMagThreshOffset + posMagThresh;
                % reset, new hill to climb
                tempPeakMag =  min(x); %0
                foundPeak      = 0;
            end
        end

    end % positive slope end

    % temporarily store peak candidate
    if x(i) > tempPeakMag
        tempPeakBin      = i;
        tempPeakMag = x(i);
    end

    % negative slope start
    % ----------------------------
    while slope <=0

        if foundPeak == 0
            if tempPeakMag - x(i) > negMagThresh
                foundPeak = 1;
                peakBin(peakCount)      = tempPeakBin;
                peakMag(peakCount)      = tempPeakMag;
                peakCount               = peakCount+1;
            end
        end

        i = i+1;
        if i > xLen-1                       % out of bound: > analysis window
            return;
        end

        slope = x(i+1)-x(i);

    end % negative slope end

    % found peak
    % -----------------------------
    if foundPeak == 1
        tempPosMagThreshOffset = x(i);
    end
end
```

Spectrogram4.m

```
% This matlab program creates a 3D PEAK spectogram of a STFT.
%
% Note: This program calls sixpeaks.m which finds the max values of the fft.
%
% Spectrogram4(infileName, fs)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%
%   ex. array = Spectrogram4('new-B.wav', 44100);
%
%   Ajay Kapur,     January 9, 2001

function array = Spectrogram4(infileName, fs)
```

```
% Initialize Variables
winsize = 1024;
fftSize = 1024;
hopsize = winsize*.5; % set hopsize to 50% of winsize

soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
soundfile1 = soundfile1 - temp;

% pre-initialize variables before loop
pos = 1;
frameIndex = 1;

while (pos+winsize) < soundsize
   s = soundfile1(pos:pos+winsize-1);
   s = abs(fft(s, fftSize));   % fft of s
   s = s(1:fftSize/2);        % s from 0 to Nyquist frequency
   sLog = log(s);               % take log of s and store in sLog
   array(frameIndex, :) = s';  % put s into 3d array with index frameIndex
   arrayLog(frameIndex, :) = sLog'; % put sLog into 3d array with index frameIndex

   % do peak search %

   % Initialize varaiables before peak search

   minLogValue = min(arrayLog(frameIndex, :)); % find min value in log data
   arrayLog(frameIndex, :) = arrayLog(frameIndex, :) + abs(minLogValue); % shift by min
value up, and then shift down again
   [peakBinl, peakMagl] = sixpeaks(arrayLog(frameIndex,:));
   arrayLog(frameIndex, :) = arrayLog(frameIndex, :) - abs(minLogValue);  % shift back
down
   LogPeakArray(frameIndex, :) = zeros(1,fftSize/2); % initialize LogpeakArray

   for (i=1:length(peakBinl))
      LogPeakArray(frameIndex, peakBinl(i)) = peakMagl(i);  % create 3d array of peaks
for log graph
   end

   [peakBin, peakMag] = sixpeaks(array(frameIndex,:));

   PeakArray(frameIndex, :) = zeros(1,fftSize/2); % initialize peakArray
   for (i=1:length(peakBin))
      PeakArray(frameIndex, peakBin(i)) = peakMag(i); % create 3d array of peaks for
linear graph
   end

   % print out Linear peaks %
   fprintf('%f\t %f\t %f\t %f\t %f\t %f\t \n', ((fs*peakBin(1))/fftSize),
((fs*peakBin(2))/fftSize), ((fs*peakBin(3))/fftSize), ((fs*peakBin(4))/fftSize),
((fs*peakBin(5))/fftSize), ((fs*peakBin(6))/fftSize));

   pos = pos + hopsize;       % increment pos
   frameIndex = frameIndex + 1; % increment frameIndex
end

waterfall(PeakArray), title('Linear graph showing Peaks of Spectrogram'); % create 3d
graph
xlabel('Frequency (Bins)');
ylabel('Time (Number of Frames)')
zlabel('Amplitude');
figure;
waterfall(LogPeakArray), title('Logrithmic graph showing Peaks of Spectrogram'); %
create 3d graph
xlabel('Frequency (Bins)');
ylabel('Time (Number of Frames)')
zlabel('Amplitude');
```

sixpeaks.m

```
% This function finds peaks in FFT spectrum by finding the six highest values in the
%  FFT array.
%
% function [peakBin, peakMag] = hillClimbing(x)
% [peakBin, peakMag] = hillClimbing(x)
%
% returns arrays peakBin[] and peakMag[]
%

function [peakBin, peakMag] = hillClimbing(x)
   temp = x; % store x in temp
   xlen = length(x); % store length of x
   minx = min(x); % minimum value in x

   % Find 6 peaks
   for (j = 1:6)
      max = minx;
```

```
        for (i = 1:xlen)
            if (temp(i) > max)
                max = temp(i);
                tempPeakBin = i;
                tempPeakMag = temp(i);
            end
        end
        % Zero out peak
        temp(tempPeakBin) = minx;
        % Zero out peak in negative direction
        slopechange = 0;
        i = 0;
        while (slopechange == 0)
            if (tempPeakBin-i-1 > 0)

                u = temp(tempPeakBin-i);
                d = temp(tempPeakBin-i-1);
                if ((u-d)>0)
                    temp(tempPeakBin-i-1) = minx;
                else
                    slopechange = 1;
                end
                i = i + 1;
            else
                slopechange = 1;
            end
        end

        % Zero out peak in positive direction
        slopechange = 0;
        i = 0;
        while (slopechange == 0)
            if (tempPeakBin+i+1 < xlen)
                u = temp(tempPeakBin+i);
                d = temp(tempPeakBin+i+1);
                if ((u-d)>0)
                    temp(tempPeakBin+i+1) = minx;
                else
                    slopechange = 1;
                end
                i = i + 1;
            else
                slopechange = 1;
            end
        end
        peakBin(j) = tempPeakBin;
        peakMag(j) = tempPeakMag;
    end
end
```

Spectrogram5.m

```
% This matlab program creates an AVERAGE 3D PEAK spectogram of a STFT.
% It also prints out the N highest peaks for a FFT over time.
% Note: This program calls peaks.m which finds the max values of the fft.
%
% Spectrogram5(infileName, fs, type, k, numPeaks, fftSize, sorter)
%
%   infileName  :   A .wav file
%   fs          :   sampling frequency
%   type        :   'Log' for Logrithmic, 'Lin' for Linear
%   k           :   the number of frames we are going to average together
%   numPeaks    :   Number of Peaks to analyze
%   fftsize     :   fftSize (number of bins)
%   sorter      :   'Mag' will sort peaks by Magnitude, 'Bin' will sort peaks by bins
%
%   ex. array = Spectrogram5('new-B.wav', 44100, 'Log', 10, 6, 1024, 'Bin');
%
%   Ajay Kapur,     January 10, 2001

function array = Spectrogram5(infileName, fs, type, k, numPeaks, fftSize, sorter)

% Initialize Variables

winsize = fftSize;
hopsize = winsize*.5; % set hopsize to 50% of winsize

soundfile1 = wavread(infileName); % get sound file
sound(soundfile1, fs); % play sound

% find size of soundfile
k1 = whos('soundfile1');
soundsize = k1.size(1);

% remove DC offset
temp = mean(soundfile1);
soundfile1 = soundfile1 - temp;

% pre-initialize variables before loop
pos = 1;
frameIndex = 1;

while (pos+winsize) < soundsize
   s = soundfile1(pos:pos+winsize-1);
   s = abs(fft(s, fftSize));   % fft of s
   s = s(1:fftSize/2);         % s from 0 to Nyquist frequency
   sLog = log(s);              % take log of s and store in sLog
   array(frameIndex, :) = s';  % put s into 3d array with index frameIndex
   arrayLog(frameIndex, :) = sLog'; % put sLog into 3d array with index frameIndex
   pos = pos + hopsize;       % increment pos
   frameIndex = frameIndex + 1; % increment frameIndex
end

% AVERAGE SIGNAL

% Initialize Variables before taking Average
frameIndex2 = 1;
pos2 = 1;

while pos2 < frameIndex
   temp = 0;
   temp2 = 0;

   for i=1:k % get average for k frames
      temp = temp + array(pos2, :);
      temp2 = temp2 + arrayLog(pos2, :);
      pos2 = pos2+1;
   end

   temp = temp/k;
   temp2 = temp2/k;

   AvgArray(frameIndex2, :) = temp(1:(length(temp)));
   AvgLogArray(frameIndex2, :) = temp(1:(length(temp2)));

   frameIndex2 = frameIndex2+1; % increment
end

frameIndex2 = frameIndex2 -1; %decrement so can use for upperbound

% do peak search %
pos3 = 1;
while pos3 < frameIndex2
   % Initialize varaiables before peak search

   minLogValue = min(AvgLogArray(pos3, :)); % find min value in log data
   AvgLogArray(pos3, :) = AvgLogArray(pos3, :) + abs(minLogValue); % shift by min
value up, and then shift down again
   [peakBinl, peakMagl] = peaks(AvgLogArray(pos3,:), sorter, numPeaks);
   AvgLogArray(pos3, :) = AvgLogArray(pos3, :) - abs(minLogValue);  % shift back down
   LogPeakArray(pos3, :) = zeros(1,fftSize/2); % initialize LogpeakArray

   for (i=1:length(peakBinl))
      LogPeakArray(pos3, peakBinl(i)) = peakMagl(i);  % create 3d array of peaks for log
graph (Magnitude)
   end
```

```
  for (i=1:length(peakBinl))
     lPeakArray2(pos3, i) = peakBinl(i); % create 3d array of peaks  for log graph (bins)
  end

  [peakBin, peakMag] = peaks(AvgArray(pos3,:), sorter, numPeaks);

  PeakArray(pos3, :) = zeros(1,fftSize/2); % initialize peakArray
  for (i=1:length(peakBin))
     PeakArray(pos3, peakBin(i)) = peakMag(i); % create 3d array of peaks for linear
graph (magnitude)
  end

  for (i=1:length(peakBin))
     PeakArray2(pos3, i) = peakBin(i); % create 3d array of peaks  for linear graph (bins)
  end
  pos3 = pos3 + 1;
end
pos3 = pos3 -1; % decrement and use for printing

% PRINTING AND GRAPHING

if type == 'Lin'
  % print out Linear peaks %
  for (i=1:pos3)
     for(j=1:numPeaks)
        fprintf('%f\t', (fs*PeakArray2(i, j))/fftSize);
     end
     fprintf('\n');
     end
  waterfall(PeakArray(:,:,:)), title('Linear graph showing Peaks of Spectrogram'); % create
3d graph
  xlabel('Frequency (Bins)');
  ylabel('Time (Number of Frames)')
  zlabel('Amplitude');
end
if type == 'Log'
   % print out Log peaks %
  for (i=1:pos3)
     for(j=1:numPeaks)
        fprintf('%f\t', (fs*lPeakArray2(i, j))/fftSize);
     end
     fprintf('\n');
  end
  waterfall(LogPeakArray(:,1:15,:)), title('Logrithmic graph showing Peaks of
Spectrogram'); % create 3d graph
  xlabel('Frequency (Bins)');
  ylabel('Time (Number of Frames)')
  zlabel('Amplitude');
end
```

peaks.m

```
% This function finds peaks in FFT spectrum by finding the six highest values in the
%  FFT array.
%
% function [peakBin, peakMag] = hillClimbing(x, sorter, numPeaks)
% [peakBin, peakMag] = hillClimbing(x)
%
% returns arrays peakBin[] and peakMag[]
%

function [peakBin, peakMag] = hillClimbing(x, sorter, numPeaks)

  temp = x; % store x in temp

  xlen = length(x); % store length of x
  minx = min(x); % minimum value in x

  %plot(x);
  %figure;
  % Find 6 peaks
  for (j = 1:numPeaks)
    max = minx; % max is max amplitude
    for (i = 1:xlen)
      if (temp(i) > max)
        max = temp(i);
        tempPeakBin = i;
        tempPeakMag = temp(i);
      end
      %temp(i)
    end
    % Zero out peak
    temp(tempPeakBin) = minx;
    % Zero out peak in negative direction
    slopechange = 0;
    i = 0;
    while (slopechange == 0)
      if (tempPeakBin-i-1 > 0)

        u = temp(tempPeakBin-i);
        d = temp(tempPeakBin-i-1);
        if ((u-d)>0)
          temp(tempPeakBin-i-1) = minx;
        else
          slopechange = 1;
        end
        i = i + 1;
      else
```

```
            slopechange = 1;
         end
      end

      % Zero out peak in positive direction
      slopechange = 0;
      i = 0;
      while (slopechange == 0)
         if (tempPeakBin+i+1 < xlen)
            u = temp(tempPeakBin+i);
            d = temp(tempPeakBin+i+1);
            if ((u-d)>0)
               temp(tempPeakBin+i+1) = minx;
            else
               slopechange = 1;
            end
            i = i + 1;
         else
            slopechange = 1;
         end
      end

      peakBin(j) = tempPeakBin;
      peakMag(j) = tempPeakMag;
   end

   if sorter == 'Bin'
      % sort the Bins in ascending order using bubble sort
      for (i=1:numPeaks)
         for(j=1:numPeaks-i)
            if(peakBin(j+1) < peakBin(j))
               tempB = peakBin(j);
               tempM = peakMag(j);
               peakBin(j) = peakBin(j+1);
               peakMag(j) = peakMag(j+1);
               peakBin(j+1) = tempB;
               peakMag(j+1) = tempM;
            end
         end
      end
   end

end
```

Appendix

D

MATLAB Code used for Sound Simulation

Simulating a Ga stroke

BayanSimulator2.m

```
% This matlab program simulates the Bayan sound of a Tabla using the Plucked String
Model.
% Given the program a starting pitch and an ending pitch and it will morph
% between the two!!!!
%
% BayanSimulator2(StartfreqHz,EndfreqHz, iterations, fs)
%
%   StartfreqHz    :  frequency in Hz of beginning tone (rounding will occur)
%   EndfreqHz      :  frequency in Hz of end tone (rounding will occur)
%   iterations     :  duration of sound file
%   fs             :  sampling frequency
%
%   ex. signal = BayanSimulator2(150, 300, 10000, 44100)
%
%   Ajay Kapur,     May 11, 2001


function signal = BayanSimulator2(StartfreqHz,EndfreqHz, iterations, fs)

startN = fs/StartfreqHz; % Actual delay time
startN = floor(startN);  % round floor down to an integer

endN = fs/EndfreqHz; % Actual delay time
endN = floor(endN);  % round floor down to an integer

% See which N is bigger, StartN or endN
```

```
if startN > endN
   N = startN;          % N will be used to allocate space
   diff = startN - endN; % diff will be used for breaking apart iterations
else
   N = endN;            % N will be used to allocate space
   diff = endN - startN; % diff will be used for breaking apart iterations
end

duration = (1000*iterations/fs); % duration is length of sample (msec)

% x(n): Create noise (length depends on fs/freqHz) NOT ITERATIONS!!!
x = 2*rand(1, N); % fill x with random numbers
x = x- mean(x);   % take away DC, signal now between -1 and 1

% y(n): Create noise (length depends on iterations)
y = [zeros(1, N+1)]; % fill y with 0's from 1 to N+1


% x(n): pad with zeros
if iterations > length(x)
   d = iterations - length(x);
   x = [x zeros(1, d)]; % add zeros after original x
end

%%%%%%%%%%%% FILTERING %%%%%%%%%%%%

% initialize variables before Filtering
temp = 0;
signal = 0;
lengthYoffset = length(y)-1;
N = startN;
for j = 1: diff
   % When j == 1 only !!!!
   if j == 1
      b = iterations/(diff);
      b = floor(b); % indixies must be integer values
      for i = 1 : b
         i = floor(i);
         temp = x(i)+ (.5)*(y(N)+y(N+1));  % This line implements the filter function
         y = [temp, y(1:lengthYoffset)];   % update y with temp
         signal = [signal temp];   % create signal
      end % for loop
   else
      a = ((j-1)*iterations)/diff;
      b = (j*iterations)/diff;
      a = floor(a); % indixies must be integer values
      b = floor(b); % indixies must be integer values
      for i = a:b
         temp = x(i)+ (.5)*(y(N)+y(N+1));  % This line implements the filter function
      y = [temp, y(1:lengthYoffset)];   % update y with temp
      signal = [signal temp];   % create signal
    end % for loop
  end % if/else

  % adjust N value
  if startN > endN
    N = startN - j; % decrease N gradually by a constant j from initial value StartN to
endN
  else
    N = startN + j; % increase N gradually by a constant j from initial value StartN to
endN
  end
end


%%%%%%%%ENVELOPE%%%%%%%%

% Initialize Variables

amplitude = 1.0; % this is the amplitude of the attack
times = [.1 .1 .7 .1]; % this array holds the length of attack, decay, sustain, release

dur = length(signal);
attack = times(1)*dur;   % attack time
decay = times(2)*dur;    % decay time
sustain = times(3)*dur;  % sustain time
release = times(4)*dur;  % release time
slevelstart = .7;
slevelend= .69;
amplitude = 1;

% perform envelope

env = [linspace(0,amplitude, attack), linspace(amplitude, slevelstart, decay),
linspace(slevelstart, slevelend, sustain), linspace(slevelend, 0, release)];  % additional zero
padding

% padding : just in case
dp= length(signal) - length(env);
if dp  > 0
  for i = 1:dp
    env = [env 0];
  end
end
envsignal = env.*signal;   % make new signal with envelope
signal = envsignal;        % set signal to enveloped signal

sound(signal, fs);         % play sound
```

CoeficientFinder.m

```
function [b,a] = CoeficientFinder(n)

close all
%m = [0 0 1 1 0 0 .8 .8 0 0 .7 .7 0 0];
m = [0 0 1 1 0 0 .5 .5 0 0 .3 .3 0 0];

f = [0 .003 .005 .006 .008 .009 .0095 .0105 .011 .019 .020 .022 .023 1.0];
[b,a] = yulewalk(n, f, m);
b;
a;
%[h, w] = freqz(b,a,128);
%plot(f,m,w / pi, abs(h));

% .0055 .01 .021 : 121.2750, 220.5, 463.0500
[h, w] = freqz(b,a);
freqAxis = length((h))
%plot([1:freqAxis] / freqAxis, abs(h)), axis([0,100 / freqAxis, 0, 0.8]), hold on;
stem(0.0055, 1, 'g');
stem(0.01, 1, 'g');
stem(0.021, 1, 'g');
```

BayanSimulator3.m

```
% function signal = BayanSimulator3(N, burst, fs, tabs, split)
%
%   N       : initial burst for signal
%   burst   : iterations of signal
%   fs      : sampling frequency
%   tabs    : number of coeficients for filter equation
%   split   : will only work with input 1
%
% ex. signal = BayanSimulator3(200, 10000, 14000, 32, 1);
%
%
% Ajay Kapur May 15, 2001

function signal = BayanSimulator3(N, burst, fs, tabs, split)

% make random signal -1 ~ 1 with DC Compensation
% -------------------------------------------------------------------------
x1 = 2*rand(1,N);
x1 = x1 - mean(x1);
burst = burst - length(x1);
```

```
x  = [x1, zeros(1, burst)];

% generate noise and init. delay line
% make sure burst and delay line agree: burst >= delay line
% -----------------------------------------------------------------------


% Filtering

% .0000025 .000025 .00025 .0055 .01 .021 .415 : 121.2750, 220.5, 463.0500

k = split;
high = 0.0;
for i = 1:k


   m = [0 0 1 1 0  0 1 1 0  0 1 1 0 0 1 1 0 0 .5 .5 0 0 .2+high .2+high 0 0];
   f = [0 .000001 .000002 .000003 .000004  .00001 .00002 .00003 .00004  .0001 .0002 .0003
.0004  .003 .005 .006 .008 .009 .0095 .0105 .011 .019 .020 .022 .023 1.0];


   [b,a] = CoeficientFinder(tabs, m, f);
   if i == 1
      d = length(x) / k;
      d = floor(d);
      temp = x(1:d);
      temp2  = filter(b,a,temp);
      signal(1:d) = temp2;
   else
      c = (i-1)*length(x) / k;
      c = floor(c);
      d = (i)*length(x) / k;
      d = floor(d);
      temp2 = filter(b, a, temp);
      temp2;
      signal(c:d) = temp2;
   end
end

signal = 2*signal / max(signal);

%%%%%%%ENVELOPE%%%%%%%%

duration = length(x);
time = [.05 .1 .45 .4];
% Initialize Variables
a = linspace(0, 1, time(1)*duration);
d = linspace(1, 0.7, time(2)*duration);
s = linspace(0.7, 0.69,time(3)*duration);
```

```
r = linspace(0.69, 0, time(4)*duration);
env = [a d s r];

signal = signal .* env;
close all; % close graphs
sound(signal, fs);
% Play sound and plot
% -----------------------------------------------------------------------
```

BayanSimulator.m

```
% This matlab program simulates the Bayan sound of a Tabla
% In this program, the filter equation changes at every iteration
% from a lower sound to a higher sound!
%
% BayanStringSimulator(burst, iterations, fs)
%
%   burst         :   to intialize delay lines
%   iterations     :   duration of sound file
%   fs            :   sampling frequency
%
%   ex. signal = BayanSimulator(250, 10000, 8000)
%
%   Ajay Kapur,     May 11, 2001


function signal = BayanSimulator(burst, iterations, fs)

N = fs/burst; % Actual delay time
N = floor(N);  % round floor down to an integer

duration = (1000*iterations/fs); % duration is length of sample (msec)

% x(n): Create noise (length depends on fs/freqHz) NOT ITERATIONS!!!
x = 2*rand(1, N); % fill x with random numbers
x = x- mean(x);   % take away DC, signal now between -1 and 1

% y(n): Create noise (length depends on iterations)
y = [zeros(1, N+1)]; % fill y with 0's from 1 to N+1


% x(n): pad with zeros
if iterations > length(x)
   d = iterations - length(x);
   x = [x zeros(1, d)]; % add zeros after original x
end
```

```
%%%%%%%%%%%% FILTERING %%%%%%%%%%%%

% initialize variables before Filtering
temp = 0;
signal = 0;
lengthYoffset = length(y)-1;
high = 0.0;
low = 0.0;
mr = 0.0;

for n = 1 : iterations

   % Coeficient finder
   m = [0 0 1+low 1+low 0  0 1+low 1+low 0  0 1+low 1+low 0 0 1 1 0 0 .5+high .5+high 0 0
.2+high .2+high 0  0];
   f = [0 .000001 .000002 .000003 .000004  .00001 .00002 .00003 .00004  .0001 .0002 .0003
.0004  .003 .005 .006 .008 .009+mr .0095+mr .0105+mr .011+mr .019+mr .020+mr .022+mr
.023+mr 1.0];
   [b,a] = CoeficientFinder(5, m, f);
   close all;
   %high = .5 - n*(.5)/iterations; % high frequency amplitude (m) approach 1 (.5+.5) and
.7 (.5+.2)
   %low = n/iterations - .8; % low frequency amplitude (m) approach .2 (1 - .8)
   %mr = .02 - n*(.02)/iterations; % variable to move pole up to a higher area

   if (n == 1)  % special case 1 --- if n = 1, dont want to have negative indexing
       y(1) = a(1)*x(n);

   elseif (n == 2) % special case 2 --- if n = 2, dont want to have negative indexing
        y(2) = a(1)*x(n) + a(2)*x(n-1)+b(1)*y(n-1);
        % upate delay lines
        y(n-1) = y(n);
        x(n-1) = x(n);
   elseif (n == 3) % special case 3 --- if n == 3, dont want to have negative indexing
        y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+ b(1)*y(n-1) + b(2)*y(n-2);
        % upate delay lines
        y(n-2) = y(n-1);
        y(n-1) = y(n);
        x(n-2) = x(n-1);
        x(n-1) = x(n);
   elseif (n == 4) % special case 4 --- if n == 4, dont want to have negative indexing
        y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+a(4)*x(n-3)+ b(1)*y(n-1) + b(2)*y(n-
2)+b(3)*y(n-3);
        % upate delay lines
        y(n-3) = y(n-2);
        y(n-2) = y(n-1);
        y(n-1) = y(n);
        x(n-3) = x(n-2);
        x(n-2) = x(n-1);
        x(n-1) = x(n);

    elseif (n == 5) % special case 5 --- if n == 5, dont want to have negative indexing
        y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+a(4)*x(n-3)+a(5)*x(n-4)+ b(1)*y(n-1) +
b(2)*y(n-2)+b(3)*y(n-3)+b(4)*y(n-4);
        % upate delay lines
        y(n-4) = y(n-3);
        y(n-3) = y(n-2);
        y(n-2) = y(n-1);
        y(n-1) = y(n);
        x(n-4) = x(n-3);
        x(n-3) = x(n-2);
        x(n-2) = x(n-1);
        x(n-1) = x(n);
    elseif (n == 6) % special case 6 --- if n == 6, dont want to have negative indexing
        y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+a(4)*x(n-3)+a(5)*x(n-4)+a(6)*x(n-5)+
b(1)*y(n-1) + b(2)*y(n-2)+b(3)*y(n-3)+b(4)*y(n-4)+b(5)*y(n-5);
        % upate delay lines
        y(n-5) = y(n-4);
        y(n-4) = y(n-3);
        y(n-3) = y(n-2);
        y(n-2) = y(n-1);
        y(n-1) = y(n);
        x(n-5) = x(n-4);
        x(n-4) = x(n-3);
        x(n-3) = x(n-2);
        x(n-2) = x(n-1);
        x(n-1) = x(n);
    else  % everyother case
        y(n) = a(1)*x(n) + a(2)*x(n-1)+a(3)*x(n-2)+a(4)*x(n-3)+a(5)*x(n-4)+a(6)*x(n-5)+
b(1)*y(n-1) + b(2)*y(n-2)+b(3)*y(n-3)+b(4)*y(n-4)+b(5)*y(n-5)+b(6)*y(n-6);
        % upate delay lines
        y(n-6) = y(n-5);
        y(n-5) = y(n-4);
        y(n-4) = y(n-3);
        y(n-3) = y(n-2);
        y(n-2) = y(n-1);
        y(n-1) = y(n);
        x(n-5) = x(n-4);
        x(n-4) = x(n-3);
        x(n-3) = x(n-2);
        x(n-2) = x(n-1);
        x(n-1) = x(n);
    end  % end if else
    %accumulate output array;
    signal = [signal y(n)];
 end

%%%%%%%%ENVELOPE%%%%%%%%
```

```
% Initialize Variables

duration = length(x);
time = [.05 .1 .45 .4];
% Initialize Variables
a = linspace(0, 1, time(1)*duration);
d = linspace(1, 0.7, time(2)*duration);
s = linspace(0.7, 0.69,time(3)*duration);
r = linspace(0.69, 0, time(4)*duration);
env = [a d s r 0];


signal = signal .* env; % set signal to enveloped signal

sound(signal, fs);        % play sound
```

Thesis Concert Program Notes

"Electronic Tabla Project"
Ajay Kapur and Friends

Taplin Auditorium
Princeton University
8:00 pm April 25th, 2002

Participating Composers, Writers, Performers & Computer Engineers

Philip Blodgett
Richard Bruno
Perry Cook
Philip Davidson
Christoph Geiseler
David Hittson
Peter Lee
Adam Nemett
Jason Park
Tae Hong Park
Audrey Wright

Acknowledgements

Special Thanks to all those who have supported this project. Professor Scott Burnham who is chair of the Music Department, Chair David Dopkin of the Computer Science Department, Dean Peter Bugucki of the Engineering School, President Harold T. Shapiro, President Shirley M. Tilghman, Vice President of Student Life Janet S. Dickerson, Professor Perry Cook, Georg Essl, Philip Davidson, my roommates David Hittson & Adam Nemett, and my family.

Composition Technique

In preparation for this concert a unique composition technique was administered. An artist would come up with a kernel idea for a composition, and would then lead other participants to produce collective product. The first author listed is the author of the kernel.

Technical Staff:

James Allington
Kiriko Murakami
Ernesto Rivera
Mary Lee Roberts

Questions / Information:

Ajay Kapur
609-818-0730
akapur@princeton.edu

Preparation & Concentration

"Samsara"

This song describes our daily cycle and its correlation to the Buddhist idea of existing in our physical world – a world of passions, attachments, multiplicities, and in a certain sense, illusions. One goes through daily routines only to find that these routines must be repeated, endlessly. In much the same way, Buddhism teaches that we are caught in Samsara, the endless cycle of birth and rebirth. We dull and cover this suffering by keeping ourselves busy, by hiding, by becoming numb. The final bell ring represents the summoning to awake. This song was written in June of 2001 in the Zookjera house in Hopewell, NJ. The lyrics, written by Adam Nemett, are included.

Composers	Ajay Kapur, Dave Hittson, Peter Lee, Adam Nemett
Performers	Dave Hittson (vocals) Peter Lee (guitar) Tae Hong Park (bass) Audrey Wright (flute) Ajay Kapur (drum set, Nepalease bells)

"Arthur"

Arthur is a sample from the musical score being composed for the forthcoming film, *Art & The Instrument.* Written by Adam Nemett, the film is about an enigmatic art school janitor (Arthur) who passes away, but leaves behind the blueprints and electronic instruments for a new system of ritual worship – a system which uses music as its driving force. This song is an attempt to intertwine the kernels behind some of Arthur's rituals, drawing on musical ideas such as Inspiration, Attack, and Consonance. This song was written in December of 2001 in the Pennington house. With any luck, *Art & The Instrument* will be showing in theaters across the nation in Spring of 2003.

Composers	Ajay Kapur & David Hittson
Performers	Dave Hittson (vocals, acoustic guitar) Philip Blodgett (vocals, bass) Jason Park (guitar) Ajay Kapur (drum set)

"Manoj"

This electrono-mified, tribal groove song is based on the power of escalating energy flow. Sit back, relax and let a higher state of mind take hold. This song was written for and dedicated to a good friend experiencing difficult times. It was written in November of 2000 in the Zookjera house in Hopewell, NJ.

Composers	Dave Hittson, Ajay Kapur, Peter Lee
Performers	Perry Cook (DGtalFlow) Peter Lee (guitar) Tae Hong Park (bass) David Hittson (guitar) Ajay Kapur (drum set)

Manifestation

Introducing the Electronic Tabla (ETabla). The Electronic Tabla triggers both sound and graphics simultaneously. It allows for a variety of traditional tabla strokes and new performance techniques, while graphic feedback allows for artistic display. Philip Davidson will be controlling shapes, colors, and textures of graphics, reacting real-time to changes in mood, tempo and style of the performed ETabla music.

"Bupali Raag"

We introduce the Electronic Tabla by playing a North Indian classical piece. The Bupali Raag (based on the major pentatonic scale) will be played over a tin taal theka (16 beat pattern).

Performers	Audrey Wright (bansuri flute) Ajay Kapur (ETabla) – World Premiere

"Fire Fly and the Ghost"

This song is based on a meditation practice developed by Abraham Abulafia, a 13th century Jewish mystic. By permutating the letters of the names of God according to a specific formula of chanted vowels, Abulafia found an effective and ecstatic method of heightening his meditative concentration. The lyrics of this song follow Abulafia's model: each line ends in an extended vowel sound, moving along the progression, OH—AH—AY—EE—OO. In the next verse, the pattern begins again at a new starting place, AH—AY—EE—OO—OH, and so on. This song was written in June of 2001 in the Zookjera house in Hopewell, NJ.

Composers	Peter Lee, Dave Hittson, Adam Nemett, Ajay Kapur
Performers	David Hittson (vocals) Peter Lee (guitar) Audrey Wright (flute) Ajay Kapur (ETabla)

"In and Out with Samba"

This song features Christoph Geiseler on The Groovebox: Roland MC-505. 'In and Out with Samba' is a demonstration of its musical potential, and even more so, an indication of the limitless plane of the modern musical era. The name of the piece indicates the fluidity of the music and is emblematic of the versatility of moving between one genre of music and another. By paying close attention to the rhythm of the piece, one can perceive the juxtaposition between the Electronic Tabla and the pre-programmed Groovebox, but simultaneously understand how the two work together to mesh an electronic element with an improvisational impulse.

Composers	Christoph Geisler & Ajay Kapur
Performers	Christoph Geisler (GrooveBox) Ajay Kapur (ETabla)

"Dissonance Ritual"

This piece has four movements of electronic dissonance. The first movement starts out with a call-and-response between the ETabla, DgtlFlow, and Groovebox. In the second movement, this energy grows into a simple melodic theme played on guitar and bass. This flows into a third movement marked by high velocity and interaction. The piece ends on a fourth movement Drum n' Bass groove.

Composers	Ajay Kapur, David Hittson, Christoph Geiseler Tae Hong Park, Perry Cook
Performers	Perry Cook (DGtlFlow) Tae Hong Park (bass) David Hittson (guitar) Christoph Geisler (GrooveBox) Ajay Kapur (ETabla)

"Harmony Ritual"

This piece is a folk-rock piece centered around the resolution of dissonances into consonances. Both the melodies and harmonic progressions make use of these resolutions to most strongly convey "consonant sweetness". This song was written in July of 2001 in the Zookjera house in Hopewell, NJ, on David's Birthday.

Composers	Dave Hittson, Richard Bruno, Jason Park
Performers	David Hittson (vocals, acoustic guitar) Jason Park (guitar) Philip Blodgett (bass, vocals) Richard Bruno (vocals, acoustic guitars) Ajay Kapur (ETabla)

Biographies

Philip Blodgett
Philip A. Blodgett is philosophy major who plays many instruments including drum set, bass, and guitar. He currently serves as vocalist, instrumentalist, and percussionist for many musical groups including the Rhythm Method and Lack of Use.

Richard Bruno
Richard Bruno is a psychology major who just finished his thesis on melodic perception. He has been writing songs and playing in bands since junior high school, and is most recently a member of The Subcons. Richard also sings with the a cappella group, Shere Khan and has taken voice lessons here at Princeton.

Perry Cook
Perry R. Cook attended the University of Missouri at Kansas City Conservatory of Music from 1973 to 1977, studying voice and electronic music. He worked as a sound engineer and designer from 1976 - 1981. He received the BA in music 1985, and the BS in Electrical Engineering in 1986 from UMKC. He received a Masters and PhD in Electrical Engineering from Stanford in 1990. He continued at Stanford as Technical Director of the Center for Computer Research in Music and Acoustics, until joining the faculty of Princeton University in 1996, where he is now Associate Professor of Computer Science, with a joint appointment in Music. He has published nearly 100 technical/music papers, and presented lectures throughout the world on the acoustics of the voice and musical instrument simulation, human perception of sound, and interactive devices for expressive musical performance. Mr. Cook has performed as a vocal soloist and as a computer musician throughout the world, and has recorded Compact Disks on the Lyricord Early Music Series Record Label with the vocal group Schola Discantus.

Philip Davidson
Philip Davidson '02 is a senior majoring in computer science with a focus on graphics and visualization. He has worked with the display wall group, the committee for abstract events, the Nassau Weekly, and terrace club. He is presently interested in human interfaces for electroaudiovisual installation and performance, research into non-photorealistic rendering methods, and finding a job. He would like to thank Lansing NY, NYC, Jersey City NJ, Washington DC, and especially Duluth MN for their fine populace.

Christoph Geiseler
Christoph Geiseler is a sophomore in the politics department. He has played classical and jazz guitar for 7 years, played in a high-school jam-band, and now DJ-ing is his current fascination. In this concert, Christoph plays the Roland's MC-505 (drum-machine/synthesizer/sequencer/mixer/sampler) which can speak the musical language in thousands of different dialects, keys, tempos, and grooves. Can you imagine speaking Chinese with an Italian rhythm or Swahili with a French moue? The groovebox literally does the same thing with all forms of music by regulating pitch, tempo, dynamics, and, most importantly, melody, to create extremely adaptable and modifiable musical sequences. Christoph hopes this experience we share together in the Electronic Tabla Project will redefine or even spark a general interest and love for music in all its various forms and functions.

David Hittson
David is a music major with experience on the bass, guitar, piano, violin and voice. His musical life began at age two and he hopes it will continue far into the future. He would like to thank his parents, teachers and the music department. Also, he would like to congratulate Ajay on his impressive thesis accomplishment, and in general.

Ajay Kapur
Ajay is a computer science major who has taken 12 courses in the Music Department. He developed the Electronic Tabla as his senior thesis under the mentorship of Professor Perry Cook, and team effort of Georg Essl and Philip Davidson. Ajay has played drum set for 12 years, and has recently started playing other world percussion instruments such as djembe, tabla, and dolak. He has played in several bands since he was in high school, the most significant one to him being Zookjera, in which he was able to find himself as a musician. Ajay would like to thank his music teahers John Arucci, John Mastriani, Tony Branker, Bob Nolte, Rakesh Kumar Parihast, and Professor Perry Cook. Ajay plans to study Indian Classical music in India next year while continuing to create new instruments for musical expression.

Peter Lee
Classically trained, Peter started lessons on the piano at age 4, moving to the violin at age 8. At age 13, he heard his first Jimi Hendrix album and was hooked. He bought an electric guitar, started a band, and has been playing ever since. He would like to thank his teachers Stephen Wolosonovich, Michael Rosenbloom, and Bruce Arnold.

Adam Nemett
Adam is a religion major involved with the creative writing department. He is Co-Editor-in-Chief of the Nassau Weekly and co-founder of the student organization, Modern Improvisational Music Appreciation (MIMA). He served as lyricist and spoken-word vocalist for Zookjera. Currently, Adam is writing and directing a feature-length film, featuring music composed by Ajay and Dave and centered around Ajay's digital musical instruments.

Jason Park
Jason Park is a philosophy major who has played guitar for 8 years. He currently plays in the Rhythm Method with Phil Blodgett.

Tae Hong Park
Tae Hong Park received his B.E degree in Electronics at Korea University in 1994 and has worked in the area of digital communication systems and digital musical keyboards at the GoldStar Central Research Laboratory in Seoul, Korea from 1994 to 1998. He received his M.A. from Dartmouth's Electroacoustic Music Program in June 2000 and is currently a Ph.D. student at Princeton's Composition program. His current interests are primarily in musical and technical issues in computer and electroacoustic music, which include composition and research in multi-dimensional aspects of timbre.

Audrey Wright
Audrey can often be found playing with some musical group or other--be it the Princeton University Jazz Ensemble, the Klez Dispensers, the Emergency Funk Squad, or the Ellipsis Jazz Project... She was recently introduced to indian classical music, and loved it so much that she decided to try playing the bansuri flute!

Lyrics – By Adam Nemett

Samsara

Wake.
Darkness melting sun rays give way to day
Top-sheet tied and bleary eyed, can't be late.
Gotta make it there by eight
Water wash away the mind
The wrinkled lines and slumber signs.

Time it unwinds, my slowing,
Why is it always so hard to get going?

Fiend.
Peeling out to fill up my gasoline
and shoot the bull and pump me full of caffeine
but a break is seldom seen
Daddy's watching me behave
The norm has formed me to a slave.

We thrive on prizes
Can this be what being alive is?

Wake,
Fiend of Day,
Escape
Then we fly away.

Hide.
Fighting too much, sliding into the night.
The quiet sparks remind the dark of its light,
But the world has tied me tight
Might need some numbing
to call the King and let me sing.

Blankets thrown over sorrow
But will it still be there tomorrow?

Curse.
Gently trapped inside the endless curl
And let unfurl the sorry Samsara world.
So we dance the daily twirl,
Means lead to ends
Or does it just begin again?

Firefly and Abulafia’s Ghost

Catch a single firefly in a field and the buzz
is slow and lonely compared to the full feel
of the congregated party,
lit like lights winking on liquid.

Blur the aimed gaze
and let eyes out to play with the periphery
like two kids taking in
the carnival of a thousand syncopating winks.
When playing hide n’ seek in waving wheat,
it’s easy to lose and find focus.

*G**o**,*
*Find me a f**a**ll*
*Spring from the d**ay**, when you can’t s**ee***
*You see what’s tr**ue**.*
You know the way,
I’ll try and meet you.

*D**a**rk,*
*I like it that w**ay**,*
*Stray from the cit**y** to the w**oo**ds*
*The green lights gl**ow**.*
Lone but complete,
Make sure you go.

Chorus

Ghost roll
Disarray
Look up
All be laughing in the passion
Play fair
Head games
Intertwine
His names

*St**ay**,*
*lay close to m**e***
*even the m**oo**n looks like it kn**ow**s*
*just where we **a**re.*
‘Cross the cartoon
of lightning bug stars.

*S**ee**,*
*Open the b**oo**k*
*Watch it unf**o**ld until lights f**a**ll*
*And dim aw**ay**.*
Call from the Ghost,
Back to the day.

Under the muffle of midnight
those whispered secrets somehow
seep between cracks
of seamless trees,
stumbling and falling
like weary travelers
coming upon the cottage
of Another's open ear.

REFERENCES

[1] Courtney, David R. *Fundamentals of Tabla: Complete Reference for Tabla*, vol. 1, pp. 1-36 (Sur Sangeet Services, 1995).

[2] Rossing, Thomas D. *The Science of Sound*, pp. 373-374 (Addison-Wesley Publishing Company, 1999).

[3] See http://www.Tabla.com/articles/part1a.html

[4] Courtney, David R. "Repair and Maintenance of Tabla", *Percussive Notes*, vol. 31, no. 7, pp 29-36, (October 1993).

[5] Kippen, James. "Tabla Drumming and the Human-Computer Interaction", *The World of Music*, vol. 34, no. 3, pp 72-98, (1992).

[6] Wright, Mattew & David Wessel, "An Improvisation Environment for Generating Rhythmic Structures Based on North Indian 'Tal' Patterns". Available at: http://cnmat.cnmat.berkeley.edu/ICMC98/papers-html/wright-wessel-tal-demo.html

[7] Kaul, Vatsala. "Talvin Singh", Available at: http://www.india-today.com/ttoday/121998/boom.html

[8] Tsering, Lisa. "Talvin Singh Has Seen the 21st Century, and It's " O.K.", Available at: http://members.tripod.com/~LisaTsering/talvin.html

[9] Parihast, Rakesh Kumar. Ustad Alla Rakha Institute of Music. Bombay, India. Private Tabla Lesson (August 2002).

[10] Available at: http://chandrakantha.com/articles/indian_music/folk_music.html

[11] Courtney, David R. "Mridangam and Tabla: a Contrast", *PERCUSSIONS: Cahier Bimensiel d'Etudes et d'Informations sur les Arts de la Percussion.* (March/April 1993).

[12] Courtney, David R. "Basic Overview of the Tabla", *Modern Drummer*, vol. 17, no. 10, (October 1993).

[13] Vir, Ram Avtar. *Learn to play on Tabla*, pg. 35, (Punjab Publications, 1982)

[14] Available at: http://www.howstuffworks.com/mouse2.htm

[15] Available at: www.interlinkelec.com

[16] Available at: http://www-ccrma.stanford.edu/CCRMA/Courses/252/sensors/node8.html#SECTION00032000000000000000

[17] "BASIC Stamp Programming Manual," version 2.0, Parallax Inc., Available at www.parallaxinc.com

[18] Available at: www.midi.com

[19] Cook, Perry R., "Serial Communications Example", Available at: http://www.cs.princeton.edu/courses/archive/fall01/cs436/InputMIDI/midisoft.html

[20] “AppKit: Using the LTC1298 12-bit Analog-to-Digital Converter”, Parallax Inc.

[21] Raman, C.V., KT., F.R.S., N.L., "The Indian Musical Drums." Proceedings of the Indian Academy of Science, A. Vol 1. 1934.

[22] Steiglitz, Ken. A Digital Signal Processing Primer with Applications to Digital Audio and Computer Music. pg. 1-6 and 125-128, Addison-Wesley Publishing Company, Menlo Park, CA, 1996.

[23] Cook, Perry R. Music, Cognition and Computerized Sound: An Introduction to Psychoacoustics. The MIT Press, pg. 1-10, Cambridge, MA: 1999.

[24] Rossing, Thomas D. The science of sound. pg. 127. Addison-Wesley Publishing Company, Reading, Mass, 2nd edition, 1999.

[25] Hussain’s, Zakir, Remember Shakti, CD 2, Track 1.

[26] Essl, Georg & Perry R. Cook, "Banded Waveguides: Towards Physical Modeling of Bar Percussion Instruments," In Proc. Int. Computer Music Conf. (ICMC), Beijing, 22-28 October, pp 321-324, (1999).

[27] Essl, Georg & Perry R. Cook, "Sound Propagation Modeling in Solid Objects," submitted to IEEE Computer Graphics and Applications.

[28] Keller, J. B. & S. I. Rubinow, “Asymptotic Solution of Eigenvalue Problems," Annals of Physics 9, pp 24-75, (1960).

[29] Stam, J., and Fiume, E., “Turbulent Wind Fields for Gaseous Phenomena”, SIGGRAPH ’93, 369-376, (1993).

[30] Cook, Perry R. “Principals for Designing Computer Music Controllers”, ACM CHI Workshop on New Interfaces for Musical Expression, Seattle, (April 2000).