Visualization of the Rhythmic Phrases of the Tabla

Lavanya Kukkemane

Honors Thesis

2020

Table of Contents

Abstract

In Hindustani classical music, the tabla, a type of paired drums, is an important aspect of creating music and beats. A cadential technique often used by tabla players are *tihais*, which are improvised, rhythmic phrases that are repeated three times to signal the end of an improvisation, section, or piece. In order to extend audiences' understanding and enjoyment of a tabla performance, I built a program with a visualization system for displaying the rhythmic cadences in an improvised performance. The program is built using a visual programming language called Max, which provides an environment for real-time audio-visual processing and analysis. Recordings of tihais played by tabla master Sandeep Das are sent through the program, resulting in a visualization of the rhythmic pattern overlaid by the recording's frequency-domain waveform. While visualization systems for instruments are commonly developed in Western musical idioms, they are less so for Hindustani classical music. For audiences unfamiliar with the intricacies of Hindustani classical music, this program provides opportunities for a deeper understanding of the skills executed by tabla artists when performing.

Introduction

Digital visualization of music uses software and computing to create interesting visuals or animations in real-time based on various characteristics of the audio. A classic example is the visualizer in the Windows XP era Windows Media Player, which had a mesmerizing, dynamic visualizer accompanying any audio play. As technology grew more advanced, so did the complexity and customization of the visualizations. Any concert today will have vastly intricate visual displays for each song, transition, and encore. The voice recording app on our phones displays an instantaneous spectrogram as you record your voice, showing the loudness and frequency content of each phrase. However, even with the expansive collection of audio to visual software in the world, one niche has yet to be explored: visualization of the tabla, more specifically, tihais.

When I first contacted Professor Umezaki about potential research opportunities, he brought up a musician he worked with named Sandeep Das, a master performer of the tabla who wanted to find a way to visualize his performances to heighten his audience's experience. Having learned a form of Indian classical singing myself, this project was the perfect mix of my background in music and my interest in visual computing.

For those unfamiliar, the tabla is a pair of drums played with one's hands. It is an important aspect of creating music and beats in Hindustani classical, pop, traditional, and folk music. One musical device often used by tabla players is a *tihai*. Tihais are improvised, rhythmic phrases that are repeated three times. They can start at any point in a rhythmic cycle, but must end on

the *sam* (downbeat) of the cycle. Tihais can range from being quite simple to being extremely complex and are used to signal the end of a musical section or improvisation.

When musicians learn how to play tihais, they are often taught in a very methodical and structured manner. Because of the relatively strict nature of tihais, in that they must be repeated three times and end on the downbeat of the cycle, there are certain calculations and enumerations that performers use to correctly compose a tihai. It is within these restrictions that performers have the freedom and challenge to improvise tihais with unique colors and characters.

Das wanted to have audiences not just hear, but also see tihais being constructed in real time. In his performances, audiences interact with Das by cueing a starting point. At that point, Das would improvise tihais that would end on the first beat of the rhythmic cycle.

Those who have experienced performances by masters like Das will get a sense that the counting needed to execute the very complicated tihais he performs is not one that is done through more explicit, structured forms of counting. Rather, it is achieved through an embodied form of counting and division that he has internalized over years of practice and learning. Providing a visual display of the rhythmic cycle overlaid in real time with a visualization of the tabla sound would give more insight and understanding into the highly nuanced practice of improvising tihais.

Concepts

To fully understand the perspective through which this project was approached, it is important to understand the concepts fundamental to sound, signal processing, and Hindustani classical music. Below, these concepts and their relevance to this project are explained.

Sound

Fundamentally, sound is created from particles vibrating through the air or another medium in a certain way to create longitudinal waves.

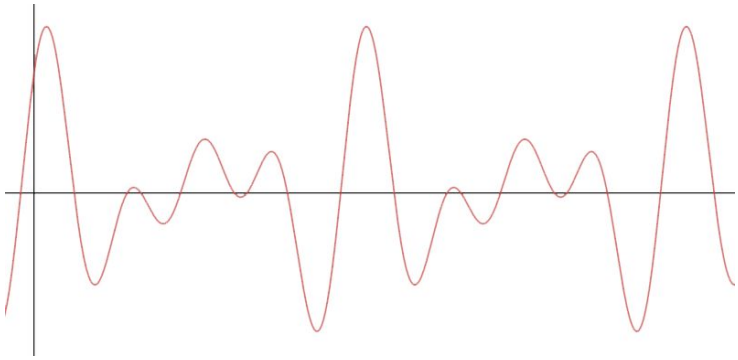


Figure 1: Time-domain graph of a sound wave.

In acoustics, sound is often represented as a time-domain graph or waveform, which shows how the air pressure changes with respect to time (Figure 1). A sound wave can be described using a combination of its properties, including frequency, amplitude, and timbre. Frequency, measured in Hertz (Hz), is the number of times that a sound wave completes a full sinusoidal cycle in one second. The human ear allows one to hear sounds between 20 Hz to 20,000 Hz. Amplitude, often represented in units of decibels (db), is related to the loudness of sounds. Timbre is the time-varying pattern created from all the partial frequency components of the sound wave, which affects the tone or color of the resulting sound. It explains why the same note can sound different when played on a piano versus on a guitar. For the purposes of this project, because I only used audio samples from a single instrument and the visualization is based on amplitude, I will only be working with the frequency and amplitude data.

Signal Processing

Signal processing is a way of representing and transforming signals. More specifically, audio signal processing is used to manipulate or analyze sounds for various purposes. Sound waves (as described above) are considered audio signals which can be used to process the sound digitally. However, digital representations of signals consist of discrete values. Since sound is continuous in nature, in order to use it on a digital platform, we need to take samples of the sound at discrete time increments. The rate at which a sound wave is sampled is called the sampling frequency or rate. The more often the sound is sampled, the more accurate the representation will be in terms of faithfully captured bandwidth. However, there is a trade-off in the amount of data that will need to be stored.

As mentioned earlier, a waveform representation of a sound wave (Figure 1) is a time-domain graph. A time-domain graph is helpful in seeing the entire audio signal and how it changes over time. However, to capture data for a specific frequency range, we need to convert the time-domain graph into the frequency-domain (Figure 2). A Fourier Transform takes a signal in the time-domain and converts it to its frequency domain, allowing us to look at the same audio data in a different way.

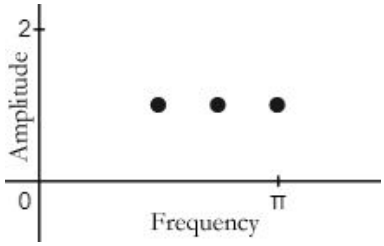


Figure 2: Frequency-domain graph of a sound wave.

The Fourier transform breaks down a complex signal into the simpler, sinusoidal waves. The sinusoidal waves resulting from a Fourier transform can be added together to recreate the original signal (Figure 3). The Fast Fourier Transform (FFT) is used specifically to calculate Fourier Transforms in which the time is discrete and has a finite duration.

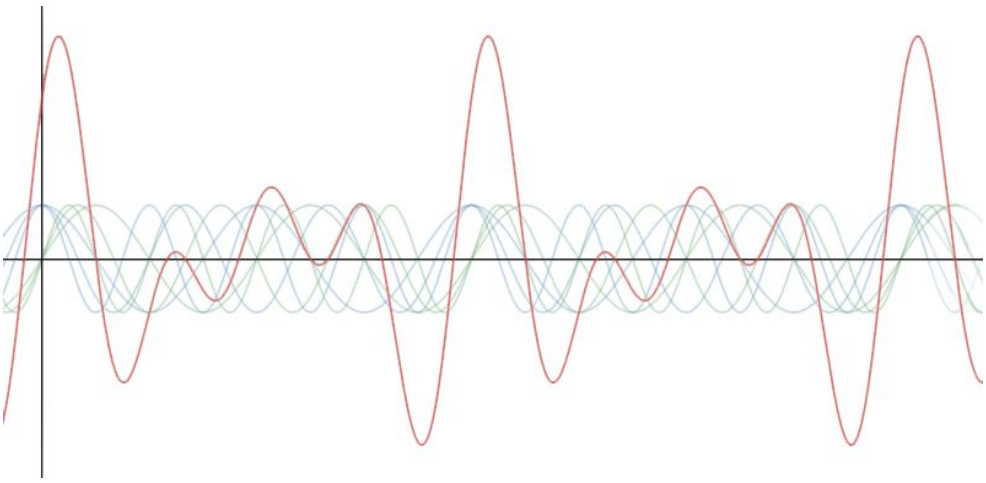


Figure 3: Example of a signal (in red) and the sine (in green) and cosine (in blue) waves that it is made from.

In addition, the amplitude from a range of frequencies, notably the spectrum of human hearing, can be separated from the rest of the signal using a low-pass filter, allowing us to keep only the data that is relevant.

Tabla and Tihais

The tabla is a type of percussive instrument originating from the Indian subcontinent and used in Indian classical music for both Hindustani and Carnatic styles. It has various functions in music, from accompanying other instruments or vocalists, to just being played by itself. Similar to the role that drums play in Western musical styles, the tabla is used to guide the rhythm and beats.

This project is focused on a specific aspect of a tabla performance called a tihai. Tihais are improvised, rhythmic phrases (or sections) that are repeated three times. They can start at any

point in a rhythmic cycle, but must end on the downbeat of the cycle, as they are used to signal the end of a musical section or improvisation. Tihais can range from being quite simple to extremely complex. Because they can be so complex, this project aims to help the average listener understand and appreciate these complexities.

Methodology

This section will go into the specifics of the project. I will describe the software and languages I used to build the software, then illustrate how I decided to approach the project.

Software and Languages

Max 8 from Cycling '74 is a program that is built for audio and visual signal processing. I chose to use this program upon recommendation from Professor Umezaki.

Max 8 also provides support for Javascript, which I used to process the audio data once it was filtered correctly and then transform that data into a visualization. I used OpenGL in Javascript to handle the visualization, as it allowed me to work with the audio data as vectors and matrices.

Project Approach

To structure the project into manageable parts, I segmented my project as follows.

First, I spent about a quarter learning more about the concepts surrounding this project and familiarizing myself with Max 8. My preliminary research centered around signal processing and tihais. I have a background in Indian classical music having learned it myself, so I only had to familiarize myself with the specifics of tabla playing and tihais. For signal processing and Fourier Transforms, I looked at a textbook[1] recommended by Professor Umezaki and other online resources. Concurrently, I learned how to use Max 8 using their developed tutorial sequence for beginner users.

After becoming familiar with the concepts and software, I discussed the specifics of the project with Professor Umezaki. The project was broken down into two main parts: (1) analyzing the rhythmic patterns of the tihai and (2) creating a visualization to represent those patterns. I decided to focus on creating a visualization first, as I wanted to build the connection between audio input and visual output. Once I was satisfied with some sort of a visualization based on the audio, I continued to analyze the audio signal more meaningfully and to enhance the visualization to reflect that analysis.

During this process, I received feedback in several ways. First, I met with Professor Umezaki on a weekly basis to update him on my progress, present the work I had done that week, and decide on next steps. This was helpful in getting technical help when I needed it and developing the

project's direction and goals as we progressed. In addition, I was able to meet and talk to Sandeep Das on a few occasions. I presented the system to him at several stages, which helped in ensuring the project was heading in an appropriate direction.

I set a deadline to have a functioning system by May 2019, as I was invited to present my project at UROP. After the symposium, I continued to work on fine-tuning the system, adding additional functionality and making a more user-friendly interface.

For a detailed breakdown of the project's timeline, please see the "Timeline" section of the thesis.

Timeline

In this section, I will outline the major stages of the project each quarter of research. Each subsection will state the goal for the quarter and describe, in detail, the development process and the state of the system at that point.

Fall 2018

Goal: Finalize project details and apply for UROP funding.

This was the first quarter of the project and our focus was on setting up a foundation for the rest of the project. I had approached Professor Umezaki about potential research opportunities and he told me about a project he had in mind involving music and integrated technology. I found this quite interesting, so I decided to join the project. Our first step was to get funding, as the software Professor Umezaki recommended would require either a paid subscription or a purchase of the software. We decided to apply for funding through UCI's Undergraduate Research Opportunities Program (UROP) and submitted a proposal in November. We also spoke more about the client, Sandeep Das, and discussed the requirements of the project.

Winter 2019

Goal: Become familiar with the concepts surrounding the project, create a concrete plan for project development, and become acquainted with the software.

First Visualization

We decided to start by bridging the gap between the audio input and visual output. I created a simple visualization by mapping the average amplitude of the audio input onto a 2 x 2 pixel black and white square in the main patcher[1]. For each average value, the patcher chose the next

[1] A patcher is the file type associated with Max 8 and the interface on which a program is developed, run, and interacted with. It can contain subpatchers, which can be used as a form of abstraction for complex patchers.

square in a clockwise direction and determined its brightness based on average amplitude across all frequencies. White represented the loudest end of the spectrum and black represented no sound.

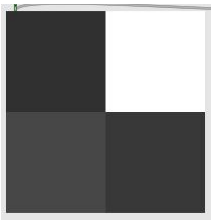


Figure 4: The first visualization of audio input.

To convert the audio input to visual output, the patcher averaged the amplitudes of all the frequencies in a 200 ms time period. Then, it scaled the average to an integer in the range 0 - 255. This is because color in Max 8 is defined using the RGB spectrum, which requires 3 8-bit bytes, each representing red, green, and blue channels, respectively. Shades of grey are created by using the same value for each color channel. I assigned the scaled average to each color channel, which gave an output of the amplitude for the given audio input.

To upgrade the visualization from a simple matrix, I mapped the output to a circular outline around the center of the matrix. I used the same black to white color scale to map the amplitude to squares, and drew the outline in a clockwise direction, using a *counter* object combined with sine and cosine operators to determine the correct pixel to color.

Fourier Transforms and Spoked Circular Visualization

Now that I had a basic visualization, I wanted to start developing something more meaningful. The first step was creating a spoked circle, akin to a wheel, instead of simply drawing around the border of a circle. Essentially, each spoke going clockwise would represent the amplitude of each sample of the audio input.

Upgrading the visualization required a more advanced visualization system. Previously, I had used a *jit.matrix* object for the visualization and a *pwindow* object to display the matrix as RGB colors. In order to have more control over the drawing process, I moved the visualization to *jit.gl.sketch*, which allowed me to use OpenGL functions to draw shapes and lines based on a standard coordinate system.

The new visualization system called for a change in the way the position of the current visual output was calculated. The *counter* option no longer made sense, as it was difficult to provide a high level of precision. Instead, I used two *cycle~* objects, which provided the x and y coordinates for each line. The *cycle~* object's speed could be changed in real-time, which allowed the amplitude to affect the speed at which the spokes moved around the circle. In addition to the coordinate calculation method, I changed the color range for the spokes from

black and white to black and magenta. With the scaled amplitude and coordinates, the patcher was able to send a few simple OpenGL commands to the *jit.gl.sketch* object to draw each spoke of the circle.

Lastly, I moved the visualization from an embedded window in the patcher to its own separate window. By the end of the quarter, the visualization presented as follows.

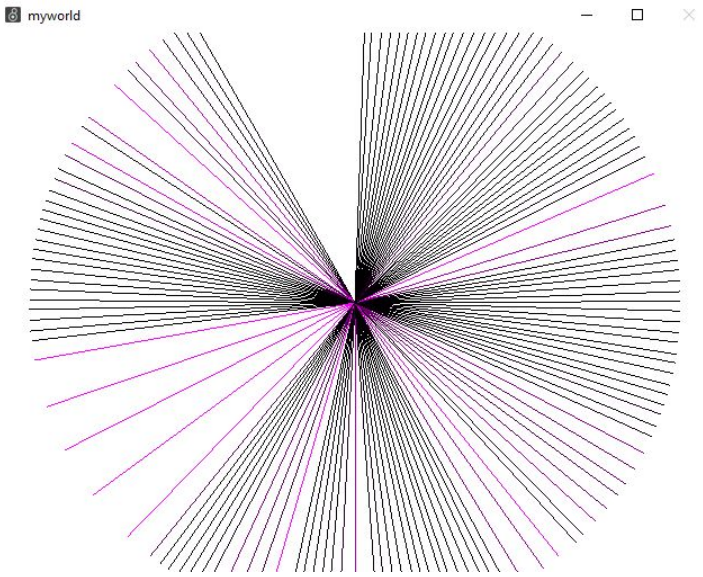


Figure 5: The spoked circular visualization.

Each 200 ms audio sample was represented as a single spoke in the circle. The speed at which the spokes moved around the circle as well as the color of the spoke were determined by the average amplitude of that sample. The louder the average amplitude was, the faster the spoke moved and the more magenta the spoke was colored. The visualization also cleared itself roughly every rhythmic cycle at the 12 o'clock mark.

Spring 2019

Goal: Create a working visualization system close to the original project goal and present the progress at UROP. In addition, show Sandeep Das the project and obtain feedback.

Mapping Frequency Bins

The next step in upgrading the system was to have meaningful data about the audio input. I wanted to continue to use the amplitude as the main variable, but add more complexity. To do this, I used a *pfft~* patcher to perform a Fast Fourier Transform on each buffered audio sample. This allowed me to analyze the amplitudes across frequency ranges within the sample. With the results of the FFT as a 256 bin matrix, I used *jit.dimop* to reduce the results, through averaging, into a 10 bin matrix.

To translate this into the visualization system, I wanted to create segments in the line for each audio sample so that each range of frequencies could be represented. To draw these new lines, I switched from solely using Max8's built in patcher and OpenGL functions to a Javascript module, which allowed me to use elements like for loops, functions, and variables more easily. The resulting visualization looked as follows.

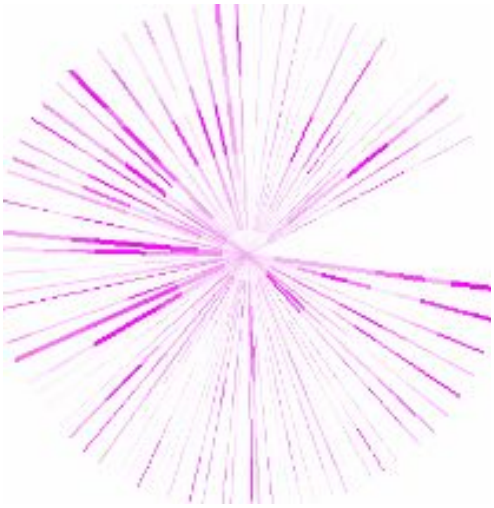


Figure 6: The spoked visualization with more complexity. Each line is made up of several line segments, which each have their own thickness and brightness.

From the lowest frequencies in the center to the highest frequencies at the edges, each line segment showed the amplitudes of the frequency ranges in the buffer of audio samples, mapping the amplitude to the brightness as before. The amplitude was also mapped to the thickness, with higher amplitudes resulting in thicker line segments. The Javascript code to draw the line segments was written as follows.

```
function draw()
{
	with (sketch)
	{
		if(x > 0.98 && y < 0.02){
			sketch.glclearcolor(1., 1., 1.);
			sketch.glclear();
		}

		gllinewidth(w);
		moveto(0., 0.);
		var i;
		for(i = 0; i <= 10; i++){
```

```
                mag = scaleValue(0.001, 3.5, 0.0, 3, matx.getcell(i))
                glcolor(255, 0., 255, mag);
                lineto(x*(i/10), y*(i/10));
            }
        }
}
```

Meeting with Sandeep Das

At the beginning of May, I had the opportunity to attend Professor Umezaki's graduate class to meet with Sandeep Das in person. This meeting was valuable in learning more about Das and his journey to becoming the master tabla player he is now. During the meeting, I presented the visualization described in the previous section, and was able to receive feedback from both Das and Professor Umezaki's experienced graduate students. Hearing their thoughts on the system helped me to see this project's position in the broader world of computer music integration and how to move forward with the project.

Linear vs Logarithmic, UROP Presentation Prep

Because hearing is measured in units of decibels, which operate on a logarithmic scale, it only made sense to change the way the frequencies were grouped into bins. Instead of assigning an equal amount of frequencies to each bin, I changed the formula to map the output from *jit.catch~* to a logarithmic scale using powers of 2, as shown below.

```
Line segment 0: bin 0
Line segment 1: bin 1-2
Line segment 2: bin 3-7
Line segment 3: bin 8-15
Etc.
```

Once the frequencies were grouped into bins, the patcher calculated the average amplitude for each bin and converted it to decibels. Since microphone input measures sound pressure levels, the decibels were calculated using the formula below:

$$20\log_{10}\left(\frac{averageMagnitude}{maxMagnitude}\right) \quad where\ maxMagnitude = \frac{FFTsize}{2}$$

Formula 1: Decibel calculation formula.[3]

With the magnitudes mapped to decibels, each bin was represented as a line segment of a single line, as described previously, with the average amplitude across all frequencies for the sample

mapped to the speed of the entire line. These were the final changes made before the UROP presentation. At this point in the project, the visualization looked as follows.

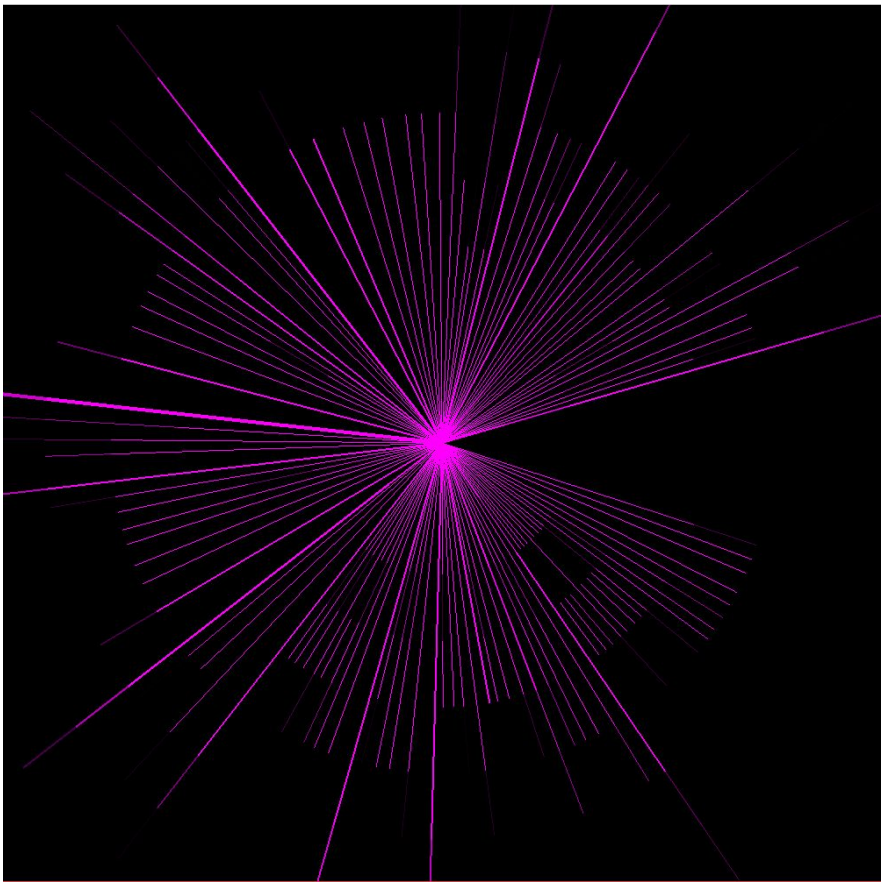


Figure 7: Visualization from UROP Presentation.

The Max patch below paired with the code segment created the essence of the system I presented at UROP.

UROP Presentation

The last milestone this quarter was my presentation at UROP. I presented the visualization system described above (Figure 7) as a live demo and explained some of the background and concepts behind the project.

Fall 2019

Goal: Finish making final touches on the visualization and work towards wrapping up the project.

Updates to Current Visualization System

Several updates were made to the visualization system this quarter. First, I changed the velocity of the spokes to no longer be dependent on the amplitude. Instead, it was a constant value that

could be changed by using the spacebar. The speed could be set by pressing the spacebar at the beginning and end of a rhythmic cycle, indicating the amount of time that the visualization should take to make a complete circle. At this point in the system, once the speed was set, I had to restart the system by toggling the on-off switch in order to draw at the new speed. I kept this additional step on purpose because the speed affected the timing used to clear the visualization every cycle, resulting in inconsistent clearing.

I fixed this problem in the second update by changing the way the visualization cleared itself. While it served its purpose, clearing the visualization every cycle created a very sharp moment in an otherwise smooth-flowing visualization. Such an abrupt change did not mimic the way people experience music, in that we don't suddenly forget the previous cycle of music as soon as a new one begins. Because of this, I changed the visualization to fade overtime instead, so that by the time the lines make their way around the circle, the previous cycle's lines are completely faded.

I implemented this in the patcher by running the visualization through a feedback loop. The Javascript module, which was triggered by a *trigger* object, drew the current line to an empty to a *jit.gl.world* canvas. The patcher saved the full, current visualization's canvas (minus the new line) from the previous line and multiplied by a factor of approximately 0.99 to apply a small fade to the old lines. Then, the two canvases were added together to become the new full visualization. By multiplying the whole canvas by 0.99 each time a new line was drawn, the older lines slowly faded as new lines were drawn.

Horizontal Visualization and UI

The last major change I added was an accompanying horizontal visualization that drew the same line segments, but horizontally across the screen. Because the horizontal visualization required a different set of coordinates, I added a separate coordinate to the Javascript module's input to specify the x position that the line would be drawn at. The horizontal visualization needed to show the last two cycles of the audio input rather than just one, which required some trial and error to achieve. At first, I used a *counter* or *cycle~* object to increment the x value, but it was difficult to control the speed with a *counter*, and complicated to convert the negative values to a positive integer when using the *cycle~*. Finally, I used a combination of logical if statements, *counter*, and a *phasor~* object to get the correct x position.

In addition to features, I improved the user interface and organization of the patcher. I reorganized the layout to make it easier to read and added color coded controls and labels to interactable elements. Lastly, I added some preset values to ensure that few adjustments would need to be made for a user to start working with the program.

Finished Project

In this section, I will thoroughly describe the project at its final stage by following the audio input's journey through the system, from the signal processing to the visualizations. I have included the final Javascript code and screenshots of the main patcher below. The files can be downloaded at the following link: *https://bit.ly/3ds652E*. In addition, I've written a user manual that can be found at the end of this paper.

1. **AUDIO INPUT**: The audio input comes into the program in two ways: an audio file or microphone input. The audio is a signal type. It goes through a *pfft~* patcher with an FFT-size of 512 and overlap factor of 1, meaning that each FFT is performed with 512 discrete samples and no part of the signal is analyzed twice. The patcher then takes the results of the FFT and converts them from cartesian to polar units, allowing us to look at the amplitudes of the frequencies separately.

2. **PFFT**: Half of the results from the *pfft~* go to a *jit.catch* object, which is triggered every 5ms by the *qmetro* object. Because half of the transform is redundant, only 256 bins are used. The patcher sends this 1 x 256 matrix to the Javascript module and triggers the module to draw the line.

3. **COORDINATES**: Meanwhile, the main patcher is continuously running the *phasor~* object, which sends out sawtooth-like ramps that feed into a *cycle~* object or the *correctXPos* subpatcher. The former keeps track of the current x and y position for the circular visualization. The latter calculates the x position for the horizontal visualization. Both are used every time the Javascript module is triggered to draw a line.

4. **SPEED**: The speed at which the *cycle~* object runs affects the speed at which the lines rotate around the circle or move across the horizontal visualization. At any point while the visualization is running, the speed can be dynamically changed using the spacebar. For example, if a certain performance piece is at a faster rhythm, the speed of the visualization can be changed so that each rhythmic cycle of the performance (or two cycles of the performance for the horizontal visualization) corresponds to a single cycle of the circular visualization. The program will calculate the difference in time between the latest two spacebar presses and match that amount of time to one walk around the circle.

5. **JAVASCRIPT**: The Javascript module takes in 4 variable inputs (and 1 trigger), which are the (x,y) coordinates for the circular visualization, (x) coordinate for the horizontal visualization, and the 1 x 256 matrix of amplitude results from the Fourier transform.

    a. First, the average of the matrix is calculated. Then, the matrix is converted to a logarithmically scaled matrix of size 8. Each bin (0 - 7) in the resulting matrix holds the average of the next 2^{bin_num} values from the input 1 x 256 matrix. This

allows the lower and mid frequency sounds to take up more space in the visualization, while meshing the higher frequencies that are harder for humans to distinguish.

b. For each bin in the new matrix, the average amplitudes are converted to decibels and scaled to the appropriate RGB value, assuming the maximum decibel amount = 256 / 2 = 128. The line segment for each bin is drawn for both the circular and horizontal visualizations. For the circular visualization, the frequencies for each line segment increase moving out from the center. For the horizontal visualization, the frequencies increase moving up.

c. For each visualization, the Javascript module outputs a black background with the single line made up of 8 appropriately colored line segments.

6. **FADED EFFECT**: Each time there is a new output from the Javascript module, the previous visualization is multiplied by a factor of 0.98-0.99, depending on the speed of the visualization, to create a faded effect. Then, it is added to the output from the Javascript module. This creates a faded effect in the visualizations, so that the newest data is the brightest and the older data becomes progressively darker, creating a visual feedback loop. In the main patcher, both the current line being drawn and the composite (aggregated) visualization can be seen. The composites for each visualization are also drawn in separate windows.

From the viewer/audience's perspective: The two visualizations will show the drum beats based on their amplitude and pitch (from the frequency analysis). The visualization can be synced to the tempo of the performance, allowing the audience to intuitively understand when each cycle of the performance begins and ends. This in turn allows audiences to see that the musician is masterfully playing the tihais in following the tempo and downbeats. The visualizations compliment the aural experience of the performance, showing the patterns and beats that have been and are being played.

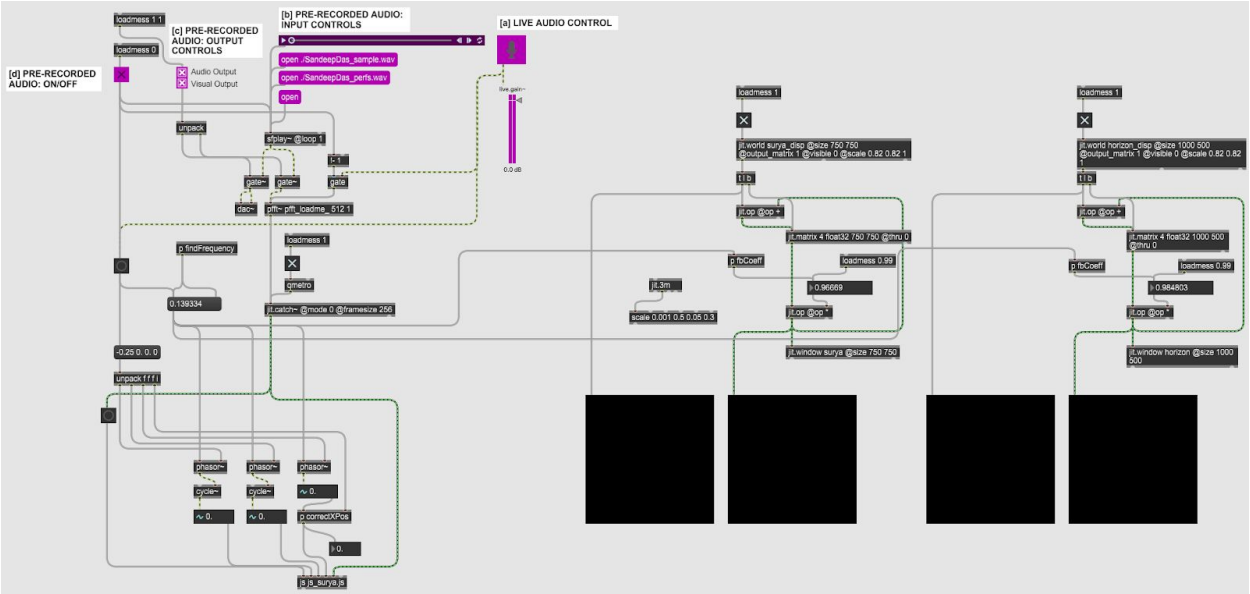


Figure 8: Final patcher.

JAVASCRIPT MODULE

```
autowatch = 1;
inlets = 6;
outlets = 1;

//inputs
//surya coordinates
var x = 0.;
var y = 0.;
//horizon coordinates
var xHor = 0.;
//audio inputs
var matx = new JitterMatrix(4, "float32", 256);

//other
var avg = 0.;

var suryaSketch = new JitterObject("jit.gl.sketch", "surya_disp");
var horizonSketch = new JitterObject("jit.gl.sketch", "horizon_disp");
with(suryaSketch)
```

```
{
	suryaSketch.blend_enable=1;
	suryaSketch.depth_enable=0;
	suryaSketch.glclearcolor(0., 0., 0.);
	suryaSketch.glclear();
}

with(horizonSketch)

{
	horizonSketch.blend_enable=1;
	horizonSketch.depth_enable=0;
	horizonSketch.glclearcolor(0., 0., 0.);
	horizonSketch.glclear();
}

draw();
refresh();

//receives phase values in coordinate form as input from bang
function msg_float(f)
{
	if(inlet == 1){x = f;}
	if(inlet == 2){y = f;}
	if(inlet == 3){xHor = f;}
}

//receives amplitude matrix as input from bang
function jit_matrix(mname)
{

	if(inlet == 4)
	{
		inmat = new JitterMatrix(mname)
		matx = linToLog(inmat);

		var sum = 0.;
		for(var i = 0; i < inmat.dim; i++)
		{
			sum = sum + inmat.getcell(i)[0];
```

```
            }
            avg = scaleValue(0.0, 2., 1.0, 5.0, (sum/256));
      }
}

//takes incoming matrix of size 256
//returns log-scaled matrix of size 8
function linToLog(matrix)
{
      var log_avg = new JitterMatrix(1, "float32", 8);
      var i = 0;
      var j = 0;
      while(j <= 8)
      {
            var sum = 0.;
            var count = 0.;
            var exp = Math.pow(2, j);
            while((i < exp) && (i < matrix.dim))
            {
                  sum = sum + matrix.getcell(i)[0];
                  count = count + 1.;
                  i++;
            }
            log_avg.setcell1d(j, sum/count);
            j++;
      }
      return log_avg;
}

//takes a magnitude and the maximum magnitude
//returns the magnitude mapped to decibels
function toDecibels(mag, max)
{
      return 20 * Math.log(mag/max);
}

//takes an input value and input/output ranges
//returns scaled output value
function scaleValue(imin, imax, omin, omax, value)
{
```

```
	return ((omax - omin) * (value - imin) / (imax - imin) + omin);
}

function draw()
{
	//draws a line on the circular visualization
	with (suryaSketch)
	{
		suryaSketch.glclearcolor(0., 0., 0.);
		suryaSketch.glclear();

		moveto(0., 0.);
		var i;
		//draws each segment of the line
		for(i = 0; i <= matx.dim; i++){
			suryaSketch.gllinewidth(avg);
			mag = toDecibels(matx.getcell(i), 256/2);
			mags = scaleValue(-150., 0., 0., 3., mag);
			glcolor(255, 0., 255, mags);
			lineto(x*((i+1)/matx.dim), y*((i+1)/matx.dim));
		}
	}

	//draws a line on the horizontal visualization
	with (horizonSketch)
	{
		horizonSketch.glclearcolor(0., 0., 0.);
		horizonSketch.glclear();

		moveto(xHor, -1);
		var i;
		//draws each segment of the line
		for(i = 0; i < matx.dim; i++){
			horizonSketch.gllinewidth(avg);
			mag = toDecibels(matx.getcell(i), 256/2);
			mags = scaleValue(-150., 0., 0., 3., mag);
			glcolor(255, 0., 255, mags);
			lineto(xHor, 2*(i+1)/matx.dim-1);
		}
	}
```

```
}

//"main" : runs the module by calling all necessary functions
//is called with a bang from the patcher
function bang()
{
	msg_float();
	jit_matrix();
	draw();
	refresh();
}
```

Future Implications

This project adds to the spectrum of ways that music visualization can help overcome cultural or knowledge-based barriers when listening to different types of music. The ideas from this project have the potential to develop much further into more complex analysis and visualization. One way would be to analyze the beats based on their pitches or duration by pattern matching with systems trained with pre-performance data. In addition, the program could differentiate the different types of beats by organizing them into some sort of priority from weak beats to stronger ones. The tabla sound vocalized as “dha” might have more significance in understanding a certain pattern than the sound “tak”, which could be reflected as such in the visualization. Das also brought up the idea of developing an artificial intelligence that could predict the tihais he was playing. Through machine learning techniques, the program would be able to calculate and recreate the tihai in real time after hearing Das’s first improvised repetition. This would allow Das to “compete” with the AI, adding an additional layer of interaction and fun to his performance, further engaging his audience. The essence of the final project shows potential for further development in music visualization for the tabla and other instruments and musical styles less common in the current field of audio-visual software.

The Surya System - User Manual

The Surya System is designed to help create a complementary and immersive experience of a live tabla performance. By providing real-time visualization of the tabla beats, the system allows viewers to better understand the performance, enhancing their experience of the music.

How it Works

Essentially, the software takes a live or pre-recorded tabla performance and maps the amplitude of the audio waves to a visualization system. A more detailed process is described below:

1. The system receives audio input from a live microphone or pre-recorded audio file.
2. The audio input's signal (currently in the time domain) is sent to a pfft~ object, which performs a Fourier Transform on the signal. This transforms the time domain signal to a frequency domain signal.
3. The frequency domain signal, as well as coordinates for both the circular and horizontal visualizations, are sent to a Javascript module.
4. This module takes the frequency domain signal, which is in a linear scale, and transforms it into an 8-bin logarithmic scale. This 8-bin representation of the frequency domain is drawn as a spoke or line in the visualization.
5. Each of the bins' amplitude is mapped to its own line segment, creating a single line made up of 8 segments representing the lowest to highest frequency ranges. The lowest frequencies are in the middle of the circular visualization and the bottom of the horizontal visualization.

How To Use It

Below is a step-by-step guide on how to use the system to create a real-time visualization of a tabla sound. All interactive buttons on the patcher have a magenta-colored background, and each set of controls is labeled (see *Controls* below). These labels are used for clarification in the steps below.

If using live audio:

1. Turn on the Microphone from the [a] Live Audio Control panel. Ensure that the pre-recorded audio toggle [d] is off.
2. Turn on the main audio on the bottom right corner (should turn blue when it is on).
3. Start playing music! The visualizations will appear in the patcher itself, as well as separate windows called "horizon" and "surya."
4. To control the speed of the visualization, press the spacebar at the start of every downbeat; the visualization will adjust accordingly.

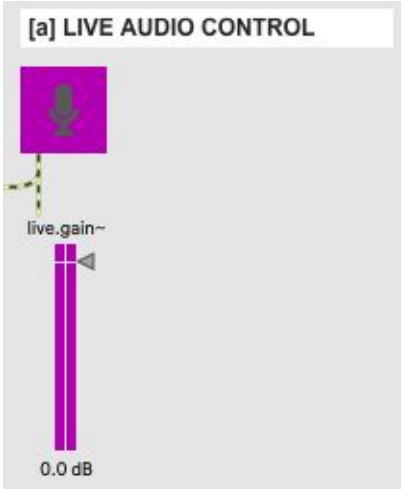
If using pre-recorded audio from an audio file:

1. Turn on the audio on the bottom right corner (should turn blue when it is on).
2. Choose an audio file from the pre-recorded audio controls [b]. Two presets are included, or choose "open" to use a file from your computer.
3. By default, both the audio and visual will be produced. If you would like to change this, do so in the output menu [c].
4. Toggle the [d] on/off button to start the visualization and audio. The visualizations will appear in the patcher itself, as well as separate windows called “horizon” and “surya.”
5. To control the speed of the visualizations, press the spacebar at the start of every downbeat; the visualizations will adjust accordingly.

Controls

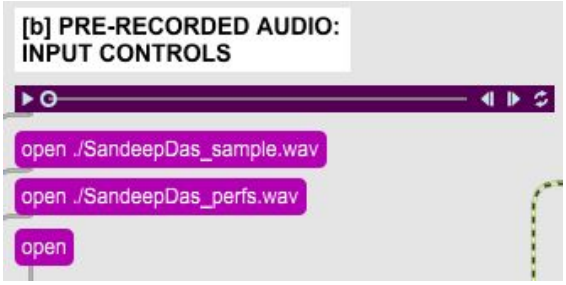
[a] Live Audio Control

Turn the microphone input on or off and adjust the gain amount. *Note: If pre-recorded audio toggle is on and the overall audio is on, the microphone will show as on, but live input will not be processed.



[b] Pre-Recorded Audio: Input Controls

Select a preset file or a custom file from your computer. If desired, you can pause/play or move to a certain part of the audio file.



[c] Pre-Recorded Audio: Output Controls

Select what output you would like the system to produce. The system can produce just the visual, just the audio, or both.

[d] Pre-Recorded Audio: On/Off

Once the pre-recorded file is chosen, toggle this button to start the system.

Acknowledgements

I would like to thank my research advisor, Professor Kojiro Umezaki, for trusting me with this project and guiding me throughout the entire process. This project would not be possible without his constant support and working with him made the experience that much more fun. I would also like to thank Sandeep Das, whose idea of visualizing his improvisations was the inspiration behind the whole project. He was kind enough to give us his feedback and ideas during the project's development and provide us with audio samples and a live demonstration of his tabla performance. Lastly, I would like to thank my family and friends for everything they have done along the way, from encouraging me to attending my UROP presentation to helping me edit this thesis.

I have always loved music, art, and computers. Growing up, I learned classical Indian Carnatic singing and have always loved being around and creating both music and art. During my time at UCI as a computer science major, I saw the potential to combine my interests in the arts with computer science. This research was a remarkably perfect amalgamation of these interests, and I consider it lucky that I had the opportunity to work on such a project.

References

[1] Rockmore, Daniel N., and Alan Gleason. *Who Is Fourier?: a Mathematical Adventure*. Language Research Foundation, 2012.

[2] Kulkarni. *Frequency Domain and Fourier Transforms.* www.princeton.edu/~cuff/ele201/kulkarni_text/frequency.pdf.

[3] LaTeX Equation Formatting from https://www.codecogs.com/latex/eqneditor.php

[4] *Cycling '74*, cycling74.com/.

[5] "Max 8 Documentation." *Cycling '74*, docs.cycling74.com/max8.