Audio Spatialization Toolkit for Personal Music Production

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Antti Meriluoto

M.Eng Field Advisor: Hunter Adams

Degree Date: December 2024

ABSTRACT

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Audio Spatialization Toolkit for Personal Music Production

Author: Antti Meriluoto

Abstract:

In the age of the Digital Audio Workstation, music production has arguably never been so accessible and democratic. Purchasing effects pedals or plugins for audio manipulation, however, can add up quickly in expenses. To counteract this, one can enjoy a significant degree of agency over the music processing they employ in personal productions through the use of open source software and programming languages like Python. In this report and project, I examine a toolkit I developed using Python to encode a sense of spatial dynamics into a mono audio track through its manipulation with some trivial digital signal processing into a stereo format. The report will go into some of the key design principles of the toolkit, such as its parametric design, enabling an iterative approach to its finetuning based on qualitative assessments of its creative effects.

EXECUTIVE SUMMARY

For my ECE M.Eng Final Design Project, I set out to develop a parametrizable tool with Python that would enable me to encode a sense of motion and space into music I was producing. In order to do so, I first explored the techniques and theory behind spatial audio. I then sought to implement them using a Python script that would be able to take a .wav file as an input and output a stereo .wav file containing the encoded dynamics. All development was done with Python and required just a handful of libraries: Scipy, for extracting the amplitudes of every sample from an input wavefile, Numpy, for allowing numeric manipulation of the audio signals, Matplotlib, for analyzing the output waveforms and providing a graphical display of the dynamics being encoded, and TKinter, for creating the user interface. The end result was an application that could spatialize an audio file according to the motion of a particle representing a source of sound along the xy-plane in 3D space. The aim for this project was to create a tool that I could use and expand upon in the future to produce music in my sparetime.

Another tangential goal of mine was to make the tool easily accessible for others to use should its results prove compelling. For this reason, sticking with a plug-and-play architecture like Python proved doubly useful: it trivialized the project set-up for myself and potentially for others.

Ultimately, the tool is able create a compelling sense of motion on the xy-plane using a mono track input, though the effect still leaves much room for improvement in terms of conveying motion along the y-axis. I would like to expand upon this tool in the future to potentially explore motion along the z-axis, as well.

ACKNOWLEDGEMENTS

I would like to thank Hunter Adams and Bruce Land for their continued support and assistance with the ideation and development of this project.

BACKGROUND

Project Conception

The idea for this project came to me when I was taking ECE 5760: Advanced Microcontrollers, taught by Professor Adams and Professor Land. The first lab of this course involves sonifying the famous Lorenz Attractor, a set of ordinary differential equations that are a flagship example of the field of mathematical chaos in that they are sensitive to infinitesimaly small differences in initial conditions. The Lorenz Equations contain a few parameters that, within a certain range, result in a very elegant, bimodal pattern when numerically integrated, shown below.

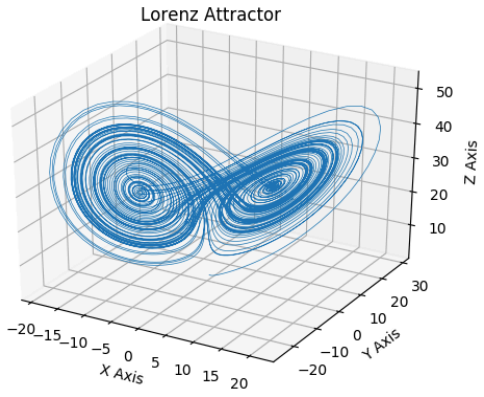

Figure 1: The Lorenz Attractor visualized in a 3D Plot

Professor Adams' and Professor Land's class explores the use of digital systems engineering as a medium for artistic expression, so the M.Eng project seemed like an appropriate opportunity to further explore this concept. In the same semester I was taking this class, I also began songwriting as a hobby. The title of one of the songs I wrote that semester was "Strange Attractors", a term used in mathematical chaos that encompasses the set of ordinary differential equations, including the Lorenz Attractor, with a characteristic sensitivity to infinitesimally small noise in their initial conditions. I felt like it would be satisfying to listen to said song and identify a facsimile of the titular system in the motion of certain parts and instruments, namely an organ.

Modeling our Auditory Perception of Space

Spatial audio techniques make use of the Head-Related Transfer Function, which is a function in signal processing that characterizes how our ears perceive spatial information from a source of sound along with the signal processing involved in that process, which is a function of the shape of the listener's head and ears as well as the position of the source. I gained most of my theoretical basis for this project from discussions with Professor Land and my M.Eng Advisor, Professor Adams, although there exists plenty of academic literature to more formally characterize the Head-Related Transfer Function[2].

Based on the width of a user's head, parametrized by the variable w, and the sampling frequency of a mono audio track, one can replicate the delayed arrival of sound between an audio source and a user's left and right ears by phase shifting the original signal between the left and right stereo output channels.

$$sampling\ frequency\ =\ \omega\ =\ 44100\,Hz;\ sampling\ period\ =\ T\ =\ 1/\omega$$

$$speed\ of\ sound\ =\ c\ =\ 343\,m/s;\ w\ =\ 0.1\,m$$

$$channel\ delay\ =\ w\ /\ c\ \approx\ 0.00029\,s$$

$$number\ of\ samples\ to\ shift\ by\ =\ 0.00029\ /\ T\ \approx\ 13$$

Figure 2: Calculations showcasing the derivation of the phase shift in the most extreme case where an audio signal is directly to one side of the listener

In a digital setting, the phase shift is discretized by a certain number of samples, by which we shift the audio signal between the left and the right channel.

Another element that plays into our perception of space in sound is the difference in frequency that we perceive between what we hear in our left ear and what we hear in our right ear. The ear that is less dominant in perceiving an audio signal will experience a moderate amount of frequency attenuation on the sound it picks up because the head separating the ear from the source of sound functions like a low-pass filter. Lastly, one can capture the impacts of distance on the way sound is perceived by attenuating the amplitude of the mono audio source to give the illusion of its removal in space from the listener.

I use classic example of a Resitor-Capacitor Low Pass Filter circuit as inspiration for its implementation in my code. I include the derivation of my code below:

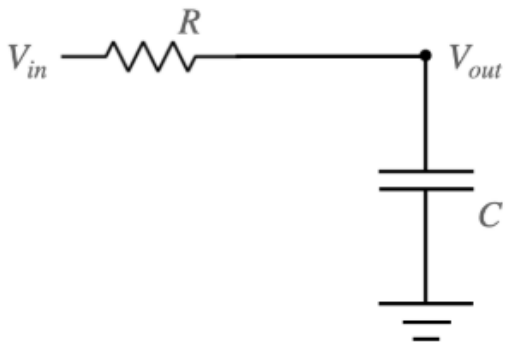

Figure 3: Circuit diagram of a Resistor-Capacitor Low Pass Filter[1]

Following the derivation outlined on Professor Adams' site[1], we eventually arrive at the discretized equation for our digital low-pass filter. We begin with the observation that the current across the Resistor and the Capacitor must be equivalent by virtue of them being in series:

$$i_r = V_{in} - V_{out}/R \;=\; i_c \;=\; C dV_{out}/dt$$

Solving for dV_{out}/dt, we find:

$$dV_{out}/dt \;=\; (V_{in} - V_{out})/(RC)$$

To discretize our equation, we use the following notation:

$$V_{out}(t) \;=\; V^N$$

$$V_{out}(t \;+\; \Delta t) \;=\; V^{N+1}$$

Since our timestep *dt* is quite small (1 / 44100 Hz $\approx\; 2.27\; x\; 10^{-5}\; s$), we can approximate $V_{out}(t \;+\; \Delta t)$ using a first-order taylor approximation, i.e.

$$V_{out}(t \;+\; \Delta t) \;\approx\; V(t) \;+\; [dV_{out}/t]_t \Delta t$$

If we simplify using our discretized notation, we have the following expression:

$$V^{N+1} = V^N \;+\; \Delta t\; (V_{in} - V^N)/(RC)$$

This equation dictates that the filtered sample, V^{N+1} in our output channel at $t=N+1$, should be the sum of the previously filtered sample, V^{N}, and the difference between it and the raw sample in our input audio file, V_{in}, multiplied by our sampling period and divided by the constant RC. This constant provides us with the ability to parametrize the intensity of our lowpass filter, and modulate its intensity based on the position of an audio source in space.

The last signal processing technique we need to apply is amplitude modulation based on the distance of an audio source from the listener's ears. We can approximate this by multiplying the samples from the raw audio signal by a simple constant that scales linearly with distance.

By conbining phase shifts between our left and right output channels and modulation of the frequency and amplitude of samples from a mono audio as a function of a sound's position over time in spacesource, we can manipulate the audio into a stereo output to create the illusion of its motion in the space surrounding a listener.

PLANNING

In order to produce metrics of success for the project that could account for the difficulty in empirically validating the end deliverable of an audio file due to its qualitative nature, I had to decide on a few core design principles that I would adhere to throughout the development process. One of the major motivations of the project was maximizing creative control over the tool. Since the quality of the output would be measured based on the qualitative experience of listening to the output audio, I wanted to make the program readily parametrizable, so that I could iteratively finetune the toolkit and rapidly assess the resulting changes. Based on the relaxed inquiry I made into the Head-Related Transfer Function, the features I selected for parametrization were the width w of the user's head which impacted the maximum degree phase

shift experienced between the two channels, the lowest cutoff frequency for the low-pass filter I would implement, and the coefficient of amplitude attenuation that would be scaled by the distance between each ear and the audio source.

Another motivation behind this project was eventually releasing it as an open-source application once I had achieved a level of quality with which I was satisfied. As such, I decided that a key aspect of the deliverable would be a user interface that could expedite the utility of the tool, without necessarily requiring the user to know how to operate the code. Because I was implementing in Python, I decided that TKinter would be a good API to create the GUI.

Lastly, I knew the development process was going to be very iterative on account of the qualitative nature of the deliverable. As such, I decided that a Jupiter Notebook would be a great medium for development, since I could introduce functionality step-by-step while caching the results of earlier implementations.

DEVELOPMENT

Before I began implementing the required functionality using an iterative development approach, I first defined a list of deliverables I would work towards that would culminate in the end deliverable of an audio track encoded with motion mimicking the Lorenz Attractor. The deliverables I decided upon were as follows:

1. A demonstration of a stationary audio track directly to one side of the user
    a. This would entail implementing a phase shift and amplitude attentuation
2. A demonstration of a dynamic audio track orbiting around the user
    a. This required the implementation of the low-pass filter

3. A demonstration of a dynamic audio track following the path of the Lorenz Equations
4. An animation of a 3D plot containing an output audio that would visualize the motion of the audio source in the space around the listener

Static Audio

To help me develop the first deliverable, Professor Adams linked me to some code demos on his site that showcased how to use Python to modify a mono audio signal into a stereo format. From this demonstration, I gained familiarity with the libraries I would need to complete this project, namely NumPy, for the array data structure used to reshape the mono=signal into a stereo format, and SciPy, for the Wavefile library, enabling .wav file data parsing and manipulation. In the process of development, I used a few audio clips: one of a single drum beat and another of an 852 Hz pure tone. I passed these clips through my prototypes and qualitatively assessed their functionality. My first experiment with stereo audio simply involved taking a mono signal and outputting a stereo audio file with one channel muted and the other channel set to the source audio. This provided a sanity check for my development and clarified how I would go forward to implement something more complex.

In order to implement a demonstration of an audio source manipulated to sound like it was positioned to the side of the listener, I incorporated logic conducting the phase shift across the two channels on top of some trivial amplitude modulation in the less dominant channel. The first demonstration was of an audio signal emerging from the right. As established in the background section, I had to populate the left channel with attenuated samples from the original source, 13 timesteps apart; the right channel was a simple one-to-one copy of the original source. This implementation required the first thirteen samples of the left channel to be populated with

zeros. Likewise, the right channel required additional padding of thirteen zeros appended to its end so that the shape of the output arrays matched, a requirement of the Wavefile library.

During the early stages of the development process, there were a few interesting bugs I encountered that derived from integration errors with the libraries I was using. Periodically, when I would apply a spatial effect to an input Wavefile, I would encounter an error from NumPy complaining about a dimensional mismatch pertaining to the samples I was manipulating. I ultimately ascertained that, certain Wavefiles, when unpacked by the Wavefile, would resolve to two-dimensional NumPy arrays as opposed to a single-dimension. This was likely due to the fact that some of my input audio files were in a stereo format without me realizing. I implemented logic to reshape the audio data into a one-dimensional format in these instances as a workaround.

Dynamic Audio

The stationary deliverable already proved quite compelling, so I had a lot of enthusiasm for adjusting the output to something more dynamic. As a preliminary proof of concept, I first developed an algorithm to stitch together the spatialiazed output of an audio source for three different positionss: a drum beat to the right of the listener, a drum beat directly ahead of the listener, and a drum beat to the left of the listener.

In order to spatialize an audio source based on the position of a particle over time, I created a function that would take in as an input a set of x and y-coordinate values at each timestep or sample. The function would then iterate through the input audio data sample by sample, indexing the current position of the particle based on the index of the loop, then populating the left and right channels accordingly. Determining how to delegate each sample to the channels at every given timestep would be a function of the particle's position of space. In

the end, through consultation with Professor Adams, I implemented a Ring Buffer data structure with three different pointers to facilitate the process of populating the left and right channels while executing the spatial sample shift. I describe the data structure using the diagram below:

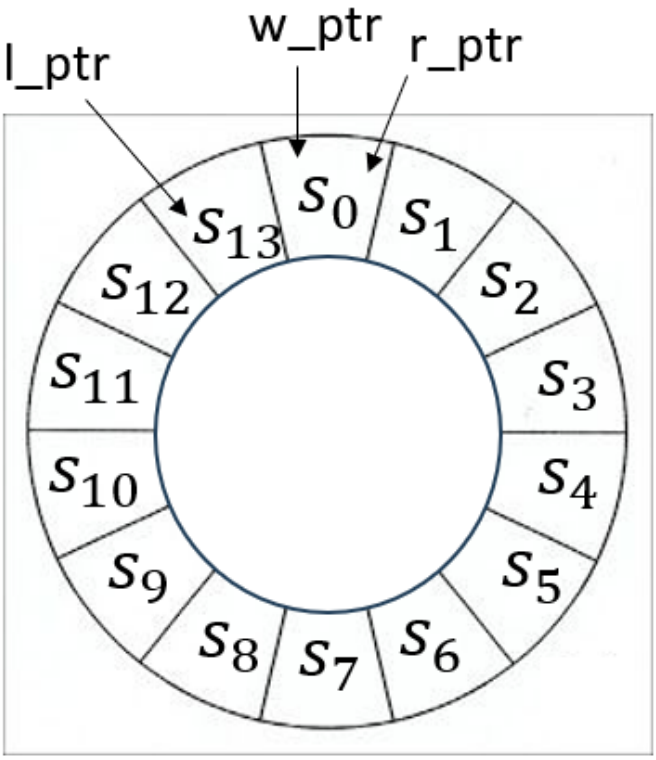

Figure 4: The ring buffer used to implement the sample shift, representing an audio source directly to the left of the listener at N=13

The ring buffer uses a write pointer to point to the next element in the 14 wide array to be overwritten; the size of the array was determined by the maximal delay a sample would experience from one channel to the other, which was parametrized by the approximate width of the user's head, as mentioned in the background section of the report. The buffer also uses a "left" and "right" read pointer to point to the next element in the array to append to the left and right channels respectively. In short, while iterating through the n-th sample in the raw audio, I

populate the ring buffer with the n-th sample and pass in the position of the audio source at that point in time. The ring buffer then adjusts the left and right pointers accordingly based on the time delay necessary to reflect the distance differential of the source to each ear, and to poll the correct sample in the buffer at timestep n.

Though the ring buffer trivialized the implementation of the dynamic spatialization, the audio effect left something to be desired. It was at this point when we began discussing the addition of a low-pass filter to the effects chain. Similar to the sample shift, the low-pass filter would be applied to the least-dominant channel based on the position of a particle in space. I created a function such that, based on the angle of a particle and the origin, Θ, the intensity of the low-pass filter would change for the left channel linearly from its maximum at $\Theta = 0$ to its minimums at $\Theta = \pm\ \pi/2$; the function was similar for the right channel, for whom filtering was active for particles on the left of the y-axis.

One optimization that I later implemented was parametrizing the spatialization algorithm to accept a lambda function that could take in as an argument the timestep of the sample and return a particle's position. This enabled me to reduce the overhead associated with precomputing position data.

Animation and GUI

Combining the audio spatialization results with animation to visualize the audio source maneuvering around the listener involved the use of a few additional libraries, such as Matplotlib and MoviePy. Matplolib provided the framework for plotting the motion of an audio source in a 3D space, while MoviePy enabled the automated synthesis of audio and video.

One of the primary objectives of the project was to develop with the intention of releasing the spatialization toolkit as an open source application, meaning I needed to incorporate

a GUI. ChatGPT proved an invaluable resource in providing a quick TKinter interface that I could incorporate with the spatialization library to function as a frontend for the toolkit.

RESULTS

The project ultimately was successful in producing an effect I could quickly introduce to an audio source for music production, though the evaluation process was primarily qualitative. I found that the quality of the effect varied based on the nature of the input audio file. For example, when spatilizing an 852 Hz pure tone, I noticed slight buzzing effects in the less-dominant channel that I was unable to debug. These effects were not present when I used a recording of an instrument, such as a guitar or an organ. I was personally most satisfied with the sound of the organ in conjunction with the spatialization effect, specifically when orbiting the head and when modeling the Lorenz Equations. Based on initial feedback I gathered from others, the ability to garner a sense of motion is variable person to person. I discuss the reasoning behind this along with some potential improvements in the next section.

I have included the input and output waveforms for a puretone positioned immediately to the right of the user below. These visualizations are helpful in noting the phase shift and amplitude attenuation effects of the toolkit.

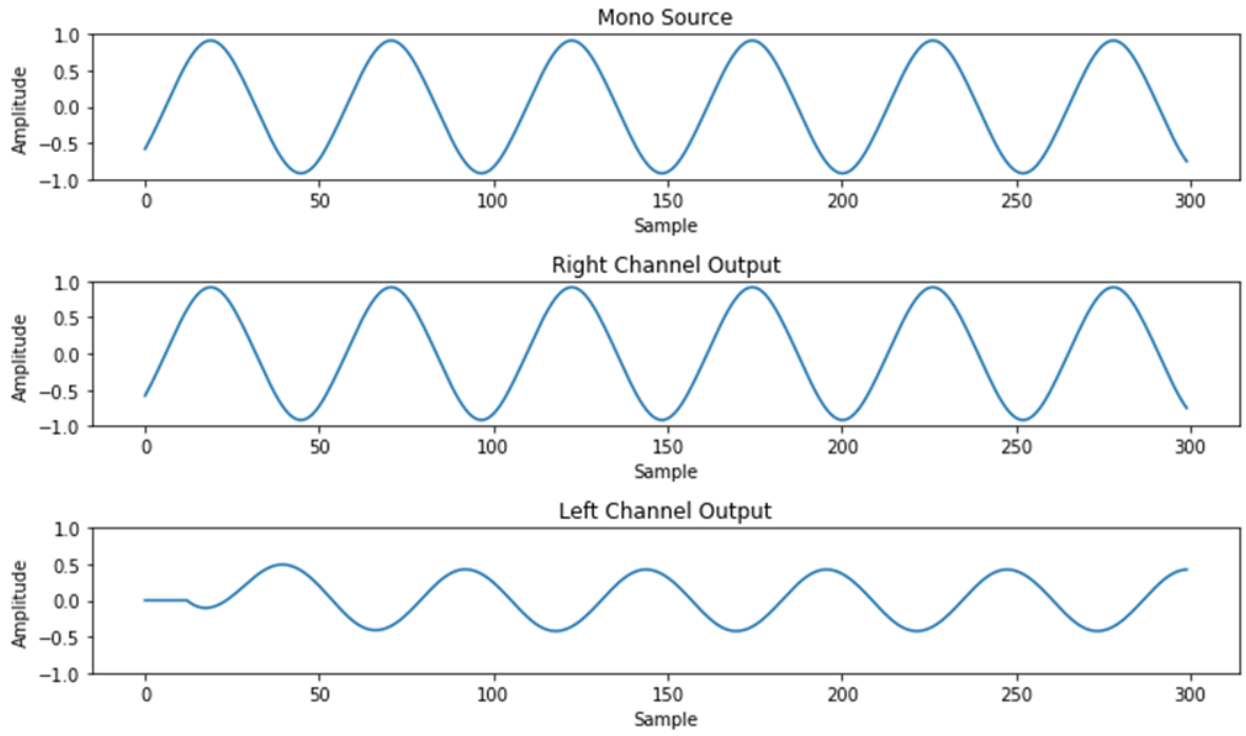

Figure 5: The state of each channel's output from an 852 Hz pure tone source to the immediate right of the user

In terms of the requirements I laid out originally, I was able to parametrize the design of the toolkit to facilitate any future design iterations I might like to implement. I was also able to create a simple user interface, intended as the starting point of the frontend for an eventual application or website.

FUTURE WORK

Though I am happy with the progress I made on this project, I still believe there is much work to be done in the future to fully realize the vision I set out for this project. Most trivially, I would like to modify the GUI so that the application can be hosted on a website, potentially leveraging .HTML and CSS.

Another aspect of the spatialization effect that left something to be desired was the audio's motion in the y-direction. It proved simple enough to create the illusion of an audio's position motion in the x-direction because I could reflect motion of this fashion with the sample shift algorithm and the low-pass filter. Motion in the y-direction was thus simply achieved by amplitude attenuation relative to the sound's distance from the listener along; much of the illusion relied upon the "hallucination" of the audio's motion in the y-direction to complete the mental model of the audio source revolving around the listener's head.

On top of optimizing motion in the y-direction, I would very much like to determine how to process the audio in order to convey a sense of motion in the z-direction. I am unfamiliar with the theory behind how such motion is conveyed, so this would likely be a very involved and rigorous task. Nonetheless, it would completely fulfill the project's intended scope and allow for the maximum degrees of freedom with which I could encode dynamics into the music.

The last addition, and perhaps the most difficult to implement, that I could imagine would be to enable full customization from the user-level of the differential equations describing the motion of the song. I could envision using some kind of large language model to generate a numerical integrator for a custom set of ODEs that gets passed into the spatialization function as a lambda function.

CONCLUSION

Over the course of the M.Eng design project, I managed to create a toolkit that provides infrastructure I can use to program a sense of motion through space into a stereo audio track from a mono input. Despite this, through my personal qualitative analysis of the project's outputs and through the feedback I've received from others, I have ascertained that there remains much

work to be done for the project in order to produce a more compelling tool that I would distribute online. The sense of motion in the y-direction is rather understatred, and the project does not at all tackle motion in the z-direction. In truth, I feel like I underdelivered over the course of this semester for a project whose premise already left a lot of leeway in terms of evaluation, due to its qualitative, as opposed to quantitative, measure of success, engendered by its function as artistic tool. I hope to make up for the shortcomings of the technical side of this project by integrating the tool with music I write in the future, as I do believe the effect it produces offers a compelling change to the listening experience of an otherwise stagnant track.

REFERENCES

1. Adams, H. (n.d.). *Data Display*. Van Hunter Adams.

https://vanhunteradams.com/Pico/Helicopter/Display.html

2. C. Avendano, V. R. Algazi, R. O. Duda, & D. M. Thompson. (2001). THE CIPIC HRTF DATABASE. *U.C. Davis*.

https://www.ece.ucdavis.edu/cipic/wp-content/uploads/sites/12/2015/04/cipic_WASSAP_2001_143.pdf

APPENDIX

1.1 spatializer.py

```
import numpy as np
from scipy.io import wavfile
import math

# ========================================
# Constants
# ========================================

t_per_sample = 0.00002267573

sample_T = 1 / 44100 # Period of sample (in s)

# ========================================
# Parameters to tune for audio processing
# ========================================

g = 0.6 # Amplitude multiplier

# We use cartesian coordinates to model the motion of a point
audio source
# around a listener centered around the origin

# The left ear is located at (-10, 0), the right ear at (10, 0)

left_ear_pos = np.array([-10, 0])
right_ear_pos = np.array([10, 0])

class RingBuf:
    # Constructor (initializer)
    def __init__(self):
```

```
        self.buf = [0. for _ in range(14)]
        self.l_ptr = 0
        self.r_ptr = 0
        self.w_ptr = 0

    def write(self, val : float, span):
        self.buf[self.w_ptr] = val
        left_sample_shift = 0
        right_sample_shift = 0
        if span >= 50:
            left_sample_shift = int(0.0003 * ((span - 50) / 50)
/ sample_T)
            self.r_ptr = (self.w_ptr) % len(self.buf)
            self.l_ptr = (self.w_ptr - left_sample_shift) %
len(self.buf)
        else:
            right_sample_shift = int(0.0003 * ((50 - span) / 50)
/ sample_T)
            self.l_ptr = (self.w_ptr) % len(self.buf)
            self.r_ptr = (self.w_ptr - right_sample_shift) %
len(self.buf)
        self.w_ptr = (self.w_ptr + 1) % len(self.buf)


    def read(self):
        #span = get_span_from_point(pos)
        return (self.buf[self.l_ptr], self.buf[self.r_ptr])

    def get_read_ptrs(self):
        return ((self.w_ptr - self.l_ptr) % len(self.buf),
(self.w_ptr - self.r_ptr) % len(self.buf))


def get_span_from_point(point : np.ndarray):
```

```
    point = np.array(point)
    #dist = np.linalg.norm(left_ear_pos - point)
    r_dist = np.linalg.norm(right_ear_pos - point)
    l_dist = np.linalg.norm(left_ear_pos - point)

    diff = abs(r_dist - l_dist)
    diff = min(diff, 20)

    if r_dist <= l_dist:
        span = 50 + 50 * (diff / 20)
    else:
        span = 50 - (50 * (diff / 20))

    return span


def low_pass_divisor(theta):
    #print(theta)
    assert 0 <= theta <= math.pi
    if 0 <= theta <= math.pi / 2:
        right = 1
        slope = (16 - 1) / (math.pi / 2)
        left = 1 + slope * -(theta - math.pi / 2)
    elif math.pi / 2 < theta <= math.pi:
        slope = (16 - 1) / (math.pi / 2)
        right = 1 + slope * (theta - math.pi / 2)
        left = 1

    #if theta == math.pi / 2:
    #    print(f"(left, right) {left}, {right}")
    return left, right
```

```
#=== Models particle traveling with circular motion around
listener with a period of 4 seconds
def circular_orbit(t):
    return (-10 * np.sin(2 * np.pi * t / 4), 10 * np.cos(2 *
np.pi * t / 4))

def lorenz_orbit(t):
    return (0,0)


def spatialize_over_time(input_path : str, output_path : str,
f):
    sample_rate, data = wavfile.read(input_path)
    audio_data = np.array(data, dtype=np.float32)

    if data.dtype == np.int16:
        audio_data = audio_data / 32768.0  # 16-bit audio
normalization
    elif data.dtype == np.int32:
        audio_data = audio_data / 2147483648.0  # 32-bit audio
normalization
    elif data.dtype == np.uint8:
        audio_data = (audio_data - 128) / 128.0  # 8-bit audio
normalization

    sample_T = 1 / sample_rate # Period of sample (in s)

    # -100 = 0.0003
    # 0 = 0
    # 100 = 0.0003

    #print(audio_data.shape)
```

```
    # Known Issue: Sometimes audio data will duplicate samples
into two columns

    if(len(audio_data.shape) > 1):
        print("whoopsy daisy")
        audio_data = audio_data[:, 0:1]
        audio_data = audio_data.flatten()

    channel_buf = RingBuf()
    #left = [0 for _ in range(13 + len(audio_data))]
    #right = [0 for _ in range(13 + len(audio_data))]
    left = []
    right = []

    left_filtered = []
    right_filtered = []

    dt = sample_T

    for i in range(len(audio_data)):
        #pos = pos_lst[i]
        pos = f(i * dt)
        span = get_span_from_point(pos)
        right_span = span
        left_span = 100 - span

        point = np.array(pos)
        r_dist = np.linalg.norm(right_ear_pos - point)
        l_dist = np.linalg.norm(left_ear_pos - point)
        #print(point)
        #return

        theta = math.atan2(pos[1], pos[0])
```

```
        # Find theta for vector reflected across x-axis if theta
> pi (logic is equivalent)
        if theta < 0:
            theta = -theta

        channel_buf.write(audio_data[i], span)
        l, r = channel_buf.read()

        if span >= 50:
            r_ptr = i
            l_ptr = i + int(0.0003 * ((span - 50) / 50) /
sample_T)
        else:
            l_ptr = i
            r_ptr = i + int(0.0003 * ((50 - span) / 50) /
sample_T)


        left_amp_mult = (1 - l_dist / 20)
        right_amp_mult = (1 - r_dist / 20)


        left.append(float(l * left_amp_mult))
        right.append(float(r * right_amp_mult))

        left_divisor, right_divisor = low_pass_divisor(theta)


        if i == 0:
            left_filtered.append(left[i])
            right_filtered.append(right[i])
        elif i > 0:
            left_filtered_sample = left_filtered[-1] + ((left[i]
- left_filtered[-1]) / left_divisor)
```

```
            left_filtered.append(left_filtered_sample)
            right_filtered_sample = right_filtered[-1] +
((right[i] - right_filtered[-1]) / right_divisor)
            right_filtered.append(right_filtered_sample)

    left_channel = np.array(left_filtered, dtype=np.float32)
    right_channel = np.array(right_filtered, dtype=np.float32)


    assert len(left_channel) == len(right_channel)

    tone_y_stereo=np.vstack((left_channel, right_channel))
    tone_y_stereo=tone_y_stereo.transpose()
    wavfile.write(output_path, 44100, tone_y_stereo)

#spatialize_over_time("shorter_guitar.wav",
"guitar_parametrized_function.wav", circular_orbit)
```

1.2 https://github.com/AMeriluoto/MEng-Project