Converting Images To Sound: A Lightweight Sensory Substitution Device

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

Masters of Engineering, Electrical and Computer Engineering

Submitted by

Cicci Chen, Sabian Grier, Vidhula Pallavor

M.Eng Field Advisor: Hunter Adams

Degree Date: May 2025

TABLE OF CONTENTS

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title:

Converting Images To Sound: A Lightweight Sensory Substitution Device

Authors:

Cici Chen, Sabian Grier, Vidhula Pallavor

Abstract:

Sensory Substitution Devices (SSDs) seek to map the information from one sense to another, meaningfully encoding the most relevant information in the transfer. Sight-to-sound mapping is a salient example of sensory substitution because of its promising and demonstrated potential to assist individuals with visual impairments. A number of SSDs exist that have been proven to successfully enable individuals to develop functional perceptions of their physical environment using sound. With the appropriate training, individuals have developed the ability to auditorily map their physical world with impressive fidelity, including the capacity for navigation, and even facial and object recognition. This impressive capability is indicative of a unique and impressive neurological phenomenon, suggesting that our brains possess the capacity

to rewire themselves to “see” using sound. Although demonstrated in practice, the underlying neural process related to sensory substitution remains largely a mystery. The first step in gaining an understanding of sensory mapping is developing an animal model. In this project we seek to develop a set of tools leveraging the capabilities of the RP2040 microcontroller to help facilitate sensory substitution experimentation on tree shrews with the broader goal of assisting in the development of an animal based sensory mapping model.

Executive Summary:

In our project, our efforts were focused on developing an understanding of the multiple avenues through which SSDs can be designed, implemented and utilized in the unique case of tree shrew experimentation. Considering the small form factor required by this application, this imposed significant constraints on our design. With our primary pursuit being progress toward developing a compact SSD device suitable for these animals, we needed to be intentional and creative in our approach to fit within the size and compute limitations of this use case. Considering these factors, our process began with a high-level exploration of sight-to-sound mapping abstracted from specific hardware implementations. This exploration enabled us to develop a foundational understanding of sensory substitution and the critical and noncritical elements of a functional sensory substitution system.

We began by developing an idealized method for sight-to-sound conversion. By 'idealized,' we assumed that the physical implementation of the approach would fit within the size and computational constraints of our system. Before we began development, in conjunction with our project advisor, we assessed the viability of various sight-to-sound conversion protocols, of which one stood out as a promising direction of exploration. In this method, we would take a frame from a video input, perform edge detection on the frame, and extract the contours from the image. We then reconstruct that contour's points using a Fourier series. We can extract the components from that Fourier series, map them to an audible range, and convert them to an audio output using Direct Digital Synthesis. We could perform this operation in near real time with sufficiently fast computation, creating an effective method of encoding visual data into audio. Although this method was promising and we were able to develop the functional implementation, demonstrating its potential, it became clear we needed to pivot in another direction for our design due to several factors. To minimize the amount of physical hardware being placed on the shrew we explored alternative means of outsourcing the computation to the

environment. We settled on an approach where the shrew would wear an IR transmitter and receivers would be strategically placed in the environment on important features. When these features would sense the IR signal denoting the shrew was looking in its direction, it would output a pure tone. This approach would maintain all the essential functionality the experiment requires in a simplified design that overcomes the limitations uncovered in our previous design direction.

1. Background

The inspiration for this project stems from the pursuit to develop an animal-based neural model that strives to understand and explain the process of sensory mapping in the broader context of our understanding of neuroplasticity. The project scope defined by this experiment involved the design and development of a head mounted SSD concept suitable for tree shrews that meaningfully and efficiently converts one sense, sight, to another, sound. A fundamental component of this process was developing an understanding of what constitutes a meaningful sight-to-sound mapping, or more specifically, the fundamental visual cues that make sight valuable, for example, depth perception, shape recognition, and pattern recognition. These examples only illustrate a few of the meaningful facilities provided by the sense of sight, but for the sake of discussion relating to sensory mapping, we can try to understand these examples in terms of their relative value.

In certain instances, it can be reasoned that your ability to understand an object's position relative to oneself is more important than the ability to identify what that object is. In reality, our capacities for localization, recognition, and other visual functions are synthesized to form a meaningful understanding of the physical world. This is because the many nuances of our environment often undermine rigid hierarchies that treat visual functions, like depth perception or object recognition, as independently or universally more valuable than one another. But what we seek to highlight here is that, in the context of an experiment conducted in a controlled environment, meaningful insight can be gained by developing a sensory mapping system that doesn't attempt to do it all, but instead focuses on translating a specific subset of the information encoded in vision. The ability to construct the test environment allows the experimenter to forgo the real-world nuance that typically blurs the relative value of different visual functions.

Our initial approach to this design problem attempted to maximize the visual information encoded in our audio output. Using a small camera, we intended to design a sight-to-sound SSD that would translate individual video frames to audio using some conversion methodology. The ultimate design would be a head-mounted unit that integrates a camera, microcontroller, and left and right audio channels for each ear. Before any hardware-related design choices, it was important to understand and design a method of sight-to-sound mapping. In consultation with our project advisor, we ultimately devised an approach using the Fourier series and direct digital

synthesis. We chose this method because of its relative feasibility, but more importantly, it is an intuitive mapping from sight to sound. In a few steps, it can create a direct map from a visual to an audio output in a manner whose functionality is easy to validate. This conversion is a multi-step process that begins with edge detection on an image. This can be performed with any edge detection algorithm, but we used Sobel and Canny edge detection. Once the edges have been extracted from an image, we identify relevant contours and represent them as a series of points, which we then reconstruct and approximate using a Fourier series. We can then take that Fourier Series, extract the frequency components, and map them to an audible range. Once that mapping is complete, we can finally convert those signals to an audio output using direct digital synthesis and a digital-to-analog converter (DAC). In practice, if you use an oscilloscope to probe the DDS output of the Fourier series, you would be able to observe the original contour on the oscilloscope.

It's important to consider that there are many other ways to do this. Beyond ease of validation, there is no inherent value in your audio signal being able to "draw" your image. Functionally, what your signal looks like when probed doesn't matter at all. What does matter in this case is that there exists some meaningful mapping between the visual input and the audio output. This may involve observing similar frequency components for similar shapes, or using any other mapping method capable of characterizing audio consistency or similarity that aligns with our understanding of visually similar cases. Another example is an increase in volume as something becomes larger. The critical dilemma with other approaches is, how do you verify that some meaningful mapping actually exists? This is not an effort to suggest that there is no verification strategy, but rather any significant effort to try and answer this question would be rigorous and time-intensive, falling outside the realistic scope of this project. We hope to illustrate here that although there are other methods of handling this problem, our design choice is motivated by the relative and intuitive ease with which we can analyze and debug our system.

2. Theory

2.1 Edge Detection and Contour Extraction

An edge is created by a sudden change in brightness in an image. Edge detection is the process of identifying where these sharp changes in brightness occur.[6] We tested two different edge detection algorithms- Sobel (using OpenCV[2]) and Canny (custom implementation).

The **Sobel operator** uses two 3x3 convolution masks- one to detect changes in gradient in the x direction and the other in the y direction. These gradients are then combined to highlight the edges of an image.
For example, we OpenCV uses the mask-

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

Then,

$$G = \sqrt{G_x^{\,2} + G_y^{\,2}}$$

The Sobel operator is popular because it combines smoothing with differentiation. The central row/column of each kernel effectively averages neighboring pixels, making Sobel less sensitive to noise.

Then we used OpenCV's **findContours** function to extract the boundaries of the connected edge regions and then pick the largest boundary by area.

Because OpenCV has user-accessible edge detection algorithms, it enabled us to begin experimenting with multiple algorithms while also making progress on the other elements of the approach. Using OpenCV edge detection allowed us to work on our Fourier series contour approximations and the Direct digital synthesis by forging an initial hurdle of creating a functional implementation of an edge detection algorithm. However, in the next step of hardware implementation, we would not have access to Python libraries when programming a

microcontroller. In turn, we would be responsible for performing our own edge detection. OpenCV enabled us to progress on the other design components, but we still needed to create the edge detection functionality when we moved to the hardware implementation step. With this being the case, in parallel with the other design elements, we made a functional implementation of Canny edge detection by following the Canny algorithm steps described above that didn't utilize any Python libraries to help develop an understanding of the algorithm and gain experience writing our own implementation in case it was necessary in the next steps of our project.

Canny Edge detection is designed to satisfy three main objectives: good detection of true edges, accurate localization of those edges, and minimal response, meaning each edge is marked only once. The algorithm follows-

1. Gaussian filter to reduce noise

$$G(x,y) = \frac{1}{2\pi\sigma^2} exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

2. Calculate intensity gradient in horizontal and vertical directions (using the Sobel operator as described above) and compute the gradient magnitude and direction at each pixel.

$$Magnitude\ (G) = \sqrt{G_x^{\ 2} + G_y^{\ 2}},\ Direction\ (\theta) = arctan\left(\frac{G_y}{G_x}\right)$$

3. We would want to preserve only the sharpest edges. The pixel whose gradient magnitude is a local maximum in the direction of the gradient is preserved.
4. Then we set a threshold. Any pixel below this threshold is 'weak' and is suppressed.
5. In the final step, we use edge tracking to connect weak and strong edges if adjacent. This step helps eliminate spurious responses and preserves true edges that are weak due to noise or lighting.

2.2 Fourier Series

Our idea stemmed from Prof. Van Hunter Adams' blog, where he sonifies Picasso's line drawings using Fourier Series to convert visual forms into sound.[1] Inspired by this approach, we aimed to explore how a two-dimensional image could be abstracted and translated into an auditory experience by treating image contours as signals. The Fourier Series is ideal for this task

because it allows a periodic function to be expressed as a sum of sine and cosine waves of varying frequencies and amplitudes—mirroring the way sound can be decomposed into its harmonic components.

Mathematically, the Fourier Series represents a signal f(t) as:

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{2\pi nt}{T} + b_n \sin\frac{2\pi nt}{T}\right)$$

where the coefficients a_n and b_n define the contribution of each harmonic, and are given by:

$$a_0 = \frac{1}{T}\int_0^T f(t)\,dt$$

$$a_n = \frac{2}{T}\int_0^T f(t)\cos\frac{2\pi nt}{T}\,dt$$

$$b_n = \frac{2}{T}\int_0^T f(t)\sin\frac{2\pi nt}{T}\,dt$$

However, for discrete data such as image contours, continuous integration is impractical. Prof. Adams derived a discrete approximation suitable for embedded applications, which we adopted in our work. His formulation expresses the Fourier coefficients as:

$$a_n = \frac{T}{2\pi^2 n^2}\sum_{j=1}^{K}\frac{\Delta x_j}{\Delta t_j}\left(\cos\frac{2\pi nt_j}{T} - \cos\frac{2\pi nt_{j-1}}{T}\right)$$

$$b_n = \frac{T}{2\pi^2 n^2}\sum_{j=1}^{K}\frac{\Delta x_j}{\Delta t_j}\left(\sin\frac{2\pi nt_j}{T} - \sin\frac{2\pi nt_{j-1}}{T}\right)$$

Low-frequency components represent the broad shape of the contour, while higher frequencies correspond to finer details. In audio, these translate to smoother versus more textured sounds. We limited the number of harmonics to ensure clarity and prevent aliasing, selecting only the lowest frequency components to represent the most dominant visual features.

2.3 Infrared Transmission/ Reception

Infrared (IR) radiation is a type of electromagnetic radiation with wavelengths longer than visible light, typically ranging from 700 nanometers to 1 millimeter. Because it is invisible to the

human eye but easily detectable by electronic sensors, IR technology is widely used in communication and proximity sensing.

In an IR communication system, data is transmitted wirelessly over short distances using modulated infrared light. In our setup, the IR signal is modulated at 56 kHz, a common carrier frequency that helps distinguish intentional signals from ambient IR noise. For sensing applications, an IR transmitter (typically an IR LED) emits modulated pulses of infrared light, while an IR receiver module detects the reflected or transmitted light. This forms the basis for line-of-sight proximity detection.

Our system uses a Raspberry Pi to control both the transmitter and receiver, enabling flexible control over IR signal generation and detection. While a microcontroller like the Raspberry Pi is convenient for programmable modulation and data handling, the transmission side can also be implemented with discrete hardware such as a 555 timer IC configured to generate a stable 56 kHz square wave for continuous or pulsed IR emission.

IR systems are inexpensive, energy-efficient, and reliable over short distances, making them highly suitable for embedded and real-time sensory substitution applications.

2.4 Direct Digital Synthesis

Direct Digital Synthesis (DDS) is a digital technique used for generating precise and tunable waveforms, especially for sinusoidal waves, using numerical methods rather than analog components. It is widely applied in modern communication systems, signal generators, audio synthesis, and embedded systems where compact, efficient, and flexible waveform generation is required. Based on these reasons and inspired by Prof. Van Hunter Adams' blog, we use DDS for audio synthesis in both the image processing approach and the real-time sound system. [3]

The essential components of a DDS system include a phase accumulator, a waveform lookup table, a reference clock, and a digital-to-analog converter (DAC). The DDS algorithm operates by incrementally updating the phase accumulator at a fixed clock rate. This updated phase value is used to access the lookup table, which returns a digital representation of the waveform. The digital values are then converted to analog form using a DAC, producing a continuous

waveform. The output frequency is determined by the rate at which the phase accumulator is incremented, allowing for precise and rapid control of signal frequency.

The phase accumulator can be seens as a digital register that holds a numerical value representing the current phase of the waveform. On every clock cycle, a constant known as the phase increment is added to the accumulator. Because the accumulator has a limited number of bits, it wraps around when it exceeds its maximum value, producing a repeating cycle that corresponds to one full period of the waveform. The most significant bits of the accumulator are used to index into a lookup table, which contains precomputed samples of a sine wave over one cycle. The corresponding digital amplitude is retrieved and passed to the DAC. Mathematically, the relationship between output frequency and phase increment is given by:

$$F_{out} = \frac{\Delta\phi \cdot F_s}{2^N}$$

Where:

- $\Delta\phi$ is the phase increment value,
- F_s is the clock or sampling frequency,
- N is the number of bits in the phase accumulator.

This formula shows that the output frequency is directly proportional to the phase increment and inversely proportional to the size of the accumulator.

In this project, the DDS engine is implemented using a 32-bit phase accumulator and a 256-entry sine lookup table. This configuration is both memory-efficient and capable of high resolution. An implementation example is provided for further introduction.

Given:

- Sampling frequency F_s=50,000Hz
- Desired output frequency F_{out}=1,000Hz
- Phase accumulator width $N = 32$
- Lookup table size: $2^8 = 256$ entries

The phase increment $\Delta\phi$ is calculated as:

$$\Delta\phi = \frac{F_{out} \cdot 2^{32}}{F_s} = \frac{1000 \times 2^{32}}{50000} \approx 97,612,884$$

This value is added to the phase accumulator on each update. The accumulator itself is a 32-bit unsigned integer that wraps around upon overflow, simulating continuous phase rotation. To retrieve the current amplitude, the upper 8 bits of the accumulator are used as an index:

$$Index = phase\_accum \gg 24$$

This index accesses a sine lookup table containing 256 precomputed values scaled to range from −2047 to +2047. To match the 12-bit DAC range (0 to 4095), the value is offset by 2048:

$$DAC\ output = sine\_table[Index] + 2048$$

The timer or interrupt routine updates this output at a fixed interval, determined by F_s, resulting in a stable and smooth analog waveform.

3. Methodology

Building upon the mathematical framework outlined in Section 2, we implemented a computer vision and signal synthesis system in Python for Raspberry Pi Pico's RP2040 microcontroller. The methodology involved extracting contours from images, transforming them into the frequency domain using Fourier analysis, and synthesizing corresponding audio signals via Direct Digital Synthesis (DDS).

3.1 Computer Vision Approach

The system was implemented in Python using the following libraries:

- OpenCV (cv2) for image processing and contour extraction
- NumPy for numerical computations
- Matplotlib for visualization
- SciPy for audio file operations
- PyDub for audio manipulation

3.1.1 Image Processing and Contour Extraction

We start by converting an input image to grayscale and applying Gaussian blur to reduce noise. We first implemented Sobel edge detection, which computes image gradients in both horizontal and vertical directions to identify regions of rapid intensity change. From the edge-detected image, we extract contours using OpenCV's findContours function. The largest contour, assumed to represent the object of interest, is isolated for further analysis.

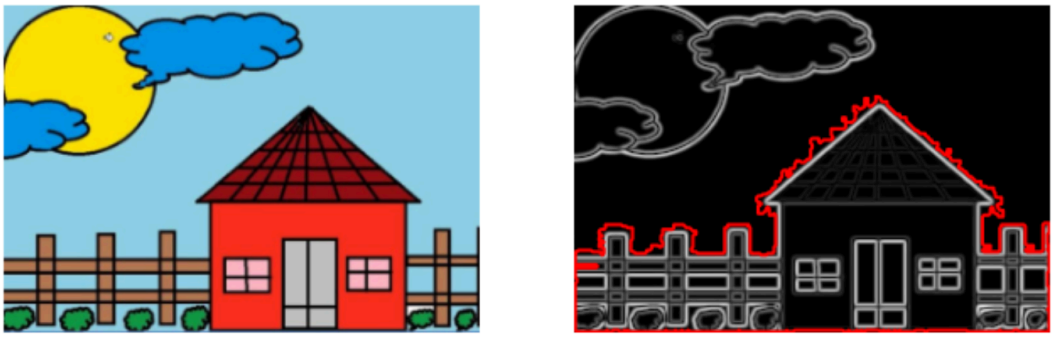


Figure 1. Result from the contour extraction (in red).

To ensure compatibility with embedded deployment and to gain full control over the processing pipeline, we implemented a custom version of the Canny edge detection algorithm instead of using high-level library functions. This allowed fine-grained tuning of each stage and better optimization for hardware implementation. The pipeline begins with Gaussian smoothing to reduce image noise, using a manually generated kernel. This is followed by gradient computation using Sobel filters to estimate edge strength and orientation across the image. Next, non-maximum suppression is applied to thin the edges by retaining only the local maxima along the direction of the gradient. The resulting image then undergoes double thresholding, which classifies pixels into strong, weak, and non-edges based on predefined intensity thresholds. Then, edge tracking by hysteresis is performed, promoting weak edges that are connected to strong ones while discarding isolated weak responses. This modular approach mirrors the original Canny method while offering the flexibility and control required for low-level hardware implementation.

3.1.2 Fourier Transform of Contour

We implemented a custom fourier_coefficients function for computing the Fourier series coefficients of a two-dimensional closed contour represented by its x and y coordinates. The

function first calculates the local differences (dx, dy) and arc lengths (dl) between successive points along the contour, allowing it to establish a parameterization based on cumulative arc length. It then applies a piecewise-linear approximation to integrate sine and cosine components over the contour, producing sets of Fourier coefficients (an, bn for the x-component, and cn, dn for the y-component). These coefficients compactly represent the spatial structure of the contour in the frequency domain. The total arc length (L) of the contour is also returned, enabling the reconstruction or synthesis of the shape using harmonic summation.

The Fourier representation effectively compresses the contour information into frequency components, where lower frequencies capture the general shape and higher frequencies represent finer details. This approach enables us to:

- Represent the shape with a controlled level of detail by adjusting the number of coefficients
- Create a frequency-based representation that naturally maps to audio signals
- Achieve dimensionality reduction while preserving essential shape characteristics

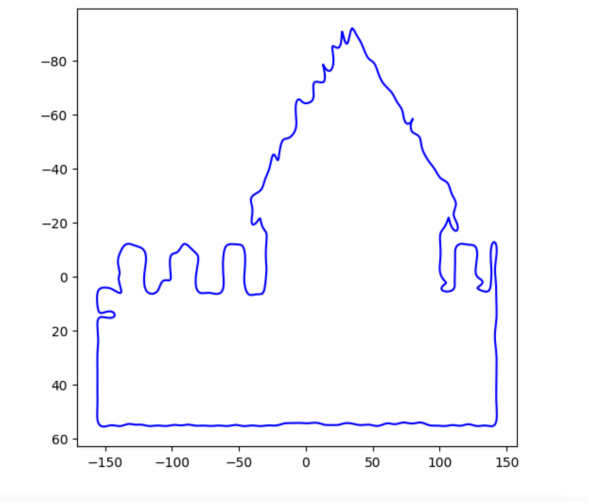


Figure 2. Plot of the contour converted reconstructed through (Inverse) Fourier Series.

3.1.3 Signal Generation with Direct Digital Synthesis (DDS)

To transform the Fourier representation into audible signals, we implemented a Direct Digital Synthesis (DDS) approach, which provides precise frequency control.

Our system uses a phase accumulator with a 32-bit resolution to generate accurate sine waves:

```
sampling_freq = 44000
phase_increment = 262 * (2**32) / sampling_freq * np.arange(1, 101, 1)
```

This formula establishes frequency increments scaled by harmonic number, starting from a fundamental frequency of 262 Hz (middle C). Then the phase accumulator is updated for each sample.

```
phase_sin += phase_increment
phase_sin = phase_sin % (2**32 - 1)
```

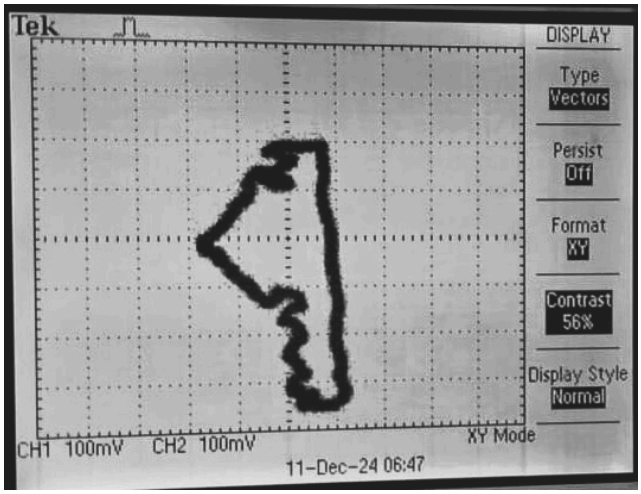
We constructed an 8-bit sine Look Up Table (LUT with 256 entries) to return/store the magnitude of the sine and cosine wave.

The final audio signals are generated by combining the sinusoidal components weighted by their respective Fourier coefficients.

This approach effectively reconstructs the contour in the time domain as two separate audio signals representing the x and y coordinates. By mapping the x and y contour coefficients to the left and right audio channels respectively, the stereo sound field effectively encodes spatial information of the original shape.

3.1.4 Audio Output

The generated waveforms are scaled to fit the 16-bit audio format and saved as .wav files. The left and right channels respectively represent the x and y components of the contour. The resulting stereo audio file encodes the shape information in a way that could potentially be interpreted through listening, establishing a mapping between visual and auditory domains.

(a) 

(b)Play to hear the sound file

Figure 3. (a) Reconstructed image using audio signals on an oscilloscope (b) Audio generated.

Pivot:

In our efforts, we were able to design a functional proof of concept that demonstrated the feasibility of this approach. However, when it came time for hardware implementation, we decided to pivot in another direction. Although our prior development yielded promising results, we were likely to encounter several challenges when implementing hardware. Edge detection and Fourier coefficient calculation are mathematically intensive processes that may not meet the timing demands of near real-time video-to-audio conversion, especially as we increase the number of coefficients to improve our contour approximation. The broader concern of this approach was related to the size constraint of our system. Testing on tree shrews has regulations that dictate the maximum weight they can wear in peripheral devices relative to their body weight. With this specific limitation, integrating the necessary hardware to facilitate this approach would become very challenging. It's also important to discuss the additional design challenges we would need to overcome for this approach to be ready for deployment. When constructing our Fourier series, it only approximates the largest contour in the image. This prompts the question: Does the largest contour in an image appropriately capture all the relevant information in the visual? The answer depends on a number of factors. What parameters are being passed into your edge detection algorithm? What is the visual? What information are you trying to capture? The problem is that the answer to whether it captures all relevant information is circumstantial. It depends on both environmental factors, design choices, and parameter selection. The broader implication is that a configuration that performs meaningful mapping

across all cases likely doesn't exist. So, how do we design a system that can evolve dynamically to yield consistent performance across varying conditions? For this to work, we would need to devise a method allowing the system to quantify its performance and make informed adjustments. Considering validation challenges previously discussed, it's evident that this becomes an exceedingly complex problem. An alternative approach we explored is considering more than just the largest contour. This introduces its own set of difficulties. Do we have enough Fourier coefficients to construct the contours? How do we choose the order to pass through multiple contours to construct the image? Again, this does not suggest that there is no legitimate method. Still, it circles back to the same issue discussed previously: if your Fourier transform doesn't look like the image, how do you quantify whether it's actually a meaningful mapping?

The key takeaway here is that beginning to answer these questions would also result in a design effort that reaches beyond the realistic scope of the project. The primary appeal of this initial approach was its versatility in use cases, with the intention of it being able to perform sensory substitution in non-controlled environments. Considering the circumstantial nature of this method's performance, bridging the gap from testing to controlled to non-controlled environments is a significant undertaking that is not feasible within our timeline. This new understanding of the system requirements motivated a pivot in design direction. For controlled testing, the experiment necessitates a relationship between the tree shrews movement and auditory feedback in relation to their environment. However, if we have environmental control, we can also address our design's size and weight constraints by shifting the computation to the environment. In our revised approach, the treeshrew only wears an IR transmitter. We then construct an environment where the salient features contain strategically placed receivers that generate a pure tone when the signal is detected. This new mapping from sight, in the form of the visual field of view modeled by an IR transmitter, to sound, audio output from features being observed creates a simplified and intuitive audio-specialized environment. In conclusion, contingent on the ability to design and control our environment, this new solution offers the best case performance of the previous approach with a greatly simplified design.

3.2 Real-time sound system approach

3.2.1 Software Approach

INTERRUPT SERVICE ROUTINE

An interrupt service routine (ISR) is a function that executes automatically in response to a hardware or software interrupt. In embedded systems, ISRs are commonly used to handle time-critical tasks, such as responding to sensor input or updating peripheral devices. When an interrupt occurs, the processor temporarily pauses its current task and executes the corresponding ISR. After the ISR completes, normal execution resumes. ISRs are generally kept short and efficient to minimize disruption to the main program. In this project, timers trigger ISRs at fixed intervals to manage both IR signal detection and audio duration control.

SPI PROTOCOL

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used to exchange data between microcontrollers and peripheral devices. SPI typically involves four lines: SCK (clock), MOSI (master out, slave in), MISO (master in, slave out), and CS (chip select). Data is transmitted in full-duplex mode, allowing simultaneous sending and receiving. In this system, SPI is used to control the MCP4822 digital-to-analog converter. The microcontroller acts as the SPI master, transmitting 16-bit digital values to the DAC, which updates its analog output accordingly. This interface allows precise and high-speed control over waveform generation.

IR SIGNAL TRANSMISSION

The IR transmitter operates by configuring a GPIO pin to output a 56 kHz carrier wave using the Pulse Width Modulation (PWM) feature of the Raspberry Pi Pico. This frequency is typical for IR communication and is compatible with our IR receiver. A function named *send_pattern( )* is responsible for emitting bursts of IR signals. Each burst consists of 40 cycles of the 56 kHz carrier, which is approximately 714ms in duration. After each burst, the transmitter pauses for 500ms before sending the next burst. This creates an alternating on-off pulse pattern that is recognizable by IR receiver modules.

The burst generation is handled by the *send_burst()* function. This function calculates the duration of the burst by converting the number of carrier cycles into microseconds, sets the PWM duty cycle to 50 percent to activate the IR transmitter, and uses *time.sleep_us* to maintain the signal for the required duration. Once the burst is complete, the duty cycle is set to zero to stop the signal. The main execution loop continuously calls *send_pattern()* to trigger the IR transmitter.

IR SIGNAL RECEPTION AND AUDIO PLAYBACK CONTROL

The IR receiver is implemented using a second Raspberry Pi Pico. It monitors incoming IR signals using two GPIO input pins, each connected to a discrete IR sensor. These sensors produce a digital low signal when infrared radiation is detected. To manage real-time responsiveness, the receiver utilizes two hardware timers configured in periodic mode, each associated with a specific ISR. This design allows for efficient non-blocking processing of signal detection and audio playback duration.

The first timer is configured to trigger every one millisecond and is linked to the *check_ir()* function. This ISR continuously monitors the IR input pins for transitions from high to low, which indicate the arrival of a modulated IR burst. When such a transition is detected and the system is idle, the ISR initiates a state change to the active mode and begins audio playback through the DDS engine. To avoid multiple triggers from a continuous signal, the routine compares the current state of the IR input with the previously recorded state, ensuring that playback is only initiated on a falling edge.

The second timer is also configured to fire every one millisecond and is associated with the *alarm()* function. This ISR manages the duration of the audio output. While the system remains in the active state, a counter is incremented during each timer interval. If the IR signal is no longer being detected and the counter exceeds a predefined threshold, which is typically set to 1000 milliseconds, the audio output is stopped and the system resets to the idle state.

The coordinated use of these two timers enables clear separation of concerns within the system: one ISR handles real-time signal detection, while the other manages audio duration. This approach ensures deterministic and responsive system behavior. Since the ISRs execute

independently at consistent intervals, the system maintains precise timing without interfering with the main execution thread or other operations. Together, these mechanisms form the foundation of the receiver's real-time signal processing and playback control.

Once audio output is started, the alarm timer begins incrementing a counter at a one-millisecond rate. If the IR signal is no longer present and the counter reaches the defined duration threshold, which is typically set to 1000 milliseconds, the DDS output is stopped and the state machine returns to the idle state. This mechanism ensures that tones are played for a consistent duration even in the presence of transient or noisy IR inputs. It also prevents tones from continuing indefinitely in the event of a detection failure.

AUDIO GENERATION

Different IR receivers are assigned to trigger distinct sound pitches upon detecting IR signals. This design enables the tree shrew to perceive different sound frequencies depending on its orientation. Incorporating such natural auditory cues into the generated sounds is essential for helping the tree shrew associate spatial information with auditory input.

3.2.2 Hardware Implementation

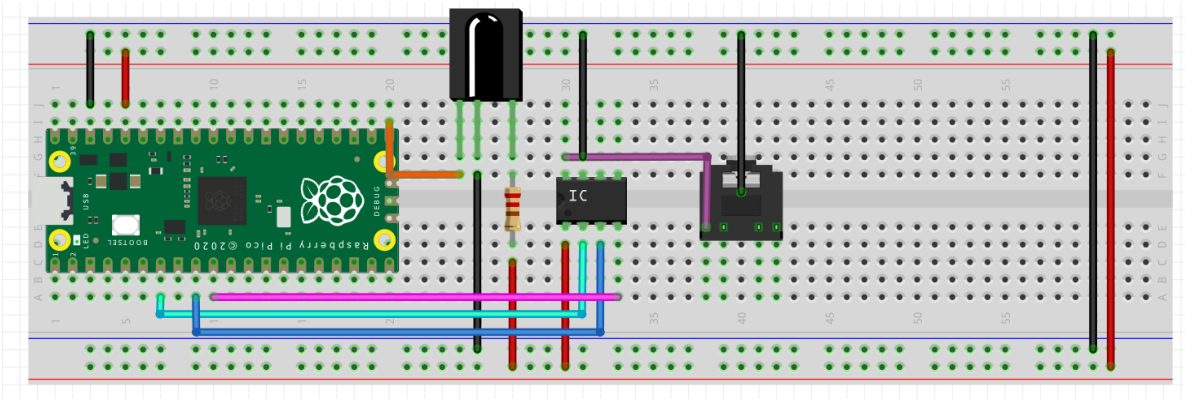


Figure 4. Circuit Diagram of IR receiver and sound playback part.

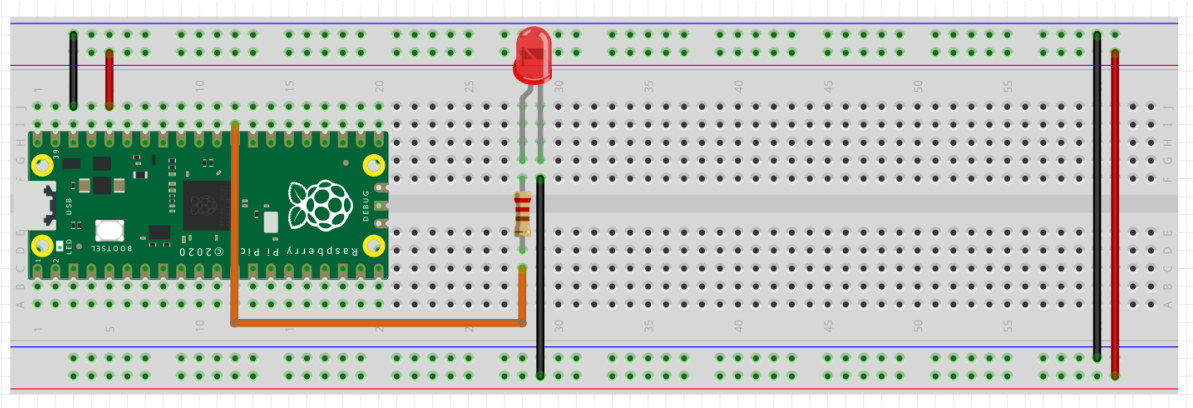


Figure 5. Circuit diagram of IR transmitter.

The real-time sound system is implemented using the Raspberry Pi Pico's RP2040 microcontroller, in conjunction with an LTE-4208 infrared (IR) transmitter, a TSOP34156 infrared receiver, and an MCP4822 digital-to-analog converter (DAC).[4][5] These components operate together to support the real-time detection of directional infrared signals and the generation of corresponding audio feedback. Figure 1 shows the circuit diagram of the IR receiver with its sound playback system, and Figure 2 shows the circuit diagram of the IR transmitter.

The IR transmitter emits infrared signals modulated at a carrier frequency of 56 kHz.[4] This modulation is generated by the PWM hardware of the RP2040 microcontroller, which ensures accurate and stable frequency output. The transmitter operates continuously, producing bursts of

pulses that can be detected by an appropriately tuned IR receiver. For reliable detection, the receiver module requires a minimum of six consecutive pulses within the burst based on its dataset.

The DAC is interfaced with the RP2040 over the SPI bus. The DAC receives digital waveform samples from the microcontroller and converts them into analog voltage signals that drive connected speakers. DAC features two independent output channels, allowing the system to generate distinct audio tones for multiple spatial directions.

In our preliminary implementation of DDS, we explored using micropython due to its relative ease of use and pico support. We developed a module for simple pure tone generation using DDS on a 2-channel MCP4822 DAC. We were cognizant of the significant slowdown incurred by using micropython over C, but we reasoned that micropython speed was sufficient to support audio frequency generation through DDS. Inside this module, we create a class specific to DAC control. The MCP4822 uses SPI communication with our MCU to manipulate the analog voltage output on one of its two channels, A and B. This class abstracts away the SPI communication from the user by creating accessible methods that handle the formatting of the 16-bit words responsible for output voltage manipulation and channel selection. In addition to our DAC class, there is also a DDS class, which uses the methods of the DAC class to generate sine waves of user-programmable frequencies on the DAC output channel of their choice.

A lookup table is precomputed to forgo the need for sine value computation during wave generation. A 32-bit phase accumulator controls the amount by which the lookup table is incremented in each timestep. A hardware timer interrupt is configured to trigger DAC updates at a fixed sample rate. The sine wave frequency is defined by the phase accumulator-informed step size, which influences how fast you move through the sin table. Larger step size results in higher frequency. Additionally, there are *start()* and *stop()* methods. The *start()* method takes in a frequency, calculates the phase increment needed to achieve the desired frequency, and creates the timer interrupt responsible for DAC updates.

The *stop()* method halts the timer, which stops the output. Lastly, there is an interrupt service routine (ISR). The interrupt calls the ISR, which uses the phase to get a sine value from the lookup table and outputs it to the appropriate DAC channel. Minimizing the logic included in an

ISR is important due to the timing constraints. This implementation yielded fairly good results in the case of a single output. However, when generating multiple outputs for different frequencies on each channel. It created significant audio distortion. This is likely due to the two timer interrupts being run at the same frequency. This was likely creating collisions, resulting in the undesirable behavior. In practice, the implementation should have been designed in such a way that it would use the same interrupt to update both channel outputs with different phases. Still, rather than going through the exercise of trying to resolve this, we decided to switch to a C-based implementation. We achieved this functionality in C by modifying an example provided by Professor Adams and creating an ISR capable of simultaneous updates of both channels when enabled simultaneously. In our system, the IR receiver informs whether a channel is on or off, meaning if it detects a signal, it will enable the channel, causing a pure tone to be generated at the specified frequency. In this case, the speaker, driven by the DAC, is physically collocated with the receiver. Generally speaking, it's likely not necessary for a pico to be able to drive more than one channel, so it's fair to assume that our micropython implementation would have been sufficient for this use case. However, in the pivot to C, we can run our ISR at a higher frequency and generate smoother sine waves. Furthermore, it offers more flexibility if it's preferred that a single pico drive multiple audio sources.

The RP2040 microcontroller plays a central role in coordinating the entire system. It generates the 56 kHz PWM signal needed to drive the IR transmitter and controls the burst pattern to modulate the on-off frequency of the infrared signals. It continuously monitors the IR receiver's output through a time-triggered interrupt service routine and communicates with the DAC via SPI channels to transmit audio samples. In addition, the RP2040 uses two independent hardware timers, Alarm 0 and Alarm 1, to generate precise periodic interrupts. These timers support both infrared signal detection and audio waveform synthesis, ensuring consistent timing and synchronized operation across all system components.

This on-board system allows us to demonstrate the core functionality by mounting the IR transmitter on the head of a tree shrew whose eyes are covered. As the animal turns to face different directions, the transmitter sends out infrared signals. When these signals are received by an IR receiver located in that direction, a corresponding audio signal with a distinct pitch is generated.

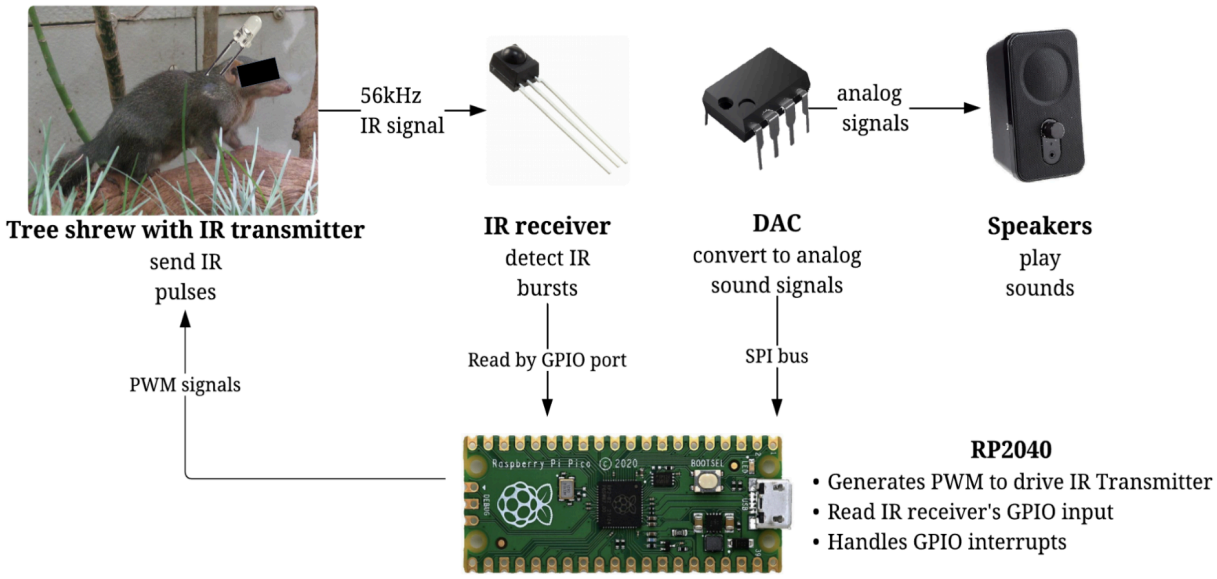
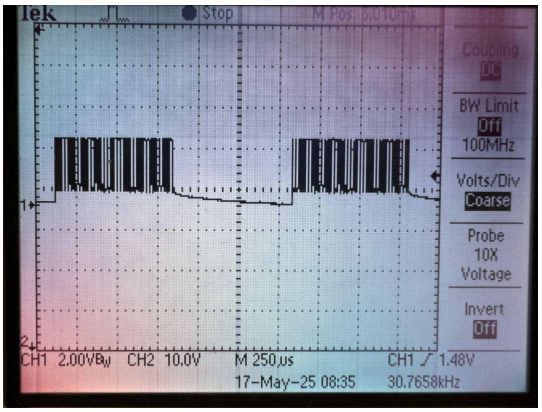


Figure 6. Block diagram showing IR-triggered sound system for tree shrews using RP2040, DAC, and speakers.

Figure 6 shows the block diagram describing the whole system setup. As the tree shrew approaches the IR receiver and its associated speakers, the audio output increases in volume. In this way, the animal can learn to associate specific sound pitches and intensities with both direction and distance. This setup is designed to help the tree shrew interpret spatial information through auditory cues. The described scenario represents the ideal experimental configuration we intend to explore in future behavioral studies.

4. Result

We verified the correct operation of the IR transmitter and receiver by comparing the signals observed on the oscilloscope at the GPIO input of the IR transmitter with the simulated modulated carrier signal. The carrier frequency measured by the oscilloscope, as shown in Figure 7(b), was 56 kHz, matching the expected value. The modulated carrier waveform, captured at the output and shown in Figure 7(a), closely aligns with the simulated signal presented in Figure 8. The strong agreement between the measured and simulated signals confirms that the IR transmitter and receiver are functioning as intended.

(a) 

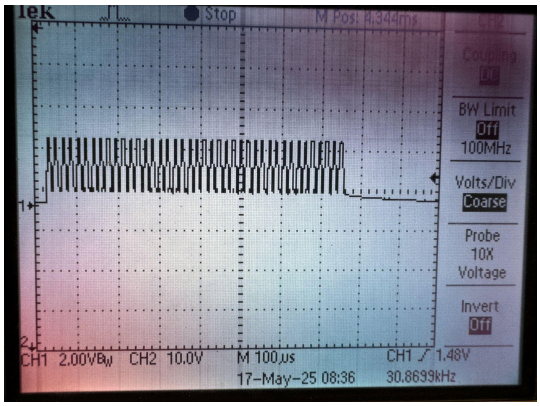
(b) 

Figure 7. (a) Modulated carrier frequency (b) Carrier signal.

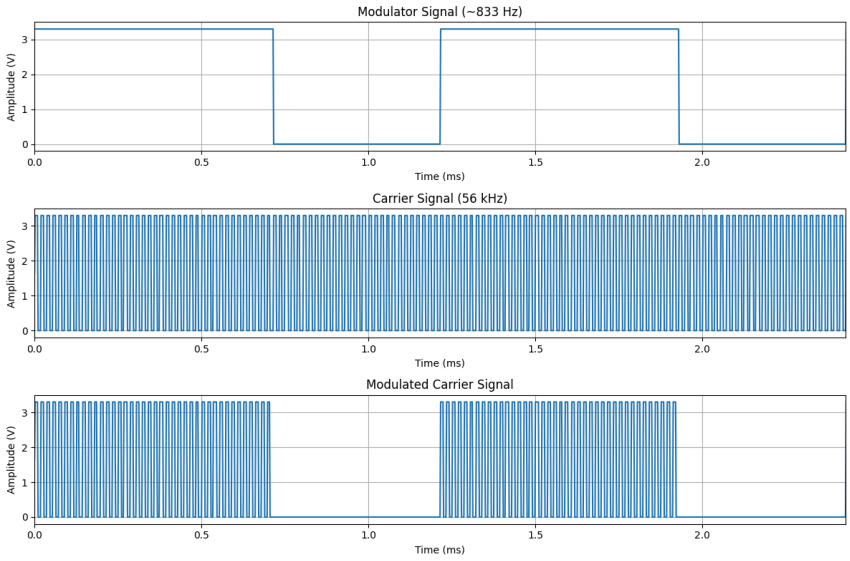


Figure 8. Simulated modulator signal, carrier signal and the modulated carrier signal.

When comparing the sound signals generated by the C and MicroPython implementations of our DDS system using an oscilloscope, we observed that the C version produced significantly smoother sine waves. This difference in signal quality was a primary reason for transitioning our implementation from MicroPython to C.

One key factor contributing to this improvement is the execution speed of the Raspberry Pi Pico when running C code. The C environment allows for much higher sampling rates, which are critical for achieving high-fidelity audio. In our tests, the C implementation reliably reached a sampling frequency of 50,000 Hz, while MicroPython struggled to exceed 4,000 Hz without encountering timing violations or jitter. The limited performance in MicroPython results from the overhead of the interpreter and less efficient interrupt handling, which reduces the system's ability to maintain precise update intervals required by DDS.

Figure 9(a) and 9(b) illustrate the effect of sampling frequency on signal quality. At higher sampling rates, the discrete steps in the digitally synthesized waveform become smaller and more closely spaced, approximating a continuous sine wave. This leads to a smoother analog output when passed through the DAC. In contrast, lower sampling rates produce more noticeable quantization artifacts and distortions, resulting in a rougher waveform. These observations reinforce the importance of timing precision and execution efficiency in DDS-based sound generation.

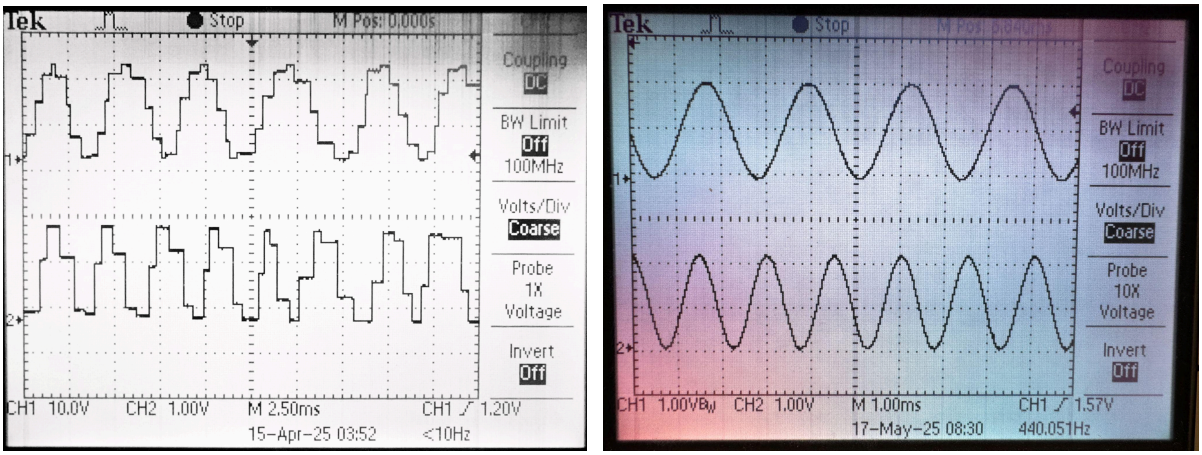


Figure 9. Sound signals when both IR detectors detect IR signals simultaneously in the (a) MicroPython version (b) C version.

5. Conclusion

This project culminated in the successful development of a compact, animal-compatible sensory substitution system that translates visual orientation into spatialized audio feedback. Our initial approach focused on real-time image processing and Fourier-based audio encoding, demonstrating a functional pipeline from contour extraction to stereo audio synthesis. However, we ultimately transitioned to a more pragmatic IR-based system due to hardware limitations and the weight constraints associated with testing on tree shrews.

Our final design leverages infrared transmission and strategically placed receivers to produce location-dependent audio tones, creating a simplified yet effective mapping from the animal's visual attention to auditory feedback. This approach maintains the core goal of sensory substitution while optimizing for experimental feasibility, enabling controlled behavioral studies without overburdening the subject.

Beyond its technical implementation, this work contributes to the broader effort of understanding neural plasticity and sensory remapping by providing a scalable platform for animal-based sensory substitution research. Future iterations could explore dynamic environments, more complex auditory encodings, or even reintegration of visual processing using offloaded computation. Ultimately, our system represents a meaningful step toward developing intuitive, lightweight SSDs that support fundamental neuroscience investigations.

6. Citations and references

[1] H. V. Adams, “Picasso, by way of Fourier,” *vanhunteradams.com*, https://vanhunteradams.com/Math/Fourier_drawing/Fourier_Drawing.html (accessed Sep. 15, 2024).
[2] OpenCV.org, “OpenCV: Open Source Computer Vision Library,” https://opencv.org/ (accessed Sep 15, 2024).
[3] H. V. Adams, “Direct Digital Synthesis (DDS),” *vanhunteradams.com*, https://vanhunteradams.com/DDS/DDS.html (accessed Nov 16, 2024).
[4] Lite-On Technology Corporation, *LTE-4208 Infrared Emitting Diode Datasheet*, Rev. C, May 21, 2004. [Online]. Available:

https://optoelectronics.liteon.com/upload/download/DS-50-92-0015/LTE-4208.pdf (accessed Feb 9, 2025).
[5] Vishay Semiconductors, *TSOP321, TSOP323, TSOP325, TSOP341, TSOP343, TSOP345 IR Receiver Modules for Remote Control Systems*, Rev. 2.3, May 17, 2024. [Online].Available: https://www.vishay.com/docs/82490/tsop321.pdf (accessed Feb 9, 2025).
[6] GeeksForGeeks, "Edge detection using Prewitt, Scharr and Sobel Operator", *geeksforgeeks.org*. Available: https://www.geeksforgeeks.org/edge-detection-using-prewitt-scharr-and-sobel-operator/, (accessed Nov 24, 2024)