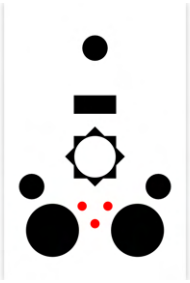
raivBox: Neural Waveform Synthesis on a Low-Resource Embedded Device

Jack Tipper



Submitted in partial fulfillment of the requirements for the
Master of Music in Music Technology
in the Department of Music and Performing Arts Professions
Steinhardt School
New York University

Advisor: Dr. Tae Hong Park
Reader: Dr. Alexander Sigman

May 8th, 2022

ABSTRACT

Recent advances in deep learning-based audio creation are fueling the rise of a new approach to sound design: neural synthesis. Until now, robust neural synthesizers for musical sounds have been relegated to traditional desktop and cloud-based computing environments. Hosting these generative systems on low-cost devices will broaden offline and untethered access to this fledgling technology, helping to make way for a new era of modular music and audio experimentation. In this thesis, the implementation of neural waveform synthesis on a resource-limited embedded development platform is explored using a modified version of Google Magenta's Differentiable Digital Signal Processing (DDSP) timbre transfer pipeline on an Nvidia Jetson Nano 2GB embedded prototyping board. Timbre transfer is a type of neural synthesis achieved by extracting audio features from an input signal and utilizing those data to generate a new waveform in the target timbre. This project introduces the raivBox: a functional neural synthesizer in the form of an affordable, unified hardware product. It features a physical control interface that provides responsive audio input and output, a selection of five neural timbre models to choose from, and a straightforward user experience that facilitates ease of use. The reworked neural synthesis pipeline produces sounds significantly faster than its predecessor while maintaining similarly compelling outputs. Qualitative research data indicate that users find the raivBox to be both intuitive and fun to interact with, and also suggest that it is perceived to be a valuable new tool for music creation and sound design. As this project is designed to be readily replicable by end users, the raivBox source code, build instructions, and a variety of audio examples are publicly available via the project's GitHub repository.

ACKNOWLEDGEMENTS

This thesis is dedicated to my two wonderful grandmothers. May you rest in peace.

Thank you to my parents, for your endless love and support, and my two siblings, for your enduring camaraderie. I love you.

Special thanks to my advisor Tae Hong Park, for your inspiration, guidance and perspective along the way, as well as my dear friends James Thornton, Brian MacFadyen, Sean Goldie, Jeff Holland, Arthur Jinyue Guo, Gregor McWilliam, and Travis Shetter, for your expertise and encouragement throughout this journey.

Thank you to my long-time mentor Peter Hamlin for always believing in me and my dreams.

Additional thanks to professors Steven Litt, Brian McFee, Agnieszka Roginska, Schuyler Quackenbush, Ernesto Valenzuela, Dirk Vander Wilt, Jeffrey Buettner, Dafna Naphtali, Morton Subotnick, Robert Rowe, Spencer Shafter, Paul Geluso, and Alan Silverman.

Thank you to the participants in my qualitative research for contributing time and crucial feedback to this work.

This work would not have been possible without the foundational neural synthesis research from Google Magenta, among many others. I am standing on the shoulders of giants.

A big thank you to my extended family and friends for being there for me, wherever we may be. I am also grateful to my business partners and clients for trusting in my craft and supporting my professional growth.

Lastly, thanks to my peers and cohort for sharing in this whirlwind of an experience.
I look forward to our future successes and collaborations!

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION AND MOTIVATION

Modular embedded computing in the field of music is the natural successor to Eurorack: a flexible hardware archetype born from the mind of Dieter Doepfer in 1995 as a way to enable creatives to assemble their own economical audio equipment and synthesizers (Rumsey, 2020). Alongside the persistent progress of computer processor technology in terms of both power and affordability, low-cost embedded prototyping platforms such as Raspberry Pi[1] and Arduino[2] have fostered accessible, community-driven computer music experimentation in realms that were previously out of reach for many (Franco & Wanderley, 2015; Surges, 2012).

As we launch forward into the artificial intelligence (AI) era, new opportunities for musical sound design and audio synthesis are emerging for investigation (Zhao et al., 2020). Deep learning takes advantage of modern computing capabilities, allowing us to leverage data like never before—to identify patterns and tackle problems that once seemed impossible to address. In the field of music, deep neural models are now being used to directly generate audio as waveforms in the time domain. This nascent sound design technique is currently being harnessed for artificial speech synthesis throughout the consumer technology industry; however, its application to musical creation has yet to reach the mainstream. Neural synthesis techniques have grown increasingly efficient in recent years, and the proliferation of high-quality open-source code libraries are broadening access to the underlying technology. Just as the Eurorack framework strove to democratize the world of modular music hardware, advances in embedded computing and prototyping environments suggest the imminent arrival of robust neural synthesis on affordable, resource-capped devices. Novel combinations of these groundbreaking computer music techniques could shape a new age of popular creative tools, as embedded implementations drive down development costs, and gracious do-it-yourself (DIY) communities nourish artistically-minded technological exploration.

This thesis is motivated by the hypothesis that neural audio synthesis is ready for widespread adoption in music production and performance contexts, and can be realized on an independent system for no more than a couple hundred dollars in overhead. The primary purpose of this project is to prove this premise by achieving competent and compelling neural synthesis

[1] https://www.raspberrypi.com
[2] https://www.arduino.cc

on a low-resource embedded device, thereby substantiating that this technology can be an accessible tool for musical creation.

A new generation of audio software tools will likely emerge from the neural synthesis paradigm. That said, current research focuses mainly on bringing this technology to artists and producers via conventional digital audio workstations. AI-powered audio editing software already exist, such as iZotope RX[3], but standalone generative hardware is missing from the picture. This project provides a framework for separating neural synthesis from the traditional confines of the desktop/laptop computing environment, allowing for experimentation in creative settings that might otherwise be overlooked or perceived as impractical areas for deep learning, such as on a pedalboard or in complex modular synthesis daisy chains. For instance, a musician might train a custom timbre model on their own singing voice and feed monophonic keyboard audio as the control signal for the neural synthesizer, forming a one-person "duet". By establishing this precedent and documenting the DIY process, this thesis broadens access to neural synthesis technology and opens new avenues for creativity in the music community. These objectives are attained through the design and development of the raivBox prototype: an affordable, user-friendly device hosting an optimized yet robust neural synthesis pipeline.

[3] https://www.izotope.com/en/products/rx

2. LITERATURE REVIEW

Crafting and understanding the neural synthesis device produced herein requires domain knowledge of embedded development platforms, system-level Linux[4], hardware-software interaction, deep learning-based waveform generation techniques, and low-latency/real-time digital signal processing (DSP). This section introduces these foundational pillars, with additional attention dedicated to relevant research developments where applicable. Throughout this stage of laying groundwork, recent findings from the intersections of these fields are presented to offer context for this project. The review concludes with a brief focus on recent work that sets precedent for this research and supports the feasibility of the key objectives.

2.1 Computer Music on Embedded Development Platforms

When an information processor is built into an enclosing product for the purpose of handling a certain task, the unified outcome is called an *embedded system* (Marwedel, 2011). These systems surround us, as they are found in all manner of electronic equipment, from vehicles to appliances, communications infrastructure, manufacturing tools and more (Cardoso et al., 2017). Embedded devices have played a crucial role in the rise of our current information era, stemming from the complex integrated circuit, microcontroller, and microprocessor technology that made direct interplay between hardware and software components feasible during the second half of the 20th century (Betker et al., 1997; Kilby, 2000; Teich, 2012; Wolf, 1994). Unlike most general-purpose desktop and laptop computers, embedded computing systems tend to be specifically tailored to perform a narrow set of tasks as efficiently as possible (Rumsey, 2020). These systems consist of a combination of integrated modules, typically including input/output (I/O) connections, memory, storage, and at least one microcontroller or microprocessor at the core of the operation.

Embedded development platforms are physical devices that facilitate hardware prototyping. Popular platforms such as Arduino and Raspberry Pi provide an accessible entry point to begin exploring the world of embedded computing (Zyskowski & de Oliveira, 2017). Arduinos are built around a microcontroller chip, offering developers access to low-level

[4] https://www.linuxfoundation.org

programming control without an extraneous desktop operating system (OS) (Richards, 2013). Michon et al. (2019) expressly identify the "Arduino revolution" as a major contributor to the DIY electronic music community, highlighting how the broad availability of these inexpensive boards and their low barrier-to-entry promoted the adoption of microcontroller technology for musical instrument and interface development in recent decades. Richardson and Wallace (2016) explain that in contrast to Arduino, the Raspberry Pi prototyping platform revolves around a comparatively powerful microprocessor instead of a microcontroller. The processing core on a Raspberry Pi runs Linux, an open-source OS that allows for potential development advantages such as network access and complex cross-language software interactions. Even though an Arduino and a Raspberry Pi share many hardware components, they are fundamentally different development environments, Raspberry Pi arguably sharing more features with a desktop computer than with an Arduino. The Arduino platform shines when applied to straightforward tasks that blur the distinction between software and hardware, such as irrigating succulents at a given time interval, or generating simple sound waves and effects. Raspberry Pi, on the other hand, is apt for projects that require substantial software-based system design and processing, like watering the plants based on calculations made from a variety of weather data streams, or implementing complex audio data processing pipelines (Richards, 2013).

Despite the widespread use of Arduino and Raspberry Pi for computer music experimentation, lesser-known embedded prototyping platforms such as BeagleBoard, Teensy, and Nvidia's Jetson series all offer their own distinct benefits for certain applications (Franco & Wanderley, 2015; Michon et al., 2019). For instance, Nvidia's Jetson Nano 2GB[5] is particularly qualified to fit the needs of the AI developer on a budget, packing an impressive 128-core graphics processing unit (GPU) into a kit that retails for less than sixty dollars (Cass, 2020; Salih & Basman Gh., 2020). While microprocessors are well-suited for running a single stream of complex calculations, they waver when assigned large-scale parallel processing tasks such as graphical rendering and machine learning (Daghero et al., 2021). This is where GPUs excel, as they are optimized to handle numerous streams of simple operations with their sprawling multi-core architectures (LeCun et al., 2015; see also: Lee et al., 2017). Mittal (2019) asserts that the Nvidia Jetson platform should be a top contender for embedded deep learning projects due to

[5] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano

its access to Nvidia's CUDA[6] programming model, which allows unrivaled developer control over the on-device GPU and other hardware-acceleration tools (see also: Kirk, 2007).

2.2 System-Level Linux Development

Linux offers a native system for managing background processes called *systemd*[7] (from *system daemon*). It is mainly used to run tasks at startup, and has a wide range of important functions in the core Linux OS. David Both (2020) describes systemd as "the mother of all processes", referring to its responsibility of initializing and managing nearly all of the essential aspects of the active host machine. systemd—written in lowercase, per the official documentation—conveniently operates using simple ASCII text files with basic sudo access restrictions, making it a relatively straightforward system-level configuration tool. For example, systemd can be used to run a Python[8] file that plays audio during the startup sequence. Linux-based projects and products that execute custom code functionality at system boot use systemd to coordinate and call these processes (Both, 2020).

2.3 Hardware-Software Interaction via GPIO

Many embedded development platforms feature general-purpose input/output (GPIO) pins to help users facilitate custom interaction between components and devices. The 40-pin GPIO pinout arrangement on the current flagship Raspberry Pi boards has become an industry standard, and is also found on the Nvidia Jetson Nano line of prototyping systems. These pins serve a variety of purposes, the most fundamental being the provision of power, electrical ground connection, and of course I/O control. Through the GPIO pins, a wide variety of programming languages can communicate with external hardware components such as sensors or lightbulbs (Richardson & Wallace, 2016). GPIO also enables direct coupling with breadboard kits, eliminating any barriers to circuit prototyping. Moreover, GPIO pins can be used to plug directly into specialized hardware extensions known as *hats* or *shields* that provide additional functionalities to the host board.

[6] https://developer.nvidia.com/cuda-toolkit
[7] https://systemd.io
[8] https://www.python.org

2.4 Neural Waveform Synthesis

The field of AI is built around the notion of an "intelligent agent" that evaluates inputs based on a model to make decisions in the form of outputs (Russell & Norvig, 2003). In general terms, AI models are systems of mathematical functions that parse input data through a series of actions in order to achieve a desired outcome. Machine learning is a specific type of AI that takes a data-driven approach to problem solving. Alpaydin (2014) depicts a common contemporary scenario, where a problem is so large-scale or nuanced that it is challenging to solve through procedural programming logic, yet relevant data abound. A machine learning algorithm can be fed those data and charged with the responsibility of creating the model on its own. Deep learning describes a particular kind of machine learning where the series of input processing functions has multiple hidden layers of abstraction before reaching the output stage (Goodfellow et al., 2016). These systems have proven fruitful for unearthing elaborate patterns from vast multi-dimensional data sets, and have recently shown promise as generative systems for audio and music (Briot & Pachet, 2018; LeCun et al., 2015).

In 2015, McFee et al. unveiled *librosa*[9]: a foundational Python library for music and audio signal processing that provides many of the core functionalities needed for research at the intersection of deep learning and music, a crossroads known as *music information retrieval* (MIR). Python is an interpreted programming language, meaning that it is sequentially translated into low-level code at runtime. This approach sacrifices a degree of processing efficiency in order to provide considerable advantages in readability and ease of use (De Pra et al., 2018). Python is also open source and has a wealth of high-quality scientific libraries, making it a popular choice for machine learning applications (McFee et al., 2015). Since the introduction of librosa, Python has become a go-to language for MIR projects, as the convenient workflow offered by libraries and the interpreter is of greater importance to many research initiatives than execution speeds (De Pra & Fontana, 2020).

Deep learning research reached a new threshold of development in 2016 with imminent implications for the world of digital sound when Oord et al. (2016) revealed WaveNet, the first deep learning-driven waveform synthesizer capable of producing natural-sounding raw audio. WaveNet was introduced in the context of speech generation, but the authors noted that it also

[9] https://librosa.org

performs admirably when trained with musical material. Prior to this breakthrough, deep learning techniques for generative music were in large part reliant on symbolic data representations beyond the audio domain. As opposed to producing raw audio, most legacy techniques generated performance instructions such as sheet music or MIDI (Briot et al., 2017). According to Oord et al. (2016), WaveNet is autoregressive and wholly probabilistic, meaning that it creates waveforms sample-by-sample, basing each output value on those that precede it. Its neural architecture consists of dilated causal convolutional layers, each tier representing a logarithmically scaled receptive field length. This model allows for temporal awareness proportional to the longest dilation.

Although the original implementation of WaveNet is robust with respect to the realism of its generative output, its sequential calculation process makes inefficient use of computing hardware resources (Zhao et al., 2020). In response to this challenge, Oord et al. (2018) proposed a method for converting a trained WaveNet into a parallel feed-forward network optimized for GPU rendering, achieving significantly enhanced processing speeds (see also: Purwins et al., 2019). In the wake of the success of WaveNet, comparable audio generation methods have proliferated and flourished. Among them are SampleRNN, WaveRNN, WaveGlow, WaveGAN, and others (Kalchbrenner et al., 2018; Mehri et al., 2016; Prenger et al., 2019; Yamamoto et al., 2020). Each of these waveform synthesis variants strives to solve the rendering rate deficiency present in the original WaveNet by employing a different neural structure, underscoring the research interest in optimizing these synthesizers for real-time audio generation. The recently unveiled Differentiable Digital Signal Processing (DDSP) library[10] and the neural source-filter (NSF) modeling framework also offer a set of efficient alternatives to autoregressive models (Engel et al., 2020; Wang et al., 2019). According to the qualitative experiment by Zhao et al. (2020), the NSF model is especially well-suited for musical applications. Taken together, all of these innovations are spurring the inevitable ascent of a new deep learning-based sound design methodology: neural synthesis.

Developed by Google Magenta, DDSP is a Python library of AI-powered sound shaping modules designed to enable interplay between traditional DSP methods and cutting-edge deep learning techniques. It builds on the work in NSynth, a pioneering neural synthesis data set and WaveNet-style pipeline for musical instrument sound generation developed by the same team in

[10] https://magenta.tensorflow.org/ddsp

2017 (Engel et al., 2017). Musical timbre-transfer is one of the most notable neural synthesis use cases illustrated in the DDSP documentation: By extracting the fundamental frequency and loudness features from an input signal, timbre information learned from an audio data set can be synthesized to match the pitch and loudness characteristics of the original input. These features are extracted using the CREPE[11] pitch tracker by default (Kim et al., 2018); however, there are many alternative feature extraction methods available throughout the field (Babacan et al., 2013).

DDSP provides pre-trained timbre models for neural synthesis, as well as tools for training new models from custom audio data sets. Rather than generating every aspect of the output waveform from scratch with a massive neural network, Engel et al. (2020) argue that using deep learning for specific purposes alongside classical DSP components allows the resource-intensive neural modules to focus only on the most important symbolic elements of the audio, thereby decreasing the computational workload for the entire pipeline.

2.5 Low-Latency/Real-Time DSP

Processing digital signals in real time imposes rigid requirements on integrated systems to perform predefined operations within an imperceptibly small time interval. To achieve real-time results, processors must return completed outputs faster than they receive inputs (Kuo et al., 2013). These systems either process analog input signals that have been converted from continuous time to discrete time, or generate outputs based on symbolic digital input data such as MIDI. Although signal processing hardware capabilities advance each year, there is still a relatively low ceiling for real-time performance, especially when monetary expense is a factor of concern (Michon et al., 2019). One common method for addressing this challenge is to reduce the frequency of I/O operations by processing data as buffered frames instead of as individual samples. Alas, this approach introduces a trade-off in the form of an output delay dependent on the length of the buffer (Vaseghi, 2008). Any latency created within the signal processing pipeline must be taken into account during evaluation for real-time applications.

The Python programming language is rarely considered for real-time audio signal processing applications, as the latency introduced by the code interpreter tends to exceed acceptable levels (De Pra & Fontana, 2020). In order to teach engineering students about

[11] https://marl.github.io/crepe

real-time DSP, Wickert (2018) presented a Python-based real-time audio pipeline. While appropriate for his specific use case, Wickert's implementation resulted in output latencies exceeding 70 ms even with minimal processing, making it unsuitable for settings with tighter requirements. De Pra et al. (2018) compared possible remedies for this deficiency involving the use of Cython[12] and C code modules for handling the real-time processing components. Python was originally written in the C programming language, and the native Python/C API facilitates the integration of raw C code into Python for applications where processing speed is critical. Major audio and data processing Python libraries such as librosa, Numpy[13], and TensorFlow[14] all rely extensively on C or C++ code to optimize performance where possible. Unfortunately, the API for creating these modules is difficult to implement without extensive C programming knowledge. In response to this challenge, Behnel et al. (2010) released the Cython extension in a bid to preserve Python's strengths while allowing programmers to tap into the speed of the low-level C language. In their experiment, De Pra et al. (2018) found that binding C code to Python produces the most efficient real-time audio processing outcomes by a significant margin, confirming that low-level control is necessary to achieve brief enough latencies for music performance applications. In 2020, they expanded on the scope of their research to consider the just-in-time Numba[15] compiler as a contender for Python-based real-time audio (see also: Lam et al., 2015). De Pra and Fontana (2020) concluded that between Cython and Numba, Python extensions can achieve sufficient speeds for hosting real-time DSP projects; however, minimizing processing latency depends on the strategic design of any such audio pipeline to take advantage of each extension.

Real-time latency challenges can be conveniently sidestepped by introducing offline processing into the equation. This refers to the background rendering of audio that cannot be delivered in real-time. Other tasks can be executed while this processing is taking place, such as playing an existing audio file and adjusting real-time effects parameters. Once the offline render is complete, the newly processed audio is available to trigger for real-time playback.

[12] https://cython.org
[13] https://numpy.org
[14] https://www.tensorflow.org
[15] https://numba.pydata.org

2.6 Neural Synthesis on Embedded Devices

Embedded systems will serve a vital role in the application of deep learning technology throughout our physical world, as smart devices increasingly permeate modern-day life (Srinivasan et al., 2019; Süzen et al., 2020). In 2017, Lane et al. reported that local training of deep neural networks on resource-limited embedded devices was widely viewed as impractical under present-day hardware constraints, though even at their time of writing, on-board execution of pre-trained networks was already a blossoming area of research. Bechtel et al. (2018) presented a miniature autonomous vehicle capable of calculating real-time steering decisions on a Raspberry Pi, all based on a deep convolutional neural network (CNN) architecture. Two years ago, Bhattarai et al. (2020) demonstrated how a deep CNN can be run on an Nvidia Jetson-based embedded system to provide firefighters with real-time augmented reality support tools. Drakopoulos et al. (2019) showed that even a previous-generation Raspberry Pi 3 Model B+ is capable of real-time deep learning-driven audio signal processing, with output latencies climbing in direct relation to the quantity of parameters in the pre-trained network. These recent achievements bode well for future research into embedded applications for AI-powered real-time DSP on GPU-optimized systems.

A crucial consideration for local deep learning inference on embedded devices is the on-board memory and computational cost of the pre-trained model. To address this type of challenge, Evci et al. (2020) introduced RigL: a neural network sparsification toolset that allows users to impose strict memory and inference time constraints for the sparse model to accommodate. The authors demonstrated that RigL leads the field in reducing neural operations while maintaining accuracy. Sparsified, compressed, or otherwise condensed models may prove essential for achieving real-time neural synthesis on a low-resource embedded system.

The DDSP library has also emerged as a likely cornerstone for the goal of achieving robust neural synthesis on an embedded prototyping board, as the modular structure of the library is ideal for isolating and optimizing resource-hungry elements (Engel et al., 2020). After separating the neural components from traditional DSP and clerical operations (such as library imports and initializations, etc.), computationally intensive modules can then be modified or replaced with less expensive alternatives. Costly processing tasks can even be systematically ported to lower-level code and augmented to support hardware acceleration.

3. METHODOLOGY

In order to realize responsive DSP and neural waveform synthesis on the Nvidia Jetson Nano 2GB, custom foundational software and hardware infrastructure must be preemptively designed and implemented. This section delves into the specific approaches and procedures employed to pursue these goals, and is framed as a guide for readers wishing to follow along. First, the core hardware and software environments are researched and selected based on their suitability for the project. Next, the software workspace is initialized on the chosen hardware to promote efficient experimentation and iteration through the tandem development phases. With the workspace refined and the peripheral components confirmed, the first hardware prototype is finalized as a unified product to host the ultimate stages of the software development campaign. Lastly, once the key software functionalities are achieved on the finished physical prototype, the proof-of-concept device is evaluated for its performance and provided to users for qualitative feedback. The final product yielded by this project is hereinafter referred to as the *raivBox*.

3.1 Preparing the Jetson

Due to the dual needs for an AI-focused host platform and affordability, the Nvidia Jetson Nano 2GB development board is a natural choice for hosting the raivBox. Prior to the initial boot-up, a micro SD card and USB-C power supply must be acquired, prepared and connected to provide the board with storage and power. The micro SD card should have at least 32 GB of storage capacity and be free of important data, as this process begins with a memory wipe. The Jetson also lacks a built-in WiFi card; however, Nvidia now ships the development kit with a compatible USB-A adapter to offer internet connectivity and convenient command line access over the local network.

First, the blank OS for the Nvidia Jetson Nano 2GB must be loaded onto the micro SD card using a desktop or laptop computer. The compressed OS file is publicly available on Nvidia's website, but care should be taken to avoid selecting the "Nvidia Jetson Nano" OS file, as the 2 GB model has a unique OS. At the time of writing, Nvidia propagates the assumption that the relatively expensive 4 GB version of the device is being referenced when "2GB [*sic*]" is missing from the title. The image flashing process can be conducted using any standard

imager/etcher software; the free Raspberry Pi Imager[16] application for macOS[17] is used for this project. In the "Operating System: Choose OS" dropdown menu of the Raspberry Pi Imager, the final option, "Use custom", must be selected. The blank OS image .zip file is picked via the browser popup, and then the target SD card is chosen from the "SD Card: Choose SD Card" dropdown. After confirming that the target drive selection is correct, the Write process is ready for execution. This process erases any remaining data on the SD card, replacing the entirety of its storage with the Jetson's OS. Once the OS has been properly installed and verified, the SD card can be ejected from the host computer and clicked into its slot on the Jetson.

Unless a dedicated power switch adapter is connected between the USB-C power supply and the Jetson, plugging it in will automatically launch the Jetson's startup sequence. A small yellow-green light-emitting diode (LED) on the board indicates that it has been powered on. During the first boot-up of the board and initial setup process, a USB keyboard/mouse and an HDMI display can be helpful. These peripherals must be plugged into the ports on the Jetson, or connected via Bluetooth adapters. The automatic login option offered during the setup wizard is recommended for this project. Once the username, password, and other basic user settings have all been confirmed, the Jetson can be connected to a local WiFi network and prepared for secure shell (SSH) and virtual network computing (VNC) access via local devices over WiFi.

3.2 Initializing the Software Environment

After the initial setup, the neural synthesis-bound development board requires initialization via the installation of numerous dependency libraries, including librosa, TensorFlow, DDSP, and many others. These libraries often have dozens of dependencies themselves, and some require that specific versions of each dependency be installed in a particular order to run properly. This project notably uses DDSP v1.9.0, Tensorflow 2.5.0, and librosa 0.8.1. Linux shell scripting provides a pragmatic way to systematically iterate through terminal installation commands, allowing for efficient debugging and convenient documentation. Many other important settings such as VNC parameters and login preferences can be adjusted via shell scripting during this process as well. The comprehensive custom initialization script

[16] https://www.raspberrypi.com/software
[17] It should be noted that a secondary host computer running macOS is used during this development process; references to external CLI commands assume a Unix-based system

assembled for the raivBox constituted a major developmental milestone, as it paved the way for confirming the viability of this project's goals. Achieving installation of all of the required libraries and getting them to successfully run in tandem showed that the basic functionalities do indeed exist in the Jetson's Linux environment.

Before the script can be executed on the Jetson, it needs to be loaded onto the board's *Desktop* directory, which can be done via a USB-A flash drive or by downloading directly from a hosting platform over the internet. When opening the script in a text editor, an important terminal command is exposed that must be executed independently before attempting to run the script. This command changes the shell script's system permissions to enable its execution, and then proceeds to commence the initialization. At the beginning of the script, core user data is captured to be used for subsequent settings and commands. Text prompts are displayed in the terminal to assist in guiding the user through the process. The entire initialization takes upwards of 3 hours to complete. Early on, the script sets up VNC and corrects all the settings to facilitate developer access. It then purges unnecessary pre-installed bloatware from Nvidia to free up storage space. Next, the *jetson-inference* package from Nvidia is pulled from GitHub, prompting a round of user inputs to select modules to install. None of the modules in this bundle are required for this project, but PyTorch[18] may prove useful for future work. The following installation group, which is consequently the most time-consuming, contains a wide variety of essential Python libraries, along with software development applications VSCode[19] and Jupyter[20] for added convenience. The libraries provide required infrastructure for DDSP, as well as interactive control features for custom hardware-software interplay. Lastly, the system is calibrated for the serial peripheral interface (SPI) GPIO communication protocol. After a final reboot and a round of system updates/upgrades, the Jetson board is now primed for neural synthesis experimentation.

Some of the crucial libraries are particularly time-consuming to build and install, necessitating the step of *baking*, or backing up the full OS image post-library install. By baking the properly initialized image, the long install process only needs to be completed once and can subsequently be bypassed by simply flashing the baked OS onto a Jetson's core SD card. ApplePiBaker[21] is a free GUI application for Apple's macOS that makes it easy to back up these

[18] https://pytorch.org
[19] https://code.visualstudio.com
[20] https://jupyter.org
[21] https://www.tweaking4all.com/software/macosx-software/applepi-baker-v2

OS images in an optimized, compressed file. To bake the initialized OS, the Jetson is first powered off, and then the SD card is removed. Next, the SD card is connected to a separate host computer running the ApplePiBaker application. When the SD card is recognized by the host, it can be selected from ApplePiBaker's "Select Disk(s)" menu. In the options menu, the "Enable Linux Partition Resize" toggle must be activated. Otherwise, the application will create a baked backup file equal in size to the SD card's total storage capacity, wasting significant local resources on the host computer. At this stage, the backup is ready to begin. In the save window, the ".zip" extension must be selected. If performed properly, the resulting file should be approximately 10 to 20 GB in size. This baking procedure is also useful for saving work prior to important developmental junctures, lest an install or environment is corrupted. Fully baked images can be loaded back onto the SD card with the Raspberry Pi Imager, using the same process as was previously used for flashing the blank OS during the initial setup.

3.3 Gathering the Hardware Components

The Nvidia Jetson Nano 2GB does not offer built-in analog audio I/O hardware connections out of the box, but this deficiency can easily be patched with an inexpensive USB-A adapter. At the time of writing, hardware brand Sabrent[22] offers a suitable audio I/O adapter for under $10, which is used for this project. Linux comes with Advanced Linux Sound Architecture (ALSA)[23] and PulseAudio[24] command line interfaces (CLI), making the core audio functionality of this board (and other similar Linux-based systems) relatively straightforward to access. These resources are well documented and optimized, and can be called via programming languages such as Python and C++.

Since this project aims to divorce neural synthesis from the confines of the traditional desktop computer interface, alternative forms of visual feedback are necessary to provide an enjoyable and intuitive user experience. Like the ubiquitous Raspberry Pi embedded development platform, the Nvidia Jetson Nano line includes 40 pins of GPIO connections. These pins facilitate communication between software and hardware elements such as LEDs, display screens, potentiometers, buttons, as well as many other sensors and controls. To preserve DIY

[22] https://www.sabrent.com/product-category/audio
[23] https://alsa-project.org
[24] https://www.freedesktop.org/wiki/Software/PulseAudio

simplicity, the GPIO-connected external hardware elements for this project are minimized to include only three LEDs, three potentiometers, a miniature organic light-emitting diode (OLED) display, and two arcade buttons. A handful of internal intermediary components are also needed for these connections, namely resistors for the buttons and LEDs, and an analog-to-digital converter (ADC) to digitize the potentiometer readings.

3.4 Prototyping the Unified Hardware Device

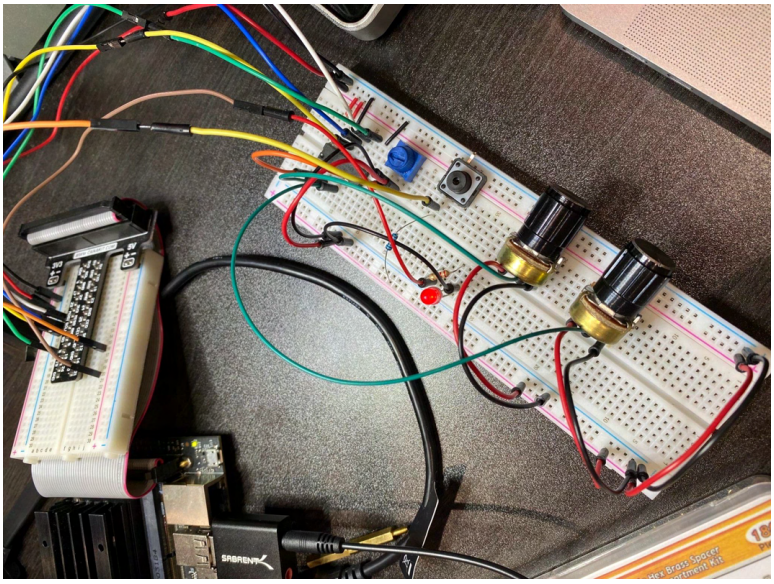


Figure 3.1: The breadboard configuration used for hardware prototyping

During the prototyping stages for the hardware control interface, breadboards and a GPIO breakout connector are employed. These tools provide a workspace to test the circuits and develop/assemble the necessary software to interact with each component. The first and most basic hardware connection to test is LED control. By connecting a GPIO control pin to a 220Ω resistor, the resistor to the anode of an LED, and then the cathode of the LED to ground, the control signal sent via GPIO can be used to turn the LED on and off with a high degree of precision control. The Nvidia Jetson's factory OS comes with a modified clone of the Raspberry Pi's *RPi.GPIO* Python library, called *Jetson.GPIO*, which enables most of the same code bindings from ubiquitous Raspberry Pi GPIO tutorials to work on Jetson boards as well. Using this library, simple commands can be assembled into expressive LED blink sequences among other visual control responses.

The two arcade buttons are connected to the Jetson's GPIO pins in a slightly different configuration from the LEDs: for each button, a 3.3V power pin is connected to a 1kΩ resistor, and on the other side of the resistor are connections to one of the button's two terminals as well as a GPIO control pin. The button's second terminal is connected to ground. Now, when the button is pressed and the connection to ground is completed, the voltage entering the GPIO control pin will drop, which is read by the Jetson board as a signal of binary state change. By identifying the relevant control pins in Python with the *Jetson.GPIO* library, these signals can be employed to trigger specific system commands. The main purpose of the two buttons is to provide direct recording and playback controls on the raivBox. On the left side is the record button, and on the right is the play button, sending ALSA commands to write and read audio.

Multiple steps are necessary to provide hardware controls for the dynamic adjustment of the Sabrent USB adapter's audio input and output volume levels. First, analog potentiometer data must be converted to digital information. Unlike the Arduino line of embedded development kits, the Nvidia Jetson does not offer built-in analog-to-digital conversion, so an external ADC chip is acquired for the task. The MCP-3008 integrated ADC chip is a common and inexpensive solution to this problem. It features eight data channels and is popular in the Raspberry Pi community, making it suitable for this use case. The MCP-3008 needs to be connected to one of the sets of SPI pins on the Jetson board's GPIO header, in addition to a normal control GPIO pin, a 3.3V power pin, and a ground pin. Since this project only requires three 10kΩ analog potentiometers, just three of the ADC's eight channels are used. Those channel pins are connected to the center pin of the three-pin potentiometers, with the other pins connected to 3.3V power and ground, respectively.

When properly connected, the data from the potentiometers via the ADC is available to access and manipulate with Python. Using libraries and MCP-3008 code examples provided by Adafruit[25] and other open-source developers, the raw ADC data are parsed and scaled to a user-defined range, and then piped into terminal commands to manipulate system features on the Jetson. Two of the potentiometers control the audio I/O levels using the PulseAudio CLI, and the third potentiometer is used to select which neural timbre model to use for audio synthesis. Furthermore, specific potentiometer states can be mapped to execute additional commands. For instance, in the functional prototype version of the raivBox described in later sections, setting the

[25] https://learn.adafruit.com/mcp3008-spi-adc/python-circuitpython

model selection knob at the top of the device all the way to the left displays a prompt on the OLED display screen reading “To SHUT DOWN, set all knobs to zero”. As indicated, also turning both of the volume level knobs all the way down will initiate the system’s shutdown sequence. This is done by adding conditional statements to the core code loop that checks the potentiometer values and the resulting software parameter adjustments. If all three knobs report values matching those criteria, the loop is exited and the shutdown command is called.

Adding the miniature OLED display screen is an exercise in both improving the user experience of the raivBox and future-proofing its early prototype design. The screen provides visual feedback to complement the hardware controls, enabling clarity of use for increasingly complex control combinations and patterns. The screen selected for this project is made by Adafruit for the Raspberry Pi, but it is used in at least one official project tutorial advertised by Nvidia for the Jetson Nano line. The Jetson-focused educational website JetsonHacks.com also published a clear guide[26] on how to set up this very display with the Jetson Nano 2GB board, making it straightforward to integrate into this project. This small screen module connects to the upper six pins of the 40-pin GPIO header and displays a selection of core system status data. It utilizes systemd to automatically start at system boot and stop at shutdown. With the OLED installed, important system information is available without needing to connect to any external displays, networks, or devices. This convenience makes the raivBox a significantly more independent and user-friendly device in its complete prototype form.

All of the faceplate-mounted hardware components need to be individually connected to the Jetson board’s GPIO pins in order to communicate with the software for this project. Designing a custom printed circuit board (PCB) is the clear solution for streamlining the wiring process. Using the free PCB design software KiCad[27] and the finished wiring prototype on the breadboard, the raivBox’s custom PCB is fashioned in the form of a hat/shield-style removable header attachment. This way, the added hardware can be easily modified or replaced without requiring high-stakes resoldering that could potentially harm the Jetson itself. See Figure 3.2 for the PCB schematic, and Figure 3.3 for renders of the PCB design:

[26] https://jetsonhacks.com/2019/12/03/adafruit-pioled-on-jetson-nano
[27] https://www.kicad.org

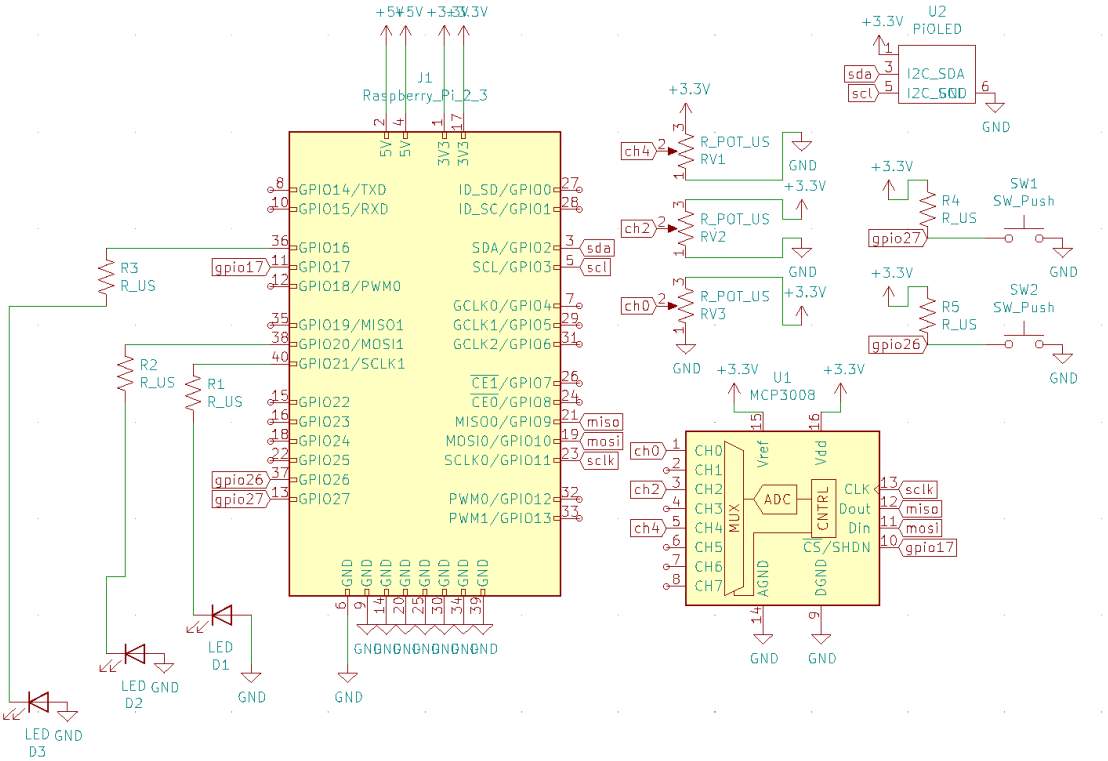


Figure 3.2: KiCad PCB schematic for the raivBox hardware prototype

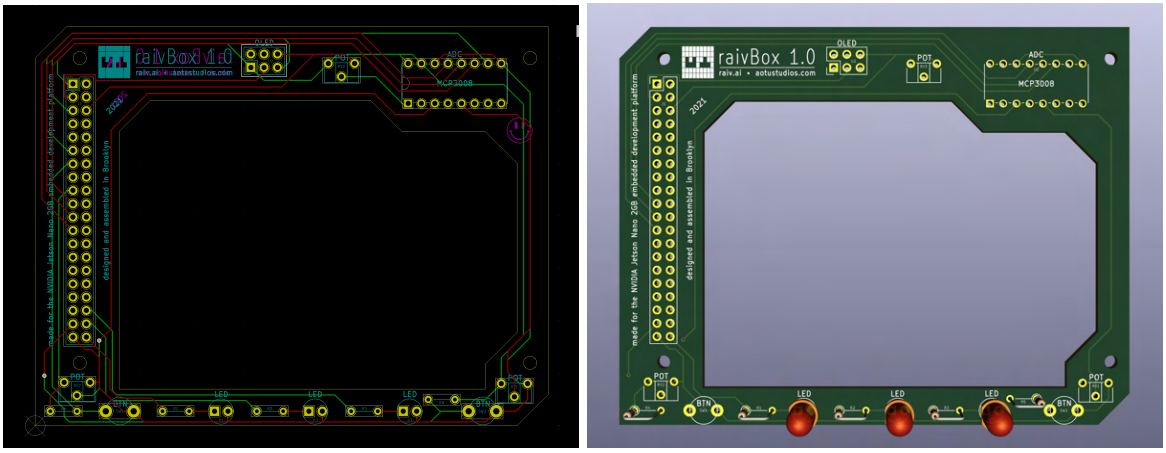


Figure 3.3: Trace pattern and render of the KiCad PCB design for the raivBox hardware prototype

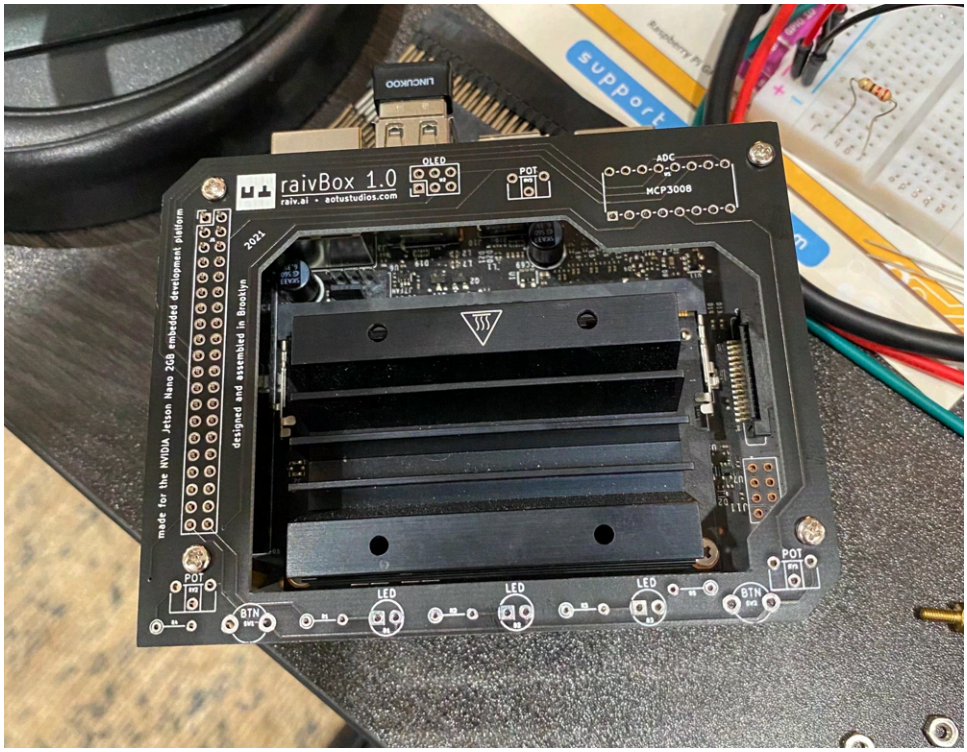


Figure 3.4: The unsoldered raivBox PCB attached in position on the Nvidia Jetson Nano 2GB

The unified product version of this project is intended to be usable, serviceable, and even replicable by musicians, hobbyists, and laypeople alike, so the custom hardware components must reflect that goal. For this reason, 3D printing is an appropriate option for creating the product housing, as printed components are inexpensive, accessible, and can also be tailored to precise dimensions. Autodesk's Fusion 360 software[28] is used for modeling the printable parts. There are a total of three 3D printed components: the five-walled base of the housing, the faceplate with mounting holes for the hardware controls, and a small U-shaped plug that helps push the power-cutoff button on the unit's internal battery pack.

Planning and designing the 3D printed housing is a gradual process during the hardware component selection stage. The initial constants are the Jetson Nano 2GB, the included WiFi adapter, and a compatible cooling fan attachment for the integrated heatsink, but these elements quickly snowball to include the Sabrent USB audio adapter, a USB-C battery pack (and the short USB-C cable needed to connect it to the Jetson), the custom PCB to organize the GPIO connections, as well as all of the previously addressed hardware control and interfacing

[28] https://www.autodesk.com/products/fusion-360

components. Physical size is carefully minimized while still allowing for adequate internal airflow and structural integrity.

What follows is an overview of the device's assembly, starting from the base: First, the battery pack fits into a narrow slot at the bottom of the housing, with a 1.5 mm gap above it for airflow. A solid lip at the front of the base holds the battery pack in place. Two prongs extending from the U-shaped plug protrude from a pair of holes on the right side of the housing, creating pass-through access to the native power cutoff button on the battery pack itself. The Nvidia Jetson Nano 2GB is mounted directly above the battery using M2.5 standoffs that fit into four dedicated holes in the housing base. The cooling fan is mounted directly to the top of the Jetson's built-in heatsink. At this point, the Sabrent USB audio adapter can be plugged into its port on the Jetson through its dedicated hole in the back of the housing. The fully wired custom PCB header is then plugged into the Jetson's entire 40-pin GPIO pinout using header extension adapters, and screwed down to the same mounting holes as the Jetson board itself via additional stacking M2.5 standoffs. All of the faceplate components or their respective wires are connected to the PCB header: the OLED display, ADC chip, potentiometers, buttons, resistors, and LEDs. The arcade buttons snap into their holes on the faceplate, and can then be attached to their wire connections from the PCB. The LEDs are similarly plugged into their wire connectors and fitted into their holes on the faceplate. Next, the potentiometers can be pulled through their holes on the panel and secured with their included fastening nuts; then their knob caps easily press into place. When properly mounted to its pin connectors on the PCB, the OLED perches in position under its window on the faceplate. After securing the control panel to the base of the housing using M2.5 standoffs and its four corner screws, the physical construction of the raivBox is complete. To provide battery power, attach the short double-sided USB-C cable into the Jetson and the battery pack through the two remaining holes on the back of the housing. If the battery is sufficiently charged, the raivBox will now boot up and await user input. Alternatively, the raivBox can be powered by connecting the Jetson port to a standard outlet using a 3.5A USB-C power supply. Charging the battery is conducted through connection to a wall socket in this same manner.

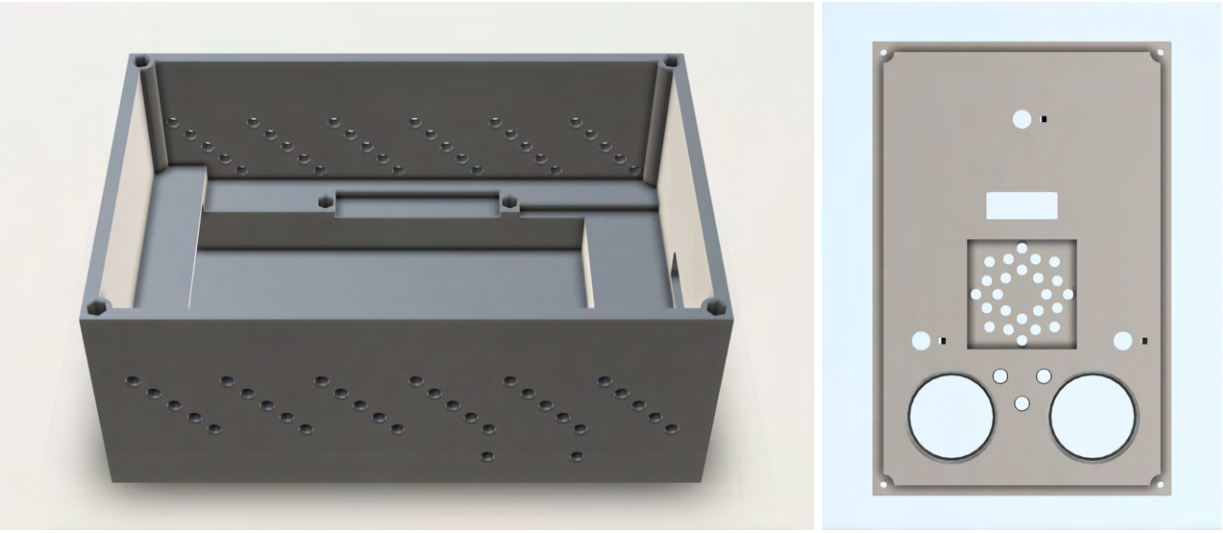


Figure 3.5: The final 3D models of the raivBox housing base (left),
and faceplate (right; seen from underneath)

The raivBox is designed to be either held with both hands or placed on a table. Together, the two large arcade buttons constitute the main performance controls, with the three potentiometers intended to be set and then left in position. Each button is situated on the faceplate with the goal of ergonomically lining up with a user's thumbs when the device is held, like a video game controller. When placed flat on top of a table, the buttons can be performed with one hand while the other is used elsewhere, for instance, to control a separate instrument or creative tool. Between the two arcade buttons, the three LEDs are arranged in an inverted triangle. These LEDs provide a variety of dynamic status signals, including boot-up and shutdown blink sequences, an audio processing indication, and basic button feedback. An audio level control knob is positioned next to its corresponding button: the input volume beside the record button on the left, and the output volume beside the play button on the right. In the center of the faceplate, there is a grate of holes to provide air intake for the Jetson's fan. Above the fan's position on the faceplate lies the OLED display, with the neural timbre model selection knob near the top. Figure 3.6 below shows multiple renders of the final raivBox prototype design, and Figure 3.7 shares photographs of the internal assembly:

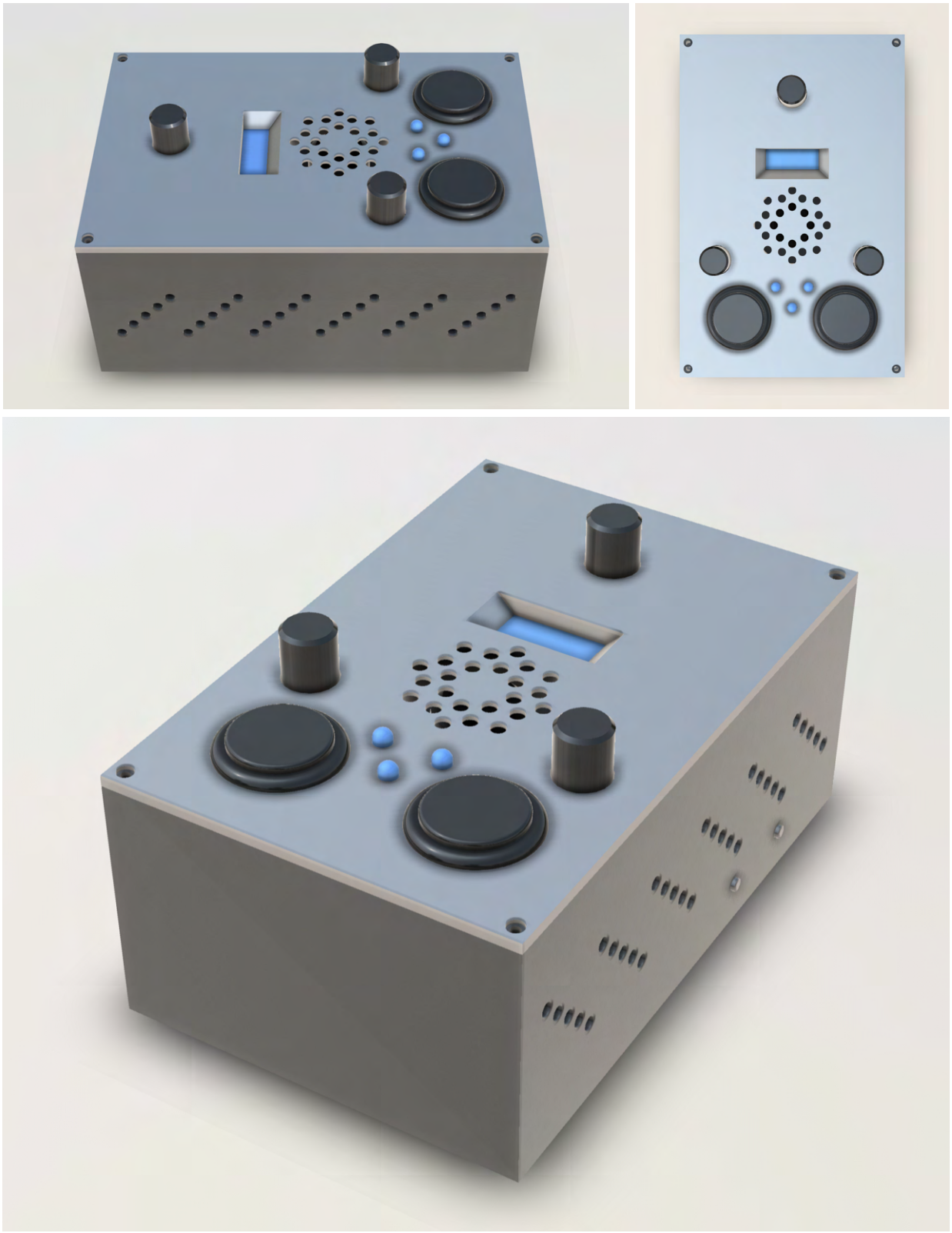


Figure 3.6: Various render angles of the final raivBox hardware prototype design

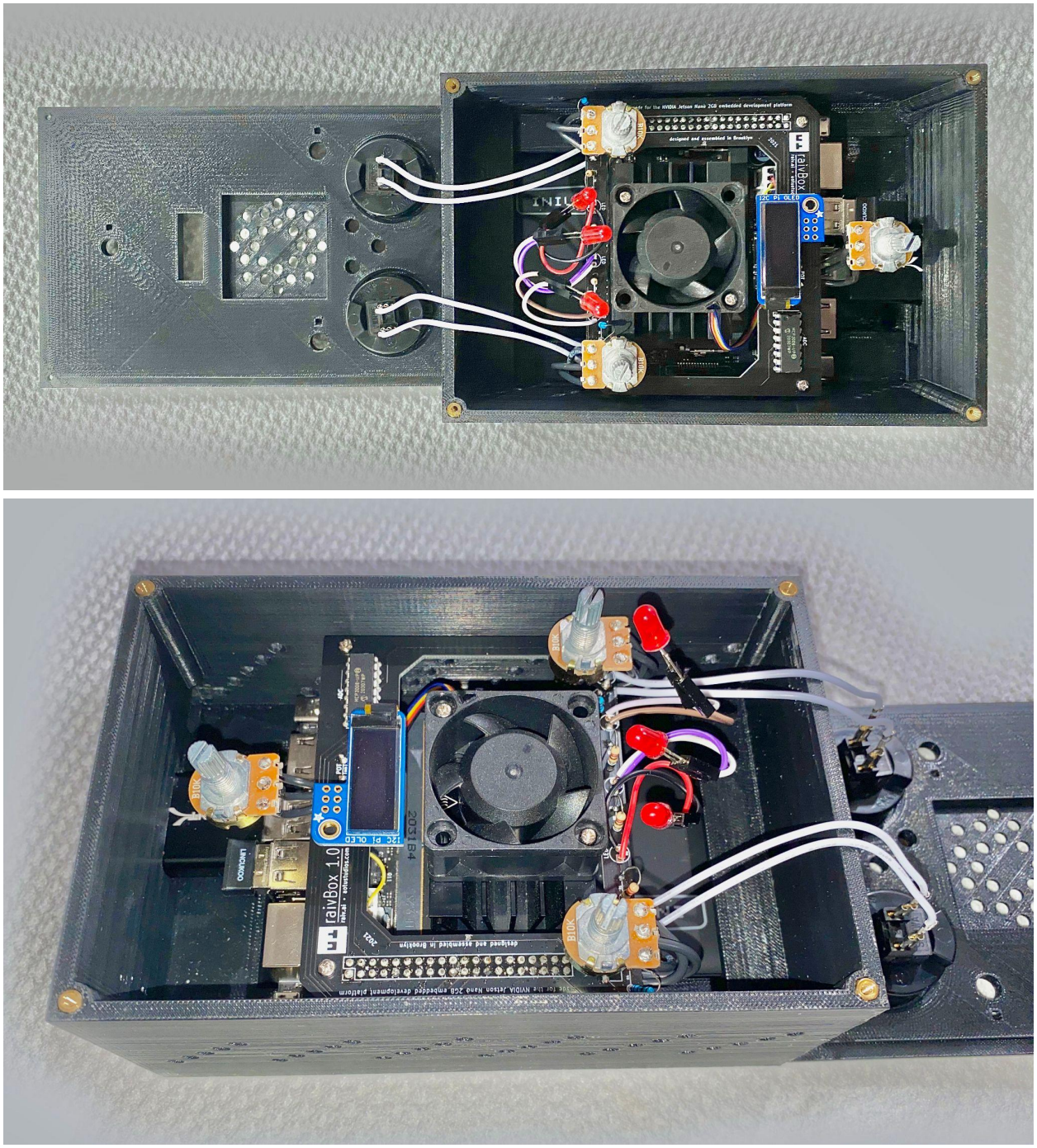


Figure 3.7: Two perspectives of the fully-assembled internals of the final raivBox prototype

3.5 Achieving Hardware-Software Interaction

Like the OLED display, all of the buttons, knobs and LEDs are automatically initialized at boot using systemd. To set up a systemd service, a *.service* file must be created in the *systemd* directory. This file type is used to specify what a background process does and how it is accomplished. For example, the service for the status LEDs on the faceplate of the device has a description of what it is (Description=LED control and hardware initialization), and detailed instructions for which environments and files to run. The main task for this service is to execute a Python file that first performs a blink sequence indicating that the system has started up, before proceeding to accept button input. When the two buttons are pressed at once, a second custom Python program titled *buttons.py* is triggered to initialize the internal audio signal flow pipeline.

Both system services and user services exist under the systemd umbrella. The system services operate with root permissions by default and therefore cannot manage certain important user-specific PulseAudio CLI operations. On the other hand, user services fail to automatically gain necessary sudo permissions at boot without piping in the administrator password—an action that would create security vulnerabilities. For these reasons, separate user and system services are needed for different aspects of the raivBox's custom systemd background process architecture. A system service handles the LEDs and buttons, and a user service is mapped to the knobs that control the PulseAudio source and sink volume levels.

The custom analog hardware interaction software called by the systemd services centers around four main Python files. The first file, *boot-leds.py*, initializes the LEDs and buttons, and then awaits input to switch into synth-mode. Next is *boot-knobs.py*, which initializes the three potentiometers and begins listening for audio level/neural timbre model selection changes. When *boot-leds.py* detects a call to enter synth-mode (the two buttons are pressed simultaneously), it begins a new background process by calling the third Python file, the aforementioned *buttons.py*. This third file sets up the buttons to function as record/play controls, and also calls the fourth file, *synth.py*, between these operations. When *buttons.py* receives a user command to record, it sends that audio recording to *synth.py*, which processes the audio in the background before returning its completed output file to the *audio* folder. As each audio file is overwritten by new recordings and newly processed outputs, the old files are renamed and moved to the dedicated *archive* folder in case the user wishes to retrieve them at a later date. Once the audio processing

via *synth.py* is complete, it indicates so with a solid status LED and a flag for *buttons.py*. Having received this notification, *buttons.py* can now trigger playback of the newly processed audio. This four-file core software architecture allows for programming modularity, as *synth.py* can potentially be replaced using a more resource-efficient coding language in future versions.

When a program is executed by a systemd service, it is effectively under that service instance's purview until the service is terminated or restarted. This means that changes to the main Python files after they have been run by *boot-leds.service* or *boot-knobs.service* will not be reflected until the device is rebooted or the services themselves are manually reengaged. Since the two main systemd services for this project are run with different privileges, one in the system space and the other in the user space, inter-program communication poses a challenge. To address this problem, the raivBox core software architecture introduces a solution that allows for thread-safe data transfer between the system and user permission levels, and enhances debugging capabilities for end users: A variety of specific *flags* are arranged as simple text files that can be read and edited by the core Python programs as well as the user. This structure streamlines the process of adding or replacing neural timbre models, provides directly accessible data to compare and debug control values, and enables the system to natively manage threading. Additionally, the raivBox relies on another systemd system service and Python file pair to communicate with the OLED display, *oled-stats.service* and *stats.py*. The *flag* arrangement provides a straightforward access point for all of the important data shown on-screen. A simplified diagram of the software architecture is shown in Figure 3.8.

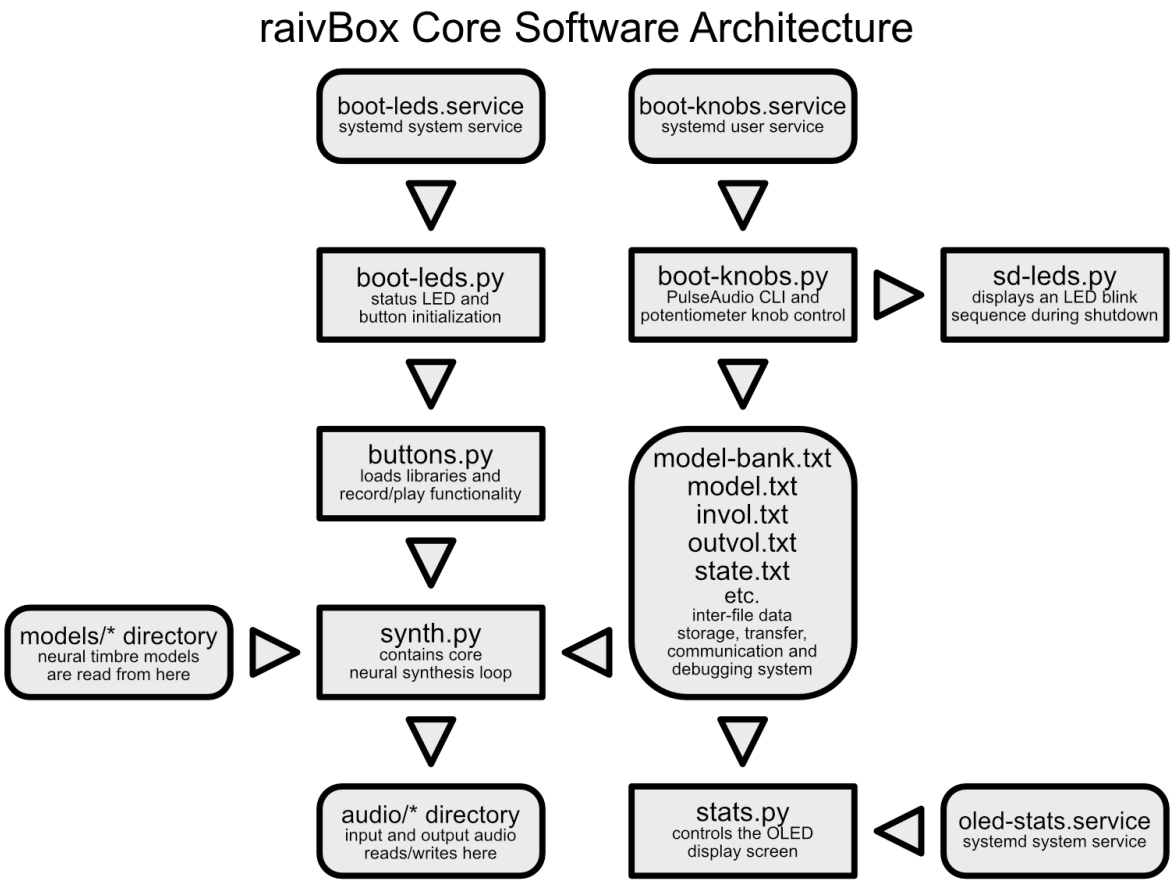


Figure 3.8: Simplified diagram of the raivBox core software architecture

3.6 Tailoring the Synthesis Pipeline

Now that the raivBox's core software architecture is in place, the neural waveform synthesis pipeline is ready to be optimized for inference on the embedded host device. This honing process commences with an initial implementation of the original DDSP timbre transfer pipeline on the raivBox.

First, the components of Google Magenta's timbre transfer demonstration Jupyter notebook are retrieved and isolated into modules. The main chunks are the library imports, the input audio feature extraction, the neural timbre model load-in, and the synthesis of the output audio. A variety of custom helper functions are loaded alongside the libraries as well. In early

tests, the initial library import process proved to be a time-consuming operation on the raivBox, necessitating a synthesizer structure where the libraries are loaded once at boot rather than starting over for each input. To achieve this, the raivBox imports all libraries at the beginning of *synth.py*, which then proceeds to a while loop to await new input audio via a flag trigger. This keeps the libraries available in random access memory (RAM) while the synthesizer is on.

The next step of the timbre transfer pipeline is the input audio feature extraction. In Google Magenta's original version, pitch is tracked using CREPE, a state-of-the-art pitch estimator based on a CNN (Kim et al., 2018). CREPE offers a selection of model capacities, with the smaller settings corresponding to reduced processing complexity with a slight trade-off in decreased estimation accuracy. At full capacity, CREPE is intolerably slow on the raivBox, as explored in the Analysis and Results section. After testing the smallest CREPE capacity options and determining that they are also too tedious, the decision is reached to replace CREPE altogether with a significantly faster pitch tracker: YIN (named after "yin and yang" [陰陽] from ancient Chinese philosophy). The YIN algorithm is a computational approach to pitch estimation introduced by Alain de Cheveigné and Hideki Kawahara in 2002. It is fundamentally based on autocorrelation, but employs various enhancements to reduce errors. This project makes use of the YIN implementation available in librosa. After swapping CREPE for YIN, the inference time decreases dramatically, and the outputs remain convincing and interesting to the ears of the author, especially in environments and use settings with minimal input signal interference. For instance, if a clean signal from an external instrument is fed into the raivBox as an input, YIN exhibits high accuracy; that said, an input with heavy reverb, polyphony, or background noise produces temperamental results (Babacan et al., 2013).

Loading in the neural timbre models also balloons the storage and time costs of each synthesis operation on the embedded prototyping platform, so methods for reducing the model size and pruning unnecessary aspects of the load-in procedure are sought. Success is achieved by converting the TensorFlow *Checkpoint* model format into a TensorFlow *SavedModel* file using the *ddsp_export* CLI provided with DDSP, and creating a new model directory consisting of the SavedModel *.pb* file, the export variables, and the *.gin* file from the original TensorFlow Checkpoint. The entire new model now takes up less than half the storage space of the original model. In the raivBox synthesizer Python file, this new model is loaded using the "model.load_weights()" method from TensorFlow, and is then run directly on the input audio

features without the step of building the model by running a preliminary batch through it. This final detail of ignoring the dry run model building step is the key processing latency enhancement; the model file size reduction provides negligible benefits in this area. Additionally, the author found the sonic characteristics of the smaller model files to be unique and appealing, and decided to incorporate them into the prototype despite the relatively minor improvement of the smaller size on disk. It is still possible to revert to the original Checkpoint model load-in style from Google Magenta's demonstration without sacrificing processing latency; however, this requires the heftier model files.

3.7 Refining the raivBox

With all of the critical software and hardware functioning properly in tandem, the raivBox is now ripe for refinement. It is intended to be a friendly device that pleases and engages users, so this polishing stage is important to the success of the evaluation phase.

The OLED display plays a key role in this refinement process, as it provides the basic future-proofing for the hardware prototype design. Rather than continuously displaying the same system data from boot to shutdown, the *stats.py* file that controls the OLED is modified to follow along through the startup process, providing guidance and brief ASCII animations for the user.

First, after the initial boot-up sequence, the screen shows a blinking prompt to "Tap both buttons simultaneously". This action begins the heavy audio processing Python library load-in procedure, accompanied by a dancing animation of the text "Loading heavy libraries". Once the libraries are fully loaded, the raivBox enters its main "synth" state, and the OLED switches to showing important data.

As mentioned previously, when the model selection knob is set near 0%, the screen displays "To SHUT DOWN, set all knobs to zero", along with the current percentage readings for the input and output gain knobs. These gain readings update as the user interacts with the knobs, making the resulting actions abundantly clear. The OLED panel has four available lines to display ASCII text, each offering room for a total of 22 monospaced characters. If the model selection knob is turned away from the shutdown position, the OLED shows the chosen neural timbre model, the IP address (if connected to WiFi), and the available memory, in the first, second, and third text lines, respectively.

The fourth and final line of text on the screen dynamically changes in response to the gain knob positions. When either of these knobs are set below 50%, both of the current readings are displayed. Having the input level too low causes pitch estimation errors and low output gain makes the outputs inaudible, so it is important to convey these data to the user. If both of these knobs are set sufficiently high, the output level percentage provides two information display state options for the OLED screen: When over 80%, the SD card storage usage is shown, and when set from 50% to 80%, the time it took to process the previous output is displayed in seconds.

After an audio input has been recorded using the arcade button at the bottom-left of the raivBox's faceplate, the screen shows the message "AUDIO PROCESSING" in the third text line of the OLED. If the input recording is too short for the neural synthesizer pipeline to ingest, it will not begin processing and thus the message will not be displayed.

Together, these data readouts and positional triggers expand the depth with which users can interact with the raivBox, by supplying consequential information without requiring connection to additional hardware.

4. ANALYSIS AND RESULTS

Now that the raivBox has reached a successful prototype state (see Figure 4.1), it is ready to be appraised for its performance and the quality of the experience it provides for end users. In this section, the functional details of the present iteration of the raivBox and its core neural synthesis pipeline are examined and then compared with Google Magenta's original demonstration of DDSP timbre transfer. Next, the device is tested and evaluated by a sample of music technology students at New York University. The results are presented and analyzed to provide clarity and insight into the current strengths and limitations of the raivBox.

Figure 4.1: Product photo of the functional raivBox prototype (center) sending audio into a teenage engineering OP-1 synthesizer (top), with an Nvidia Jetson Nano 2GB shown to the left, and an unsoldered raivBox PCB on the bottom right, among other props

4.1 The Working Prototype

As previously described, the raivBox prototype version introduced for this project features a customized adaptation of Google Magenta's DDSP timbre transfer audio processing pipeline, streamlined and optimized to run on a resource-limited embedded development platform, in this case the Nvidia Jetson Nano 2GB. The modified timbre transfer module is integrated into a signal flow software architecture that makes use of the Linux PulseAudio and ALSA CLIs, as well as the systemd background service manager, to offer direct user interaction via an original hardware control interface, in addition to convenient file archiving and SSH/VNC access. Users communicate with the raivBox by way of the hardware controls on the face of the device, and the information conveyed by the prominent OLED display screen and status LEDs. Its exterior measurements are 110 mm wide x 167 mm long x 71 mm deep, with an additional 14 mm in height due to the potentiometer caps. Having the power plug attached to the internal battery increases the working length of the device by another 44 mm. The sum total retail price for all hardware components at the time of writing is approximately $180.

4.1.1 Basic Operation

The first step in using the raivBox is to power it on. As mentioned in the Methodology section, the raivBox can be powered on by one of two methods, the first of which is to connect the internal battery to the Jetson's power jack with a male-to-male USB-C cable (see Figure 4.2). If the battery itself has been powered off with its native power button, it can be rebooted by simultaneously pressing both nodes of the power plug on the right side of the raivBox (see Figure 4.3). When the raivBox has been properly connected to power, the fan in the middle of the faceplate will twitch forward, and a small yellow-green LED light inside the housing will turn on. This light can be observed by looking into the ventilation holes on the left side of the device (see Figure 4.4). The second method for booting the raivBox is to connect the Jetson's power port to a compatible USB-C cord plugged into a standard wall socket. A 3.5A Raspberry Pi 4 power supply from Canakit[29] is tested and recommended for use with this prototype; it is also suitable for charging the raivBox's internal battery.

[29] https://www.canakit.com

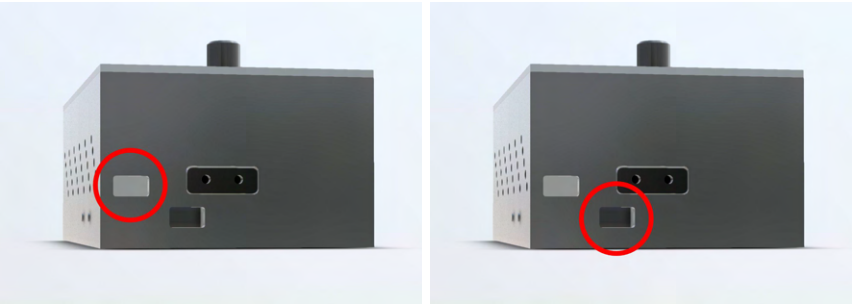


Figure 4.2: Render of the USB-C power jack aperture on the back of the raivBox (left)
Render of the USB-C battery connection aperture on the back of the raivBox (right)

Figure 4.3: Render of the two protruding nodes from the U-shaped
battery power-cutoff plug on the right side of the raivBox

Figure 4.4: Render of the yellow-green startup LED seen through
the ventilation holes on the left side of the raivBox

After the yellow-green startup LED activates, the raivBox takes approximately 30 seconds to fully boot up. At the end of the boot-up sequence, a brief animation is displayed on the OLED screen, followed by simple initialization instructions for the user. As communicated by the on-screen prompt, both main control buttons must be tapped simultaneously to initialize the device. Once this initialization call has been made, the OLED display shows a status message for the duration of the heavy library load-in procedure, which also has a duration of about 30 seconds. With the libraries now accessible from the device's RAM, it is ready to process audio.

The two 3.5 mm TRS analog audio I/O ports are found on the back of the device, where the USB audio adapter is flush-mounted with the housing. The input socket on the adapter is color-coded with a red ring (red for record), and the output socket is colored green. According to the hardware specifications, any standard microphone with a 3.5 mm male TRS output plug can be connected directly to the input jack on the USB adapter. The microphone tested for use with this prototype is an inexpensive ($5 USD) generic gooseneck with a native 3.5 mm output connector. The PulseAudio settings on the raivBox have been calibrated to automatically recognize a microphone as the default audio input device when it is plugged into the input socket of the adapter. Listening to the audio output from the raivBox is as simple as connecting headphones or a speaker to the USB adapter's 3.5 mm output port.

Figure 4.5: Render of the 3.5 mm audio I/O connections on the rear of the raivBox

With the libraries all imported and the analog I/O connections in place, the user can now record, process, and play audio using the neural synthesizer. The OLED display screen is set up

to show a variety of useful system status information, notably including the current neural timbre model selection. It can also show the IP address of the raivBox if it is connected to a local WiFi network, the amount of RAM in-use, the amount of storage available on the SD card, and the amount of time it took to process the most recent audio output. If either the input or output gain knobs are set to 50% or less, their levels will be displayed on the screen in percentage format to inform the user. The input audio is normalized for processing, which exacerbates the noise floor of faint input signals, making this information valuable. Having these settings displayed directly on the OLED also streamlines debugging of the potentiometers, if their intended values are ever returned incorrectly by the ADC.

The uppermost knob of the raivBox is used to choose from the bank of available neural timbre models for the synthesizer to generate outputs with. At this stage of development, four of the timbre models are size-reduced versions converted from the Google Magenta DDSP timbre transfer demonstration, and one is a custom trained model made for this project. The Google models are of a tenor saxophone, a trumpet, a flute, and a violin, and the custom model is trained on an original Roland TB-303-style acid bass sound created by the author in Ableton Live[30].

The raivBox's main performance controls are located in the lower half of the faceplate. On the left side is the record button and input gain knob, with the output gain knob and play button mirrored on the right side. With a gooseneck microphone attached at the back and extended over the top, the raivBox is ready to be held with two hands and used. It can also be placed on a tabletop for a similar albeit less-ergonomic experience. Figure 4.6 identifies each of the key components on the faceplate.

[30] https://www.ableton.com/en/live

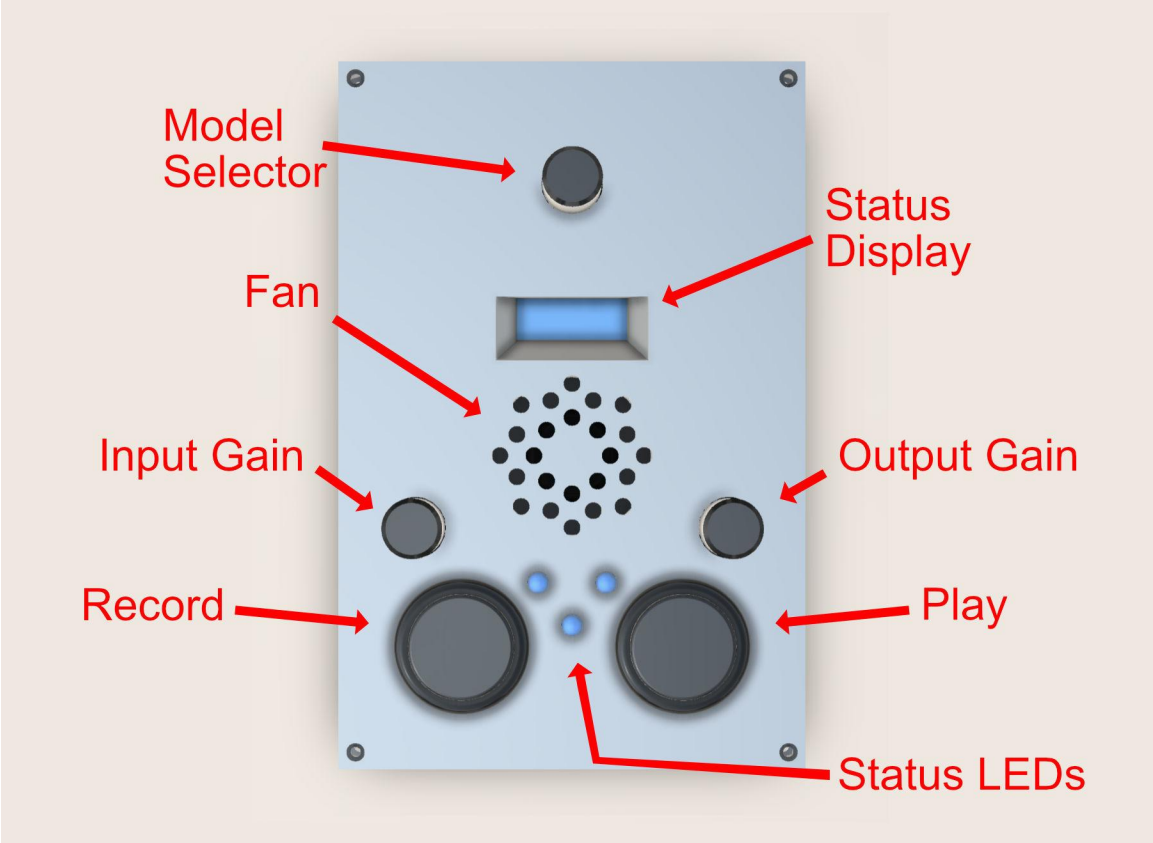


Figure 4.6: Labeled render of the raivBox faceplate control panel

4.1.2 Advanced Operation

For experienced professionals or hobbyists, the raivBox offers under-the-hood access via two main avenues. First, it is possible to connect to the raivBox via WiFi and modify or control it directly using SSH or VNC. The raivBox is set up to automatically connect to local WiFi in the test environment in which it was developed, making SSH/VNC access a simple command line operation. When the raivBox is connected to a network, the IP address is displayed on the OLED screen for convenient reference. By default, the raivBox's login credentials have the username "rb" and the password "raivbox". Its local IP address will look similar to "192.168.1.225" on standard systems, and can be connected to with this command format on Unix-based systems:

```
ssh rb@192.168.1.225
```

If the raivBox is already recognized by the local WiFi network and connects automatically at boot, it can likely be accessed with the following command:

```
ssh rb@raivbox.local
```

The above command may fail if a separate raivBox with the same device name has previously been connected to the network or the SSH host computer.

The most straightforward way to connect the raivBox to a local WiFi network for the first time is to remove the faceplate and plug the device into a router using an ethernet cable, and then find the device on the network to access via SSH. From there, the raivBox can be switched over to WiFi and configured for VNC. Before opening the faceplate, the raivBox must be powered down and disconnected from any power sources, including the internal battery. First, if the raivBox has not yet been powered down, all knobs are set to zero to initiate the shutdown sequence. Once the animation is finished and the OLED display is blank, any USB-C connections are unplugged. At this point, the knob caps can be safely removed by carefully pulling them off, and the mounting nuts then unscrewed from each potentiometer. Next, the four fastening screws should be removed from the corners of the faceplate, and the panel then lifted from its seat on the housing. The three potentiometers, two main buttons, and the status LEDs are all mounted in the faceplate. The potentiometers must be gently pushed down through the panel, whereas the buttons and LEDs can be cautiously removed from their connections to the PCB wires inside the raivBox as needed. Lastly, the USB audio adapter is removed from its port at the back of the Jetson board, and the newly freed space is used to attach the ethernet cable.

The second approach for accessing the underlying software on the raivBox also requires the user to open up the faceplate. With the internal components revealed, users can connect directly to the Nvidia Jetson Nano via USB and/or HDMI without requiring network access. When connected to an external HDMI display and USB mouse/keyboard, the Jetson has a full desktop computer interface. With the faceplate detached, the SD card with the OS image is accessible from the front of the Nvidia Jetson, underneath where the LEDs had been mounted.

To add custom neural timbre models, users can employ the command line or the desktop GUI of the Jetson board to copy the formatted timbre model files over SSH into the *models* folder within the main *raivBox* directory, and then add the new model's folder name to one of the five lines in the *model-bank.txt* file within the *models* directory. For example, to add a new model called "bassoon" to the raivBox, one would first convert a trained bassoon timbre model

into the proper TensorFlow SavedModel format, then match the file structure and formatting to the existing raivBox models, then copy the *bassoon* folder into the *models* folder on the raivBox, and lastly add "bassoon" to an available line in *model-bank.txt*. On the next boot, the raivBox should now show "BASSOON" as an option with the model selection knob. The current version of the raivBox software allows support for up to five timbre models at once, as model numbers exceeding this limit may prove challenging to select due to the precision constraints of the model selection potentiometer. That said, the only true limit to the number of models supported by the raivBox at any given time is the storage capacity of the SD card, provided that the potentiometer could be replaced with a higher-precision component.

If advanced users wish to modify the synthesizer itself or contribute additional DSP to the audio processing pipeline, they can directly access and edit *synth.py* in the *raivBox* directory on the desktop. The core synthesis while loop at the end of *synth.py* has a demarcated section with instructions for adding additional DSP code as desired (lines 812-826 at the time of writing).

4.2 Performance

The augmentations made to Google Magenta's DDSP timbre transfer pipeline for the raivBox are designed to reduce the computational load on the inference machine while preserving output quality. This enables robust timbre transfer on the Nvidia Jetson Nano 2GB host device with minimized processing latency. Although the audio outputs from the raivBox are similar to those of the original Google pipeline, there are noteworthy differences in performance and character. The key alterations made to Google's version are in the input audio feature extraction and trained model loading steps of the neural synthesis process.

4.2.1 Feature Extraction

While Google Magenta's timbre transfer pipeline uses the machine learning-based CREPE pitch estimator to perform audio feature extraction, the raivBox pipeline instead relies on YIN pitch tracker from the librosa audio processing library. YIN offers an efficient computational approach—it is significantly less resource-intensive than even the smallest available CREPE model capacity. By using YIN, processing power and time are both conserved

for the inference stage of the timbre conversion process rather than feature extraction. In contrast to this latency improvement, YIN is a less accurate and reliable pitch estimation method when compared with CREPE. The data used to train the neural timbre models is also produced using CREPE within Google's open-source training pipeline, meaning that inconsistencies in feature data formatting between the YIN implementation and CREPE may lead to unpredictable results. To compensate for this eventuality, the YIN data in the raivBox pipeline are reformatted to better match the CREPE data expected by the model, by filling the "f0_confidence" argument with an array of floating-point 1 values. Each 1 corresponds to full confidence, instructing the pipeline to proceed with the frequency values predicted by YIN without reservation.

4.2.2 Trained Model Footprint and Load-In

The model files created using Google Magenta's publicly available timbre model training Jupyter notebook take up a relatively large amount of storage space in the raw Checkpoint file format they are generated as, weighing in at approximately 58 MB in total per model. In order to reduce model size on disk and decrease the DDSP timbre model load-in overhead on the raivBox, the model import module in the raivBox pipeline has been modified to read in the TensorFlow SavedModel file format rather than requiring the full raw Checkpoint. The Checkpoint is converted into the SavedModel format using the *ddsp_export* CLI command included with the DDSP Python library:

```
ddsp_export --model_path=/path/to/model \
     --inference_model=autoencoder
```

After converting the Checkpoint into a SavedModel, a few additional files from the original Checkpoint folder and the SavedModel export process must be included alongside the SavedModel file to load the timbre data. Specifically, *operative_config-0.gin* from the original Checkpoint folder, as well as both *variables.data-00000-of-000001* and *variables.index* from the SavedModel export are needed. This all results in a working timbre model that clocks in at approximately 25 MB, less than half the size of the original Checkpoint. As previously noted, this reduction of the model file size on disk does not produce noteworthy improvements to processing latency.

In a final step of the original Google Magenta model load-in procedure, the model is built by executing a dry run before performing inference on the input audio features. This building step does not deliver noticeable improvements to sonic character or performance, and is thus removed from the raivBox version. As shown in the following section, disabling this step drastically reduces the processing latency of the pipeline.

4.2.3 Results

The raivBox pipeline provides a substantial processing latency improvement over the original Google Magenta pipeline it is built upon. Figure 4.7 shows box plots of data obtained by running each pipeline one hundred times on a 2018 Intel-based Apple MacBook Pro laptop via JupyterLab, first using a 5-second long input audio file, and then a 20-second file. These data do not differ significantly when run with the custom acid timbre model made for this project versus Google's stock saxophone model. For the 5-second input file, the median processing time for the modified raivBox pipeline across both models is approximately 2.16 seconds, whereas for the original Google Magenta pipeline, the median is 17.96 seconds. In this instance, the raivBox modification offers a processing speed increase of over 800%. The median processing times for the 20-second input are 6.55 seconds for the raivBox version, and 69.37 seconds for the Magenta original, indicating an average latency reduction of over 10x when using the raivBox pipeline. The results from six different 10-second input files are visualized in the Appendix.

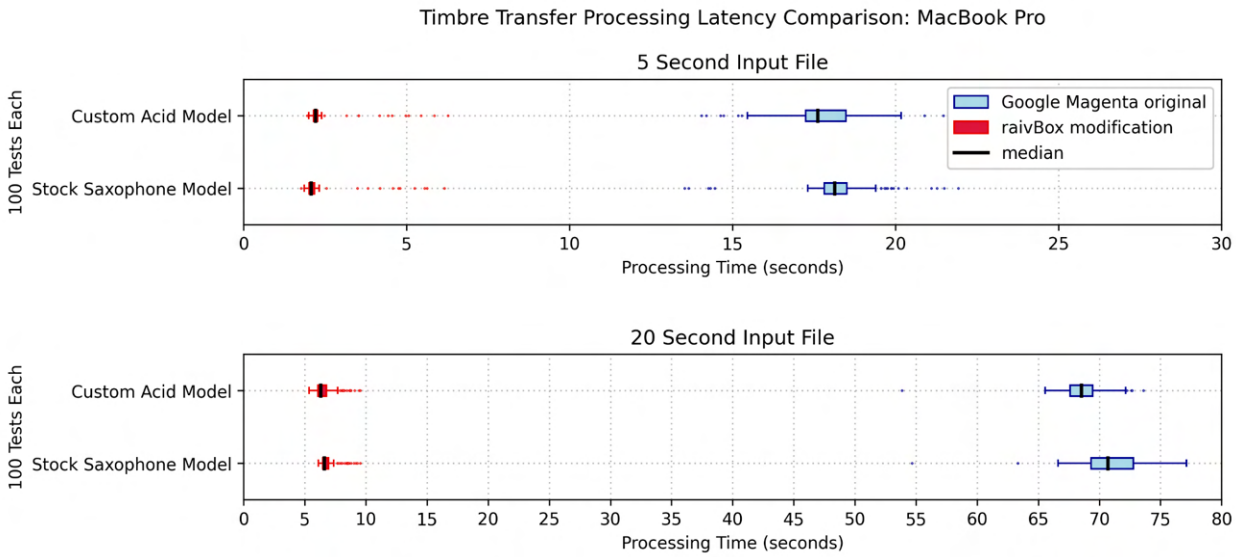


Figure 4.7: Box plots comparing the processing latency of the original Google Magenta DDSP pipeline and the modified raivBox pipeline on a MacBook Pro, using both the custom acid timbre model and the stock saxophone timbre model (additional latency comparisons in the Appendix)

After confirming the performance gains in the MacBook Pro test environment, various iterations of the two pipelines are implemented for analysis on the Nvidia Jetson Nano 2GB inside the raivBox itself. First, the Google Magenta pipeline extracted from the timbre transfer demonstration notebook is run, with minor modifications made to ensure basic compatibility with the Jetson. This pipeline defaults to the 'full' CREPE model capacity for feature extraction, which is relatively computationally intensive. In the second iteration, CREPE is manually instructed to use its least expensive model capacity, 'tiny'. This change yields a massive performance improvement. The third iteration swaps the usage of CREPE altogether for the relatively lightweight YIN pitch estimator. Switching to YIN offers another marked latency decrease. Finally, the original Checkpoint model load-in process from Google Magenta's timbre transfer demonstration is replaced with the lightweight raivBox SavedModel implementation, and the time-consuming model-building dry run step is removed. These final modifications to the original pipeline provide a considerable latency improvement as well, resulting in a combined total median processing time reduction of over 1000% in the final raivBox version when compared with the original version (see Figure 4.8). Supplementary box plots for these data are displayed in the Appendix to highlight outliers and spread.

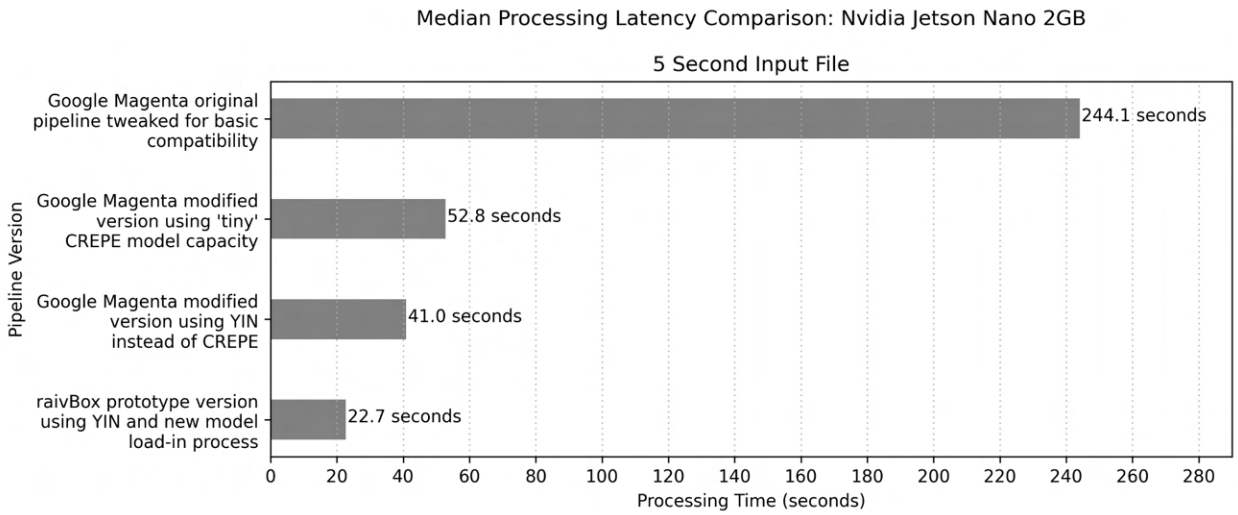


Figure 4.8: Bar chart comparing the median processing latency of various timbre transfer pipeline versions on the Nvidia Jetson Nano 2GB within the raivBox

Switching from CREPE to YIN for feature extraction poses two main hurdles for the raivBox, the first being the quality of the pitch estimation data. YIN is much more susceptible to degraded accuracy when background noise is present in the input signal, leading to garbled output audio as the pitch data fluctuate through extremes. The second challenge is that YIN does not natively produce estimation confidence measures; however, this complication provides an opportunity to further reduce the processing load and also introduce a differentiating element to the raivBox timbre transfer pipeline. In the version of timbre transfer demonstrated by Google Magenta, output audio sections with low confidence ratings from CREPE are cleverly filtered or attenuated to mask potential voicing issues. Instead of calculating or estimating pitch prediction confidence for the input audio, the raivBox automatically proceeds with full confidence for every frequency data point estimated by YIN, producing full-voiced and brazen-sounding audio outputs for even the most discrepant feature data. This design decision imbues raivBox timbre transfer with a unique sonic style and character that helps differentiate it from the original Google implementation it is built upon. Additionally, forgoing all confidence calculation saves computational resources for the inference step of the timbre conversion process.

As mentioned in the previous section, the size-reduced timbre models utilized by the raivBox also produce a subtle alteration to the audio outputs when compared with results obtained using the full Checkpoint models, but this change is by no means an objective deterioration. If anything, the difference is merely another nominal mutation in sonic character,

contributing to the uniqueness of the raivBox pipeline. The log-frequency power spectrograms in Figure 4.9 compare the performance of the raivBox pipeline with Google Magenta's original pipeline on various input signals. Input File 1 is a sine wave melody created in Ableton Live, and Input File 6 is a recording of the author whistling. All of the audio files used as inputs for these tests are available in the public project GitHub repository[31].

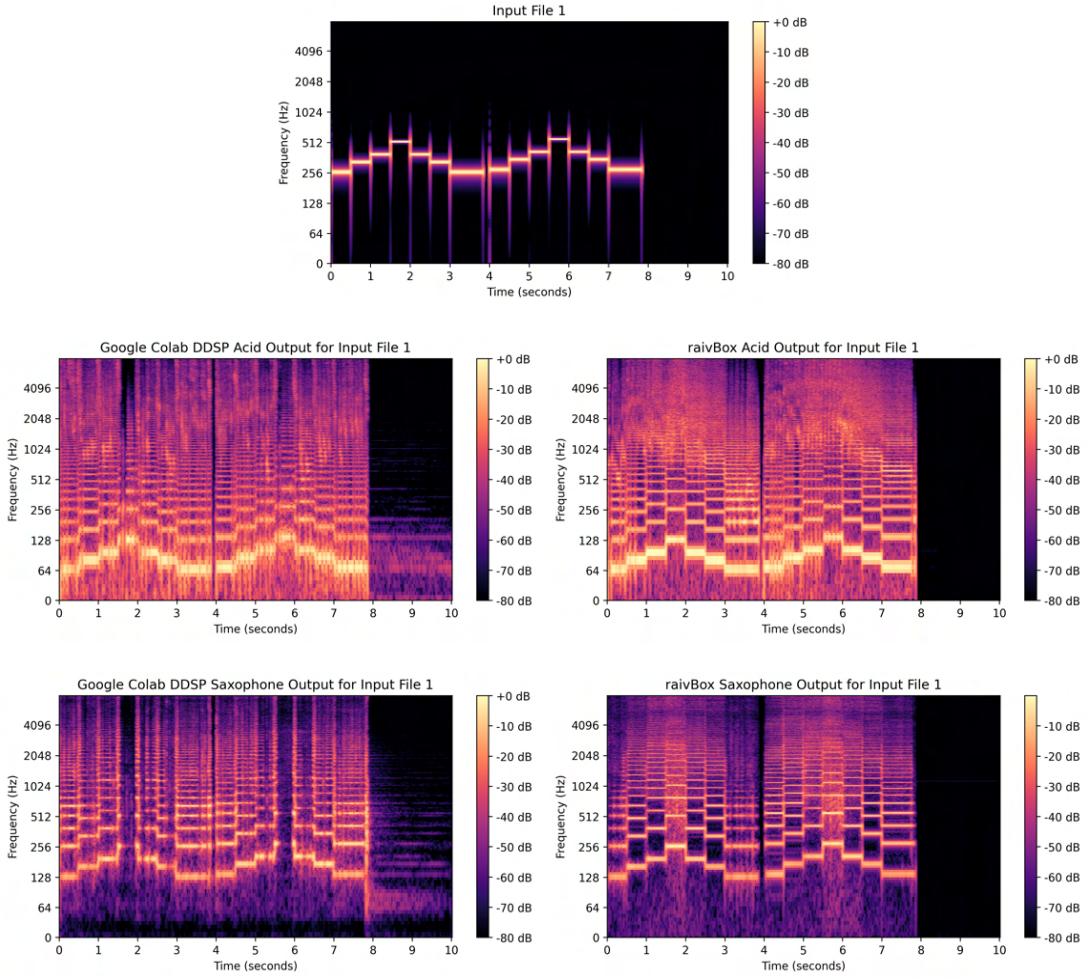


Figure 4.9: Selection of log-frequency power spectrograms of audio signals before and after timbre conversion, comparing the original Google Magenta DDSP pipeline to the modified raivBox pipeline (additional signal comparisons in the Appendix)

[31] https://github.com/jacktipper/raivBox

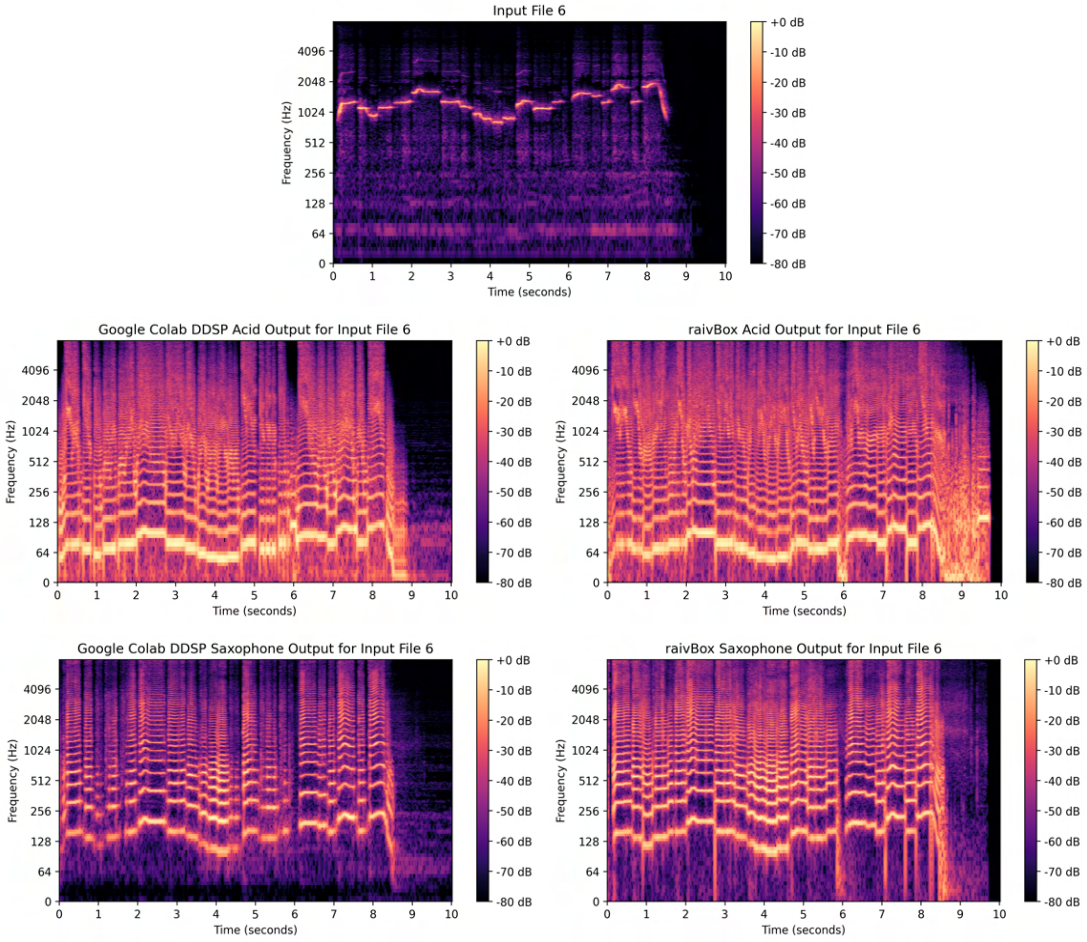


Figure 4.9 (continued): Selection of log-frequency power spectrograms of audio signals before and after timbre conversion, comparing the original Google Magenta DDSP pipeline to the modified raivBox pipeline (additional signal comparisons in the Appendix)

In the experiments with digitally-generated inputs, the raivBox pipeline does not produce a phantom release tail, whereas the original Google Magenta pipeline does. During full-voiced moments, the raivBox outputs seem drier and more direct, with the Magenta outputs on the other hand sounding like they have been run through a short-tailed reverb. The input signals with segments of low-level noise often result in chaotic outputs via the raivBox pipeline, whereas Google's original introduces a spectral filtering effect that suppresses these low-confidence moments. The author finds that the outputs from the original pipeline sound relatively damp and diffident. Even though it is more likely to generate erratic sounds, the sonic traits produced by the raivBox pipeline exude a buoyancy and vibrance that add to the device's creative charm.

At present, the raivBox runs timbre transfer inference using its central processing unit (CPU), not the on-board GPU. This is due to the fact that in the versions of TensorFlow[32] available for the Jetson Nano at the time of writing, when TensorFlow is instructed to use the GPU, it automatically pre-allocates all of the available GPU memory on the system for the task at hand. The Nvidia Jetson Nano shares RAM between the CPU and GPU, so this operation rapidly and unnecessarily depletes a critical resource, resulting in crashes and out-of-memory errors. Even when experimenting with memory growth limitation, the RAM reserved for GPU inference is not properly released until the entire TensorFlow instance is terminated, further complicating the objective of maximizing performance. In contrast to this, when running inference on the CPU, TensorFlow dynamically allocates and releases memory. CPU inference also has access to swap memory, which is space on the SD card that can be used if RAM overflow is needed. For this case, dynamic allocation and access to swap memory vastly increase system stability, helping to guard against memory-related crashes. That said, as previously detailed in the Literature Review section, relying on the CPU for this deep learning task is not an ideal use of hardware resources. In future versions of the raivBox, achieving full GPU compatibility for timbre transfer inference is a key priority.

4.3 Qualitative Evaluation (N=20)

Demonstrating the fitness of the raivBox as a prototype for a consumer-oriented musical audio creation tool is a core goal for this project, making the stage of gathering feedback from a sample of potential end-users especially valuable to the forthcoming discussion and conclusions. For this reason, participants with a basic working knowledge of music hardware devices were sought for this study by soliciting volunteers from within the Music Technology program at New York University's Steinhardt School.

4.3.1 Procedure

After proposing and clearing the user study action plan with the Institutional Review Board (IRB) at New York University, music technology students were invited to participate in

[32] https://www.tensorflow.org/guide/gpu

the qualitative evaluation of the raivBox prototype. Twenty students stepped forward to appraise the raivBox during the study period at the end of March 2022, and all of the evaluations were submitted before the beginning of April 2022. To preserve the privacy of the participants in accordance with IRB policy, all collected data have been aggregated and thoroughly anonymized. Drawing correlations between responses and any specific participant subgroups is therefore impossible for this study.

Once agreement to the participation consent form has been established, each volunteer was provided with the raivBox prototype for as long as they wished to evaluate it. Evaluation sessions took place in common areas of the Steinhardt Education Building on the New York University campus, and, per the discretion of the participants, ranged in length from less than 10 minutes to more than 2 hours (see Figure 4.10). A large majority of participants chose a session length between 10 minutes and 30 minutes. Each session was conducted as follows:

1. The study administrator briefly demonstrates the basic controls and functionalities of the raivBox for the participant.

2. The participant is given the device to explore and evaluate as they see fit.

3. The participant is briefly interviewed about their experience with the raivBox, with the administrator noting key themes and/or takeaways.

4. When the participant is satisfied with their evaluation, the administrator provides a link to an anonymous Google survey form to collect the evaluation.

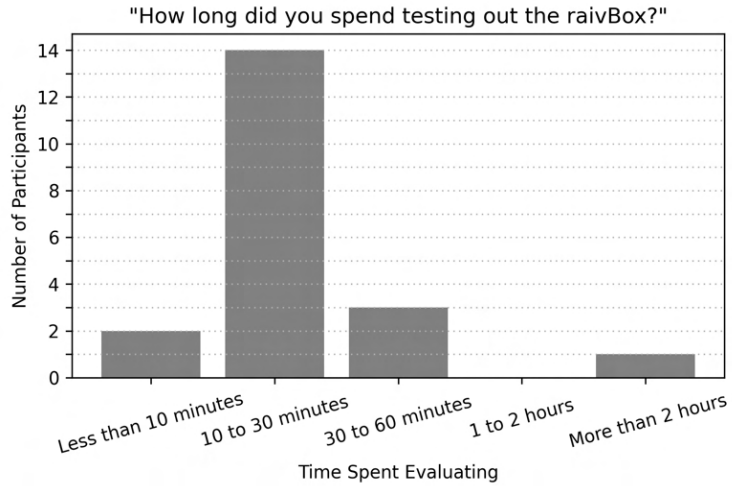


Figure 4.10. Histogram of participants' time spent evaluating the raivBox

4.3.2 Demographics

The participant birth years reported in the survey data range from 1986 to 2002, with a 40% plurality of participants born between 1994 and 1998 (see Figure 4.11). Among the participants, 35% indicated that they have 2 to 6 years of experience as an audio, sound, and/or music professional, with 55% claiming more than 6 years of experience and only 10% claiming less than 2 years of experience (see Figure 4.12).

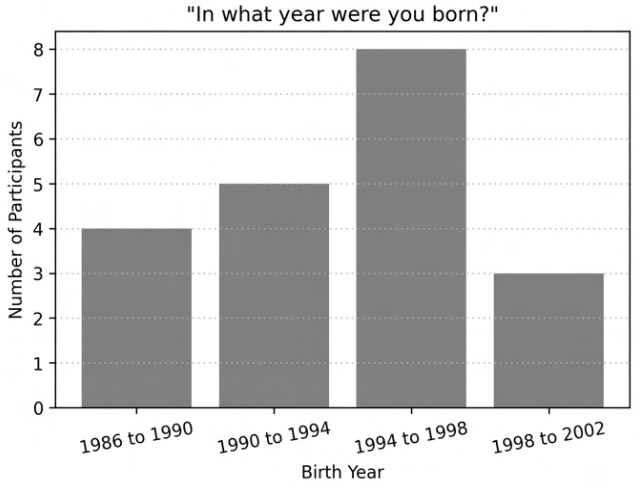


Figure 4.11: Histogram of participants' reported birth years

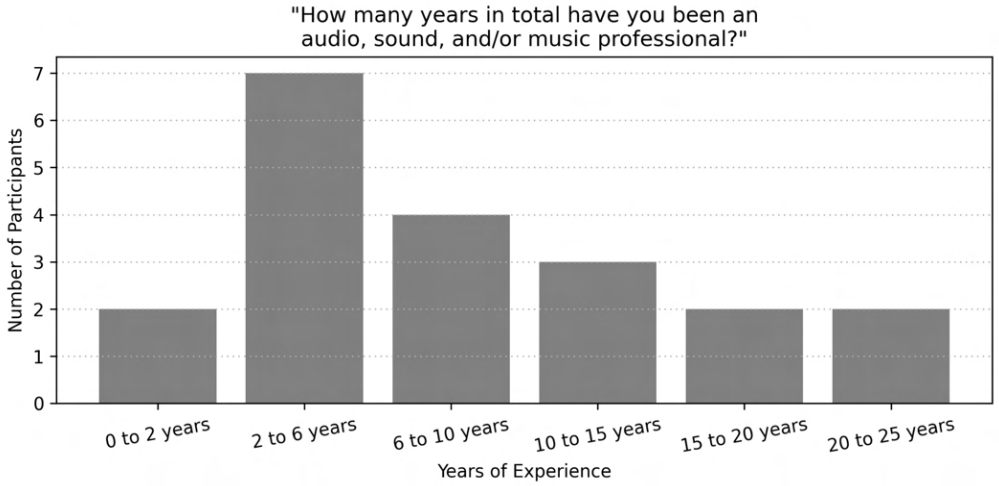


Figure 4.12: Histogram of participants' reported professional experience with audio, sound, and/or music

As shown in Figure 4.13, the participants have a broad diversity of experiences with various audio hardware devices. Based on the data, most of the participants own and/or regularly use MIDI controllers, drum machines, guitar/synthesizer pedals, and audio mixers/interfaces/DJ gear. The only hardware category with a majority of study participants reporting use or ownership of one or fewer devices is *rack modules*, which includes equipment like professional analog studio tools and eurorack synthesizer components. A minority of participants indicated that they have extensive experience with certain device families, including two outlier data points: one claiming the use or ownership of more than 20 rack modules, and the other reporting over 20 MIDI controllers. Almost all participants confirmed experience with more than one MIDI controller (95%), and more than one audio mixer/interface/DJing device (90%). These findings support the hypothesis that this sample of users is likely to be representative of people with basic working knowledge of music hardware devices.

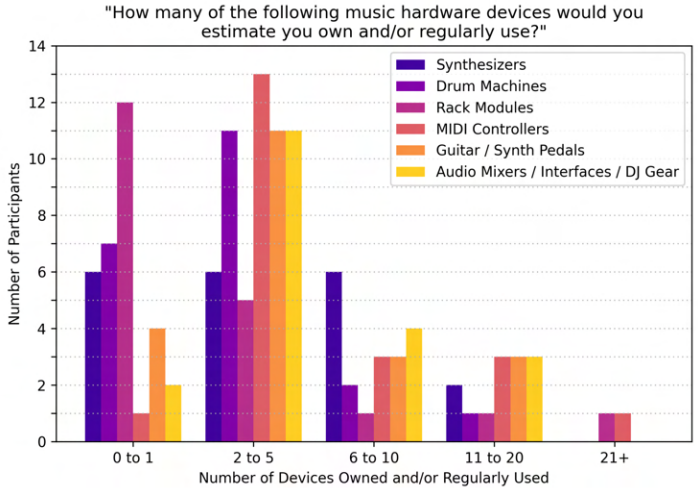


Figure 4.13: Histogram of participants' experience with various music hardware devices

4.3.3 Results

In the most substantial portion of the raivBox evaluation questionnaire, participants were asked to rate their level of agreement with a variety of statements concerning the raivBox on the Likert scale. The data collected on these statements are exhibited herein using box plots to facilitate analysis of the underlying themes and trends.

Figure 4.14 presents the ranges of responses to statements regarding the sonic output of the raivBox. As we can see, an unequivocal majority of participants expressed strong agreement with the statement “The raivBox produces interesting sounds”, with simple agreement being the outlier and no other data points represented. This result indicates a unified sentiment among the respondents, which can be extrapolated to similar populations.

“The raivBox produces high-quality sounds” was a more controversial statement, with a median response of “Agree” and a mean falling between “Agree” and “Neutral”. The interquartile range (IQR) for this raivBox sound quality appraisal spans between “Neutral” and “Agree”, with the lower quartile from “Disagree” to “Neutral” and the upper quartile from “Agree” to “Strongly Agree”. Although these results are by no means scathing, they do suggest that additional research and attention toward improving the quality of the sounds produced by the raivBox should be an area targeted for future work.

The two following statements concerning music creation produced interesting data, as the responses appear to embody a subtle dissonance. “The raivBox is useful for music creation” was met with a broader range of reactions including an outlier “Disagree”; however, the bulk of responses are concentrated from “Agree” to “Strongly Agree”. The median response is split between agreement and strong agreement as well, with the mean falling slightly below the median within the same range. In contrast, “I would use the raivBox in my music creation workflow” has a tighter range of responses from “Neutral” to “Strongly Agree”, and a median below slightly the mean, right around the “Agree” marker. At least one participant disagreed that the raivBox is useful for music creation, but was at the very least neutral about whether they themselves would use it in their own creative workflow, perhaps alluding to the presence of a personal interest in the device that outweighs their perception of its usefulness. On the other hand, multiple respondents felt a stronger agreement for the general usefulness of the raivBox as a tool for music creation, but were comparatively hesitant to claim that they would use it in their own workflows.

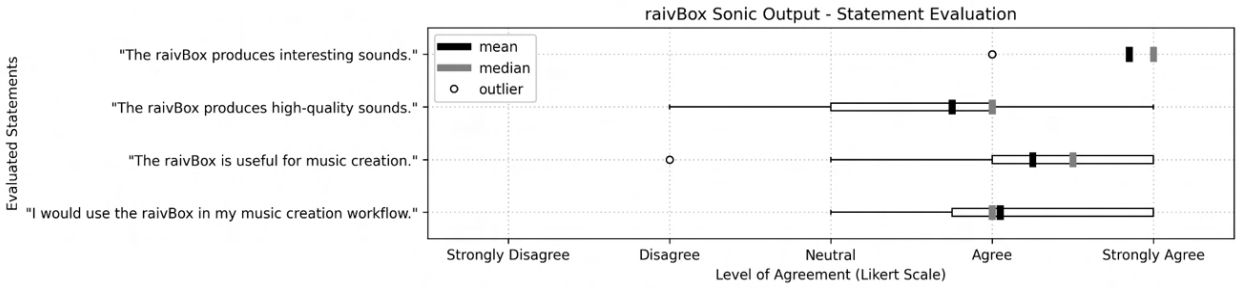


Figure 4.14: Box plots of statement evaluation data concerning the sonic output of the raivBox

The next set of box plots in Figure 4.15 examine the participants' perception of the raivBox user interface. As shown below, these statements elicited a variety of responses and produced valuable trends for analysis.

First, the statement "The raivBox feels like a professional tool" received mixed feedback, with "Agree" as both the median and the top of the IRQ, but "Neutral" and "Disagree" dragging down the mean, free from outliers. This implies that the current version of the raivBox does not sufficiently inspire confidence in its veracity as a professional audio device, and additional work remains to achieve this perception.

The next statement proposes that "The design of the raivBox is visually appealing", earning positive ratings from the respondents. A strong majority of participants selected "Strongly Agree" for this statement, and "Agree" earned second place, as evidenced by the distribution of the box plot and the central tendency indicators. "Neutral" also made a noteworthy showing at the bottom of the lower quartile, above the disagreeing outlier. From these data, it can be inferred that the visual appearance of the device is not generally impeding its acceptance by end-users.

The data for the following four statements form a cohesive narrative—that the raivBox user interface is intuitive and by no means confusing. That said, the user experience would foreseeably improve with the availability of additional control parameters on the faceplate. These results were the trend, but not unanimous. For the first evaluated statement, "The user interface of the raivBox is confusing", there were two outliers, one of whom agreed, and another who strongly agreed. Beyond these extraneous data points, the majority of participants still selected "Strongly Disagree", with the IQR spanning only to "Disagree" and the outlier-excluded maximum at "Neutral". "The raivBox has too many controls." had a similar response

concentration but without the agreeing outliers, suggesting a tighter range of participant sentiment. Most respondents agree that “The controls on the raivBox are intuitive”, but with a less overwhelming trend when compared with the previous three statements. It appears that most users found the controls intuitive, but not impressively so, signifying that additional attention to improving this domain may be valuable. To compound this conclusion, “The raivBox has enough controls to achieve the sound I want” received mostly neutral feedback, with only a slight trend toward agreement. This prompts the deduction that adding hardware control for a wider variety of sound-shaping parameters would likely benefit the overall user experience of the raivBox.

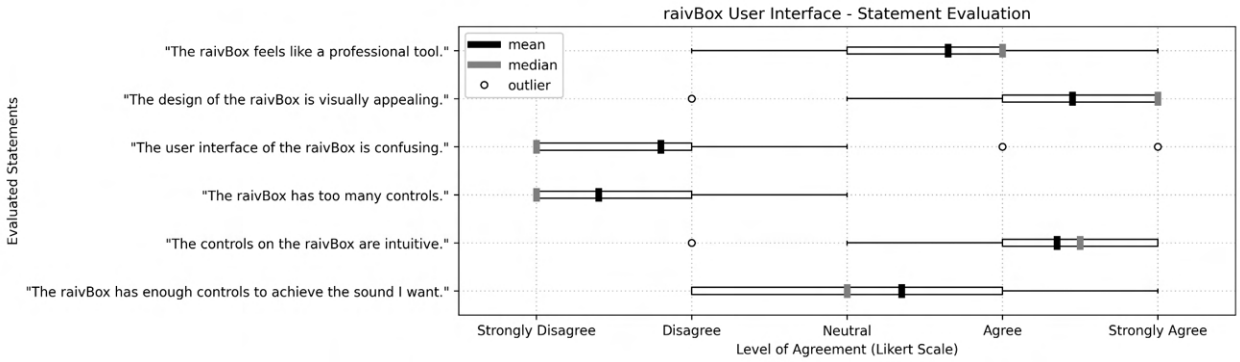


Figure 4.15: Box plots of statement evaluation data concerning the user interface of the raivBox

Lastly, the final collection of statements explore the participants’ direct experience using the raivBox (see Figure 4.16). To start with the most positive data, respondents unanimously indicated that they had a fun experience testing out the raivBox prototype! “Strongly Agree” took such an overwhelming majority of the votes that simple agreement with the statement “The raivBox is fun to use” is an outlier data point.

“The raivBox is comfortable and ergonomic to use” saw a wider array of reactions than its preceding statement, with responses tending toward “Agree”. Disagreement is an outlier for this statement, but the spread from “Strongly Agree” to “Neutral” begets the supposition that the ergonomics of the raivBox could benefit from a round of improvements in a later version.

None of the participants agreed that “The raivBox is too complicated”, though neutrality on this statement was not an outlier data point. Similarly to the previous statement, this range suggests that there may be room for refinement in this area, but it does not particularly stand out as a priority based on the data.

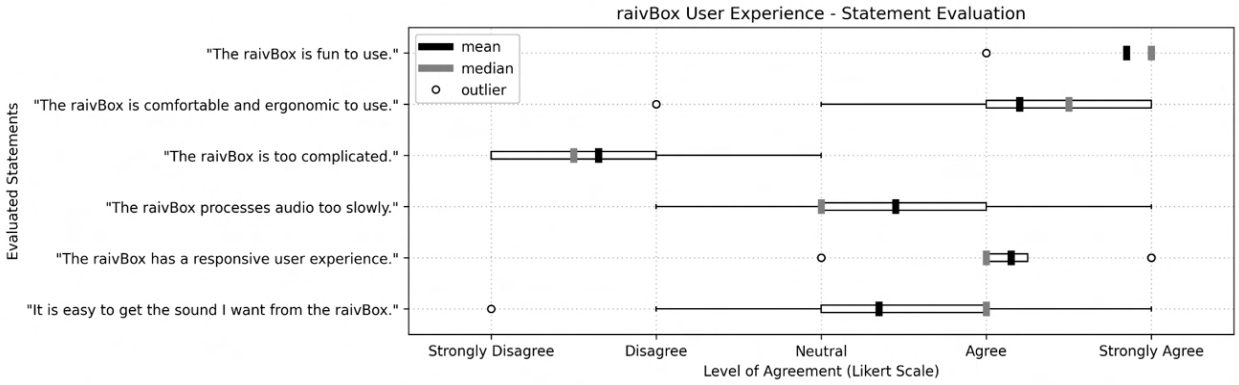


Figure 4.16: Box plots of statement evaluation data concerning the user experience of the raivBox

The following statement that “The raivBox processes audio too slowly” on the other hand is a major indicator of where future work and research resources should be allocated. The median response was neutrality, but this critical statement earned a mean value trending toward agreement, and “Strongly Agree” is not even an outlier. Not a single respondent chose “Strongly Disagree”; however, the data report that some charitable participants did disagree with the statement. This shows that improving processing latency on the raivBox should be a core priority going forward. Be that as it may, the next statement offers a valuable comparison, as the vast majority of respondents agreed that “The raivBox has a responsive user experience”. This statement is markedly more abstract than its predecessor, describing the perception of latency when operating the raivBox. It appears that although the processing time for the neural synthesizer was found to be less than adequate, the experience of interacting with the controls interface and visual feedback mechanisms of the device did feel responsive for users. Maintaining and advancing this aspect of the user experience while enhancing the audio processing speed is therefore an ultimate goal for future versions of the raivBox.

Finally, one of the more controversial statements in the questionnaire: “It is easy to get the sound I want from the raivBox”. This statement earned responses across the full range of the Likert scale, with a tendency toward “Agree”. In the brief interviews with participant volunteers, a theme emerged: participants who are actively interested in exploring novel and avant-garde approaches to sound design were especially drawn to the audio outputs of the raivBox and were also much more observably satisfied with unconventional or unexpected sonic results, whereas

participants who tended toward traditional or classic approaches to music and/or sound design saw less potential opportunity in the raivBox's sounds. To give an example, multiple participants said that they would prefer to just use a saxophone or a trumpet to make its own sound over converting another sound into its timbre, but that the converted sounds are interesting and perhaps valuable for their own unique sonic characteristics. In direct contrast to this, multiple other participants likened the raivBox outputs on dialogue input signals from a noisy test environment to various science-fiction creature voices from their experiences in the film world. Many of the study volunteers expressed a desire for a wider variety of synthesized or otherwise unusual timbre models to facilitate broader exploration of the range of possibilities. This trend paints another clear target for future work: to train a variety of new and novel neural timbre models for the raivBox. Additionally, a large number of the test users requested additional DSP and effects capabilities, such as reverb and delay. Although there is the possibility of contributing DSP code to the *synth.py* file directly, this is not directly accessible via the hardware control interface, and should be considered for subsequent iterations of the device.

At the end of the questionnaire, users were asked to assess their perception of the monetary value of the current raivBox prototype version (Figure 4.17):

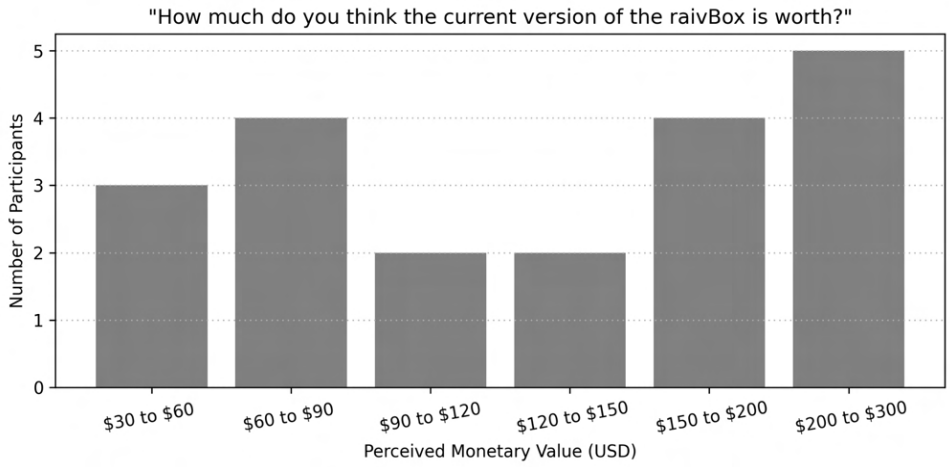


Figure 4.17: Histogram of participants' perception of the monetary value of the raivBox

This question prompted interesting results, with a large concentration of responses between $150 and $300, as well as a slightly smaller grouping between $30 and $90. One quarter of all participants reported that they felt the raivBox to be worth between $200 and $300. None of the

respondents indicated a perception of the raivBox being worth less than $30 or more than $300, and only one fifth of the responses chose the middle ranges from $90 to $150. The author hypothesizes that this dual-peak division of density between lower and upper value ranges suggests an experiential disconnect between users who found the raivBox prototype to feel like a professional tool and those who felt it to be more of a toy device. Having “the current version of the raivBox” specified in the statement could also imply a sense of room for growth, since the phraseology anchors the user sentiments to the prototype they personally experienced rather than a speculation about a future iteration.

5. DISCUSSION AND CONCLUSIONS

Neural waveform synthesis in the field of music is still bleeding edge technology despite the monumental advances in recent years. Building on the pioneering work by the Google Magenta team among others, the raivBox prototype introduced herein offers a novel entry point for those interested in this new sound design and music creation paradigm. This thesis was motivated by the goal of eliminating barriers to the use and adoption of neural audio synthesis through the development of a dedicated physical product. By shifting the neural synthesis user experience away from laptop/desktop computing environments and into a simple, mobile, fun, and accessible music hardware device, the raivBox can help to elevate the underlying technology, ushering in a new era of sonic exploration.

To ensure the feasibility of the core objectives, the execution of this project began with background research and preliminary testing of neural audio synthesis techniques and embedded device prototyping. After confirming viability and forming a course of action, the necessary hardware and software components were gathered and assembled into a unified product. Custom parts and modules were designed for key developmental steps, including the 3D-printed housing, the PCB, and the core software architecture. The neural synthesis pipeline underwent tailoring to optimize its performance, replacing and removing performance bottlenecks where possible to minimize the computational load on the host machine. Once the device had reached full hardware and software functionality, the refinement process commenced to iron out user experience deficiencies. Finally, with the working prototype of the raivBox complete, it was thoroughly tested to gather performance metrics and provided to users for qualitative evaluation.

Analysis of the raivBox performance data show major improvements to processing latency as well as model size on disk when compared with its predecessor. The tailored raivBox neural synthesis pipeline consistently outperforms the original Google Magenta pipeline in terms of processing speed by over 800%, and runs using trained timbre model files less than half the size of the original. The model size reduction results in subtly altered sonic characteristics in the synthesizer audio outputs. That said, spectral analysis and listening shows that these differences are a matter of individual preference rather than objective quality. Additionally, should the user wish, the model load-in process can be fully reverted to the large files in order to restore the original character, without sacrificing the processing latency gains.

As demonstrated in the qualitative study, most users find the raivBox to be fun, intuitive, and useful for music creation, and are also interested in incorporating it into their own artistic workflows. These responses evidence the device's high degree of perceived approachability—a key success of this research. At present, the raivBox source code is available in the project GitHub repository alongside the digital assets and other requirements for constructing the device.

One important takeaway from the user evaluation process is that the raivBox shines as a relatively unpredictable sound design device; however, it is much less suited to reliably generating convincing recreations of well-known acoustic musical timbres. The qualitative user interviews and questionnaire responses showed that many users find the raivBox outputs to feel organic and expressive in a unique way, and suggest that it is exactly this whimsical volatility that makes the raivBox a valuable creative tool. Many existing electronic music devices are capable of producing familiar instrument timbres. Some, such as the microKORG[33] for example, even have vocoder functionality reminiscent of the raivBox's direct microphone input. What sets the raivBox apart is the idiosyncratic sonic characteristics imbued by the timbre models and neural synthesis process, as well as the inventive artistic decisions these results can inspire.

Multiple participants in the qualitative assessment of the raivBox noted that the results it produces when run on dialogue and in noisy environments often bears resemblance to creature voices from science-fiction films, across a variety of timbre models. The experimental sound design potential necessary to stir this response from several isolated individuals during the user study interview stage was palpable. Film sound applications aside, the lively outputs obtained from the raivBox foreshadow innovative implementations in electronic music as well, as a tool for generating playful sonic elements across the spectrum of the avant-garde.

The raivBox prototype presented in this thesis only has a single channel for neural synthesis and output playback, but this starting point constitutes a ripe foundation for application in a multi-voiced device, without even needing to implement sophisticated parallelism or port the core audio software to a lower-level coding language. A subsequent version of the raivBox with additional play buttons could address this expansion by simply generating multiple outputs successively, each rendered clip triggered for playback by a dedicated button. The interface for play button assignment may be executed in a variety of ways, such as pressing a record button and the target play button simultaneously. Having an array of output trigger buttons would enable

[33] https://www.korg.com/us/products/synthesizers/microkorg

improved musical performance functionality similar to a hardware looper or sampler, but with the addition of the neural synthesis pipeline. If combined with conventional audio effects and performance tools such as reverb, delay and sequencing, this trajectory could assist the raivBox in evolving to surpass its current sonic flexibility limitations.

Creating a true real-time version of the raivBox is another exciting goal for future work. At present, the development of real-time timbre transfer pipelines is beginning to garner modest attention in the field. As these techniques continue to develop and improve, low-resource implementations will be a valuable contribution to broaden the scope of the advances. A real-time raivBox could function without the main record and play buttons, making it akin to a guitar or synthesizer pedal in practice.

Apart from its original intended use as a neural synthesizer, the open-source core software architecture of the raivBox also shows promise as a DSP sandbox for hobbyists or educators. For instance, in a future version of the raivBox software, the model selection framework could be repurposed to iterate through different blocks of DSP code via a switch statement in *synth.py*. The processes inside the synthesis loop are currently designed to be easily augmented or even replaced with new audio processing commands, while avoiding the compromise of the hardware control interface.

As argued in this thesis, the raivBox and its underlying technology show vast potential for curious creativity and progress in the field. Grasping the sheer breadth of possibilities is beyond the scope of this work; the author hopes readers will join in going forth to explore.

REFERENCES

Alpaydin, E. (2014). *Introduction to Machine Learning (3rd Edition)* [NYU ProQuest Ebook Version]. Retrieved from https://ebookcentral-proquest-com.proxy.library.nyu.edu/

Babacan, O., Drugman, T., d'Alessandro, N., Henrich, N., & Dutoit, T. (2013, May). A comparative study of pitch extraction algorithms on a large variety of singing sounds. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 7815-7819. IEEE. https://doi.org/10.1109/icassp.2013.6639185

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2010). Cython: The best of both worlds. *Computing in Science & Engineering, 13*(2), 31-39. https://doi.org/10.1109/MCSE.2010.118

Bechtel, M. G., McEllhiney, E., Kim, M., & Yun, H. (2018, August). DeepPicar: A Low-Cost Deep Neural Network-Based Autonomous Car. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* 11-21. https://doi.org/10.1109/RTCSA.2018.00011

Betker, M. R., Fernando, J. S., & Whalen, S. P. (1997). The history of the microprocessor. *Bell Labs Technical Journal, 2*(4), 29-56. https://doi.org/10.1002/bltj.2082

Bhattarai, M., Jensen-Curtis, A. R., & Martínez-Ramón, M. (2020). An embedded deep learning system for augmented reality in firefighting applications. *arXiv preprint arXiv:2009.10679*. https://arxiv.org/abs/2009.10679

Both, D. (2020). systemd. In: *Using and Administering Linux: Volume 2*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-5455-4_13

Briot, J. P., Hadjeres, G., & Pachet, F. D. (2017). Deep Learning Techniques for Music Generation—A Survey. *arXiv preprint arXiv:1709.01620*. https://arxiv.org/abs/1709.01620

Briot, J. P., & Pachet, F. (2018). Deep learning for music generation: challenges and directions. *Neural Computing and Applications, 32*(4), 981–993. https://doi.org/10.1007/s00521-018-3813-6

Cardoso, J. M. P., de Figueiredo Coutinho, J. G., & Diniz, P. C. (2017). *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation*. Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-804189-5.00001-6

Cass, S. (2020). Nvidia makes it easy to embed AI: The Jetson nano packs a lot of machine-learning power into DIY projects [Hands On]. *IEEE Spectrum, 57*(7), 14-16. https://doi.org/10.1109/MSPEC.2020.9126102

Daghero, F., Pagliari, D. J., & Poncino, M. (2021). Energy-efficient deep learning inference on edge devices. *Advances in Computers*. https://doi.org/10.1016/bs.adcom.2020.07.002

De Cheveigné, A., & Kawahara, H. (2002). YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America, 111*(4), 1917-1930. https://doi.org/10.1121/1.1458024

De Pra, Y., Fontana, F., & Simonato, M. (2018, February). Development of real-time audio applications using Python. *Proceedings of the XXII Colloquium of Musical Informatics, Udine, Italy*, 22-23.

De Pra, Y., & Fontana, F. (2020). Programming Real-Time Sound in Python. *Applied Sciences, 10*(12), 4214. https://doi.org/10.3390/app10124214

Drakopoulos, F., Baby, D., & Verhulst, S. (2019, September). Real-time audio processing on a Raspberry Pi using deep neural networks. *Proceedings of the 23rd International Congress on Acoustics (ICA 2019)*, 2827-2834. Deutsche Gesellschaft für Akustik.

Engel, J., Hantrakul, L., Gu, C., & Roberts, A. (2020). DDSP: Differentiable Digital Signal Processing. *arXiv preprint arXiv:2001.04643*. https://arxiv.org/abs/2001.04643

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Norouzi, M., Eck, D., & Simonyan, K. (2017, July). Neural audio synthesis of musical notes with wavenet autoencoders. In *International Conference on Machine Learning, 70*, 1068-1077. PMLR.

Evci, U., Gale, T., Menick, J., Castro, P. S., & Elsen, E. (2020, November). Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, 2943-2952. PMLR. https://arxiv.org/abs/1911.11134

Franco, I., & Wanderley, M. M. (2015, May). Practical Evaluation of Synthesis Performance on the BeagleBone Black. *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME '15),* 223-226. https://dl.acm.org/doi/abs/10.5555/2993778.2993836

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* [NYU ProQuest Ebook Version]. Retrieved from https://ebookcentral-proquest-com.proxy.library.nyu.edu/

Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., Oord, A. V. D., Dieleman, S., & Kavukcuoglu, K. (2018). Efficient neural audio synthesis. arXiv preprint arXiv:1802.08435. https://arxiv.org/abs/1802.08435

Kilby, J. S. (2000, January). The integrated circuit's early history. *Proceedings of the IEEE, 88*(1), 109–111. https://doi.org/10.1109/5.811607

Kim, J. W., Salamon, J., Li, P., & Bello, J. P. (2018). CREPE: A convolutional representation for pitch estimation. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161-165. https://arxiv.org/abs/1802.06182

Kirk, D. (2007, October). NVIDIA CUDA software and GPU parallel computing architecture. *Proceedings of the 6th International Symposium on Memory Management (ISMM '07), 7,* 103-104. https://doi.org/10.1145/1296907.1296909

Kuo, S. M., Lee, B. H., & Tian, W. (2013). *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications* [NYU ProQuest Ebook Version]. Retrieved from https://ebookcentral-proquest-com.proxy.library.nyu.edu/

Lam, S. K., Pitrou, A., & Seibert, S. (2015, November). Numba: A llvm-based python jit compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1-6. https://doi.org/10.1145/2833157.2833162

Lane, N. D., Bhattacharya, S., Mathur, A., Georgiev, P., Forlivesi, C., & Kawsar, F. (2017). Squeezing Deep Learning into Mobile and Embedded Devices. *IEEE Pervasive Computing, 16*(3), 82-88. https://doi.org/10.1109/MPRV.2017.2940968

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521(7553),* 436-444. https://doi.org/10.1038/nature14539

Lee, S., Son, K., Kim, H., & Park, J. (2017, July 17-19). Car plate recognition based on CNN using embedded system with GPU. *2017 10th International Conference on Human System Interactions (HSI),* 239-241. https://doi.org/10.1109/HSI.2017.8005037

Marwedel, P. (2011). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems (2nd Edition).* Springer. https://doi.org/10.1007/978-94-007-0257-8

McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., & Nieto, O. (2015, July). librosa: Audio and Music Signal Analysis in Python. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015), 8,* 18-25.

Mehri, S., Kumar, K., Gulrajani, I., Kumar, R., Jain, S., Sotelo, J., Courville, A., & Bengio, Y. (2016). SampleRNN: An unconditional end-to-end neural audio generation model. *arXiv preprint arXiv:1612.07837*. https://arxiv.org/abs/1612.07837

Michon, R., Orlarey, Y., Letz, S., & Fober, D. (2019). Real Time Audio Digital Signal Processing With Faust and the Teensy. *Proceedings of the Sound and Music Computing Conference (SMC-19), Malaga, Spain*.

Mittal, S. (2019). A Survey on Optimized Implementation of Deep Learning Models on the Nvidia Jetson Platform. *Journal of Systems Architecture, 97,* 428-442. https://doi.org/10.1016/j.sysarc.2019.01.011

Oord, A. V. D., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., & Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*. https://arxiv.org/abs/1609.03499

Oord, A. V. D., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., Driessche, G. V. D., Lockhart, E., Cobo, L. C., Stimberg, F., Casagrande, N., Grewe, D., Noury, S., Dieleman, S., Elsen, E., Kalchbrenner, N., Zen, H., Graves, A., King, H., ... & Hassabis, D. (2018, July). Parallel WaveNet: Fast High-Fidelity Speech Synthesis. *Proceedings of the 35th International Conference on Machine Learning (PMLR 80)*, 3918-3926.

Prenger, R., Valle, R., & Catanzaro, B. (2019, May 12-17). WaveGlow: A Flow-based Generative Network for Speech Synthesis. In *2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 3617-3621. https://doi.org/10.1109/ICASSP.2019.8683143

Purwins, H., Li, B., Virtanen, T., Schlüter, J., Chang, S. Y., & Sainath, T. (2019). Deep Learning for Audio Signal Processing. *IEEE Journal of Selected Topics in Signal Processing, 13*(2), 206-219. https://doi.org/10.1109/JSTSP.2019.2908700

Richards, J. (2013). Beyond DIY in electronic music. *Organised Sound, 18*(3), 274-281. https://doi.org/10.1017/S1355771813000241 12

Richardson, M., & Wallace, S. (2016). *Getting Started with Raspberry Pi (3rd Edition): Getting to Know the Inexpensive ARM-powered Linux Computer*. Maker Media.

Rumsey, F. (2020). Modular Synths and Embedded Computing. *Journal of the Audio Engineering Society, 68*(3), 234-237. https://www.aes.org/e-lib/browse.cfm?elib=20730

Russell, S. J., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach (2nd Edition)* [Hathi Trust Digital Library Version]. Retrieved from https://catalog.hathitrust.org/Record/004917484

Salih, T. A., & Basman Gh., M. (2020, July 15-16). A Novel Face Recognition System Based on Jetson Nano Developer Kit. *IOP Conference Series: Materials Science and Engineering, 928,* 032051. https://doi.org/10.1088/1757-899x/928/3/032051

Srinivasan, V., Meudt, S., & Schwenker, F. (2019). Deep Learning Algorithms for Emotion Recognition on Low Power Single Board Computers. In *Lecture Notes in Computer Science,* 59–70. Springer. https://doi.org/10.1007/978-3-030-20984-1_6

Surges, G. (2012). DIY Hybrid Analog/Digital Modular Synthesis. *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME '12).*

Süzen, A. A., Duman, B., & Şen, B. (2020, June 26-28). Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry Pi using Deep-CNN. *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 1-5. https://doi.org/10.1109/HORA49412.2020.9152915

Teich, J. (2012, March 22). Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE, 100 (Special Centennial Issue),* 1411-1430. https://doi.org/10.1109/JPROC.2011.2182009

Vaseghi, S. V. (2008). *Advanced Digital Signal Processing and Noise Reduction*. John Wiley & Sons. https://doi.org/10.1002/9780470740156

Wang, X., Takaki, S., & Yamagishi, J. (2019). Neural Source-Filter Waveform Models for Statistical Parametric Speech Synthesis. *IEEE/ACM Transactions on Audio, Speech, and Language Processing, 28,* 402-415. https://doi.org/10.1109/TASLP.2019.2956145

Wickert, M. (2018, July). Real-Time Digital Signal Processing using pyaudio_helper and the ipywidgets. *Proceedings of the 17th Python in Science Conference, Austin, TX, USA*, 9-15.

Wolf, W. H. (1994, July). Hardware-software co-design of embedded systems. *Proceedings of the IEEE, 82*(7), 967-989. https://doi.org/10.1109/5.293155

Yamamoto, R., Song, E., & Kim, J. M. (2020, May 14). Parallel WaveGAN: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 6199-6203. https://doi.org/10.1109/ICASSP40776.2020.9053795

Zhao, Y., Wang, X., Juvela, L., & Yamagishi, J. (2020, May 14). Transferring neural speech waveform synthesizers to musical instrument sounds generation. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 6269-6273. https://doi.org/10.1109/ICASSP40776.2020.9053047

Zyskowski, C., & de Oliveira, M. (2017, May 11). An Analog Audio Sensor Board for Microcontrollers. *Audio Engineering Society 142nd Convention,* Engineering Brief 353. https://www.aes.org/e-lib/browse.cfm?elib=18728

APPENDIX

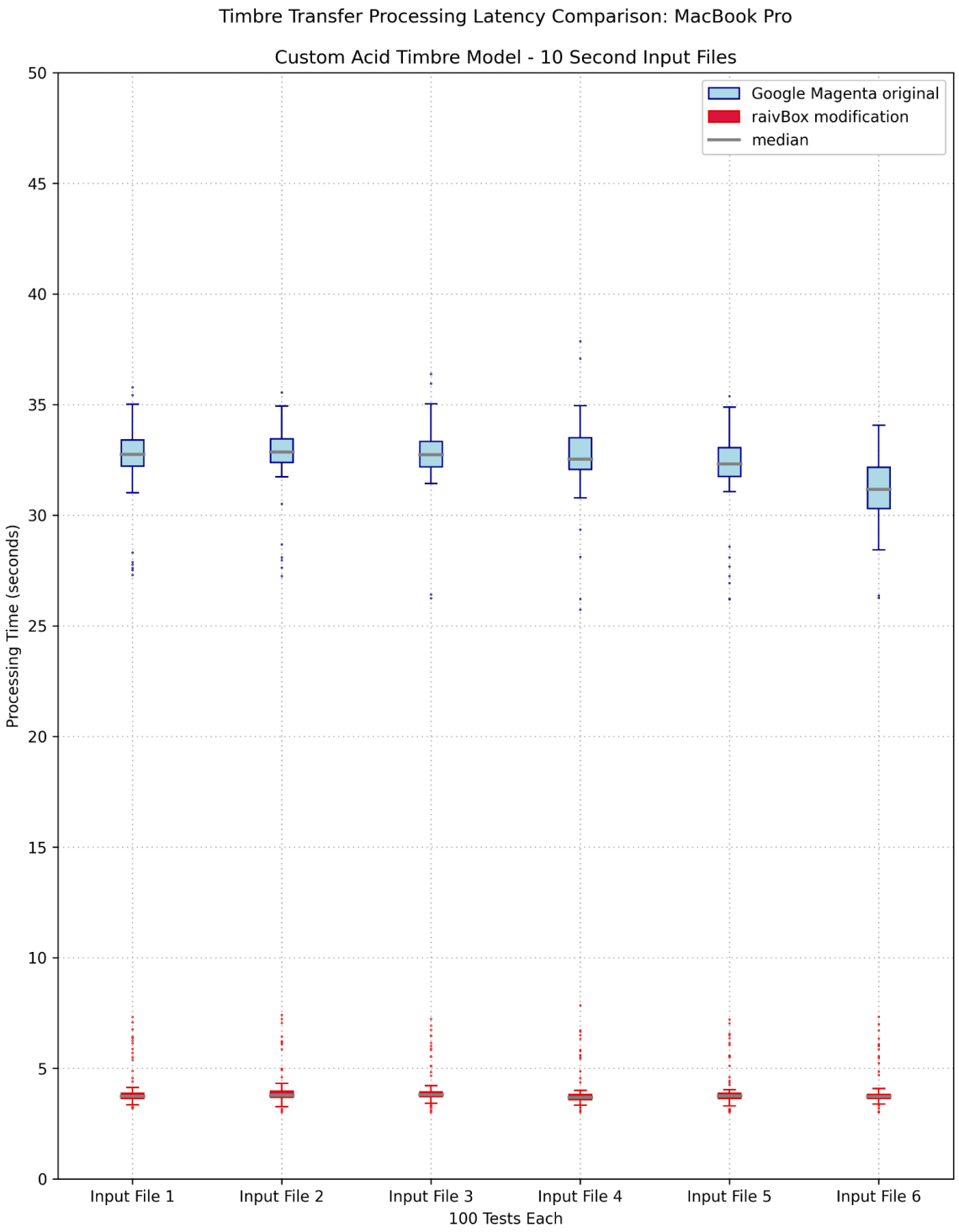


Figure A.1: Acid model processing latency comparisons on a MacBook Pro

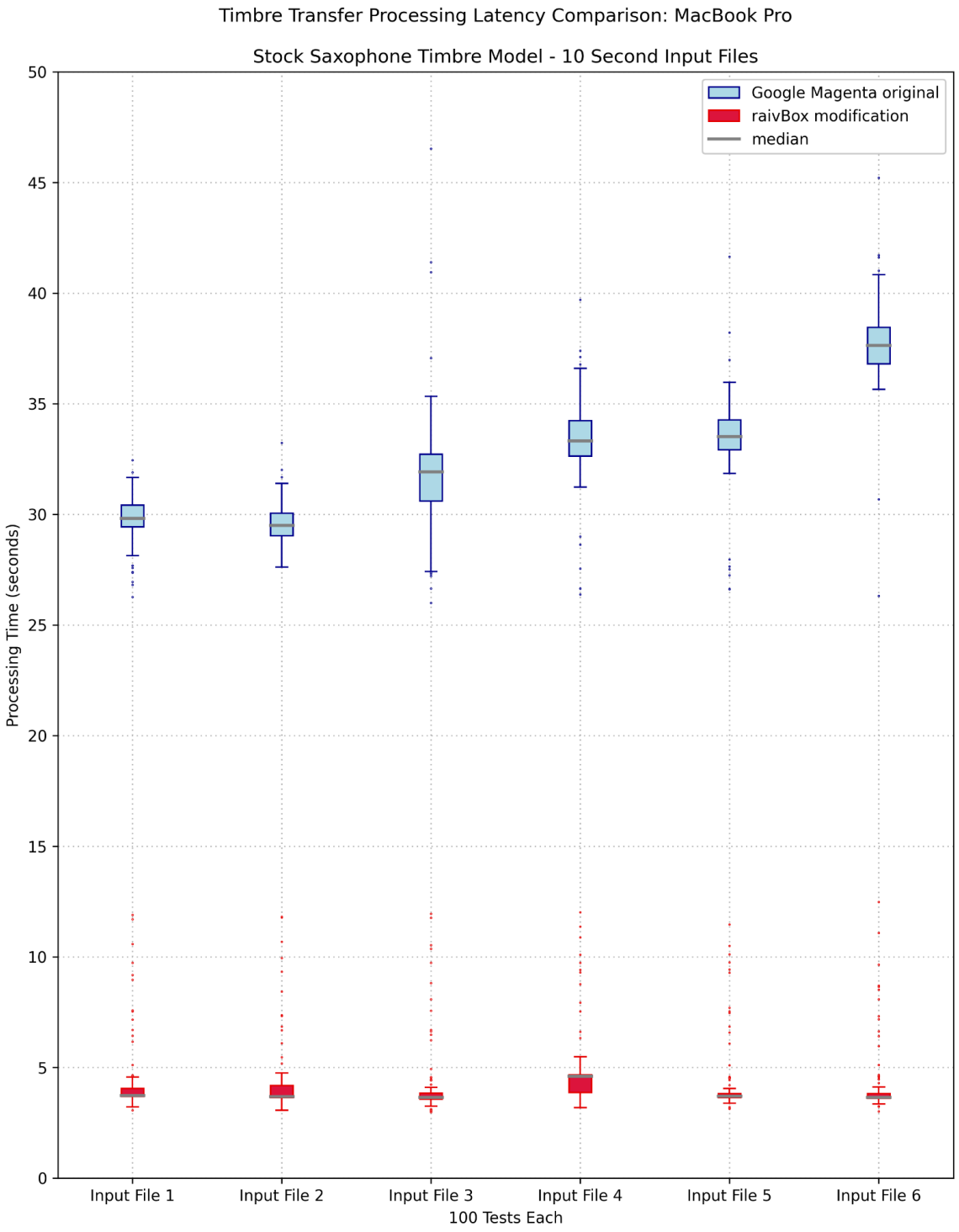


Figure A.2: Saxophone model processing latency comparisons on a MacBook Pro

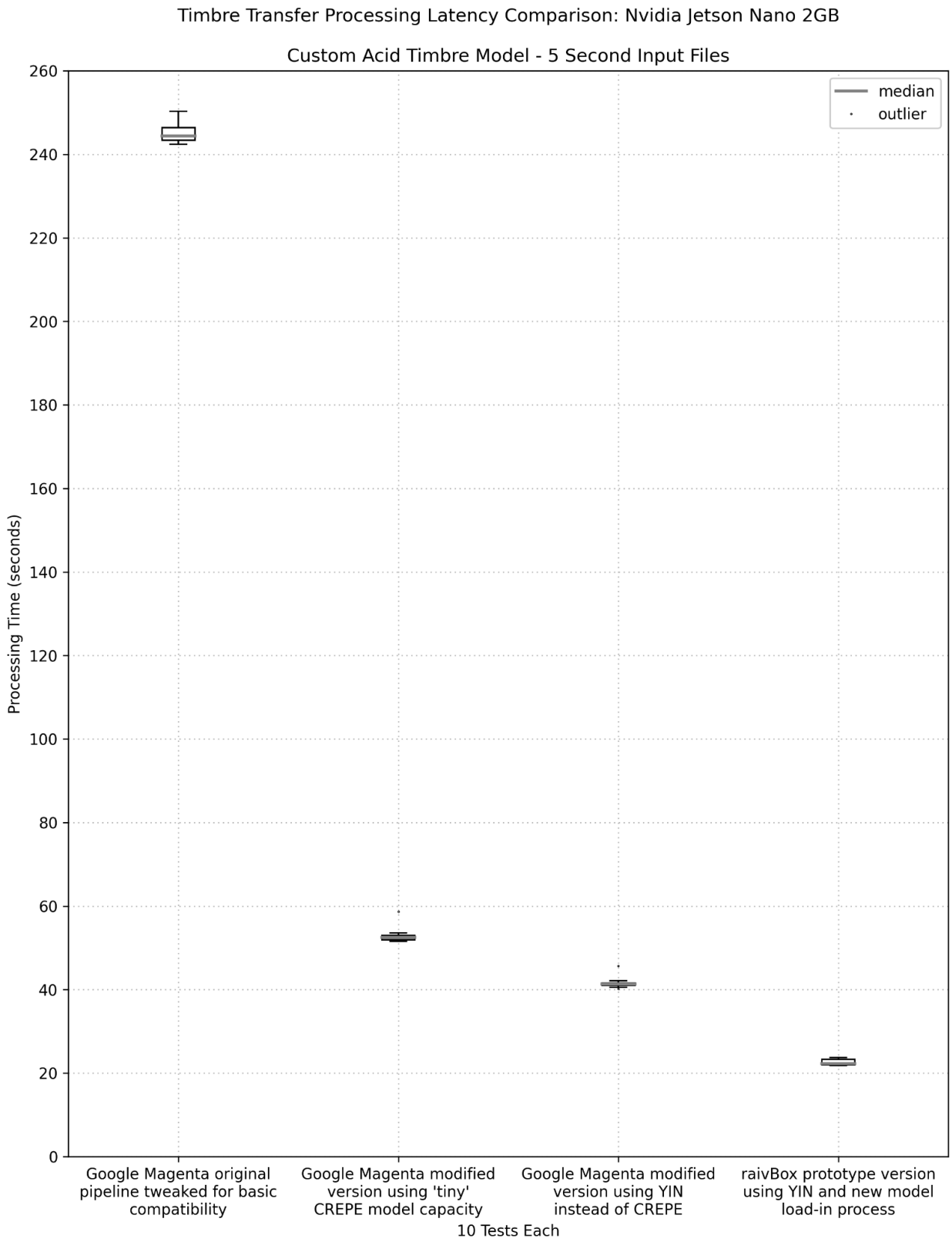


Figure A.3: Acid model processing latency comparisons on the raivBox

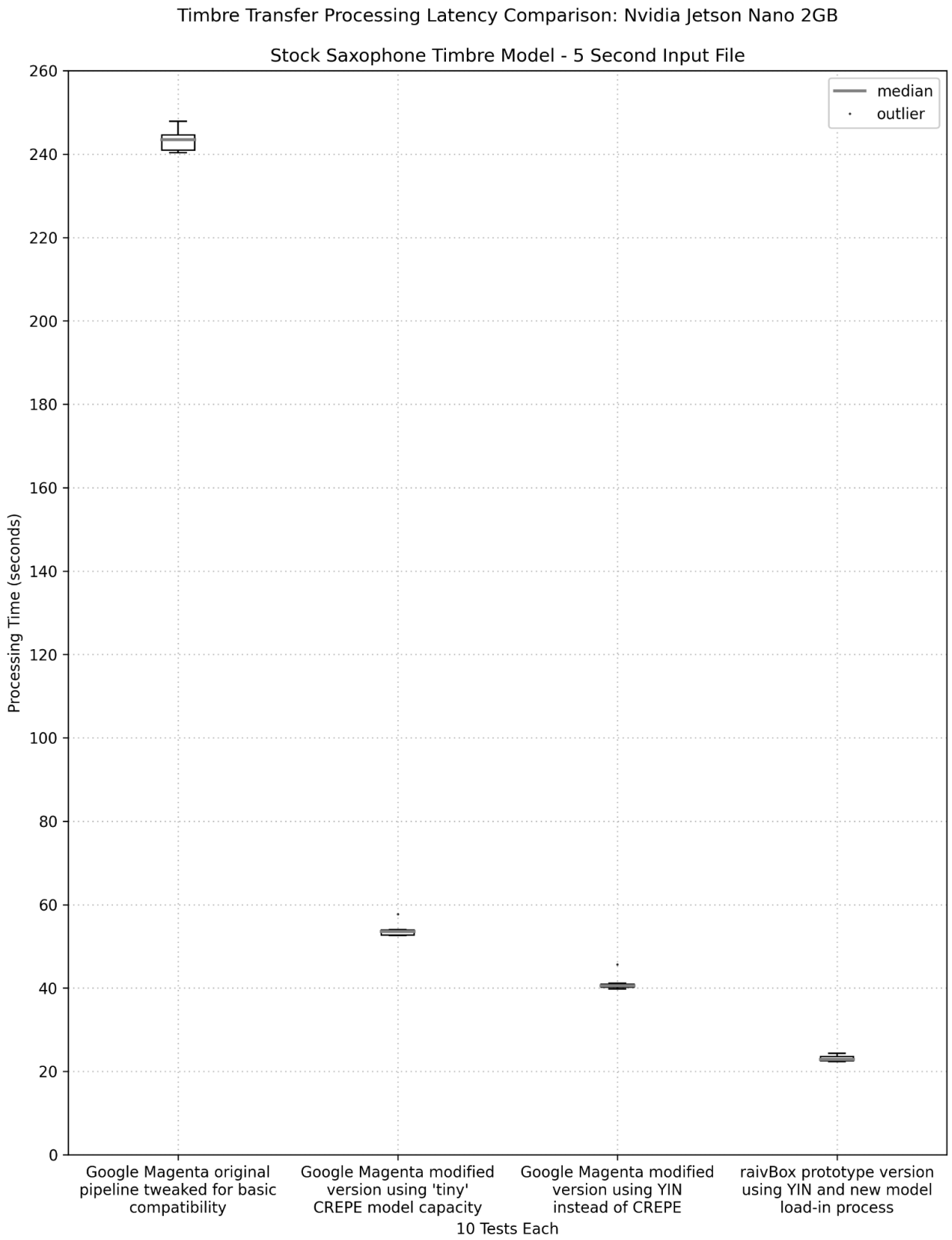


Figure A.4: Saxophone model processing latency comparisons on the raivBox

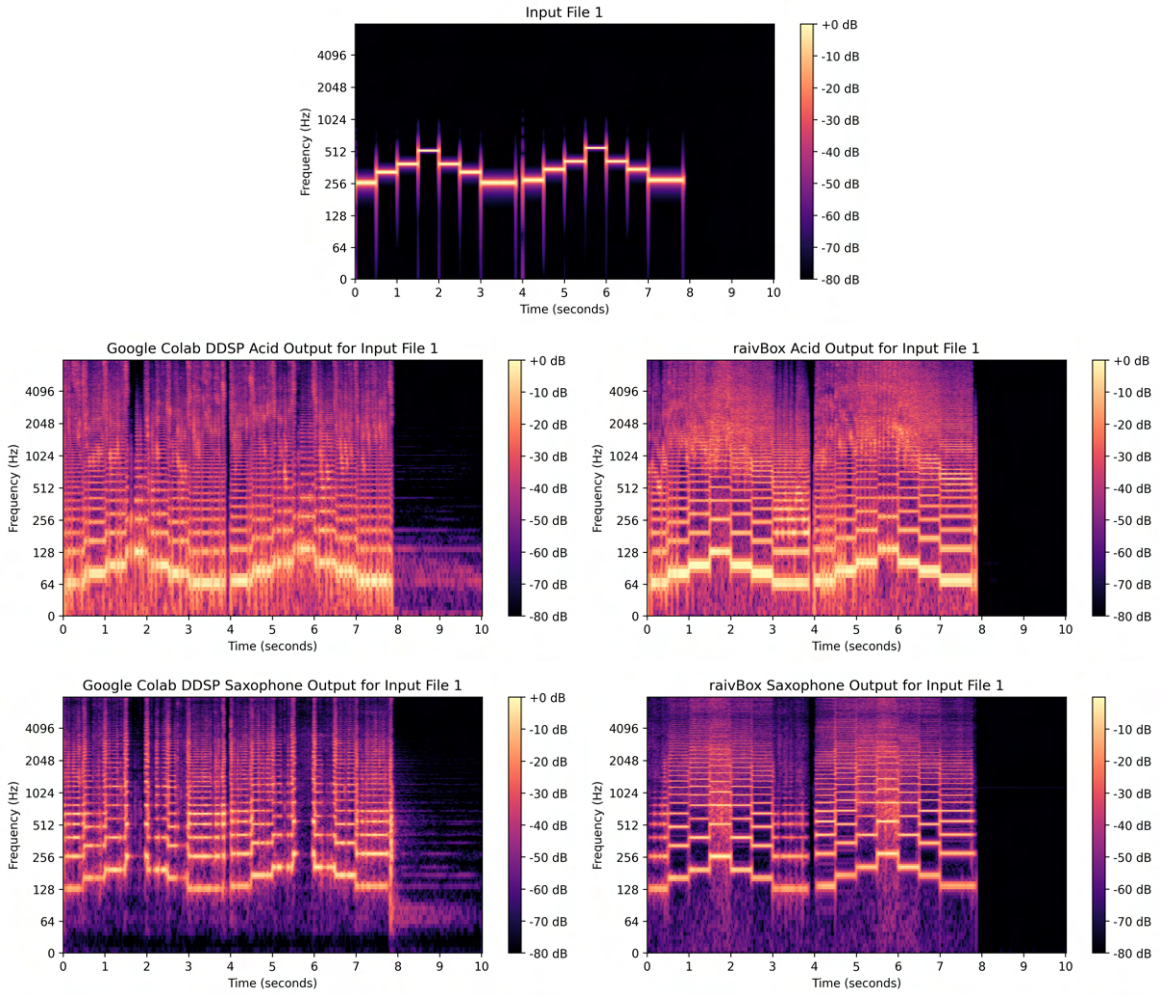


Figure A.5: Log-frequency power spectrograms of sine melody timbre conversions

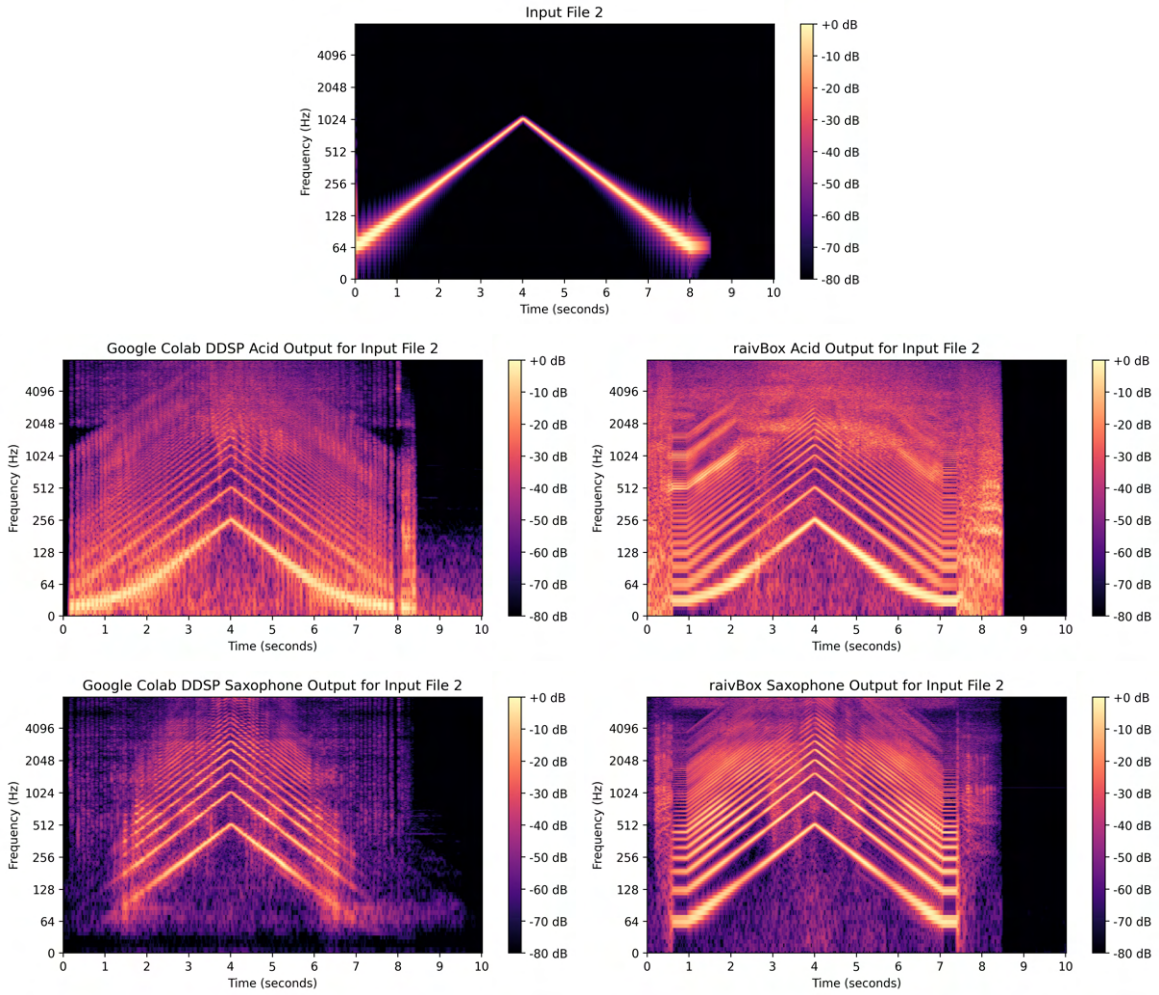


Figure A.6: Log-frequency power spectrograms of sine sweep timbre conversions

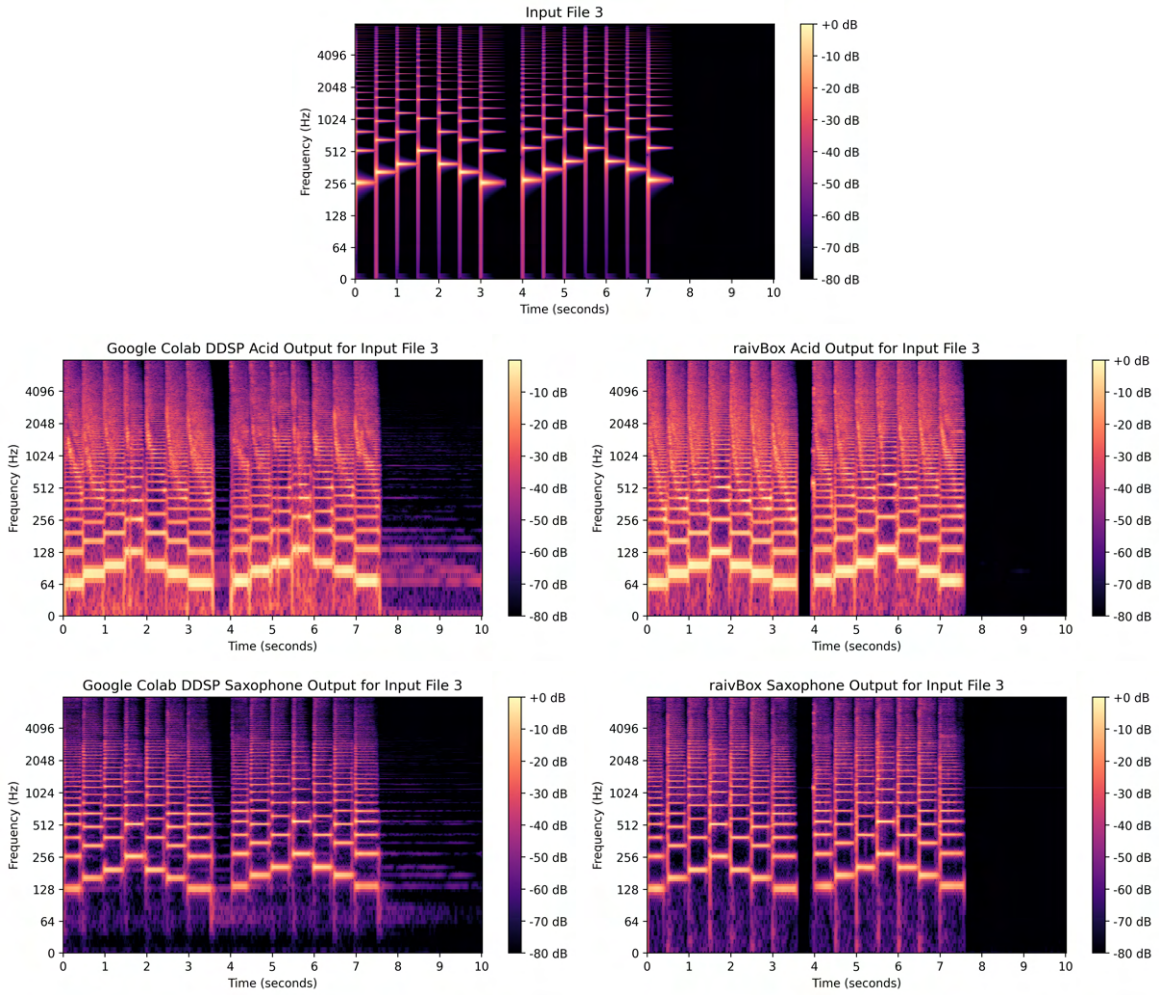


Figure A.7: Log-frequency power spectrograms of saw melody timbre conversions

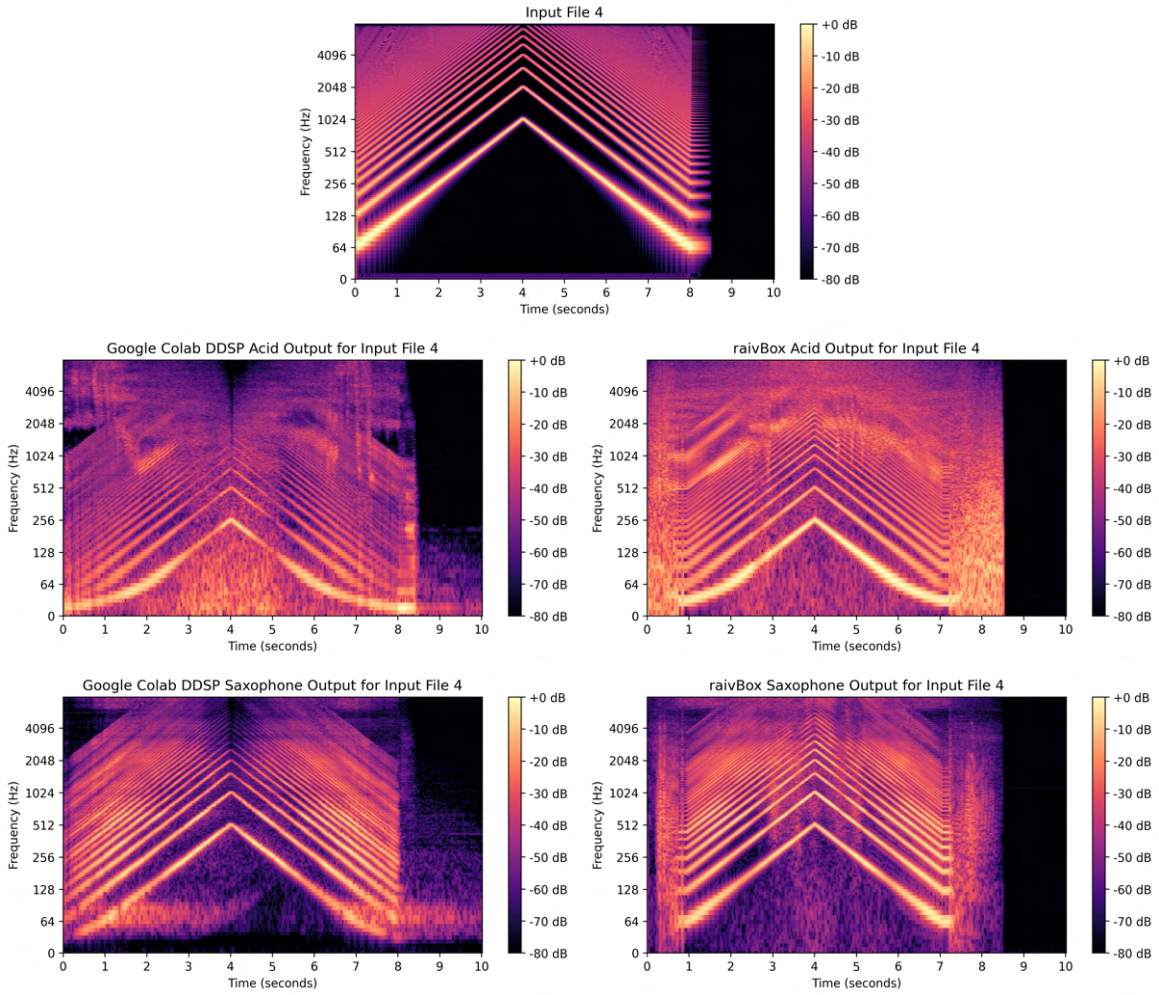


Figure A.8: Log-frequency power spectrograms of saw sweep timbre conversions

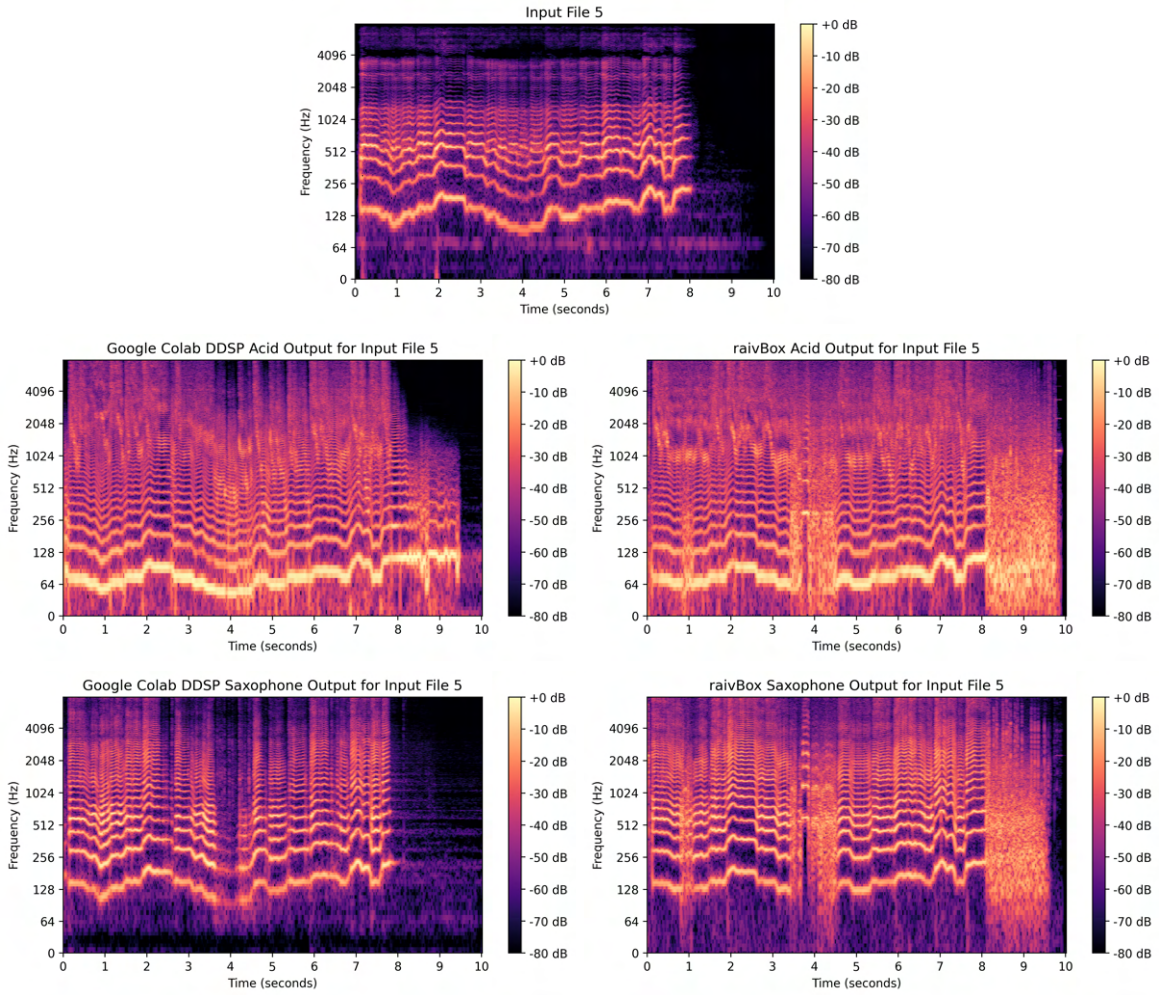


Figure A.9: Log-frequency power spectrograms of singing timbre conversions

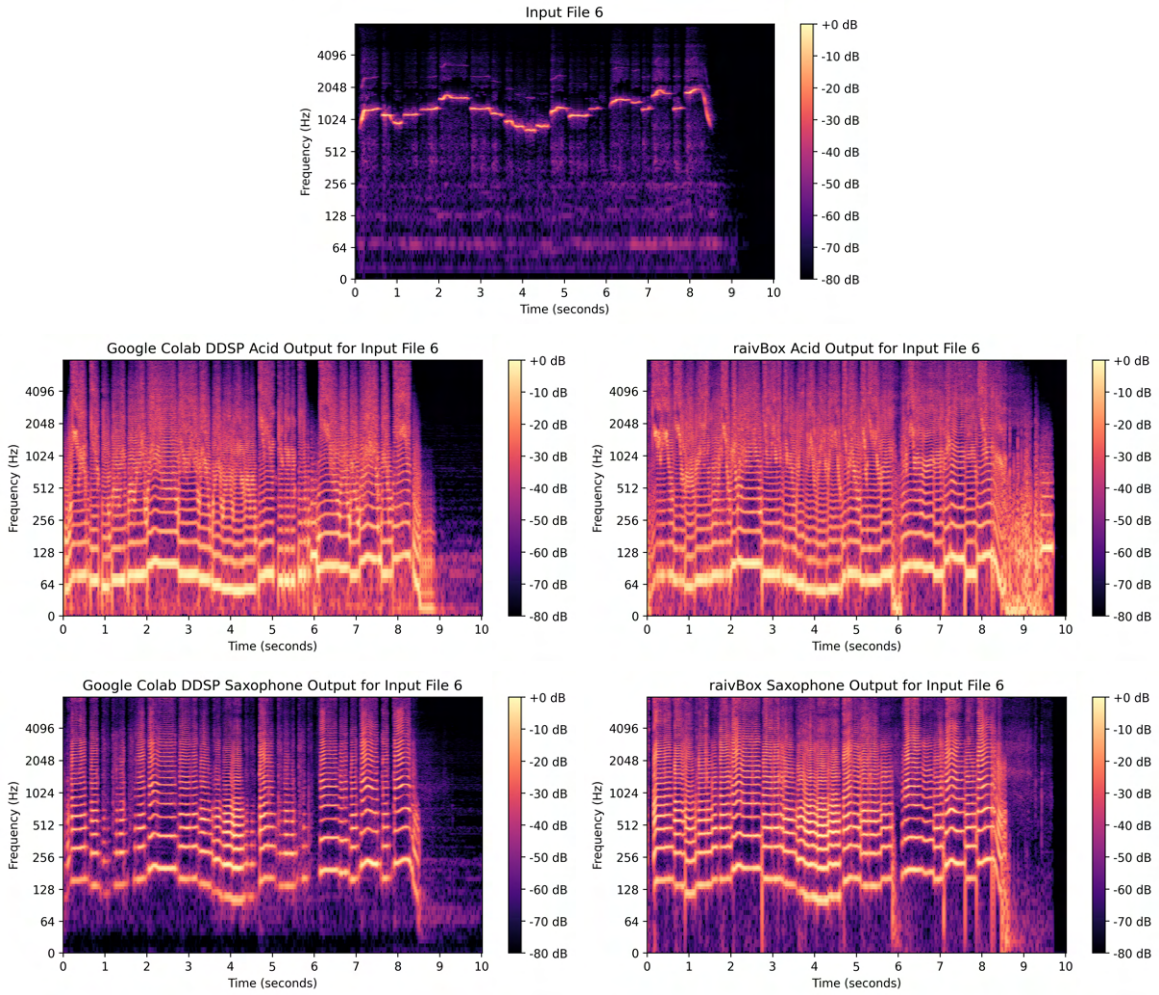


Figure A.10: Log-frequency power spectrograms of whistling timbre conversions