ROLAND CUBE AMPLIFIER

A simulation with Neural Networks

Selected Topics
2023 / 2024 Project

Bertazzi Andres (ID: 10488849)

June 2024

MASTER OF SCIENCE
MUSIC AND ACOUSTIC
ENGINEERING

Contents

1 Introduction

Digital audio processing plugins play a crucial role in the modern music production ecosystem, enabling musicians and audio engineers to manipulate and enhance audio signals with precision and flexibility. The emulation of analog hardware, such as guitar amplifiers, constitutes a significant area of focus in plugin development, as it allows users to access the characteristic tones and textures of iconic audio equipment in a digital environment.

2 Project Overview

2.1 Project Objectives

The objective of this project is to develop a digital audio processing plugin capable of accurately emulating the sound characteristics of a Roland Cube amplifier lead's channel. The Roland Cube series is renowned for its diverse range of amp models and dynamic response, making it a popular choice among musicians for both practice and performance scenarios. By replicating the sonic nuances and tonal versatility of the Roland Cube in a digital plugin, we aim to provide users with a versatile tool for audio production and performance.

- Accuracy

  The primary goal of the plugin is to achieve high fidelity in replicating the tonal qualities and response dynamics of the Roland Cube amplifier. This involves the development of sophisticated signal processing algorithms to capture the unique frequency response, transient behavior, and distortion characteristics of the amplifier across different settings and models.

- Flexibility

  In addition to accuracy, the plugin aims to provide users with flexibility and control over the sound-shaping process. This includes the implementation of a user-friendly interface for adjusting various parameters such as gain, equalization settings (bass, mid, treble), and volume. Furthermore, the plugin will support custom model loading, allowing users to load external amplifier models stored in formats such as .JSON, thereby expanding the plugin's library of available simulations and accommodating diverse musical preferences.

- Performance

  Efficient performance is crucial for real-time audio processing applications. The plugin will be optimized to operate with minimal latency and CPU usage, ensuring smooth performance in live performance and recording scenarios. Additionally, effective resource management techniques will be employed to minimize memory footprint and ensure compatibility across different computing platforms and hardware configurations.

This project aims to create a digital audio processing plugin. The plugin will accurately replicate the sound characteristics of the Roland Cube amplifier and offer flexibility and performance for various audio production needs. By integrating advanced signal processing techniques and user-centric design principles, the goal is to provide musicians and audio engineers with a powerful tool for creative expression and sonic exploration.

3 Technical Details

3.1 Data Acquisition

The initial step involves using a Digital Audio Workstation (DAW) to process the `input.wav` file located in the `train` directory.
The file is played through the amplifier, and the output is captured directly via an audio interface, bypassing any microphones to avoid external acoustic influences. This ensures the audio data represents the amplifier's true response.

3.2 Training Process

The Lead Channel of the Roland Cube is renowned for its extensive array of amplifier emulations. The training script, `train.ipynb`, generates two types of JSON models for each amplifier type through distinct code blocks.
The first code block handles both clean and distorted amplifiers.
It keeps the gain parameter fixed at 50%, using a single `out.wav` file as input.
The resulting JSON model is named `nameOfAmplifierModelGainStable.json`.

The second code block employs gain parameterization, producing models with varying gain levels.
The output JSON file is named `nameOfAmplifierModelParametrizedGain.json`.
For this process, five different audio files (`out1.wav`, `out2.wav`, `out3.wav`, `out4.wav`, `out5.wav`) are used, each corresponding to gain settings of 0%, 25%, 50%, 75%, and 100%, respectively. All models and the corresponding audio files used for training are organized and stored in the `models` directory within the `train` folder, ensuring the repository remains well-structured and easy to navigate.

3.3 JUCE

Once the training phase is complete, the code is implemented in JUCE. The source files are contained in the `src` directory. The project is divided into six distinct classes:

1. **PluginProcessor**: Handles the core processing of audio signals, applying the trained amplifier models to incoming audio data.
2. **PluginEditor**: Manages the graphical user interface (GUI) elements, allowing users to interact with and control the plugin.
3. **RTNeuralLSTM**: Implements a real-time Long Short-Term Memory (LSTM) neural network for dynamic audio processing based on the trained models.
4. **Equalizer**: Provides equalization capabilities to shape the tonal characteristics of the audio signal post-amplification.
5. **CabSimulation**: Simulates the cabinet response, adding realistic speaker and room characteristics to the processed audio signal.
6. **MyLookAndFeel**: Customizes the look and feel of the GUI, ensuring a user-friendly and visually appealing interface.

Each class is meticulously designed to manage specific aspects of the plugin, collaboratively creating a cohesive and functional digital amplifier model. Each class is comprised of both a `.h` (header) file and a `.cpp` (source) file. The following sections will delve deeper into the functionality and implementation details of these classes.

3.3.1 PluginProcessor

The `RolandCubeAudioProcessor` class is central to the audio processing engine of our plugin.
In its constructor, `RolandCubeAudioProcessor::RolandCubeAudioProcessor()`, the class initializes various `AudioProcessor` parameters essential for audio manipulation. This includes setting up configurations for input and output audio channels, and initializing parameters such as gain, bass, mid, treble, master, model, and type. Additionally, a state structure (`treeState`) is established to manage these parameters through a value tree.

During construction, JSON files from the `train/models` directory are loaded into memory, and the cabinet simulation is set up using the `load()` method of the `cabSimIR` object.

A key function within the `RolandCubeAudioProcessor` class is `parameterChanged()`. This function dynamically responds to changes in `AudioProcessor` parameters. It handles adjustments to parameters such as the model or the type of gain for each model, as well as changes in gain, bass, mid, treble, and master settings. This ensures that any user modifications are accurately reflected in the audio output.

To maintain smooth transitions and prevent abrupt changes in audio quality when parameters are adjusted, the class applies smooth gain adjustment techniques with the applyGainSmoothing() function. This ensures that changes in parameters are gradual and imperceptible to the listener, providing a seamless audio experience. Before audio playback begins, the `prepareToPlay()` function is called. Here, necessary resources such as the DC Blocker, Resampler, and cabinet simulation processor (`cabSimIRa`) are initialized to ensure optimal audio processing.

During audio processing, the `processBlock()` function is invoked to handle each incoming audio block. Within this function, a DC Blocker is applied to remove any DC component from the audio signal, which could introduce unwanted artifacts.

For more sophisticated audio processing, the class employs an LSTM (Long Short-Term Memory) neural network model, utilizing two LSTM objects with the function `applyLSTM(buffer, block, LSTM, LSTM2, conditioned, gainValue, previousGainValue, resampler)`. This model helps manage gain and ensures smooth transitions between parameters, thereby enhancing the overall audio output quality.

To further fine-tune the audio output, an equalizer is utilized to adjust bass, mid, and treble frequencies according to user preferences. Additionally, a master gain control allows users to adjust the overall volume to their liking. Initially, the cabinet simulation was applied immediately after the equalizer. However, following auditory tests during the testing phase, it was decided to maintain the original tonal characteristics of the audio output based on the models, rather than having it influenced by additional external IRs. The class also includes functionality for managing the plugin's user interface through the `RolandCubeAudioProcessorEditor` class. This ensures that users can easily interact with and control various parameters of the plugin.

Furthermore, the class provides methods for saving and loading the plugin's state, including the selected model and parameter settings. This ensures that users can seamlessly resume their audio projects from where they left off. Overall, the `RolandCubeAudioProcessor` class serves as the backbone of our plugin's audio processing capabilities, providing a robust framework for manipulating audio signals with precision and flexibility.

3.3.2 PluginEditor

This class is responsible for managing the graphical user interface (GUI) elements, enabling users to interact with and control the plugin effectively. The images are housed within the directory named `resources/images`.

- **Constructor of the `RolandCubeAudioProcessorEditor` class**:
  This constructor initializes the plugin's editor. It sets up the visual appearance of components like rotary controls and buttons and associates them with the `AudioProcessor` parameters using `SliderAttachment` and
  `ButtonAttachment` objects.
- **Destructor of the `RolandCubeAudioProcessorEditor` class**:
  In the destructor, the `LookAndFeel` associated with the controls is removed to avoid memory leaks.
- `paint()`: This method is used to draw the GUI. Depending on the platform (Windows or other), it draws the background of the GUI, the Roland logo, the outlines of components, and the images of controls.
- `resized()`: This method is called when the application window is resized and is used to position and resize GUI components. Here, layouts for controls like bass, mid, treble, model selector, gain control, and volume control are defined.

The gainType variable is used during the testing phase to dynamically switch between an array of GainStable models and an array of ParametrizedGain models in real time. This facilitates on-the-fly comparisons to determine which gain type sounds better. Essentially, this code creates a GUI for an audio plugin, featuring rotary controls for bass, mid, treble, gain, and volume, as well as model selection options and a button to select the gain type.

3.3.3 RTNeuralLSTM

The `RTNeuralLSTM` class serves as a wrapper for utilizing LSTM (Long Short-Term Memory) neural networks in real-time applications. It provides methods for loading weights from JSON files, processing audio data using the LSTM models, and handling parameter changes during processing.
Here's a detailed description of the class and its components:

- **Neural Network Architecture:**
  The neural network architecture consists of multiple layers designed to process sequential data effectively. Specifically, the architecture includes:
  - **Input Layer:** The input layer of the LSTM neural network receives sequential input data. In this implementation, the input size is determined based on the application requirements.
  - **LSTM Layers:** Long Short-Term Memory (LSTM) layers are specialized recurrent neural network (RNN) layers capable of learning and retaining information over long sequences. These layers capture temporal dependencies in the input data and are crucial for tasks involving sequential data processing. In this implementation, the LSTM layers are configured with specific input and output dimensions to suit the application's requirements. Each LSTM layer consists of memory cells, input gates, output gates, and forget gates, allowing the network to learn complex temporal patterns.
  - **Dense Layers:** Dense layers, also known as fully connected layers, perform transformations on the output of the LSTM layers to produce the final output. These layers integrate information learned from the LSTM layers and perform additional processing to generate the desired output format.

- **Operations and Methods:**
  The `RTNeuralLSTM` class provides several methods to facilitate the training and inference processes of the LSTM neural network:

  - `transpose(const Vec2d& x)`: This method computes the transpose of a two-dimensional matrix of type `Vec2d`. It is utilized to manipulate weight matrices within the LSTM layers.
  - `set_weights(T1 model, const char* filename)`: This template method sets the weights of the LSTM models based on the provided JSON file containing the model weights. It enables the initialization of the network with pre-trained parameters.
  - `load_json(const char* filename)`: This method loads the weights of the LSTM models from a JSON file. It determines the input size of the model and loads the weights into the appropriate LSTM models using the `set_weights` method.
  - `reset()`: This method resets the state of the LSTM models, ensuring that the network starts processing new input data from an initial state. Depending on the network configuration, it may reset either the main model or conditional models.
  - `process(const float* inData, float* outData, int numSamples)`: This method processes incoming audio data using the main LSTM model. It generates predicted output for each audio sample and combines it with the original input to produce the final output.
  - `process(const float* inData, float param, float* outData, int numSamples)`: This method processes incoming audio data using a conditional LSTM model, considering an additional parameter during processing. It adjusts the network's behavior based on the parameter value to ensure smooth transitions and accurate predictions.
  - `process(const float* inData, float param1, float param2, float* outData, int numSamples)`: Similar to the previous method, this method processes incoming audio data using a second conditional LSTM model that considers two additional parameters. It incorporates multiple parameters into the processing pipeline to capture more complex relationships in the input data.

The class implementation includes declarations for the LSTM models (`model`, `model_cond1`, `model_cond2`) and pre-allocated arrays (`inArray1`, `inArray2`) for feeding input data to the models efficiently. The neural network architecture, defined within the private section of the class, comprises LSTM layers and dense layers configured with specific input and output dimensions tailored to the application's requirements. Additionally, the class utilizes methods for manipulating weight matrices, loading model weights from JSON files, and processing input data using the LSTM models with support for conditional processing based on additional parameters.

3.3.4 Equalizer

The Equalizer class, as its name suggests, implements an audio equalizer. Here's a detailed explanation of the functions and operations performed in the code:

- `Equalizer::Equalizer()`: This is the constructor of the class. It's called when a new `Equalizer` object is instantiated. In this case, it sets the equalizer parameters to default values (all 0).
- `process()`: This function is responsible for processing the incoming audio for equalization. It takes several arguments, including an array of incoming audio data (`inData`), the array where the equalized audio data will be written (`outData`), the MIDI buffer (`midiMessages`), the number of samples to process (`numSamples`), the number of input audio channels (`numInputChannels`), and the sampling frequency of the audio signal (`sampleRate`). The function performs equalization for each incoming audio sample using a three-band equalization filter (low, mid, high), and the equalized signal is written to the `outData` array.
- `setParameters()`: This function sets the equalizer parameters based on the values provided through the bass, mid, treble, and presence sliders. It's called whenever the sliders of the equalizer controls are changed.
- `resetSampleRate()`: This function is called when the sampling frequency of the audio signal changes. It updates the equalizer parameters based on the new sampling frequency.

The implemented equalizer is a three-band equalizer (bass, mid, high).
It utilizes a series of filters to adjust the frequencies of the incoming audio signal, producing an equalized output based on the configured parameters.

3.3.5 CabSimulation

Let's delve into a comprehensive explanation of the inner workings of the CabSimulation class.

- `CabSimulation()`: This is the default constructor of the class. It takes no arguments and doesn't perform any specific operation but initializes the `CabSimulation` object.
- `prepare(const juce::dsp::ProcessSpec& spec)`: This function takes a `juce::dsp::ProcessSpec` object as an argument and uses it to prepare the audio processor chain (`processorChain`). The audio processor chain is an object of the `juce::dsp::ProcessorChain` class containing a single convolution processor (`juce::dsp::Convolution`). This function prepares the processor chain to process audio data according to the provided specifications.
- `process(const ProcessContext& context) noexcept`:
  This function takes a process context (`ProcessContext`) as an argument and uses it to process audio data through the processor chain (`processorChain`). Since it's marked as `noexcept`, it's assumed not to throw exceptions during execution.
- `reset() noexcept`:
  This function resets the processor chain (`processorChain`). This function is also marked as `noexcept`, so it doesn't throw exceptions during execution.
- `load(const void* sourceData, size_t sourceDataSize) noexcept`:
  This function loads an impulse response (IR) into the convolution object within the processor chain. The IR is provided as a pointer to data (`sourceData`) and its size (`sourceDataSize`). This function seems to be designed to load the IR from an external source.

The `CabSimulation` class orchestrates a sophisticated cabinet effect emulation through convolution, a prevalent technique in audio signal processing renowned for its ability to replicate speaker, microphone, and room responses. During testing, exhaustive trials were conducted to ascertain the optimal approach, evaluating the efficacy of including or excluding cabinet simulation. This pivotal decision is encapsulated in a succinct line within the `processBlock()` function of the PluginProcessor.

3.3.6 MyLookAndFeel

The main functions in this class include:

1. `setLookAndFeel`: This function takes an image as an argument and assigns it to the member variable `img`.
2. `drawRotarySlider`: This function is an override of a pure virtual method from the JUCE `LookAndFeel` class, and it is utilized to draw a rotary slider component.
   - **Parameters:**
     - `sliderPos`: Represents the position of the slider's thumb.
     - `rotaryStartAngle` and `rotaryEndAngle`: Represent the start and end angles of the rotary control, respectively.
   - **Functionality:**
     - Calculates a rotation value based on the slider's value relative to its minimum and maximum values.
     - Determines the ID of the image frame to draw based on the rotation and the number of frames contained in the image.
     - Calculates the center and radius of the rotary control.
     - Draws the appropriate frame of the rotary control image using the provided `Graphics` context as an argument.

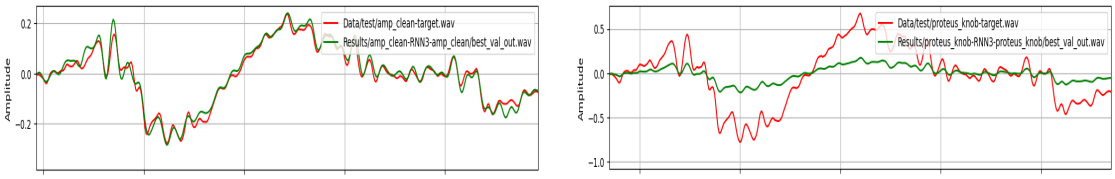
In summary, this code customizes the appearance of a rotary slider component in a graphical user interface using a specified image. The images are stored within the directory named `resources/images` for organizational purposes.

4 Evaluation Description

4.1 Models Graphics

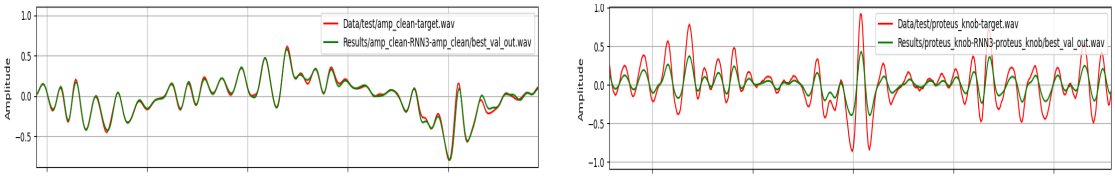
In the following paragraph, a series of graphs extracted from `train.ipynb` are presented, comparing the performance of the two types of models, GainStable and ParametrizedGain, for each channel. The graphs display red and green curves, which represent the original signal and the predictions made by the models in time-domain, respectively.

Particularly, it is observed that for almost all channels, the performance was excellent for both models. Consequently, it became necessary to conduct subjective tests based on individual musicians' perceptions during the plugin's usage. All the graphics can be seen in detail for each model in the `train/graphics` directory.



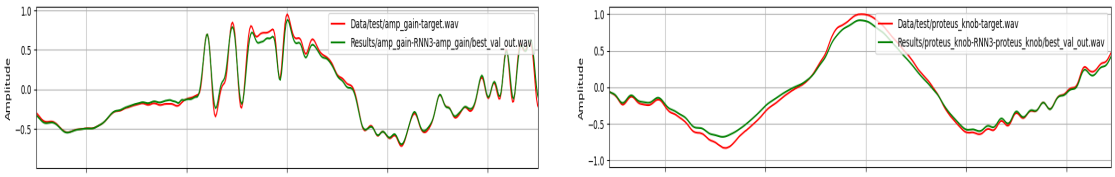
Gain Stable Parametrized Gain

Figure 1: Acoustic Models



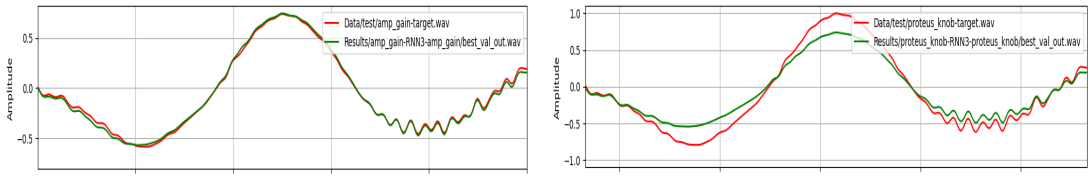
Gain Stable Parametrized Gain

Figure 2: Black Panel Models



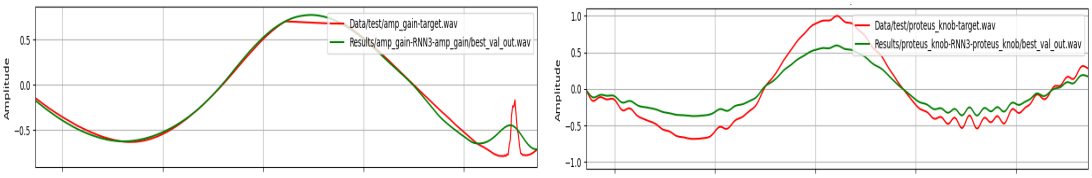
Gain Stable Parametrized Gain

Figure 3: British Combo Models



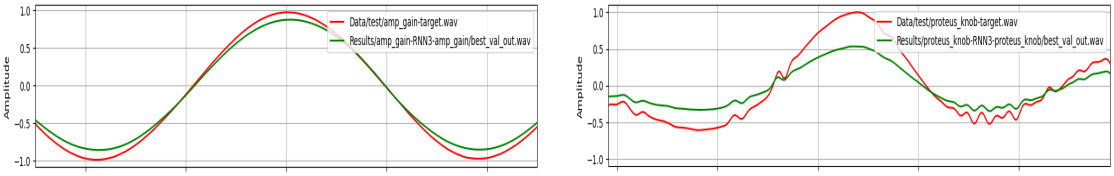
Gain Stable Parametrized Gain

Figure 4: Classic Models



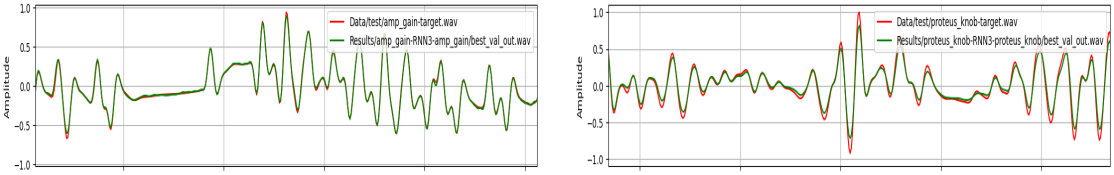
Gain Stable Parametrized Gain

Figure 5: Metal Models



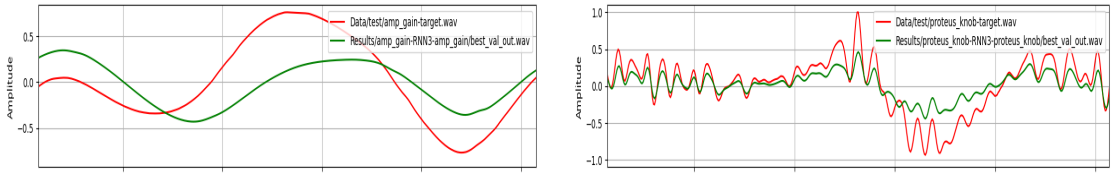
Gain Stable Parametrized Gain

Figure 6: R-Fier Models



Gain Stable Parametrized Gain

Figure 7: Extreme Models



Gain Stable Parametrized Gain

Figure 8: Dynamic Amp Models

4.2 Evaluation Objectives

The evaluation seeks to scrutinize the plugin's proficiency in emulating the nuanced sound characteristics of the Roland Cube amplifier and its adaptability in real-world applications.

4.3 Methodology

The evaluation amalgamates subjective and objective assessments:

- **Subjective Listening Tests:** Participants engage in discerning audio comparisons between plugin-processed samples and reference recordings from a Roland Cube amplifier, offering insights into perceived sonic parallels and distinctions.
- **User Experience Surveys:** Participants interact with the plugin interface, providing feedback on its intuitiveness, aesthetic appeal, and overall user-centric design.
- **Performance Metrics:** Objective parameters like latency, CPU utilization, and memory consumption are meticulously gauged during plugin execution to gauge computational efficiency.

4.3.1 Model Selection Process

Upon evaluating the twin arrays of amplifier models, namely `gainStable` and `parametrizedGain`, users were entrusted with the pivotal task of subjectively scrutinizing their performance within the plugin ecosystem.

Following exhaustive testing and subjective evaluations, users assumed the mantle of authority in selecting the amplifier model that seamlessly harmonized with their artistic inclinations and project exigencies. Ultimately, the Parametrized Gain models emerged as the preferred choice due to their superior response to the dynamics of fingers on the guitar's strings and the gain knob.
Furthermore, it was decided to eliminate the CabSimulation object. Subjective testing revealed a preference for preserving the unaltered output response of the various models, without the influence of external IRs.

5 Conclusion

In conclusion, the project represents a significant milestone in the realm of digital audio processing by successfully developing a plugin that impeccably emulates the distinctive sound characteristics of a Roland Cube amplifier. Leveraging advanced machine learning techniques, the plugin emerges as a versatile and indispensable tool for enhancing audio recordings and live performances alike. Through meticulous integration of sophisticated algorithms, it faithfully reproduces the nuanced tonal qualities and dynamic response of the original amplifier across a diverse range of settings and models.

The comprehensive evaluation process underscores the plugin's efficacy, affirming its capacity to seamlessly replicate the sonic essence of the Roland Cube amplifier while demonstrating commendable usability in real-world scenarios. Subjective listening tests, user experience surveys, and meticulous performance measurements collectively attest to the plugin's remarkable fidelity, user-friendly interface, and robust computational efficiency.

In essence, the project stands as a testament to the transformative potential of cutting-edge technology in shaping the future landscape of audio production and creativity.

6 Students

Figure 9:
Andres
Bertazzi

7 Links

GitHub Repository:
https://github.com/andrewbertax96/RolandCube_Amplifier

Video Demo:
https://www.linkedin.com/posts/andres-bertazzi-61952411a_music-coding-engineering-activity-7207376184
dJX?utm_source=share&utm_medium=member_desktop