REAL-TIME TIMBRE TRANSFER and SOUND SYNTHESIS USING DDSP

Ganis, Francesco; Knudsen, Erik F.; Lyster, Søren V.K.; Otterbein, Robin; Südholt, David; Erkut, Cumhur

Published in:
SMC 2021 - Proceedings of the 18th Sound and Music Computing Conference

DOI (link to publication from Publisher):
10.5281/zenodo.5043235

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

REAL-TIME TIMBRE TRANSFER AND SOUND SYNTHESIS USING DDSP

Francesco GANIS (fganis20@student.aau.dk) [1], **Erik F. KNUDSEN** (eknuds20@student.aau.dk) [1], **Søren V. K. LYSTER** (slyste20@student.aau.dk) [1], **Robin OTTERBEIN** (rotter20@student.aau.dk) [1], **David SÜDHOLT** (dsudho20@student.aau.dk) [1], and **Cumhur ERKUT** (cer@create.aau.dk) (0000-0003-0750-1919) [1]

[1] *Sound and Music Computing, Department of Architecture, Design, and Media Technology*, **Aalborg University**, A.C. Meyers Vænge, Copenhagen, DK-2450 Denmark

ABSTRACT

Neural audio synthesis is an actively researched topic, having yielded a wide range of techniques that leverages machine learning architectures. Google Magenta elaborated a novel approach called Differential Digital Signal Processing (DDSP) that incorporates deep neural networks with preconditioned digital signal processing techniques, reaching state-of-the-art results especially in timbre transfer applications. However, most of these techniques, including the DDSP, are generally not applicable in real-time constraints, making them ineligible in a musical workflow. In this paper, we present a real-time implementation of the DDSP library embedded in a virtual synthesizer as a plug-in that can be used in a Digital Audio Workstation. We focused on timbre transfer from learned representations of real instruments to arbitrary sound inputs as well as controlling these models by MIDI. Furthermore, we developed a GUI for intuitive high-level controls which can be used for post-processing and manipulating the parameters estimated by the neural network. We have conducted a user experience test with seven participants online. The results indicated that our users found the interface appealing, easy to understand, and worth exploring further. At the same time, we have identified issues in the timbre transfer quality, in some components we did not implement, and in installation and distribution of our plugin. The next iteration of our design will address these issues.

1. INTRODUCTION

Sound synthesizers have been widely used in music production since the late 50s. Because of their inner complexity, many musicians and producers polish presets' parameters until they reach the desired sound. This procedure is time-consuming and sometimes results in failed attempts to achieve a desired sound.

Much research has been done in the area of automating the generation of these sounds through the aid of machine learning and neural networks. Common approaches included directly generating the waveform in the time domain [1] or predicting synthesis parameters based on hand-picked analysis features [2]. In their 2020 paper on Differentiable Digital Signal Processing (DDSP) [3], Engel et al. proposed a novel approach to neural audio synthesis. Rather than generating signals directly in the time or frequency domain, DDSP offers a complete end-to-end toolbox consisting of a synthesizer based on Spectral Modeling Synthesis (SMS) [4], and an autoencoder neural network architecture that takes care of both extracting analysis features and predicting synthesis parameters.

The authors of the DDSP paper released a public demonstration of "tone transfer" [1], allowing the user to upload their own recordings, select from a list of models trained on various instruments and "transfer" their recorded melodies to the sound of a trumpet, a violin etc. Based on these, we implemented the DDSP back-end as a virtual instrument playable in real-time. Figure 1 shows the GUI of our synthesizer.

This paper documents the background, our requirement-driven design and implementation approach, including model components and training, the GUI design, and user experience evaluation. The structure of this paper follows these main topics in order.

Besides our contribution to the real-time neural audio synthesis and its user experience evaluation, we release our real-time MATLAB and JUCE implementations at `https://github.com/SMC704/juce-ddsp` and `https://github.com/SMC704/matlab-ddsp`, respectively. We also provide a demonstration video at `https://share.descript.com/view/hXAZLCPJNqm`.

Figure 1. Our real-time DDSP Synthesizer GUI.

[1] `https://sites.research.google/tonetransfer`, last accessed on 2020-11-30

2. BACKGROUND

In addition to the DDSP paper [3], our work is inspired by the commercially produced additive synthesizer called *Razor* by Native Instruments [5]. Razor's core consists of a powerful additive synthesizer and features various modulation options for manipulating the sound output. What is especially interesting about Razor is that every modulation option (e.g. filters, stereo imaging, reverbs and delays) is actually modulating individual partial harmonics (non-integer multiples of the fundamental frequency) in the additive synthesis engine. Furthermore, Razor enables musicians and producers to intuitively control partials via different parameters while relying on a visual representation of partial manipulation. We therefore focused on the harmonic and the stochastic components of the DDSP.

2.1 Harmonic Oscillator / Additive Synthesizer

The additive synthesizer is the main core of the whole synthesis and is responsible for generating all the harmonic components of the reconstructed sound. The output is characterized by the sum of several harmonic integer multiples of the fundamental frequency f_0:

$$f_k(n) = k \cdot f_0(n). \tag{1}$$

In order to generate the harmonics, we can implement k oscillators in the discrete time:

$$x(n) = \sum_{k=1}^{K} A_k(n) \cdot \sin(\phi_k(n)), \tag{2}$$

where $A_k(n)$ is the time-varying amplitude of the k_{th} sinusoidal component and $\phi_k(n)$ is its instantaneous phase. $\phi_k(n)$ is obtained by integrating the instantaneous frequency $f_k(n)$ [3]:

$$\phi_k(n) = 2\pi \sum_{m=0}^{n} f_k(m) + \phi_{0,k}. \tag{3}$$

The only two parameters necessary to control the synthesizer are the frequency $f_0(n)$ and the harmonic amplitudes $A_k(n)$. These are retrieved directly from the input sound using the encoder contained in the autoencoder network. As reported in [3], the network outputs are scaled and normalized to fall within an interpretable value range for the synthesizer.

2.2 Filtered Noise, Subtractive Synthesizer, and Reverb

The subtractive synthesis is used to recreate the non-harmonic part of natural sounds. The parameters necessary to obtain a frequency-domain transfer function of a linear time-variant finite impulse response (LTV-FIR) filter are retrieved from the neural network in frames that are subsets of the input signal. The corresponding impulse responses (IRs) are calculated and a windowing function is applied. The windowed IRs are then convolved with white noise via transformation to and multiplication in the frequency domain. Another LTV-FIR filter acts as a reverberator, performing esentially a convolution reverb in the frequency domain.

2.3 Research question & design requirements

Based on this background we have formulated the following research question: How can we develop a playable software instrument, based on the DDSP library, that would: a) allow customization of model-estimated synth parameters through top-level macro controls, b) enable existing workflow-integration in Digital Audio Workstations (DAWs), and c) facilitate a simple approach for beginners without limiting usability for expert music producers?

Based on this research question, we have identified five user-objectives [6], matched them with a solution, and reformulated them as design requirements that address the following functionality: building a playable real-time software-instrument plugin that supports different composition techniques by having audio and MIDI input modes. The instrument must include at least four models which serve the purpose of estimating synthesizer parameters to output a desired sound. Finally, the instrument must include graphical user interface components providing intuitive controls for the manipulation of synthesizer and effect parameters. The design requirements are documented on Table 1.

3. DESIGN & IMPLEMENTATION

3.1 Architecture overview

To meet our criteria of creating a real-time software instrument, we decided to build the plugin in C++ using the JUCE application framework [2] . With JUCE, we had a multi-platform supported audio plugin template that was handling MIDI and audio inputs and outputs. This allowed us to mainly focus on the audio processing and GUI.

Creating a real-time implementation of the non-real-time DDSP library posed some immediate challenges. To analyze and understand these challenges we decided to start by doing a direct translation of the additive and subtractive synthesizers from the DDSP library into MATLAB. The synthesizers could then be changed into real-time implementations and tested. In order to use our MATLAB implementation in the JUCE framework, we used inbuilt MATLAB tools to generate C++ code.

We transformed the autoencoder models pretrained by Google into models that could be used to estimate synthesizer parameters directly from our plugin's user input.

A general overview of this architecture can be seen in figure 2. The following sections will discuss each component in more detail.

3.1.1 Synth in MATLAB

MATLAB's environment and visualization tools gave us access to quick prototyping and testing. This allowed us to do the implementation over multiple iterations. We tested our synthesizers' compatibility with the predicted parameters from the DDSP models by invoking the encoders and decoders in isolation through MATLAB's Python interface.

At first we implemented the non-real-time synthesis algorithms of the DDSP library. Then the synthesizers were changed to real-time, i.e., synthesizing a single frame at

[2] `https://juce.com/`, last accessed on 2020-12-15

#	User Obj.	Solution	Design Requirement
1	Provide a new playable instrument for unique sound generation and inspiration	Real-time implementation	*Must work in real-time as a playable software instrument.*
2	Conveniently integrate into existing workflows	Plugin format application	*Must be implemented as a software plugin.*
3	Adapt to different composition methods	Allow line and MIDI input	*Must allow switching between Line and MIDI input.*
4	Easy fast unique sound generation	Choose models for sound generation	*Must implement at least four pre-trained models.*
5	Convenient customizability of sounds	Tweakable parameters that effects the audio output	*Must include GUI components for intuitive manipulation of synth and effects parameters.*

Table 1. Documentation of Design Requirements

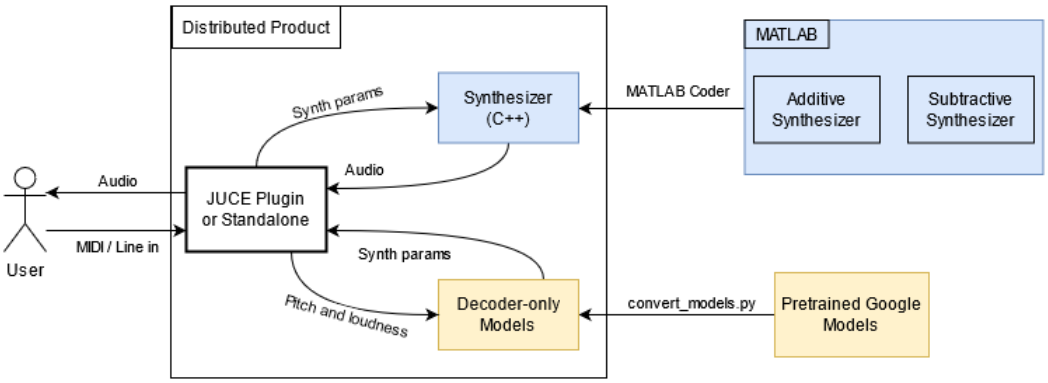

Figure 2. Schematic overview of the project architecture.

a time. Using the MATLAB Audio Test Bench, we could then test the functionality of the synthesizer components and parameters with real-time audio and varying sample rate and buffer size. The last iterations consisted of optimizing the code with the constraints of real-time audio processing on CPUs.

3.1.2 MATLAB to C++

Using the MATLAB coder tool [3] we were able to generate C++ functions from the MATLAB code. For the simplest integration between the generated C++ functions and the JUCE plugin we chose to limit the function inputs and outputs to built-in and derived C++ data types. This required our MATLAB functions to have fixed-sized inputs and outputs. We decided on a maximum input/output size of 4096 double-precision floating point numbers, this being the maximum buffer size the plugin could handle.

A helper file was created to ensure code consistency, allowing the user and MATLAB coder to verify the functions with different inputs. Having this setup made it easy to go back to the MATLAB code and generate updated C++ functions without breaking the JUCE plugin.

3.1.3 TensorFlow in C++

Running the DDSP TensorFlow implementation in a real-time audio application is a heavy computational challenge. Moving from TensorFlow in Python to the TensorFlow C API [4] allowed us to integrate the models into the C++ codebase. By moving the TensorFlow computations to a separate thread, we load the models, set the inputs, run the parameter estimation and save the outputs, without experiencing buffer underruns in the main audio processing thread.

3.1.4 Input signals

The DDSP autoencoder needs the input values *fundamental frequency* (f_0) and *loudness*. Since we allow both MIDI and line-in audio, two separate implementations are needed to calculate these values, which were first created in MATLAB. In the C++ implementation we chose the YIN pitch tracking algorithm [7] from the C library Aubio [8], since it yielded more precise results.

3.2 Training models

DDSP autoencoders are trained to reconstruct waveforms with minimal perceptual loss. Similarity of the raw waveform however is not a good indicator for perceptual similarity, which is why the DDSP library makes use of multi-scale spectral loss [3]. The total reconstruction loss is the sum of multiple spectral losses, i.e., the difference of the magnitude spectrograms, over various time scales. Moreover, the linear magnitude losses are sensitive to the peaks, whereas logarithmic magnitude losses are sensitive to the quiet regions of the signals. Therefore, the sum of losses $L = \sum_i L_i$ in DDSP are calucated in six different frame sizes by

$$L_i = ||S_i - \hat{S}_i||_1 + ||log(S_i) - log(\hat{S}_i)||_1. \quad (4)$$

[3] https://se.mathworks.com/products/matlab-coder.html, last accessed on 2020-12-15

[4] https://www.tensorflow.org/install/lang_c, last accessed on 2020-12-15

3.2.1 Pre-trained models

Next to the *tone transfer* website mentioned in the introduction, the authors of the DDSP paper also published a Jupyter Notebook Demo on Google Colab called *timbre transfer.* [5] We accessed the available checkpoint files for violin, flute, tenor saxophone and trumpet from this notebook for our real-time implementation of the timbre transfer. However, we were not immediately able to use them in the JUCE plugin. The DDSP models are trained using TensorFlow's *eager execution mode*, while the TensorFlow C API is constructed around *graph mode*. Additionally, since we required the models to be controllable by MIDI input, we needed direct access to the decoder part of the model instead of supplying audio to the encoder.

The `convert_models.py` script from the Python folder of the plugin code repository deals with these requirements by loading the eager model from the downloaded checkpoint file, constructing a graph-based model only containing the decoder and then copying all weights from the old model to the new one. The resulting checkpoint now contains a graph that can be loaded by the TensorFlow C API.

3.2.2 Custom models

In order to make use of the DDSP training library and extend the synthesizer with additional models, we created four custom models trained on:

- Bass sounds of the Moog One, Moog Minimoog and Moog Minitaur synthesizers
- Studio recordings of Middle Eastern instruments, the Hammered Dulcimer and Santoor
- Studio recordings of a Handpan (also known as Hang Drum)
- Nature field recordings of birds chirping

For training we used the official DDSP (version 0.14.0) Jupyter notebook on Google Colab called *train autoencoder* [6] which allows training on a Google Cloud GPU using own data. According to the recommendations of the DDSP authors given in the notebook, trained models perform best using recordings of a single, monophonic sound source, in one acoustic environment, in .wav or .mp3 format with a total duration of 10 to 20 minutes. Since the DDSP Autoencoder is conditioned on the loudness A and the fundamental frequency f_0, i.e., the model learns to associate different synthesizer configurations to specific value pairs of (A, f_0), training on multiple instruments, acoustic environments or polyphonic sounds prevents the autoencoder to learn a unified representation. Although the recordings listed above are less conform with these training guidelines, we chose them to challenge the DDSP autoencoder, exploring limitations and opportunities in a musical context by deliberately bending the recommended usage.

The training process is performed as follows. The first step is comprised of data generation and pre-processing of the training data. The raw audio is split into short parts of a few seconds, each analyzed on the specified features, i.e., the fundamental frequency and loudness, and finally saved in the TensorFlow *TFRecord* format. The fundamental frequency is thereby estimated by using the state-of-the-art pitch tracking technique, called *CREPE* by Kim et al. [9] that applies a deep convolutional neural network on time-domain audio.

The second step is the actual training, using a Python based configuration framework for dependency injection by Google, called *Gin* [7] . In this way, all available training hyperparameters can be defined in a gin config file that is passed to the training function. The training process does not include any optimization techniques, such as a hyperparameter search or early stopping, the authors just recommend in the code documentation to train for 5,000 to 30,000 steps until a spectral loss of about 4.5-5 is reached for an optimal learning representation without overfitting.

The third and last step was a short evaluation based on resynthesis. Here, a training sample was randomly picked, passed through the autoencoder, and checked if it was perfectly reconstructed based on the learned features.

We successfully conducted training of all four models and validated their performance in the previously mentioned timbre transfer demo. While validation using the DDSP library went smoothly and showed musically interesting results, we ran into issues during inference using the TensorFlow C API within our plugin. We monitored a much higher loudness of the custom models compared to the pre-trained models, resulting in a distorted, clipping sound. Furthermore, we detected a constant harmonic distribution independent of the incoming pitch and loudness while the pre-trained models adapt harmonics and frequency response according to these inputs. The overall experience with the training script provided by the DDSP authors is that it works without problems for standard parameters, but as soon as own hyperparameters within the gin framework are chosen, a lot of side-effects appear. For the mentioned reasons, integrating and possibly adapting the custom-trained models to make them work in the DDSP synthesizer will be a part of future work.

3.2.3 Real-time implementation of the models

The original DDSP implementation synthesizes several frames before processing them into one output. Reading through the DDSP code base, we experienced the number of frames (time steps) to be defined by the size of the input audio and a hop size defined by constants in the gin config file of the selected pre-trained model.

For our real-time implementation we wanted to calculate one frame with a size of the input buffer each time the buffer is ready. Given the static nature of our TensorFlow model implementation we were not able to change the number of time steps on the run. Therefore, we set the number of time steps to one. Each run of the TensorFlow model would then

[5] `https://colab.research.google.com/github/magenta/ddsp/blob/master/ddsp/colab/demos/timbre_transfer.ipynb`, last accessed on 2020-12-15

[6] `https://colab.research.google.com/github/magenta/ddsp/blob/master/ddsp/colab/demos/train_autoencoder.ipynb`, last accessed on 2020-12-15

[7] `https://github.com/google/gin-config`, last accessed on 2020-12-15

return a set of values for one time step, independent of the buffer size.

3.3 Additive synthesizer

The implementation of the additive synthesizer can be found in the `additive.m` MATLAB code file. During the development of the DDSP synthesizer we went from a re-implementation of the DDSP equivalent to an adapted real-time optimized version with additional parameters for high-level control. While the original DDSP library provides two different implementations of the additive synthesis, the harmonic and sinusoidal approach, this work focuses on the harmonic synthesis that models a signal by adding only integer multiples of the fundamental frequency.

In the following, the initial implementation as well as the main modifications in its final state are clarified. As already explained in 2.1, the additive synthesizer models audio using a bank of harmonic sinusoidal oscillators. The synthesis algorithm takes amplitudes, harmonic distribution and fundamental frequencies for a specified number of frames as input and computes the sample-wise audio signal as output. The harmonic distribution provides frame-wise amplitudes of the harmonics. The additive synthesis as implemented in the DDSP library is performed in two main steps: 1) Translation of neural network outputs to the parameter space of the synthesizer controls, and 2) Computing the output signal from synthesizer controls.

For 1), the amplitudes were scaled and the harmonic distribution was scaled, bandlimited (i.e., removing the harmonics that exceed Nyquist frequency) and normalized, while the fundamental frequencies remained unchanged. After retrieving valid synthesizer controls in step 1), the harmonic synthesis is performed. Since the DDSP approach works frame-based while the output needs to be delivered sample-based, the synthesizer controls need to be upsampled. This is done by linearly interpolating the frequency envelopes and windowing the amplitude envelopes by using 50% overlapping Hann windows. Having calculated all controls on a sample basis, the signal can be synthesized by accumulative summation of the corresponding phases, i.e., adding the calculated sinusoids together, sample by sample.

The following changes were made to optimize the algorithm for a real-time application and to add additional high-level control for the synthesis.

- Since the frame-based calculation was computationally too heavy, we adapted the code so that the input is always one frame (equivalent to the buffer size) and all computations are sample-based. Therefore, no resampling or windowing is needed.
- Each time the function is called, the phases of all harmonics are saved and returned along with the signal and added as offset in the next call to avoid artifacts caused by phase jumps.
- In order to be able to optionally introduce non-harmonic partials to the signal, a stretch parameter was added that transforms the distance between the integer multiples while maintaining the fundamental frequency. An additional shift parameter adds the functionality to modify the fundamental frequency from one octave below to one octave above the current pitch in a continuous scale.

3.4 Subtractive synthesizer

This component is responsible for the non-harmonic parts of instrument sounds, such as the audible non-pitched flow of air that accompanies the harmonic part of a flute sound. Our implementation, which can be found in the `subtractive.m` MATLAB code file, generates a frame of random noise and then filters it according to a given frequency response.

The function's parameters are the frame length (number of samples), noise color (see below) and the frequency response, which is given as a vector of N magnitudes $m_0, \ldots, m_{N-1}$, where m_0 corresponds to the DC component and m_i to frequency $f_{\text{nyquist}}/(N-i)$ with $f_{\text{nyquist}} = f_s/2$ and samplerate f_s.

While we started with a direct re-implementation of the DDSP FilteredNoise approach described in 2.2, we made the following adaptations over the course of the project:

- **Simplified filtering**: The DDSP synthesizer processes multiple frames at once. For real-time implementation, we removed the step of calculating the impulse response for each frame and applying a windowing function. Instead, we simply perform a Fourier transform on the generated noise and multiply the result with the filter magnitude response that the model predicted for the single current frame.
- **Noise color**: We provide functionality to shape the frequency distribution of the generated noise. Noise color generally refers to the frequency f being emphasized proportionally to $1/f^\alpha$ for some exponent α [10]. $\alpha < 1$ results in higher frequencies becoming more prominent, while $\alpha > 1$ increases the energy of the lower frequencies. Uniform white noise is achieved by setting $\alpha = 1$.

3.5 Graphical User Interface

After the development of all the features of our synthesizer, we focused our attention on designing an interface with high-level controls for the additive and the subtractive synthesis, the reverb, the modulation and the models. Our process started from a list of all the parameters we wanted to manipulate. We also looked for some inspiration from well-known VST synthesizers, comparing them in terms of usability and trying to understand what their best interaction features were. Later we organized the controls of our synthesizer in different modules and displayed them in a rectangular interface, trying to find a layout that was pleasant but also respectful of the instrument's architecture logic. In table 2, we list all the controls for each module of our synthesizer. Because of the particular choice of a graphic control for the harmonics' amplitude, the team opted for a spectrogram representing the output of our plugin. In this way, the user is able to clearly see which harmonics are being played.

Module	Feature controls
Input selector	MIDI/line selector
Models selector	Violin Flute Saxophone Trumpet Moog Bass (not included) Dulcimer (not included) Handpan (not included) Chirps (not included)
Additive synthesis	Graphic harmonics editor f_0 shift Harmonics stretching Global amplitude
Subtractive synthesis	Noise color Global amplitude
Modulation	Modulation rate Delay control Amount
Reverb	Dry/wet mix Size Glow
Output	Master gain
Spectrogram	Clear visualization of the output

Table 2. List of GUI's features

Once we defined the layout and the parameters that we wanted to control, we moved to the software development in JUCE. In order to customize the appearance of knobs, we used the "Custom LookandFeel" objects while we designed ad hoc images for the buttons and background texture using a vector graphics software. Figure 1 previously presented the GUI of our synthesizer.

3.6 Plugin setup

The synthesizer ended up being built as a standalone executable and a DAW plugin using Steinberg's VST3 format.

Using JUCE's `AudioProcessorValueTreeState` class we are exposing the different controllable parameters to the DAW, allowing control and automation of the plugin. Using this class we will also be able to easily store and read plugin states, enabling generation of presets, though this has not been implemented yet.

The synthesizer is configured to load the models from a given path with subfolders containing the individual models, as well as configuration files containing key-value pairs such as number of harmonics and scaling values.

4. EVALUATION

In order to understand the strengths and weaknesses of our product to improve it, we designed an evaluation strategy for both User Experience (UX) and sound output. Our target users are musicians and music producers. Accordingly, we shared a release of our VST plugin with selected sound engineers, musicians and producers to collect opinions and user insights. Moreover, we designed two different questionnaires and asked participants to evaluate the UX and the sound accuracy of our software. The DDSP Synthesizer as well as the two questionnaires have been distributed online and the participants received an email with all the indications to properly conduct the test.

In the following, we mainly describe the UX evaluation, including our approach, desired outcome, survey design and results.

4.1 User Experience Evaluation

4.1.1 Approach

The aim of this evaluation was to collect feedback about the user interface from people with experience on synthesizers and music production. One of the goals of our project was to design a simple and efficient interface able to control several parameters with a single gesture without giving up functionality in the pursuit of simplicity. After a trial period where the participants had the chance to familiarize themselves with the software, we asked them to complete a survey.

4.1.2 Survey structure

We designed the survey with different sections to group the questions by theme. We included an experiment in order to ask each participant to load and perform some changes to a model and export the result in an audio file. In this way, we ensured that every participant had at least used and interacted with the plugin for a while. Moreover we are able to compare each audio export to understand if some of the instructions were not clear or if the UX itself was not effective.

Four usage questions have been asked to collect information about the user's DAW and for how much time they used the plugin. In the next sections we asked the participants to report their experience during the experiment and evaluate the user interface rating 9 different statements with a Likert-scale, a widely used bipolar symmetric scaling method in questionnaires. In this way, users were able to express their agreement/disagreement related to each sentence. Furthermore, we asked 4 open questions to let the participants express their opinion about the overall UX. Finally we added 8 questions to locate demographics and musical-related personal experiences. Table 3 summarizes the content of each section.

#	Section	Content
1	Introduction	Aim of the questionnaire
2	Experiment	Task instructions
3	Usage	4 mixed questions
4	UX evaluation	9 Likert scale evaluations
5	UX experience	4 open questions
5	Demographics	8 mixed questions

Table 3. Content of the UX survey

4.1.3 Expected results

Considering that the software was still under development, we were expecting reports about compatibility issues with different DAWs as well as some stability problems. Moreover, because of the VST's instability in the first release, it is possible that some users will not be able to conduct the small experiment that requires the plugin to be embedded in a DAW track. Considering the whole interface, one of the main points of our design requirements was the simplicity and thus our hope is to facilitate the user's interaction. Even if the number of participants is limited, we expect that the users will approximately identify 75% of the UX issues accordingly to Nielsen's model [11].

4.1.4 Results

We received seven answers. Five participants identified as males, one female and one preferred not to say. The age average is 28.57 years (STD 8.42). Six of them declared that sound production is their hobby while one said music production is related to their job. The mean experience in the music production field is 7.43 years (STD 4.87). Six users do not have experience with machine learning VST plugins and only one of them does not know if she/he ever used one. Each user spent an average of 23.57 minutes using our synthesizer (STD 17.74). We suppose that some mistake has been made reporting the usage time for at least one user. In table 4 we report the number of user tests per different software environment.

# users	**Environment**
3	Reaper
2	Ableton Live
1	Cubase
1	Standalone version

Table 4. List of used DAWs in the evaluation.

In general, the experiment has been rated a medium difficult task with a mean rating of 3.43 in a scale from 1 to 5 being 1 "easy to accomplish" and 5 "hard to accomplish". In figure 3 we summarize the answers obtained from the questions with an associated Likert scale. The users were asked to rate each sentence from 1 to 5 with 1 corresponding to "strongly disagree" and 5 to "strongly agree". We can observe that the graphical user interface has been really appreciated with a 4.43 mean value while the interface's controls seem not to let the participants easily reach the wanted results. The other statements reported in the Likert section obtained a medium rating between 3 and 3.86 which might mean that the GUI is in general appreciated.

As expected, some of the participants encountered difficulties in the installation procedure of the VST3 plugin in both Windows and macOS environments while the standalone version seems to be more stable. Furthermore, three users reported an unsatisfactory audio result related to the presets obtained from models. Here we report part of one of the feedback: "*[...] It's possible to get some cool sounds but the default sound when you just start it is not so nice.*". On the other hand, the audio input feature was appreciated: "*[...] I think the audio input feature has a lot of potential and I caught myself experimenting with this a lot and loosing track of time.*". Two participants reported that the possible interaction with the interface for the additive synthesizer was not immediate to spot and they realized its features after a while. For this reason they suggest a graphical indication to guide the user to the interaction with the harmonic sliders. A significant outcome is the unexpected audio results that participants reported. Even though they described output sounds as "awkward", they highlighted the new creative way of producing unexpected sounds, finding the whole synthesizer experience engaging.

4.2 Real-time timbre transfer

Running DDSP decoder models in a real-time plugin is computationally feasible. As the demonstration [8] shows, the plugin does not exceed 20% CPU load on an AMD Ryzen 7 with a clock speed of 2.9 GHz. Similar results were measured on a MacBook Pro 2013, with a 2GHz Core i7 processor.

We found the quality of the timbre transfer in our real-time implementation below that of the demonstrations published by the Magenta team. Our converted models preserve some characteristics of the original ones, such as wind noises in the flute model, but do not accurately reproduce the timbre overall. We confirmed that on the level of a single frame, our models produce the same output as their original counterparts; will investigate and improve the quality in the future. Additionally, we would like to further investigate why we were unable to perform the timbre transfer with models that we trained both within the framework provided by Magenta, and within custom environments.

4.3 Distribution as a VST3 plugin

When it came to distributing our project to users, we encountered some difficulties in packaging the required libraries and model files together with the generated VST3 plugin. Some of the DAWs that users tested on, like Ableton or Reaper, did not recognize the plugin or experienced stability issues during its usage. Although the core functionality could still be accessed via the standalone application generated by JUCE, the project was designed first and foremost as a plugin. Functionality like handling of external audio sources and wet/dry mixing was expected to be handled by the host DAW. Users who had to resort to the standalone when their DAW did not recognize or stably run the plugin reported those features as missing.

Thus, we would like to improve the distribution process in the future, ensuring that the project can be seamlessly installed as a plugin in multiple DAWs on Windows and macOS.

5. CONCLUSION

In this paper, we presented an approach to integrate the DDSP library into a real-time plugin and standalone application using the JUCE framework. We succeeded in

[8] `https://share.descript.com/view/hXAZLCPJNqm`

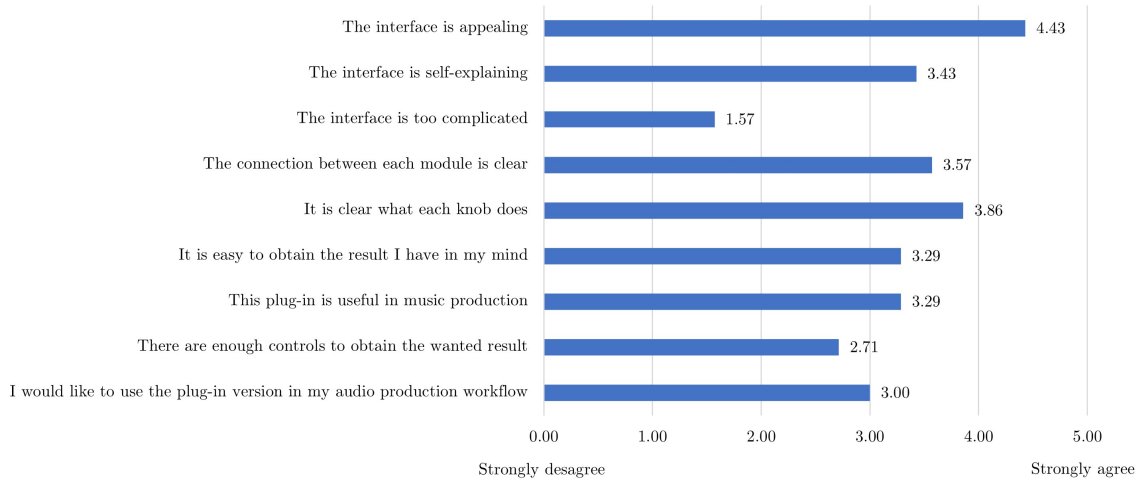

Figure 3. User experience evaluation - Likert scale

implementing a synthesizer playable based on pure user input. While we were generally able to use the output from pre-trained models to control the DDSP backend, further research is needed to match the sound quality of these real-time models to that of the offline timbre transfer examples provided by the DDSP authors.

A recently released realtime reimplementation of DDSP in PyTorch [9] provides a possibly more seamless way of interfacing with DDSP models in C++ that proved compatible with our plugin and JUCE. Extending that API to allow the user some control over the synthesis parameters seems a promising avenue to improve the sound quality of our timbre transfer.

6. REFERENCES

[1] C. Donahue, J. McAuley, and M. Puckette, "Adversarial Audio Synthesis," *arXiv:1802.04208 [cs]*, Feb. 2019, arXiv: 1802.04208. [Online]. Available: http://arxiv.org/abs/1802.04208

[2] M. Blaauw and J. Bonada, "A neural parametric singing synthesizer modeling timbre and expression from natural songs," *Applied Sciences*, vol. 7, p. 1313, 2017.

[3] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable Digital Signal Processing," *International Conference on Learning Representations*, 2020.

[4] X. Serra and J. O. Smith, "Spectral modeling synthesis. A sound analysis/synthesis system based on a deterministic plus stochastic decomposition," *Computer Music Journal*, vol. 14, no. 4, pp. 12–24, 1990.

[5] Native-Instruments, "Razor," 2011. [Online]. Available: https://www.native-instruments.com/en/products/komplete/synths/razor/

[6] D. Pandey, U. Suman, and A. Ramani, "An effective requirement engineering process model for software development and requirements management," in *2010 International Conference on Advances in Recent Technologies in Communication and Computing*, 2010, pp. 287 – 291.

[7] A. Cheveigné and H. Kawahara, "YIN, A fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, pp. 1917–30, 2002.

[8] P. Brossier, "Automatic annotation of musical audio for interactive applications," Ph.D. dissertation, Queen Mary University of London, 2006.

[9] J. W. Kim, J. Salamon, P. Li, and J. P. Bello, "CREPE: A convolutional representation for pitch estimation," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 161–165.

[10] N. J. Kasdin, "Discrete simulation of colored noise and stochastic processes and $1/f^{\alpha}$ power law noise generation," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 802–827, 1995.

[11] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1990, pp. 249–256.

[9] https://github.com/acids-ircam/ddsp_pytorch