Vocoding with Differentiable Digital Signal Processing: Development of a Real-Time Vocal Effect Plugin

David Südholt
Master's Thesis in Sound and Music Computing at Aalborg University Copenhagen
dsudho20@student.aau.dk

ABSTRACT

This project explores two approaches to creating vocal effects using Differentiable Digital Signal Processing (DDSP).

In the first approach, a pretrained DDSP decoder is used to generate a signal based on pitch and loudness information extracted from the incoming vocal content. The generated signal and the vocal input are then blended together in the spectral domain to preserve the phonetic content.

In the second approach, autoencoder models are trained on datasets consisting of both vocal and instrument training data. To apply the effect, the trained autoencoder reconstructs the vocal input. The latent space is able to encode information about the phonetic content, while the inclusion of other instruments in the training data affects the timbre of the reconstructed signal.

A real-time vocal effect plugin based on the first approach is designed and implemented. The two methods are compared through a perceptual evaluation.

1. INTRODUCTION

Synthesizing and manipulating the sound of the human voice through signal processing methods has a long history. Homer Dudley's "Voice Operating Demonstrator" (Voder) was introduced in 1939 [1] and musical applications of the same principle soon followed [2]. Using a series of bandpass filters, analog devices like the Moog Vocoder [1] could analyze the spectral content of an incoming modulator signal, and process an incoming carrier signal through the same filterbank to output a hybrid signal. When a synthesizer is used as the carrier and a singing or speaking voice as the modulator, this creates the "robot voice" effect made famous by artists such as Kraftwerk, Daft Punk and Electric Light Orchestra.

When the effect is implemented digitally, it is common to make use of the forward and inverse Fourier transform as the analysis-synthesis framework instead of bandpass filters [3, 4].

Neural network models have seen immense success in countless fields of study over the past decade, and the sound and music computing field is no exception. WaveNet [5] was able to generate realistic-sounding speech and music by generating raw audio sample-by-sample. Used in an autoencoder architecture, it allowed seamless morphing between different instrument timbres [6].

[1] https://www.moogmusic.com/news/moog-vocoder-returns

Differentiable Digital Signal Processing (DDSP) [7] moved away from directly generating audio in the time or spectral domain, and achieved impressive results in multiple tasks by predicting controls for a library of synthesizer components. This carries the added advantage of directly interpretable model output.

In this project, the potential of creating vocal effects by combining DDSP with classic vocoding techniques is explored and implemented as a real-time vocal effect plugin. Additionally, methods of exploiting latent variables of DDSP models as vocal effects are investigated.

The rest of this report is structured as follows. Section 2 gives an overview of the relevant background concerning vocoding methods and DDSP. Section 3 describes the two approaches to creating vocal effects in detail. Section 4 covers the design and implementation of the plugin. A perceptual evaluation of selected effects is performed and discussed in section 5. Section 6 concludes the report.

2. BACKGROUND

2.1 Vocoding

In the context of musical applications, *vocoding* typically refers to the process of cross-synthesis, where the modulator is a vocal signal and the carrier typically a spectrally rich synthesized signal, such as a saw wave. Vocoders can be characterized as either *channel vocoders*, which is the typical approach used by analog devices, or *phase vocoders*, which use digital signal processing (DSP) methods.

2.1.1 Channel Vocoder

The channel vocoder can be traced back to Homer Dudley's Voder [1]. It follows an encoder-decoder architecture. The encoder processes a speech signal through a number of bandpass filters (*channels*), whose center frequencies are chosen to cover the frequency range relevant to understanding human speech. The output of each channel is measured by an amplitude follower. The decoder uses a noise source or an oscillator rich in harmonics to generate a carrier signal. The carrier is filtered through the same channels, and the output of each channel is multiplied with the corresponding amplitude extracted by the encoder. All channels are then added together to reconstruct the speech signal. This initially found use in transmitting and encrypting speech, since only the channel amplitudes need to be communicated instead of the entire signal.

2.1.2 Phase Vocoder

The phase vocoder segments a digital input signal into overlapping windows, which are transformed to the spectral domain via the Fast Fourier Transform (FFT), yielding a time-varying spectrum. The spectral information - phase and amplitude - can then be modified, before the spectra are converted back to the time domain via the inverse FFT (IFFT). The output signal is obtained by overlapping-and-adding the resulting windowed output segments. A popular application is found in time-scale modification, which is achieved by manipulating the phase derivatives before resynthesis.

The phase vocoder is somewhat related to the channel vocoder, since the FFT can be interpreted as a bank of complex bandpass filters. This view also allows to perform a perfect resynthesis via a sum of sinusoids, one for each of the bins of the FFT. Vocoding in the musical sense can be achieved by manipulating the amplitudes of the sinusoids according to the spectral content of a modulator signal.

2.2 Differentiable Digital Signal Processing

Differentiable Digital Signal Processing (DDSP) [7] proposes an end-to-end learning approach for neural audio synthesis. Instead of generating signals sample-by-sample in the time domain, or time-varying spectra in the frequency domain, DDSP offers a library of synthesizer components implemented entirely within a framework supporting auto-differentiation. A variational autoencoder (VAE) model is trained to encode audio into pitch, loudness and timbre information, and to reconstruct it by generating time-varying control parameters for the synthesizers. The loss is calculated by comparing the spectrogram of the generated audio from the synthesizers to that of the original audio on multiple timescales. The auto-differentiable implementation allows the gradient of the loss to backpropagate through the synthesizers to update the model weights of the VAE.

2.2.1 Model Architecture

Figure 1 details how the VAE and the synthesis components work together to reconstruct audio. An encoder extracts pitch and loudness information, and optionally a latent timbre encoding z, from the input. In the original paper, the pitch is detected using a pretrained CREPE [8] model, whose weights are frozen during training. The loudness calculation is based on the root-mean-square (RMS) value.

The decoder predicts control parameters for the synthesizer components. The decoder architecture most commonly used in the original paper consists of a stack of fully connected layers for each of the inputs (pitch, loudness, optionally z). The outputs of the stacks are concatenated and processed through a recurrent neural network (RNN) layer and finally projected into the control parameter space.

2.2.2 Synthesis Components

DDSP was conceived as a modular library, so the choice of synthesis method is independent from the model architecture. This project focuses on the harmonic + noise + reverb synthesizer structure that was used in the original DDSP paper, which is based on the concept of Spectral Modeling Synthesis (SMS) [9]. A synthesis method based on a dictionary of learnable wavetables has recently been shown to be effective [10].

In the SMS framework, the harmonic components of the sound are generated by a sum of K sinusoids, where K is a manually chosen hyperparameter. The decoder predicts K time-varying amplitudes $A_k(n)$, referred to as the *harmonic distribution*, since the sinusoids are defined to oscillate at integer multiples of the (also time-varying) fundamental frequency $f_0(n)$ extracted by the encoder. Thus, the output of the harmonic component x_h can be formulated as

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

where $a(n)$ is a global amplitude predicted by the decoder, and ϕ_k is the instantaneous phase of the k-th harmonic, given as

$$\phi_k(n) = 2\pi \sum_{m=0}^{n} k f_0(m) \,. \tag{2}$$

Filtered noise is used to generate the non-harmonic components of the sounds. The decoder predicts the time-varying magnitude responses of a finite impulse response (FIR) filter. The magnitudes are transformed into the time domain through an IFFT, where the resulting impulse responses (IRs) are windowed. White noise is generated and filtered through convolution with the IR, implemented as multiplication in the frequency domain.

The final component of the synthesizer structure is a learned IR for convolution reverb. The IR is not time-varying and thus not part of the parameters predicted by the decoder. Dereverberation can then easily be achieved by omitting the reverb component during reconstruction.

In summary, the synthesizer controls generated by the decoder are a time-series of global amplitude, harmonic distribution, and filter magnitudes. The pitch detected by the encoder is an additional parameter to the synthesis. A single *time step* is used to generate a manually defined number of samples.

2.2.3 Timbre Transfer

Timbre transfer was one of the use cases in the original DDSP paper that garnered the most attention. A decoder solely conditioned on pitch and loudness features is trained on audio recordings of a specific instrument, e.g. a violin. After training has completed, extracting pitch and loudness from any input audio can be used to generate the same melody line in the sound of a violin by using the trained decoder to predict corresponding synthesizer controls. A web demonstration [2] of this principle was made available online by Google Magenta.

2.2.4 Applications to Speech and Singing Synthesis

The model architecture depicted in figure 1 was shown to be capable of synthesizing a human singing voice using a

[2] `https://sites.research.google/tonetransfer`

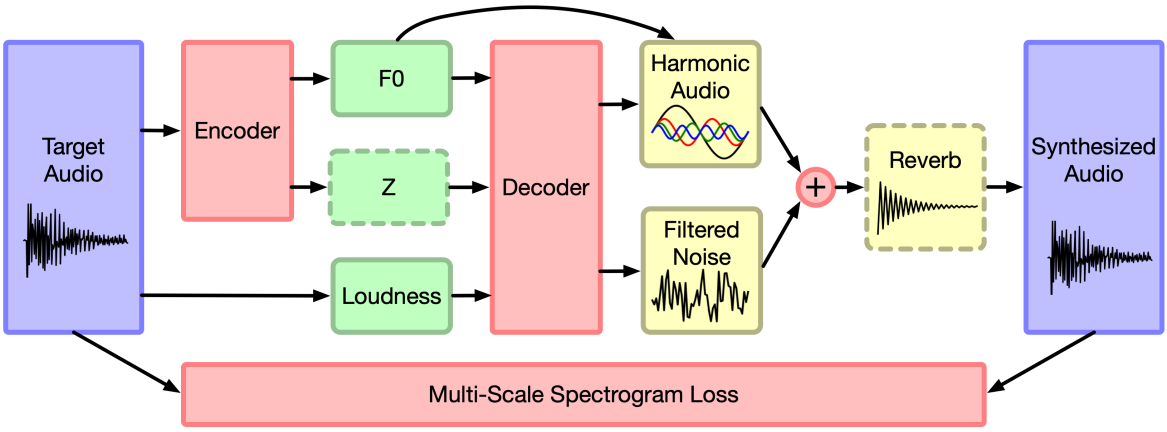

Figure 1: DDSP autoencoder architecture (taken from [7]).

latent z encoding generated from Mel Frequency Cepstral Coefficients (MFCCs) processed through an RNN layer [11].

In [12], explicit phonetic information was added to the decoder input, resulting in an effective algorithm for voice conversion following the same principles as in the timbre transfer task.

2.3 Real-Time Neural Audio Synthesis

While neural audio synthesis has made great progress over several years, high-fidelity real-time applications have only emerged very recently, but the field seems to move at a very quick pace.

The original DDSP release written in TensorFlow worked in a strictly offline manner, although a PyTorch reimplementation capable of running in real-time as a PureData external was quickly made available [3]. An interface for interacting with the synthesis parameters in real-time was proposed in [13]. Since then, the library has been extended by decoder architectures meant for online synthesis such as in a VST plugin. The spring of 2022 saw beta releases of two plugins capable of performing timbre transfer in real-time: Mawf [4] and DDSP-VST [5].

At the same time, a real-time implementation [14] of the RAVE autoencoder [15] was released, which is based on generating the time-domain audio waveform at multiple frequency bands, combined with a DDSP-like filtered noise synthesizer.

3. VOCAL EFFECTS WITH DDSP

This section describes the main two approaches to generating vocal effects with DDSP that were explored in this project.

The vocoding approach is based on utilizing timbre transfer to generate control parameters for the synthesizer components, which are then modified with vocoding-like methods to preserve the lyrical content of the input.

[3] `https://github.com/acids-ircam/ddsp_pytorch`
[4] `https://mawf.io/`
[5] `https://magenta.tensorflow.org/ddsp-vst`

In the latent encoding approach, the encoder calculates MFCCs from the input and processes them through an RNN to generate the latent variable z. The decoder-generated control parameters are not modified in this approach, and no traditional vocoding methods are used. Effects are achieved through manipulation of the latent variable and the training data.

Jupyter notebooks for inference with both approaches are supplied, named `Vocode` and `Latent`.

3.1 Vocoding Approach

For the vocoding approach, pitch and loudness are extracted from the vocal input. The decoder will then output a global amplitude and a harmonic distribution A_k for $k = 1, \ldots, K$, which are used to generated the harmonic output as described in equation (1), and filter magnitudes. Let

$$V_m = \sum_{n=0}^{N} v(n) e^{-i2\pi \frac{m}{N}} \tag{3}$$

denote the N-point short-time Fourier transform of the windowed vocal input $v(n)$ of length N at the current time step.

A_k is the amplitude of the k-th harmonic, i.e. an oscillator with frequency $k \cdot f_0$. The spectral information of the vocal input corresponding to the frequency $k \cdot f_0$ is contained in the frequency bin V_{m_k}. The index m_k can be calculated as

$$m_k = \left\lfloor \frac{k \cdot f_0}{f_s} N \right\rceil , \tag{4}$$

where f_s is the sampling rate of the vocal input.

The idea of the vocoding approach is to obtain a modified harmonic distribution A'_k by interpolating between the vocal input amplitude spectrum $|V|$ and the decoder-generated harmonic distribution A_k. To that end, two user-supplied control parameters are introduced: An interpolation factor $p \in [0, 1]$ and the number of neighboring bins $B \in \mathbb{N}_0$ that will also be taken into account. The modified distribution can then be calculated as

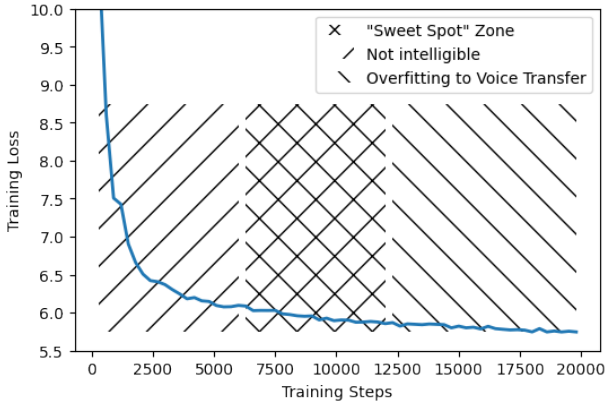

Figure 2: The training process of a latent encoding model on the CSD + Brass dataset (cf. section 3.3.1). Early during training, it cannot reconstruct intelligible lyrics yet. Then it transitions into the "sweet spot" where lyrical content is preserved, but the timbre is affected by the additional instrument. After further training, that effect disappears, and the model performs regular voice transfer.

$$A'_k = \begin{cases} (1-p)A_k + \dfrac{p}{2B+1}\displaystyle\sum_{b=-B}^{B} |V_{m_k+b}| & kf_0 < \frac{f_s}{2} \\ 0 & \text{else} \end{cases}, \tag{5}$$

taking care not to include oscillators at frequencies exceeding the Nyquist limit of $f_s/2$. Note that for $p = 0$, $A'_k = A_k$.

3.2 Latent Encoding Approach

In the latent encoding approach, the encoder generates a vector z in addition to pitch and loudness information. This project used an encoder provided in the DDSP library that calculates MFCCs of the input audio at every time step, and processes them through an RNN before projecting them to the latent space.

In this approach, no modifications are applied to the decoder output. Instead, the effect is generated through selection of the training datasets. As explored in [11] and confirmed through preliminary experiments, simply training a VAE as shown in Figure 1 on recordings of a singing voice can be sufficient to obtain a model capable of reconstructing a vocal input from a different singing voice in the style of the training data with intelligible lyrics.

The idea behind this approach is to add in other monophonic instruments, such as a trumpet or a synthesizer, to the training data, in the hopes that it will influence the timbre of the reconstructed vocals in musically interesting ways.

During the experiments, it became clear that if the model is trained until the training loss converges, the decoder learns to distinguish between vocal input and the additional instrument, and is able to reconstruct both accurately. This results in a model that is effectively just performing voice transfer.

However, there seems to be a "sweet spot" early on in training, where the model is already able to reproduce the lyrical content of the input, but has not yet learned to fully distinguish between the different input sources. At this point, the timbre of the reconstructed vocals is affected noticeably by the additional instrument. This is illustrated in Figure 2.

Another explored approach was to add an explicit label $l = 0, \ldots, L$ to the training data indicating whether it is a recording of a singing voice or a specific instrument, with L being the number of different sources in the total dataset. The model was then extended to include L different encoders with their own sets of weights, but a single shared decoder. During training, the label of the training data determined which encoder would generate the latent encoding. During inference, both the vocal encoder and an instrument encoder would be used to generate the encoding, in the hopes that interpolating between them would lead the decoder to generate lyrically intelligible vocals with a modified timbre.

In preliminary experiments however, the different encoders produced embeddings sufficiently close to each other that interpolation barely affected the sound.

3.3 Experiments

The nature of the project does not lend itself to a traditional evaluation of model performance using cross-validation and a hold-out test set. There is no "ground truth" by which to evaluate the models when the goal is the highly subjective creation of a "musically interesting" effect. Accordingly, the training runs were mostly evaluated by manual inspection of the resulting audio.

3.3.1 Datasets

Data for vocal performances was collected from the Children's Song Dataset (CSD) [16] and from the MUSDB18 corpus [17]. A voice transfer model was trained on a single performer from the CSD corpus. From the MUSDB18 corpus, shorter excerpts from multiple singers from the medley recordings named "Music Delta" were compiled and used to train models on joint datasets with instrument recordings.

Brass, string, and wind instrument datasets were created by combining respective instrument recordings taken from the University of Rochester Multi-Modal Music Performance dataset (URMP) [18]. Additionally, a synthesizer performance was obtained by processing randomized MIDI at varying velocities and pitches through the "Harsh Lead 1" preset of the Helm synthesizer [6].

The recordings were transformed into DDSP-readable datasets using the `ddsp_prepare_tfrecord` command.

3.3.2 Training

For the vocoding approach, models are trained using the standard timbre transfer setup presented in the original DDSP paper, where an estimate of 30k-50k training steps is given as a good balance between accuracy and overfitting. Training hyperparameters such as learning rate were left to the default settings of the DDSP library, version 3.2.0.

[6] `https://tytel.org/helm/`

Training models for the latent encoding approach also followed the default settings found in the `ae.gin` file of the DDSP library. The multi-encoder experiments required a minor custom subclassing of data provider and encoder classes to implement the labeling and encoder selection, which can be found in the supplied file named `MultiEncoderClasses.py`.

4. REAL-TIME PLUGIN

Based on the vocoding approach, a real-time vocal effect plugin was designed and implemented using the JUCE framework [7]. The reason for only including the vocoding approach in the plugin was that the DDSP library at version 3.2.0 provides a mechanism for training decoders for online synthesis (i.e. generating synthesis parameters a single step at a time), but not for the MFCC encoders which are necessary for the latent encoding approach.

The plugin can be built for Windows and Mac environments by opening the provided `.jucer` file in the Projucer application, which is the standard way to generate build configurations for JUCE projects.

4.1 User Interface

A screenshot of the user interface is shown in Figure 3. The controls are grouped by function.

4.1.1 Pre- and Postprocessing

The plugin offers an input noise gate to remove unwanted background noise. After the vocoded audio is generated, the user can choose to apply dynamic range compression, controllable by the standard parameters of threshold, ratio, attack and release. Finally, the overall output volume can be adjusted.

4.1.2 Synthesis

The *Synthesis* group of controls affects the generated carrier signal. By default, a preset, fixed harmonic distribution is used. The "Load Model" button opens up a file selection dialog to load a JSON file containing weights of a DDSP decoder exported with the provided "Save to JSON" Jupyter Notebook. While a model is loaded, the button text (and its functionality) is changed to "Unload Model".

While no model is loaded, random noise is still generated when unvoiced input is detected, to be able to preserve unvoiced consonants. The volume of this noise can be controlled by adjusting the "Unvoiced Strength" parameter.

The "Harmonic Sensitivity" controls a volume threshold that the input needs to cross to be considered "harmonic" and affect the pitch of the synthesized signal.

4.1.3 Vocoding

The *Vocoding* group offers two controls to insert the lyrical content of the vocal input into the signal generated in the *Synthesis* section. "Vocal Clarity" can be interpreted as the strength of the vocoding; at high values, the input signal will dominate, at low values, little of the input will be audible in the final output.

[7] `https://juce.com/`

The "Formant Shift" control is used to shift the center frequencies used during vocoding. This can be used to give the impression of a "higher" or "lower" voice without altering the pitch.

4.1.4 Tuning

The *Tuning* group offers controls for explicit pitch shifting as well as an automatic pitch correction functionality. When the "Enable Tuning" box is ticked, the pitch extracted from the input vocals will be adjusted to lock to the chosen scale before the carrier signal is generated. Twelve major and minor scales are available, as well as a "chromatic" option. The "Tuning Attack" control affects how aggressively the pitch will be corrected. A setting of 0 ms immediately locks the pitch to the scale, while higher settings apply the correction more gently.

"Shift Input Pitch" will be applied to the detected pitch before the correction, "Shift Output Pitch" after the correction.

4.2 Implementation

The bulk of the processing is done in the `HumanoidEngine` class. The engine provides a `processBlock` method, which is repeatedly called from the plugin host and delivers a chunk of audio samples to process. Algorithm 1 provides an overview of the method's structure.

Algorithm 1 The main processing algorithm.

```
Copy the received samples into the input buffer
while input buffer size ≥ w_a do
    Calculate pitch and RMS of the current analysis window
    Perform FFT on input
    if model is loaded then
        Calculate power based on RMS
        Call decoder to generate controls
        Set harmonic distribution to decoder output
        Generate w_s samples of noise filtered with generated magnitudes
    else
        Set harmonic distribution to chosen preset
        if current analysis window is unvoiced then
            Generate w_s samples of noise and perform vocoding with the input magnitudes
        end if
    end if
    Add w_s samples of harmonic output to the noise, based on harmonic distribution and amplitude spectrum of the input
    Advance the input buffer by w_s samples
end while
Fill received buffer with samples from output buffer
```

The analysis frame size w_a (referred to in code as the `frameSize`) and the synthesis frame size w_s (referred to in code as the `hopSize`) are determined by the how the loaded decoder was trained, or set to a fixed default value while no model is loaded.

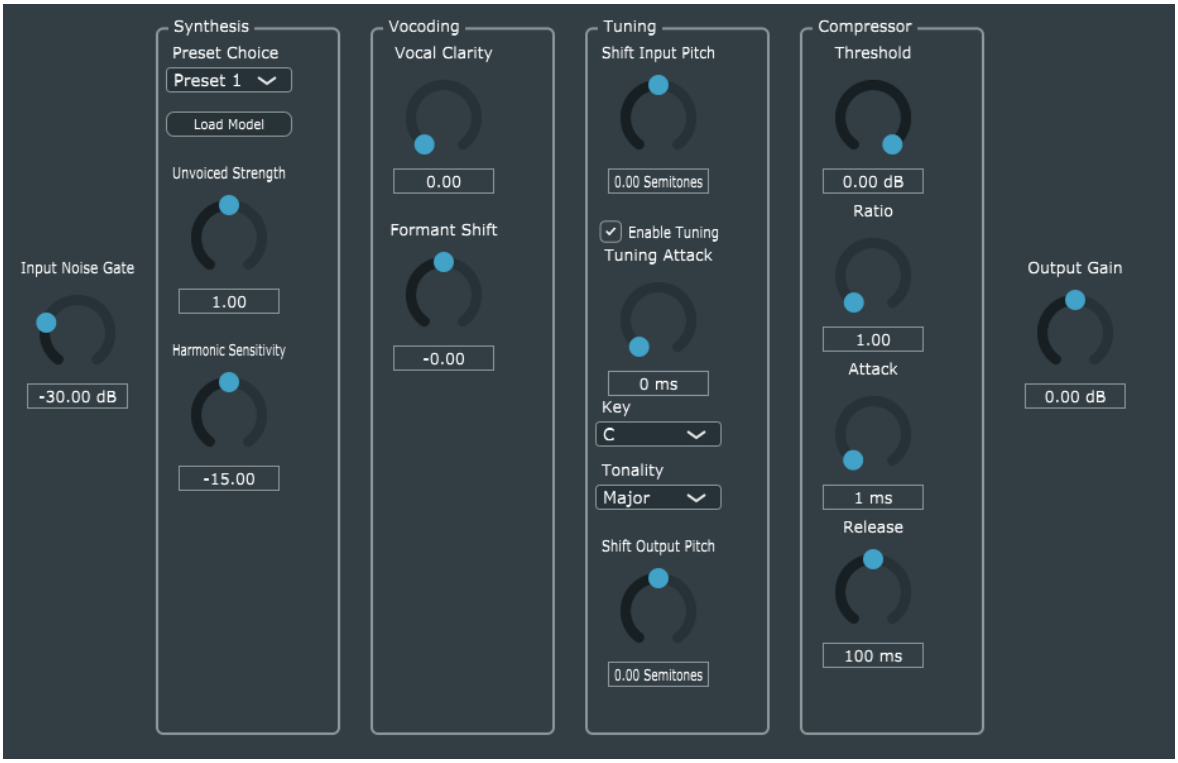

Figure 3: User interface of the plugin.

To account for variable sizes of the buffer passed to the `processBlock` method, which might be changed at any time depending on the run-time environment of the plugin, the processing is decoupled from the externally received buffer. Instead, the input is written to and retrieved from an internal circular buffer. Similarly, the output is written to an internal buffer in chunks of size w_s, from which the externally received buffer is filled as needed.

4.2.1 Pre- and Postprocessing

For the noise gate, compressor, and output gain, standard implementations offered by the JUCE library are used. Therefore, the plugin only needs to expose the respective parameters to the user interface and the plugin host. These pre- and postprocessing steps are applied to the input and output of the engine, rather than by the engine itself.

4.2.2 Loading DDSP Decoders

To enable running pretrained decoders inside of the plugin, the inference operations of an `RnnFcDecoder` as provided by the DDSP library were implemented using the Eigen linear algebra library [19].

This required the implementation of two main types of layers. *Fully connected layers* as used in the DDSP library refer to the following sequence of operations:

1. Dense layer: Multiplication with a weight matrix followed by addition with bias vector
2. Layer normalization: Subtract the mean, divide by the variance, followed by the application of a learned element-wise linear function
3. Application of the Leaky ReLU nonlinearity

The RNN structure used in all models is a Gated Recurrent Unit (GRU). Omitting biases for notational simplicity, at time step t the GRU uses its previous output h_{t-1} and the current input vector x_t to produce the output h_t according to the following operations:

$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{t-1}) \tag{6a}$$
$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{t-1}) \tag{6b}$$
$$n_t = \tanh(W_{in}x_t + r_t * (W_{hn}h_{t-1})) \tag{6c}$$
$$h_t = (1 - z_t) * n_t + z_t * h_{t-1} \tag{6d}$$

Here, r_t, z_t and n_t refer to the "reset", "update" and "new" gate, respectively. σ is the logistic sigmoid function and $*$ denotes element-wise multiplication. All weight matrices W are learnable parameters.

The `Decoder` class in the `nn` folder uses these layers to implement the decoder architecture as described in section 2.2.1. The `DecoderLoader` is responsible for reading the JSON file containing the weights and creating a decoder with the appropriate dimensions – number of layers, hidden nodes etc. The JSON file also contains information about the sampling rate, frame rate and hop size that the decoder was trained to operate with, which is important to performing the synthesis accurately.

4.2.3 Synthesis and Vocoding

The pitch of the current analysis window is estimated using the YIN algorithm [20]. Rather than using a fixed confidence threshold for the pitch detection, the window is treated as not containing voiced content if the detected

pitch is outside of human singing range [8].

The harmonic distribution either comes from the decoder output, if one is currently loaded, or one of two presets. In either case, the vocoding is performed as described in section 3.1, where the interpolation parameter p from equation (5) is controlled by the "Vocal Clarity" knob. The "Formant Shift" knob is implemented by introducing a scaling factor α into the index calculation from equation (4):

$$m_k = \left\lfloor \frac{\alpha k \cdot f_0}{f_s} N \right\rfloor \tag{7}$$

If a decoder is loaded, the sampling rate it was trained with determines the Nyquist limit, but the plugin sampling rate is used for the index calculation. The hop size and plugin sampling rate together determine the amount of samples produced per frame by the `HarmonicSynth`. The filtered noise on the other hand is generated with respect to the model's original sampling rate and then resampled to the plugin sampling rate.

To avoid clicks and discontinuities, the `HarmonicSynth` stores the harmonic distribution from the previous time step. When called with a new distribution, it linear interpolates between the two sample-by-sample.

If no decoder is loaded, noise is only generated when the input is determined to be unvoiced. In that case, the amplitude spectrum of the vocal input is used as the filter for the noise, so that the specific consonant is preserved.

4.2.4 Tuning

Tuning is implemented as simply modifying the recognized pitch before synthesis. Locking to a scale is done by linearly scaling the recognized pitch to the closest note belonging to that scale in MIDI space.

5. EVALUATION

Four models were chosen for a perceptual evaluation of the created effects:

VC-Synth: Timbre transfer model trained on the synth dataset, vocoding approach

VC-Brass: Timbre transfer model trained on the brass dataset, vocoding approach

Z-Vocals: Latent encoding voice transfer model trained on the CSD

Z-Mixed: Latent encoding model trained on a mixed dataset from the MUSDB18 medley vocals and the synth dataset

5.1 Listening Test Design

Two vocal samples, one performed by a male, one by a female singer, were processed by all four models. Although not necessarily matching the intended use case, the Multiple Stimuli with Hidden Reference and Anchor (MUSHRA) format was determined to be a good fit for the perceptual evaluation, due to the availability of a user-friendly implementation as a web application [21].

Participants performed the test [9] without supervision. At the start of test, participants were asked to adjust their volume to a comfortable level. Each of the two samples, together with its four processed versions, was then presented to the participants three times. Every time, they were asked to compare the (unlabeled) recordings under a different aspect using the interface shown in Figure 4. The three aspects are:

1. Perceived audio quality
2. Intelligibility of the lyrics
3. How musically interesting the effect is

After completing the test, participants were asked about their age, their experience with music and vocal production, and the listening environment.

5.2 Results

A total of 15 participants took part in the test. The general participant demographic consisted of university students from Denmark, Germany and Estonia. 9 of the participants reported no experience with music or vocal production. Ages ranged from 21 to 38, with a mean age of 25.9 years. The listening devices were distributed rather evenly between in-ear headphones, over-ear headphones, and speakers. Figure 5 plots the results of the 6 total evaluations.

The clearest result can be found in the rating of the lyrical intelligibility aspect on the female input sample, where the latent encoding models clearly outperform the vocoding models. The same trend, although to a lesser degree, is shown in the evaluation of the male input sample. This seems to confirm that the MFCC + RNN encoder is already capable of reproducing intelligible lyrics without any explicit phonetic information. The Z-Vocals model performs noticeably better under this aspect on the female than the male input; a likely reason for this difference is that the model was trained exclusively on female performances.

None of the models are rated particularly favorably under the aspect of perceived audio quality, although the latent encoding models perform slightly better than the vocoding models. All of the evaluated models were trained with a sampling rate of 16 kHz. A comparison with models trained on higher-fidelity audio would be desirable; however, some experimentation showed that training at higher sampling rates is much more computationally intensive and prone to instability. More time would be needed to obtain usable high-fidelity models.

Unsurprisingly, the highly subjective rating according to "musical interest" shows the highest variance of the ratings, although a slight trend favoring the latent encoding models seems to exist.

[8] Arbitrarily defined to be between 100Hz and 700Hz, roughly corresponding to the notes G_2 and F_5

[9] available online at `http://dsuedholt.de`

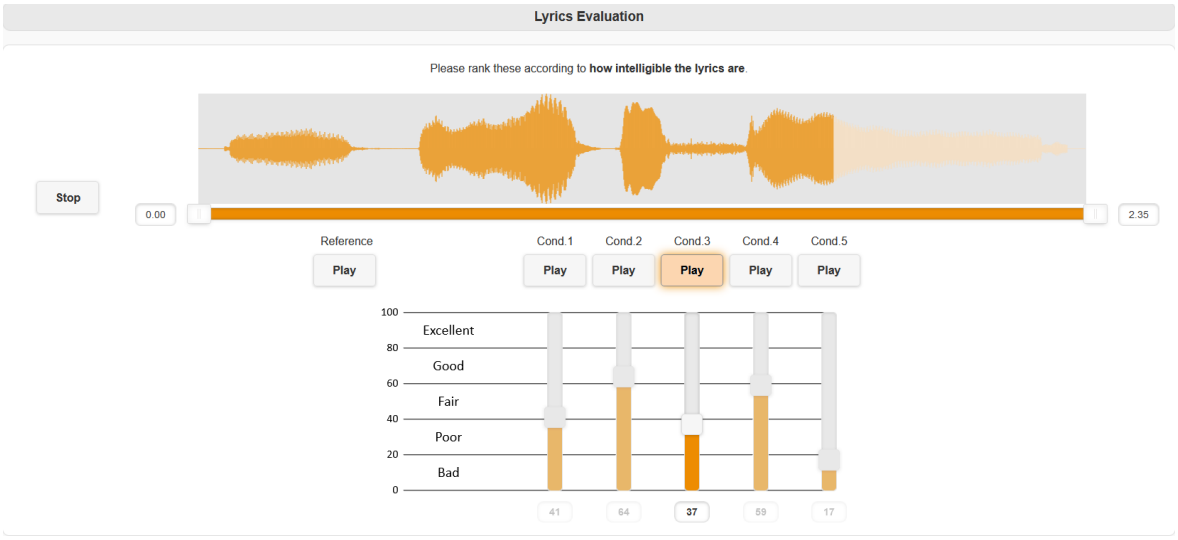

Figure 4: The webMUSHRA interface used for the evaluation.

6. CONCLUSION

Two methods of producing vocal effects with the DDSP library were designed, implemented and perceptually evaluated. A vocal effects plugin capable of performing real-time timbre transfer as well as the vocoding-inspired vocal effect was designed and implemented.

While the vocoding approach is more straight-forward and lends itself to easily explainable control parameters, such as the interpolation and formant shift parameters, the evaluation suggests that the latent encoding approach is perceived to be of higher quality and to preserve the lyrical content to a higher degree. Following the success of RAVE, this would suggest that the next step for this project could be to extend the real-time plugin to include models implementing the latent encoding.

Additionally, the latent space potentially offers the possibility for creative controls affecting the sound. While the multi-encoder approach explored in this project was unsuccessful, further experimentation with varying architectures and datasets seems promising to develop interesting DDSP-based vocal effects.

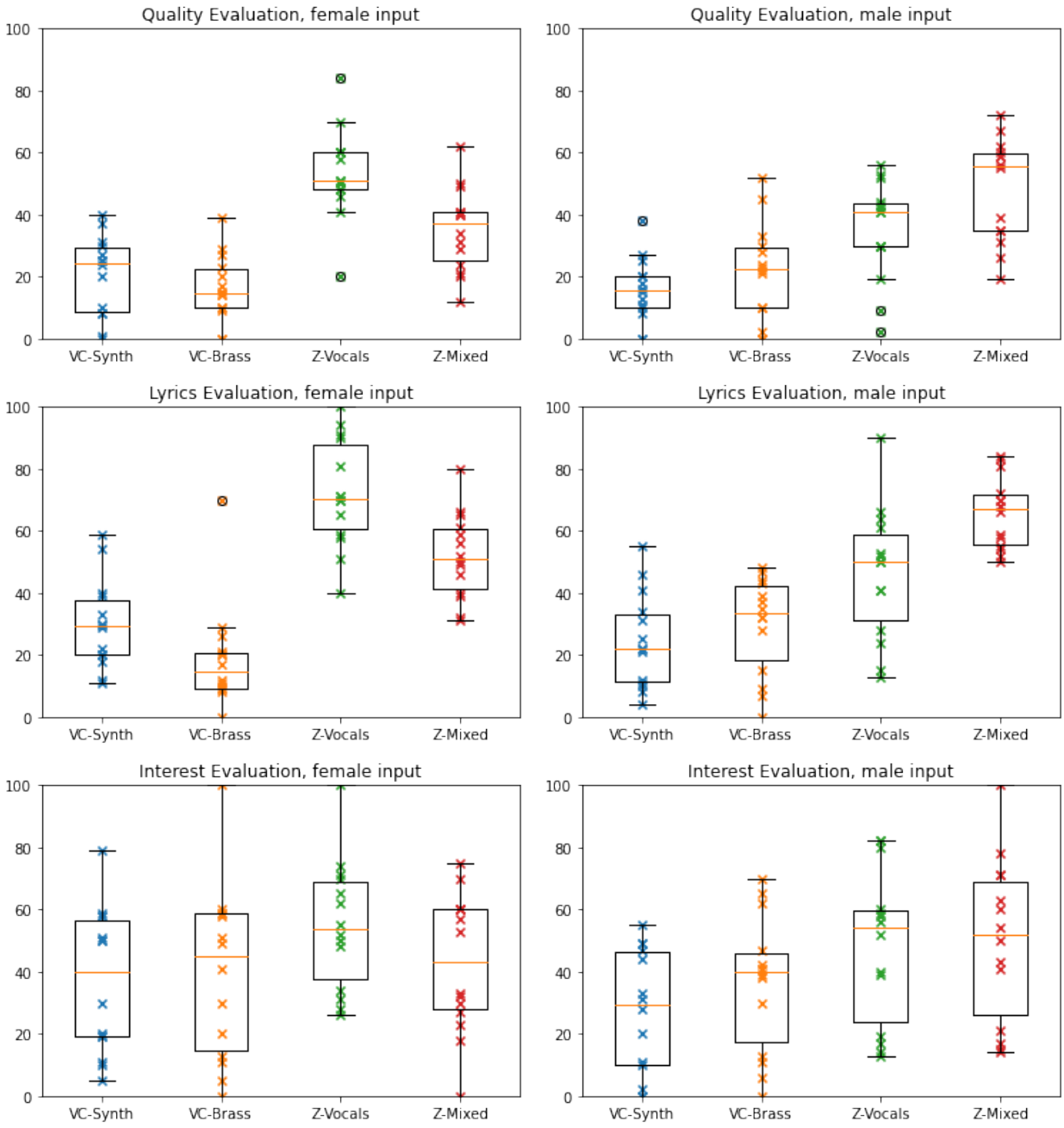

Figure 5: Results of the perceptual evaluation. All individual ratings are displayed as a scatter plot. A box plot marks the median rating with a horizontal line. The box itself extends from the first to the third quartile of the ratings.

7. REFERENCES

[1] H. W. Dudley, "The vocoder," *Bell Labs Rec.*, 1939.

[2] W. Meyer-Eppler, *Elektrische Klangerzeugung: Elektronische Musik und synthetische Sprache*. Ferd. Dümmlers Verlag, 1949.

[3] U. Zölzer, X. Amatriain, D. Arfib, J. Bonada, G. De Poli, P. Dutilleux, G. Evangelista, F. Keiler, A. Loscos, D. Rocchesso *et al.*, *DAFX – Digital audio effects, 2nd Edition*. John Wiley & Sons, 2011.

[4] J. O. Smith, *Spectral Audio Signal Processing*. `http://ccrma.stanford.edu/~jos/sasp/`, accessed 2022, online book, 2011 edition.

[5] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[6] J. Engel, C. Resnick, A. Roberts, S. Dieleman, M. Norouzi, D. Eck, and K. Simonyan, "Neural audio synthesis of musical notes with wavenet autoencoders," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1068–1077.

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

[8] 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)*, 2018, pp. 161–165.

[9] 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.

[10] S. Shan, L. Hantrakul, J. Chen, M. Avent, and D. Trevelyan, "Differentiable wavetable synthesis," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2022.

[11] J. Alonso and C. Erkut, "Latent space explorations of singing voice synthesis using DDSP," in *Proc. of the 18th Sound and Music Computing Conference*, July 2021.

[12] S. Nercessian, "End-to-end zero-shot voice conversion using a DDSP vocoder," in *2021 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 2021.

[13] F. Ganis, E. F. Knudsen, S. V. K. Lyster, R. Otterbein, D. Südholt, and C. Erkut, "Real-Time Timbre Transfer and Sound Synthesis using DDSP," in *Proc. of the 18th Sound and Music Computing Conference*, July 2021.

[14] A. Caillon and P. Esling, "Streamable neural audio synthesis with non-causal convolutions," 2022. [Online]. Available: https://arxiv.org/abs/2204.07064

[15] ——, "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis," 2021. [Online]. Available: https://arxiv.org/abs/2111.05011

[16] S. Choi, W. Kim, S. Park, S. Yong, and J. Nam, "Children's song dataset for singing voice research," in *International Society for Music Information Retrieval Conference (ISMIR)*, 2020.

[17] Z. Rafii, A. Liutkus, F.-R. Stöter, S. I. Mimilakis, and R. Bittner, "The MUSDB18 corpus for music separation," Dec. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1117372

[18] B. Li, X. Liu, K. Dinesh, Z. Duan, and G. Sharma, "Creating a multitrack classical music performance dataset for multimodal music analysis: Challenges, insights, and applications," *IEEE Transactions on Multimedia*, vol. 21, no. 2, 2019.

[19] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[20] 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.

[21] M. Schoeffler, S. Bartoschek, F.-R. Stöter, M. Roess, S. Westphal, B. Edler, and J. Herre, "webMUSHRA — A Comprehensive Framework for Web-based Listening Tests," *Journal of Open Research Software*, vol. 6, p. 8, 2018.