A REAL-TIME AUDIO EFFECT PLUG-IN INSPIRED BY THE PROCESSES OF TRADITIONAL INDONESIAN GAMELAN MUSIC

Luke M. Craig

Department of Computer Science
Appalachian State University
Boone, NC, USA
lukemcduffiecraig@gmail.com

R. Mitchell Parry

Department of Computer Science
Appalachian State University
Boone, NC, USA
parryrm@appstate.edu

ABSTRACT

This paper presents Gamelanizer, a novel real-time audio effect inspired by Javanese gamelan music theory. It is composed of anticipatory or "negative" delay and time and pitch manipulations based on the phase vocoder. An open-source real-time C++Virtual Studio Technology (VST) implementation of the effect, made with the JUCE framework, is available at github.com/lukemcraig/DAFx19-Gamelanizer, as well as audio examples and Python implementations of vectorized and frame by frame approaches.

1. INTRODUCTION

Gamelan music is the traditional court music of Indonesia, featuring an orchestra of pitched-percussion instruments. It has had a profound influence on Western composers, most famously on Debussy [1, 2]. Gamelanizer is a real-time audio effect plug-in inspired by the music theory of the Javanese variety of gamelan music.

This paper is organized as follows: Section 1.1 gives background information on gamelan music theory. Section 1.2 compares Gamelanizer to similar works and explains the motivation for creating it. Section 2 details our method for achieving this effect as a real-time Virtual Studio Technology (VST). Section 3 details our results. Section 4 discusses the considerations of our user interface, potential applications, and future work.

1.1. Background

There is a large variety of gamelan music theory. Our work is influenced specifically by the Javanese variant because it is arguably the most rule-based [3]. Specifically, we are concerned with the process known as *Mipil*, which translates to "to pick off one by one" [4]. The following is a description of the *Mipil* process. Because there is still debate in the ethnomusicology community concerning gamelan music, we stress that this description is an oversimplification. However, for our audio effect, this oversimplified understanding is still useful.

The numeric notation system used for gamelan music only indicates the *balungan* (melodic skeleton) that the *saron* instrument plays. This is because the notes that the *bonang barung* and *bonang panerus* instruments play can be generated, to a degree, from this base set [5]. In a hypothetical piece, if the first *balungan* notes are those in Table 1 then the elaborating *bonang barung* musicians will know to play the notes in Table 2. However, their notes will be played twice as fast as the *saron* and will occur half a beat earlier.

Copyright: © 2019 Luke M. Craig et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Table 1: *The balungan, or base melody.*

saron	3	5	2	6

Table 2: *The first level of subdivision.*

bonang barung	3	5	3	5	2	6	2	6

Therefore, the fourth note that the *bonang barung* plays will coincide with the second note that the *saron* plays. The eighth note of the *bonang barung* will coincide with the fourth note of the *saron*. The *bonang panerus*, which is an octave higher than the *barung*, repeats this subdivision process, now playing the notes in Table 3 twice as fast as the *bonang barung* and four times as fast as the *saron*. The combination of these three interlocking parts is shown in Table 4.

1.2. Motivation

There has been previous work related to gamelan and digital audio. Some have focused on synthesizing the unique harmonic structure of the instruments in a gamelan orchestra [6]. Others have tried to infer compositional rules by using real compositions [7]. Others have noted the algorithmic nature of some types of gamelan music [3]. Various computer-aided composition tools have been developed that apply the processes of gamelan music theory to numbered musical notation [8–10].

There is potential to extend concepts from gamelan music theory to audio signals, rather than note numbers, for music production and sound design. Given a digital audio workstation (DAW) with standard time and pitch modification tools, an audio engineer or musician could manually apply the process known as *Mipil* (see Section 1.1) to audio clips. We tested the suitability of the effect by performing this manual operation in a variety of musical mixing contexts. The results were intriguing. While it is a difficult effect to characterize, it is probably most similar to granular delay effects [11], specifically ones that can operate on large grain sizes.

Performing this process manually with the editing tools of a DAW is slow and can become monotonous. Furthermore, if a producer decides to alter a single note in the source, that change must be propagated down the chain of subdivision levels. It is hard to maintain perspective in this situation. Additionally, the process could only be applied to audio clips and thus is not insertable at arbitrary points in the signal chain without rendering audio. If audio

Table 3: *The second level of subdivision.*

bonang panerus	3	5	3	5	3	5	3	5	2	6	2	6	2	6	2	6

Table 4: *All three instruments.*

saron					3				5				2				6			
bonang barung			3		5		3		5		2		6		2		6			
bonang panerus		3	5	3	5	3	5	3	5	2	6	2	6	2	6	2	6			

has to be rendered, the non-destructive editing workflow of digital audio production is lost.

This paper proposes a solution to achieve this digital audio effect without the arduous and creatively limiting manual work. Specifically, a real-time VST plug-in. The effect has three responsibilities:

- time-scaling the quarter notes into eighth, sixteenth thirty-second, and sixty-fourth notes,
- pitch shifting the notes by a user defined amount, and
- placing and repeating the altered notes correctly in time, including earlier than the input signal.

2. METHOD

It is easiest to think of the problem from the perspective of a quarter note (hereafter referred to as a beat). Each beat needs to be pitch shifted and time scaled by the correct factor for each subdivision level and placed and repeated correctly. Then each level, including the base, would need to be delayed by the summation of the duration of one note of each of the levels that are higher pitched than it. Then latency compensation would be used to realign the base level with the input signal and place the subdivision levels earlier than the input.

While this naive method is relatively straightforward, it is not usable for a real-time plug-in. For one audio callback, the processing time of the naive method is often smaller than our proposed method. However, the naive method occasionally has too long of a processing time during a single audio callback, which causes audio dropouts from buffer underflow. Considering the different general audio effect frameworks of [12], the naive method is like a "frame by frame" approach that treats an entire beat as a frame, rather than the usual case of the block size of the audio callback. Our method, outlined in Figure 1, is a "frame by frame" approach that instead treats the much smaller fast Fourier transform (FFT) frames as the fundamental unit of work. In this manner, the computational load is more evenly distributed. We consider the process from the perspective of an individual sample:

1. For each sample, we pass it to each of the subdivision level processors (Section 2.1). These independently perform time compression and pitch shifting through the use of the phase vocoder technique (Section 2.2). They also handle duplicating the notes in the correct positions (Section 2.3).
2. We also copy each sample to a delay buffer for the base level and use latency compensation to make part of the output signal occur earlier than the input signal (Section 2.4).
3. We then update the subdivision levels' write positions if the play head in the DAW would be on a beat boundary (Section 2.5).

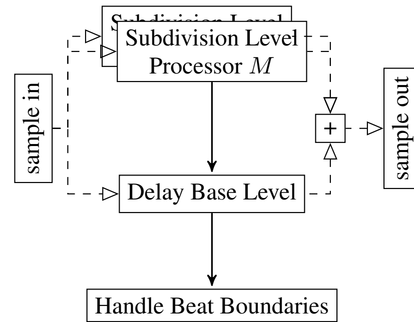


Figure 1: *An overview of our method.*

2.1. Subdivision level processors

The bulk of the work on each input sample is done by each of the subdivision level processors. Each subdivision level processor in the plug-in corresponds to the role of an elaborating instrument in the gamelan orchestra. The pitch shifting parameter of a subdivision level in the plug-in corresponds to the *tessitura* (normal pitch range) of an elaborating instrument. The time scaling factors of a subdivision level in the plug-in correspond to the rhythmic densities of an elaborating instrument.

In each subdivision level processor i, where $i \in \{1, 2 \ldots, M\}$, we perform time compression and pitch shifting via the classical phase vocoder technique [13]. Then we overlap-and-add (OLA) the synthesis frames multiple times in an output buffer channel associated with the subdivision level.

2.2. Phase vocoding

The phase vocoding consists of two stages: pitch shifting by resampling and time scaling. The effective time scaling factors $\mathbf{r}$ of the subdivision levels are

$$r[i] = \frac{1}{2^i}\,. \tag{1}$$

Given pitch shift amounts $\mathbf{c}$, in cents, the frequency scaling factors $\mathbf{p}$ are

$$p[i] = 2^{\frac{c[i]}{1200}}\,, \tag{2}$$

and the actual time scaling factors $\mathbf{v}$ applied after resampling are

$$v[i] = r[i]p[i]. \tag{3}$$

In other words, if the pitch shift amount is an octave (1200 cents), then no actual time scaling needs to be done because the resampling operation of pitch shifting will result in a note length that is $\frac{1}{2}$ the original beat length.

The frame size of the time scaling operation is fixed and independent of $p[i]$ and $v[i]$. If $v[i] \geq 1$, as is the case in the usual operation of our plug-in, the number of samples output from the resampler will be less than the number input to it. Therefore, by ordering the resampler first in the phase vocoders, we reduce the number of analysis-to-synthesis frame conversions that need to be performed per unit of time.

2.2.1. Pitch shifting by resampling

We use a stateful interpolator to accomplish resampling.[1] It needs to be stateful otherwise there would be discontinuities between blocks of data. The phase vocoders operate on overlapping frames that are one analysis hop size, h_a, samples apart

$$h_a = \frac{N}{o_a}\,, \tag{4}$$

where N is the length of the FFT and o_a is a constant analysis overlap factor.[2] Therefore, we always request h_a samples of output from the resampler. Because it is stateful and will be operating at subsample positions, the number of samples consumed by the resampler will vary each time. Therefore, we push the input samples onto a first-in, first-out (FIFO) queue and only pass them to the resampler when there is at least the number needed by the resampler available on the queue. After the resampler finishes, we pop the number of input samples it used off the queue. We then put the h_a samples of resampled audio data into a circular buffer of length N. The buffer is filled one hop at a time and we do not read from it until it is full. Each time we read from it, we will convert the analysis frame it contains to a synthesis frame.

2.2.2. Converting an analysis frame to a synthesis frame

Converting analysis frames to synthesis frames and OLA are the core elements of the phase vocoder technique, which has been written about extensively [13–21]. Normally, the phase vocoder is presented as operations done on a short-time Fourier transform (STFT) matrix [21]. For our real-time implementation, we instead follow the approach of only considering the current and previous length N overlapping frames of audio samples [18, Sec 5.3]. We also follow the common approach of analysis and synthesis windowing with a Hann window of length N [21]. Because we know when the beat boundaries occur, we can take advantage of the scheme proposed in [16, p. 327] of initializing the phases with the first analysis frame of each new beat. To scale the phases of the current frame each subdivision level processor only needs to store the scaled phases of the previous frame, the unmodified phases of the previous frame, and the synthesis overlap factor of the subdivision level i:

$$o_s[i] = \frac{o_a}{v[i]}\,. \tag{5}$$

2.3. Adding synthesis frames to output buffers

The number of samples $s[i]$ in one note of subdivision level i, is given by:

$$s[i] = \frac{s_b}{2^i}\,, \tag{6}$$

s_b being the number of samples per beat in the base level:

$$s_b = \frac{60 f_s}{t}\,, \tag{7}$$

where f_s is the sample rate of the DAW in Hz and t is the tempo in BPM. Figure 2, which we will refer to throughout this section, shows a hypothetical situation with a tempo of 80 BPM and a sample rate of 44.1 kHz. On the right side of Figure 2, $s[i]$ and s_b are visualized by the lengths of the final "D" notes of each subdivision level and the delayed base.

There are M subdivision levels and each one has a channel in the M channel circular buffer $\mathbf{B}$. When playback is begun from sample 0 in the DAW, we initialize $\mathbf{w}$, the lead write head positions in $\mathbf{B}$, with:

$$w[i] = 2s_b + s[M] + \sum_{j=i+1}^{M} s[j]\,, \tag{8}$$

so that the subdivision level M, which has the highest rhythmic density, will have the earliest position in time. Additionally, the beginning of subdivision level $M-1$ will occur with the beginning of the second note of subdivision level M. The initial values of $\mathbf{w}$, each at the beginning of the first note of their associated subdivision level, can be seen on the left side of Figure 2c.

We OLA the N samples from the synthesis frame $\mathbf{u}$ to $\mathbf{B}$, repeating the appropriate number of times for the subdivision level i. This is shown in Algorithm 1. To simplify the notation, the modular indexing of the circular buffer is ignored.

Algorithm 1 OLA while duplicating notes for subdivision level i

```
1: for j ← 0 to 2^i do                    ▷ note number j
2:     w_j ← w[i] + j(2s[i])
3:     B[i, w_j : w_j + N] += u
4: end for
```

In line 2 of Algorithm 1, the offset $j(2s[i])$ specifies that we skip a note every time we duplicate a synthesis frame $\mathbf{u}$. In Figure 2, the play head in the DAW (the blue triangle at sample 33075 in Figure 2a) has just reached the end of beat "A." The diagonal shading in Figures 2b and 2c shows the data that has been written to the output buffers at this time. Algorithm 1 is what creates the fractal-like pattern of these shaded regions. The gaps between the shaded regions will be filled, one overlapping synthesis frame of length N at a time, as the play head in the DAW moves over the next beat, "B."

Next, the lead write position of the subdivision level i is incremented by its synthesis hop size $h_s[i]$

$$w[i] \leftarrow w[i] + h_s[i]\,, \tag{9}$$

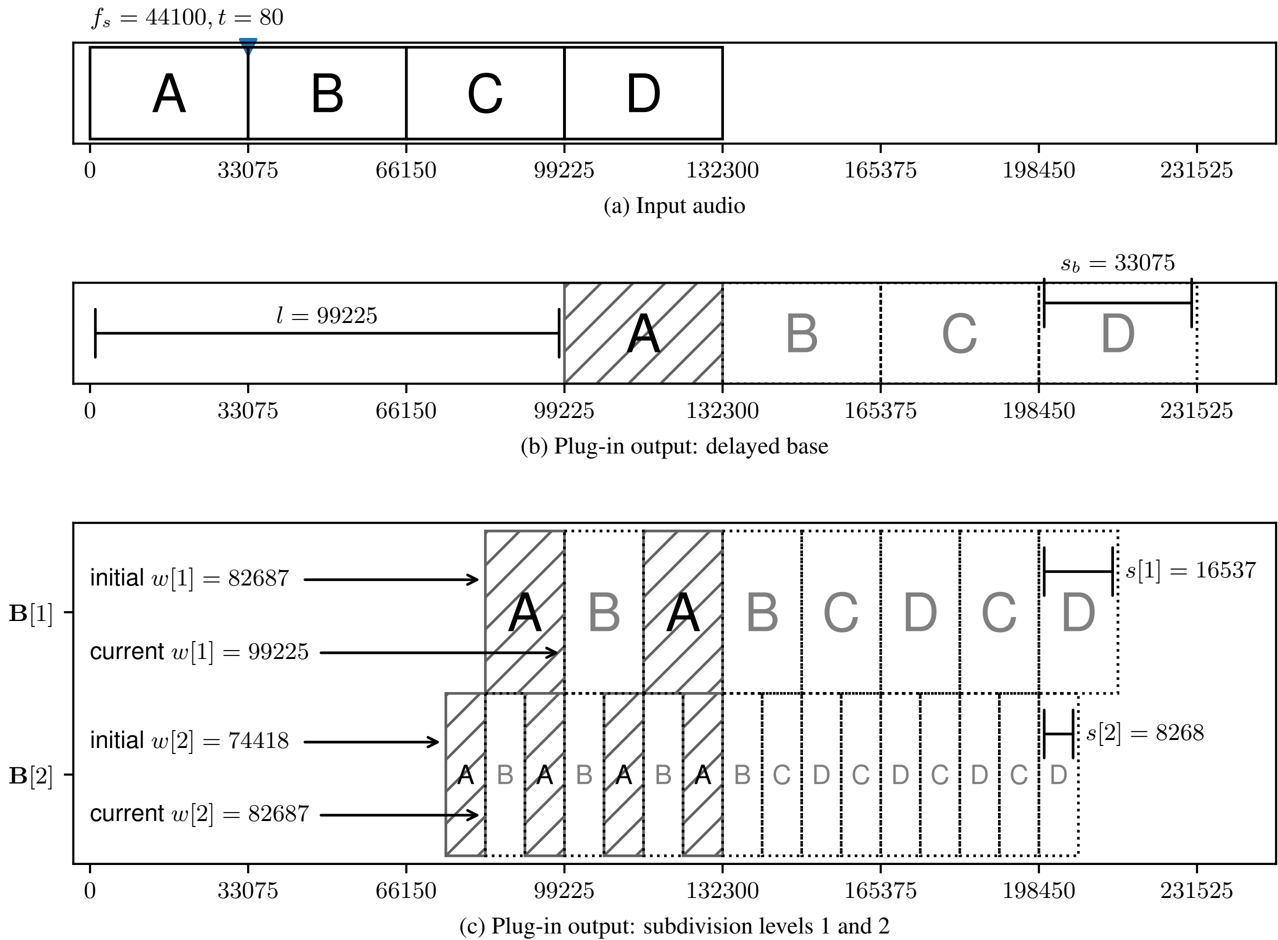
following the standard phase vocoder technique [13]. The synthesis hop size, the number of samples between the beginnings of overlapped synthesis frames, is determined by the analysis hop size h_a and the actual time scaling factor $v[i]$ of the subdivision level i:

$$h_s[i] = h_a v[i]\,. \tag{10}$$

In Figure 2, the write positions have already been incremented many times by equation (9). Their current values, each at the end of the first shaded note for an associated level, are displayed on the left side of Figure 2c.

[1] We implemented this resampling with the CatmullRomInterpolator class from the JUCE framework.

[2] An overlap amount of 75% is suggested in [14] so we set o_a to 4.



(a) Input audio

(b) Plug-in output: delayed base

(c) Plug-in output: subdivision levels 1 and 2

Figure 2: *The output of the plug-in without latency compensation. The play head in the DAW has just reached the beginning of the beat labeled "B" in the input audio (a). The x-axis tick marks indicate the sample positions of the beat divisions at a tempo of 80 BPM. In (b) and (c), the diagonally shaded regions show the data that has been written at this point. The unshaded regions with faded text show where data will be written as the play head progresses. When latency compensation takes effect the base level output (b) will realign with the input (a) and the subdivision level outputs (a) will occur earlier than the input.*

2.4. Delaying the base level

To place time-compressed versions of the entire first beat before the first beat begins in the DAW, we need to request a latency compensation amount that is larger than the initial positions of the lead write heads. Then we need to delay the base level by this same amount to realign it with the input signal and the rest of the tracks in the DAW. The plug-in needs to request l samples of latency compensation from the DAW:

$$l = 3s_b\,, \tag{11}$$

which is also the number of samples we delay the base level by. Given an input of four beats from the DAW, refer to Figure 2 to see what the output from the plug-in would be before latency compensation is applied. The area marked l on Figure 2b shows how delaying the base level aligns it with the subdivision levels in Figure 2c. Keeping this relative alignment and requesting l samples of latency compensation would realign the base level with the input audio (2a) and each subdivision level would begin before the input, as we desire.

2.5. Handling beat boundaries

Whenever the play head of the DAW moves into a new beat, we adjust the lead write heads (Section 2.5.1). If the beat we just finished processing was the second beat in a pair, we adjust the lead write heads further (Section 2.5.2). We also reset the phase vocoders now because the next piece of audio input should be from a new note with unrelated phases, as mentioned in Section 2.2.2.

2.5.1. Adjust lead write heads

When beginning each new beat, we reset $\Delta w[1] \ldots \Delta w[M]$, the number of samples each lead write head has moved during a beat, to zero. Every time a lead write head position $w[i]$ is incremented

with equation (9), we accumulate that change:

$$\Delta w[i] \leftarrow \Delta w[i] + h_s[i] \,, \tag{12}$$

where $0 \leq \Delta w[i] \leq s[i]$. Using $\mathbf{\Delta w}$, when the play head of the DAW[3] is on a beat boundary (for example 33075 or 66150 in Figure 2a), we move all the lead write head positions, $\mathbf{w}$, forward a small amount to the exact starting positions of their next subdivided notes:

$$w[i] \leftarrow w[i] + (s[i] - \Delta w[i]) \,. \tag{13}$$

This is necessary because their normal movements are not synchronized, due to the different resampling factors. Moving the write heads like this may leave a small gap between notes. However, it will never cause a "click" artifact from a sample-discontinuity because we are only ever adding windowed and complete synthesis frames.

2.5.2. *Handling second beats*

Every time we transition to a new beat we also determine if we were on the second beat of a pair. In Figure 2a, these are the beats labeled "B" and "D." If we are finishing with the second beat, then we increment each lead write head position to the starts of their next unwritten regions:

$$w[i] \leftarrow w[i] + s[i](2^{i+1} - 2) \,. \tag{14}$$

Each level has 2^{i+1} notes per pair of beats. After processing the data for a pair of beats, the lead write head of a subdivision level will be two note-lengths, $2s[i]$, past their initial position. That is why we subtract two in equation (14).

For example, as the play head in the DAW reaches the end of beat "B," at sample 66150 in Figure 2a $w[2]$ in Figure 2c will be at the beginning of the second "A" note of level 2, at sample 90956. Accordingly, by moving six notes ahead it will be at the beginning of the first "C" note in level 2, at sample 140568. Due to these increments and the duplication of notes, the plug-in always maintains an equal output rate with the input rate, even though we are consistently time-compressing.

2.6. Starting playback from arbitrary timeline positions

If the user of the DAW starts playback from some arbitrary position, we must determine what the values of $\mathbf{w}$ would be at this position, had the user started from the beginning of the session. Likewise, the user could also change playback positions without stopping the DAW, so we have to check for that as well. Instead of writing a mathematical function to determine the correct values of $\mathbf{w}$, we instead choose to simply run through the main processing method until reaching the play head position of the DAW, skipping any costly operations unrelated to updating $\mathbf{w}$. This way, the software is more flexible to change because we do not have to keep remodeling the behavior.

[3]From the perspective of a plug-in, the reported play head position from the DAW is incrementing an entire block size at a time, rather than sample by sample. Therefore, we have to calculate the expected position of the play head internally, for every sample that we process.

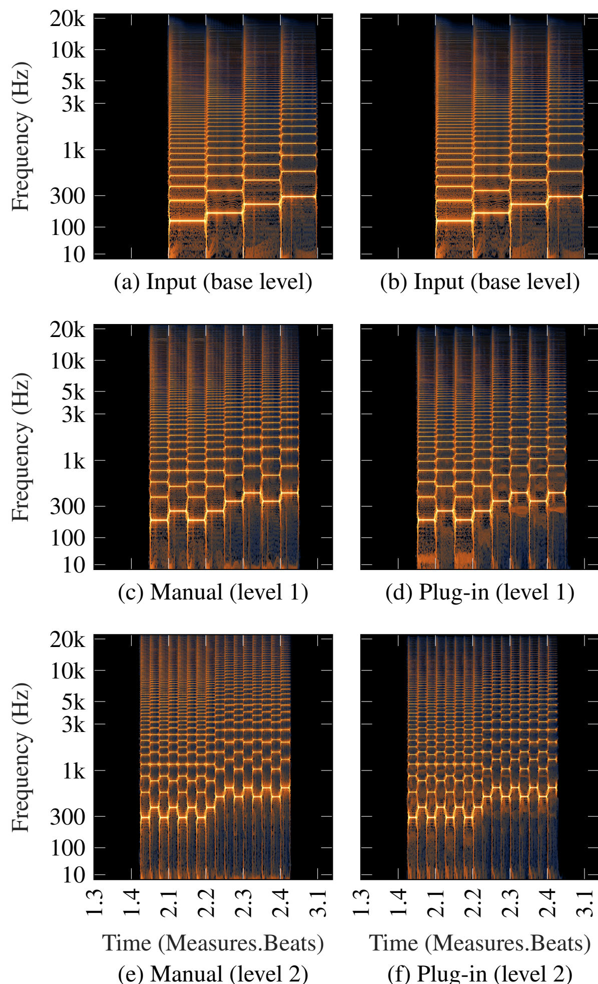
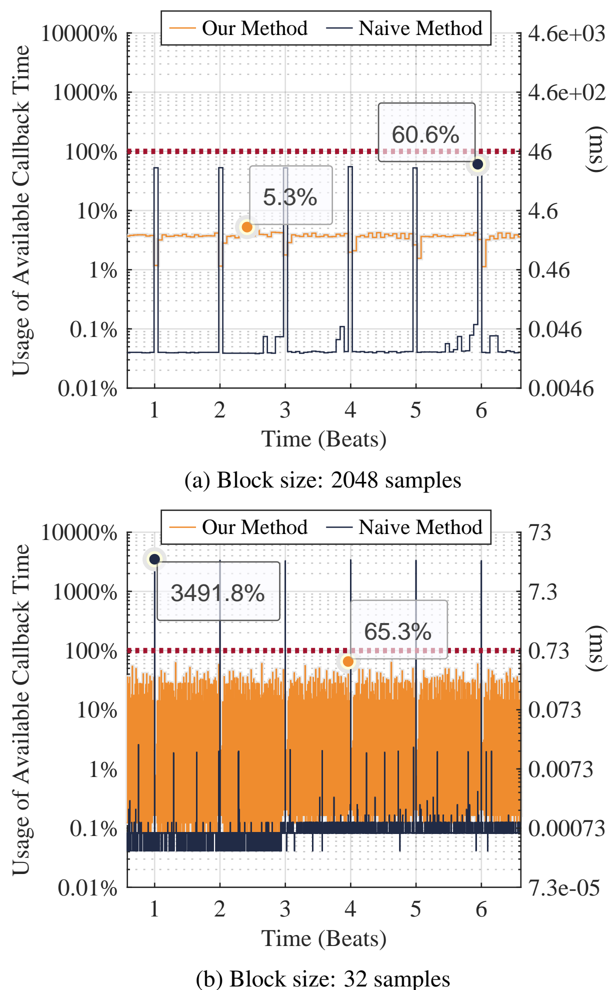


Figure 3: *Spectrograms comparing the outputs of the manual process (our desired result made with editing tools of a DAW) and the plug-in. The input signal (a and b) is four ascending quarter notes at 120 BPM. The first note of the input signal begins on beat 1 of measure 2.*

3. RESULTS

Figure 3 shows the output of the manual process, as described in Section 1.2, compared to the output of our real-time plug-in, using the same input signal. One can see the results are similar. Our method successfully time-compressed and placed all the notes in their correct places, including before the input signal. For level 1, the pitch shift factor was seven semitones (700 cents) and, as can be seen on the spectrogram of the plug-in output (3d), it matches the manual output (3c) sufficiently. For level 2, the pitch shift factor was fourteen semitones (1400 cents) and the plug-in output (3f) matches the manual output (3e) sufficiently again.

Figure 4 shows the performance measurements of the naive algorithm that is described in the beginning of Section 2 and our method. The measurements were collected with the plug-in running four subdivision levels with an FFT length of 1024 samples for each phase vocoder. Tests were conducted with block sizes of 32 and 2048 samples at a sample rate of 44.1 kHz. In both



(a) Block size: 2048 samples

(b) Block size: 32 samples

Figure 4: *Performance measurements, relative to available callback time, of four subdivision levels with pitch controls set to stacked fourth and an FFT length of 1024 samples. The tempo was 80 BPM. The four annotated points indicate the maximum time used for processing in the 32 bars that were measured for each test. Points above the dashed line are unacceptable.*

cases, the naive method has dramatic increases in processing time on the beat boundaries. The amount of samples being processed on these boundaries is independent of block size. In this example, 33075 samples are being processed at the beat boundaries because the tempo is 80 BPM. When the block size is 2048 samples, the plug-in has no more than 46 milliseconds (ms) to process these samples, so the naive method is still usable if no other plug-ins are running. However, when the block size is 32 samples, the plug-in has no more than 0.73 ms to do the same amount of work and so it causes buffer underflow and is unusable. Additionally, if the session tempo were slower, even a block size of 2048 may result in buffer underflow due to the increased number of samples per beat.

Our method, in both block size cases, has much better performance. The processing load is decoupled from the number of samples per beat and so there are no longer dramatic time increases at the beat boundaries. In fact, there are decreases in processing time at the beat boundaries because the initialization scheme we use does not alter the first frame of each beat. For the resampling stage of our processor (Section 2.2.1), the processing load scales with the block size, which is good. However, for the other part of the phase vocoding technique (Section 2.2.2), the processing load is independent of the block size, scaling instead with the FFT frame size. Still, because the load is independent of the number of samples per beat, lowering the tempo will not cause the block size of 32 samples to become unusable. If better performance for small block sizes was needed, reducing the FFT frame size would help, at the expense of audio quality.

4. DISCUSSION

4.1. Accompanying Material

The accompanying repository is available at `https://github.com/lukemcraig/DAFx19-Gamelanizer`. Included is the C++ code and VST builds for macOS and Windows. Audio examples of input and output are also included. Python implementations of vectorized and frame by frame approaches are also available in the repository.

The VST has only been tested to work in the DAW REAPER. Because of the necessity of unusually large amounts of latency compensation, DAWs that limit the amount of latency compensation a plug-in can request will not be able to use the effect, unless the user is comfortable with the subdivision levels occurring after the input. The user could render the output and then drag it into place, which would still be faster than doing it manually.

Another limitation is that the tempo cannot change. Likewise, if the input audio is not perfectly quantized to the grid, the output may sound random, in terms of the rhythmic context. This is because an instrument's note that begins before the beat will have the portion of the signal before the beat treated as if it were the end of the previous note and be moved and repeated in that manner.

4.2. User interface

The user interface is shown in Figure 5. The tempo in the DAW can be specified by the user in the text box at the top. This box is unchangeable during playback. The leftmost vertical slider controls the gain of the delayed input signal (the base level). Sometimes in gamelan music the *balungan* is not played by any instrument but merely implied by the elaborating instruments [22]. Likewise, it can be useful for the user to completely remove the input signal by pressing the mute button (labeled "M"), and only output the subdivision levels. This works well if the first subdivision level has no pitch-shifting applied to it.

Each subdivision level has an identical set of controls that are grouped by a thin rectangle. These can be thought of as channel strips on a mixing console. The largest rotary slider controls the pitch shift factor of that level. The long gray lines inside the rotary arc indicate the octave positions. The small ticks inside the arc indicate the positions of perfect fourths. The small ticks outside the arc indicate the positions of perfect fifths. The user can optionally snap to any of these three intervals by holding different modifier keys while dragging. Changes to pitch shifting are allowed during playback but do not take effect until the subdivision level processor that is being changed is between analysis frames.

Each subdivision level processor also has a gain slider and high and low-pass filters that are applied after pitch-shifting. They

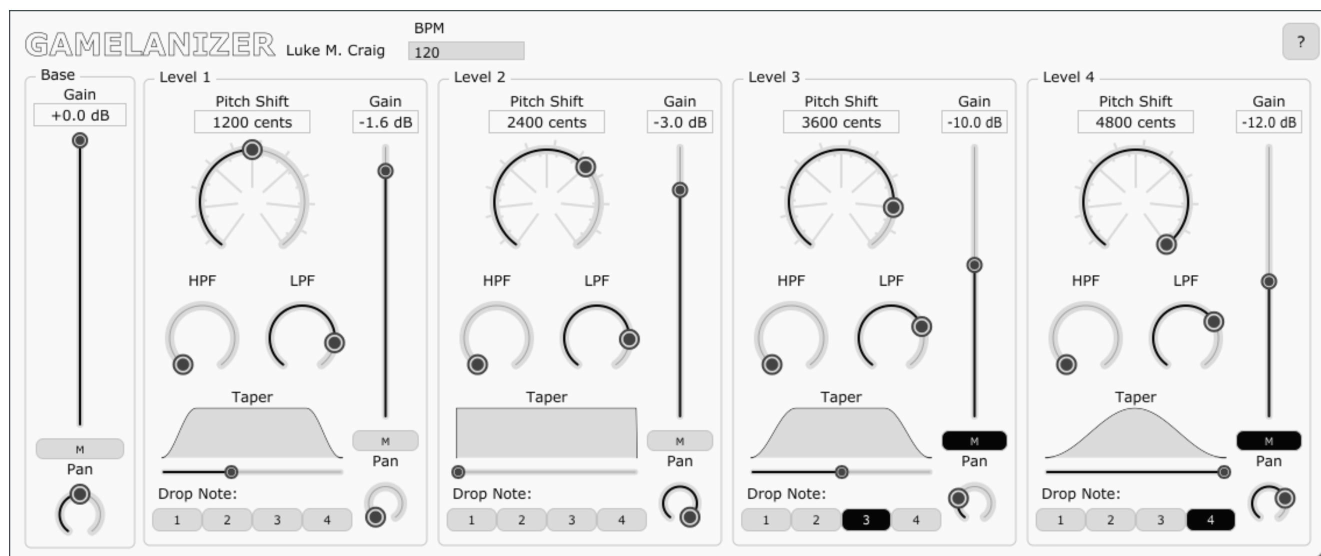


Figure 5: *The Gamelanizer GUI with four subdivision levels. Pitch shifting controls are set to stacked octaves.*

each also have a horizontal slider that specifies the α parameter of a Tukey window [23] that is applied to the input signal before phase vocoding. This can be useful for tapering the input signal to achieve different attack and release characteristics per level. The displays above the horizontal sliders show the resultant Tukey windows.

When any of the four buttons labeled "Drop Note" are toggled, those notes become muted. For instance, if the fourth note was dropped from the fourth subdivision level, then every fourth sixty-fourth-note would become a rest.

Each subdivision level can be output independently or mixed into stereo. By routing them independently the user has more control and can apply other plug-ins and routing to mix them as they wish. When mixed into stereo the pan knobs below the mute buttons can be used.

4.3. Applications

The plug-in is useful for at least two general scenarios. First is the scenario of a composer or musician who is writing new music. If a composer was writing a base melody that they wanted the *Mipil* process to be applied to, they could use the Gamelanizer to get almost instant feedback and make adjustments. In this scenario the plug-in could be considered a computer-aided composition tool similar to those mentioned in Section 1.2. The advantage of using Gamelanizer over those tools is that the composer can transform any sound, such as a human voice, rather than just transforming Musical Instrument Digital Interface (MIDI) input to virtual instruments. Another advantage is that many tech-savvy musicians today compose purely by ear, and using our plug-in would be more intuitive for them than thinking about symbolic representations of notes. For this use case, we suggest starting with only the first and second subdivision levels because higher levels can sound irritating.

Second is the scenario of an audio engineer mixing someone else's music, or creating strange effects for sound design. In this scenario the plug-in is more of an effect that alters the character of the signal without altering the tonal harmony of the music. We recommend pitch shifting with octaves or no pitch shifting to achieve this use case. Filters are useful for adding depth by moving subdivision levels into the background. This can help perceptually separate the effect from the dry signal. Tapering is also useful here for imposing a rhythmic structure on an input signal that does not have one. The third and fourth levels are more useful in this usecase than they were in the first because the rapid rhythmic density of these levels can be mixed to sound less separated from the input source.

4.4. Future work

There are several possible future works, related to gamelan theory, that would improve this software. First, in gamelan music the elaborating instruments often sustain notes instead of playing on top of the second beat in a pair. We have implemented turning the notes into rests but not sustaining the previous notes. This sustain could be achieved by implementing the "spectral freeze" as described in [24]. Similarly, gamelan musicians add variations to their pitch and octave choice [5]. It may make the digital audio effect sound more organic to introduce some stochastic variation to emulate this. Another unimplemented aspect of gamelan music is that the orchestra's instruments are crafted in pairs that are intentionally out of tune to create a shimmering amplitude modulation that is intended to induce a religious trance-like effect [5]. Therefore it would be nice to include a stereo detuning parameter to the plug-in. Lastly, sliding tempo changes are an important part of gamelan music. Our method does not handle tempo changes. We are interested in Celemony Software's new Audio Random Access (ARA) [25] as a means to deal with this.

There are also possible future works unrelated to gamelan theory. One would be making use of beat detection so that the input signal does not have to already be perfectly quantized in time. Perhaps the work in [26] would prove useful for this. Another improvement would be to implement pitch detection on the input notes and limit the shifted notes to only be mapped to MIDI side-

chain input. That way, the tonal harmony of a song being mixed would not become cacophonous. Another improvement we are investigating is performing the pitch shifting without resampling as described in [17]. Another improvement would be to implement something like a noise gate with hysteresis so tails of notes that extend past the beat borders are not repeated. This may be simple to implement given that our method is aware of its location in the beat, and thus should know when to expect low and high amplitudes. Finally, spatialization with head-related transfer functions (HRTF) [27] would be an interesting feature to include.

5. ACKNOWLEDGMENTS

We would like to thank the Department of Computer Science at Appalachian State University, Will Padgett for testing the plug-in and composing example songs, Eyal Amir for the C++ advice, Dr. Virginia Lamothe at Belmont University for the introduction to gamelan music in 2011, and the anonymous reviewers for their feedback.

6. REFERENCES

[1] Richard Mueller, "Javanese influence on Debussy's 'Fantaisie' and beyond," *19th-Century Music*, vol. 10, no. 2, pp. 157–186, 1986.

[2] Neil Sorrell, "Gamelan: Occident or accident?," *The Musical Times*, vol. 133, no. 1788, pp. 66–68, 1992.

[3] Charles Matthews, "Algorithmic thinking and central Javanese gamelan," in *The Oxford Handbook of Algorithmic Music*. Oxford University Press, Feb. 2018.

[4] R. Anderson Sutton, *Traditions of Gamelan Music in Java: Musical Pluralism and Regional Identity*, CUP Archive, Apr. 1991.

[5] Jeff Todd Titon, Ed., *Worlds of music: an introduction to the music of the world's peoples*, Schirmer Cengage Learning, Belmont, CA, 5th ed edition, 2009.

[6] Lydia Ayers, "Merapi: A composition for gamelan and computer-generated tape," *Leonardo Music Journal*, vol. 6, 1996.

[7] Khafiizh Hastuti, Azhari Azhari, Aina Musdholifah, and Rahayu Supanggah, "Building melodic feature knowledge of gamelan music using apriori based on Functions in Sequence (AFiS) Algorithm," *International Review on Computers and Software (IRECOS)*, vol. 11, no. 12, Dec. 2016.

[8] Charles Michael Matthews, *Adapting and applying central Javanese gamelan music theory in electroacoustic composition and performance*, Ph.D thesis, Middlesex University, May 2014.

[9] Khafiizh Hastuti and Khabib Mustafa, "A method for automatic gamelan music composition," *International Journal of Advances in Intelligent Informatics*, vol. 2, no. 1, pp. 26–37, Mar. 2016.

[10] Gerd Grupe, *Virtual Gamelan Graz: Rules, Grammars, Modeling*, Shaker, 2008.

[11] Vincent Verfaille, Catherine Guastavino, and Caroline Traube, "An interdisciplinary approach to audio effect classification," in *Proc. Digital Audio Effects (DAFx-06)*, Montreal, Canada, Sep. 18-20, 2006.

[12] Daniel Arfib, "Different ways to write digital audio effects programs," in *Proc. Digital Audio Effects (DAFx-98)*, Barcelona, Spain, Nov. 19-21, 1998, pp. 188–191.

[13] Mark Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, no. 4, 1986.

[14] Dan Barry, David Dorran, and Eugene Coyle, "Time and pitch scale modification: a real-time framework and tutorial," in *Proc. Digital Audio Effects (DAFx-08*, Espoo, Finland, Sep. 1-4 2008.

[15] Eric Moulines and Jean Laroche, "Non-parametric techniques for pitch-scale and time-scale modification of speech," *Speech Communication*, vol. 16, no. 2, pp. 175–205, Feb. 1995.

[16] J. Laroche and M. Dolson, "Improved phase vocoder time-scale modification of audio," *IEEE Transactions on Speech and Audio Processing*, vol. 7, no. 3, pp. 323–332, May 1999.

[17] J. Laroche and M. Dolson, "New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects," in *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics. WASPAA'99 (Cat. No.99TH8452)*, New Paltz, NY, USA, 1999, pp. 91–94, IEEE.

[18] Amalia De Götzen, Nicola Bernardini, and Daniel Arfib, "Traditional (?) implementations of a phase-vocoder: The tricks of the trade," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00)*, Verona, Italy, 2000.

[19] Magnus Erik Hvass Pedersen, "The phase-vocoder and its realization," May 2003.

[20] Joshua D. Reiss and Andrew McPherson, "The phase vocoder," in *Audio Effects: Theory, Implementation and Application*. CRC Press, 2015.

[21] Alexis Moinet and Thierry Dutoit, "PVSOLA: A phase vocoder with synchronized overlap-add," in *Proc. Digital Audio Effects (DAFx-11)*, Paris, France, Sept. 2011.

[22] Sumarsam, *Gamelan: Cultural Interaction and Musical Development in Central Java*, University of Chicago Press, Dec. 1995.

[23] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, Jan. 1978.

[24] Jean-François Charles, "A tutorial on spectral sound processing using Max/MSP and Jitter," *Computer Music Journal*, vol. 32, no. 3, pp. 87–102, Sept. 2008.

[25] Celemony, "ARA Audio Random Access | Celemony | Technologies," Available at https://www.celemony.com/en/service1/about-celemony/technologies, Accessed April 18, 2018.

[26] A. M. Stark, M. D. Plumbley, and M. E. P. Davies, "Real-time beat-synchronous audio effects," in *Proceedings of the 7th international conference on New interfaces for musical expression - NIME '07*, New York, New York, 2007, ACM Press.

[27] Durand R. Begault and Leonard J. Trejo, "3-D Sound for Virtual Reality and Multimedia," Technical Report NASA/TM-2000-209606, NAS 1.15:209606, IH-010, NASA Ames Research Center, Aug. 2000.