*Proceedings of the 18th Linux Audio Conference (LAC-20), SCRIME, Université de Bordeaux, France, November 25–27, 2020*

# ON CHOOSING BEST SAMPLES FOR VIRTUAL DRUMS

*André Nusser*

DrumGizmo
Saarbrücken, Germany
andre.nusser@gmail.com

*Bent Bisballe Nyeng*

DrumGizmo
Aarhus, Denmark
deva@aasimon.org

## ABSTRACT

Sampling drum kits well is a difficult and challenging task. Especially, building a drum kit sample bank with different velocity layers requires producing samples of very similar loudness, as changing the gain of a sample after recording makes it sound less natural. An approach that avoids this issue is to not categorize the samples in fixed groups but to simply calculate their loudness and then dynamically choose a sample, when a sample corresponding to, e.g., a specific MIDI velocity is requested. We present a first investigation of algorithms performing this selection. We implemented the seemingly best candidate in DrumGizmo – a free software drum plugin – and we do experiments on how our suggested algorithm performs on the sampled drum kits.

## 1. INTRODUCTION

When creating virtual instruments that correspond to a certain analog instrument, we naturally always aim at making them sound as realistic as possible. There are at least two ways to achieve this. First, we can use physical simulations like the famous Pianoteq virtual instrument [1] does or, second, we can use real samples from the instrument as, e.g., most drum plugins do. In this article we focus on the second approach. When using this approach, a question that naturally comes to mind is how to use the sample data to get the most realistic sound. There are two orthogonal directions to tackle this problem. First, when getting the input of a programmed drum (e.g., as MIDI), we want to humanize it such that the MIDI velocities are according to how a real drummer would play this piece. Second, after applying such a humanization, we get to a lower-level problem which is that of choosing the right sample from our limited data set. Again, we focus on the second point in this article.

The arguable standard for choosing samples is the famous Round Robin algorithm. This algorithm groups samples of similar loudness together and then selects them in a circular manner (first sample, second sample, ..., last sample, first sample, ...). While this is the standard, it has significant drawbacks. First, it requires a somewhat arbitrary grouping of velocities. This in turn might lead to so called "staircase effects" when playing sweeps from quiet to loud notes. This is often resolved by scaling the loudness of the sample to the corresponding MIDI velocity. However, this decouples the sample loudness from the actual strength of the hit, again potentially leading to a less natural sound as this leads to inconsistencies between the different played samples.

In this work, we introduce a new sample selection algorithm that is not based on grouping of the samples, but instead works purely on the given loudness of the samples. Additionally, it does not adjust the gain of single samples. The goal of this algorithm is to choose the best possible sample, according to the requested MIDI velocity after humanization. This means, we want to choose a sample which is as close as possible. However, if we always just choose the closest sample we run into two issues. First, this creates artifacts (as shown later in this article), and second, it leads to a robotic sound when we play the same sample(s) over and over. Thus, we additionally have to make sure to choose a reasonably diverse set of samples from our sample data set. Finally, to avoid further artifacts in the form of patterns, we additionally want randomization to help us breaking these patterns.

### 1.1. Our Contribution

To the best of our knowledge, this is the first academic article that deals with the issue of selecting best samples from a set of samples with "continuous power values". To this end, we first identify important aspects that sampling algorithms in this setting have to fulfill. After we formulate these requirements and formalize them to some degree, we present our resulting algorithm, which is based on the computation of a multi-criteria objective function. Consequently, we give an overview over an implementation of this approach and then conduct experiments to evaluate the actual quality. As reference implementation, we use the old sample selection method of DrumGizmo [2] – an open source drum machine.

### 1.2. Related Work

Regarding related work, we consider the previous sample selection methods used by DrumGizmo. DrumGizmo is an open source, cross-platform, drum sample engine and audio plugin aiming to give an output that is as close to a real drummer as possible.

In DrumGizmo, the engine gets a value $l \in [0, 1]$ which must then be used for deciding how the output should be produced. Some engines use this value as a gain factor but in the case of DrumGizmo it is used for sample selection only. The early versions used a sample selection algorithm based on velocity groups, akin to the one used by the sfz format [3], in which each group spans a specified velocity range and the sample selection is made by selecting one of the samples contained in the group corresponding to the input velocity uniformly at random. See Figure 1 for the flow diagram.

This algorithm did not give good results on small sample sets, so later an improved algorithm was introduced which was instead based on normal distributed random numbers and with power values for each sample in the set. A prerequisite for this new algorithm is that the power of each sample is stored along with the sample data of each sample. The power values of a drum kit are floating point numbers without any restrictions but assumed to be positive. Then the input value $l$ is mapped using the canonical bijection between $[0, 1]$ and $[p_{\min}, p_{max}]$. We call this new value $p$.

Now we describe the aforementioned improved sample selection algorithm. We select a value $p'$ drawn from the normal distribution $\mathcal{N}(\mu = p', \sigma^2)$, where the mean value $\mu$ is set to the input value $l$ and the standard deviation $\sigma$ is a parameter controlled by the user

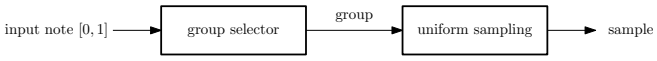


Figure 1: Flow diagram of the first sampling algorithm of DrumGizmo.

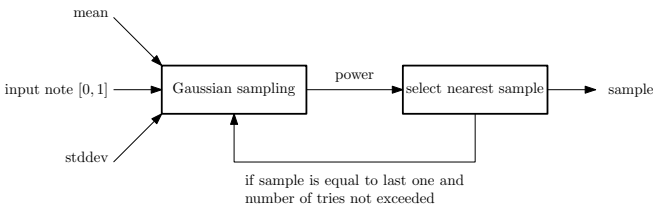


Figure 2: Flow diagram of the second sampling algorithm of DrumGizmo.

expressed in fractions of the size and span of the sample set. Now we simply find the sample $s$ with the power $q$ which is closest to $p'$. In case $s$ is equal to the last sample that we played, we repeat this process, otherwise we return $s$. If we did not find another sample than the last played after 4 iterations, we just return the last played sample, see Figure 2 for the flow diagram.

## 2. PRELIMINARIES

Drum samples are cut from recordings of drums with multiple microphones. Each hit on a drum must be distinguishable from the others and can therefore not overlap in time. Due to the multiple microphones used for the recording, each sample spans multiple channels. Additionally, due to the speed of sound in air and the distance of each drum from each of the microphones used, the initial sample position in each of the channels will not be at the same place – the channel which is the closest to the sound source of a particular instrument is the *main channel* of that instrument, see Figure 3.

A sample has a *stroke power* that is the physical power used by the drummer when making that particular hit. Since this is not something that can be easily measured, each sample power is instead calculated as the power of the signal in an initial attack period of the audio of the main channel:

$$power(s, n) = \sum_{i=0}^{n} s[i]^2,$$

where $n$ is defined on a per instrument basis and will vary from instrument to instrument, and $s[i]$ is the $i$th audio sample of the main channel.

Since the powers are simply sums of squares, they can be used for comparing one sample to another and ultimately for mapping a MIDI velocity to a matching sample.

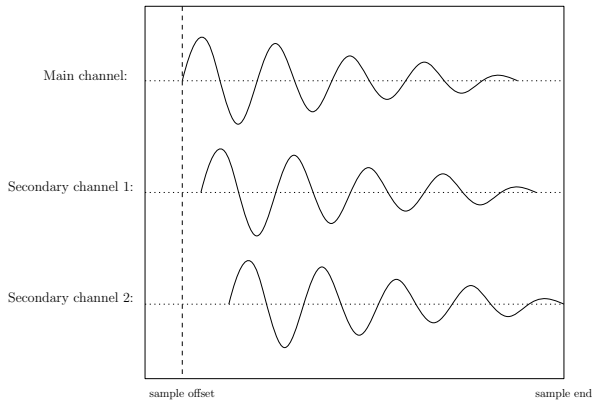


Figure 3: Sketch of the original signals of a sample recorded with multiple microphones.

### 2.1. Setting

We now describe the setting in which we want to choose the samples. We are given:

- a drum kit consisting of a set of instruments $I$
- for each instrument $i \in I$, we are given an input sample set $S_i$
- each sample $s \in S_i$ is already labeled with a power value $p_s \in \mathbb{R}^+$

After reading the drum kit, requests of the form $(i, p) \in I \times \mathbb{R}^+$ arrive. We want to answer these requests by choosing the best sample from $S_i$ for the power value $p$.

### 2.2. Notation and Terminology

We use the following notation throughout this article. An *instrument* is considered to be one of the drums of the drum kit that we sampled. A *sample* (denoted by $s, s', \dots$) is the recording of one hit on a specific instrument. Given a sample $s$, its *power* $p_s$ is the perceived loudness and can be expressed by any common loudness measure of an audio clip. If a power value is requested and does not correspond to a sample, we denote it by $p, p', \dots$. With the term *velocity*, we refer to the attack velocity of a MIDI note and it is thus between 0 and 127. We consider time in a discretized way and thus a *time point* is an integer value intuitively referring to the number of time steps passed since the beginning of time. For a sample $s$, we use $t_s$ to refer to the time point at which the sample was played last.

## 3. REQUIREMENTS

We now discuss which requirements a good sampling algorithm intuitively has to fulfill. Such an algorithm has a trade off between two main objectives: choosing a sample which is close to the requested power value, while not choosing the same sample too close to the previous time it was used. Note that if we just want to be as close as possible to the requested power value, then we would always just choose the closest sample. However, if we now play a sequence of the same instrument at the same power level, then we always play the same sample and thereby obtain a robotic sound. Thus, we want to find other samples that are not too far.

More concretely, we aim to fulfill the following requirements with our proposed algorithm.

**Close sample:** The chosen sample should be reasonably close to the requested power value, such that the listener perceives it as being played at the same velocity.

**Avoid same samples:** When we have multiple samples to choose from, we should always take one that was last played far enough in the past to avoid a robotic sound.

**Randomization:** Furthermore, to avoid patterns (like e.g. in Round Robin, where exactly every $n$th hit sounds the same when we have $n$ samples in our velocity group), we want some randomization.

**Locality:** If two samples have a similar power value, they should also be treated similarly by the algorithm. In other words, locally, samples should have almost the same probability of being chosen.

We now formalize the requirements stated above. Let $p, p'$ be two power levels. We define their dissimilarity to simply be their distance $|p - p'|$. Thus, if $p$ is the input power value and $p'$ is the power value of the chosen sample, we want to minimize the above term. Let $s$ be a sample and $t_s$ the time point it was played last. When we are now queried for a sample at time $t$, then for $s$ to be a good sample, we want $t - t_s$ to be reasonably high. Again, we just use the distance between the current time step and the last time step a sample was used. Randomization is difficult to formalize in a simple way in this context, thus, we just require that for the same history, different outcomes of choosing a sample should be possible. The last requirement we also state in a rather intuitive than formal way. Assume we are requested a sample for the power value $p$ and the two samples $s, s'$ have a very similar power value. Then, if we exchange $t_s$ and $t_{s'}$, the probability of choosing $s$ over $s'$ should be roughly the same as if we do not exchange them.

## 4. ALGORITHM

In this section we discuss the new algorithm that we suggest for sample selection. The requirements mentioned in Section 3 consist of several different objectives. Thus, we are dealing with a multi-objective optimization, where we have to somehow combine the different objectives into one. As we are dealing with an interactive setting where the quality of the solution of the optimization is determined by the user, it seems natural to make the algorithm parametrized and expose the parameters to the user. Using these parameters, the user can influence how the different objectives are weighted and thereby influence the final result, just having a very rough intuitive and non-technical understanding of the process.

Following from the above description, we choose a single objective function that we optimize. This objective function consists of the terms that were roughly outlined in Section 3 as well as the parameters that are factors in front of the requirements terms. We formulate our objective function such that smaller values are better and thus we end up with a minimization problem. Consequently, we just have to evaluate a single term on each sample and then pick the sample which has the smallest value.

We now give the objective function. Let $p$ be the power that is requested. As before, for any sample $s$, let $t_s$ be the time at which $s$ was played last (the unit does not matter as it is parametrized by $\beta$ anyway), and let $r(s, t)$ be a random number generator producing numbers in the range $[0, 1]$ uniformly and independently at random. Let $\alpha, \beta, \gamma > 0$ be the parameters that are later exposed to the user of the sample algorithm. Also, recall that $p_{\min}, p_{\max}$ is the minimal and maximal power value, respectively, and that $S$ is the sample rate, i.e., the number of time steps per second. At the current time $t$, we now want to find the sample $s$ minimizing the objective function

$$f(s,t) \coloneqq \alpha \cdot \left(\frac{p - p_s}{p_{\max} - p_{\min}}\right)^2 + \beta \cdot \left(1 + \frac{t - t_s}{S}\right)^{-1} + \gamma \cdot r(s,t).$$

Note that we have to ensure $p_{\max} \neq p_{\min}$ to avoid division by zero.

Let us now consider the objective function in more detail. The objective function consists of three summands and we will have a closer look at them in order.

The first summand, namely

$$\alpha \cdot \left(\frac{p - p_s}{p_{\max} - p_{\min}}\right)^2,$$

is for expressing the distance of the sample's power value $p_s$ to the desired power value $p$. Instead of using the absolute value $|p - p_s|$ as discussed in Section 3, we first normalize the value to be in the range $[0, 1]$ and then square it. By squaring, we put a significantly stronger penalty than the absolute value on samples whose power value is very far from the requested power value. Note that if we request a power value that lies in the middle of the power values of our sample set, then this term will maximally be around $\frac{1}{4}\alpha$. However, if $p = p_{\max}$ or $p = p_{\min}$, then we might obtain a value of $\alpha$. While this might seem unreasonable at first glance, we want to highlight that this indeed matches the desired behavior, because the penalty should be only dependent on the distance to the sample and not the possible worst case value it can attain. In other words, a bad choice should not be made to look better, if there are much worse choices possible.

The second summand, namely

$$\beta \cdot \left(1 + \frac{t - t_s}{S}\right)^{-1},$$

expresses how much time passed since we last played the sample for which we are currently evaluating the objective function. However, we want large values if little time passes and therefore we raise the whole term to the power of $-1$. To avoid extreme values, we add 1 to the normalized distance of the current time and the last time the sample was played. Note that if we would not have a "+1", then for $t - t_s$ being very small, the values of this term would be huge and thus dominate the whole objective function. We normalize by the sample rate, as we want the time distance to be in seconds and not in samples.

The third summand, namely

$$\gamma \cdot r(s,t),$$

just adds some noise to the process to make it non-deterministic and thus avoid patterns in the selection as mentioned in Section 3.

We already explained the core part of the sample selection algorithm. The remainder is now straight-forward. We simply evaluate the objective function for each sample and then pick the one with the smallest value. For completeness, Algorithm 1 shows the pseudo code for the sample selection algorithm.

Note that the worst-case complexity of evaluating the objective function is linear in the number of samples for the instrument that we are considering. However, in practice we can avoid evaluation for most samples by simply starting with the "most promising" sample and recursively evaluating the neighbors (with respect to power value) until the future possible evaluations cannot beat the currently best value.

## 5. EMULATING OTHER SAMPLE SELECTION ALGORITHMS

One of the main advantages of the described sampling algorithm is that it can emulate the most common sample choice algorithms. Sometimes this can be done by just adjusting the parameters $\alpha, \beta, \gamma$,

**Algorithm 1** This is the pseudo code of the sampling function.

**Input:** Requested power $p$, instrument $i$, current time step $t$, parameters $\alpha, \beta, \gamma$, and array $last$ with the time points a sample has been played last
**Output:** Sample $s$

```
s ← undefined
f_min ← ∞
for s' ∈ S_i do
    v ← α · ((p − p_s') / (p_max − p_min))^2 + β · (1 + (t − last[s']) / S)^−1 + γ · r(s', t)
    if v < f_min then
        f_min ← v
        s ← s'
    end if
end for
last[s] ← t
return s
```

and sometimes we have to prepare the power values of the drum kit accordingly. In the following, we describe which algorithms can be emulated and how we have to set the parameters and power values for that.

First, note that all extreme choices of the parameters – meaning that we set one parameter of $\alpha, \beta, \gamma$ to a positive value and all others to zero – emulate different selection algorithms.

**Choose Closest.** If we set $\alpha > 0$ and $\beta = \gamma = 0$, then the objective function reduces to the first summand and thus we just always choose the sample $s$ that minimizes $|p - p_s|$, i.e., the closest sample.

**Choose Oldest.** Similarly, if $\beta > 0$ but $\alpha = \gamma = 0$, then the objective function reduces to the second summand and thus is minimized by the sample $s$ that maximizes $t - t_s$, i.e., the sample that was last played the furthest back in time.

**Random Selection.** If now $\gamma > 0$ and $\alpha = \beta = 0$, then the objective function reduces to the third summand and we thus always select a sample uniformly at random.

**Round Robin.** The previously mentioned emulations were straight forward, however, the arguably most commonly used sample selection algorithm in practice is Round Robin. As already discussed in Section 1, Round Robin assumes the samples to already be grouped. In our case this means that samples $s_1, \ldots, s_k$ that belong to the same group should all be assigned the same power value, i.e., $p_{s_1} = \cdots = p_{s_k}$. Now, if there is a query with power value $p$, we want to always choose the closest group of samples, thus $\alpha$ should be large compared to $\beta$ and $\gamma$, e.g., $\alpha = 1$. After restricting to the samples of a specific group, we now always want to play the oldest sample, thus we simply want a small value $\beta > 0$, say $\beta = 1/1000$. If we additionally want to randomize Round Robin in a way that we sometimes choose the second or third oldest sample, then we want to set $\gamma$ to a small value smaller than but in a similar range as $\beta$, say $\gamma = 1/4000$.

## 6. IMPLEMENTATION

We added our new sampling algorithm to DrumGizmo, replacing the one it previously used.[1] The sampling algorithm itself did not require any particular implementation efforts. Most of the time was spent on the theoretical part of it. To give a better overview over the technicalities, we briefly list the information that needs to be stored. In a preprocessing phase, we compute $p_{\min}$ and $p_{\max}$ for each instrument and set all values of the $last$ arrays to 0. The power values of the samples are given by the drum kit in DrumGizmo. The parameters $\alpha, \beta, \gamma$ have default values that were determined experimentally. Each of them can be changed by the user, either interactively in the GUI or via the command line interface.

As DrumGizmo is free software, the exact details of the implementation can be checked by everyone.

For instruments with reasonably small sample sets, simply iterating over all samples for the specific instrument as shown in Algorithm 1 is sufficiently performant. However, imagine an instrument with an extremely large sample set. As DrumGizmo drum kits can be created by everyone, there is no restriction and we cannot assume a small sample size. To avoid performance issues arising from such a scenario, we employ a non-naive search by starting with the "most promising" sample and then inspecting its neighbors until the currently best sample is known to dominate the remaining samples. More concretely, we do the following: we start with the sample $s$ that has the closest power value $p_s$ to the requested power value $p$, i.e., we find the sample $s$ that minimizes $|p - p_s|$. The key observation why a local search often suffices is that we can lower bound the second and third summand of the objective function by 0. Thus, for a given sample $s$ and a time point $t$, we have

$$f(s,t) \geq \alpha \cdot \left( \frac{p - p_s}{p_{\max} - p_{\min}} \right)^2. \tag{1}$$

Assume now that, for some $x > 0$, we evaluated all samples $s$ with $p - x \leq p_s \leq p + x$. Then we know, by Equation 1, that for all samples $s'$ with power values outside the range $[p-x, p+x]$ it holds that

$$f(s',t) > \alpha \cdot \left( \frac{x}{p_{\max} - p_{\min}} \right)^2.$$

Note that by traversing the samples in order of their distance to $p$ (which is possible by having the samples stored in an array sorted increasingly by power value), we can stop the search as soon we searched the range $[p - x, p + x]$ for which

$$f_{\min} < \alpha \cdot \left( \frac{x}{p_{\max} - p_{\min}} \right)^2.$$

## 7. EXPERIMENTS

To conduct the experiments, we use the implementation of the new sampling algorithm in DrumGizmo. As a base-line for comparison, we use the previous sample selection algorithm of DrumGizmo. We want to evaluate how the sample selection algorithm performs in practice, therefore we use a drum kit of DrumGizmo. As normally the snare drum is the instrument with the highest number of samples, we choose this drum for our experiments. More precisely, we use the Crocell kit, which has 98 snare samples. In particular, we use the power value distribution of the samples of this kit. See Figure 4 for a visualization of the power level distribution. In this plot, we can see that the sampling is heavy on the more quiet samples and significantly sparser on the louder samples. We ask the reader to keep this distribution in mind for the remainder, when considering the experimental results. Due to this distribution, we expect that when playing

[1] The source-code is available through git at `git://git.drumgizmo.org/drumgizmo.git`, and the source code can be browsed online at `http://cgit.drumgizmo.org/drumgizmo.git/`.

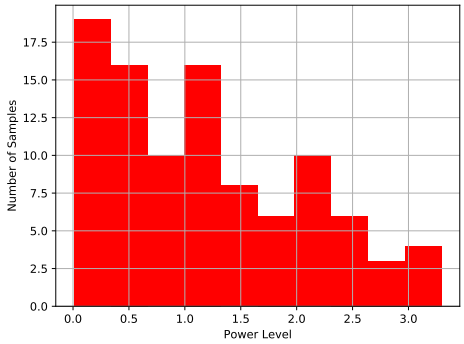


Figure 4: The power value distribution of the Crocell kit.

notes all over the power spectrum, the number of times each individual sample is played will be higher for louder samples as there are significantly less to choose from.

We want to test the following hypotheses with our experiments:

1. Two samples with similar power values are chosen similarly often.
2. Playing the same MIDI note over and over again plays a reasonably varied set of samples.

To test the above hypotheses, we conduct the following experiments:

1. Playing fast sweeps from MIDI velocity 0 to MIDI velocity 127, 8 times
2. Playing a single velocity fast for 1015 times

To be able to compare the results of these experiments, we have to somehow visualize the selections. We choose to plot histograms. More concretely, for a run of the experiment, we plot the number of times a sample was selected over its index in the array of ordered samples. For the first experiment, i.e., the sweeps, you can see the histograms in Figure 5. The plot of the old sampling algorithm clearly is less homogeneous and has larger variance than the plot of the new sampling algorithm. Especially, the old sampling algorithm uses several samples a lot while barely using others, which can lead to a more robotic sound. Especially, it seems a waste of storage and sampling effort to not use the full potential of the data.

The second experiment we conducted for two different MIDI velocities: MIDI velocity 80 (Figure 6) and MIDI velocity 112 (Figure 7). Let us first discuss Figure 6. The most significant shortcoming of the old algorithm is that the histogram does *not* roughly resemble a bell curve. Instead, we have some samples that are barely used which are very close to samples which are used predominantly. Especially, the property that samples with similar power values are used similarly often is violated. The new algorithm clearly improves on these shortcomings and gives us the desired result. In Figure 7, we can see a situation where there are just a few samples available. Here, the old algorithm and the new algorithm perform similarly, except that the new algorithm again distributes the usage of the samples more evenly. However, more importantly, even though the new algorithm does not specifically restrict to a specific power value range, it still does not choose samples that are far away from the desired power value.

To also get an idea of the performance of the new sampling algorithm, we want to see how many power values of samples are evaluated per query. Without the search optimization described at the end of Section 6, this number would always be the number of samples. However, we expect that the search optimization significantly reduces the number of evaluations. To test this hypothesis, we take the above experiment and look at the number of evaluations. Recall that the Crocell kit contains 98 snare samples and thus naively evaluating the objective function would lead to 98 evaluations. You can see the outcome of the experiment in Table 1. The mean number of evaluations is significantly less (at most 14!) than the worst-case 98 evaluations per sample selection and also the variance is very low, meaning that even for large sample sets, this sample selection algorithm should work well in real-time scenarios.

Table 1: Number of evaluations per query. The experiments with the numbers (16, 48, 80, 112) are the experiments from above of repeatedly playing a MIDI note. The number gives the MIDI velocity of this MIDI note.

| experiment | sweep | 16 | 48 | 80 | 112 |
|---|---|---|---|---|---|
| mean evaluations | 6.81 | 13.99 | 12.93 | 10.88 | 4.00 |
| stddev evaluations | 2.47 | 0.19 | 1.53 | 0.50 | 0.00 |

In summary, the experiments show that the new sampling algorithm is clearly superior to the old method that was employed in DrumGizmo and fulfills all the requirements formulated in Section 3.

## 8. CONCLUSION AND FUTURE WORK

This article presented a new algorithm for choosing samples in a setting where the samples are annotated with power values, with the desired behavior of choosing samples close to the input power value while having a reasonable diversity and no significant patterns in the sequence of selected samples. We first formulated the requirements, which lead us to an algorithm that we added to DrumGizmo. Through experiments, we showed clear improvements over the old method and also the fulfillment of the desired requirements.

However, there are still some open problems and directions to be addressed in future work. First, in this article we assumed the power values of the samples to be given. We could alter the setting by allowing a transformation on the power values, thus, for example, distributing them better over the power range. Second, the objective function that we used could still be further refined. For example, the diversity term (controlled by parameter $\beta$) could be modified to have a relatively larger penalty for samples that were played very recently, e.g., one could move away from just using linear terms. Third, having a more general view, one could try to adapt our work to instruments that are significantly different from drums. And finally, the real quality of this work should be determined by the users themselves. In this direction, a user study could lead to insights about how to further refine our approach.

# References

[1] MODARTT, *Pianoteq Website*, 2020 (accessed November 22, 2020), `https://www.modartt.com/pianoteq`.

[2] DrumGizmo Team, *DrumGizmo Website*, 2020 (accessed November 22, 2020), `https://drumgizmo.org`.

[3] *SFZ Format*, 2020 (accessed November 22, 2020), `https://sfzformat.com/`.

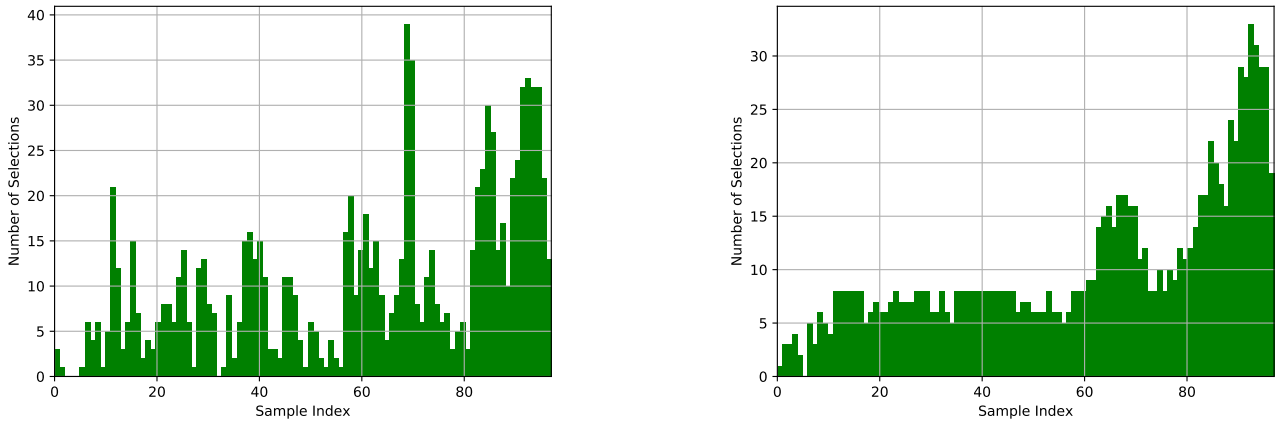


Figure 5: The histogram of the old algorithm (left) and the new algorithm (right) of playing fast sweeps from MIDI velocity 0 to MIDI velocity 127, 8 times.

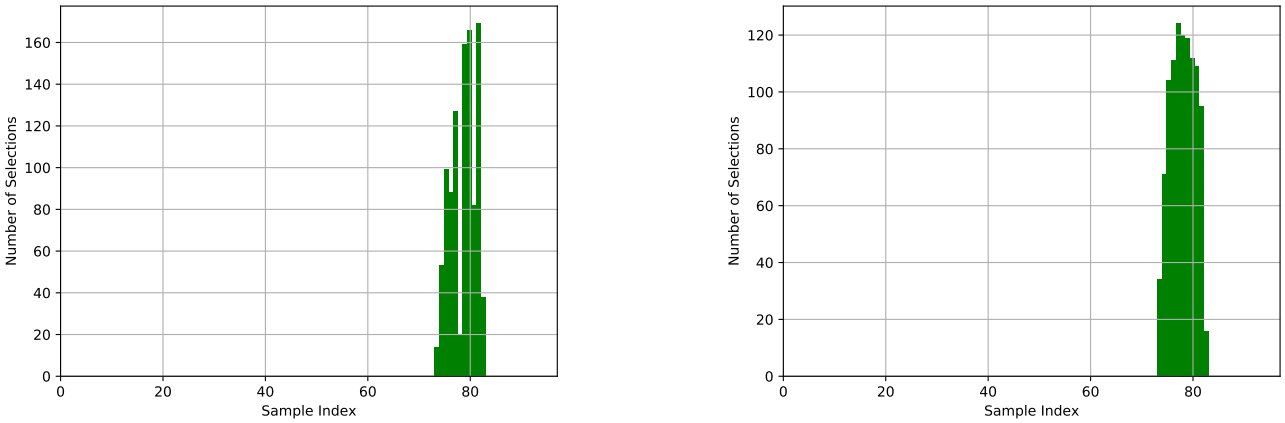


Figure 6: The histogram of the old algorithm (left) and the new algorithm (right) of playing MIDI velocity 80 fast for 1015 times.

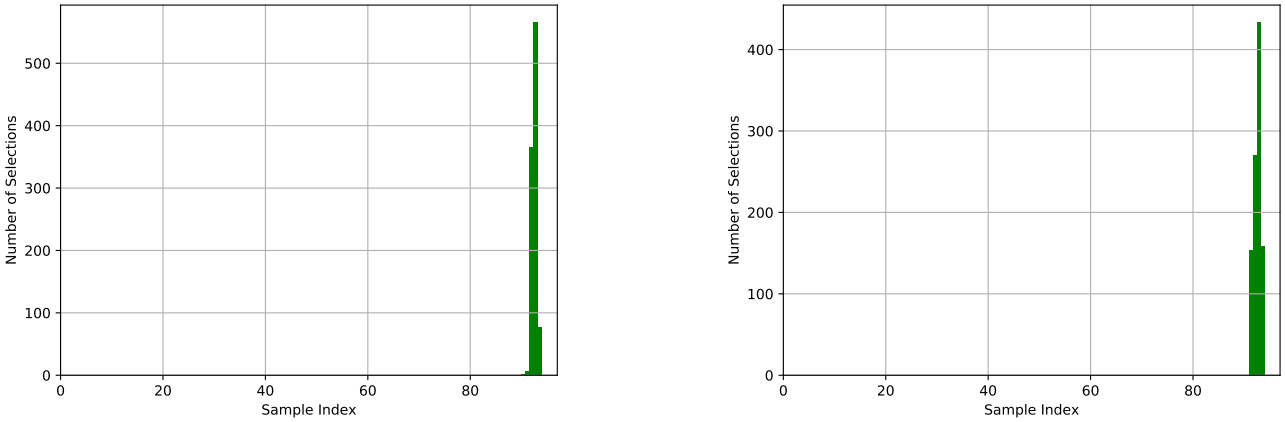


Figure 7: The histogram of the old algorithm (left) and the new algorithm (right) of playing MIDI velocity 112 fast for 1015 times.