Sound Matching in SuperCollider

Gerard Roma
School of Computing and Engineering
University of West London
gerard.roma@uwl.ac.uk

ABSTRACT

Sound matching is a classic problem in sound synthesis, originally proposed for controlling FM synthesis. Given a target sound, the problem is to find the parameters that a synthesizer could use to approximate the target. This paper describes an implementation of sound matching in SuperCollider, using the Fluid Corpus Manipulation toolbox. Given a synth definition designed by the user, the system trains a model that can approximate existing sounds by predicting synthesizer parameters. Matching can be performed for a whole buffer or continuously in real time. problem.

1. INTRODUCTION

Electronic and digital sound synthesis techniques are often developed as a dialogue between the aim of imitating real world sounds and the interest in exploring the possibilities of technology for creating new sounds. A the same time, the success of different synthesis techniques, such as subtractive, additive, wavetable or FM synthesis, has been heavily influenced by the balance between usability an sonic potential. The difficulty of programming FM synthesis for reproducing natural sounds fostered the emergence of *sound matching* [1]. This technique originally intended to automatically find the parameters required for an FM synthesizer in order to recreate an existing sound, such as a musical instrument sound. More generally, sound matching can be described as an interaction paradigm for automatic programming of a sound synthesizer based on a target sound. With the popularization of machine learning techniques for automating different tasks in music technology, interest in sound matching techniques has increased. Several works have continued to focus on the difficulty of programming FM synthesizers [2, 3, 4].

The SuperCollider (SC) synthesis server and environment offer a fertile ground for experimentation with synthesizer sound matching. The combination of a high-level language and a sound synthesis engine makes it particularly easy to experiment with different synthesis techniques. SC users typically create *synth definitions* that define the signal processing graph of a synthesizer. In this context, sound matching can be used to improve different workflows.

From a 'traditional' perspective, sound matching can be helpful to discover configurations and create presets for unpredictable synthesis techniques. In this sense, matching existing sounds can be useful when developing new synthesizers.

Sound matching can also be useful during sound design when trying to reproduce specific natural sounds. In this case, creating a synthesizer version of a sound allows further tweaking the synthesizer to create different variations, or to adapt to specific constraints such as pitch or duration.

Generally, sound matching can be seen as an interaction paradigm, where a sound is used as an intuitive interface for a complex system. This is similar to audio query-by-example (QbE) [5], where sounds similar to a target sound are retrieved from a database based on acoustic features, or concatenative synthesis [6], where the target sound is reconstructed from a database of sound particles. From this point of view, sound matching can be seen as an automatic mapping from audio features to to synthesizer parameters.

This paper describes an implementation of sound matching in SuperCollider, using the Fluid Corpus Manipulation toolbox. The next section reviews the implementations used in recent literature. Section 3 describes the proposed implementation. Section 4 describes some examples an initial impressions. There are many possibilities for further exploration. Section 5 concludes with a brief discussion and ideas for future work.

2. BACKGROUND

Early research on synthesizer sound matching was based on searching for suitable parameters using genetic algorithms (GA)[1]. Beyond searching for a set of parameters, GAs approach can also be used for generating synthesizer patches. For example, a system implemented in Pd was described by Macret and Pasquier [7]. However, such search-based approaches are inherently limited for interactive use, as each search process may take hours to complete. Most recent work has used regression models, typically neural networks, which are trained to predict the parameters of a synthesizer given the resulting sound. Once trained, the regression model can be used in a real-time context.

From an implementation perspective, model-based sound matching requires a sound synthesizer and a machine learning library. Typically, a software synthesizer is used along with Python code for the optimization.

Given the original focus on FM synthesis, several works have continued to explore the use of FM synthesis. A commonly used software plugin [2, 8, 9, 4] is *dexed* [1] , an open source clone of the Yamaha DX7.

[1] https://asb2m10.github.io/dexed/

Some studies [3, 4, 10] have focused on subtractive synthesis using Diva [2], a popular commercial virtual analogue synthesizer plugin.

Serum [3] is a commercial wavetable synthesizer that has also been used in some sound matching systems [10, 11].

It is also worth mentioning that a sound matching interface is available for the commercial synthesizer Synplant [4]

Popular synthesizers often have a large number of parameters, and this complexity supports the use case of sound matching for facilitating sound design. On the other hand, the more parameters are used, the harder the regression problem will be. Thus, in some studies a subset of parameters is used, and most works train on databases of existing presets.

In order to programmatically use the plugins from Python, some works use Renderman [2] or DawDreamer [12]. However, as noted by Barkan and Tsiris [13], in addition to the difficulties of integrating VST plug-ins with Python, using commercial synthesizers makes it more difficult to reproduce existing results. In this case, the authors developed their own synthesizer using JSyn [14]. A similar example is the project by Masuda and Saito [15] where the authors used the Faust language [16]. These projects use open-ended systems for developing synthesizers, but the implementation is specific to the presented experiments. In contrast, this paper proposes a tool for integrating sound matching into SuperCollider sound design and musicking workflows. This follows previous work that combined SuperCollider with Python [17].

3. SYSTEM

Model-based sound matching is a regression problem where the model, based on a given synthesizer, learns a mapping from a target sound to a set of parameters for the synthesizer, which it uses to approximate the target sound. This can be implemented in SuperCollider using the Fluid Corpus Manipulation Toolbox (FluCoMa)[18]. The toolbox contains several objects for feature extraction and a multilayer-perceptron (MLP) regression object (*FluidMLPRegressor*). The help file for FluidMLPRegressor already contains a small example of sound matching. This section presents a class-based implementation that offers a high-level interface for both buffer-based and real-time sound matching. An overview is shown in Figure 1. Since the FluCoMa toolbox may need to block the server for some operations, the class uses a private server for feature extraction and training.

3.1 Synth definition

The main user interface element of the system is a synth definition. Although a synthesis function could also be used, synth definitions are preferred as a more general representation. The user is expected to design or use existing synth definitions to train a synthesizer programming model. The provided definition is expected to use the main output bus as its output, which means that when recording a dataset, the different sounds generated by randomly sampled parameters can be heard. So far this has proven useful for gauging the possibilities of the input synth definition. A more specific constraint is that, for the purpose of random sampling, all parameters are assumed to range from 0 to 1 so they need to be mapped inside the definition.

[2] `https://u-he.com/products/diva/`

[3] `https://xferrecords.com/products/serum/`

[4] `https://soniccharge.com/synplant`

3.2 Feature extraction

Different features can be used to parametrize the sound coming from the synths for use by the regression model. The options are: raw audio, Mel bands or Mel frequency cepstral coefficients (MFCC). Raw audio is generally more expensive but can be feasible for short sounds. By default, MFCCs are used. Except when using raw audio, the user may also chose to aggregate features into statistics, which greatly reduces the amount of information and speed of the training process. Following existing practice, the default is to use the whole matrix of MFCCs, as this can potentially allow better estimation of temporal envelopes.

3.3 Regression

The FluidMLPRegressor object learns a mapping between the input features and the synthesizer parameters. By default, two hidden layers are used The hidden layer sizes are derived from the number of output parameters. The output layer activation is set to be a sigmoid, which works well for the parameters generated in the $[0, 1]$ range. The FluidMLPRegressor implementation uses a mean-square error (MSE) loss function, which is used for the target parameters. For the hidden layers, sigmoid functions are used as well, unless raw audio is used as a feature, in which case tanh activations are used.

After dataset creation and training, the system can be used for matching arbitrary sounds. This can be done at once for an audio buffer or in a continuous fashion.

3.4 Buffer-based matching

In buffer-based matching, the user provides a buffer, which may come from an audio file or recording. The buffer must be allocated in the private server, an have the same duration as the sounds used for training. The system predicts the synthesizer parameters to approximate the whole sound. This can be useful for mass-production of synthdefs with envelopes generators or other temporal processes controlled by few parameters.

3.5 Real-time matching

Another option is matching in real time. For this case, features are averaged both for training and prediction, so each training example is represented as a single spectral frame. The parameters of the synthesizer are thus assumed to continuously control the spectrum. For inference, in this case the system creates an analysis and prediction synth which continuously listens to a user-specified bus. This synth wraps the function in the synth definition provided for training so that it can be controlled with the predicted parameters.

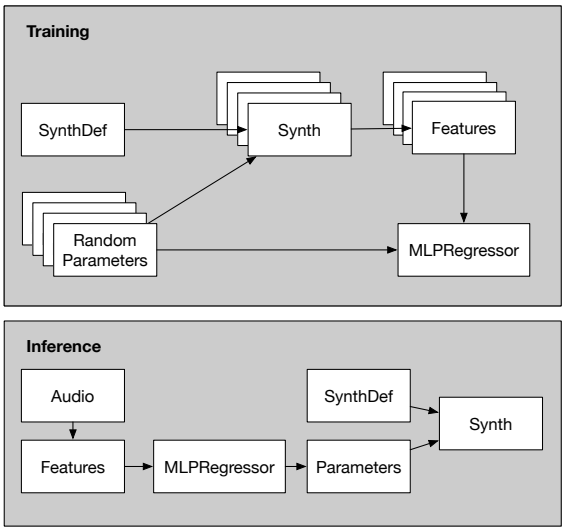

Figure 1. System diagram.

```
(
var def = SynthDef(\perc,
  {|freq1, freq2, rq1, rq2, balance, decay|
    var noise = PinkNoise.ar(0.05);
    var ring1 = Ringz.ar(noise, freq1.linlin(0,
    1, 50, 2000), rq1.linlin(0,1,0.001, 0.01));
    var ring2 = Ringz.ar(noise, freq2.linlin(0,
    1, 50, 2000), rq2.linlin(0,1,0.01, 0.1));
    var mix = (balance * ring1) + ((1-balance) *
     ring2);
    var env = Decay.ar(Impulse.ar(0), decay.
    linlin(0, 1, 0.01, 0.7),);
    Out.ar(0,  mix * env);
  }
);
```

Figure 2. Synth definition for buffer-based matching

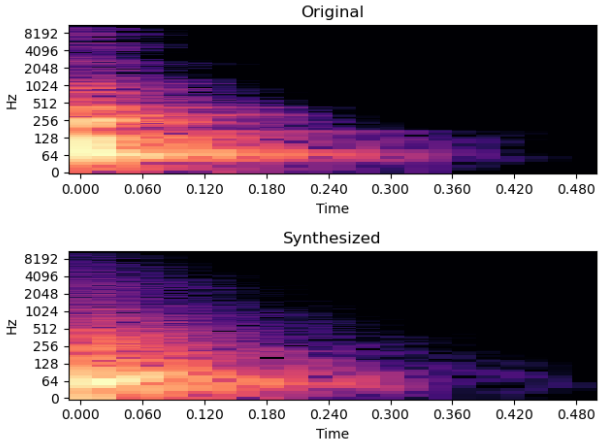

Figure 3. Percussive sounds matched from buffer

4. INITIAL IMPRESSIONS

The proposed system is implemented in a SuperColier class, *SynthSoundMatch* [5] , which requires the *FluCoMa* library. The code is still in early development stages, but can be used as a simple interface to experiment with sound matching in SuperCollider. Initial experiences have shown that, while it is still challenging to train models with complex synth definitions or produce realistic approximations of natural sounds, training models with very simple synth definitions is promising. While recent research in sound matching often employs GPU-based training of relatively heavy architectures, working within an open ended interactive creative coding environment benefits from a focus on synthesizers with very low numbers of parameters, which are easy to train.

Figure 2 shows an example of a simple synth definition that produces percussive sounds by filtering noise. A model was trained with 14400 half second random examples and then used to match a collection of recorded percussion sounds obtained from the Freesound database [6] . An example is shown in Figure 3. While the synth definition is not able to model specific partials, it can be seen that the model approximates the frequency distribution and decay time.

Figure 4 shows another synth definition using two formant oscillators. In this case, the model was trained with 3600 half a second examples which where averaged and used for real-time matching with a speech recording, also from Freesound [7] (Figure 5). The model tracks the pitch and amplitude, and some of the timbre of the original can be heard in the resulting sound.

It can be seen that the code maps the parameters from the $[0, 1]$ range internally. In general, linear mapping resulted in better performance when training the regression model, despite that some parameters would be easier to control manually with an exponential mapping.

[5] https://github.com/g-roma/SynthSoundMatch

[6] The sample pack was uploaded by user *soneproject*: https://freesound.org/people/soneproject/packs/23903/

[7] The sample uploaded by user *alphahog*: https://freesound.org/people/alphahog/sounds/46366/

5. CONCLUSIONS

This paper has described the early stages of an implementation of synthesizer sound matching in the SuperCollider environment. Adopting this methodology in an open-ended synthesis can be puzzling as, if the goal is to approximate a given target sound, there endless possibilities for obtaining interesting results through manual sound design. When training sound matching models via random sampling of parameters and CPU-based training, designing synth definitions for matching requires thinking about the target sounds while at the same time keeping the number of parameters low. In this context, a promising application of sound matching the automation of nonlinear mappings from acoustic features to synthesizer parameters, which can be used to create hybrid sounds.

Future work will explore different synthesis techniques and investigate different approaches for sampling the parameters, as well as add more options for acoustic features.

6. REFERENCES

[1] A. Horner, J. Beauchamp, and L. Haken, "Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis," *Computer Music Journal*, vol. 17, no. 4, pp. 17–29, 1993.

```
var def = SynthDef(\formants,
{|freq, ffreq1, ffreq2, amp|
  Out.ar(0, amp * Mix.new(
    Formant.ar(
      freq.linlin(0,1, 60, 200),
      [ffreq1.linlin(0, 1, 200, 900),
      ffreq2.linlin(0,1, 600, 2500)])
  ));
});
```

Figure 4. Synth definition for real-time matching

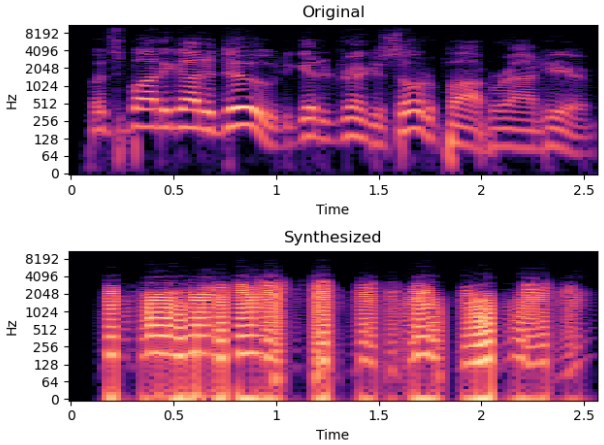

Figure 5. Sound matched in real time

[2] M. J. Yee-King, L. Fedden, and M. d'Inverno, "Automatic programming of VST sound synthesizers using deep networks and other techniques," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 2, pp. 150–159, 2018.

[3] P. Esling, N. Masuda, A. Bardet, R. Despres, A. Chemla *et al.*, "Universal audio synthesizer control with normalizing flows," in *International Conference on Digital Audio Effects (DAFx 2019)*, 2019.

[4] N. Masuda and D. Saito, "Quality Diversity for Synthesizer Sound Matching," in *2021 24th International Conference on Digital Audio Effects (DAFx)*. IEEE, 2021, pp. 300–307.

[5] M. Helén and T. Virtanen, "Audio query by example using similarity measures between probability density functions of features," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2010, pp. 1–12, 2009.

[6] A. J. Hunt and A. W. Black, "Unit selection in a concatenative speech synthesis system using a large speech database," in *1996 IEEE international conference on acoustics, speech, and signal processing conference proceedings*, vol. 1. IEEE, 1996, pp. 373–376.

[7] M. Macret and P. Pasquier, "Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 309–316.

[8] Z. Chen, Y. Jing, S. Yuan, Y. Xu, J. Wu, and H. Zhao, "Sound2synth: Interpreting sound via fm synthesizer parameters estimation," *arXiv preprint arXiv:2205.03043*, 2022.

[9] G. Le Vaillant, T. Dutoit, and S. Dekeyser, "Improving synthesizer programming from variational autoencoders latent space," in *2021 24th International Conference on Digital Audio Effects (DAFx)*, 2021, pp. 276–283.

[10] D. Faronbi, I. Roman, and J. P. Bello, "Exploring Approaches to Multi-Task Automatic Synthesizer Programming," in *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2023, pp. 1–5.

[11] C. Mitcheltree and H. Koike, "SerumRNN: Step by step audio VST effect programming," in *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*. Springer, 2021, pp. 218–234.

[12] D. Braun, "DawDreamer: bridging the gap between digital audio workstations and Python interfaces," *arXiv preprint arXiv:2111.09931*, 2021.

[13] O. Barkan and D. Tsiris, "Deep synthesizer parameter estimation," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 3887–3891.

[14] P. Burk, "JSyn - A Real-time Synthesis API for Java," in *Proceedings of the 1998 International Computer Music Conference, ICMC 1998, Ann Arbor, Michigan, USA, October 1-6, 1998*. Michigan Publishing, 1998.

[15] N. Masuda and D. Saito, "Quality-diversity for synthesizer sound matching," *Journal of Information Processing*, vol. 31, pp. 220–228, 2023.

[16] Y. Orlarey, D. Fober, and S. Letz, "Faust: an efficient functional approach to dsp programming," *New computational paradigms for computer music*, pp. 65–96, 2009.

[17] G. Roma, "Sound matching using synthesizer ensembles," in *27th International Conference on Digital Audio Effects (DAFx)*, 2024.

[18] P. A. Tremblay, G. Roma, and O. Green, "Enabling programmatic data mining as musicking: the fluid corpus manipulation toolkit," *Computer Music Journal*, vol. 45, no. 2, pp. 9–23, 2021.