Audio Engineering Society

Convention Paper 10377

Presented at the 148th Convention, 2020 June 2-5, Online

SpiegeLib: An automatic synthesizer programming library

Jordie Shier[1], George Tzanetakis[1], and Kirk McNally[1]

[1]*University of Victoria*

Correspondence should be addressed to Jordie Shier (`jshier@uvic.ca`)

ABSTRACT

Automatic synthesizer programming is the field of research focused on using algorithmic techniques to generate parameter settings and patch connections for a sound synthesizer. In this paper, we present the Synthesizer Programming with Intelligent Exploration, Generation, and Evaluation Library (`spiegelib`), an open-source, object oriented software library to support continued development, collaboration, and reproducibility within this field. `spiegelib` is designed to be extensible, providing an API with classes for conducting automatic synthesizer programming research. The name `spiegelib` was chosen to pay homage to Laurie Spiegel, an early pioneer in electronic music. In this paper we review the algorithms currently implemented in `spiegelib`, and provide an example case to illustrate an application of `spiegelib` in automatic synthesizer programming research.

1 Introduction

The use of sound synthesizers in the fields of music composition, production, and performance is widespread, but the task of programming a synthesizer is complex and requires a thorough understanding of technical details. It is not uncommon for a software synthesizer to have 30+ parameters displayed on a user interface (UI) and labelled using technical names [1]. Manually programming sounds using such a large set of parameters is a daunting task. Synthesizer programming is further complicated by the fact that modifications to parameters are often not intuitively reflected in the end sonic result. This disconnect can be disruptive to the creative process. Automatic synthesizer programming (ASP) is the field of research focused on addressing these challenges in programming synthesizers.

Early ASP research emerged in the late 1970s with work that focused on the use of analytic methods to estimate the parameters for frequency modulation (FM) synthesis [2]. That work was an example of synthesizer sound matching in which a system estimates synthesizer parameters to replicate a target sound. Since then a large volume of work on synthesizer sound matching has been published and has explored a variety of synthesis techniques and algorithmic methods. One popular approach is the use of evolutionary algorithms [3, 4, 5, 6, 7, 8, 9, 10]. More recently, deep learning techniques have been explored [11, 12, 13]. Other methods that have been studied include semantic descriptions [14, 15, 16], interactive methods [17, 18, 19], and sound matching with vocal imitations [20, 21].

A recent user study conducted by Kreković et al. confirmed the desire among synthesizer users for improved means of working with their synthesizers [22]. While recent research has produced promising results, automatically programming a modern software synthesizer still presents challenges. Current evolutionary techniques face issues including time complexity [9], while recent deep learning approaches have challenges in consistently producing accurate reproductions [11]. The desires expressed by the users in Kreković et al.'s study, coupled with the need for further research and improvement noted in the existing body of work, point to the need for further development in ASP.

The work presented here attempts to continue this development, and promote collaboration and reproducibility in ASP research through the introduction of `spiegelib`, an open-source library written in the Python programming language. Vandewalle et al. argue that reproducibility in computational science research increases the impact of a work and they provide a framework for evaluating the quality of reproducibility [23]. The aim of `spiegelib` is to provide a platform for researchers of automatic synthesizer programming to develop, test, and share implementations in a way that promotes reproducibility at the highest level. `spiegelib` stands for Synthesizer Programming with Intelligent Exploration, Generation, and Evaluation Library. The name `spiegelib` was chosen to pay homage to Laurie Spiegel, an early pioneer in electronic music composition. Laurie Spiegel is known for utilizing synthesizers and software to automate certain aspects of the music composition process. Her philosophy for using technology in music serves as a motivation for the `spiegelib` software library: "I automate whatever can be automated to be freer to focus on those aspects of music that can't be automated. The challenge is to figure out which is which." [24]

The remainder of this paper is structured as follows: Section 2 presents a survey of related work to provide background for the algorithmic techniques included in `spiegelib` as well as an overview of available open-access ASP software, Section 3 provides an overview of the `spiegelib` software library. An example case illustrating the use of `spiegelib` in a research context is presented in Section 4.

2 Related Work

This section provides a brief summary of the main algorithmic methodologies that have been used in previous ASP research, namely, optimization and deep learning techniques. Other methods that have been used in ASP research that are beyond the scope of this paper include include fuzzy logic [25, 26], linear coding [27], and query approaches [20]. An informal survey of open-access software that supports reproducibility is also included at the end of the section.

2.1 Optimization

The optimization approach was first introduced in 1993 with Horner et al.'s work on sound matching for FM synthesis using genetic algorithms [3]. A genetic algorithm (GA) is a method for solving an optimization problem using techniques based on the principles of Darwinian evolution, and is part of a broader class of evolutionary algorithms [28]. In a GA, a potential solution (an individual) is represented as an array of bits. An initial set of individuals is randomly generated, and then iteratively evolved using biologically inspired processes including selection, breeding, and mutation. Individuals are ranked using an evaluation function that measures the $fitness$ of a given solution. The objective of a GA is to minimize that value (or maximize it, depending on the problem definition). The best candidates are selected for further evolution until either an optimal solution is found or a set number of evolutions has been completed.

In the case of sound matching, the *fitness* of a potential solution is determined by measuring the error between a target sound and a candidate. Typically, an audio transform or audio feature extraction is performed prior to calculating *fitness*. The first works on synthesizer sound matching with GAs used the Short Time Fourier Transform (STFT) in the evaluation function [3, 29]. Mel-frequency Cepstral Coefficients (MFCCs), an audio representation using a non-linear frequency scaling that is more relevant to human hearing, have also been used [6, 8, 30, 10]. Tatar et al. introduced the use of a multi-objective GA for synthesizer sound matching that used three different methods for calculating $fitness$ values: the STFT, Fast Fourier Transform (FFT), and signal envelope [9]. Alternatives to GAs that have been used for sound matching include Particle Swarm Optimization (PSO) [7] and Hill-Climbing [8, 31].

Researchers have also used Interactive Genetic Algorithms (IGAs) that allow users to interactively hear and rate potential synthesizer patches [17, 18, 19]. In contrast to the sound matching case, the evaluation function in an IGA relies on user feedback during each iteration as opposed to measuring error between a candidate and a target.

Automatic programming using semantic sound descriptions has also been explored, and is a further methodology that has used GAs [16].

2.2 Deep Learning

Deep learning is subset of machine learning that utilizes artificial neural networks to learn patterns in data and make predictions based on those patterns [32]. Deep learning architectures contain multiple layers comprised of simple non-linear modules. Through iterative training, the layers are able to extract features from raw input data and learn intricate patterns in high-dimensional data. These multi-layer architectures have enabled deep learning models to excel at complex tasks including image recognition, speech recognition, and music related tasks such as audio source separation [33].

In the context of an ASP sound matching experiment, a deep learning model accepts an audio signal as input and predicts synthesizer parameter settings to replicate that audio signal. Audio signals are often preprocessed using audio feature extraction or an audio transform. Models are trained using a large set of example sounds generated from a synthesizer and use the parameter settings that generated a particular sound as the ground truth. During training, the error between predicted parameter settings and the actual parameter settings (the ground truth) are used to evaluate how well the model is learning and to iteratively update variables within the model to improve performance.

Several researchers have explored the use of deep learning for ASP. Yee-King et al. reviewed several deep learning architectures for FM synthesizer sound matcing [11]. In their work, they compared multi-layer perceptron (MLP), Long Short Term Memory (LSTM), and LSTM++ networks. Barkan et al. explored sound matching using convolutional neural networks (CNNs) [12]. They framed the problem as an image classification task and used the STFT to create spectrogram images of target sounds to use as input to the CNNs. Esling et al. recently presented a novel application called *FlowSynth* that uses a generative model based on Variational Auto-Encoders and Normalizing Flows [13]. In addition to performing well on sound matching tasks, they also showed that their approach supported novel interactions including macro-control of synthesizer parameters.

2.3 Software in ASP Research

In Vandewalle et al.'s paper on reproducibility in computational sciences, they advocate providing other researchers with "all the information (code, data, schemes, etc.) that was used to produce the presented results"[23]. Several authors of ASP research have started to make their work open-access with source code available online.

Martin Roth and Matthew Yee-King developed *JVstHost*, a Java-based Virtual Studio Technology (VST) plugin host that was published by Matthew Yee-King [34] and was a component of *SynthBot* [6]. However, the code for *SynthBot* itself was not released. Matthew Yee-King also shared the source code for *EvoSynth*, an application for interactive synthesizer patch exploration [19]. A version of *EvoSynth* is hosted online allowing for immediate experimentation[1]. Kreković et al. released source code for their *MightyKnob* system [16]. Esling et al. released open-source code and a Max4Live[2] application for *FlowSynth* [13]. Yee-King et al. recently took initial steps towards a software framework for ASP research with the release of source code that provides functionality for generating research datasets and a set of algorithms for parameter estimation [11]. Along with that work they released the *RenderMan*[3] library for programmatically interacting with VST synthesizers using the Python programming language.

`spiegelib` builds upon this work with the goal of supporting and encouraging reproducibility within the ASP research community. `spiegelib` is inspired by the steps that Yee-King et al. took towards creating a software library for ASP research and extends that work with the inclusion of: an object-oriented API, base classes for customization, more robust evolutionary techniques, basic subjective evaluation, complete documentation, and packaging and delivery. It provides a framework for authors to share implementations in an open-access way that allows other researchers to quickly recreate results using a clearly documented set of freely-available tools.

[1]`http://www.yeeking.net/evosynth/`
[2]`https://www.ableton.com/en/live/max-for-live/`
[3]`https://github.com/fedden/RenderMan`

Table 1: Algorithms currently implemented as classes in spiegelib

Algorithms in **spiegelib**		
Feature Extraction	**Deep Learning Estimators**	**Optimization Estimators**
FFT	MLP [11]	Basic GA
STFT	LSTM [11]	NSGA III [9]
MFCC	LSTM++ [11]	**Objective Evaluation**
Spectral[1]	Conv6 [12]	MFCC Evaluation

[1] Spectral bandwidth, centroid, contrast, flatness, and rolloff.

3 Design of SpiegeLib

spiegelib is designed to be as extensible as possible to allow researchers to develop and test new implementations of components for conducting ASP research. Base classes with functionality for interacting with software synthesizers, audio feature extraction, parameter estimation, and evaluation provide an API to support development of custom implementations that will work with other components of the library. A number of utility classes are also provided for handling audio signals, generating datasets, and running experiments.

spiegelib is written in the Python programming language and utilizes Python packages common in research including numpy, scipy, tensorflow, and librosa. spiegelib itself is a python package and is available through the Python Package Index (PyPI) with pip[4]. All dependencies, except for librenderman, are python packages available through the PyPI and will be automatically installed by pip. For more information on installation, system requirements, and detailed library documentation, please refer to the online documentation.[5]

A summary of the currently implemented algorithms is shown in table 1. A brief overview of these components and the main classes and functionalities of spiegelib is provided in the following sections.

3.1 AudioBuffer

The AudioBuffer class is used to pass audio signal signals throughout the library. It holds an array of audio samples and sample rate information. Methods of the AudioBuffer class provide functionality for loading audio from a variety of file formats, resampling, normalizing, time segmenting, plotting spectrograms, and saving audio as WAV files.

[4]https://pypi.org/

[5]https://spiegelib.github.io/spiegelib/

3.2 Synthesizers

The SynthBase class is an abstract base class that provides an interface for creating programmatic interactions with software synthesizers. SynthBase stores information and contains methods required for interaction with other components in spiegelib, including getting parameter lists, setting and getting patch configurations, overriding/freezing parameters, triggering audio rendering using MIDI notes, getting audio samples as AudioBuffers, and requesting randomized patch settings. All patch settings are stored as a list of parameter tuples which contain the parameter number and parameter value. All parameter values are expected to be floating point numbers in the range [0.0, 1.0]. No requirement is made on how underlying synthesis engines are implemented, however, inheriting classes must provide parameter descriptions in a class attribute during construction and must provide implementations for four abstract class methods related to loading patches, randomizing patches, rendering audio, and returning an AudioBuffer of rendered audio.

SynthVST is an implementation of SynthBase and provides an interface for interacting with VST synthesizers. SynthVST is a wrapper for the *RenderMan* Python library developed by Leon Fedden in conjunction with research by Yee-King et al. [11].

3.3 Audio Feature Extraction

The abstract base class FeaturesBase provides an interface for audio feature extraction tasks. The getFeatures() abstract method must be overridden in inheriting classes and is where feature extraction algorithms are run. FeatureBase also includes functionality for normalizing results from feature extraction. By default, data is normalized by removing the mean and scaling to unit variance. Settings for normalization can be saved from a set of data, reloaded, and applied to new feature extraction results to ensure that normalization is carried out using the same parameters. Currently, implemented feature extraction classes utilize the `librosa` library [35] and include Mel Frequency Cepstral Coefficients (`MFCC`), Short Time Fourier Transform (`STFT`), Fast Fourier Transform (`FFT`), and a set of time summarized spectral features (`SpectralSummarized`).

3.4 Estimators

All parameter estimation classes implement the `EstimatorBase` abstract base class. `EstimatorBase` is a minimal base class with one abstract method, `predict()`, that has an optional input argument. Implementations of estimators are split into deep learning approaches and other approaches including evolutionary algorithms. The included algorithms do not represent a comprehensive set of methods for ASP research but are meant to cover common methods informed by previous work. Six estimators are currently implemented and the authors plan to add 5 more in the near future: a hill climbing optimizer [11], a particle swarm optimizer [7], additional configurations of 2D CNNs [12], a 1D CNN for raw audio input [12], and a recent generative approach [13]. The authors hope that other researchers will add their new algorithms to the library as well.

3.4.1 Deep Learning Estimators

All deep learning models are implementations of the `TFEstimatorBase` abstract base class which utilizes the `tensorflow`[6] and `keras`[7] machine learning libraries. `TFEstimatorBase` implements `EstimatorBase` and provides wrapper functions for setting up data for training and validation, training models, running predictions, and saving and loading model weights. While these methods are designed to help in handling of data typical to a synthesizer parameter estimation problem, all methods for a `tf.keras.Model` can be accessed directly from the `model` class member. Classes that inherit from `TFEstimatorBase` define models in an implementation of the `buildModel()` method which is automatically called during construction in the base class. This allows new models to be quickly designed, switched out, and compared with minimal effort.

[6] `https://www.tensorflow.org`
[7] `https://www.tensorflow.org/guide/keras`

Fig. 1: Example of `spiegelib` performing a sound match from a target WAV file on a VST synthesizer. A pre-trained LSTM deep learning model is used with MFCC input.

```python
import spiegelib as spgl
import spiegelib.estimator.TFEstimatorBase

# Load VST and set parameters from JSON file
synth = spgl.synth.SynthVST('./Dexed.vst')
synth.load_state('./dexed_simple_fm.json')

# MFCC Audio Feature Extractor
ftrs = spgl.features.MFCC(normalize=True)

# Load saved normalization parameters
ftrs.load_normalizers('./normalizers.pkl')

# Load LSTM model from saved model file
lstm = TFEstimatorBase.load('./fm_lstm.h5')

matcher = spgl.SoundMatch(synth, lstm, ftrs)

target = spgl.AudioBuffer('./target.wav')
output = matcher.match(target)
output.save('./lstm_predicted_audio.wav')
```

Currently, implementations for a Multi-Layer Perceptron (`MLP`), Long Short Term Memory (`LSTM`), Bi-directional Long Short Term Memory with Highway Layers (`HwyBLSTM`), and a convolution network with 6-layers (`Conv6`) are included. An example code listing of sound matching using a trained LSTM model is shown in figure 1.

To save training and validation progress, the `TFEpochLogger` class can be passed in as a callback during model training. `TFEpochLogger` stores training accuracy and loss, and validation accuracy and loss over training epochs in a dictionary object which can be plotted after training.

3.4.2 Optimization Estimators

Two optimization estimators are currently implemented and utilize the DEAP python library [36]. A basic GA (`BasicGA`) is included as well as a multi-objective non-dominated sorting genetic algorithm III (`NSGA3`). Both GAs require feature extraction objects, or a list of feature extraction objects in the case of the multi-objective algorithm, which are used in the GA evaluation function.

3.5 Datasets

The `DatasetGenerator` class provides functionality for creating datasets of audio samples, feature vectors, and associated parameter settings from a synthesizer. An implementation of `SynthBase` and `FeaturesBase` are passed in as arguments to the `DatasetGenerator` constructor. To generate a dataset, random patches for the synthesizer are created and feature extraction is performed on the resulting audio. In this way, datasets for training and validating deep learning models, as well as datasets for evaluating sound matching experiments can be automatically generated. External datasets can also be used within `spiegelib` and the `AudioBuffer` class provides support for loading folders of audio samples for processing.

3.6 Evaluation

Objective evaluation of results can be carried out by measuring error between audio samples. `EvaluationBase` is an abstract base class for calculating evaluation metrics on a set of target and prediction data. A list of target values and lists of predictions for each target are passed into the constructor. `EvaluationBase` provides functionality for calculating statistics on results, saving results as a JSON file, plotting results in histograms, and calculating metrics including mean absolute error, mean squared error, euclidian distance, and manhattan distance. Inheriting classes must implement the `evaluate_target()` method which is called for each target and associated estimations and is expected to return a dictionary of metrics for each estimation. The `MFCCEval` class implements `EvaluationBase` and calculates metrics on MFCC vectors for targets and estimations.

Functionality for conducting subjective evaluation of results is provided in the `BasicSubjective` class. This class accepts a set of audio files and runs a locally hosted server that generates a simple web interface for listening to and ranking audio files in terms of similarity or preference. For sound matching experiments, audio targets can be passed in along with a set of predictions for each target, and a sound similarity test will be generated with options for randomizing the ordering of targets and predictions. Results can then be saved as a JSON file.

4 Example Case

4.1 Dexed: FM Sound Matching

To illustrate the use of `spiegelib` in the context of an ASP sound matching experiment, we present an example study that compares six different parameter estimators acting on *Dexed*[8], a VST software emulation of the Yamaha DX7 FM synthesizer. The methodology used for this experiment is modelled after Yee-King et al.'s study on deep learning for automatic synthesizer programming [11], but with a simplified synthesizer configuration and a unique set of estimators. In alignment with Vandewalle et al.'s criteria for reproducible research, implementation details, code, and datasets are all available on the `spiegelib` online documentation.[9]

The first step is defining the synthesizer setup and creating data for training deep learning models. *Dexed* has 155 parameters controllable through the `SynthVST` class. A subset of nine of these parameters were used in this experiment to turn *Dexed* into a simple two-oscillator FM synthesizer. The `SynthVST` class provides methods for overriding and freezing parameters as well as saving and loading parameter settings as JSON files.

The `DatasetGenerator` class was then used to create datasets for deep learning training. All deep learning models, except for the CNN, used a 13-band MFCC calculated with a frame size of 2048 samples and a hop size of 1024 samples. The input of the CNN was the magnitude spectrum from a STFT, calculated using an FFT with 512 bins and a hop size of 256 samples. Using an instance of `DatasetGenerator`, 50,000 training examples and 10,000 validation examples were generated by randomly sampling the nine parameters from *Dexed*. Resulting feature vectors were normalized by removing the mean and scaling to unit variance using methods within the `FeatureBase` base class. Datasets and settings for normalization were then stored as NumPy[10] files for later use.

All deep learning models are defined in classes within `spiegelib` and all inherit from `TFEstimatorBase`. Models were trained using an *Adam* optimizer [37], batch sizes of 64, and

[8] `https://asb2m10.github.io/dexed/`
[9] `https://spiegelib.github.io/spiegelib/examples/fm_sound_match.html`
[10] `https://numpy.org/`

Table 2: Results from sound matching evaluation

Method	*Mean*	*SD*	*Min*	*Max*
MLP	8.55	6.77	1.92	34.12
CNN	7.88	4.26	2.68	20.89
LSTM	6.12	3.76	1.20	19.36
LSTM++	4.91	6.50	2.12	21.51
GA	2.25	2.58	0.70	11.17
NSGAIII	**0.81**	**0.89**	**0.001**	**3.06**

Values shown are calculated from the mean absolute error (MAE) calculated during MFCC evaluation. Smaller MAE values indicate more similar matches. The NSGA III estimator received the best scores, which are shown in bold font.

used an early stopping callback which halted training if validation loss was stagnant for 10 epochs to prevent overfitting. Logging of training and validation progress was recorded using the `TFEpochLogger` class. The loss function in all models measure the RMS error between ground truth parameter settings and prediction parameter settings.

Two GAs were used: a basic single-objective GA (`BasicGA`) and a multi-objective NSGA III (`NSGA3`). Evaluation functions for the GAs use audio feature extraction and measure the mean absolute error (MAE) between the target and individuals to calculate *fitness*. The `BasicGA` used a 13-band MFCC in the evaluation function and the `NSGA3` used three different extractors: a 13-band MFCC, STFT, and five spectral features. Both genetic algorithms were run for 100 generations for each sound target.

An evaluation dataset containing 25 random sounds from the same nine-parameter *Dexed* configuration was generated using the `DatasetGenerator` class. All estimators were run on each one of the 25 target sounds using the `SoundMatch` class. `SoundMatch` is a functional class that uses an estimator to predict synthesizer parameter settings for an implementation of `SynthBase` in order to match a target sound. This resulted in a set of audio files generated from *Dexed* using the estimated parameters from each estimator run on each of the 25 target sounds. These audio files were then used for objective evaluation.

To evaluate to the resulting predictions the `MFCCEval` class was used, which calculates error and distance metrics on MFCCs of a target and prediction. Results for mean absolute error (MAE) which have been summarized using mean, standard deviation, minimum, and maximum, are shown for each estimator in table 2. Both GAs performed better than the deep learning approaches with the NSGA III having the best overall score. For deep learning approaches, the LSTM++ model achieved the best mean score. Histograms of the the MAE were also plotted for each estimator using the `plot_hist()` method in `EvaluationBase`. Histograms of the MAE for all predictions made by all estimators are shown in figure 2. Spectrograms of one target sound and sound match predictions made by each of the estimators for that target are shown in figure 3. For this particular target, spectrograms reveal that while the frequency and distribution of the harmonics was relatively close for each estimation, all estimators except for the NSGA III struggled with matching the temporal envelope of the spectrum.

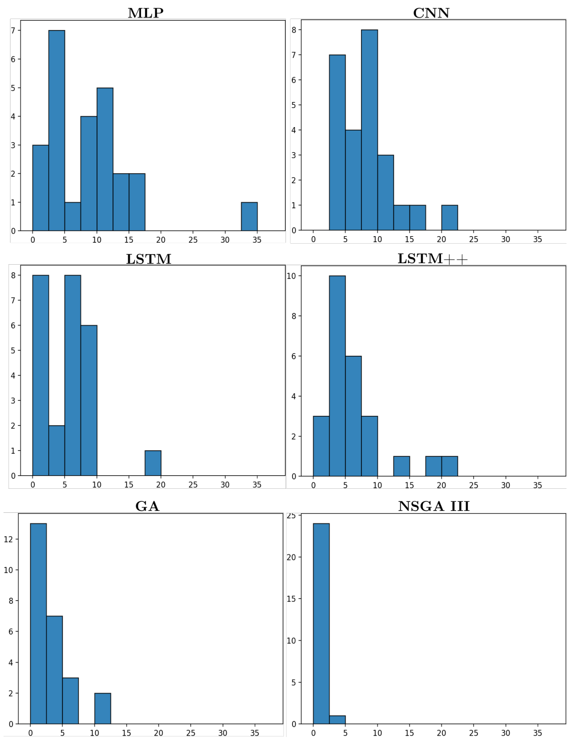

Fig. 2: Histogram shows the MAE values resulting from MFCC evaluation run on a set of 25 sound targets for all estimators. Lower MAE values indicate a closer sound match.

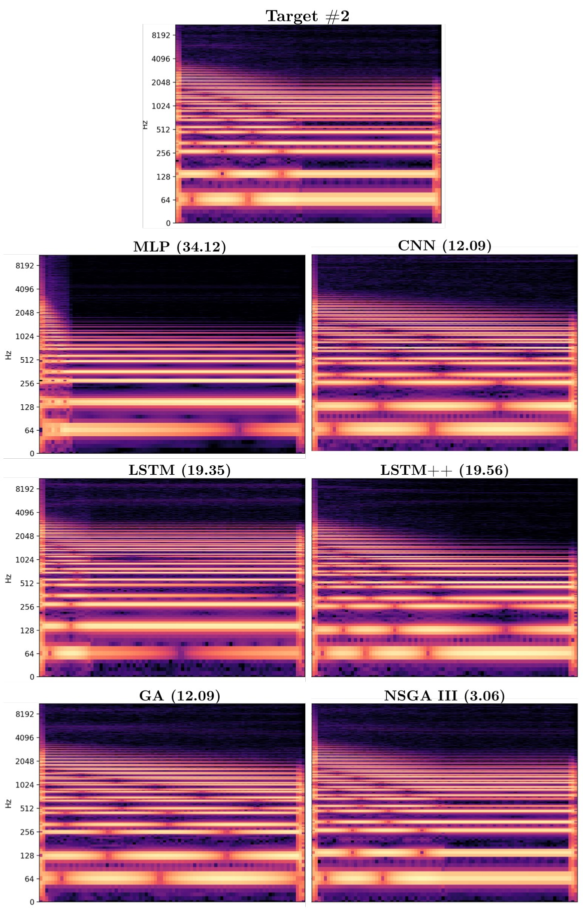

Fig. 3: Spectrogram plots of a target sound and sound match predictions made by each estimator. The value next to the estimator name is the MAE value from MFCC evaluation for that prediction (lower MAE values indicate a closer match).

5 Future Work and Conclusion

Development of `spiegelib` is ongoing and a number of expansions to the current library are planned. First, we would like to continue to expand the number of estimators available and plan on integrating the following: a hill climbing optimizer [11], a particle swarm optimizer [7], more 2D CNN configurations [12], a 1D CNN for raw audio input [12], and a generative approach [13]. Second, we would like to expand on the type of interactions available such as automatic programming from vocal imitations [20] and interactive methods. Finally, we would like to encourage developers and researchers from the automatic synthesizer programming community to contribute to `spiegelib`. Information on contributing is available online.[11]

This work has introduced `spiegelib`, an open-source automatic synthesizer programming library. `spiegelib` is an object-oriented library that was designed with the goal of supporting development, collaboration, and reproducibility in the field. The library includes implementations of classes for conducting ASP research. These classes contain functionality for interacting with VST synthesizers, extracting audio features, creating datasets, estimating synthesizer parameters, and evaluating results. Six implementations of deep learning and evolutionary parameter estimation techniques based on previous work are included, with more planned. An example case of an automatic synthesizer sound matching study using the library was shown. This example case, along with the supporting code and data available online showcases how `spiegelib` can be used to support reproducible research.

This work was supported by a NSERC CGS-M fellowship. Thank you to Colin Malloy for his help with testing.

References

[1] Rasmussen, C., *Evaluating the Usability of Software Synthesizers: An Analysis and First Approach*, Master's thesis, 2018.

[2] Justice, J., "Analytic signal processing in music computation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(6), pp. 670–684, 1979.

[3] Horner, A., Beauchamp, J., and Haken, L., "Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis," *Computer Music Journal*, 17(4), pp. 17–29, 1993.

[4] Mitchell, T. J. and Creasey, D. P., "Evolutionary sound matching: A test methodology and comparative study," in *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*, pp. 229–234, IEEE, 2007.

[5] Yee-King, M. J., "The evolving drum machine," in *Music-AL workshop, ECAL conference*, volume 2007, 2007.

[11] `https://spiegelib.github.io/spiegelib/contributing.html`

[6] Yee-King, M. and Roth, M., "Synthbot: an Unsupervised Software synthesizer Programmer." in *Proceedings of the International Computer Music Conference*, 2008.

[7] Heise, S., Hlatky, M., and Loviscach, J., "Automatic cloning of recorded sounds by software synthesizers," in *Audio Engineering Society Convention 127*, Audio Engineering Society, 2009.

[8] Roth, M. and Yee-King, M., "A comparison of parametric optimization techniques for musical instrument tone matching," in *Audio Engineering Society Convention 130*, Audio Engineering Society, 2011.

[9] Tatar, K., Macret, M., and Pasquier, P., "Automatic synthesizer preset generation with PresetGen," *Journal of New Music Research*, 45(2), pp. 124–144, 2016.

[10] Smith, B. D., "Play it Again: Evolved Audio Effects and Synthesizer Programming," in *International Conference on Evolutionary and Biologically Inspired Music and Art*, pp. 275–288, Springer, 2017.

[11] Yee-King, M. J., Fedden, L., and d'Inverno, M., "Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2), pp. 150–159, 2018.

[12] Barkan, O., Tsiris, D., Katz, O., and Koenigstein, N., "InverSynth: Deep Estimation of Synthesizer Parameter Configurations From Audio Signals," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 27(12), pp. 2385–2396, 2019.

[13] Esling, P., Masuda, N., Bardet, A., Despres, R., and Chemla-Romeu-Santos, A., "Flow Synthesizer: Universal Audio Synthesizer Control with Normalizing Flows," *Applied Sciences*, 10(1), p. 302, 2020.

[14] Ethington, R. and Punch, B., "SeaWave: A system for musical timbre description," *Computer Music Journal*, 18(1), pp. 30–39, 1994.

[15] Johnson, C. G. and Gounaropoulos, A., "Timbre interfaces using adjectives and adverbs," in *Proceedings of the 2006 conference on New interfaces for musical expression*, pp. 101–102, IRCAM, 2006.

[16] Kreković, G., Pošćić, A., and Petrinović, D., "An algorithm for controlling arbitrary sound synthesizers using adjectives," *Journal of New Music Research*, 45(4), pp. 375–390, 2016.

[17] Johnson, C. G., "Exploring the sound-space of synthesis algorithms using interactive genetic algorithms," in *Proceedings of the AISB'99 Symposium on Musical Creativity*, pp. 20–27, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1999.

[18] Dahlstedt, P., "Creating and Exploring Huge Parameter Spaces: Interactive Evolution as a Tool for Sound Generation." in *ICMC*, 2001.

[19] Yee-King, M. J., "The use of interactive genetic algorithms in sound design: a comparison study," *ACM Comput. Entertainment*, 14(3), 2016.

[20] Cartwright, M. and Pardo, B., "SynthAssist: Querying an Audio Synthesizer by Vocal Imitation," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 363–366, 2014.

[21] Zhang, Y. and Duan, Z., "Visualization and interpretation of Siamese style convolutional neural networks for sound search by vocal imitation," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2406–2410, IEEE, 2018.

[22] Kreković, G., "Insights In Habits and Attitudes Regarding Programming Sound Synthesizers: A Quantitative Study," in *Proceedings of the 16th Sound and Music Computing Conference*, 2019.

[23] Vandewalle, P., Kovacevic, J., and Vetterli, M., "Reproducible research in signal processing," *IEEE Signal Processing Magazine*, 26(3), pp. 37–47, 2009.

[24] Hinkle-Turner, E., *Women Composers and Music Technology in the United States: Crossing the Line*, Ashgate Publishing, Ltd., 2006.

[25] Mitchell, T. and Sullivan, C., "Frequency Modulation Tone Matching Using a Fuzzy Clustering

Evolution Strategy," in *Audio Engineering Society Convention 118*, 2005.

[26] Hamadicharef, B. and Ifeachor, E. C., "Intelligent and perceptual-based approach to musical instruments sound design," *Expert Systems with Applications*, 39(7), pp. 6476–6484, 2012.

[27] Mintz, D., *Toward timbral synthesis: a new method for synthesizing sound based on timbre description schemes*, Master's thesis, Citeseer, 2007.

[28] Whitley, D., "A genetic algorithm tutorial," *Statistics and computing*, 4(2), pp. 65–85, 1994.

[29] Horner, A., "Wavetable matching synthesis of dynamic instruments with genetic algorithms," *Journal of the Audio Engineering Society*, 43(11), pp. 916–931, 1995.

[30] Macret, M. and Pasquier, P., "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*, pp. 309–316, ACM, 2014.

[31] Luke, S., "Stochastic Synthesizer Patch Exploration in Edisyn," in *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pp. 188–200, Springer, 2019.

[32] LeCun, Y., Bengio, Y., and Hinton, G., "Deep learning," *nature*, 521(7553), pp. 436–444, 2015.

[33] Hennequin, R., Khlif, A., Voituret, F., and Moussallam, M., "Spleeter: A Fast And State-of-the Art Music Source Separation Tool With Pre-trained Models," Late-Breaking/Demo ISMIR 2019, 2019, deezer Research.

[34] Yee-King, M. J., *Automatic sound synthesizer programming: techniques and applications*, Ph.D. thesis, University of Sussex, 2011.

[35] McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., and Nieto, O., "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th python in science conference*, volume 8, 2015.

[36] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., and Gagné, C., "DEAP: Evolutionary Algorithms Made Easy," *Journal of Machine Learning Research*, 13, pp. 2171–2175, 2012.

[37] Kingma, D. P. and Ba, J., "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.