THE FEATSYNTH FRAMEWORK FOR FEATURE-BASED SYNTHESIS: DESIGN AND APPLICATIONS

Matthew Hoffman
Princeton University
Department of Computer Science

Perry R. Cook
Princeton University
Department of Computer Science (also Music)

ABSTRACT

This paper describes the FeatSynth framework, a set of open-source C++ classes intended to make it as easy as possible to integrate feature-based synthesis techniques into audio software. We briefly review the key ideas behind feature-based synthesis, and then discuss the framework's architecture. We emphasize design choices meant to make the framework more flexible and straightforward to use. A number of illustrative examples of applications and tools developed using FeatSynth are presented to highlight different ways in which the API can be used. These examples include a command-line system for performing (among other things) offline non-phonorealistic analysis-synthesis transformations, a system allowing users to interactively manipulate the feature values used to control synthesis, and a ChucK FeatSynth plugin making use of the new "Chugin" ChucK plugin framework. The framework can be downloaded from http://featsynth.cs.princeton.edu.

1. INTRODUCTION

Feature-based synthesis is a technique for synthesizing audio characterized by certain quantitative, automatically extractable characteristics, or features [3]. Each of these features, which are commonly taken from the music information retrieval literature (see, e.g., [2]), is intended to automatically capture and encode some acoustic quality of recorded audio corresponding to a dimension of human auditory perception. The goal of feature-based synthesis is to synthesize audio characterized by a pre-specified set of feature values.

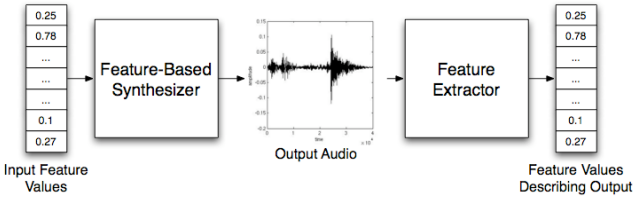

Figure 1. Given a vector of input feature values, we try to synthesize audio that, when analyzed by the chosen feature extractor, will produce feature values similar to the input feature values.

Developing a feature-based synthesis system is not a trivial process even using the frameworks integrating analysis and synthesis that do exist (e.g. CLAM [1] and MARSYAS [4]). The FeatSynth framework, a set of open-source C++ classes and tools, is intended to make developing feature-based synthesizers and integrating them into musical applications as easy as possible.

2. FEATURE-BASED SYNTHESIS

We treat frame-level feature-based synthesis as a combinatorial optimization problem, where the objective is to find a set of synthesis parameters for a given parametric synthesizer that produce a frame of audio samples characterized by the desired feature values. Although for some pairings of feature set and synthesis algorithm a closed-form solution to this problem exists, often such a solution is not available and we must use some sort of iterative heuristic search technique to find a near-optimal set of parameters.

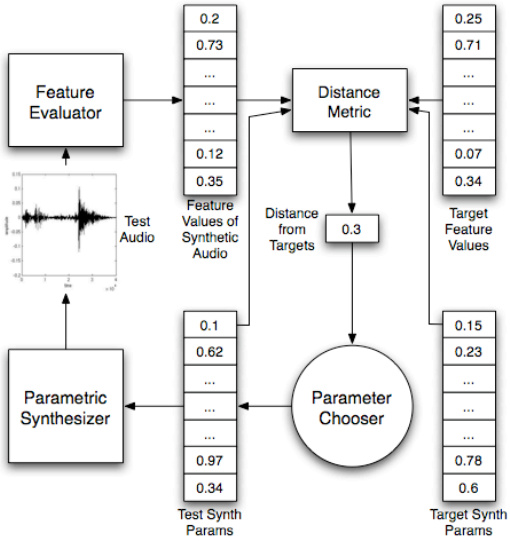

Figure 2. Optimization loop. Test parameters are used to generate audio, which is then analyzed. The resulting feature values are compared with the desired values, and new test parameters are chosen according to some heuristic and evaluated.

Our approach allows us to break a feature-based synthesizer into four or five basic modules: the parametric synthesizer that generates audio, the feature extractor that analyzes that audio, the distance metrics that determine how similar or dissimilar pairs of feature or parameter vectors are, the optimization algorithm that searches the parameter space to find good feature values, and optionally a database keeping track previously computed mappings between synthesis parameters and feature values. The addition of this last component can greatly improve performance by providing the optimization algorithm with a good starting point. If a good mapping has already been found, the database can eliminate the need for iterative optimization altogether. Figure 2 shows the main optimization loop that finds synthesis parameters. Note that we may wish to additionally constrain the system to

find synthesis parameter vectors that are close to those used in the previous frame to improve the perceived "smoothness" of the synthesized audio.

Please refer to [3] for a more detailed discussion of feature-based synthesis and previous work on related problems.

3. FRAMEWORK DESIGN

This modular perspective makes it possible to design a feature-based synthesizer simply by plugging previously implemented components into the framework above. We have implemented a number of synthesizers, feature extractors, optimizers, distance metrics, and types of mapping databases, and a plugin architecture is included to enable the component library to be expanded dynamically without recompiling the entire system.

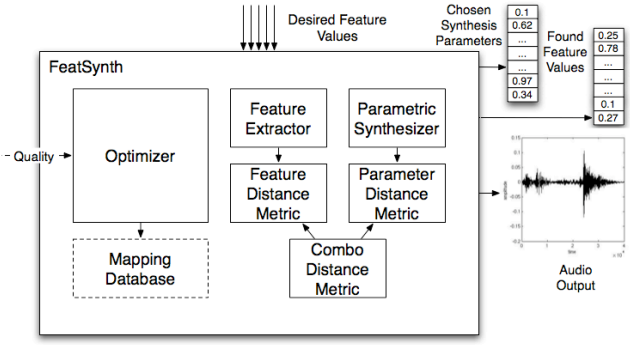

Figure 3. Contents of a FeatSynth object. Since the object is self-contained, it can simply take in a set of parameters (the target feature values) and continuously output audio like a traditional parametric synthesizer.

3.1. The FeatSynth class

Figure 3 shows the contents of a feature-based synthesizer implemented using a FeatSynth object from our framework. A feature extractor, parametric synthesizer, and optimizer all must be explicitly specified. The optimizer may optionally make use of a saved database of previously computed mappings of synthesis parameters to feature values (which it may continue to populate as it tests new parameters). Distance metrics associated with the feature extractor and synthesizer instances compute distances between the target and test parameter and feature values. (These distance metrics can be specified explicitly, but each feature extractor and synthesizer includes a default distance metric that should be appropriate for most applications.) Another metric combines two resulting distances into an overall cost function that is minimized by the optimizer. By default, this metric simply takes a weighted combination of the two distances.

Once a FeatSynth object has been built from these modular components, it can be used in the same way as any other parametric synthesizer, taking as input a vector of feature values and producing a frame of audio. The FeatSynth object automatically mediates all of the necessary communication between its components. FeatSynth objects can also be serialized, stored, and dynamically reloaded in a fairly straightforward manner, offering applications some flexibility in choosing at runtime what feature-based synthesizer to load.

In addition to the synthesized audio, the FeatSynth object can output the synthesis parameters used to generate the audio and the feature values that characterize it. This information may be useful to certain applications, such as the FeatSlider example presented in section 4. The object also has one additional input control parameter, a quality level that is passed to the optimizer and determines how long it may work before giving up and choosing a final set of parameters. Being able to make this tradeoff between speed and quality is particularly critical for interactive and real-time applications, since in many cases finding an optimal set of parameters to match the desired feature values cannot be done in real time and a "good enough" solution is better than no solution.

3.2. The ParamSynth and FeatureExtractor Classes

The interfaces for these classes are designed to be as simple as possible. Classes inheriting from FeatureExtractor are required to implement an extract() method that takes an arbitrary number of samples as input and outputs a vector of floating-point feature values. Classes inheriting from ParamSynth are required to implement a synthesize() method that takes a vector of floating-point parameters as input and outputs an arbitrary number of samples.

The simplicity of these interfaces makes it easy to write a new subclass of ParamSynth or FeatureExtractor, and enables these subclasses to be seamlessly integrated into the FeatSynth framework. Although a FeatSynth object only contains one of each of these classes, we provide several other classes that can be used to build more interesting extractors and synthesizers from simpler building blocks.

3.3. Combining Synthesizers, Extractors, and Metrics

In practice, we are often interested in combining a number of simple synthesizers to increase our available range of sounds, or we may want to combine multiple types of feature extractors to increase the amount of control we have over our system's output. The FeatSynth framework provides the ComboSynth and ComboExtractor classes to make it possible to construct more powerful ensembles of synthesizers and features without having to write a new synthesizer or extractor class. These classes respectively subclass the ParamSynth and FeatureExtractor specifications, so they can be used as components of a FeatSynth object. This also means that ComboSynths can contain other ComboSynths, allowing synthesizers to be designed hierarchically from basic building blocks.

The ComboExtractor class also provides a default distance metric that combines the N distances calculated by the N metrics associated the contained feature extractors as though they were each individual feature values. The ComboSynth class provides a similar default metric, treating the distances for the N parameter sets as individual parameters, but also considering the N weights associated with the N synthesizers.

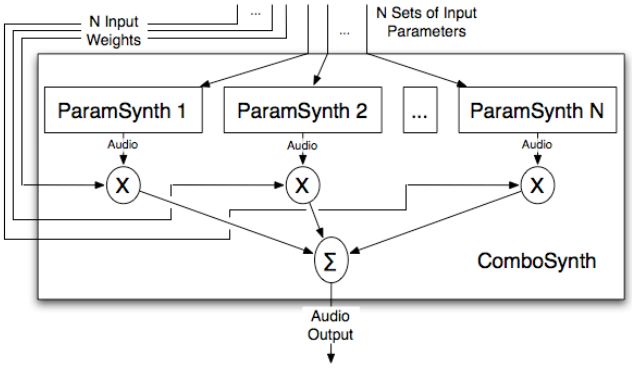

Figure 4. A ComboSynth object contains N parametric synthesizers, and takes as input a vector containing all of the parameters for each of its component synthesizers and N weights specifying how much gain to apply to each synthesizer's output.

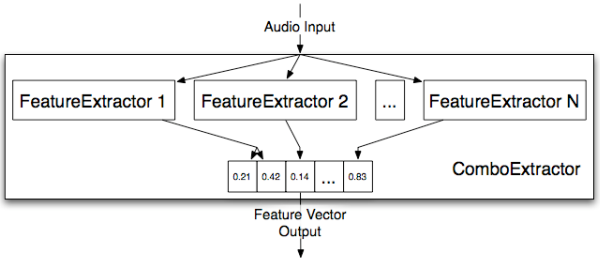

Figure 5. A ComboExtractor object contains N feature extractors, runs them all on a single input frame of audio, and outputs the concatenated feature vectors produced by each feature extractor.

3.4. Audio Processing with SynthEffects

We also provide a SynthEffect class inheriting from ParamSynth. A subclass of SynthEffect contains one or more other ParamSynths, and processes the audio produced by its ParamSynths based on an additional set of parameters tacked onto the parameter vectors that its ParamSynths would normally take. The SynthEffect class provides default distance metrics similar to the ones described above for ComboSynth.

3.5. Strengths and Limitations of Our Synthesis and Feature Extraction Models

Our synthesis and feature extraction models were designed to strike a good balance between flexibility, convenience, and reusability. The simplicity of the interfaces makes it straightforward to port existing C or C++ code into the framework, and the helper ComboSynth and SynthEffects classes make it possible to approximate the design flexibility offered by a unit generator-oriented synthesis model.

We do lose some flexibility by requiring that synthesis parameters not change over the course of a frame. Our approach relies on the assumption that a synthesizer's output depends solely on its input parameters. Allowing parameters to vary over the analysis frame would violate this assumption, so synthesizers must be designed so that a finite set of parameters encapsulates their behavior over a full frame.

Our analysis model may sacrifice some computational efficiency when redundancies exist between information extracted by multiple feature extractors. Since allowing information to be shared between different feature extraction classes would significantly compromise our goal of data encapsulation between modules, we chose to keep the architecture as it is. We are currently working on ways to make it possible to automatically eliminate these redundancies without sacrificing the simplicity of our architecture.

4. EXAMPLE APPLICATIONS

We have developed several applications using the FeatSynth framework, both because we hope they will be useful and to demonstrate various ways in which FeatSynth objects can be integrated into computer music software.

4.1. Synthesis from Feature Scores – FeatScore

This command-line application uses a saved FeatSynth object to synthesize and write to disk a series of frames of audio matching the feature values specified in an ASCII text score file. The score file consists simply of a series of rows of floating point numbers, with each column containing time-series data describing the desired evolution of an individual feature through time.

This is perhaps the simplest application one can imagine using the FeatSynth framework, since it simply serves as a wrapper for the basic functionality of the FeatSynth class. Nonetheless, it can be used in several interesting ways depending on how the score file was generated. For example, one could:

- Hand-code musical gestures into each time series, drawing the trajectory of each feature in MATLAB (or some other program) and saving the result as an ASCII text file.
- Extract feature values from an existing digital recording and resynthesize a new non-phonorealistic[1] version of the original recording matching only its extracted feature content. One could also transform the extracted feature values to achieve different results.
- Sonify existing time-series data from some other source, such as financial markets, population statistics, traffic patterns, etc.

This application offers no interactive control, but is useful for situations where offline batch processing is acceptable. An important advantage to offline synthesis is that, especially for larger feature sets and more complex synthesizers, finding a set of synthesis parameters that produces audio accurately matching the desired feature values can be quite time-consuming, and operating at sub-interactive speeds often produces substantially better results.

4.2. Interactive Feature-Based Synthesis – FeatSlider

FeatSlider is a simple graphical application developed with OpenGL that allows users to manipulate sliders

[1] Non-phonorealistic synthesis being analogous to the problem of non-photorealistic rendering in graphics.

that interactively control the feature content of sound synthesized in real time using a saved FeatSynth object. When the user moves a slider, the FeatSynth object attempts to synthesize audio matching the new set of feature values specified by the slider positions. Once the user lets go of the slider, all slider values are updated to reflect the actual feature content of the synthesized audio. The application allows the user to explore the space of different feature values and gain some insights into exactly what sonic information is being captured and controlled by each feature.

The slider values can also be manipulated in real time by MIDI control messages, permitting multiple feature values to be modified simultaneously, and allowing for more expressive control.

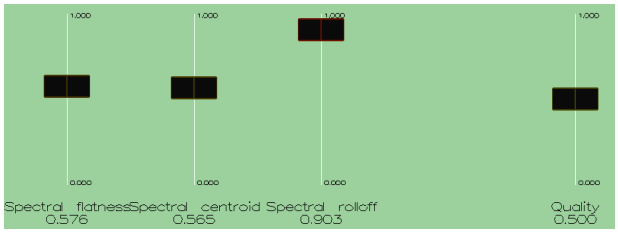

Figure 6. FeatSlider interface screen capture.

Here again the application provides a fairly simple wrapper around the basic functionality of the FeatSynth class. In this case, however, it makes use of more of the information that the FeatSynth object returns (i.e., the feature values describing the synthesized audio). It also specifies the speed/accuracy tradeoff explicitly, since it is much more critical in this real-time application that audio be delivered even if it does not yet match our desired feature values as closely as we would like. If the slider values have not changed, it can make another attempt at more accurately matching them in the next frame. This application will usually need a fairly well populated mapping database to produce reasonable results efficiently.

4.3. ChucK FeatSynth Unit Generator Plugin

Taking advantage of the new "Chugin" architecture for generating ChucK unit generator plugins, we have developed a FeatSynth UGen for the ChucK audio programming language [5]. This plugin loads a saved FeatSynth object and continuously generates audio matching feature values set by a ChucK program.

Behind the scenes, it synthesizes the audio in advance, one frame at a time, and returns the next sample from a stored buffer. When the target feature values are changed, it keeps loading samples from the stored buffer until they are exhausted, then it synthesizes the next buffer of samples based on the new feature values. The ChucK program can set the size of this buffer. Longer buffers may lead to lower computational load (since the parameters need only be optimized once per buffer), but may also hurt responsiveness (since the last buffer must be sent out before new audio is synthesized).

This plugin was written in relatively few lines of code, but allows any FeatSynth object to be easily imported into ChucK, permitting the use of feature-based synthesis in the context of a MUSIC-N-style audio programming language with built-in features such as strong timing, concurrency, MIDI/Open Sound Control support, etc. This is a pleasing result, given our stated intention of making feature-based synthesis techniques easy to integrate into C++ projects.

5. FUTURE WORK

One component still lacking from the FeatSynth framework is a GUI-based development environment for building FeatSynth objects. Such a tool would make it possible to build new FeatSynth objects from a library of synthesizers, feature extractors, and optimizers without needing to write a single line of code.

There is also more theoretical work to be done to improve the underlying feature-based synthesis techniques, most notably having to do with automatically finding explicit models of the relationships between synthesis parameters and features to reduce our dependence on expensive iterative optimization. Another important question to explore further is how best to extend these techniques to capture information about how sounds evolve through time.

6. CONCLUSION

We have developed a C++ framework for developing feature-based synthesizers and integrating them into applications. Our objective has been to make the somewhat arcane art of feature-based synthesis more accessible and easier to use. The framework is based on a modular architecture that only asks that developers specify what components they want to use, and does not require a low-level understanding of the various implementation tricks necessary to produce a working feature-based synthesizer. We hope that the community will find this framework and the tools developed using it useful.

7. REFERENCES

[1] Amatriain, X., de Boer, M., Robledo, E., Garcia, D., "CLAM: an OO framework for developing audio and music applications," Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Seattle, Washington, USA, 2002.

[2] Bray, S. and Tzanetakis, G., "Distributed audio feature extraction for music," *Proceedings of the International Conference on Music Information Retrieval*, London, UK, 2005.

[3] Hoffman, M. and Cook, P.R., "Feature-based synthesis: mapping acoustic and perceptual features onto synthesis parameters," *Proceedings of the International Computer Music Conference*, New Orleans, USA, 2006.

[4] Tzanetakis, G. and Cook, P.R., "MARSYAS: A framework for audio analysis." Organized Sound (1999), 4(3): pp. 169-175.

[5] Wang, G. and Cook, P.R., "ChucK: a concurrent, on-the-fly, audio programming language," *Proceedings of the International Computer Music Conference*, Singapore, 2003.