Programming-by-Example for Audio: Synthesizing Digital Signal Processing Programs*

Mark Santolucito
Computer Science
Yale University
New Haven, CT, USA
mark.santolucito@yale.edu

Kate Rogers
Computer Science
Yale University
New Haven, CT, USA
kate.rogers@yale.edu

Aedan Lombardo
Computer Science
Yale University
New Haven, CT, USA
aedan.lombardo@yale.edu

Ruzica Piskac
Computer Science
Yale University
New Haven, CT, USA
ruzica.piskac@yale.edu

Abstract

Programming by example allows users to create programs without coding, by simply specifying input and output pairs. We introduce the problem of digital signal processing programming by example (DSP-PBE), where users specify input and output wave files, and a tool automatically synthesizes a program that transforms the input to the output. This program can then be applied to new wave files, giving users a new way to interact with music and program code. We formally define the problem of DSP-PBE, and provide a first implementation of a solution that can handle synthesis over commutative filters.

CCS Concepts • **Applied computing → Sound and music computing**;

ACM Reference Format:
Mark Santolucito, Kate Rogers, Aedan Lombardo, and Ruzica Piskac. 2018. Programming-by-Example for Audio: Synthesizing Digital Signal Processing Programs. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242903.3242906

*This research sponsored by NSF grants CCF-1302327 and CCF-1715387.

Many thanks to Thomas Murphy for his guidance in thinking through this problem and many solution attempts. Thanks to the anonymous reviewers whose comments helped raise the bar on this paper.

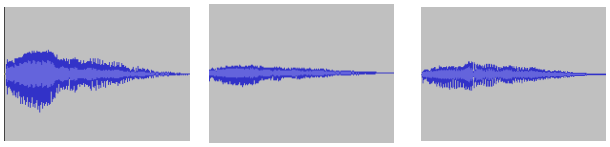
FARM '18, September 29, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5856-9/18/09…$15.00
https://doi.org/10.1145/3242903.3242906

1 Introduction

The great proliferation of computer music programming languages points to the difficulty of building a natural interface for users that want to computationally interact with musical data. Programming applications in the domain of computer music, and specifically digital signal processing (DSP), requires that users not only grasp fundamental programming techniques, but also have a large domain specific knowledge of time and signal manipulations. The amount of prerequisite skill and effort to overcome these barriers is often higher than many users are able to commit.

Furthermore, the difficulty of programming DSP applications is often not commensurate with the scope of the creative intentions. A simple creative choice may require a disproportionate technical effort. As a motivating example, imagine a user hears a sample in a piece of music, and again hears the same sample later in the piece with some added effects. In order to reuse this effect in the user's own musical composition, the user must now reconstruct the filter that was used to transform an audio clip. In this case the user has the original audio file, and the transformed audio file, but does not know exactly how this transformation happened. In the standard approach, a user would need to be a domain expert and listen to the two files, and aurally estimate which kinds of filters were used to achieve the transformation. Once the user has some suspicion as to the appropriate filter types that will be needed, the user must write a program in some language (SuperCollider [McCartney 2002], CSound [Boulanger et al. 2000], PureData [Puckette et al. 1997], etc) to implement the DSP filter the user has in mind. Further still, the user will then need to spend time tweaking the filter parameters to find the best fit.



(a) Input example **(b)** Output example **(c)** Generated

Figure 1. The waveforms (a) and (b) are provided as examples, and DSP-PBE synthesizes a filter that produces (c).

To simplify this process, we introduce *DSP programming by example* (DSP-PBE). With DSP-PBE, the user simply provides our tool with the original audio (input), and the transformed audio (output), and the tool will automatically construct a DSP filter that approximates the transformation.

We formally define the problem of DSP programming by example as follows: Given an input waveform I and an output waveform O, construct a DSP filter $\mathcal{F}$, to minimize the aural distance $dist$ between O and $\mathcal{F}(I)$. In a single line,

$$\text{Find } \mathcal{F}, \text{ such that } dist(O, \mathcal{F}(I)) = 0$$

In the sequel we describe our approach to the two key components of this statement; the definition of distance, and a search technique to find $\mathcal{F}$. A distance metric that is faithful to the psycho-acoustics of the human ear is critical for a useful DSP-PBE tool. As an example, taking a trivial distance function that returns the difference in length of the two audio samples will allow a delay filter to satisfy any example pair of samples. Additionally, an efficient search algorithm is critical, as the space of possible DSP filters is very large. Not only do we need to consider a wide variety of filters, we need to consider the space of parameters for each filter, as well as the different ways of combining multiple filters.

2 Motivating Example

As a motivating example, imagine a user was to reconstruct the filter that was used to transform an audio clip, as shown in Figure 1. In this example, a user provided a clip of a `cartoon-spring.wav` in Figure 1a, and the same sound as it had been transformed with a low-pass filter at 800 Hz, $lpf(800)$, as shown in Figure 1b. However the nature of the transformation is unknown to the user and they wish to discover the filter needed. Our DSP-PBE tool is able to synthesize a filter $lpf(1989)$, that when applied to the original sound, produces the waveform shown in Figure 1c. While the solution is not exact, the difference is not significantly noticeable to an untrained ear.

3 Background

To give context for DSP-PBE, we first explain the traditional concept of programming by example [Cypher and Halbert 1993; Gulwani 2012; Lieberman 2001]. Programming by example (PBE) is a synthesis technique that automatically generates programs that coincide with given examples. An example is specified as a tuple of input and output values. Given a set $S = \{(i_1, o_1), \ldots, (i_n, o_n)\}$ of input/output examples, the goal is to automatically derive a program P such that for every j, $P(i_j) = o_j$.

PBE is in line with one of the often repeated high level goals of functional programming – to describe *what* a program should do, and not *how* the program should do it. Instead of writing code, the user provides a list of relevant examples and the synthesis tool automatically generates a program. In this way, the examples can be seen as an easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it still might not correspond to the user's intentions. Examples are, by nature, an incomplete specification.

PBE is a promising research direction that enables easy manipulation of data even for non-programmers [Gulwani et al. 2012]. Recent work in this area has focused on manipulating fundamental data types such as strings [Menon et al. 2013; Singh and Gulwani 2012] and lists [Feser et al. 2015; Osera and Zdancewic 2015]. The success and impact of this line of work can be estimated from the fact that PBE ships as part of the popular Flash Fill feature in Excel 2013 [FlashFill 2013].

The core difference between traditional PBE and DSP-PBE is in the application domain of Digital Signal Processing. Digital Signal Processing (DSP) programming languages provide users with an interface to build signal processing programs in domain specific languages. Some of these languages provide their own implementations of signal processing primatives, such as SuperCollider [McCartney 2002], CSound [Boulanger et al. 2000], and PureData [Puckette et al. 1997]. Other DSP languages provide alternative front-ends to these languages, such as Vivid [Murphy 2018], which provides Haskell bindings to Supercollider.

Although many DSP languages are full featured enough to write general purpose programs, in this work we focus on the construction of DSP filters. A DSP filter is, broadly speaking, any program that transforms a digital signal from one form to another. An example of a DSP filter is a low-pass filter, which takes an input signal and generates an output signal that keeps frequencies below some frequency threshold, but removes frequencies above that threshold.

The most closely related work in audio signal processing is a technique called resynthesis [Masri and Bateman 1996]. Resynthesis is the process of decomposing a sound into its spectrogram, and then building a synthesizer to recreate a similar sound. The limitation here is that resynthesis builds a generative synthesizer, which does not take into account any information about the components used to create the original sound. This limitation means that resynthesis cannot be applied in a new context, whereas DSP-PBE allows us to

construct a DSP program that can be used with various new input samples to create novel sounds. For example, DSP-PBE could be given a sample of a trumpet and a trombone, and the generated DSP program could be applied to a violin to hear what a violin sounds like if it was a trumpet that had been turned into a trombone. In this case we can discover the analogy *trumpet:trombone :: violin:?*.

From a machine learning perspective, the above example use case is closely related to work on learning analogies [Mikolov et al. 2013], where the goal is to discover relations such as *man:king :: woman:queen*. To do this, words are embedded in a vector space, so that the transformation from *man* to *king*, can be directly applied to *woman*. There are two keys differences between this approach and DSP-PBE. The first is that DSP-PBE should produce a human readable transformation. We would like to generate DSP programs that can be used verbatim, but also inspected and modified by the user. While program code provides this readability, vector transformations are not comprehensible in the same way. Second, word embeddings require that the semantics of an object can be embedded into a vector space. As we will see in Sec 4, a semantic representation of an audio file (what a human perceives) is not immediately recoverable from its direct representation.

4 Aural Distance

As a distance metric, we used as a starting point the literature on acoustic fingerprinting [Casey et al. 2008]. Acoustic fingerprinting is the concept of creating a condensed, distinct summary of an audio file that can be used later to identify that audio file or to look it up in a database. Acoustic fingerprints turn an audio file into a represent of how the file will sound to the human ear regardless of how it is represented in a digital format [Casey et al. 2008]. There are numerous ways to develop acoustic fingerprints and companies like Shazam and Sound-Hound have developed complex algorithms to create accurate fingerprints even from low quality files recorded on a cellphone mic. For this work, we used the work of Shazam [Wang 2003] as an inspiration for our distance metric calculation.

As an intuition, the psycho-acoustic identity of a sound file (how humans distinguish between one sound and the next) can be captured by taking every "moment" of audio, and listing the predominate frequencies for that slice of time. This intuition can be represented with a waterfall plot, as shown in Figure 2, which plots how the frequencies change over time. A waterfall plot uses the Fast Fourier Transform (FFT) to calculate many discrete Fourier transforms over small times slices. In this way, a waterfall plot is a representation of an audio file as a list of spectrograms plotted over time. In the Shazam method, the peaks are selected from each time slice of audio and used to create a "constellation" of peaks over time. This constellation is then used to build a hash that acts as a fingerprint to uniquely identify the audio sample. We use a similar strategy by first performing a real Fast Fourier Transform on the audio file and then picking out the frequency peaks in each time frame. However, the key difference in DSP-PBE is that we do not use the constellation as a hash for lookup in a database (as Shazam and Sound-Hound do), but instead, we need a distance metric between two constellations to provide a measure of how close we are to synthesizing the correct DSP filter. Distance metrics are common in music synthesis tasks, for example, in the generation of jazz improvisations, where the improvisation should stay close by some measure to the original melody [Donzé et al. 2014].

Fast Fourier Transforms (FFT) are the key to a good acoustic fingerprint. The FFT, however, cannot be taken as a black-box in our application. The two factors we need to consider are 1) the window-size for how many samples will be used to calculate the FFT, and 2) the bin size which roughly speaking, defines the resolution of the FFT.

Each return element is a frequency *bin*, and depending on the scale of your return array the size of these bins varies. In order for each bin to correspond to 1 Hz the size of the return vector must be equal to the sampling frequency (44,100 Hz). If each bin is not 1 Hz, the effects of spectral leakage will be seen. This occurs when the bins do not correspond to the exact frequency peaks of the sound. The amplitude from the peaks that fall in between bins will *leak* over into the closest bin and create a distorted spectrogram. For this reason we had to adjust the size of the FFT return arrays to be 44,100, as 44,100 Hz is a common format for audio. Although this slows down the process of FFT, it provides the most accurate representation of the sound and for our purposes frequency accuracy is paramount.

With our constellations created from the waterfall plot, we constructed a `dist` function that measures the aural distance and is faithful to the psycho-acoustics of the human ear. Our implementation takes the Euclidean distance of the peaks in a time slice on the frequency-amplitude axis. In order to define this distance more formally, we introduce the notation $c@t$ to indicate selecting time slice t from constellation c. We also use the function $peak :: Int \rightarrow Constellation \rightarrow Peak$ to select a peak from a constellation, where the peaks are in sorted order based on frequency. Then, for an audio clip x and an audio clip y, and a function $toC :: Audio \rightarrow Constellation$ to transform the audio clip into a constellation with ts time slices and p peaks in each time slice:

$$\sum_{t=0}^{ts} \sum_{i=0}^{p} euclid\Big(peak(i, toC(x)@t), \\ peak(i, toC(y)@t) \Big)$$

Note that this definition requires the audio clips to be temporally aligned, which is not always a fair assumption in

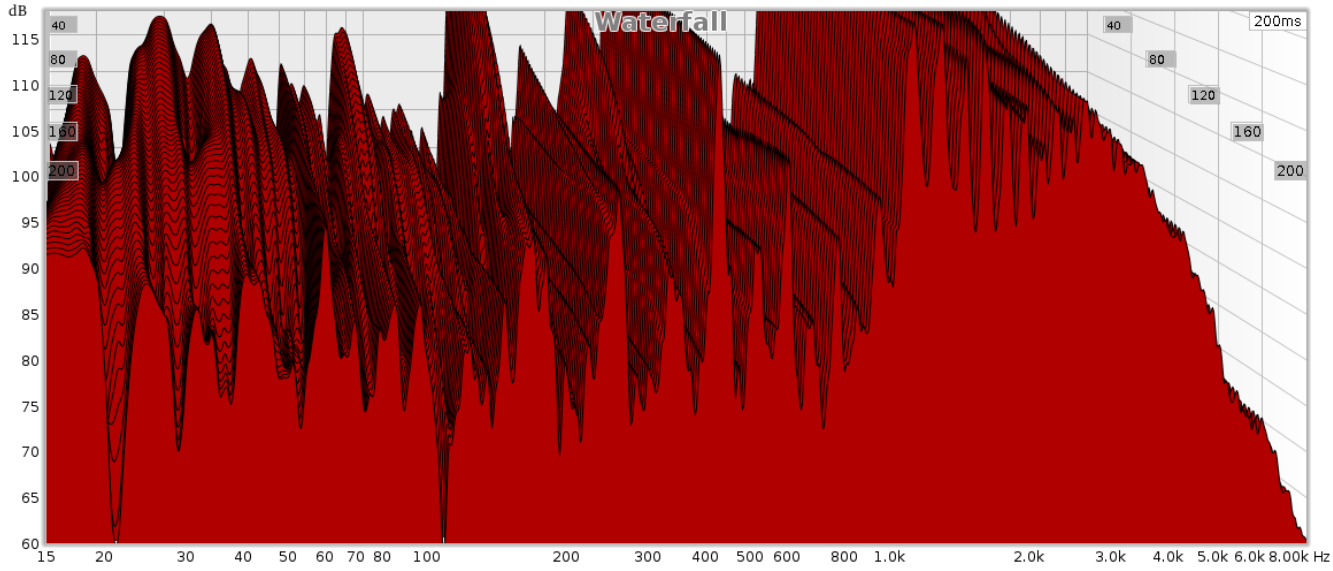


Figure 2. A waterfall plot of the `cartoon-spring.wav` from 0-200 ms between frequencies of 20-8000 Hz, where the height indicates the amplitude of each frequency. This plot was created with the REW tool [REW 2018].

the real world. We leave the exploration of a temporal offset between two example audio samples to future work.

As a sanity check that this distance metric matches the psycho-acoustic definition of distance, we used the test cases listed in Table 1.

The goal in the synthesis procedure is to find a DSP filter program, F, such that $\texttt{dist}(O, F(I)) = 0$. However, in practice the DSP-PBE Synthesizer can only get us so close to this metric and we instead just minimize this distance. To do this, the user specifies a default threshold distance for the aural distance. The threshold distance defines how close is acceptably close, and can be changed by the user depending on their needs or requirements.

5 Search

As the search space of possible DSP program is extremely large, our search procedures must be exceptionally efficient. As a first foray into DSP-PBE, we restrict ourselves to only synthesizing low-pass and high-pass filters, and global volume adjustment. These two filters have the key property that they are quasi-commutative – when the thresholds of these filters do not overlap, applying a low-pass and then a high-pass is the same as applying a high-pass and then a low-pass. Although our approach has no theoretical basis for being applicable to non-commutative filters (for example, delay lines or ring filters), we do attempt to use our approach on such filters in Sec 8. We leave a more thorough exploration of non-commutative filters to future work.

5.1 Gradient Descent

Gradient descent is a technique commonly used in modelling and machine learning. Given a cost function, which represents the disagreement between a proposed model and the actual data, gradient descent can be used efficiently to minimize the cost and generate the model of best fit. Gradient descent is only guaranteed to terminate with the globally minimal cost if the cost function being optimized is convex – this is because gradient descent will "descend" along the surface of the cost function, in each step following the steepest gradient. While we were not able to design our aural distance function from Section 4 to be convex, our cost function does demonstrate some properties of convexity that allow gradient descent to produce useful results, even if the result is not guaranteed to be the global minimum. We will describe here some properties of our distance metric that were helpful in minimizing the cost of the synthesized filter, as well as the shortcomings of our design, and how we try to overcome them by adjusting our implementation of gradient descent.

In order to visualize the rough shape of our distance metric, we plot the distance between pairs of examples, and various possible DSP filters in Figure 3. Here we only visualize the distance curves in the dimension of the low-pass filter. Notice that the curves exhibit a clear "saddle", which represents the minimum cost. In the ideal case, gradient descent will find these points. Note that we do not have these graphs available during synthesis – producing the entire graph as in Figure 3 is prohibitively expensive.

In Figure 3, the last curve we plot is the distance between `cartoon-spring.wav` and `cartoon-spring-hpf1500.wav`,

Table 1. Test cases to evaluate distance metric. The exact values are only important in relationship to the others.

Test Name & Expected Result	Value 1	Value 2
Identity Value 1 = 0	(PianoC, PianoC) = 0	NA
Commutativity Value 1 = Value 2	(PianoC, PianoCSharp) = 5.635	(PianoCSharp, PianoC) = 5.635
Commutativity Value 1 = Value 2	(PianoC, HornCSharp) = 20.500	(HornCSharp, PianoC) = 20.500
Filter less than pitch Value 1 < Value 2	(PianoC, PianoFilterC) = 3.749	(PianoC, PianoCSharp) = 5.635
Filter less than pitch+instrument Value 1 < Value 2	(PianoC, PianoFilterC) = 3.749	(PianoC, HornCSharp) = 20.500
Pitch less than pitch+instrument Value 1 < Value 2	(PianoC, PianoCSharp) = 5.635	(PianoC, HornCSharp) = 20.500

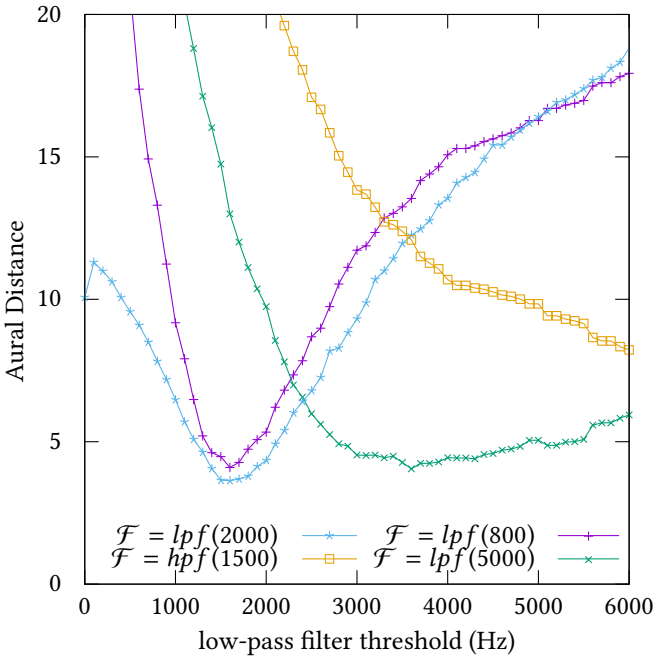


Figure 3. The distance curves showing the convex-like shape of the aural distance function. Each curve is the distance between an input file, and a filter applied to that file - $dist(I, \mathcal{F}(I))$.

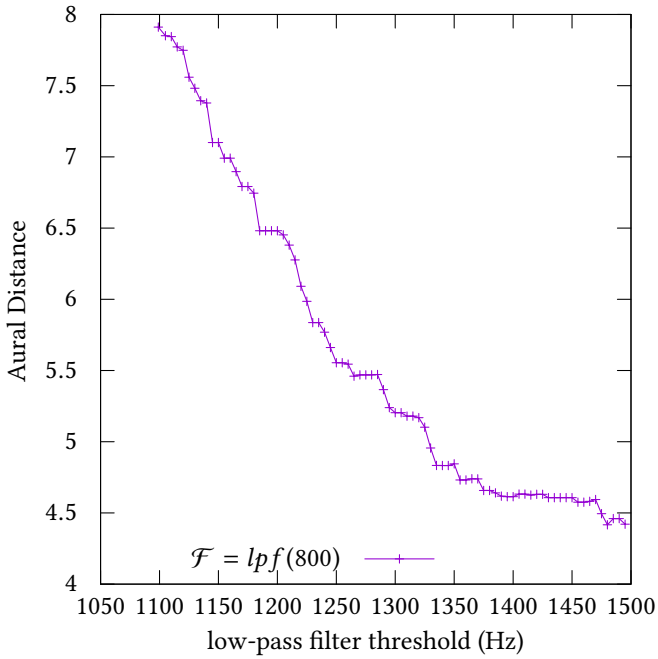


Figure 4. Zooming in (1000 to 1500 Hz) on a portion of a curve from Figure 3, we see the aural distance function is not perfectly convex on the micro scale.

the same file with a high pass filter applied with a threshold of 1500 Hz. Notice that as the threshold of the low-pass filter applied to the input example (`cartoon-spring.wav` increases, the distance to the output example decreases. This is because as a low-pass filter's threshold increases, it allows more and more frequencies to pass into the output – thereby having less of an effect. Whereas in the case of the `cartoon-spring-hpf1500.wav`, the true filter is a high-pass filter, so the less we apply a low-pass filter, the closer we get to the correct filter.

Although Fig. 3 depicts on one dimension of the search space (low-pass filter threshold), the actual space we need to search has many more dimensions. In our implementation, we only explore a space of two DSP filters and volume adjustment, but this already results in 5 dimensional space (each filter requires both a threshold value and an amplitude value for how much of the filter to apply). In general, this space

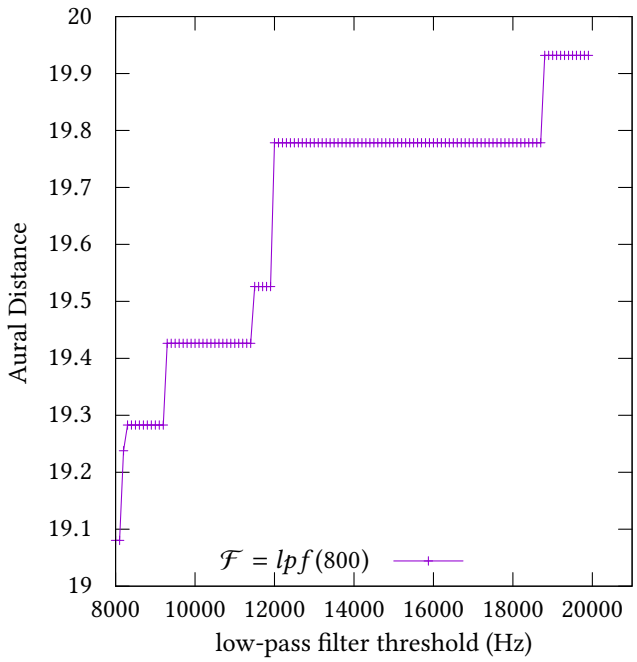


Figure 5. Looking at the portion of a curve from Figure 3 between 8k Hz and 20k Hz, we see the aural distance function is not perfectly convex on the macro scale. In this case, that is because the sample has very few frequencies above the 8k Hz range.

becomes even larger for DSP-PBE as more DSP primitives (ring filter, white noise, delay etc) are added. To speed up gradient descent, we use stochastic gradient descent, so that in each step, we only move in $d < 5$ number of dimensions.

5.2 Dealing with Non-convexity

There are a number challenges with working with gradient descent in the aural DSP domain because our distance metric is not convex. On the micro scale, the distance function is susceptible to noise and not entirely smooth, as shown in Figure 4. In order to handle the micro scale variations, we use a periodic restart of the gradient descent. This means that every n rounds, as defined by the user, the gradient descent will backtrack to the best solution it has found so far. Intuitively, the choice of n represents how far gradient descent is allowed to explore a path of optimization before it is forced to give-up on that direction if it has not found any benefit to this direction. The best value for n then must be determined based on the trade-off of potential time wasted on poor choices, and the potential benefit of these choices. In our implementation we use $n = 4$ after a holistic evaluation of the convexity of the aural distance function. The stochastic gradient descent will then continue, selecting dimensions to explore in each round using a new random seed.

On the macro scale, we face the challenge that the distance function is again not convex – there are many local minima and long plateaus, as shown in Figure 5. In order to overcome this, we must carefully pick the initial value for gradient descent. If we pick a value in the middle of a plateau, the gradient descent algorithm will not find any significant gradient, and conclude we have reached the convergence condition. In our current implementation, we iterate at large intervals (1000 Hz) of possible threshold values for both low and high pass filters. We choose possible DSP programs that use only low pass, only high pass, and both low and high pass filters. After evaluating these, we take the lowest cost initial DSP program, and start gradient descent from that point.

Finally, one of the key parts of a good application of gradient descent is the choice of the parameters such as the learning rate and the convergence goal. These parameters must be adjusted based on the values observed from the cost (in our case, distance) function. While the details of tuning gradient descent are outside the scope of this paper, it suffices to note that any change in the distance metric will likely also require a readjustment of these parameters.

6 Evaluation

We implemented a DSP-PBE tool based on the approach described in Section 4 and Section 5. Our tool is available open-source at www.github.com/santolucito/DSP-PBE[1]. Our tool is mostly written in Haskell and uses the Vivid library [Murphy 2018] for bindings to SuperCollider [McCartney 2002]. Haskell allows easy access to type information and metaprogram construction tools that are useful for program synthesis, however the programs themselves are easily translated back to SuperCollider "synth defs", which are DSP filter programs. We use the scipy python module for calling the FFT since the library is quite mature and provides a simplified interface specifically for calling FFT on audio.

One key implementation point is that we use a separate representation of a DSP for running gradient descent, and for actually processing the audio. Gradient descent works best when all parameters are in the same scale, so we map the frequencies [0,20k] Hz to a [-1,1] scale. Likewise, we map the application levels for each filter (how much of the filtered output should be included in the final mix) on a [-1,1] scale.

In Table 2, we show the results of running our tool on a set of benchmarks of input/output example audio samples. The audio samples were transformed in Audacity, using the Low Pass Filter and High Pass Filter effects. Since we use SuperCollider's filter implementations on the backend, there may be very slight variation, but this is to be expected in real-world application as well. All experiments were run on

[1]The exact version of the code used for this evaluation is available at commit https://github.com/santolucito/DSP-PBE/tree/d022954164b830395bddb21cdc94046ed6882083.

Table 2. Time to converged on a solution DSP program for various benchmarks. The programs may not match the known DSP program, but may still be psycho-acoustically equivalent depending on the expertise of the listener.

Description	True DSP	Synth'ed DSP	Time (sec)
Cartoon Spring	$lpf(800)$	$lpf(1989)$	56.195
Cartoon Spring	$lpf(5000)$	$lpf(4000) \gg hpf(7000)$	54.004
Cartoon Spring	$hpf(1500)$	$lpf(1000) \gg hpf(1000)$	53.964
BTS DNA (Kpop)	$lpf(2000)$	$lpf(1996)$	56.874
Holst Mars	$hpf(3500)$	$lpf(10000) \gg hpf(1000)$	55.444

an Intel Core i7-6820HQ CPU @ 2.70GHz with 16 GB of RAM and an Intel Sunrise Point-H HD Audio sound card.

We can also breakdown the runtime cost of synthesis into the two different stages - 1) initial program selection, and 2) gradient descent. The initial program selection phase is a mostly fixed cost, as we always evaluate the same distribution of initial value. On average this process takes roughly 40 seconds. We outline future directions of research that may be able to reduce this cost in Section 7.

7 Refinement Type Driven Synthesis

In order to find an initial value for gradient descent, we could use refinement types [Freeman and Pfenning 1991]. In this section we explore a possible optimization for selecting an initial DSP program for gradient descent. This has not yet been implemented, but we present the theory behind the approach.

7.1 Refinement Types for DSP

Refinement types are a way of giving an abstract description of the behavior of a function. For example, using a similar syntax to the refinement type system for Haskell, Liquid-Haskell [Vazou et al. 2014], given the function `map :: [a] → [b]` we can further provide a refinement types that captures some properties of the behavior of this function over values:

```
f :: xs:[a] → ys:[b] | length xs == length ys
```

In this case, the refinement type describes that the length of the lists are still equal after applying the `map` function.

In a similar style for DSP, we can write predicates about the filters available to us during synthesis. For example, a low-pass filter could be described as the refinement type that says the amplitude of the frequencies greater than the threshold frequency have decreased in the output Audio. For brevity in notation, we will only treat a single time slice from the waterfall plot here, but the concept generalizes when quantified over all time slices as well.

$$\texttt{lpf :: t:Float} \rightarrow \texttt{xs:Audio} \rightarrow \texttt{ys:Audio} \mid$$
$$\forall f_1 \in \texttt{spectrogram(xs)}.\ \forall f_2 \in \texttt{spectrogram(ys)}.$$
$$(f_1 > \texttt{t} \wedge f_2 > \texttt{t} \wedge f_1 == f_2) \implies \texttt{amp}(f_1) > \texttt{amp}(f_2)$$

Where `t` represents the level at which the lowpass filter is applied, `spectrogram` represents the spectrogram of the sound sample, f_i represents a frequency, and `amp()` represents the amplitude of the frequency.

Additionally, a high-pass filter could be described as the refinement type that says the amplitude of the frequencies less than the threshold frequency have decreased in the output Audio.

$$\texttt{hpf :: t:Float} \rightarrow \texttt{xs:Audio} \rightarrow \texttt{ys:Audio} \mid$$
$$(\forall f_1 \in \texttt{spectrogram(xs)}.\forall f_2 \in \texttt{spectrogram(ys)}).$$
$$(f_1 < \texttt{t} \wedge f_2 < \texttt{t} \wedge f_1 == f_2) \implies \texttt{amp}(f_1) > \texttt{amp}(f_2)$$

Notice that in these refinement types, we only need to calculate the spectrogram for the input and output statically. As opposed to the current technique of generating filters, applying them, and the calculating the aural distance, this approach is relatively static. We could quickly check many threshold values over the input and output examples. This will only yield a rough boolean estimation of whether this threshold should even be considered, but this is enough information for us to select an initial program to pass to our gradient descent algorithm. As the search for an initial filter takes roughly 40 seconds out of our current benchmarks, this could dramatical increase the speed of synthesis.

7.2 Combination of Search Algorithms

Beyond just using the refinement types to select an initial program for gradient descent, we can use refinement types in as part of the main search strategy as well. We briefly describe here a way to use refinement types in combination with gradient descent to handle more complex combinations of DSP filters. So far in our work (*c.f.* Sec. 6) we have synthesized filters with a fixed form - all our solutions use a single low-pass filter, and a single high-pass filter. Ideally, we would be able to synthesize solutions that use any arbitrary combination of filters. In order to do this, we would need an iterative solution that can find one filter at a time.

In this approach, given input example `x:Audio` and output example `y:Audio`, we would first find a filter $\mathcal{F}$' using the approach described in Sec. 5 and Sec. 7.1. We will say that this $\mathcal{F}$' has the refinement type r_1. However, this filter might not return a satisfactory result. We could then continue the search using the output of $\mathcal{F}$'`(x)` as the new input example,

z:Audio. Now the synthesis task is to find a filter $\mathcal{F}$" (with refinement type r_2) using input example z:Audio and output example y:Audio. Essentially, $\mathcal{F}$' has gotten us the first half of the way, and $\mathcal{F}$" will get us the second half of the way. With this, we can start to use more information rich refinement types, such as below:

$$\mathcal{F} :: \texttt{x:Audio} \rightarrow \texttt{y:Audio} \mid \exists\, \texttt{z:Audio}.\ r_1(\texttt{x},\texttt{z}) \wedge r_2(\texttt{z},\texttt{y})$$

8 Future Work and Conclusions

The main contribution of this paper is to pose the problem of DSP-PBE. While we have presented a prototype implementation of a DSP-PBE tool, this primarily functions as a proof-of-concept. There remains significant room for optimization in both the distance calculation and the search algorithm. In future work, we also plan to expand beyond commutative filters to be able to synthesize effects such as delay lines.

We briefly revisit the motivating conceptual example from Sec. 3 where we want to synthesize the filter that transforms a trumpet into a trombone and apply that filter to a violin. In order to achieve this we need more complex filters than high-pass, low-pass, and amplitude adjustment. Despite a lack of a formal approach for non-commutative filters, we added a pitch shift filter and the `ringz` delay line filter from SuperCollider into our tool using the current approach. Though we had no reason to believe that our current approach should produce useful results when allowing more complex filters in the search space of synthesis, we found the results to be at least interesting, and even reasonable. The synthesized filter to transform a trumpet into a trombone was as follows:

```
SetVolume: 1.00%
LoPass: freq@1000 amp@0.91 >>>
HiPass: freq@100 amp@0.00 >>>
PitchShift: freq@-1600 amp@0.91 >>>
Ringz: freq@9100 delay@0.10 amp@0.23 >>>
WhiteNoise: amp@0.00
```

The trumpet and trombone input audio files, as well as the audio of a violin with this filter applied is available as a Soundcloud playlist [Santolucito 2018].

Although with the current tool, synthesis times presented might be prohibitively slow for many use cases, especially on such small programs, we should be encouraged by progress in other domains of program synthesis. In the SyGuS program synthesis competition, which has run for four years, tools have seen an exponential speed up and increase in the range of programs that can be synthesized. As one example, in the 2014 competition the `LinExpr_eq1.sl` benchmark was only solved by one tool, and took 1128 seconds [Alur et al. 2014]. In the 2017 competition, the same benchmark was solved by all tools, with the fastest taking only 199 seconds [Alur et al. 2017].

References

2018. REQ: Room Eq Wizard. http://www.roomeqwizard.com. (2018). Accessed: 2018-07-08, Version 5.18.

R. Alur, R Bodik, E. Dallal, D Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madusudan, M. Martin, M. Raghothman, S. Saha, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and E. Udupa. 2014. Syntax-Guided Synthesis. *Dependable Software Systems Engineering, NATO Science for Peace and Security Series* (2014). http://sygus.seas.upenn.edu/files/sygus_extended.pdf

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. *CoRR* abs/1711.11438 (2017). arXiv:1711.11438

Richard Charles Boulanger and others. 2000. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming.*

Michael A Casey, Remco Veltkamp, Masataka Goto, Marc Leman, Christophe Rhodes, and Malcolm Slaney. 2008. Content-based music information retrieval: Current directions and future challenges. *Proc. IEEE* 96, 4 (2008).

A. Cypher and D.C. Halbert. 1993. *Watch what I Do: Programming by Demonstration.* MIT Press.

Alexandre Donzé, Rafael Valle, Ilge Akkaya, Sophie Libkind, Sanjit A Seshia, and David Wessel. 2014. Machine improvisation with formal specifications.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 229–239.

FlashFill 2013. Flash Fill (Microsoft Excel 2013 feature). (2013). http://research.microsoft.com/users/sumitg/flashfill.html.

Tim Freeman and Frank Pfenning. 1991. *Refinement types for ML.* Vol. 26. ACM.

Sumit Gulwani. 2012. Synthesis from Examples: Interaction Models and Algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012). Invited talk paper.

Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann.

Paul Masri and Andrew Bateman. 1996. Improved modelling of attack transients in music analysis-resynthesis. In *ICMC.*

James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *ICML (1).* 187–195.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems.*

Tom E. Murphy. 2018. Vivid Synth. vivid-synth.org. (2018).

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 619–630.

Miller S Puckette and others. 1997. Pure Data. In *ICMC.*

Mark Santolucito. 2018. Trumpet is to Trombone as Violin is to ? https://soundcloud.com/mark-santolucito/sets/trumpet-is-to-trombone-as-violin-is-to. (2018).

Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *PVLDB* 5 (2012).

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

Avery Wang. 2003. An Industrial Strength Audio Search Algorithm. *(ISMIR).*