CorpusDB: Software for Analysis, Storage, and Manipulation of Sound Corpora

Thomas M. Stoll
Bregman Music and Research Studio
Dartmouth College
Hanover, NH 03755
Thomas.M.Stoll@dartmouth.edu

Abstract

CorpusDB is a system for representing sound files and associated analysis metadata in a structured format. The formats and conventions used in conjunction with the database allow for representation of sound files and their processed variants; multiple, overlapping, hierarchical relationships between sound files and segments thereof; and connections between sounds, their transformations, and analysis metadata. The software described in this paper is a parallel implementation consisting of SuperCollider classes, Python classes, and a common data representation of corpora that allows for seamless sharing of data between the two complementary environments. Code examples and listings of multi-step algorithms are included that demonstrate the kinds of operations possible within this system.

Introduction

CorpusDB is a software tool for concatenative and data-driven sample-based sound synthesis inspired by techniques from Music Information Retrieval (MIR). CorpusDB actually consists of two implementations of the same software: one in SuperCollider/*sclang* (McCartney 2002) and one in Python using SuperCollider's *scsynth* as its sound engine, plus a common Javascript Object Notation (JSON) data format that enables the sharing of metadata across the two implementations. There are several reasons to create and maintain a software project that includes separate implementations of the same functionality. First, each platform has its particular strengths: SuperCollider is better for live performance applications and sound design prototyping, while Python is suited for batch mode and other offline processes. SuperCollider provides *sclang*, a rich language with many built-in musical idioms. Python allows a programmer to link a program based on CorpusDB to other libraries, including Numpy (Oliphant 2006), Matplotlib (Hunter 2007), machine learning libraries, and others.

Following a very brief overview of concatenative synthesis, we present the system, design decisions, and features that relate to representing music and sonic relationships in a database. Various access functions are presented as vital tools capable of translating stored sounds into meaningful abstractions of sonic or musical material, plus several search tools that allow for similarity-based search with certain constraints. Finally, we demonstrate the power of numerical analysis applied to compositional materials and incorporated into the compositional process.

Concatenative Synthesis

Corpus-based concatenative synthesis (CBCS) is a synthesis process where units–segments of sampled sound files–are arranged in sequences much like tiles or shingles. The corpus refers to a collection of sounds and accompanying metadata that allows for the manipulation of these units as abstractions of sound and data. Plunderphonics (Oswald 1999) and granular synthesis are immediate ancestors of CBCS. During resynthesis, as the lengths of units are decreased and the number of overlapping units is increased, the resulting sound approaches that of a granulation effect.

Concatenative synthesis has existed as a tool for composers for more than a decade. Diemo Schwarz has produced two overviews (Schwarz 2006; 2007), and other systems such as SoundSpotter (Casey 2009) have been described. Furthermore, MIR-inspired tools for large-scale automated compositional tasks have been described (Collins 2012; Eigenfeldt 2011). Each system imposes certain working methods, limited in some respects by the specific features they implement. For instance, SoundSpotter does not impose any structure on the sound corpus; it simply allows for similarity search within a large collection of pre-analyzed tiles.

Overview of CorpusDB

CorpusDB represents sounds and certain relationships among a collection of sounds. This representation is used to both categorize and compare those sounds or segments thereof. Two basic code objects are used to represent sound files and segments: tree/node structures and units.

A sound file is associated with a sampler node (`SamplerNode`). Each sampler node is linked with a SuperCollider synthesis object (Synth) that plays the sound file from disc. Accordingly, effects nodes (`EfxNode`) are defined by a Synth that further processes the sound from a linked sampler node. Transposition is considered within this system as a component or characteristic of the parent file, not a transformation. Parent-child relationships (Figure 1) are formed between sampler nodes and effects nodes; there

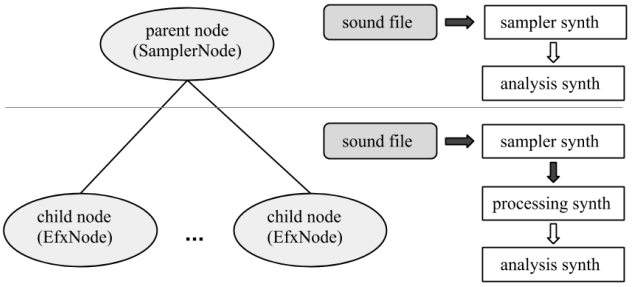

Figure 1: Parent and child nodes.

can be parent nodes with no children, but there are no free-floating child nodes. Each node contains a recipe for recreating the associated sound.

Given trees of sound-producing nodes, analysis metadata is represented in the form of units. Each unit defines a link between a segment of a sound file node and its metadata. The majority of the data is collected automatically or as the byproduct of dividing a sound into units. There is a tag category that allows the user to mark up each unit in a corpus with an integer that can identify some user-specified aspect of the underlying data. An example of the use of this tag information is included below.

The relationships between parent and child nodes are encoded directly in the unit table for a corpus: a unit with matching parent ID and sound file ID is a parent node, whereas a unit with differing values is easily identified as a child or effects node. These child nodes carry references to their parents, so that their sound processing chains can be reconstructed by connecting nodes. The depth of the tree is currently limited, so that there can be no children of children.

Corpus units contain only integers and floating point numbers, and are meant to be useful for large-scale numerical processing. Each unit in a corpus contains information identifying its source sound file, transposition, duration, as well as the analysis metadata. Multidimensional raw analysis data (Mel-frequency Cepstral Coefficients or MFCCs) collected per-frame are averaged over a segment comprised of multiple frames. Parametric data–that is, data with a single value per frame–is stored as several statistical measures over the whole segment: mean, max, initial and final values, and the slope from initial to final value. In the current version, amplitude–actually, spectral power–parametric data is collected. Overall, just two tables are maintained: the sound file tree and the corpus-wide list of units.

Access to Corpus Data

The entire collection of segmented and analyzed sound file units is available in a variety of forms. Depending on the needs of the user, a corpus can be indexed globally, relative to sound files, or relative to user-specified groupings or criteria. Global access to metadata is possible as both a mapping between integer indices and metadata rows, or in a tabular format.

These table views (Figure 2) can be limited by types

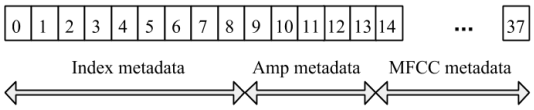

Figure 2: Corpus unit row.

of metadata: all columns, just the index (or identifying) metadata, just the amplitude metadata, or just the MFCC metadata. Using the index metadata, access can be controlled so that only units meeting certain requirements are returned. For instance, one might construct a query that returns only untransposed units (units with a transposition value of 1.0) that do not come from the same sound file. This can be accomplished in a variety of ways: through the use of list comprehensions, *sclang* list functions, Numpy's search functions, etc. Access filters of arbitrary complexity can be chained together, and the results can be stored for future use. Furthermore, introspective methods may be used to control the quality of the data returned. For example, sets of units returned may be subject to a minimum size requirement, should the user require a certain minimum number of intermediate results to pass on to further stages of processing.

One important access-pattern bears special mention. It is possible to tag units with integers that identify them in some way. It is left to the user to define categories or some other labeling scheme. One possible example involves tagging units as head, body, and tail units. A head unit is the first unit from a sound file or those units identified as containing attacks of sounds. A tail unit is the last unit of a sound file or those units which precede units tagged as attacks in sound files with multiple sonic events. By virtue of not being a head or tail, all other units are labeled as body units, as Figure 3 shows. This tagging scheme allows for the grouping of sounds so that a search amongst a group of tail units returns a tail unit. This mitigates a situation where a unit tagged as an attack potentially returns matches that do not contain attacks, and likewise, matches for body or tail units do.

Since units can be tracked as lists of integer indices, it is easy to create multiple, overlapping groupings of units, either on demand or offline. These groupings of units are simple abstractions of sound material. This also means that audio segments are always referred to symbolically, reducing computational complexity and memory requirements.

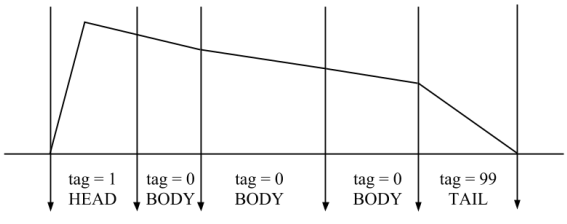

Figure 3: Tagging units.

Searching Corpus Data

Once sound files are analyzed and segmented, the resultant metadata is accessed in order to perform a range of tasks. The user can execute a series of modular operations in order to search, rank, reorder, swap, and otherwise manipulate sounds as segments or tiles. The targets for these search steps are metadata from units, sequences or collections of units, or even virtual representations of units (e.g., the mean MFCC values over a combination of several actual units). New unit sequences are created based on existing sequences, via search protocols that may be designed with many different combinations of and variations on the following steps:

1. Group, filter, and partition units (see above).
2. Select a target unit or sequence.
3. Search by similarity (with optional weighting).
4. Re-rank, filter, or limit the number of results.
5. Reject results based on heuristics or analysis of results.
6. Represent the results in some form, either symbolically or as resynthesized audio.

The selection of a unit or sequence of units to act as targets for search, itself a significant choice, is left to the user. A sequence of units can, and often does, correspond to a real series of segments in a sound file or the results of a previous substitution operation, although extensions to this method are discussed below. It is possible to chain together a series of searches for a unit similar to a target, using the resulting unit as the target in each subsequent search. This is very similar to classic concatenative synthesis, implemented in a number of ways, notably in CataRT.

Selection and Search

Search-by-similarity, as currently implemented in CorpusDB, is measured using Euclidean distance between MFCC data. Since the SuperCollider version is focused on real-time use, KDTree (Stowell 2013), a plugin, is useful for basic search-by-similarity. Various offline comparison functions are included as part of a Python class. These Python functions utilize Numpy for computations involving large sets of numerical data. Since both versions allow for the caching of search results, this allows for one-time calculations of nearest neighbors and the mapping of each unit to a limited number of nearest units. The limiting number functions as a cap on the potential connections made between units within an analyzed corpus.

Search may be performed based on weightings or maskings of certain channels of search metadata. For instance, search may be simplified by assigning higher weights to the first few channels of MFCC data. Masking channels, by completely stripping the same channels from both the target unit and potential matches, serves to reduce the dimensionality of the data involved and can increase computational efficiency. Figure 4 demonstrates a series of simple operations using *sclang*.

The ability to cache search results and store the resultant rankings lends itself to weighted random selection. The distances of the nearest units may be used directly, or the rankings from closest to furthest may be mapped to a corresponding series of weights. Utilizing weighted random selection, the nearest unit to the target is more likely to be chosen than those further away. Furthermore, the rankings and/or the weights might be stored and tracked as part of a dynamic selection model where those rankings are adjusted after every search. Numpy and other libraries provide tools to work with large sets of unit data. The code in Figure 5 demonstrates search using Python similar to the above *sclang* example.

```
// a particular target unit
~targetUnit = mycorpus.cuTable[223];

// all units not labeled as heads or tails
~unitGrouping = mycorpus
        .convertCorpusToTaggedArray(tag:0);

// calculate an array of weights
~weights = [1, 0.98, 0.96, ...];

// return the ten closest units
~result = mycorpus.findNNearestWeighted(
        targetUnit,
        unitGrouping,
        weights,
        n:10);
```

Figure 4: *sclang* code for finding similar units.

```
# Create a pool of MFCC data for units
# with tag 0.
unitdata_pool = myCorpusDB
        .convert_corpus_to_tagged_array(
                'M',
                tag=0)

# Select random unit to serve as a target.
pool_size = unitdata.shape[0]
index = random.randint(0,pool_size)
target = unitdata_pool[index]

ct = CorpusTracker(myCorpusDB)

# Return the ten closest units to the
# target unit.
nearest_ten = ct.rank_units_by_euc_dist(
        target,
        unitdata_pool,
        depth=10)
```

Figure 5: Python code for performing ranked search.

Re-ranking, Filtering, and Limits on Search Results

Once a series of nearby units has been gathered, a subsequent ranking made based on a distinct criterion may be used to reorder the units. The re-ranking step refines the results of the first search, which is based on matching similar ma-

terial. It is different from weighted random selection, in that it re-ranks based on criteria independent, or partially independent, of the original distance measure. This re-ranking step is an intuitive way to force a search algorithm to make perceptually relevant decisions. For instance, if the original selection is based on timbre similarity, a re-ranking based on the average power values of the similar segments can be designed to favor those with amplitudes similar to the target unit's and thus more amenable to insertion while minimizing discontinuity (see the above discussion of tagged units).

A simple filtering of all units whose power is lower than the target's (or below an appropriate relatively lower threshold) will leave only those units that can be substituted with a minimal amount of attenuation (or amplification) to match the power of the target unit. This extension is inspired by previous work (Maestre et al. 2006) specifically aimed at segmentation of a corpus of prerecorded monophonic instrumental recordings, using the sounds' amplitude envelopes to guide concatenation. It must be stressed that the ordering of the various phases of grouping, search, ranking, and general post-processing is fairly open-ended and modular. The filtering of units according to power may happen as part of the formation of pools of potential matches before search.

Multiple searches, with the results averaged or subjected to a majority-voting scheme, are possible. More generally, filtering and limiting of the number of results may be performed at any step. This last procedure, limiting the number of results, is optional, but for practical purposes should be performed when finally returning results or in intermediate steps. It may be difficult to know how many results to retain; it is left to the user to determine a workable limit.

Complex Search Example

In order to illustrate the use of several selection and search functions, a more involved example is in order. While this algorithm should return useful results, given sufficient numbers of units with compatible metadata, this example is primarily meant to show a multi-part search process. The following steps can be coded (see Figure 6) in either version of the software:

1. Divide corpus into sub-corpora based on transposition values (line 1).
2. Select a sound file to be the target (line 8) and iterate over each of its units (line 13).
3. For each unit, search for the nearest unit, searching only among the sub-corpus with the same transposition (line 15).
4. Limit the results of each search to the top 1000 hits after ranking by calculated Euclidean distance (line 18).
5. Reorder those results based on the differences between each result and the target unit's amplitude (line 20). Filter out any units with power, as averaged over the segment, more than 3 dB below the target's (line 24).
6. Collect the results of this re-ranking (line 28).

These results are stored for each unit in the sequence, and a map from target units to corpus units is created. This mapping of units in a sequence to similar units from a (sub-)corpus can be used to create similar sequences of units and thus comprises a particular way of producing remapped sound files or variational edits. This is just one example of how several of the above steps are combined. Most of the above steps are facilitated by specific functions implemented in both variants of CorpusDB. Of course users are free to adapt their own functions or create new functions that fulfill specific requirements.

```
1  all_transp_0_5 = ct.all_transpositions(0.5)
2  # ...
3  # all units with transp 0.5 are in our pool
4  mfccs_transp_0_5 = mfccs[all_transp_0_5]
5  amps_transp_0_5 = amps[all_transp_0_5][:,0]
6
7  # assume sound file 35 has transp = 0.5
8  sf_35_units = ct.all_of_sf(35)
9  sf_35_mfccs = mfccs[sf_35_units]
10 sf_35_amps = amps[sf_35_units]
11
12 result = []
13 for i,mfcc_row in enumerate(sf_35_mfccs):
14
15   ranked = ct.rank_units_by_euc_dist(
16     target_mfccs = np.atleast_2d(mfcc_row),
17     pool_mfccs = sf_35_mfccs,
18     depth = 1000)
19
20   reranked = ct.rerank_units_by_amp(
21     target_amp = amps_transp_0_5[i],
22     pool_amps = amps_transp_0_5[ranked])
23
24   filtered = ct.amps_greater_than(
25     subset_ids = reranked,
26     threshold_unit = int(sf_35_amps[i,0]))
27
28   result += filtered
```

Figure 6: Python code for complex multi-stage search.

Further extending the power of CorpusDB

The above sections outline the basic components of CorpusDB and their use in building musical or compositional work flows. There are further extensions of the basic functionality of CorpusDB that may be applied to the process of creating new series of sonic tiles.

Searching on Sequences of Units

When more than one target unit is used, the target unit becomes a sequence. The Python version of CorpusDB includes an implementation of a search function that searches for one anchor unit as in Figure 7, and then re-ranks results based on user-specified, variable numbers of neighboring units. This is but one possible algorithm that would search based on sequences. Furthermore, the algorithm allows the user to specify a variable swap region: the units on which search is based do not have to be the only units that are swapped in for the target units.

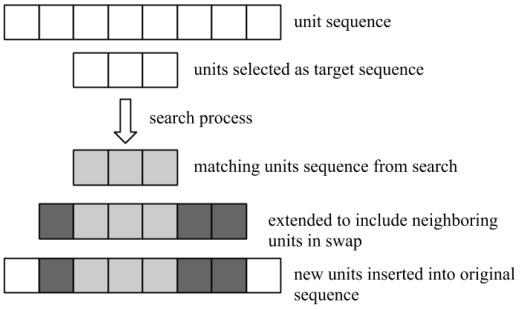

Figure 7: Search based on unit sequences.

The author has found that the results of a mapping from targets to corpus oftentimes over-represents some units in the corpus. These well-connected units are heard in resulting substitutions, and the ear can become attuned to the repetitions and familiarity caused by hearing them more often in the results. This situation can be remedied by a post-processing phase where the results lists are compared as a group. Those units with a high rate of occurrence as potential matches are assessed a selection penalty (in a weighted scheme) or removed altogether. This step will inhibit the selection of over-represented units, and, as a corollary, promote the relative representation of units with lower overall connectivity within the corpus.

Statistical Analysis and Views of Units

The observation that units are likely to be less than uniformly distributed within a corpus or subset thereof, leads to a desire for analytical tools to inspect the results of a search and view the relative distribution of units that comprise some grouping, or to view the general connectivity of the corpus. Both implementations of CorpusDB, but especially the Python version, have functions that allow for the introspection of corpora and other groupings of units.

It is easy to view units in terms of their relative connectivity, as based on timbral features. Using Numpy and Matplotlib, the Python version can be used to produce a distance or similarity matrix (Figure 8) that shows the relative distances from each unit in the corpus to every other unit. This example shows a corpus made up of two source sound files, which can be seen in the graphic. Simple statistical analysis will reveal the means and variances across each channel of multidimensional data. This statistical data can be used in the search process to normalize distance functions so that channels with large variance do not dominate channels with relatively narrow ranges of values. While much of this analysis is possible using the *sclang* version, Python's libraries for numerical computation, graphing, and visualization are superior and easier to use.

Using Metadata to Drive Resynthesis and Resampling

The results of search algorithms are lists of unit IDs that correspond to segments of sound files. One must reconstruct or, in the case where sequences of noncontiguous sound file

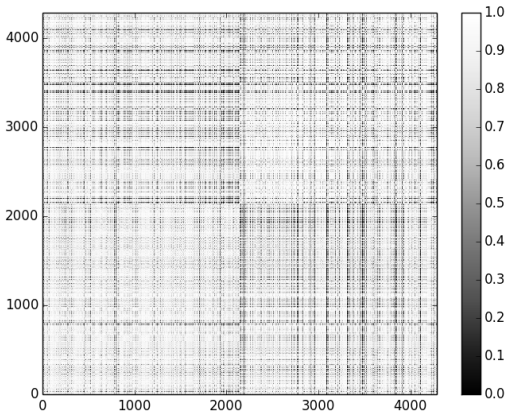

Figure 8: Similarity matrix.

segments are derived, construct sound files by sampling the appropriate files based on timing data and metadata, identifying which file to use for each specific unit. It is straightforward to assemble the appropriate data and create audible results of a swapping or remapping operation:

1. Gather a list of unit indices and convert that list into unit metadata.
2. Iterate over that list and convert unit sequences that map contiguous segments within the original sound files to segments with onsets, durations, and sound file names.
3. Loop over the assembled segments.
4. Pass each consecutive segment to a synthesis object(s) that renders the sound. Compose the signal chain for parent sound file nodes and child or transformed nodes by properly instantiating synthesis modules and connecting audio buses.
5. Wait for the duration of each segment.

Using either version of CorpusDB, it is easy to implement a function to perform the reconstruction offline and save the result to disc. This method can allow for the rapid creation of multiple variations on a target sound file. The SuperCollider version includes functionality for online or real-time assembly of sound tile sequences. Both versions include example code for assembling audio segments for playback.

Conclusion

This paper demonstrates several use-cases and applications of the methods included as part of CorpusDB. The user is left to construct organization and analysis algorithms composed of these methods that utilize the particular structure of a CorpusDB instance in order to accomplish the desired musical goals. CorpusDB's open-ended and modular design is intended to serve as the starting point for diverse algorithmic procedures including composition, analysis, exploration, etc. CorpusDB is open source software available from github.com/kitefishlabs/CorpusDB (Python

project) and github.com/kitefishlabs/cbpsc (SuperCollider *sclang* project).

Acknowledgements

The author wishes to thank all the reviewers for their thoughtful comments.

References

Casey, M. 2009. Soundspotting: a new kind of process. *The Oxford handbook of computer music* 421.

Collins, N. 2012. Automatic composition of electroacoustic art music utilizing machine listening. *Computer Music Journal* 36(3):8–23.

Eigenfeldt, A. 2011. Real-time composition as performance ecosystem. *Organised Sound* 16(2):145–153.

Hunter, J. D. 2007. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering* 90–95.

Maestre, E.; Hazan, A.; Ramirez, R.; and Perez, A. 2006. Using concatenative synthesis for expressive performance in jazz saxophone. In *Proceedings of International Computer Music Conference*, 219–222. San Francisco, California: International Computer Music Association.

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

Oliphant, T. E. 2006. *Guide to NumPy*. Provo, UT.

Oswald, J. 1999. Plunderphonics. http://www.plunderphonics.com/.

Schwarz, D. 2006. Concatenative sound synthesis: The early years. *Journal of New Music Research* 35(1):3–22.

Schwarz, D. 2007. Corpus-based concatenative synthesis. *Signal Processing Magazine, IEEE* 24(2):92–104.

Stowell, D. 2013. http://sourceforge.net/p/quarks/code/2634/tree/DataStructures/KDTree/. Accessed: 2013-08-19.