DANA GRETTON
MIT EECS COURSE 6 2016
11/12/16

Signal Processing for Placement of Audio Events in Time and Aural Space for Sound Collaging

Report for *Undergraduate Advanced Project (UAP)*
with Professor Dennis Freeman, Faculty Supervisor

Code available at https://github.com/dgretton/mu-sp-UAP

Introduction

Over the last 2 ½ years, the author has pursued the development of a sound collaging software system applying the mathematical tools acquired in MIT undergraduate course work. This project stems from a desire to find a useful application for the purely theoretical techniques covered in class, and a lifelong interest in vocal and instrumental music.

Certain types of music and other structured sounds can be thought of as being recursively defined: sounds are composed of shorter sounds, and extents of time encompassing musical phrases or sound events are composed of shorter intervals. While most music notation explicitly spells out each note, this self-similar structure naturally lends itself to a recursive functional expression style. Programmatic sound specification and synthesis software packages have been developed for many years, but they tend to follow a sequential design pattern similar to sheet music, where sounds are placed one at a time with coordinates. JSML and JSynth work this way, for example. They have a strong focus on explicit definitions and temporal sequences. It was desired to create a software representation of sound compositions with recursive structure to enable development of these compositions directly in the domain in which they are perceived.

A recurring characteristic of the currently available tools is their failure to take into account the way that sounds can be perceived as hollow or fake if they are repeated exactly. For example, a drum track with the same repeated snare sound is easily identifiable as a synthetic

sound. One goal of the ongoing project is to provide tools for specifying and composing "compilable" sounds, that is, sounds whose blueprints are written out, but whose details are not filled in until they are compiled, possibly not the same way every time. The primary application is in music production, and it might also find uses in multimedia or scientific explorations into auditory perception. Since a sound's placement within a recursively defined time structure and the properties of a sound could all be interrelated, an integrated solution that does all of its own signal processing whenever necessary was pursued.

A large part of the interest in a sound is in the sensation of its having a defined position in a spatial environment. At the time of the proposal for this work, the author's project could only spatially render sources by delays to ears, localizing only within the horizontal plane. That shortcoming limited the system in its goal of providing the richest possible set of sound collaging options. In order to reproduce the experience of a sound emanating from a point in space with headphones, each ear must receive a signal slightly different from the source sound, with its frequency content modified in amplitude and phase to reflect the way sounds change as they travel around the head and enter the inner ear. Included in this adjustment is a slight delay between ears, related to the delay between the arrival time of the sound at each ear depending on its distance from the source, for example. These combined effects are represented by a Head-Related Transfer Function (HRTF), which is an impulse response that captures the adjustments to a sound due to its particular location that were experienced by a dummy with a microphone in its ear. A collection of HRTFs from points all around the head in three dimensions comprises a data set capable of rendering sounds that seem to emanate from arbitrary points in space. Many HRTF data sets are available from laboratories that do audio research. The 1994 KEMAR-dummy data set from MIT's Media Lab was used in this project.

The work documented here comprises a new set of tools to be used in the ongoing sound collaging project to provide convincing stereo sound rendering in 3D aural space using frequency-domain manipulations of mono source data. The concept of an aural space, a simulated setting from which sounds seem to originate, was explored and made concrete with the AuralSpace class. Aural space transfer functions (ASTFs) were defined to generalize HRTFs to enable some effects that go beyond simple frequency response, like distance delay, echo, and reverberance. HRTFs were incorporated into the general framework of AuralSpace and ASTF to accomplish 3D sound rendering.

Structure of the pre-existing system

INTO WHICH THE WORK FOR THE MIT EECS 2016 UAP WAS INCORPORATED

Usage

The system is presented to the user as a Python module. To compose a sound collaging project, any methods available in Python may be employed to assemble the provided programmatic objects to specify the framework elements: for example, Tracks, Sounds, and

Beats are all elements the user might create and arrange using loops, recursive functions, their own imported metadata, other modules, or any other procedural means.

The output of the system is one stereo audio file at a time. The completed framework definition made of Tracks, Sounds, Beats, etc. is passed to an efficient rendering pipeline, coordinated by a Mixer, that reads only the minimum data and performs only the minimum calculations necessary for a particular requested output, e.g. rendering a short segment in low fidelity to a temporary file to play a prototype out loud, rendering particular sounds or tracks to separate files, or compiling the whole work in full definition.

Elements

The following are classes from which Python objects may be instantiated. No calculations on audio data are performed upon construction of any of these objects, which is why they are collectively referred to as the "framework" of the user's project.

Beat

The Beat is the basic unit used for specifying every audio event's time. The whole work is one large Beat, which is recursively "split" into a tree structure of nested sub-Beats, each of which represents a fractional part of its parent's duration. Beats never overlap their siblings. Whenever a Beat's duration is set, it propagates its setting to constrain all other Beats to the maximum extent allowed by their definitions. (A leaf Beat whose duration is set to a tempo could determine the length of the whole work, for example.) An error is raised if a Beat becomes over-constrained.

Beats may be split, returning a list of child beats:

- *evenly:* a Beat is split into some specified integer number of children constrained to have equal duration.
- *proportionally:* a Beat is split into fractions according to a list, e.g. [.25, .25, 2] constrains the first two of the Beat's three children to one-eighth the duration of the last.
- *elastically:* one or more of the Beat's children are given unassigned durations; their durations must be resolved by constraints within their Beat sub-trees.
- *rhythmically:* the Beat is split as in the proportional case, but an external "rhythm" file is imported that sets some saved relationship between the child Beat durations.

Sound objects (see below) are "attached" to Beats along with the locations from which they will be rendered. All of a Beat's Sounds may be enumerated with a function call.

A Beat's "time" property is evaluated lazily when it is requested by a function call, at which time it searches up its sub-tree for a defined beat from which it can derive its own

duration and time. If a beat tree is collectively under-constrained when a Beat's time is requested, an error is raised.

Sound

Sound is a base class that provides some basic functionality for all derivative Sound objects, including the stereo rendering functions, caching infrastructure, and others. A global parameter "quick play" controls whether all sounds will be rendered quickly at low fidelity.

Sounds are initialized independently and may exist on their own. Most sounds take other sounds as arguments in their constructors, with the exception of RawSounds, which handle mono data from disk. Sounds do not have locations, but may be rendered from any specified location. A sound will only be rendered if it is attached to a beat, or explicitly rendered by itself.

There are many types of Sound.

- **RawSound:** constructed with a file name and only returns the file data rendered to stereo.
- **RandomSound:** when rendered, selects one from a list of sounds and renders that one from its location.
- **SpreadSound:** renders many sounds around the location from which it is rendered with a Gaussian spread.
- **ClippedSound:** cuts a segment out of a longer Sound.
- **RandomIntervalSound:** takes random pieces out of a sound and strings them together for an arbitrarily long time to produce relatively uniform, long-duration textures
- **ResampledSound:** uses a provided function to change the instantaneous pitch and speed of a Sound

PitchedSound is a derivative class of Sound that maintains a property "pitch;" whenever this type of sound is rendered, it automatically resamples itself to change the pitch of its source to its pitch. PitchedSounds are pluripotent respecting pitch, but may be converted to ordinary Sounds by their "for pitch" class function.

- **RawPitchedSound:** extends RawSound.
- **RandomPitchedSound:** extends RandomSound.

When a sound returns its rendered data, it returns both a stereo array and a "registration point" that specifies where in the data the sound should be regarded to start (a reversed sound, for example, might want its registration point at its end).

Sounds are of indeterminate length until rendered. To find out whether Sounds should or should not be rendered for a passage, each sound is obliged to provide an estimate of its rendered duration that must be with very high probability an overestimate at worst.

Track

Tracks are containers for Sounds attached to Beats that have concrete places in time. Each track has a "top Beat," and the sub-tree of the top Beat contains all of the Track's Sounds. Given a start time and an end time, Tracks decide which Sounds to render, managing render buffers and Sound-level mixing. A Mixer, defined below, collects several tracks, and might render them all together or to separate files.

Mixer

The Mixer is the front-facing component of the render pipeline. To play or save a composition, the user makes a function call to a Mixer suitably configured with Tracks, Beats, and Sounds. "Quick play" mode may be specified when making a Mixer call to skip computationally intensive transforms and return distance-delayed-only data in a much shorter time.

The rendering pipeline

Whenever a Track passage is rendered, a start time and an end time are specified. Rendering begins by initializing a working buffer of enough zero samples to fill the specified delta in time. Each Beat's time is calculated by in-order traversal, and a determination is made whether its attached sounds could overlap the passage based on their times, registration points, and estimated maximum durations.

Overlapping sounds are rendered in turn. Whenever a sound involving mono data is rendered (that is, a function is called that returns two arrays of numbers representing its final stereo waveform), it is rendered from the location that was specified when it was attached to its Beat, passing its mono data to the function *_to_stereo(mono_data, location)*. Here, the mono data is either shifted and scaled to make a rough stereo approximation, if a quick test play is desired, or rendered fully. In the case of a full render, the mono data is fourier transformed, frequency-domain adjustments are applied, and the result is transformed back to the time domain. These adjustments include the time delay between ears associated with the direction of sound at the time of this proposal.

Many sounds don't generate their own data at all, but only select what data is generated or edit or compose their child sounds' data. Original rendered data may be passed up a stack of sounds applying consecutive modifications. When the original sound returns its rendered data, it is mixed into the data buffer; any samples going off the edge of the passage are discarded. The working buffer is written out to a file and, optionally, played immediately.

Simulated Aural Spaces

The project for the MIT EECS 2016 UAP is an augmentation of the pre-existing system that supplies a suite of tools for rendering sounds in particular locations within a simulated aural space. An aural space is a theoretical model that establishes what it means to render a sound "at a location:" the model may be simple or complex, but it always specifies the way that a sound changes as it interacts with objects in a simulated listening context, including the simulated ears of the listener. Using the aural space framework, sounds could seem to originate across a reverberant concert hall, over a broad lake, or inside a small library room. One simplifying assumption made for all aural spaces is that they are causal linear, time-invariant (LTI) systems. They are causal in that they disallow sounds' having effects in their aural spaces' outputs before the sounds start. Aural spaces' linearity means that two sounds rendered simultaneously in an aural space give the same result at the ears as the same two sounds rendered separately and added up. Aural spaces are time-invariant in that their properties only change slowly compared with the durations of their sounds, or not at all. A general property of an LTI system is that it can be completely described by adjustments to the frequency spectrum of its inputs, captured by a transfer function. Aural spaces are therefore generators of transfer functions that are dependent on 3D location. A new base class, AuralSpace, was created to provide a foundation for all kinds of aural-environment-dependent frequency-domain manipulations. An AuralSpace has one job: to produce an Aural Space Transfer Function (ASTF) for any 3D location in its simulated environment. The new ASTF class is described below.

ASTF (Aural Space Transfer Function)

An ASTF object is a symbolic representation of a frequency-domain causal transfer function for a particular location. It has a generator function that, when called, produces concrete transfer function data for its location. This data includes the length of the impulse response of the transfer function. Usually, the ASTF is used once to convert a sound to stereo, and its memory is then freed. Still, AuralSpaces might hang on to and cache several ASTFs as an optimization, or as building blocks for other ASTFs.

In this sound collaging system, an HRTF (e.g. from the MIT Media Lab KEMAR HRTF set) is a type of ASTF that only depends on the azimuth and elevation angles around the head. An ASTF from an AuralSpace using the KEMAR data set is different from a pure KEMAR HRTF in that it depends on the sound's radial distance. For example, it might fade away and become more delayed with the location's increasing radial coordinate. The exact details are left up to the AuralSpace implementation, however.

ASTF objects are also the natural vehicles for caching and lazily retrieving pre-rendered transfer function data.

AuralSpace

AuralSpace is an abstract class that requires its descendants to produce an ASTF object for any given 3D location. It also provides some general functionality for "moving" ASTFs to a first approximation between locations by adding a delay and scaling their volumes according to their differences in coordinates. Lightweight functions for moving ASTFs are useful for AuralSpace subclasses that create approximate ASTFs but need to tweak them before they are applied.

Several subclasses of AuralSpace were created.

- **EarDelayAuralSpace:** This simple aural space produces ASTFs that only delay sounds and scale their volumes using the AuralSpace ASTF-moving utility functions. EarDelayAuralSpace ASTFs give the same frequency response for each circle around the horizontal axis, so though they are good for localizing sounds between the sides of the head, they provide no cues about vertical position, nor whether sounds come from the front or back.. EarDelayAuralSpace effectively simulates a horizontal, two-dimensional, empty context for sounds with no walls or objects.
- **DiscreteAuralSpace:** This general-purpose subclass provides tools for its own subclasses to keep track of and cache a discrete set of ASTFs. It requires its subclasses to provide methods for specifying the nearest discrete locations to any particular location (N Nearest Locations) and creating source ASTFs (Create ASTF). Since it caches, loads, and manipulates ASTFs from predetermined points, it is ideal for HRTF implementations.
- **DiscreteEarDelayAS:** Subclassing both EarDelayAuralSpace and DiscreteAuralSpace, DiscreteEarDelayAS effectively wraps EarDelayAuralSpace, caching EarDelayAuralSpace ASTFs from a set of discrete points in the horizontal plane forming a circle around the listener. This class served as a test ground for DiscreteAuralSpace.
- **KemarAuralSpace:** This AuralSpace fulfills the specification outlined in the project proposal. Subclassing DiscreteAuralSpace, KemarAuralSpace utilizes the MIT 1994 Media Lab KEMAR dummy HRTF data set to simulate the frequency response expected for any angle with the head. EarDelayAuralSpace effectively simulates a three-dimensional empty context for sounds with no walls or objects. More details of KemarAuralSpace follow.

KemarAuralSpace

The incorporation of the MIT Media Lab KEMAR HRTF data was accomplished by creating a DiscreteAuralSpace with discrete points at the KEMAR HRTF sample locations, each point corresponding to an impulse response file from the data set. As a DiscreteAuralSpace, KemarAuralSpace must provide two methods: N Nearest Locations, and Create ASTF.

N Nearest Locations

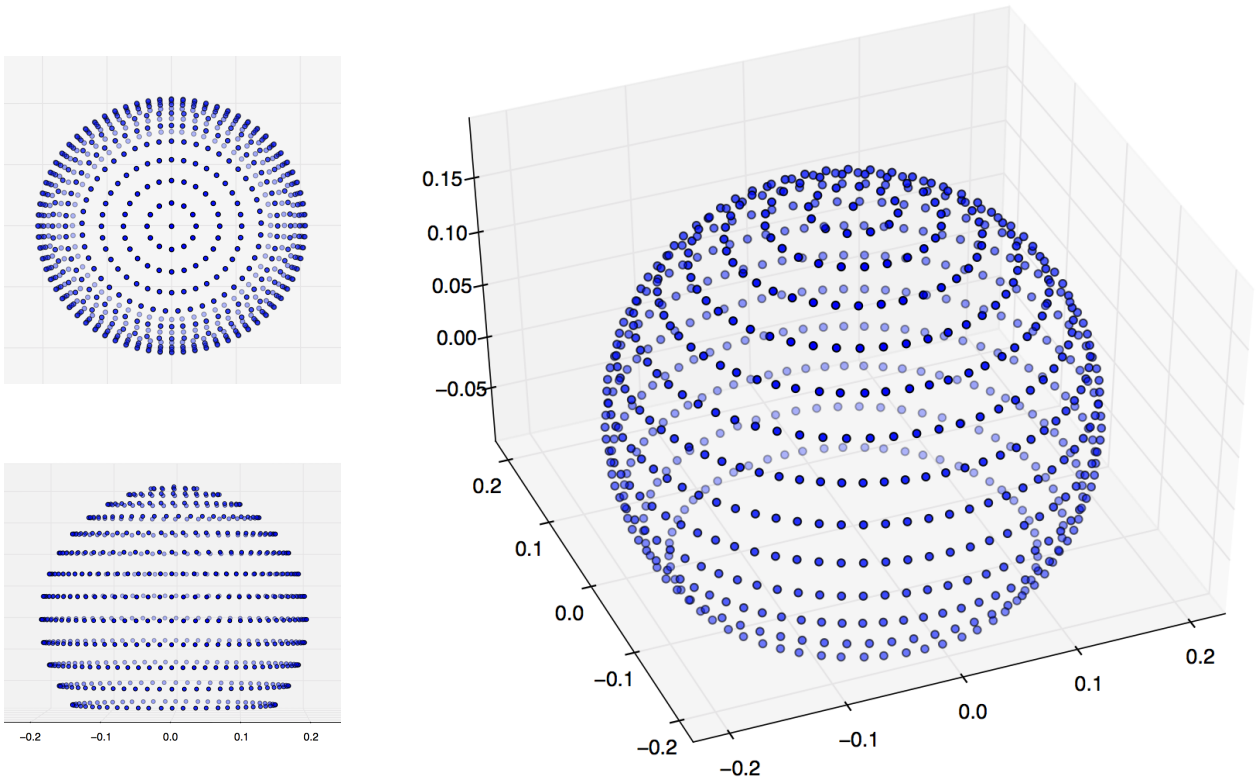

Fig. 1: Top, side, and oblique views of the KemarAuralSpace HRTF locations positioned at the standard radius 0.2m

The KEMAR HRTF impulse responses were collected by the MIT Media Lab researchers from 368 unique locations all lying on a sphere of radius 1.4 meters at 10 degree latitude intervals and roughly 5 degree longitude intervals. Only points on one half of the sphere, the "left" half with respect to the listener, were measured. The other half were assumed to be symmetric: the left and right stereo channels are swapped to create the impulse responses from the other half of the sphere. (Points lying on the meridian have identical impulse responses in both channels.) Points below 40 degrees latitude from the horizontal plane also were not measured due to geometric constraints on the MIT Media Lab researchers' test setup. Not double-counting the meridian points, there are 710 locations with unique HRTFs, shown in Fig. 1.

To find the N nearest locations with unique HRTFs to a given location, the vector cosine similarity is used to assess "nearness." If each HRTF location is represented by the vector $\vec{a}$, and the given location is represented by the vector $\vec{b}$, the cosine similarity is defined as

$$\cos\theta = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

The cosine similarity peaks for an HRTF location at $\vec{a}$ when the angle between the given location and $\vec{a}$ is smallest. The list of KEMAR HRTF points is sorted in decreasing order by the cosine similarity with the given location at $\vec{b}$. Then, the first N points in the sorted list of HRTF locations are returned.

Create ASTF

ASTFs of the form required by the interface of the ASTF class are created lazily, that is, as needed, from the KEMAR impulse responses at render time. Pre-processing is applied first, before readying the ASTF in the host computer's Random Access Memory (RAM). The HRTF impulse response is pre-processed to:

- convert its file name, which contains its latitude and longitude information, to a location object with the standard distance .2 meters from the listener;
- add zeros to its trailing end to size it for time blocks appropriate for the overlap-add convolution method (details below);
- transform it into the frequency domain with a real fast Fourier transform from the Numerical Python (numpy) FFT package;
- normalize it with respect to the other KEMAR HRTFs so that they collectively have a unitary effect on sounds (they do not affect the sounds' amplitudes on average).

The DiscreteAuralSpace parent class uses the KemarAuralSpace N Nearest Locations and Create ASTF functions to prepare an ASTF for any given location. By default, it creates the 1 nearest neighbor to the given location and uses DiscreteAuralSpace's ASTF-moving utility methods to delay and volume-adjust the ASTF data to match the delays and volumes expected from the given location. DiscreteAuralSpace caches the unadjusted data to speed up retrieval of ASTFs from similar locations in the future.

Fig. 2 shows the ASTFs created for some sounds in a complicated curved path around the listener. The points in a cluster close to the origin are the unadjusted ASTFs generated and cached by KemarAuralSpace. The further points connected by lines show the moves KemarAuralSpace made using the DiscreteAuralSpace ASTF-moving functions. KemarAuralSpace makes the shortest possible move in each case.

Overlap-Add Implementation

When a sound is rendered by the sound collaging system in an AuralSpace, the frequency and phase adjustments captured by an ASTF from that space must be applied to the raw mono-channel audio data to hear its effects. Application of a transfer function amounts to convolution of the sound with the ASTF in the time domain or multiplication of the sound by the ASTF in the frequency domain. Specifically, either the impulse response of the ASTF must be

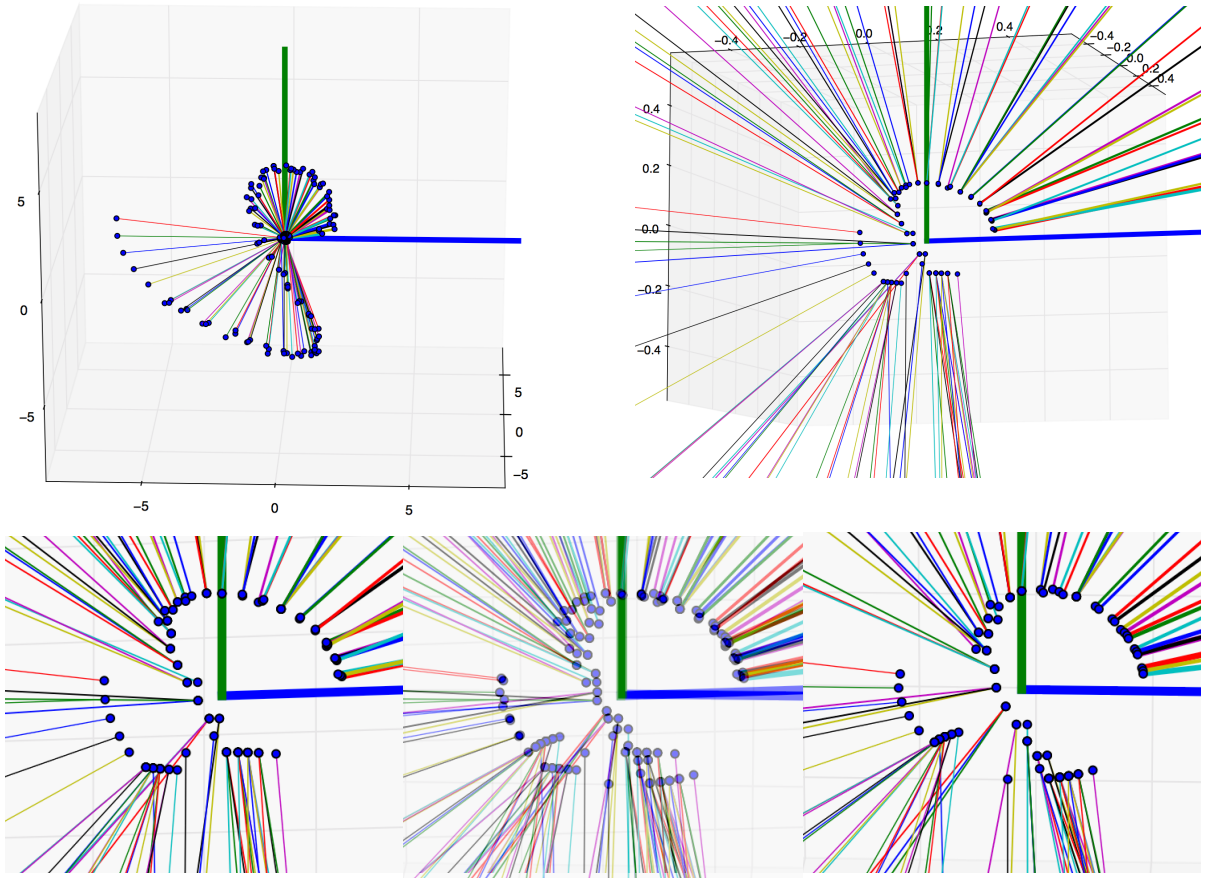

Fig. 2: ASTFs generated by KemarAuralSpace for sounds rendered in a complex curved path around the listener. Thick blue line is listener's "forward" and thick green line is listener's "up." Zoom shows set of unadjusted ASTFs generated from the partial sphere of KEMAR data points, and lines show how KemarAuralSpace moved these ASTFs to their final locations. The closest KEMAR HRTF was used in each case. Bottom triptych attempts to disambiguate the zoomed image by showing the 3D graph view tipping downward a few degrees.

convolved with the source sound data, or the Fourier transform of the sound must be multiplied by the Fourier transform of the ASTF, each method giving the same results. Frequency domain ASTF multiplication is in general asymptotically more efficient than time domain ASTF convolution for the following reason: fast Fourier transforms (FFTs) of numerical arrays of length n run in $O[n \log(n)]$ time and multiply in $O[n]$ time, while convolutions of numerical arrays of length n run in $O[n^2]$ time. However, this explanation is incomplete because the ASTF impulse response may be much shorter than the sound. In that case, the impulse response is padded with zeros to length n before the transform step to ensure commensurate frequency-domain multiplication. The asymptotic run time of the frequency domain multiplication approach is still $O[(n)\log(n)]$, despite the very short impulse response length a, while that of the time domain convolution approach is $O[na]$. In this case, as n becomes large, time-domain convolution is favored. To realize the performance benefits of the frequency domain multiplication approach, it

Constant of proportionality *k*	chosen *m*	Avg. benchmark time over 3 trials (s)
1.1	256	34
1.5	256	34
2.1	512	24
2.5	512	24
3.1	512	24
3.9	512	24
4.1	1024	21
4.5	1024	21
8.1	4096	21
175	32768	**20.7**
400	65536	22
800	131072	25
—	(prime) 32771	**136**

Table 1: Benchmark execution times for various constants of proportionality used to determine minimum ASTF length, and demonstration of low performance with prime *m*. Benchmark times in bold show a large difference in performance despite very similar *m*.

must be avoided to slow down the Fourier transform with the $O[n]$ padding zeros added to the a-length impulse response before the transform.

Overlap-Add Method

The overlap-add method is a convolution technique designed to handle an impulse response much shorter than an input by splitting the longer n-length sound into smaller equal-length blocks, treating each block separately, and recombining the blocks into the finished sound data array. In this way, it limits the number of superfluous padding zeros added to the impulse response in each block. The impulse response of length a is padded to length m where $m > a$. To maximize the efficiency of the FFT, in this implementation m is chosen to be a power of 2. Then, a block of the sound data of length $m - a + 1$ is padded with zeros in a second data length m to fit the convolution of the ASTF impulse response with the block of sound data exactly. Since physical sound impulse responses are causal, the convolution will not add samples backward in time, so only the end of the data block needs to be padded. The Fourier

transforms of the impulse response and the data block, both now padded to length m, are calculated, multiplied, and the result inverse Fourier transformed.

Processing one block of sound data yields a new finished block that is longer than the original by a - 1 samples. The extra samples are accounted for by the response of the last several samples of the block, whose influence should extend over the next block if the convolution is carried out correctly. The blocks from the original sound overlap after they are processed. So, the end of the block overhanging the original length is saved, and when the next block has finished processing, the saved overhanging values are added in a vector fashion to the first values beginning the next block. The parts of the finished blocks that overlap each other are added, which gives the name "overlap-add" to the method.

Complexity analysis

Since about (m - a) samples of the input data are processed for each block, each of about $n/(m - a)$ blocks are individually processed. Each block is transformed, multiplied, and inverse transformed in $O[m \log(m)]$ time. Overall, the overlap-add run time is the product, $O[n \log(m)\ m/(m - a)]$. When $m/(m - a)$ is a positive constant, meaning that m is greater than a and proportional to a (e.g. the block length is double, triple, or another factor of the ASTF length), the overall runtime can be $O[n \log(a)]$: linear in the length of the sound, and logarithmic in the length of the ASTF. Compared with the convolution time $O[na]$, which is linear in both the length of the sound and the length of the ASTF, overlap-add outperforms convolution for large a (a still $<< n$). The performance boost lent by the frequency domain approach as a increases is important in the sound collaging system because new aural spaces with more complex echo and reverberant characteristics could have much longer impulse responses than those that exist now.

For fixed a, m is a choice to optimize performance. From above, it should be proportional to a but also a power of 2. Accordingly, this overlap-add implementation chooses m as the least power of 2 greater than a constant of proportionality k multiplied by a. A long benchmark task rendering many sounds of several seconds' length was used to find a minimal k (Table 1). Empirically, k = 1.1 times the impulse response ran very slowly. Larger k up to 175 showed an upward trend in speed. By k = 400 and 800, the $\log(m)$ term began to show as a slight slowdown. k = 4.1 was chosen because there was no significant difference between the k = 4.1 trial and the best trial.

As a demonstration of the importance of choosing m to have many small prime factors (implemented here by choosing m to be a power of 2) for achieving high fast Fourier transform efficiency, the benchmark was run with m as the nearest prime to the block size used in the fastest trial. Adding just three extra samples, from m = 32768 in the fastest trial to m = 32771 in the demonstration trial, the benchmark ran 6 ½ times slower.

The transform of the padded impulse response, the frequency domain ASTF, can be calculated just once and used repeatedly in the overlap-add method. DiscreteAuralSpace caches the frequency domain ASTF data because the transforms of ASTFs for similar locations can be saved and reused.

Verification

A short test program was composed using the sound collaging system to confirm the accuracy of the 3D KEMAR aural space simulation (shown below). Sounds were set up to emanate from each corner of a cube around the listener. Corners were were presented in randomized order. A batch of several instances of the same sound were rendered sequentially from each trial location, plus a small amount of jitter so that several HRTFs were engaged.

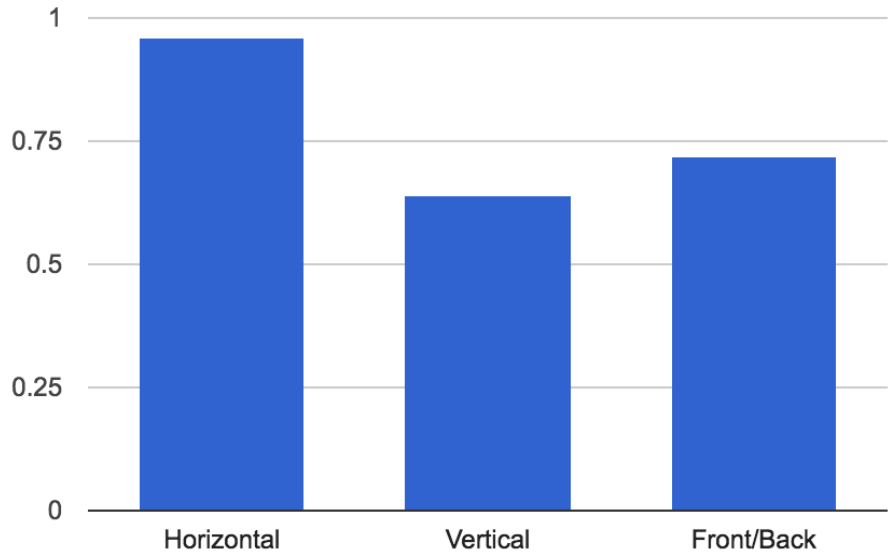

Fig. 3: Accuracy of subjects' identification of the direction from which the test program sounds originated. Discrimination between left/right was almost perfect, while vertical and front/back discrimination were not significantly above chance (subjects consistently gave the same answers repetitively).

The overall result was that there was good discrimination left/right, but most sounds gave the impression of coming from behind and above the listener, regardless of the test sound. The fetched ASTFs were verified to be the correct ones by four indicators: printing their names in the terminal showed the right range of angles for the vertices; a 3D graph showed the points noisily spread around cube vertices (Fig. 4); jittered ASTFs sounded different, within a small range; and trials corresponding to different cube vertices sounded significantly different, even if auditory localization was poor.

```
import random
from musp import * # import all Sound types, Mixer, Location, Track
from itertools import product
from mpl_toolkits.mplot3d import Axes3D

delay = .5 # time between sound repetitions in each batch
batch_interval = 2 # time between batches
batch_num = 7 # number of sounds in each batch

datadir = os.path.expanduser("~/.mu-sp") # data directory

def aulib(sound_dir): # construct path string for audio directory
    return os.path.join(datadir, "audio", sound_dir)

cube_size = 4 # cube is 8 meters to a side
test_locs = list(enumerate([Location(x, y, z) for z, x, y in product(
       [-cube_size, cube_size], repeat=3)])) # make cube locations
random.shuffle(test_locs) # randomize test order

# create a 3D plot of the cube
plot_pts = []
print "test point order:"
for i, l in test_locs:
    print i
    plot_pts.append([c for c in l])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(*zip(*plot_pts))

# using a random draw from a collection of similar sounds. KEMAR is the default AS.
source_sound = RandomSound()
# a few different sounds were tried
source_sound.populate_with_dir(aulib("crescendo_violin"))
# add some jitter to the sound locations to avoid just hearing one ASTF
test_sound = SpreadSound(source_sound, [cube_size/6.0]*3, 0, 1)

root = Beat() # top-level beat for this composition
test_track = Track("tests!", Sound.default_rate)

# make track take up the whole top-level beat by linking its root to "root"
# split it up into as many sub-beats as we have batches/test points
for batch_beat, (loc_num, loc) in zip(test_track.link_root(root).split(
             len(test_locs)), test_locs):
    batch_beat.set_duration(batch_interval)
    # split the batch beat into a rigid piece and an elastic piece
    # to take up any time not used for the sound repetitions
    test_beat = batch_beat.split([1, None])[0]
    for each_sound_beat in test_beat.split(batch_num):
        each_sound_beat.set_duration(delay)
        each_sound_beat.attach(test_sound, loc) # attach the same sound
        # multiple times in a row but at the same test location

# make a mixer and play the track
mix = Mixer("Let's do some tests, I guess...!", Sound.default_rate, [test_track])
```

Test program: renders short batches of repeated sounds at each of the vertices of a cube around the listener in random order, used to assess trueness of 3D simulation

The 3D plot of the centers of the SpreadSounds at the cube points is shown below, along with an extra plot of the locations from which the sounds were actually rendered in one test instance.

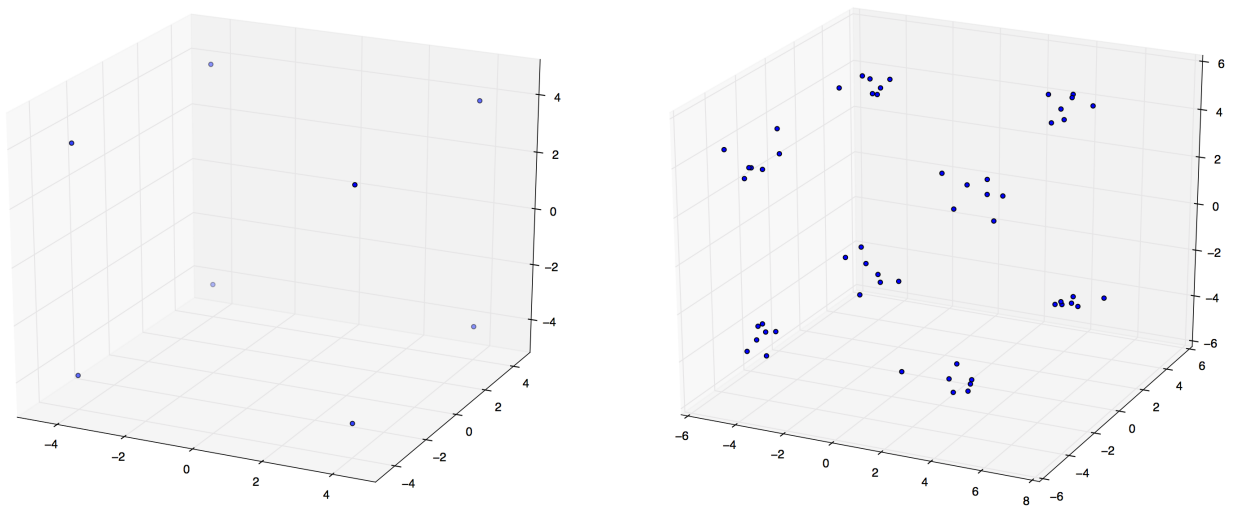

Fig.4: ASTF locations generated by the test program used to verify the subjective performance of the system with the help of some participants. Locations were chosen noisily around the centers of a cube of side length 8 meters centered on the simulated listener.

Conclusion

In the past few years, the author has pursued the development of a sound collaging software system for 1) specifying templates for algorithmically compilable sounds, which may be different with each compilation, and 2) structuring the compilable sounds in time to create compositions in a natural way that parallels recursive forms commonly found in sound art and music. At the outset of the project undertaken, the author's sound collaging software system could roughly render sounds originating at arbitrary positions in the horizontal plane by calculating delays to ears. The new AuralSpace, Aural Space Transfer Function (ASTF), and overlap-add features developed in the present project brought the software system up to the point of flexible sound simulation capability in 3D space. Aural spaces, which simulate environments where sounds might be supposed to originate, were explored as a conceptual

foundation for what it might mean to render a sound "at a location:" an aural space defines what changes will be made to a sound spectrum's amplitudes and phases when it originates from an arbitrary 3D location. Position- and aural-space-dependent adjustments to sound spectra were captured as ASTFs. The AuralSpace abstract class was defined, along with DiscreteAuralSpace, which provides some extensions and tools for working with approximate ASTFs. The ASTF class was also defined, as a means for transporting and saving ASTF data. In order to apply ASTFs to mono sound data efficiently, the overlap-add convolution method was implemented. A computationally intensive benchmark test was devised to optimize the relevant parameters of the overlap-add implementation as discovered by asymptotic analysis.

HRTF data from the 1994 MIT Media Lab KEMAR dummy head data set was used to instantiate an empty 3D aural space, KemarAuralSpace, with a realistic head-angle-dependent frequency response, as outlined in the project proposal. A test program was written using the sound collaging software system to assess how well the system did at creating sounds with apparent 3D location. Several subjects each listened to a few different versions of the randomly generated test material and reported which 3D octant the sound seemed to be coming from. It was found that the system provides excellent left-right discrimination, significant up-down discrimination, and very poor front-back discrimination. It remains to be seen in the future whether an aural space implementation using a modern HRTF data set can provide better front-back discrimination.

Expansions: Interpolation and Reverberation

Going forward, a few new types of aural spaces are clear next steps. When similar sounds are rendered very close to each other, they will sound most realistic if they exhibit beating and other interactions that can't be realized when only the single best-matching HRTF is used; these fine separating angles, while imperceptible in terms of the resolution of aural space localization, are important to resolve in this case. Linear interpolation should be performed for non-exact HRTF alignments between the top three closest HRTFs by averaging them with weights according to their angular distances from the desired point. An interpolating aural space would use a DiscreteAuralSpace's N Nearest Locations method to find these closest HRTFs. As a further expansion, sounds might be imagined to emanate from points inside a room with planar walls, an aural space scenario that would involve reverberation. A sound's energy can find multiple paths to the listener inside a room like this by reflection. The geometric reflections of the points over each planar wall give the locations from which the reflected paths will be heard; this effect can be applied recursively to capture multiple reflections. The HRTF for each reflected location should be added with a geometric decay to the original HRTF before frequency-domain multiplication. A slight random phase may also be added, if it improves realism.