DISSCO: A Unified Approach to Sound Synthesis and Composition

Hans G. Kaper
Mathematics and Computer Science Division
Argonne Nat'l Laboratory
e-mail: kaper@mcs.anl.gov

Sever Tipei
Computer Music Project
School of Music
University of Illinois
e-mail: s-tipei@uiuc.edu

ABSTRACT

DISSCO (Digital Instrument for Sound Synthesis and Composition) represents a unified and comprehensive approach to sound synthesis and composition—unified in the sense that its components share a common formal approach and use similar tools, comprehensive in the sense that they deliver a final product (a musical "event") that does not require further processing.

DISSCO consists of two parts: LASS, a C++ Library for Additive Sound Synthesis, and CMOD, a C++ Composition Module. Release 1.0 of DISSCO is available as open-source software. This article discusses some underlying ideas of music composition and sound synthesis and describes implementation details in the context of LASS and CMOD.

1. INTRODUCTION

DISSCO (Digital Instrument for Sound Synthesis and Composition) represents a unified and comprehensive approach to sound synthesis and composition—unified in the sense that its components share a common formal approach and use similar tools, comprehensive in the sense that they deliver a final product (a musical "event") that does not require further processing. DISSCO is a black box that takes data provided by the user and produces a finished object.

DISSCO consists of two parts: LASS, a C++ Library for Additive Sound Synthesis, and CMOD, a C++ Composition Module. Release 1.0 of DISSCO is available as open-source software [8]. Details of LASS and CMOD are discussed in Sections 2 and 3, respectively; some recent results obtained with DISSCO are described in Section 4; aesthetic considerations are given in Section 5; and future work is described in Section 6.

2. LASS

LASS is a C++ Library for Additive Sound Synthesis. Built around the idea that a score is a collection of sounds and a sound a collection of partials [1], LASS takes advantage of the Standard Template Library (STL) Containers and Iterators. A good number of its features and most of its functionality reflect previous work done with DIASS [4] and DISCO [3]. However, LASS is no longer a MusicN-type program; oscillators and wavetables have been replaced by function evaluations, sounds are no longer produced by "instruments," and there is no score. LASS can generate an XML file as a record of the score, but this file is not needed as input.

The paradigm underlying LASS can be summarized as follows [1]. A piece is a complex wave, the result of superimposing a collection of sounds. Each sound in turn is a superposition of its constituent partials. Partials are the elementary building blocks of a piece; they can be simple sine waves, other wave types, or even white noise. In the case of sine waves, the contribution of an individual partial to the sound is given by an expression of the form

$$p(t) = a(t)\sin(2\pi f(t)t + \phi(t)),$$

where a is the amplitude, f the frequency, and ϕ the phase of the partial. Each of these three parameters can vary with time. LASS computes the samples that describe the resulting complex wave by adding the contributions from all the partials active at the particular instant of time.

2.1. Features and Functionality

From the user's point of view, the central elements of LASS are sounds and partials. Both have static and dynamic attributes (parameters) that are associated with enums. Static parameters are the start time, duration, wave type, and relative (maximum) amplitude of each partial; examples of dynamic parameters are the frequency, phase, amplitude envelope, and amplitudes and rates of vibrato (FM), tremolo (AM), amplitude transients, and frequency transients.

2.1.1. Frequency

Parameter values assigned to a sound, such as start time and duration, are shared automatically by all the sound's partials. In addition, the user has the option of specifying individual values for each partial. For instance, a frequency value (440 Hz) can be specified for the entire sound s,

```
s.setParam(FREQUENCY, 440);
```

and all its partials will acquire integer multiples of the 440 Hz frequency; or an individual partial (for example, the third) can be assigned a specific frequency (1,234 Hz say, instead of 1,320 Hz),

```
s.get(2).setPartialParam(FREQUENCY, 1234);
```

The tuning of partials can be distorted through the FREQUENCY_DEVIATION feature, which modifies each partial's frequency by one-half the distance between itself and that of its nearest neighbor. The result is a nonharmonic tuning of the sound's components through a somewhat random process.

Like many other parameters, frequency is a dynamic variable (a function of time, which can have a constant value). For glissandi or sound bends, the frequencies of all partials in a sound will vary in the same way. On the other hand, one can "detune" a sound whose partials are in a harmonic relationship by applying a different function to each of the constituent partials, thus distorting the ratios of their frequencies. Multiplication by a factor of 1 will leave a frequency unchanged, a factor of 2 will produce a pitch one octave higher, and a factor of 0.5 will lower the pitch by one octave.

2.1.2. Envelopes

Dynamic variables, which are specified by functions of time, are better known to musicians as envelopes. LASS has an Envelope class that handles basic operations such as getting the value of an envelope at a specified time or multiplying two envelopes. An envelope can consist of any number of segments. The ith segment is defined by the coordinates (x_i, y_i) of its starting point in a time-amplitude plane, its type (LINEAR or EXPONENTIAL), and an attribute (FIXED or FLEXIBLE) indicating whether its length is fixed or can be stretched or compressed—a useful attribute, for instance, if a sound is repeated with the same attack and decay but different durations. Here is the specification of the ADSR envelope shown in Fig. 1.

```
ADSR Envelope
5
0.00 0.00 EXPONENTIAL FIXED
0.10 1.00 LINEAR FLEXIBLE
0.20 0.80 LINEAR FLEXIBLE
0.70 0.80 EXPONENTIAL FIXED
1.00 0.00
```

Envelopes are stored with an identifier (id number) in an EnvelopeLibrary. They are usually (but not necessarily) normalized, so scaling may be needed. Envelopes can be created on the fly and, as shown below, applied to any dynamic variable at various time levels (see Section 2.2).

2.1.3. Loudness

LASS pays special attention to the perceived loudness of a sound. The perceived loudness depends on the composition of the sound and is a nonlinear function of the amplitudes of the constituent partials [2]. LASS uses the ISO equal-loudness level contours and a partition of the frequency range into critical bands to adjust the amplitude of each partial so as to correctly contribute to the target loudness of the composite sound.

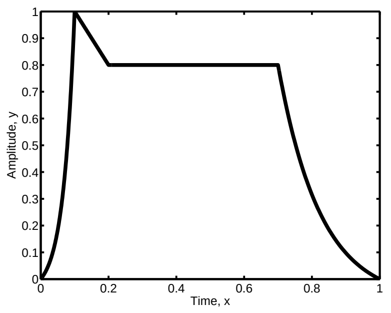

Figure 1. Typical ADSR envelope.

The evolution of a partial's amplitude is controlled by means of an envelope whose peak is scaled according to its relative strength in the spectrum. Since LASS allows for detailed control of each partial through envelopes as complex as needed and for any number of partials, a practically infinite variety of spectra can be created. The user can also obtain crescendo or diminuendo effects over the duration of a sound by taking advantage of the envelope multiplication feature and combining the envelope of each partial with other envelopes that incorporate the desired effect.

2.2. Modifiers

Frequency modulation (vibrato) and amplitude modulation (tremolo) change the main ingredients of the wave at subaudio rates. In LASS, envelopes control evolution of the frequency and amplitude. The user can specify the magnitude (amplitude) of the modulation as a fraction of the basic frequency or amplitude and its rate in hertz. To obtain a constant effect, one uses a one-segment linear envelope with y values 1 at both endpoints.

A similar treatment is given to transients, narrow spikes in the frequency and/or amplitude. In sounds produced by acoustic instruments, transients typically occur at the onset of the vibration, in a fraction of the first second. LASS enables the user to apply frequency and/or amplitude transients over any portion of the sound or over its entire duration, depending on the shape of the envelope used. As in the case of FM and AM, the magnitude of the spike and its rate of occurrence can be specified; in addition, the width of the spike can be controlled, the default value being 1,103 samples at a 44.1 kHz sampling rate, or about 0.025 seconds.

Two more features deal with modification of the sound in an acoustic environment, namely, reverberation and spatialization.

Reverberator is an implementation of the reverberator described in Moore's text [5, Section 4.4]. A simple way to use this feature is to specify only the size of the room by a number between 0 (no reverb) and 1 (max reverb). For more detailed control, one can invoke a second constructor and specify the percentage of reverberated vs. direct sound (a dynamic variable), the ratio between the response of high and low frequencies, the gain of the all-pass filter, and the delay (in seconds) of the first echo response. A third constructor replaces the high/low ratio with low-pass gains and the gains of the comb filter. Reverberator can be applied to the entire score, to selected sounds, or even to individual partials within a sound.

In LASS, sound waves are distributed by Spatializer over a number of tracks, where each track corresponds to a particular output channel. Pan, a simple spatializer, distributes the sound across two channels in a ratio specified by the user. MultiPan implements precisely controlled spatialization over an arbitrary number of speakers in either of two ways. In the first method, one specifies the fraction of the total amplitude that is assigned to each individual speaker and the precise moment of the assignment during its duration. Thus, the user can create “unrealistic” effects such as having all speakers at maximum strength at the same time. The second method assumes that the speakers are arranged in a circular pattern around the listener, and one maps the sound to speakers using polar coordinates.

2.3. Rendering

Once the samples have been computed, the score is rendered in the .au format with Score::render. Before writing the score to a file with AuWriter::write, the user has the option of applying one of several clipping management techniques:

- NONE: no clipping management is used (default);
- CLIP: any sample value outside a specified interval is clipped to the threshold value;
- SCALE: a maximum amplitude a_m is computed for the entire score, and all sample values in the score are scaled by a factor $1/a_m$;
- CHANNEL_SCALE: amplitude scaling is done as in SCALE, but for each channel instead of for the entire score;
- ANTICLIP: a maximum amplitude a_m is computed for the entire score, and all sample values that exceed a specified threshold value a_t are scaled by a factor a_t/a_m;
- CHANNEL_ANTICLIP: amplitude scaling is done as in ANTICLIP, but for each channel instead of for the entire score.

An XML file can be created as a record of the score. This file contains all the sounds and all the partials with their attributes, as well as all the envelopes employed. Unlike the score of MusicN programs, the XML file is not needed to generate the sounds.

2.4. User Access

LASS has a graphic user interface (GUI), written in Java, which generates C++ code. It is intended for use in introductory computer music courses to enable the students to experiment with various changes in the data and experience their effects on the aural qualities of a sound.

The use of a GUI is not recommended for composers, who are generally interested in producing a piece with hundreds or thousands of sounds. (It would also contradict the idea of a comprehensive system!) The sample programs offer a better option to learn the capabilities of LASS. They are short programs that concentrate on generating one or two sounds to which a particular feature can be applied. One can hear their impact on the sound and, at the same time, see clearly which lines of code were involved. A few of the more than 25 sample programs are sample_0.cpp, for creating a basic sound; sample_pan1.cpp, for panning; sample_gliss1.cpp, for creating a glissando; sample_reverb3.cpp, for reverberation; sample_transient1.cpp, for transients; and sample_cresc.cpp, for obtaining a crescendo.

Release 1.0 of LASS is available as open source software on the Web [8], along with documentation produced with Doxygen and a tutorial.

3. CMOD

LASS is a library and does not include a “main” program. To take full advantage of it as a compositional tool, one needs another piece of software to drive it. The informed user can either write the code that best suits a specific goal or use CMOD, the module that accompanies LASS.

CMOD, the Composition MODule, also written in C++, represents a “floating hierarchy” that creates certain objects. The objects are not necessarily produced in sequential temporal order. Similar to LASS, it involves collections whose members are, in turn, themselves collections of objects. CMOD consists of classes dealing with various aspects of a composition and a number of utilities. It has inherited its basic structure from DISCO [3] and takes advantage of LASS features such as the Envelope class.

3.1. The Event Class

The central element of CMOD is the abstract Event class from which other classes are derived. Each event contains an arbitrary number of layers, and each layer contains an arbitrary number of objects of various types. An object could itself be an event containing other objects; an entire piece is viewed as an event, as are each of its sections and collections of simultaneous or sequential sounds (chords or melodies). In addition, one can imagine other events representing various traditional or not-so-traditional elements of form. New objects may be added to the list, and not all events need to be active at all times. Since the number of events and their relationships may differ from one project to another, they are simply called Top, High, Medium, Low, and Bottom. At the end of this chain is the sound, which is no longer an event: beyond the Bottom object, LASS takes over.

All events have at least four attributes: name, start time, duration, and type; all except Bottom also have a density attribute. The name of an event is that of a text file containing the information needed for the realization of the event.

Two methods of the abstract Event class are shared by all derived classes: Build and CreateNewObjects. Build establishes the main characteristics of the object: the number of layers and the number of types in each layer. Since each object (except the ones created by Bottom) is itself a collection, the number of objects contained in this new collection is also determined here. CreateNewObjects is a loop whose upper limit is the number of objects contained in the event. A pass through the loop corresponds to the creation of a new object, whose start time, duration, and type are selected according to a continuous or discrete probability distribution or from a file of admissible values. The discrete values method involves a matrix whose columns are associated with time instants when events may take place (x values) and whose rows are associated with types of events (y values). Row and column entries are further enhanced by envelopes showing instants of time when each type is more likely to occur and by a probability vector reflecting the relative importance of each type. The envelopes and the vector are adjusted after each selection in a feedback mechanism. The matrix, a procedure borrowed from DISCO, is described in some detail in [3]; the other two methods will be discussed below. At the bottom of the loop, SelectNextEvent checks the name and type of the new object and calls its constructor.

3.2. Floating Hierarchies

The way CMOD is set up—classes derived from a template-like Event, objects that are in fact collections of lower level events—suggests a hierarchical order. This suggestion is reinforced by the fact that, unlike LASS, CMOD does have a “main” program that triggers the top event, the piece. This hierarchical structure in CMOD is not stationary. Levels can be skipped, a top event might be a collection of bottom objects, or the highest level of the structure might be only a low-level event; in fact, these arrangements may coexist in the same piece. Moreover, the objects of a collection do not have to be ordered in time. In this way, similar or related objects can be placed at different instances, and connections can be made between events belonging to different hierarchical levels. We believe that this model is a more realistic representation of the way a composer works. The alternative of sweeping the piece from beginning to end may be easier to implement but is certainly less common.

The “main” program acts more like a benign administrator, a facilitator who presides over an orderly process without imposing its own will. Indeed, the “main” program provides only the means (input and output files, envelope library, score, reverberator objects, etc.), starts the process and makes sure that it ends with a product (the sound file).

3.3. Implementation

The Bottom object provides instructions for LASS. It calls Implement, instead of SelectNextEvent, which creates a sound in LASS and sets parameters such as start time and duration,

```
Sound s;            //create sound
s.setParam(START_TIME, stimeSec);
s.setParam(DURATION, durSec);
```

followed by frequency, number of partials, and loudness. The user has a number of options in selecting each of these parameters. For example, the frequency could be selected randomly from a continuum, it could be part of a well tempered scale or an overtone of a low fundamental, it could be part of a sequence (for example, a tone row), or it could be filtered in by a sieve [7, Chapter 9]. Similarly, deciding the number of partials is part of a more involved process, where a spectrum is also created by choosing an envelope for each partial and scaling it according to some preexisting rule (exponential or Fibonacci decay, random, constant, etc.) or according to a rule defined by the user.

The modifiers described for LASS are present in CMOD as well. They are identified by enums, each has an associated probability of occurrence and, depending on whether they are static or dynamic, a value or a scaled envelope. A simple mechanism ensures that related features such as vibrato magnitude and rate are treated as a group. For example, one cannot specify a nonzero rate when the magnitude is zero.

Reverberation and spatialization are treated separately but similarly to the way the modifiers are handled.

3.4. Utilities

In the process of writing CMOD it became clear that, in many instances, the same or similar selection procedures apply at all levels. Such procedures were then implemented as “utilities” (not a class); they can also be found in the classes DataIn and Matrix (discussed above).

Utilities deal with three main areas: random choices and/or choices operated on a continuum, choices involving discrete elements, and envelope making. The first category includes trivial routines that produce random floats and integers within a given range, add small random quantities to a given value, and choose an element from a list by matching a random number with its probability. It also includes Stochos, where two envelopes define a range whose minimum and maximum values vary in time and a third envelope controls the distribution of values within the range. A second Stochos option stacks up a number of probabilities (also varying in time) and selects an item corresponding to one of the areas thus defined.

ValuePick, in the second category, also starts with three functions of time that define a dynamic range and a distribution over that range, but continues by creating a list of discrete values produced by a sieve or some other rationale. The sieve may be weighted so that not all members of the list have the same chance to be selected.

While envelopes can be easily retrieved from a library and scaled, the third category of utilities provides the user with the capability to create new envelopes on the fly and multiply them if needed (see Section 2.1). SegmentBuilder lets the user specify points (x, y) and segment types or select them through one of the methods described above.

Other specialized routines in the utility file are dedicated to various tasks, from partitioning a segment into golden mean ratios to translating density percentages into numbers of sounds per second or traditional note values. A separate class, DataIn, concentrates all read and write operations and the handling of files. This last capability is essential and necessary because each group of objects has a text file attached to its type that contains the data necessary for its realization and these files need to be opened, closed and rewound frequently to various locations.

3.5. Input/Output

The input files are set up in a user-friendly way; almost every line starts with a tag that identifies the operation and an abundant use of enums. They have to follow a general format, but the user has to create files appropriate to the project at hand. Selections can be made in a number of ways: READ, read the information as it is printed; RANGE_DISTRIB, one of the Stochos options; SEQUENCE, take an element of an ordered list. In the last option, the order (marked "offset" in the files) depends on the type of object (TYPE) and its number (OBJNUM, or ZERO in the case of a one-element list). The records are interspersed with lines that are ignored by the computer, which mark the functions that need the subsequent data. Following are two short examples illustrating the assignment of frequency and loudness and of reverberation,

```
             Bottom:AssignFreq
method1  WELL_TEMPERED
method2  SEQUENCE
offset   OBJNUM
step     4 58 64 68 32
             loud-ReadComputeFloat
method   SEQUENCE
sones    4 120.0 160.0 100.0 110.5
```

```
              Bottom::Reverberation
method    ROOM_SIZE
offset    ZERO
              reverb_ReadComputeFloat
method    COMPUTE
              stochos_reverb
method    RANGE_DISTRIB
numEnv    3
panEnvs   29 29 30
panScale  0.00 0.146 1.0
```

The output of CMOD is the input needed by LASS to render a score. For the user's benefit, a text file, particel, is created. It contains information that enables the user to follow the progress. Here is an example of such a progress report:

```
TOP LEVEL:  daria.dat
======================================
:HIGH 1:  H/C5
: StartTime:    241 units     241 sec
: Duration:      19 units      19 sec
--------------------------------------
|BOTTOM 1:  B/sChord05
|  StartTime:      0 units       0 sec
|  Duration:      17 units      17 sec
--------------------------------------
 Sound 1:
   start time 6 duration 7.4  type 10
   13 partials frequency=55 sones=207
--------------------------------------
```

4. RESULTS

LASS has proved to be remarkably resilient and free of problems. A fair amount of work was put in during the past two years, all of it directed toward adding new features or modifying the original ones. It has been used for four semesters in the teaching of Computer Music and Computer-assisted Composition courses at UIUC. It has also been used hands-on by students enrolled in courses dealing with late 20th century music and by nonmusic freshmen attending modules on music and technology in Discovery courses.

A tape piece dARIA was realized with LASS in 2004 and performed at UIUC and at the University of Wisconsin–Milwaukee. Because that version was produced not with CMOD (unavailable at the time) but with a less sophisticated composition system, a new version is being realized with DISSCO.

5. AESTHETIC CONSIDERATIONS

In music composition, tools are often chosen for practical reasons. At a deeper level, however, the choice reflects farther-reaching ideas akin to a worldview. DISSCO suggests the point of view that physical realities could be a model for artistic endeavors, collections of objects reoccurring at different time scales. While trying to be neutral in order to be useful in a variety of contexts and styles, CMOD embraces the concept of floating hierarchies—dear to Herbert Brün: a judicious rejection of systems based on unyielding power but equally distant from anarchy, a desirable social organization.

Scattering events in time in a nonlinear manner might reflect a type of relativism but might also correspond to the workings of memory and free association. It might also resonate with the "Momente" form of Stockhausen and Debussy or with Proust.

Indeterminacy exists in nature at various scales. DISSCO can be used to create fully deterministic works. But it also allows for the introduction of randomness in the composition at many levels and in varying degrees. If one tries to pursue John Cage's objective "to imitate Nature in its mode of operation," chance needs to be part of that project. By the same token, a comprehensive system, as defined at the beginning of this paper, casts the composer in the role of a Demiourgos, giving life to creations that exhibit closure and not to disjunct limbs. When combined with randomness, such a comprehensive approach becomes capable of generating distinct variants of an abstract idea like trees in a forest or people in a crowd—an experiment that leads to similar results under similar conditions, in other words: a manifold composition [6].

6. FUTURE WORK

Both CMOD and LASS are works in progress. They are being used while new developments take place—a situation that we expect will continue for the foreseeable future. The following additions are currently under consideration and may be added to future releases.

For LASS: the capability to use other wave types besides sine waves, the capability to import existing samples as a collage, and the capability to spatialize the sound when the speakers are not arranged in a circular pattern.

For CMOD: an expanded matrix to include more than two dimensions, intervallic relationships when choosing a new element, and functions appropriate for the notation of acoustic instruments.

DISSCO is part of a larger project that includes automatic printing, generating visual objects, and sonification of complex scientific data. The first two have already been included in the Bottom object of CMOD as possible paths (not realized at this time). Sonification requires either a counterpart for CMOD or software to create a substitute for the files described in Section 3.5.

7. REFERENCES

[1] Hans G. Kaper and Sever Tipei. Formalizing the concept of sound. In *Proc. 1999 Int'l Computer Music Conference, Beijing, China*, pages 387–390, 1999.

[2] Hans G. Kaper and Sever Tipei. Loudness scaling in a digital synthesis library. In *Proc. 2004 Int'l Computer Music Conference, Miami, Florida*, pages 398–401, 2004.

[3] Hans G. Kaper, Sever Tipei, and Jeff M. Wright. Disco: An object-oriented system for music composition and sound design. In *Proc. 2000 Int'l Computer Music Conference, Berlin, Germany*, pages 340–343, 2000.

[4] Kristopher Kriese and Sever Tipei. A compositional approach to additive synthesis on supercomputers. In *Proc. 1992 Int'l Computer Music Conference, San Jose, California*, pages 394–395, 1992.

[5] F. Richard Moore. *Computer Music*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[6] Sever Tipei. Manifold compositions a (super)-computer-assisted composition experiment in progress. In *Proc. 1989 Int'l Computer Music Conference, Columbus, Ohio*, pages 324–327, 1989.

[7] Iannis Xenakis. *Formalized Music*. Pendragon Press, Stuyvesant, New York, 1992.

[8] http://dissco.sourceforge.net.

Acknowledgments

The authors acknowledge the contributions from Braden Kowitz, who designed and implemented the basic features of LASS, and the students in the "Advanced Computer Music" seminar taught at the University of Illinois between 2002 and the present, who contributed to the further development of LASS. The work of H.K. was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.