DISSCO

DISSCO, a Digital Instrument for Sound Synthesis and Composition, provides a unified approach to composition and sound synthesis bringing them together into a seamless process. It was developed at the Computer Music Project of the University of Illinois (CMP), Argonne National Laboratory (ANL), National Center for Supercomputing Applications (NCSA) and San Diego Supercomputer Center (SDSC). DISSCO is a reliable tool in continuous transformation that has been extensively tested as a medium for the creation of original music performed at international festivals and conferences and as an education tool in classrooms at the University of Illinois.

DISSCO STRUCTURE

DISSCO is written in C++ and has three main modules: LASS, a library for additive synthesis (appendix A); CMOD, a composition module; and LASSIE, a graphic user interface through which users enter data. The interface generates a XML file which is read by CMOD where the data is processed; the results are sent to LASS where sounds are synthesized and gathered in an audio file. An alternative path uses the CMOD's Note class to generate a text file containing instructions for LilyPond, open source printing software, in order to create a score using traditional Western notation. A module for sonification experiments and a module for the production of visual events are under consideration for future development.

CMOD Events

The backbone of the Composition Module is the implementation of an acyclic directed graph (appendix B), a rooted tree whose vertices or nodes represent "Events" at different structural levels of a composition. There is only one Top event - the entire piece - and an arbitrary number of High, Medium, Low and Bottom events placed on vertices or nodes further and further away form the root. They could be regarded as sections (High), subsections (Medium), phrases (Low), chords (Bottom) or melodic sequences (Bottom) and they represent the formal construction of the composition. Actual sounds are created only inside Bottom events[1].

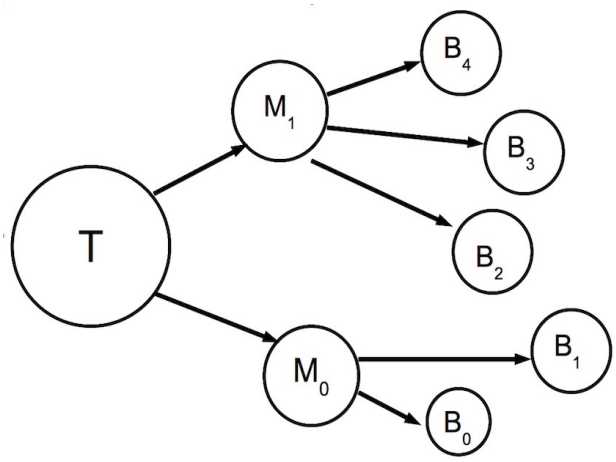


Figure 1. Only Top, Medium and Bottom Events are shown here for he sake of clarity

A *parent-children* relationship operates between structural levels and a comparison could be established with a set of matryoshka, the Russian dolls nested inside each other. Not all types of events (vertices/nodes) need to be present, new ones can be added through minimal coding effort, and the hierarchy of events is flexible: a Low event may have a Medium event as a parent while, at the same time, it can have High or Medium events as its own children.

1 Analogous with the sections of a pop tune: intro, verse, chorus or refrain, coda and their components such as melodies with their phrases, chords, etc. Or, in traditional classical styles, the Exposition, Development and Recapitulation of a Sonata form with First Theme, Transition, Second Theme, Conclusion and their own phrases, etc.

All events are assigned a start time, a duration and a type. Bottom events require extra information for sound definition which comprise frequency, amplitude, vibrato (FM), tremolo (AM), transients of frequency and/or amplitude, filters as well as the position of sound sources in space (spatialization) and reverberation. With the exception of frequency and amplitude, all other items can be allotted by any parent to its children (not recommended); in turn, a child can override the parent's contribution.

Each sound is linked to a spectrum object - comprising an arbitrary number of partials or overtones - in which a *spectral envelope* is created when each partial receives a relative strength value (amplitude) and an envelope that traces its evolution in time.

The Note class (under development) creates a score using Western traditional notation with the help of LilyPond, open source software.

Project Properties. Information necessary to initiate the project is provided here and includes:

- Project title
- Flags identifying various structural levels and the type of output (THMLBsnv, not to be changed)
- Number of channels (default 2)
- Sampling rate (default 44,100)
- Sample size (default 24)
- Number of threads (default 1)
- Sound synthesis, Score printing or both options
- Output to Particel, a partial record of the run
- Name of Top event (default 0)
- Total duration of the piece in seconds

Building children events. There are three methods through which a parent event can spawn children: *Continuum*, *Sweep* and *Discrete*. They all determine the start time, type and duration of each child event and makes sure it rests within the confines of the parent's time span.

- *Continuum* assigns the start time of individual children/events either in orderly fashion through a deterministic operation such a specific list, a sieve (see Utilities Functions), etc. or by scattering them with the help of random functions within the confines of parent's duration. It accepts both integer and floating point values for start times and durations but only integers for the types of children events.
- *Sweep* ensures that no sound begins before the preceding one has ended thus ensuring the creation of continuous melody-like sequences. Same as Continuum, it accepts both integer and floating point values for start times and durations but only integers for the types of children events.
- *Discrete* builds a three-dimensional matrix (start time * type * duration) and ensures that children with incompatible type (see below) do not collide. This method admits only integer values and uses **Sieve**.

Time units. There are also three time units that can be used: *seconds*, *Elementary Displacement Units* (EDUs) and *percentages* of the parent's duration.

EDUs form a grid of equally distanced points from which relevant ones can be selected. They are necessary when producing an output which uses Western traditional rhythmic notation or pitch scales. In the time domain, one EDU could equal 1/LCM, the least common multiple of all subdivisions of the beat that are employed. Iannis Xenakis first introduced the concept by applying the Peano axiomatic to pitches (see *Formalized Music*, p. 194) of Equally Tempered systems (Appendix C).

Types and layers. Types represent categories of sounds that have a common makeup or design such as the same or similar *spectral envelopes* or similar instrumental/vocal sound production technique. Discrete, for example, prohibits that types which designate mutually exclusive sound production techniques - such as bowing a string and plucking it or using two different mutes for brass - be employed at the same time.

Layers designate different streams of distinct sound sources corresponding to "voices" or instrumental "parts" in a traditional composition. While two or more streams/layers may occur at the same time, when using *Discrete*, incompatible types within the same layer can not.

Number of children. There are three options:

- Density, is applied globally and maps a density of 0 - 1 onto a number of sounds per second (Appendix Ma).
  - Density, a float between 0 and 1

    - Area, how many regions are considered, an integer, default 8.
    - Under One, regions of less than one sound per second, an integer, default 4.
- Fixed: the number of children a parent can create can be specified by a single integer or determined randomly
- By Layer, allows the number of children to be determined for each layer independently (Appendix Mb).

Tempo. A choice of note values with or without dots is given. Then there are two ways of specifying the tempo of an event:

- as a note value in relation to a metronome stop: a note value of a quarter at MM=60 will last 1/60 min. and there will be 60 quarter notes in a minute, each lasting 1 second (default values).
- as beats per second: number of beats each having a selected note value lasting a user specified number of seconds. (A beat might not have the same duration as the metronome note value).

The number of EDUs per beat can also be specified if EDUs are used as a time unit (default value: 6) .

Time signature. Any fraction whose denominator is a power of 2 is allowed (2/4, 7/8, 5/16, etc).
Default value: 4/4.

Common features. All three methods of building children, *Continuum*, *Sweep*, and *Discrete*, require the same information which can be entered as a single number, choosing from a list with the help of a deterministic or random Utility function, or randomly assigning values within a given range:

- Child Start Time, either integers or floats if the time unit is *seconds*, floats for *percentages*, integers if *EDU*s are used.
- Child Type, integers (first type is always 0). The actual names of children are entered in a separate window, following these entries (see below).
- Child Duration, integers or floats depending on the time unit
- Max Child duration, same. Truncates any duration greater than this limit.

- Child Type window. The names of the children are dragged from the **Objects List** on the left side of the Main Window.
    - Number of children in this layer, active only if *By Layer* were chosen, an integer
    - Child Type, the identifier integer from above list
    - Class of the parent event (Top, High, Medium, Low, or Bottom)
    - Name of the child

If *Discrete* is used, the window has more entries:

    - Weight, percentage of this type out of the total number of children
    - Attack Envelope, distribution of child's start times during the parent's time span
    - A. Env. Scaler, scaling factor for the envelope
    - Duration Envelope, distribution of this child duration values during the parent's time span
    - D. Env. Scaler, scaling factor for the envelope

NOTE: When specifying children attributes in layers, click outside the line, in the window, after each entry to secure the entry.

Extra features. Spatialization, Reverberation, Filters and Modifiers (see Bottom Event) can be assigned by a parent to its children in order to insure a uniform behavior during its duration. It is strongly recommended though that these entries be left blank and completed only for Bottom events.

Bottom Event

Bottom class inherits from the generic Event class but also includes more information needed for sound synthesis and/or notation. A Bottom Event can create synthesized sounds in which case its name should start with "s", or notated sounds in a traditional music score in which case its name should start with "n". The same Bottom Event can not create both synthesized and notated sounds.

Frequency. There are three alternative ways in which frequency can be specified:

- *Equal Temperament* uses a pitch representation similar to MIDI practice with the difference that here, $C_4 = 48$ (not 60) i.e. 4 * 12, the number of semitones from $C_0 = 0$; it takes only integers. It is possible to modify the number of equal intervals in one octave - see Advance features.
- *Fundamental* requires
  - a fundamental frequency in Hertz (a float, it can not be 0).
  - a partial/overtone number, an integer.
    NOTE: higher partials usually will have a frequency different from that of corresponding Equal Temperament pitches.
- *Continuum* has two options:
  - Hertz, a float
  - power of 2, a float. Any exponent of 2 within the range MINFREQ - CEILING (or 20Hz-15,000Hz) when C_0 corresponds to 2^0. Then, $C_0 = 16.35159783$ Hz and $C_0 * 2^n$ will give the desired frequency.

NOTE: The CEILING of 15,000 Hz corresponds to ~Bb_9 pitch and it was set due to ISO tables (see also Loudness).

Loudness. The subjective *perception* of sound pressure uses *sone* units as a close approximation of the dynamic marks used by musicians (pp, mf, ff, etc.) on a scale of float values, powers of 2, ranging between $2^0 = 1$ (silence) and $2^8 = 256$ (max. value). The computations involve a reversal of Fletcher-Munson curves (Appendix D) and use ISO tables with a max. frequency value of 15,000 Hz (CEILING) .

Modifiers. As shown in the Appendix A expression, all three elements, amplitude, frequency and phase can change in time. Modifiers are applied only to amplitude and frequency since, at the present time, phase is not handled in DISSCO (to be included in the future). They operate on:

- *Frequency* is modified in the following ways:
  - Glissando applies an envelope (see Envelopes) to the requested frequency resulting in a sliding of the pitch. The original frequency is multiplied with the value of every point on the envelope; for example, a value of 1 on the envelope maintains the original frequency, a value of 2 increases it by one octave and a value of 0.5 decreases it by an octave.
  - Vibrato (see Appendix E) produces an oscillation around the requested frequency. Three inputs are necessary for this modifier:
    - probability, a global component, determines the percentage of sounds created in this particular Bottom object that will be affected - a probability of 1 will apply the modifier to all sounds, a 0.5 probability will apply it to half of them, etc.
    - amplitude determines the size of the distortion (see FM, the I factor). A "normal" vibrato requires a very small value (eg. 0.01).
    - rate determines the rate of change of the distortion measured in Hz. A "normal" vibrato has a rate of between 6 and 8 Hz.; small values create a "siren" effect.
  - Detune amplitude envelope ?? need different input fields in LASSIE
    - detune spread value is a percent that corresponds to a range in which a partial will (randomly) fall at the max detuning portion of the sound.
    - detune direction: divergence -1, convergence +1
    - detune velocity: how soon (-1.0 nearly instantaneous) or late (+1.0 nearly instantaneous) the transition will occur.
    - detune fundamental: positive values = tune/detune the fundamental; negative values = do not tune/detune the fundamental
- *Amplitude*
  - Tremolo requires the same types of inputs as frequency and act in a similar way:
    - probability
    - amplitude

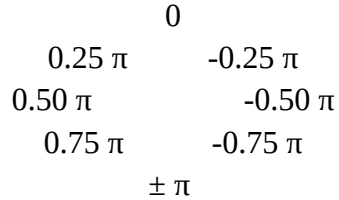
        - rate
- *Transients of amplitude and frequency* (Appendix F)
    - Both require width, an additional input.
        - probability, DISSCO allows transients to be applied during the entire duration of a sound, unlike the transients produced by acoustic instruments. An envelope can restrict the transients to smaller regions.
        - amplitude controls the maximum size of spikes and it is multiplied with a random percentage.
        - rate the value of the rate envelope is the percentage chance of a transient occurring at that particular time
        - width represents the span of a spike; default is 1130 samples or 0.025 seconds
- *Wave type*, an integer determines if sine waves (0) or white noise (1) are used in building the sounds (default 0). Filters maybe used to select frequency bands for the white noise.

NOTE: except for the wave type, all modifiers require envelopes as inputs.

- *For each modifie*r two other features are present
    - Group name makes three lists/maps
        - modifiers with no dependencies
        - modifiers grouped together
        - modifiers with direct dependencies on other modifiers.
    - Apply how, a choice between
        - Sound the modifier is applied to the entire sound, to all partials uniformly
        - Partials the modifier is applied selectively to individual partials

Spatialization. There are three options of setting up the location of sound sources:

- *SPA* contains a menu through which the user defines the location; it has three options:
    - stereo specifies how much of the sound will be projected through the left speaker: 1 will play the sound only from the left speaker, 0 from the right speaker, 0.5 will create the illusion that the sound is located half way between the two speakers. An envelope is required and could move the sound while it is played.
    - polar uses polar coordinates, assumes an arbitrary number of channels arranged in a circle on a plane and needs envelope values for
        - Theta the angle measured clockwise 0 (straight ahead) to ± π (behind) going through negative values on the right side and positive values on the left side.



        - Radius the distance from the center of the circle
    - Multi Pan may be used for any number of speakers ; the user has to manually specify the percentage of the sound placed in each channel. Envelopes are required for entering data.
- *ReadSPAFile* reads the name of an already existing file create by SPA and stored in the Spatialzation folder
- *SELECT* lets the user choose between existing files in the Spatialization folder (see Utilities Functions).

Reverberation. The reverberator was built following the model found in F. Richard Moore's book *Elements of Computer Music, i*n turn, based on Schroder and Moore work. Three levels are offered:

- *Simple* asks the user to set the size of the room (0 sound source is in very close proximity, 1 sounds like a big cathedral); all the other parameters have default values. Takes a float or an envelope. Quite effective, recommended for starting users.
- *Medium* allows for setting
    - Reverb Percentage: the amount of reverberated vs. direct sound, an envelope.
    - High/low Spread: ratio between the decay of high frequencies vs. the decay of low frequencies
    - Gain All Pass: default value is 0.7

    - Delay: he length in seconds of the first echo response.
- *Advanced* gives access to all features of the reverberator, for details see Appendix G
    - ReverbPercent, an envelope
    - CombGainList, six floats (default: 0.46, 0.48, 0.50, 052, 0.53, 0.55)
    - LpGainList, six floats (default: 0.05, 0.06, 0.07, 0.05, 0.04, 0.02)
    - GainAllPass (default: 0.07)
    - Delay, seconds

NOTE: Medium and Advanced require envelopes

Filters. This is a BiQuadFilter built by Mert Bay; there are three options available

- *MAKFILTER:*
    - Low Pass
    - High Pass
    - Band Pass
    - Notch
    - Peaking Band EQ
    - Low Shelf
    - High Shelf

All filters require

    - Cutoff Frequency (Hz)
    - Bandwidth (number of octaves)

Only Peaking Band EQ and Shelving filter require

    - db Gain - Filters gain (dB) for peaking and shelving filters only

- *READFILE*: reads an existing file
- *SELECT*: a means to make a selection between existing files (see Utilities Functions)

Spectrum

Defines the internal structure of a sound. It may contain an arbitrary number of partials each of them having its amplitude controlled by an envelope. Spectra represent the types of children/sounds generated by the Bottom event (Appendix H).

- **Number of partials**: integer, set to 1, more can be added (see Add Partials).
- **Deviation**, a float between o and 1: if set to 0, the frequencies of partials will be integer multiples of the fundamental frequency (partial 1); if set to 1, frequencies of all partials 2 and higher will be deviated randomly between ½ of the distance to the upper and ½ of the distance to the lower neighbor.
- **Generate Spectrum**, *still an experimental feature*, it creates the illusion of distance by manipulating the scaling of each envelope according to register (octave) and its ratio (percentage) to the strongest partial in a particular sound. Presently, only a Bb clarinet spectrum is considered.
- **Add Partials** allows the user to add an arbitrary number of partials by specifying and envelope and a scaling factor.

Note

Provides means to notate and print sounds created in the Bottom events. It is the equivalent of Spectrum and fulfills a similar role.

- creates a text file, .ly, with instructions for LilyPond, open source software, to print the score.
- the name of a notated sound has to start with “n” (as opposed to “s” for synthesized sounds).
- only EDUs may be used as time units but the parent’s time units could also be seconds or percentages.
- notates correctly any type of tuplet even when start times and duration are the result of random operations
- contains a list of possible modifiers that corresponds to symbols available in LilyPond.

- Configure Note Modifiers (in Project, upper left corner of main window) allows the user to check/uncheck and add/remove modifiers, Note attributes

Note is under development; presently, it can handle only one line at a time.

Envelopes

Envelopes control the distribution (in time) of values (Appendix I). The are four options available:

- **MakeEnvelope**. User can build a custom envelope by specifying *x* and *y* points on the envelope. Useful when entering a range of values for a random distribution instead of a precisely determined value.
- **ReadENVFile**, accesses an existing envelope file.
- **EnvLib**, a library of envelopes is the most used option; it gives the user access to the library by choosing an envelope number and a scaling factor.
- **Select** allows a choice between existing envelope files (see Utilities Functions).

Envelope library window lets the user draw -using the mouse- envelopes that will be stored in the Envelope Library. It can be found on the lower left corner of LASSIE display.

CMOD UTILITIES

The design of DISSCO is based on the proposition that similar tools can be used to select parameter values for various event types, at different time scales. These are general devices that can be applied both when determining structural elements of a piece or minute details of a sound. Some utilities help assign precisely determined values, integers in most cases, while others deal with randomness and probability distributions.

The Utilities object is designed to evaluate the XML strings to their proper format, whether an Event, a number, an Envelope or other objects. It also serves as the interface between the CMOD Event class and the LASS Score class. CMOD Events add the Sound objects they produce to LASS Score through Utilities.

Functions

Functions are means through which the user can enter variable values and make selections. They are a fundamental feature of the Composition Module (CMOD) and are handled through a Graphic User Interface (LASSIE). They are available as options for most of the choices available to the user. There is possible to nest almost any number of functions thus adding more sophisticated levels of control.

- **Select**. The user creates a list of nodes (values) and specifies a choosing method
  - Values, may be calculated randomly using a (nested) function or entered manually.
  - Choice Index, specifies how the values will be chosen, a random value or a *Static Function*
- **Random** generates a float given a range of possible values (flat distribution)
  - Lower Bound, a float
  - Higher Bound, a float
- **RandomInt** generates an integer given a range of possible values (flat distribution).
  - Lower Bound, an integer
  - Higher Bound, an integer
- **Randomizer** takes in a base value and a standard deviation float between 0-1 (inclusive) and returns a float that is within the range of the base value plus or minus the base value times the standard deviation.
  - Base, a float
  - Deviation, a float
- **RandomOrderInt** selects a value from a list and deletes it from that list making it unavailable for the next choice
- **Random Density** controls the density of events through an envelope. Can be used to manage the start times of events unevenly scattered over the duration of their parent. An even distribution may be obtained by using a (*x*=0, *y*=0) to (*x*=1, *y*=1) envelope or its inverse.
  - Envelope Id, choose an envelope number from EnvLib
  - Start Time
  - End Time
- **Stochos** has two main options
  - Functions, a set of stacked envelopes defining simultaneous areas. Suggested by the orchestration chart for Xenakis' *ST/10-1* (see *Formalized Music, p. 139*), selects the participating envelopes in order. The user has to make sure that the sum of all envelope values at any moment = 1.

    Add code to insure Σ = 1 ? Also, look at Density by layer

Can be used with only one envelope in the stack when access to an envelope is not available.
    - Offset option is blocked, not needed here.
    - Value, index of the scaled envelope to be used.
  - Range Distribution takes groups of three functions and uses them to randomly select values within a range that is dynamic, and an offset.
    - Offset, points to one of the groups of three envelopes, an integer. If a parent has more than one children, a group of three envelopes may be applied to a distinct child.
    - Min, lower bound of the range, a scaled envelope
    - Max, upper bound of the range, a scaled envelope
    - Distribution, a (stochastic) distribution function between the two limits, defines the probability of the values inside the range between Min and Max, a scaled envelope.
- **Markov Model Library** (see Appendix J). Can be found in the lower left corner of the screen.
  - Create New Model by right clicking on the leftmost panel - integer numbers will be assigned
  - Markov chain size - the number of possible states.
    - Set
  - Initial Distribution - initial state of the Markov chain, floats, need to add up to 1.
  - Values for states - integers
  - Transition Matrix – integers or floats, integers will be converted to floats, numbers do not have to add up to 1.
- **ValuePick**, establishes a dynamic range and a distribution within that range; applies a *Sieve on* elements within this range and assigns them weights. Similar to *Stochos* but only for integer values; a wrapper for *Sieve* (Appendix K*)*.
  - Absolute Range, the highest value allowed, a double.**OD**
    - Meaningful
      - Values, selets randomly from a list of integers within the range.
      - Offset, one for each element (even if they are the same), integers.
    - Mods, uses a sieve built via **ModParser** class or via **MakeSieve**. The low envelope usually is 1, scale to 0 and the High envelope usually is 1 scale to 1 (bsolute Range * 1).
      - Values, moduli numbers for the sieve, a Boolean expression containing integers.
      - Offset, remainder numbers, one for each modulus, integers.
    - Fake, selects randomly all integers included in the range as many values as children to be built. The Values and Offsets lines remain blank.
    - Fibonacci, uses number from the Fibonacci series NOT available
      - Value
      - Offset
  - Weights
    - Periodic assigns weights to the moduli of the sieve by cycling through the given ordered list of weights.
      - Values, integers, to be converted to percentages of the highest value
    - Hierarchic assigns weights to the moduli of the sieve by adding for each of them probabilities associated with each equivalence class.
      - Values, integers, one for each mod, to be converted to percentages of the highest value.
    - Include, a list of integers associated with a list of moduli/elements, used frequently as a companion to the Meaningful function.
      - Values, integers, one for each element, to be converted to percentages of the highest value.
  - Type
    - Variable uses the *PreferedValueDistribution* method from the *Random* class (Appendix L).
    - Constant uses the probabilities offered by the third envelope.
- **Sieve** operators:

  ~ complement (non-)
  I intersection
  U union
  - difference
- **Sieve Folder**, stores sieves
  - Sieve Builder
- **MakeSieve**

- Sieve Bilder, builds a sieve, similar to **ValueCheck**
  - MakeSieve
    - Low Bound
    - High Bound
    - Elements
      - Meaningful
      - Mods
      - Fake
      - Fibonacci
    - Offset
    - Weights
      - Periodic
      - Hierarchic
      - Include
  - ReadSieveFile
  - Select

- **GetPattern** (see also **Folder Pattern**). Used to create a local PAT file. A pattern of values is generated using a user-defined origin and a sequence of intervals; it delivers one value at a time. Same input may be used to generate various related patterns produced through mod m equivalences,
  [symmetrical set forms (P, R, I, RI), and distortions not available yet.]
  - Method of delivering the elements of the pattern
    - In Order, selects the elements of the pattern in sequential order
  - Origin, the first element of the pattern, and integer
  - Pattern to choose from includes four choices:
    - MakePattern
      - Intervals, discrete values representing the distance between adjacent elements, use **MakeList** or **RawList**, integers.
    - ExpandPattern
      - Method, a choice between ways of modifying the pattern
        - Equivalence finds all intervals within the given range that are mod. *m* equivalent with the members of the initial list. It also assigns probabilities according to the distance to the initial values.
          - Modulo interval by which each member of the pattern is incremented
          - Low, lower limit
          - High, upper limit
          - Pattern, the elements of the original pattern
    - ReadPATFile, reads a file
      - PAT File Name
    - Select, uses the Select function to choose between PAT files.

- **Folder Pattern** (see also **GetPattern**). Used to create a pattern in the Pattern folder that can be used in multiple situations.
  - PatternBuilder, chooses between the methods listed under Pattern to choose from.
    - MakePattern: creates a new pattern.
      - Intervals: discrete values representing the distance between adjacent elements, **MakeList** or **RawList**, integers
    - ExpandPattern: choice of methods of modifying the pattern
      - Equivalence: finds all intervals allowed within a given range which are equivalent mod *m* with the members of the initial list. It assigns probabilities according to the distance to the initial values.
        - Modulo: interval by which each member of the pattern is incremented.
        - Low: lower limit.
        - High: upper limit.

        - Pattern: the elements of the original pattern.
    - ReadPATFile: reads a sequence of values from an existing file.
        - PAT File Name, reads a file
    - Select, uses Select function to choose between PAT files

NOTE. Patterns of stimes (attacks) will work _only_ with the SWEEP option

- **MakeList/RawList,** **(**presently**,** only RawList is available when called from MakePattern.)

- **ChooseL**. Part of Utilities, used as a method in the Sieve class to choose an element of the sieve according to its probability. Can be called in other situations but only in connection with a sieve.
    - Sieve to choose from

Static Functions

Help determining the order on given lists.

- **CURRENT_TYPE**, Result String
- **CURRENT_CHILD_NUM**, Result String
- **CURRENT_PARTIAL_NUM**, Result String
- **CURRENT_SEGMENT** Not implemented yet
- **AVAILABLE_EDU**, Result String Not currently used; used in previous versions with FileValue

- **PREVIOUS_CHILD_DURATION**, used by Sweep in determining start times.
- **CURRENT_LAYER**, Result String Not currently used; used in previous versions with FileValue

Matrix

A 3 D matrix, start time * type * duration, used to avoid collisions between sounds of the same type.

- main tool in the *Discrete* method of creating children
    - it can be called from *Continuum* or *Sweep* as well.
- assigns each children's locations through random procedures
    - not in sequential order
- if unable to accommodate the number of children requested, the procedure is repeated up to 10 times
    - if still unsuccessful, the user is asked, at execution time, to lower the number of children.

Folders

List found on the left side of the Main window in LASSIE. They contain a list of objects that can be created and stored for future use.

- Top, High, Medium, Low, Bottom Events
- Spectrum
- Note
- Envelope
- Sieve
- Spatialization
- Pattern
- Reverberation
- Patterns
- Filter

Aesthetics.

DISSCO is primarily a software for **composition** which is connected to a sophisticated sound design apparatus.

Although DISSCO can be used to generate music in any desired style, it exhibits a bias towards the use of controlled randomness and encourages the user to plan the composition ahead of time and in detail (pre-composition work). The user can also start by designing sounds and building small sections which are connected later but a blueprint that is

allowed to change and grow, can save time and helps to focus the efforts. A completely deterministic work that contains only uniquely defined elements can be constructed with DISSCO but, when randomness is present, the composition may take the form of an experiment whose outcome can not be always fully predicted.

DISSCO is a “black box” or a *comprehensive* program: once the data is fed in, the user does not intervene during the computations, and the output does not require post-processing. Particularly, when *controlled randomness* is present, the role of the user is elevated to that of the author of a family of outcomes.

DISSCO has a hierarchical structure - the parent-child relations between Top-High-Medium-Low-Bottom events - that mirrors the structure of traditional Western music (as well as that of other music cultures) but it can also accommodate situations similar to John Cage’s chance compositions where formal schemes are avoided, by having, for example, the Top event (the entire piece) engender directly one or more Bottom events (where the actual sounds are designed.

Composing comprises making choices and, as it was mentioned before (Utilities), same or similar tools are used at different time scales to facilitate the selection process. This contributes to unify the entire process and illustrates the view of the musical space as homogeneous.

The original impetus for building DISSCO was the desire to have a mean of creating m*anifold compositions*. A *manifold composition* is a class of compositions that comprises all actual and potential variants of a musical work generated by a computer that

- runs a program containing elements of indeterminacy and
- reads essentially the same data for each variant.
-

A unlimited number of works belonging to the same equivalence class can be produced in this way. The members of a manifold composition are variants of the same piece; they share the same structure, pitch, rhythmic materials, amplitudes, spectra, etc. and are the result of the same process, but differ in the way specific events and diverse sound characteristics are distributed in time. Like faces in a crowd, they all have common basic features but exhibit particular attributes due to the fact that different seeds for the random number generator result in different outputs.

DISSCO history.

A first version, the Digital Instrument for Sound Synthesis (DIASS) was created jointly at CMP and ANL in the early and mid 1990s by Sever Tipei and mathematician Hans G. Kaper. A *MusicN* type of program based on *Music4C* (M4C), developed at CMP by James W. Beauchamp, it was used both as a sonification tool and in conjunction with other computer-assisted composition software; results were displayed in an immersive virtual environment (CAVE) using M4CAVE, also developed at CMP and ANL, and presented at SuperComputing and International Computer Music Conference (ICMC).

In 1999, Kaper and Tipei presented at the ICMC, in Beijing, China, the paper *Formalizing the Concept of Sound*, the theoretical basis for *Digital Instrument for Sonification and Composition* (DISCO) and, later for DISSCO. An initial version of the *Composition Module* (CMOD), written by Sever Tipei in 2000 and modified by students of the “Advanced Computer Music” seminar, and greatly improved later by Ryan Cavis, Andrew Burnson and Ming-Ching Chiu.

Hans Kaper worked on the *Implementation of Loudness* (presented at the 2001 Acoustical Society of America) that allows the user to specify the desired percieved loudness instead of amplitude. It was incorporated in 2004, *Music Object-oriented Sound Synthesis* (MOSS) program written by Braden Kowitzt, the first version of the *Library for Additive Sound Synthesis* (LASS).

In 2012-2013, CMOD underwent intensive refactoring and improving at CMP. The improvements include:

- integrating LASS in a tighter manner.
- adopting XML as its input format.
- including **muparser** as its math parser.

as well as the attachment of a graphic user interface (LASSIE) designed by Ming-Ching Chiu.

Since then, more features were added to DISSCO through the work of NCSA program *Students Pushing Innovation* (SPIN) and to the work performed by Paul F. Rodriguez at SDSC, under the *Extreme Science and Engineering Discovery Environment* (XSEDE) program, where a collaborative and interactive version using a Jupyter notebook was added in 2020.

DISSCO availability.

DISSCO is available on **GitHub** and presently runs only on computers using the LINUX operating system. All three components (LASS, CMOD, and LASSIE) are written in C++; Xerces, a validating XML parser, and muparser, a math expression parser library, are used to read and interpret CMOD input XML files.

In order to compile DISSCO, muParser.h, sndfile.h and premake4 are needed besides Xerces. LilyPond is necessary for printing scores for voices or acoustic instruments.

APPENDICES

A. Additive Synthesis is a technique that creates timbre by adding sine waves. The timbre of musical instruments can be considered in the light of *Fourie*r theory to consist of multiple partials or overtones. Each partial is a sine wave of different frequency and amplitude that swells and decays over time. Additive synthesis most directly generates sounds by adding the output of multiple sine waves. The basic expression used by DISSCO for one wave/partial is

$$w = a_t * \sin(2\pi f_t * t + \varphi_t)$$

where a denotes the amplitude, f the frequency and φ the phase (See also *Formalizing the Concept of Sound*, at http://cmp.music.illinois.edu/people/tipei/Papers/papers.html)

B. Directed acyclic graph (DAG). A graph consists of a series of vertexes or nodes connected by edges. A directed acyclic graph (DAG) is a graph that is directed and without cycles connecting the other edges. This means that it is impossible to traverse the entire graph starting at one edge. The edges of the directed graph only go one way.

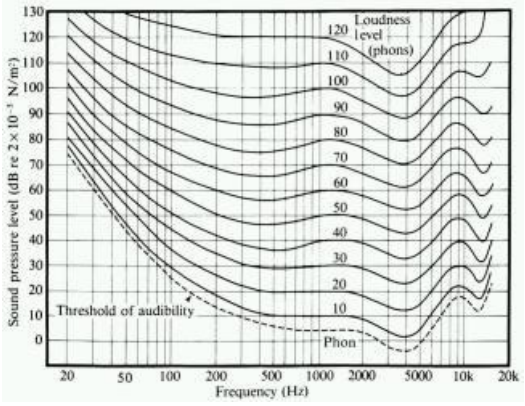
C. Equal temperament. A tuning system that divides the octave (the distance between a fundamental and its first overtone/partial) into equal segments. Present day keyboards use the Twelve-tone Equal Temperament (TET) with the octave divided into 12 semitones (half steps). To calculate the frequency ***s*** of a pitch situated ***n*** semitones away from a given frequency $\underline{s_0}$:

$$s = s_0 * 2^{n/12}$$

To calculate the size of smaller - such as quarter-tones - or larger intervals, substitute 12 with the number of desired divisions of the octave. Conversely, to find out the frequency of a given pitch:

$$n = 12 * \log_2 s/s_0$$

D. Amplitude and loudness. Loudness is a subjective perception of amplitude. In the 1930s, Fletcher and Munson measured the human ears frequency response to a range of levels (amplitudes) after testing many subjects. The average curves obtained show that the human ear is more sensitive to middle range frequencies corresponding roughly to the fundamental and first overtones of the human voice (speech).



These curves are derived empirically and there is no mathematically way of defining them. In DISSCO, the user is asked to specify the desired loudness of sounds and not their amplitude; internally, the curves are reversed and combined with other psychoacoustic consideration such as the *critical band* and *masking* that partially explain why 20 violins do not sound 20 times louder than one violin (consult an Acoustics book for more details).

E. Frequency modulation (FM) and amplitude modulation (AM) are techniques also used in broadcasting: FM and AM radio stations. FM corresponds to what musicians call vibrato and AM to tremolo. For example, the small movement of a string player's finger on a string shortening and increasing its length by a certain amount - and thus modifying the pitch - creates a vibrato effect while the bow reversing its direction - and modifying the amplitude - creates a tremolo effect. A given frequency (carrier) is modified by a modulator; larger input values create side bands (see also Bessel functions).

The FM synthesis technique, first developed by John Chowning at Stanford and available first on Yamaha and, by now on most synthesizers, exploits this feature in sophisticated ways in order to create instrumental and vocal sounds.

Similarly, Amplitude modulation also uses a carrier (the amplitude corresponding to the specified loudness) and a carrier.

F. Transients of frequency and amplitude are brief spikes that appear during a few milliseconds of the onset of a sound produced by an acoustic instrument or when it is stopped. They contribute to the perception that the sound was produced via “natural” and not “artificial” means.

G. Reverberation. In the code there are three constructors to choose from, depending on how much control you want to have (and thus, how much you want to be burdened with details).

SIMPLE constructor.
Room_size this is constructed to give some reasonable results when you input room_size values between 0.0 and 1.0. I found some parameter sets that I felt approximated various familiar rooms (lincoln hall, MB 1201, great hall, a large stadium, etc), and mapped these into a room_size space of 0.0 to 1.0. Then I used gnumeric (spreadsheet for linux, like excel) to find a linear regression to get all the parameters simply in terms of room_size. Thus a value of 0.0 will have virtually no reverb. A value of 1.0 will have a very high degree of reverb.

MEDIUM (next-most-advanced constructor).

- param reverbPercent this determines the mix (in the resultant sound) between the direct sound (the input sound) and the reverberated sound. 0.0 is no reverb, all direct sound. 1.0 is all reverb, no direct sound.
- param hilow_spread in most rooms the low frequency response will be higher than the high frequency response. If we hold other parameters constant, the math gives a range of possible ratios between low frequency to high frequency response. A value of 1.0 will choose the maximum low-frequency response, holding the high-frequency response at a given. A value of 0.0 will mean an even response across all frequencies.
- param gainAllPass Moore's reverberator has only one all-pass filter, and this is its gain. The default value is 0.7. Higher values (up to 1.0) cause more ringing at certain frequencies. Lower values (down to 0.0) cause less ringing.
- param delay this is the length (in seconds) of the first echo response.

ADVANCED constructor.

- param reverbPercent this determines the mix (in the resultant sound) between the direct sound (the input sound) and the reverbed sound. 0.0 is no reverb, all direct sound. 1.0 is all reverb, no direct sound.
- param combGainList this is the address of an array of 6 floats that are the comb filter gains.

- param lpGainList this is the address of an array of 6 floats that are the low-pass filter gains

- param gainAllPass Moore's reverberator has only one all-pass filter, and this is its gain. The default value is 0.7. Higher values (up to 1.0) cause more ringing at certain frequencies. Lower values (down to 0.0) cause less ringing.
- param delay the length (in seconds) of the first echo response.

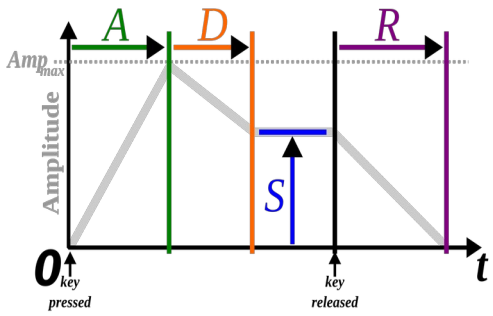
Common code used by all constructors.

- param percentReverb this determines the mix (in the resultant sound) between the direct sound (the input sound) and the reverbed sound. 0.0 is no reverb, all direct sound. 1.0 is all reverb, no direct sound.
- param combGainList this is the address of an array of 6 floats that are the comb filter gains.
- param lpGainList this is the address of an array of 6 floats that are the low-pass filter gains
- param gainAllPass Moore's reverberator has only one all-pass filter, and this is its gain. The default value is 0.7. Higher values (up to 1.0) cause more ringing at certain frequencies. Lower values (down to 0.0) cause less ringing.
- param delay this is the length (in seconds) of the first echo response.

written by Andrew Kurtz & Jim Lindstrom 2002

H. Spectrum represents the internal makeup of a sound by defining the number of partials/overtones, their frequency, and the evolution in time of their loudness/amplitude resulting in a *spectral envelope*. This is accomplished by assigning a scaled envelope to each partial. All envelopes used in Spectrum **have to** start and end with y=0 values (x=0, y=0 and x=1, y=0) a y ≠ 0 value for either x=0 or x=1 will produce a distortion (pop or click) that could even damage the speakers. A typical sound will have the fundamental (partial 1) scaled at the max value (1) and the rest of partials scaled progressively lower. Spectra are the children of the Bottom event in the sense that they define the type of each sound (actually, itself an event).

I. Envelopes. Envelopes are *functions* i.e binary relations between two sets. In DISSCO, most, but not all of them, are functions of time representing the evolution in time of some parameter. Traditionally, the most used envelope for partials is the Attack-Decay-Sustain-Release (ADSR) envelope:



It is recommended that envelopes be *normalized*, with max *y* values reaching 1, and then scaled according to needs. Using the numerical coordinates in the EnvLib function, the range of *y* values can be extended: in the case of the glissando feature, for example, max *y* values need to exceed 1. The max *x* value will be automatically extended or compressed to correspond to the duration of the sound if the envelope reflects the evolution in time of a parameter.

A straight line, *x*=0, y=1 to *x*1, *y*=1, that can be scaled, will show a parameter that does not change during the duration of the sound and it is useful when DISSCO requires the use of an envelope although a constant value is desired (ex. location of the sound source does not change).

J. Markov chains. A Markov chain is a stochastic (random) process involving a number of discrete *states* of a system and *probabilities of occurrence* of a state that depend only on the immediately preceding state of the system. It can be represented by a *transition matrix* of probabilities. Some Markov chains arrive to a stationary state in which a state or

a group of states are being repeated in similar sequences. First used in musical composition by Iannis Xenakis in late 1950s, it can be used to create sequences of events, pitches, rhythmic values, etc.

K. Sieves. Part of *Number Theory*, sieves are logical filters utilizing modulo and Boolean operations to select elements from a continuum of equally distanced elements (Peano axiomatic) such as Equally Tempered pitches. In Xenakis' notation, 3_1 denotes modulus 3, remainder (offset) 1 giving:

$3_1 = \{1, 4, 7, 10, 13,\ldots.\}$ while $3_0 \cup 3_1 = \{0, 1, 3, 4, 6, 7, 9, 10,\ldots\}$ and $2_1 \cap (3_1 \cup 3_0) = \{1, 3, 7, 9,\ldots.\}$

Applied to pitches, for C=0; 3_0 = C, Eb, F#, A (fully diminished chord) and $3_0 \cup 3_1$ will create an octatonic scale. Sieves are an essential tool in notating randomly created sounds when using Western traditional notation.

For more information about sieves: *http://cmp.music.illinois.edu/people/tipei/Papers/Files/perth.pdf*

Boolean operators.

- U union
- I intersection
- ~ complement (negation, NON)

L. Preferred Value Distribution. Finds a probability through a comparison of an average situation with a desired or "prefered" value. (Borrowed from MP1, N. Tipei expression):

probability = exp(FIRST_CONST + SECOND_CONST * pow(2, (value - 0.5)) * pow(2, checkPoint))

- FIRST_CONST = -5.54
- SECOND_CONST = -1.84
- checkPoint: the moment in time (x axis) of the testing.

Ma. Number of Children Density - Sounds per second. This method maps an "abstract" density of 0 - 1 onto a number of sounds per second by defining a total number of regions (*areas*) and the number of such areas with less than 1 sound/sec. (*underOne*). A common example:

areas	0	1	2	3	4	5	6	7
dens.	0.00	0.125	0.250	0.375	0.500	0.625	0.750	0.875	1.00
s/sec	1/16	1/8	1/4	1/2	1	2	4	8	16

Here, there are 8 *areas* (0-7) and 4 of them (0-3) include values of less than 1 sound/sec. By subtracting *underOne*, a density = 0 is mapped onto a value of 1 sound every 16 sec. and not onto 0.

In most cases *areas* = 8 and *underOne* = 4, as in the above example, are sufficient but the user can change these values.

Finding the density of an Event when its duration is known and when the number of objects/children is known:

$$\log_{10}(\text{numObjects} / \text{duration}) / (\text{area} * \log_{10} 2) + \text{underOne/area}$$

for areas = 8 and underOne = 4:

$$\log_{10}(\text{numObjects} / \text{duration}) / (8 * \log_{10} 2) + 0.5 =$$
$$\log_{10}(\text{numObjects} / \text{duration}) / 2.408239965 + 0.5$$

To find the number of objects given the density:

soundsPsec = pow(2, dens * area - underOne)

and

numObjects = duration * soundsPsec

==========

Mb. Densities for layers of the same parent:

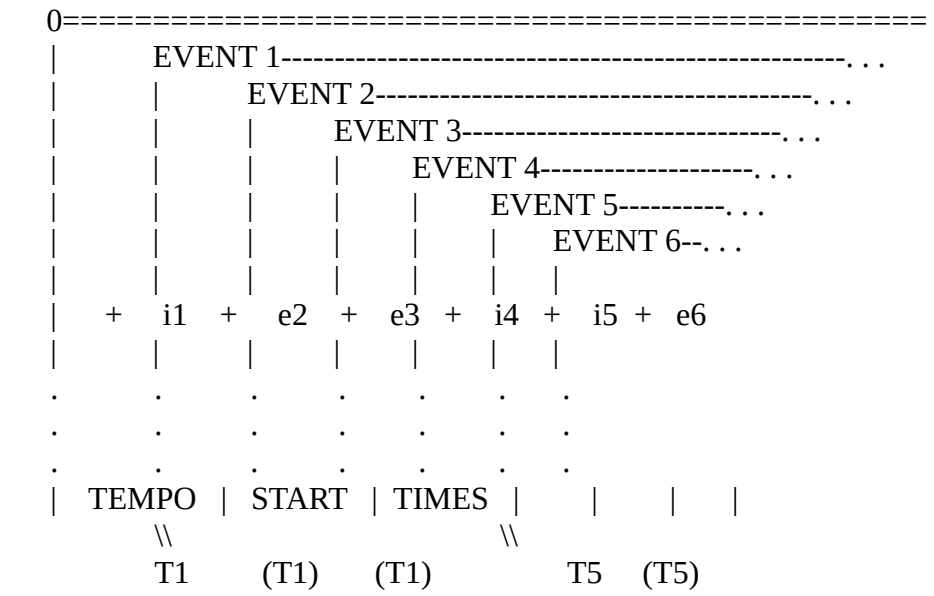
If all objects have the same duration:

1. Find out the total number of objects in that event.
2. Find out the % of time each layer is active
3. Normalize these % (divide each by their sum)
4. Find out how many objects in each layer

5. Determine the density of each layer according to the above

N. Event Offset Nesting. The following graphic attempts to show the many possibilities of nested exact and inexact offsets.

i = inexact start offset
e = exact start offset
T = tempo start time



Inexact Start Times (~ means the exact value is truncated to floating point):

Event 1 = i1
Event 2 = i1 + ~e2
Event 3 = i1 + ~e2 + ~e3
Event 4 = i1 + ~e2 + ~e3 + i4
Event 5 = i1 + ~e2 + ~e3 + i4 + i5
Event 6 = i1 + ~e2 + ~e3 + i4 + i5 + ~e6

Exact Start Times:

Event 1 = (not applicable)
Event 2 = T1 + e2
Event 3 = T1 + (e2 + e3)
Event 4 = (not applicable)
Event 5 = (not applicable)
Event 6 = T5 + e6

Possible combinations:
1) Parent inexact, child inexact (Events 4-5)
Since both are inexact, nothing further is to be done. They will both only have global inexact time offsets.

2) Parent exact, child inexact (Events 3-4)
Since the child is inexact, nothing further is to be done. The child will simply have a global inexact time offset. The parent will already have calculated its tempo start time.

3) Parent exact, child exact (Events 2-3)
Since the both are exact, the child inherits the tempo of the parent. Its exact offset is calculated by adding the exact parent start time offset. Times (~ means the exact value is truncated to floating point):
Event 1 = i1
Event 2 = i1 + ~e2
Event 3 = i1 + ~e2 + ~e3
Event 4 = i1 + ~e2 + ~e3 + i4
Event 5 = i1 + ~e2 + ~e3 + i4 + i5
Event 6 = i1 + ~e2 + ~e3 + i4 + i5 + ~e6

Exact Start Times:
Event 1 = (not applicable)
Event 2 = T1 + e2
Event 3 = T1 + (e2 + e3)
Event 4 = (not applicable)
Event 5 = (not applicable)
Event 6 = T5 + e6

Possible combinations:
1) Parent inexact, child inexact (Events 4-5)
Since both are inexact, nothing further is to be done. They will both only have global inexact time offsets.

2) Parent exact, child inexact (Events 3-4)
Since the child is inexact, nothing further is to be done. The child will simply have a global inexact time offset. The parent will already have calculated its tempo start time.

3) Parent exact, child exact (Events 2-3)
Since the both are exact, the child inherits the tempo of the parent. Its exact offset is calculated by adding the exact parent start time offset.

Important Note:
If the child attempts to override the parent tempo, it will be ignored and the above calculation. This is to prevent implicitly nested tempos, which are better handled explicitly at the moment. For example it would be very difficult to properly render "4/4 for 3 1/4 beats, then change to 5/8 for 3 beats as a child tempo." If this nesting were allowed, it would be very ambiguous as to how to return back to 4/4. Even if the 5/8 were to trigger a new tempo start time, in the score this would be misleading making it appear that the two sections were not rhythmically related, even though they inherently are by virtue of them both being exact.

4) Parent inexact, child exact (Events 1-2, 5-6)
In this case, since the parent did not have an exact offset from the grandparent, the exact child needs a new reference point. This triggers the creation of a new tempo start time *for the parent*. Since the child is offset an exact amount from the parent, the parent is the new tempo reference.

This could easily be the source of confusion: when a parent offset is inexact, and a child offset is exact, it is the parent which takes on the new tempo.

Note that this implies that the child's siblings will refer to the same new tempo start time.