ALGORITHMICALLY COMPOSED MUSIC

By: James G. Cialdea, Jr

Sufficiency Course Sequence:
MU1611 – Fundamentals of Music I – A 2005
MU2611 – Fundamentals of Music II – B 2005
MU3001 – World Music – D 2006
MU3611 – Computer Techniques in Music – A 2006
MU3613 – Digital Sound Design – B 2006
MU3612 – Computers and Synthesizers in Music – D 2007

Submitted to Professor David Linnenbank
WPI Department of Humanities and Arts
Thursday, April 26, 2007 - D Term 2007
Project Number: DL2-MU-07

Submitted in Partial Fulfillment of the Requirements of
The Humanities & Arts Sufficiency Program
Worcester Polytechnic Institute Worcester, Massachusetts

Table of Contents

1. Introduction

The goal of the project was to create a system that would compose music algorithmically, without the help of artificial intelligence practices. Originally, the system was expected to create music in a specific style, but through some research, it became clear that this has already been done very well. One such system is "Sharle", which was created in 1996 by Chong Yu at MIT as a master's thesis.[1] Full source code is available[2] so there was no sense repeating Yu's process. David Cope, currently a professor of Music at the University of California at Santa Cruz, has also done a tremendous amount of work with algorithmically composed music following a given style. He has been working on composition systems for over 15 years and has perfected many different methods, most of which use artificial intelligence very heavily.[3] After finding that these systems have already been developed and work much better than anything that could be created within this project's seven-week time constraint it was decided that the project should allow the system to create its own style by only enforcing basic music theory rules.

The system, called MLMPV, which stands for "Multi-Line, Metronome, Pitch, Velocity", was developed in an environment called Max. Max is a graphical programming language developed by Cycling'74 for working with MIDI based music. Max was a great thing for this project because it comes packaged with objects to deal with most of the low-level issues like interfacing with MIDI and creating metronome clock signals. Max also includes many probabilistic functions, which became key elements of MLMPV. There are some limits to Max, such as its inability to perform some types of recursive computations, but most were overcome without much difficulty. The "out-of-the-box" functionalities of Max allowed more time to be spent on the design and implementation of the algorithm.

[1] Chong Yu's thesis is available at http://brainop.media.mit.edu/online/net-music/net-instrument/Thesis.html
[2] Sharle source code is available at http://home.comcast.net/~chtongyu/sharle/
[3] David Cope's work and biography are available at http://arts.ucsc.edu/faculty/cope/

2. Development

MLMPV was developed in stages. Each stage built upon the previous and added new functionality or design improvements. The name of each stage's patch refers to what functionalities the patch implements.

2.1 Duel-Basic

See Also: Figure A-2.1, Audio CD Track 1

This was the first patch created for the project. Its title comes from the fact that there are two concurrent musical lines "dueling" with only "basic" probabilistic algorithms used. The patch was meant to provide the basic structure for building the algorithm. It simply creates a sequence of notes at a random pitch anywhere in the full MIDI range, a completely random velocity anywhere in the full MIDI range, and a duration less than 250 milliseconds. As soon as one note finishes playing, a new note is generated. The "music" produced is completely atonal, has no sense of rhythm whatsoever, and features huge jumps in pitch that are most unmusical.

2.2 Duel-Metro

See Also: Figures A-2.2 and A-2.3, Audio CD Track 2

In order to make the music slightly more rhythmical, it was obvious that a metronome needed to be implemented. The "Advanced Metronome" (*advmetro*) was created to time different note types simultaneously and synchronously. It uses Max's **tempo** object to produce 16 "bangs" - which are Max's "go" signals - per beat. The "bangs" are counted by **counter** objects set to different maximum counts to produce a measure count, a half note count, a quarter note count, an eighth note count, and a sixteenth note count all synchronized to the same metronome signal. All the counts, except the measure count, are relative to the current measure beats. For example, there are 16 sixteenth notes in a measure, so the sixteenth note counter counts from 1 to 16, then starts back at 1 again. The measure counter counts the number of measures since the *advmetro* was started or reset. Each count is sent out of the *advmetro* object through its own outlet and is also connected to variable names using Max's **send** object. This way, it is possible to synchronize the entire patch without needing to connect back to the *advmetro* object each time a clock signal is required.

Duel-Metro uses the *advmetro* to trigger the end of a playing note. This is a significant feature, because it forces the system to stay in predictable 4/4 time. If notes were simply given a duration, there is a fairly good chance that rhythms would not line up to with measures or phrases. For instance, it would be possible to play a half note followed by a triplet followed by a half note, which would not make very much musical sense. By selecting when the end of the note occurs, the same scenario would be immediately forced back into 4/4 rhythms. For example, if a half note were to be followed by a triplet, everything would sound normal. If the system then selected a half note to follow the triplet, what would actually play would be a note with length equivalent to 1 2/3 beats. This happens because the end of the note would be triggered by the next half note event

produced by *advmetro*, which occurs at the beginning of the next measure, 1 2/3 beats away. Any notes following the second half note will line up with the beginning of the measure perfectly.

Because *advmetro* is based in 4/4 time, the music actually takes on 4/4 characteristics because of the way note events are distributed within a measure. For instance, there is a 100% chance that there will be a note started on the first beat of a measure because every type of note has an *advmetro* event at that time. The *advmetro* event will trigger the end of all notes playing, which will cause a new note to start immediately. This observation can be applied to every part of every beat in the measure.

Beat	**1**	**1 1/4**	**1 1/3**	**1 1/2**	**1 2/3**	**2**	**2 1/4**	**2 1/3**	**2 1/2**	**2 2/3**
Notes Playing	1/4, 1/3, 1/2, 1, 2, 4	1/4	1/3	1/4, 1/2	1/3	1/4, 1/3, 1/2, 1	1/4	1/3	1/4, 1/2	1/3
Probability of event (notes/6)	100%	16.6%	16.6%	33.3%	16.6%	66.6%	16.6%	16.6%	33.3%	16.6%

Beat	**3**	**3 1/4**	**3 1/3**	**3 1/2**	**3 2/3**	**4**	**4 1/4**	**4 1/3**	**4 1/2**	**4 2/3**
Notes Playing	1/4, 1/3, 1/2, 1, 2	1/4	1/3	1/4, 1/2	1/3	1/4, 1/3, 1/2, 1	1/4	1/3	1/4, 1/2	1/3
Probability of event (notes/6)	83.3%	16.6%	16.6%	33.3%	16.6%	66.6%	16.6%	16.6%	33.3%	16.6%

Notice the close correlation between dominant beats and high probabilities. This pattern still applies if expanded to larger and smaller duration increments, such as 1/32 notes or 4 bar phrases. This shows that although the selection of duration is random, it is likely to produce a fairly pleasing rhythm, thus validating this method of duration selection.

Duel-Metro plays three musical lines simultaneously and removes some of the randomness that was observed to be so unmusical in *Duel-Basic*. Each of *Duel-Metro's* three lines produces a completely random pitch, but that pitch is limited to a single given octave. All velocities are also randomly distributed, but all velocities generated are greater than 50 to prevent major jumps in volume. The selection of which *advmetro* event used to trigger the end of a playing note is also completely random, but as stated previously, this approach works well. The music produced by *Duel-Metro* is completely atonal and still has some wild volume variations, but the rhythm is what one would expect in a 4/4 piece.

2.3 Multi-Line-Metro

See Also: Figures A-2.4 and A-2.5, Audio CD Track 3

This patch is basically an improved version of *Duel-Metro*. The first major difference is *advmetro's* ability to trigger triplets in addition to its previous capabilities. This is achieved by having *advmetro's* tempo object pump out 48 bangs per beat instead of 16 and adjusting the counters accordingly. The algorithms responsible for actually generating and playing the notes were placed into a subroutine, called *randline*, which made it much easier to modify and duplicate them. *Multi-Line-Metro* plays four musical lines at a time with the same musical qualities - or lack there of - as *Duel-Metro*. This patch's main benefit is the ability to easily add and reconfigure additional randomly generated playback lines.

2.4 Multi-Line-Metro2

See Also: Figure A-2.6 and A-2.7, Audio CD Track 4

This patch addresses some of *Multi-Line-Metro's* shortcomings. It was obvious that the bass notes should not be playing rhythms containing triplets or sixteenth notes. To remedy this, the "rhythm complexity" setting was added to the *randline* subroutine. The rhythm complexity setting simply controls the number of different note types that are allowed for a given line. For example, a line with rhythm complexity set to 1 would only have quarter notes and half notes. A line with a rhythm complexity of 4 has all five types of notes available for random selection. This patch also has the ability to turn lines on and off in a way that creates a call and response effect. This is done by toggling two lines at a time. Simple randomness still determines the pitches and velocities.

2.5 Multi-Line-Metro-Pitch

See Also: Figures A-2.8 through A-2.11, Audio CD Track 5

This is where things start to sound musical. The *Multi-Line-Metro-Pitch* patch includes a subroutine called *pitchselect*, which calls a subroutine called *pitchselect2*, to select pitches to play. The *pitchselect* subroutine simply does some math on the pitches coming out of *pitchselect2* to make them fit into the proper range. This is important, because *pitchselect2* only works on one octave. The *pitchselect* subroutine allows for much greater flexibility by automatically transposing pitches to their proper octave. The *pitchselect2* subroutine uses two lookup tables containing the probability of each note's occurrence. The tables only encompass one octave, starting at the lowest C (MIDI pitch 0) extending to the lowest B (MIDI pitch 11). The tables were developed by simple experimentation, then hard-coded into the patch. Once a key and scale type are selected, a separate note probability table is filled with note probabilities related to the scale selected. This is achieved by simply copying the probability for each related scale degree from the lookup table into the note probability table. When *pitchselect2* is then asked to select a pitch, the note probability table is simply queried and a value is returned.

The note coming out of the table might not be the note that is played, however, as there is another test within *pitchselect2* to make sure that the note is not just in key, but makes musical sense with respect to the movement of the line. To reduce random sounding jumps, minimum and maximum note movement settings were added. To make this work, each note that is selected is stored until the next note selected replaces it. Each note coming out of the note probability table is checked against the last note selected to make sure that it is at least as far away from that note as the specified minimum note movement, but no further than the maximum note movement. If the note coming out of the table meets both requirements, it is allowed to pass through and is stored to compare to the next note. If the note does not meet the requirements, it is discarded and the table is queried again recursively. Despite Max's limitations in its recursive functionality, this method is able to produce a note every 10ms without issue, which is more than enough speed for this application.

Multi-Line-Metro-Pitch also has functionality to automatically trigger the call and response toggles. A random number is generated at the beginning of each measure. If the number is 1, 2, or 3 then one of the three toggles will be triggered, which in turn will change the combination of which lines are playing. The music produced by *Multi-Line-Metro-Pitch* is tonal within a given key and includes the advanced 4/4 rhythms developed by adding triplets to the list of duration choices. The addition of automated line toggling makes the music significantly more interesting to listen to.

2.6 Multi-Line-Metro-Pitch-Velo

See Also: Figures A-2.12 through A-2.14, Audio CD Track 6

This patch is where the MLMPV system begins to come together and produce musical results. The basic functionality is identical to *Muilti-Line-Metro-Pitch*, but this patch adds the *veloselect* subroutine to select velocities, instead of the completely random approach used in all previous patches. The first implementation of *veloselect* simply selected a velocity based on which of the *advmetro* events happened last. This quickly proved to be a poor method. Velocities were not being selected correctly at times where more than one event occurred simultaneously. When the system was working as designed, it was clear that the velocities needed to have more variation between beats of the same type. For instance, if four sixteenth notes were played consecutively, the second and fourth would be guaranteed to have exactly the same velocity. To correct these issues, velocities are selected by the actual count associated with each note type, not just the fact that there was an event. In this way, certain events that happen simultaneously with others can be filtered out or ignored, and each occurrence of a certain type of note within a measure can have a different velocity associated with it. For instance, on the first beat of the measure every note type triggers an event, but only the one coming from the measure counter is important. The signal from the measure counter will cause the appropriate velocity to be set and the value from all other counters, which are all "1" at this time, will be ignored. The second sixteenth note occurring in a measure can, and does, have a different velocity value associated with it than the fourth sixteenth note. This selection process fills in a major part of the musical flow by adding expressiveness to the music.

3. Conclusion

The MLMPV system demonstrates many musical elements, but it clearly cannot do everything a human composer could. MLMPV is capable of selecting pitches in key, playing them in 4/4 time, and playing with proper volume depending on the current position within a measure. MLMPV is not capable of consistently creating catchy melodies nor can it perform any sort of repetition purposefully. It is it unable to produce music containing rests or any sort of purposeful change in style. What MLMPV produces is consistent with many important parts of music theory, but it never bends the rules it has been given, so there is little chance that MLMPV will develop any new musical concepts on its own.

Although MLMPV is limited in its functionality, it demonstrates many ways to use simple computerized algorithms in music creation. Its design is such that there is room for expansion to include new musical elements. MLMPV demonstrates that musical rules can be turned into algorithms which a computer can run through to make a song. As more features were added to MLMPV, it became apparent that as more rules were added to the system a more musical sound was produced. To improve upon MLMPV, simply add more rules. Unfortunately, time constraints prohibited the addition of features such as rests and pauses, creation of musical beginnings and endings to the song, melody writing, repetition, automatic and musical key changes, the ability to make separate lines affect each other, and many others. Each of these elements has an infinite number of possible algorithmic solutions, any of which could be added to MLMPV as a new algorithmic rule.

A-1. Key Terms Explained

Max - The software these programs are written in. Created by Cycling'74. It is a graphical programming language focused on music.

MIDI - Musical Instrument Digital Interface. MIDI is a communications protocol for talking about music. It is event based, so no sound is actually transmitted, just commands to start, end, or alter notes. There are many other commands MIDI is capable of, but they do not pertain to this project

Patch or Patcher - This is what the Max folks call the programs written in their software. There are 2 different types of patches:

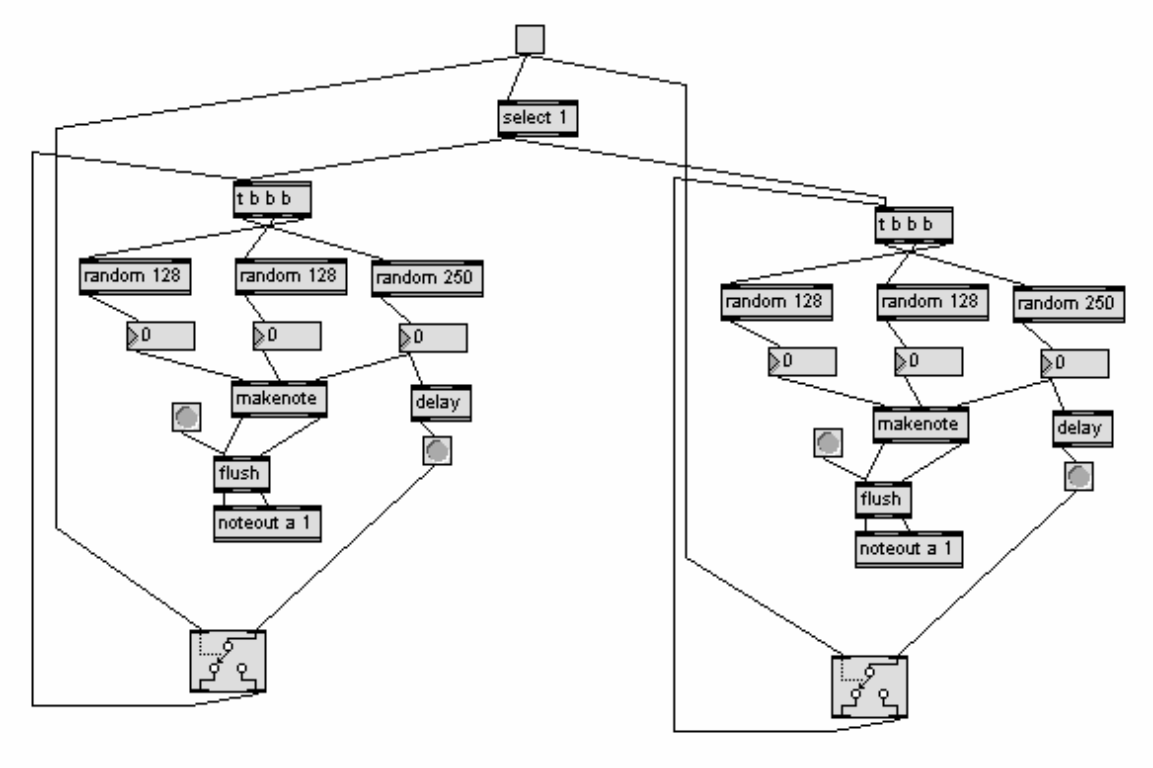
Program Patch - These patches are implementations of an algorithm. They create music (though possibly not good music).

Helper Patch - These patches generally do not do anything on their own, but their functionality can be added to program patches or helper patches quite easily. They are separated out for reuse or to keep a program patch a little cleaner looking. Usually, they will require additional information or stimulation to do their job.

Velocity - This is a MIDI word. It is a measure of how hard a note is struck. It usually corresponds directly to the volume the note is played with, and sometimes some other musical elements. Velocities range from 0 to 127, 1 being softest, and 127 being hardest. A velocity of "0" will cause a note to stop playing.

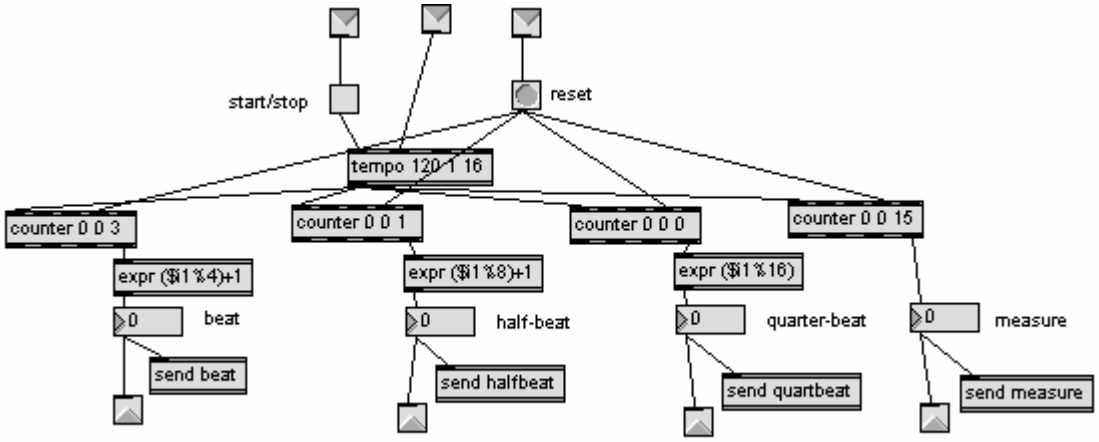
A-2. Max Patch Diagrams

A-2.1 Duel-Basic

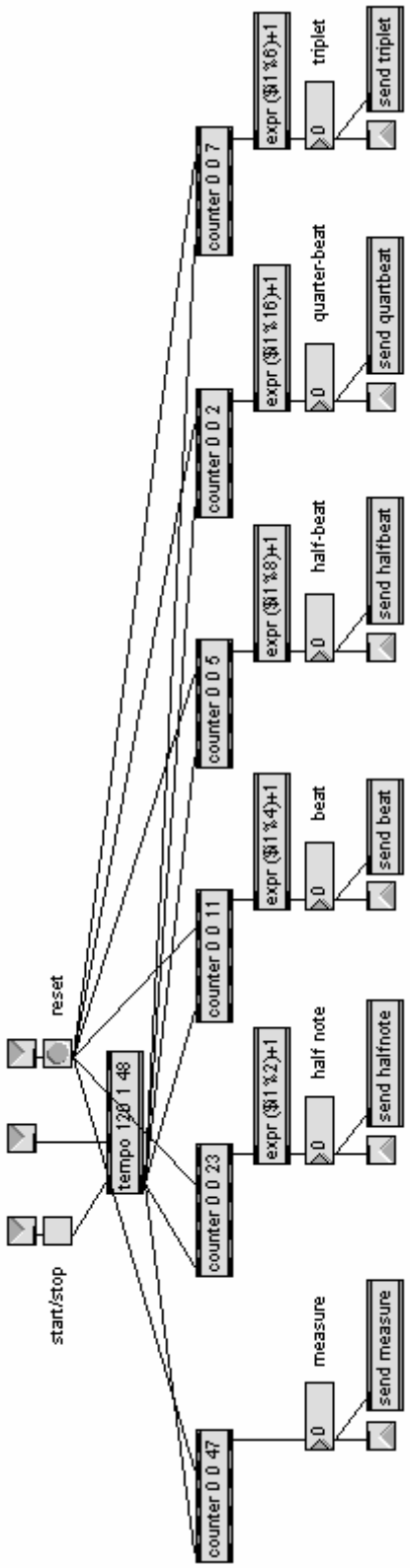


A-2.2 advmetro

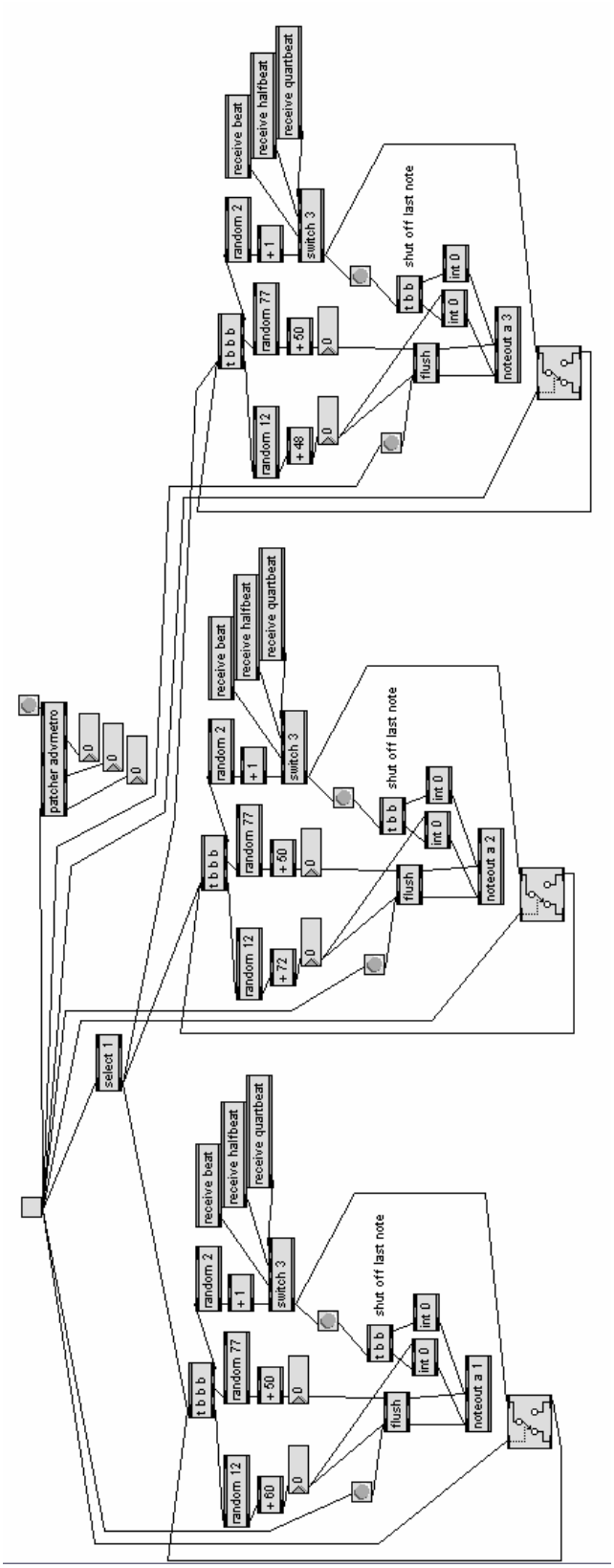
A-2.2.1 Without Triplets



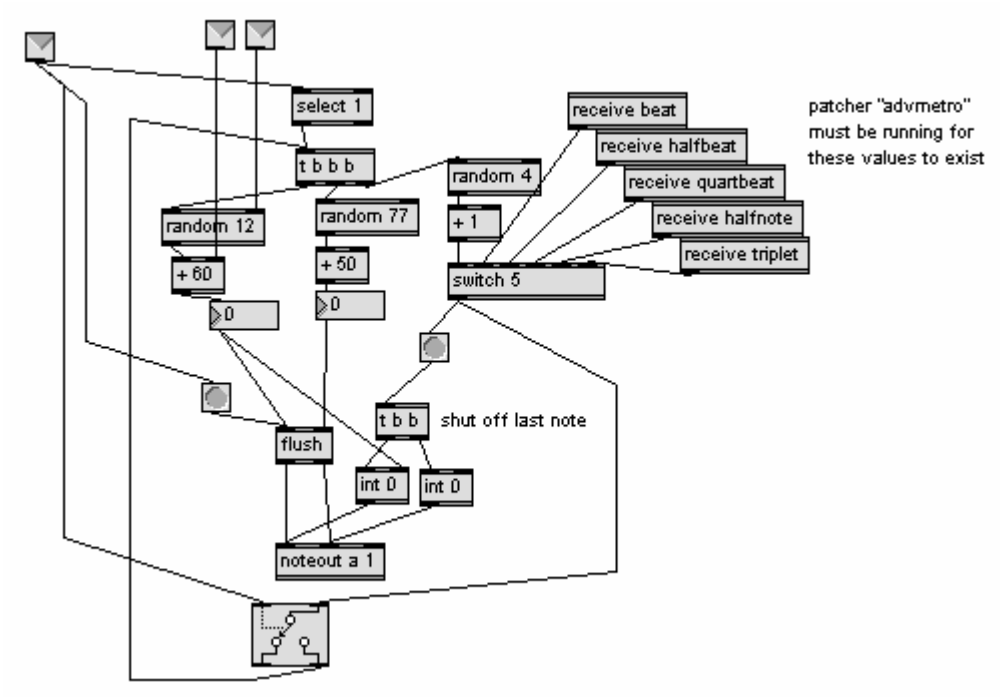
A-2.2.2 With Triplets



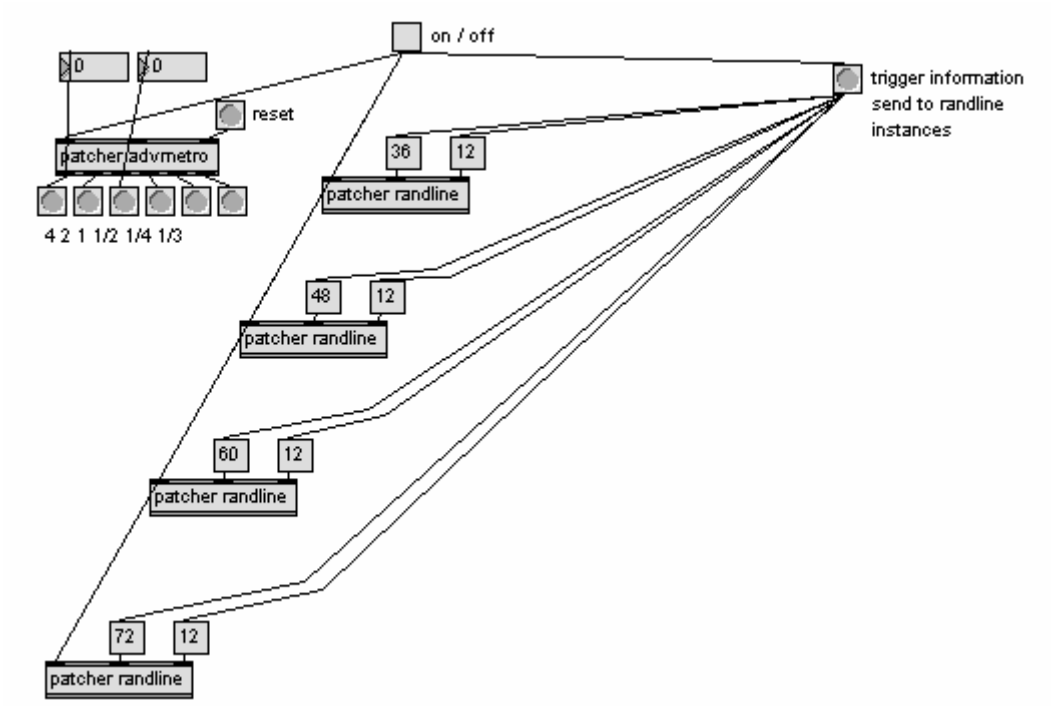
A-2.3 Duel-Metro



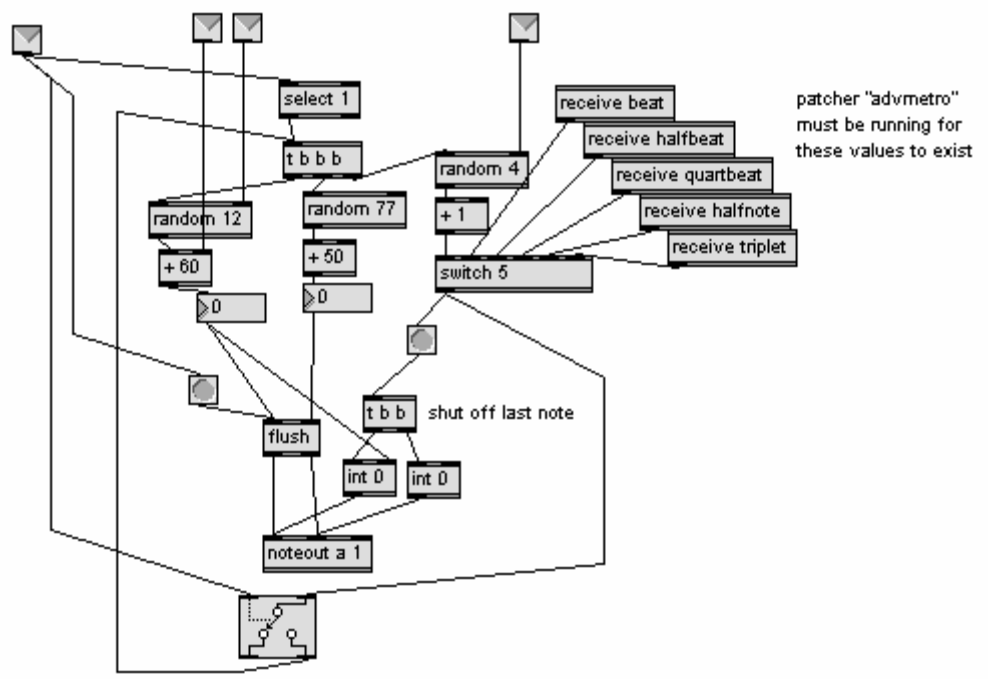
A-2.4 randline – as used in Multi-Line-Metro



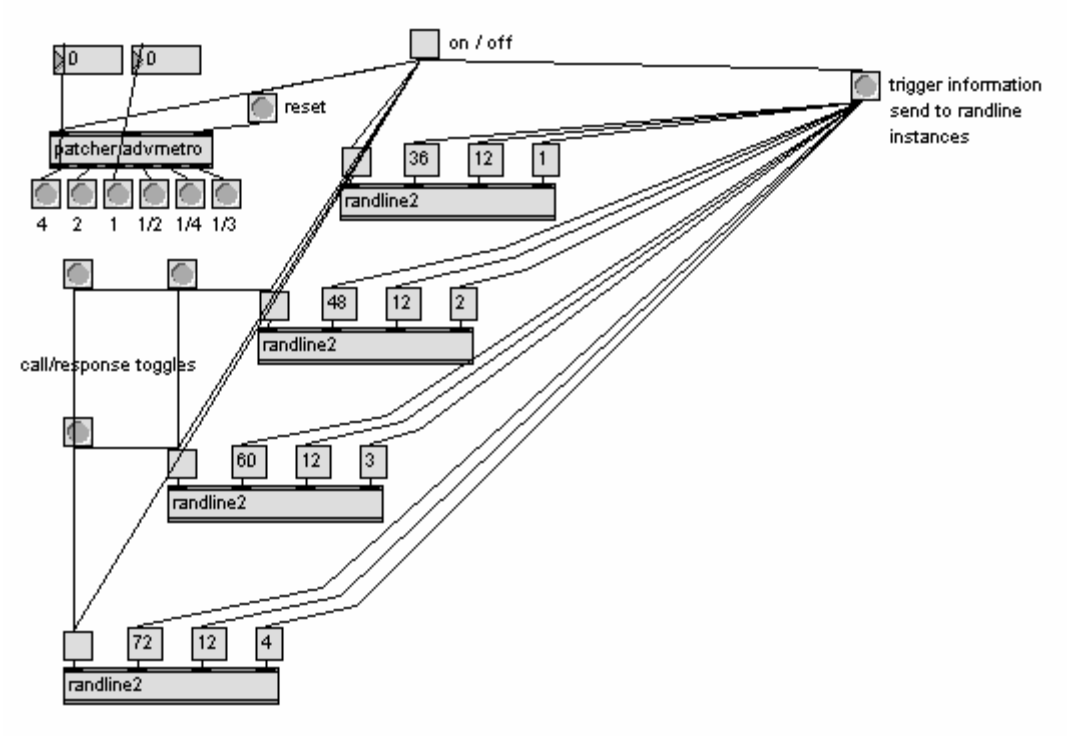
A-2.5 Multi-Line-Metro



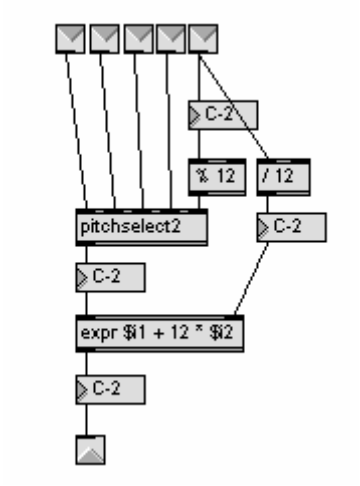
A-2.6 randline2 – as used in Multi-Line-Metro2



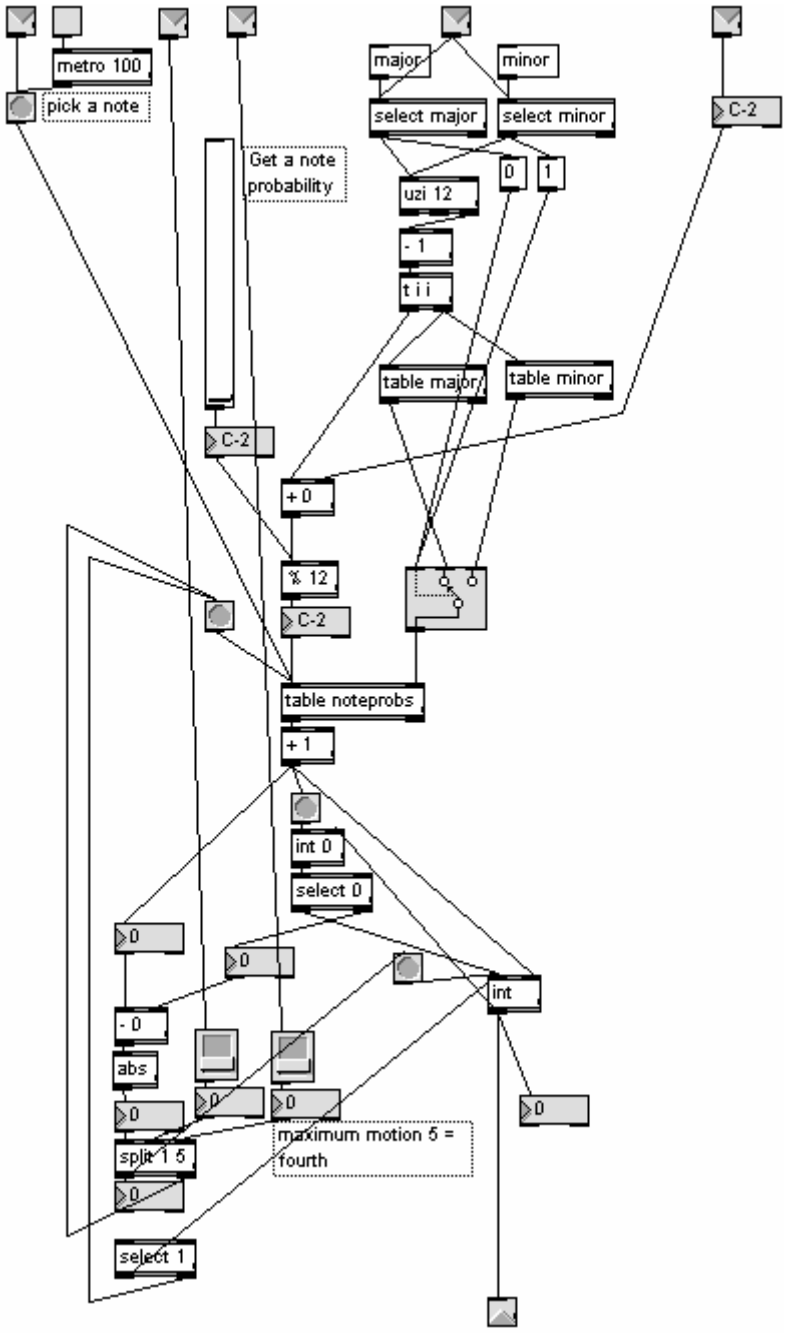
A-2.7 Multi-Line-Metro2



A-2.8 pitchselect



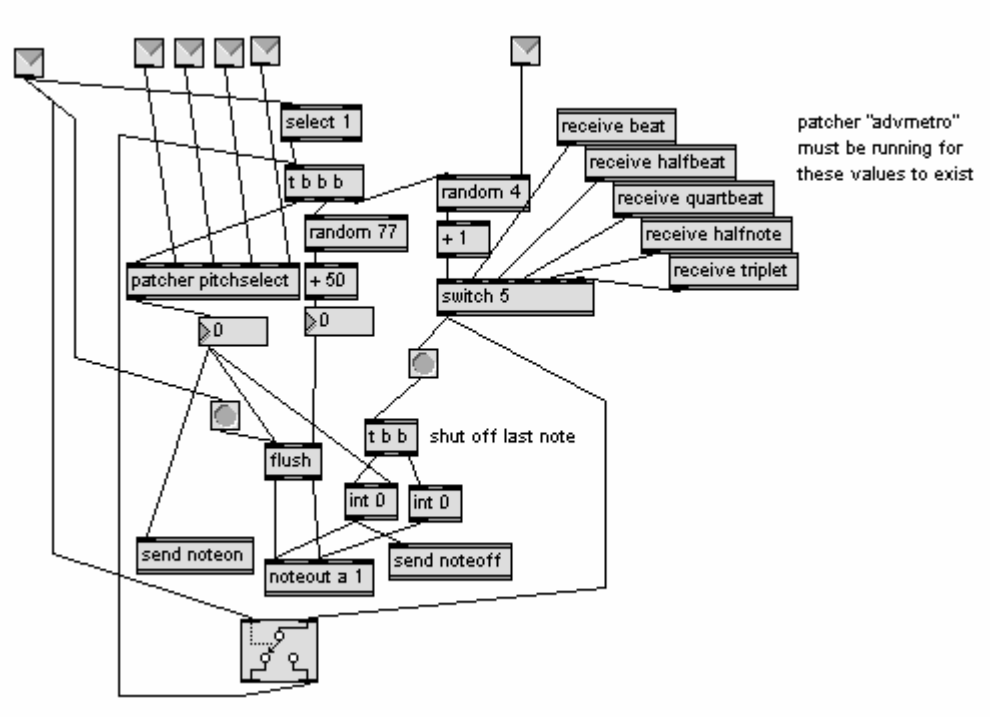
A-2.9 pitchselect2



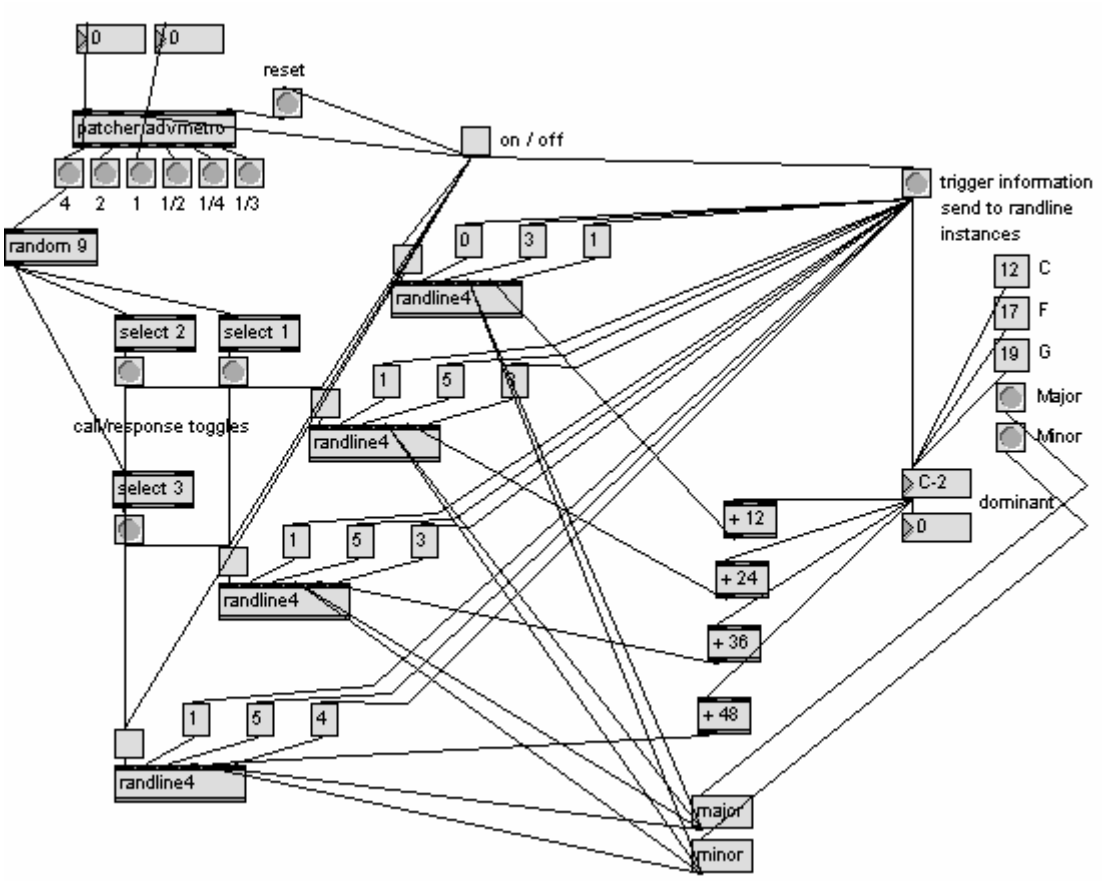
A-2.9.1 Table major

A-2.9.2 Table minor

A-2.10 randline4 - as used in Multi-Line-Metro-Pitch

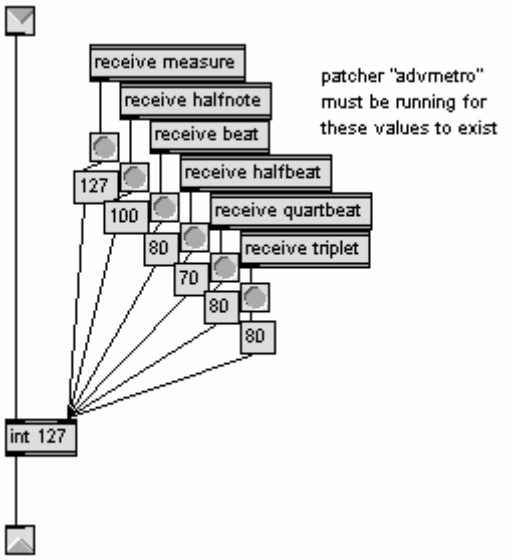


A-2.11 Multi-Line-Metro-Pitch

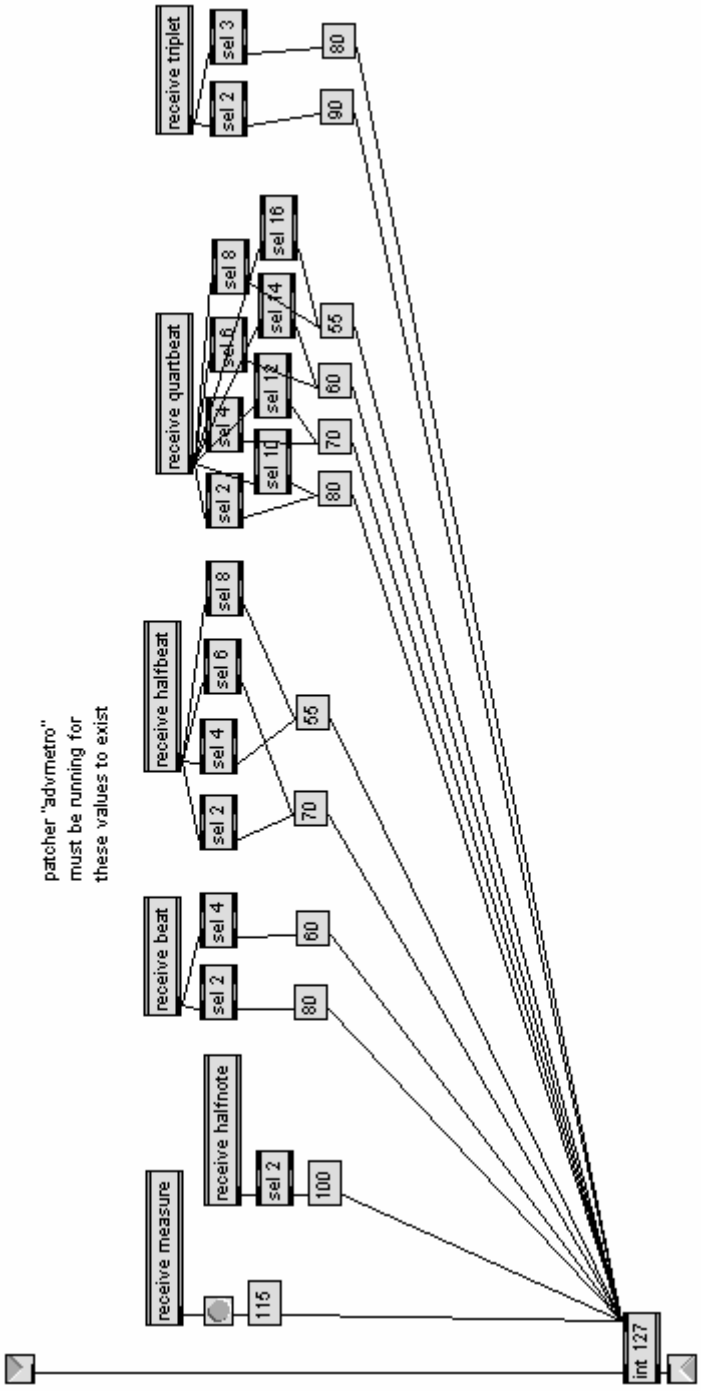


A-2.12 veloselect

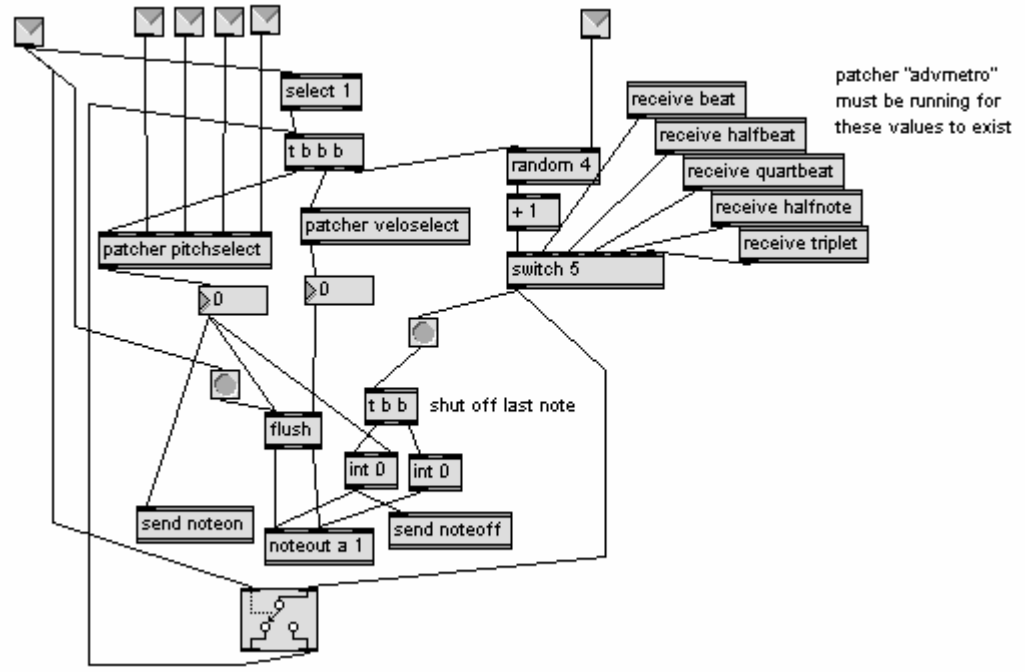
A-2.12.1 Using Single Values for Each Event Type



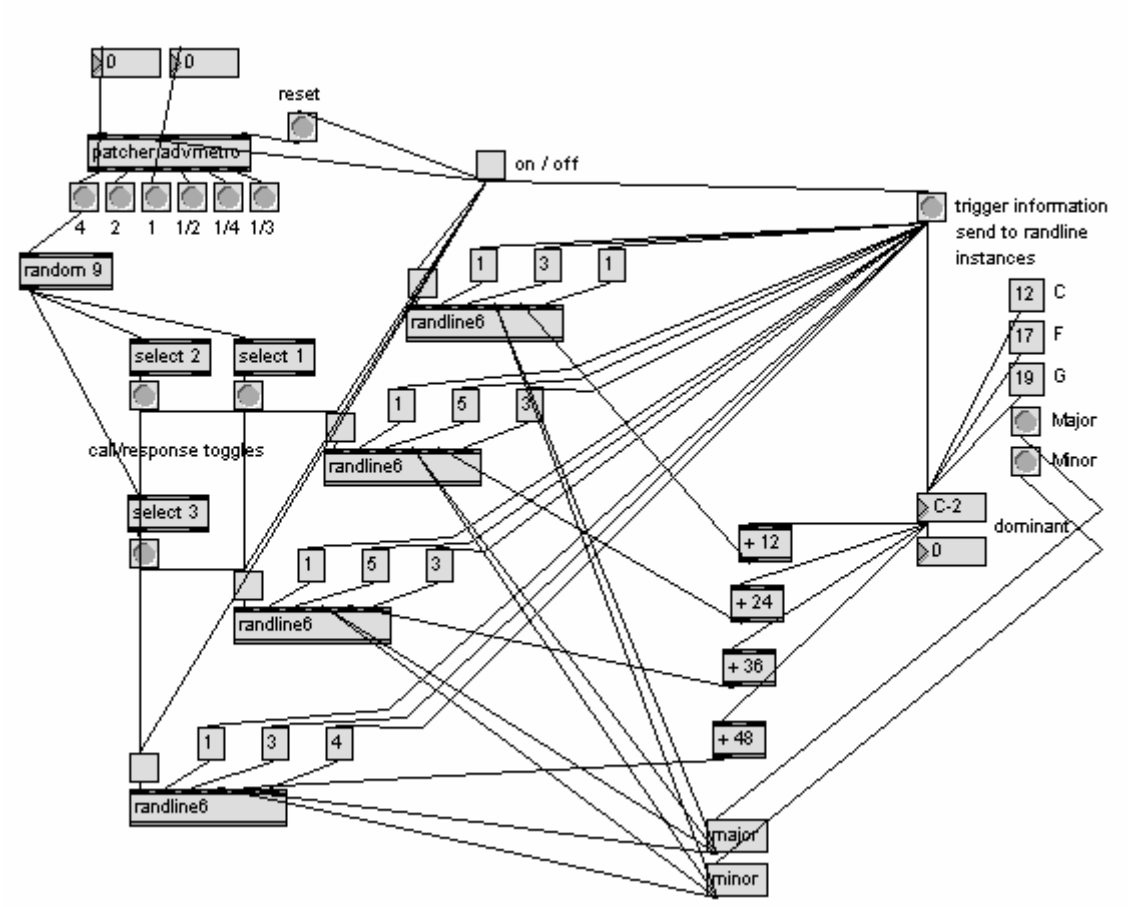
A-2.12.2 Using Multiple Values for Each Event Type Based on Beat



A-2.13 randline6 - as used in Multi-Line-Metro-Pitch-Velo



A-2.14 Multi-Line-Metro-Pitch-Velo



A-3. CD Contents

A-3.1 Audio CD

The audio CD includes a 5 minute sample of each patch's functionality.

1. Duel-Basic Example
2. Duel-Metro Example
3. Multi-Line-Metro Example
4. Multi-Line-Metro2 Example
5. Multi-Line-Metro-Pitch Example (In G Minor)
6. Multi-Line-Metro-Pitch-Velo Example (In G Minor)

A-3.2 Data CD

The data CD includes all the audio samples listed above in MP3 and AIFF format, sources for the Max patches used in their creation, and an electronic copy of this document.