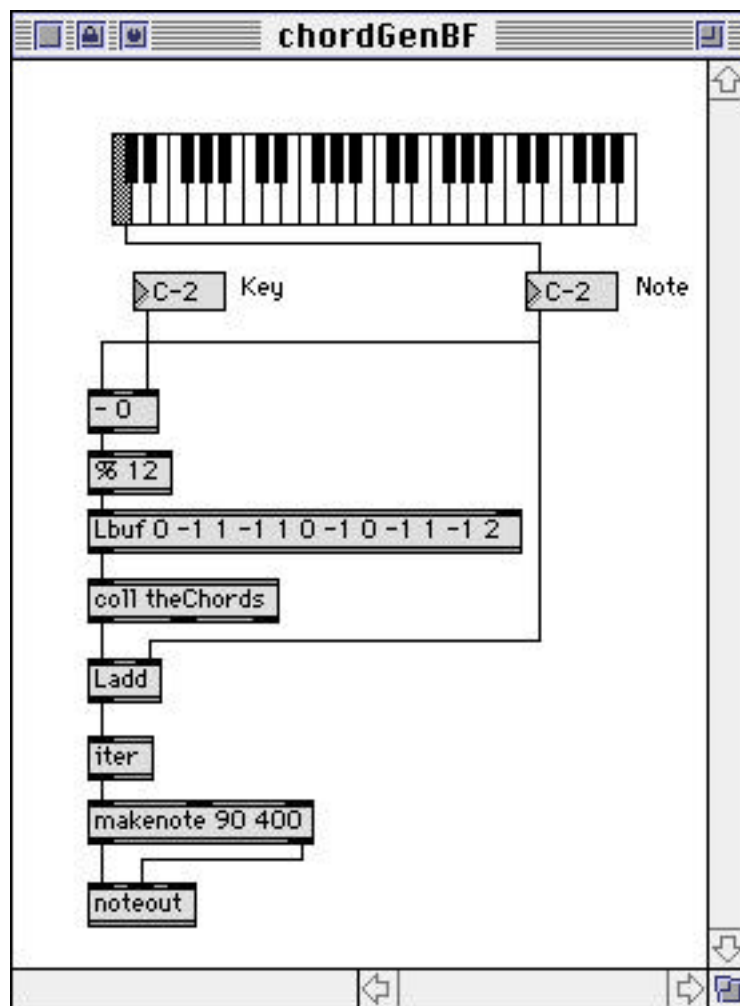


Max & Chords

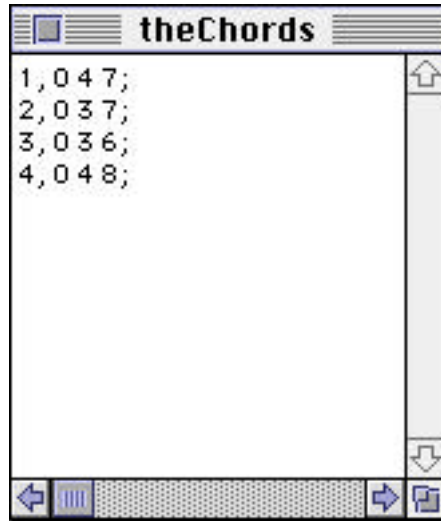
We routinely use Max to generate chords, usually intending some kind of automatic harmonization. Somewhat less often, we need to recognize chords, either as part of an analysis or to trigger specific events. This essay will give approaches to both problems.

Simple Harmony

The rules of harmonization are quite complex (not to mention personal to each composer), so I'll limit this discussion to generating chords appropriate to scale degrees in major and minor keys. We'll start with a straightforward approach:



This "brute force" approach simply picks the desired chord type out of a coll and builds it on a given root. Here's what's in the coll:



These are the intervals for major, minor, diminished, and augmented chords. They are added to the root by Ladd and played. (Notice the iter required to play chords with makenote. Makenote interprets a list as a single note.)

Instructions as to which chord to play are contained in Lbuf. The list associates chord types to scale degree, so the coding

1 0 2 0 2 1 0 1 0 2 0 3

will yield major on the first degree (pc = 0), nothing on the sharped first (there is no 0 line in the coll), minor on the second, and so forth. The pointer into Lbuf is calculated by

(pitch number - key) % 12

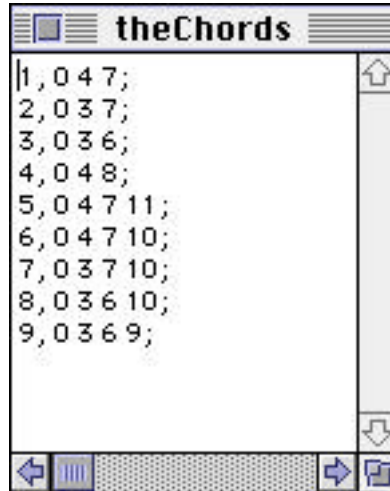
This will fail in the lowest octave, but in practice, those notes are never seen.

This is a very effective patch, suitable for most simple applications. You can easily add chords to the coll, and modify the rules by changing the list in Lbuf. This list:

2 0 3 1 0 2 1 0 1 3 1 3

Harmonizes minor pretty well.

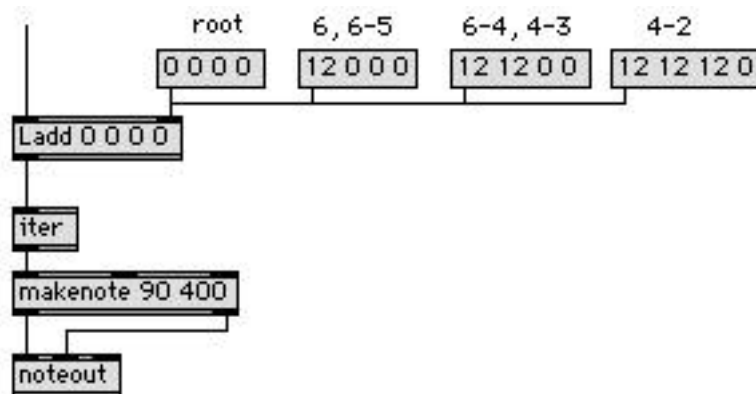
To include sevenths, we expand the coll:



Now the rule list has to be revised, or more likely, some complex logic designed that plays sevenths when the context requires.

Inversions

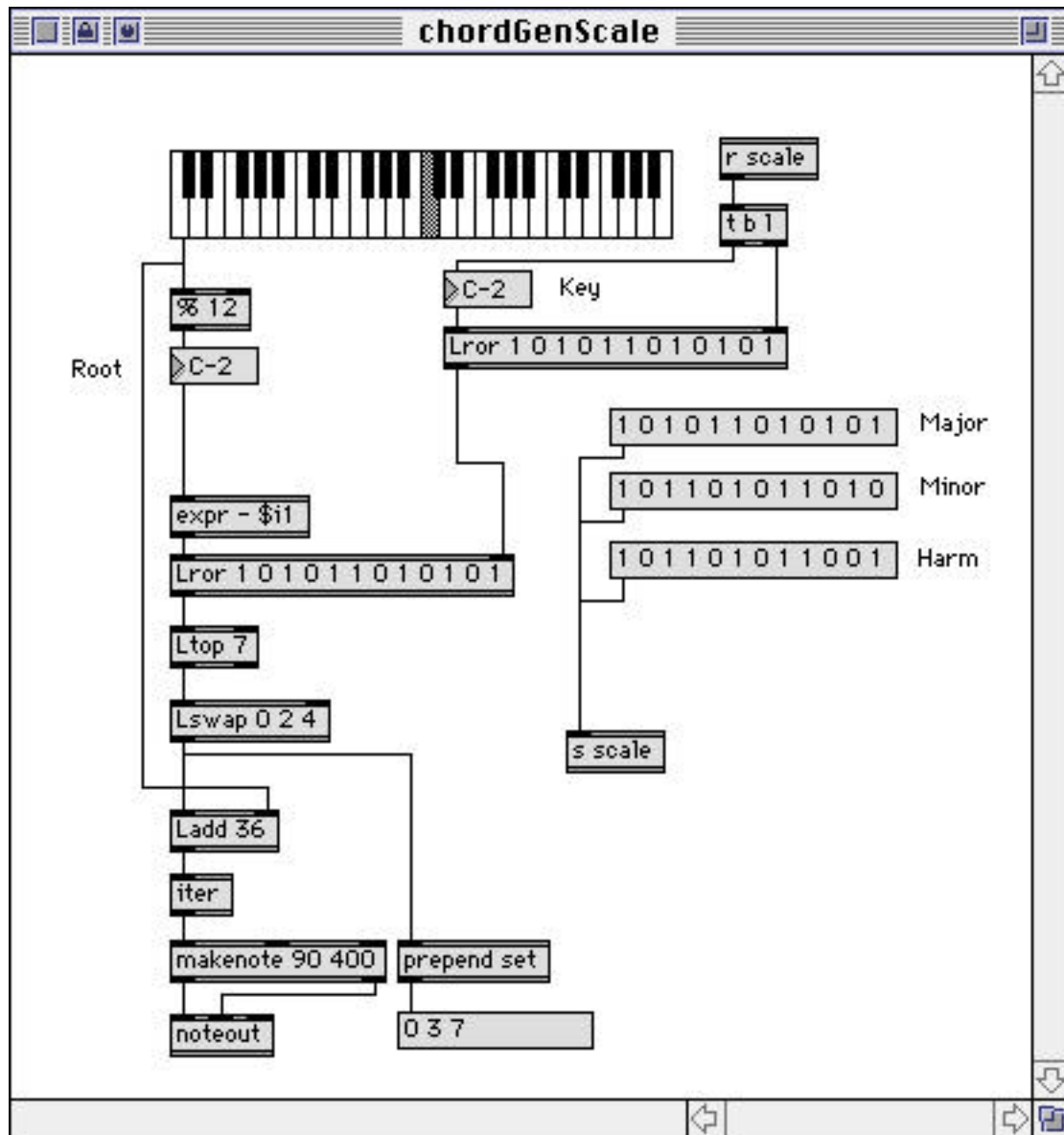
The chords produced are in root position. We can get various inversions by adding 12 to some of the pitches. This addition to the bottom of the patch will do the trick:



Again, additional logic will be needed to select the required inversion list.

Deriving Chords From Scales

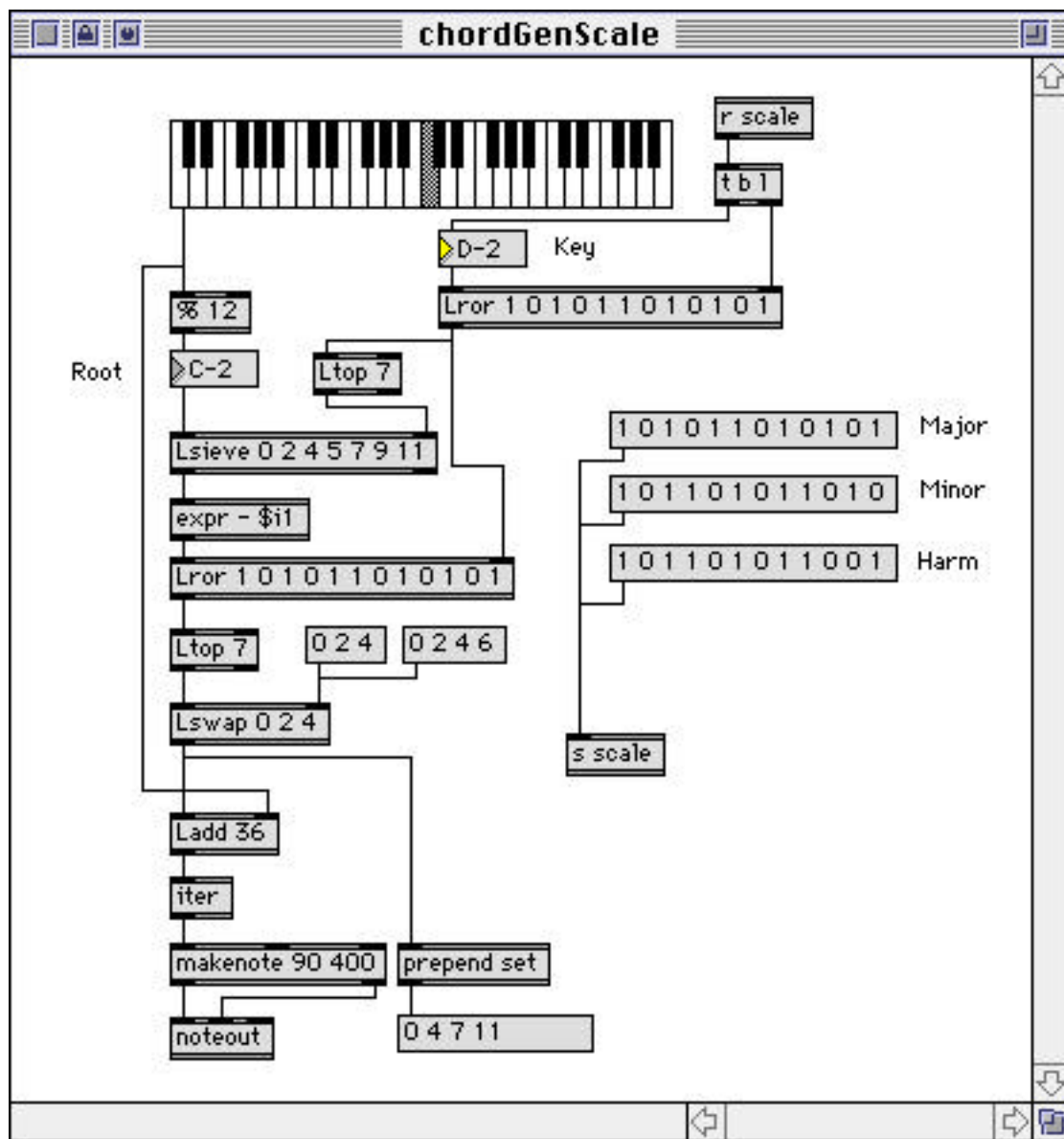
Since the progression of chords in a scale is systematic, it should be possible to generate chords without drawing up an explicit rule list. This patcher simply takes alternate notes out of scales:



The scales are notated as sets, lists of 12 members in which a one indicates the presence of a note at that position. (The positions represent pitch classes from C to B.) Each mode has its own distinctive pattern of 1s and 0s. We can change key by rotating the scale set to the right. (In rotation, numbers that fall off the end of the list are brought to the beginning. The Lror help file demonstrates.)

To generate a chord, we rotate the scale to the left till the root is in the first position. Ltop 7 produces a list of the members that have ones in them, and Lswap 0 2 4 extracts the first, third and fifth. These intervals are added to the root as in the first example. You can get seventh chords simply by changing the template in Lswap to 0 2 4 6.

Notes that are not in the scale produce the next higher chord. It's better to ignore chromatic notes- here's the patcher modified to filter them out:



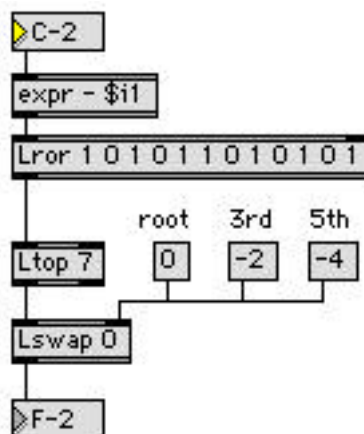
Lsieve allows specified values through. Here they are generated from the scale in the same manner that the chord tones are.

This patcher performs well, in that it systematically produces correct chords no matter what scale is put in. In actual practice, it leaves something to be desired. This is because

in minor keys, the most commonly used chords are really taken from two different scales. The harmonic minor works for all but the III chord and the i7, which are derived from natural minor. (Melodic minor brings in many more possibilities.) You could switch scales according to your preference for harmonizing various notes, but once you start doing that, you might as well revert to the brute force approach.

Generating Chords From The Third Or Fifth

Of course, any note may be harmonized three ways. The core technique of the above patcher can be used to find alternate roots. Just change the template in Lswap:



This can be used with any of the chord generators presented here, since they are all driven by the root.

Generating Chords For The Minor Scale With Fuzzy Logic

Fuzzy logic is a very powerful, relatively new system for modeling complex operations in an intuitive way. It is designed to deal with vaguely stated concepts, such as "tall" or "around 7". To give a musical example, the interval of a fifth is a vague concept, because it might be perfect, diminished, or augmented. In fuzzy terms, the fifth can be represented by the set:

0 0 0 0 0 0.7 1 0.5 0 0 0

Each position in the set represents a pitch class, just as in the scales used previously. Here, however, a pitch can have a value between 0 and 1, indicating "partial" membership or "possibility". In the example of the fifth, the diminished version is given a membership of 0.7. This does not indicate probability, it simply means that a tritone is subjectively somewhat less of a fifth than a perfect one.

With fuzzy logic, we can use these sets to evaluate problems like "find the fifth above D in the C major scale". The answer would be the note which is both a fifth above D, and found in C major. "And" type statements are evaluated by finding the intersection of two sets, like this:

A fifth:

0 0 0 0 0 0.7 1 .5 0 0 0

Rotate by 2 (D)

0 0 0 0 0 0 0 0 .7 1 .5 0

Then intersect with the C major scale. Fuzzy intersection is done by taking the lowest member of each position from the two sets:

0 0 0 0 0 0 0 0 .7 1 .5 0 0
min
1 0 1 0 1 1 0 1 0 1 0 1
0 0 0 0 0 0 0 0 1 0 0 0

The answer is the 9th position or A. The final set in a fuzzy problem is called the solution set. It is usually interpreted by finding the position of the highest value. To see how the fuzzy values work, compare a fifth above B with a fifth above F

A fifth above B
0 0 0 0 0 .7 1 .5 0 0 0 0 0
min
1 0 1 0 1 1 0 1 0 1 0 1
0 0 0 0 0 .7 0 .5 0 0 0 0 = F (diminished)

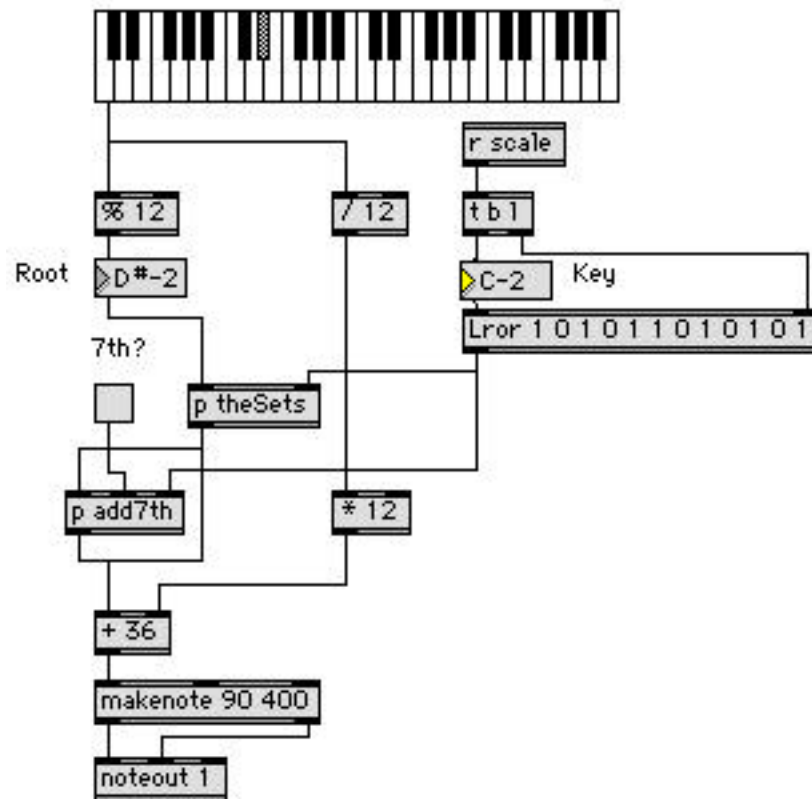
A fifth above F
1 .5 0 0 0 0 0 0 0 0 0 0 .7
min
1 0 1 0 1 1 0 1 0 1 0 1
1 .5 0 0 0 0 0 0 0 0 0 .7 = C (perfect)

For a more complete treatment of the topic, see the essay "Fuzzy Logic and Musical Choices". It's enough for this paper to state that a scale may also be fuzzy, in particular, a minor scale can be represented by

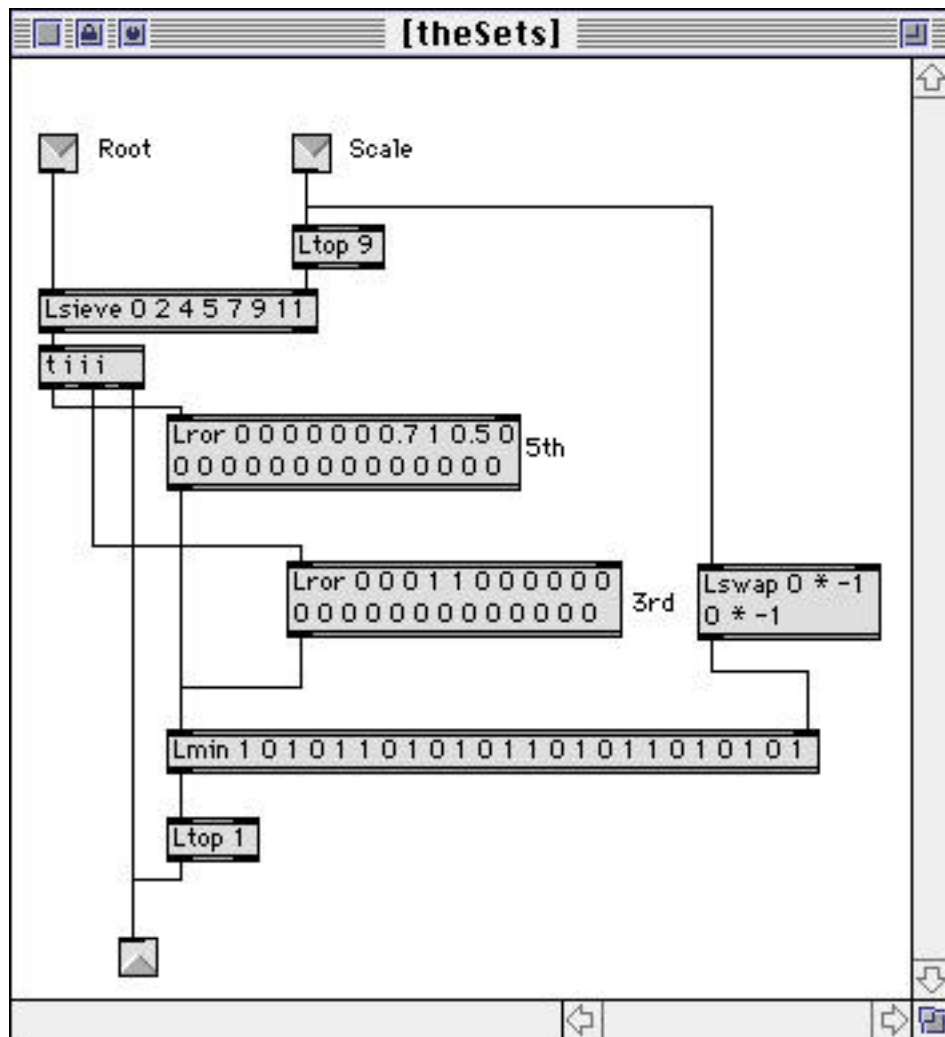
1 0 1 1 0 1 0 1 1 .7 .8 .9

where the partial memberships indicate the possibility of raised 6 and two types of seventh.

Here is a patcher that uses fuzzy operations to generate chords:

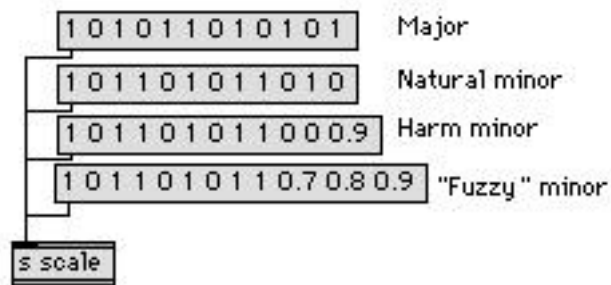


The fuzzy action is inside the subpatcher theSets:



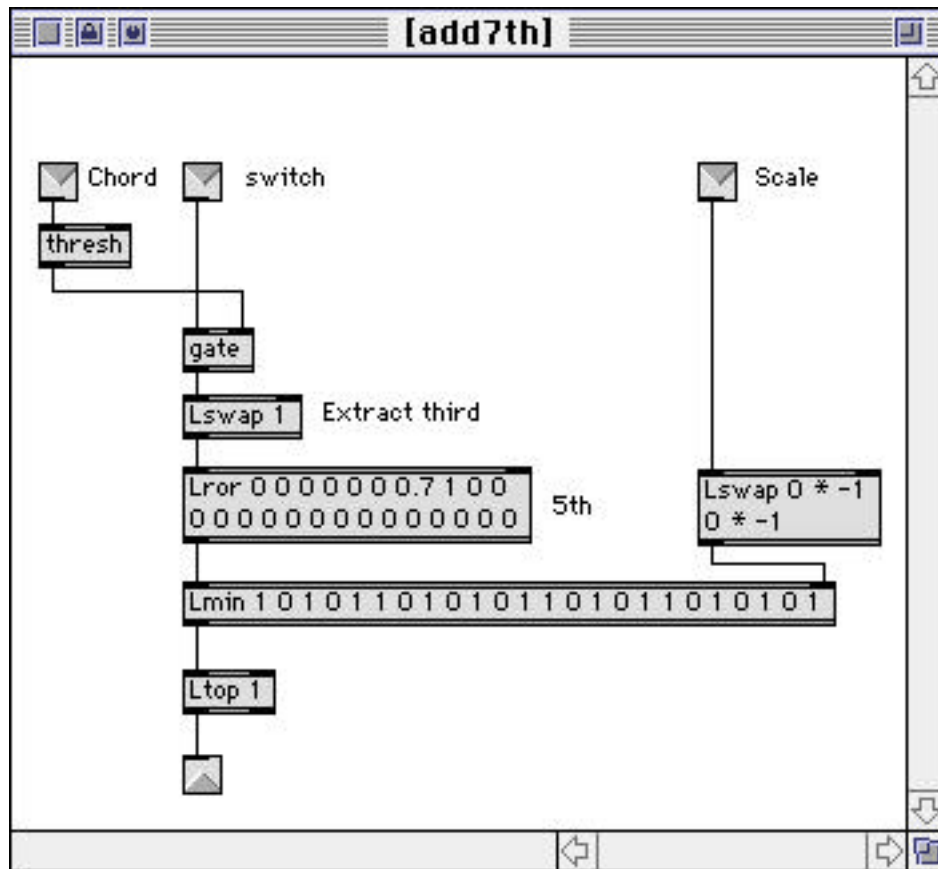
The chord is generated one note at a time. First the root is passed through, then it generates a third by rotating a fuzzy set, passing the result to Lmin with the scale. (Default scale shown is major. All of the sets cover two octaves to keep the final product in root position.) Ltop reports the position of the highest valued member of the solution set, or in the case of a tie, the first member with the high value.

The fifth is then generated in a similar manner. The three notes are output sequentially, not as a list. This mechanism will work on any of these scales:



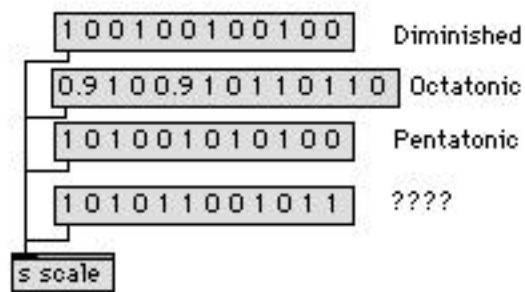
Note that the Harmonic minor is slightly fuzzy. The .9 membership of the leading tone encourages a major third in the sixth chord. The fuzzy minor scale contains both natural and harmonic minor elements. It shows a preference for the raised 7th to give a major V, but allows the lowered 7th to appear in the III. The raised 6 does not appear as the third or fifth of any chord (ii° occurs rather than ii), but it will generate a vi° when used as a root.

When desired, a seventh is produced by the add7th subpatcher:



The seventh is generated by creating a fifth above the third of the chord. (You can't just pick a seventh above the root, because the type of seventh required depends in part on whether the chord is major or minor.) This subpatcher demonstrates one of the advantages of fuzzy logic: once you have a basic problem solved, it is usually easy to add refinements. With other methods, any embellishment often means reworking from the start.

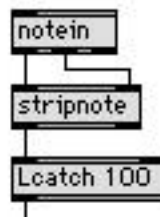
Although triadic harmony may not be appropriate, it can be entertaining to run other scales through this patcher, as for instance:



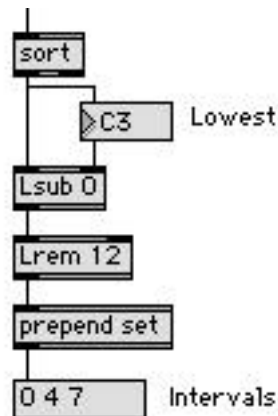
Identifying Chords

We often need to identify chords from a performance, either to trigger events or to produce an analysis. This can be difficult, since the chords will be in various inversions and the notes can occur in any order.

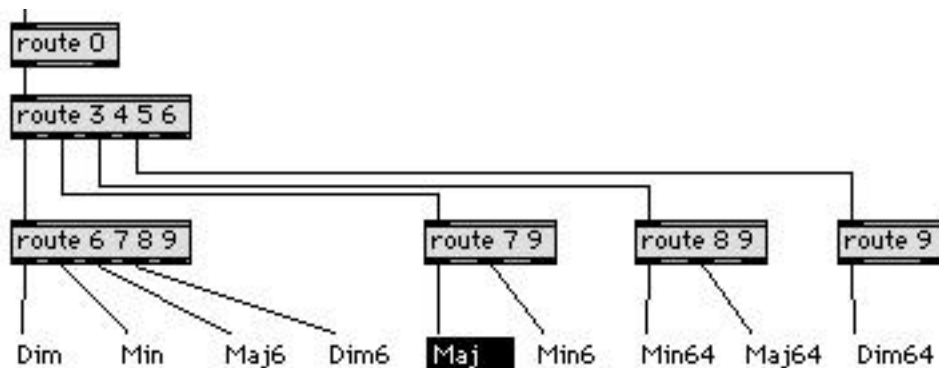
The first step is to decide what group of notes constitute a chord. The usual technique is to group notes that sound nearly simultaneously, with a pause afterwards. This is done with thresh or Lcatch, both of which hold values until there has been a pause of a defined duration, then output them as a list. You will find something like this at the top of most chord detectors:



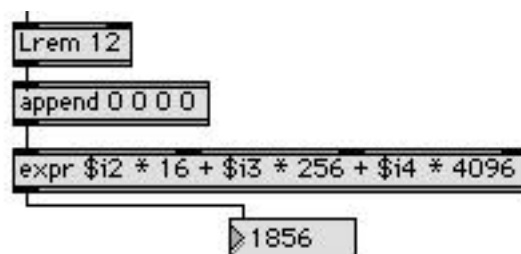
From here, there are a number of approaches. For instance, it is easy to transform the list of pitches into a list of intervals:



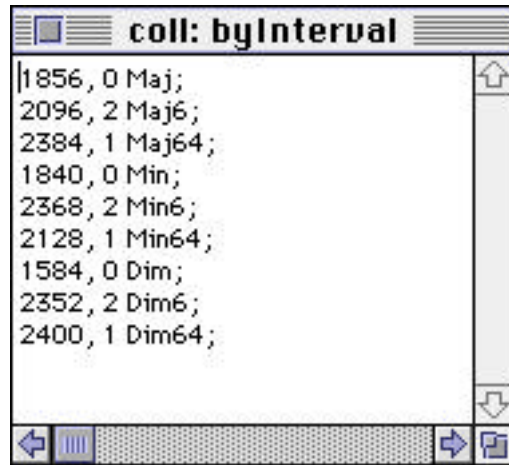
If you just need to watch for one or two types of chords these numbers can be used to route down to the answer, with something like this:



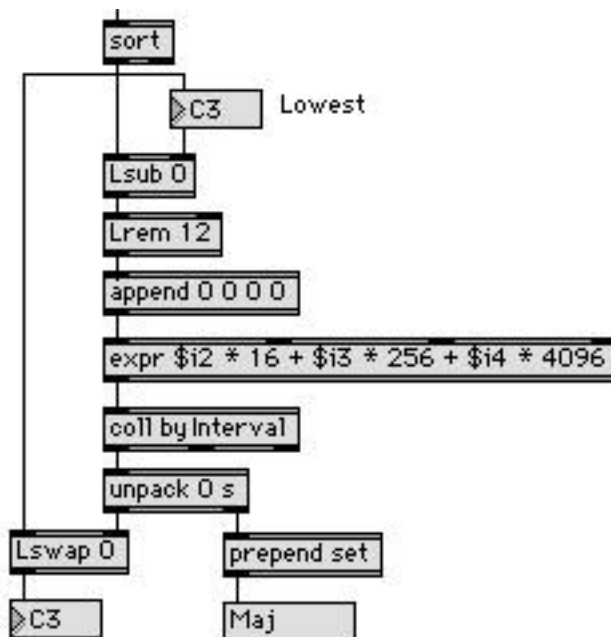
(The labels highlight with ubuttons when a chord is played.) This only tells us the quality, not the root of the chord, and it would get very complex if we tried to expand it to include all twenty varieties of seventh chords. An expandable approach is to use the intervals to address a coll. First the intervals are encoded into a single number, like this:



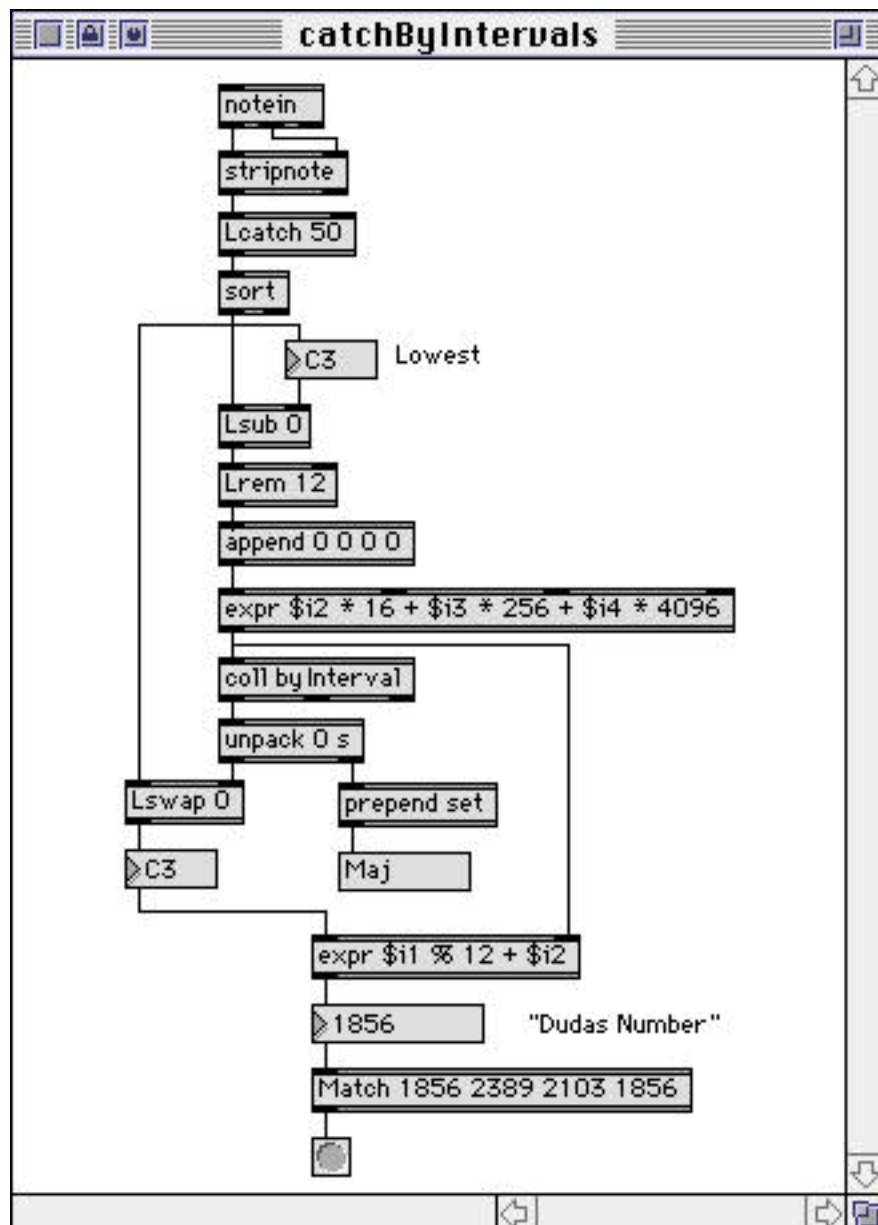
This uses four bits for each interval. If you display the resulting number in hex, you would see the intervals again. (I'm deliberately skipping the first four bits for the moment.) The number shown is what you get for a major triad. These numbers can be used as addresses in a coll, what they address is up to you. Here's one that has names for the triads:



It would be easy, if tedious, to expand the collection to include as many chords as you can think of. The number before the name tells the position of the root in that kind of chord. It can be used with Lswap to get the root from the original list of notes:



The major advantage of this method is that you get a single number for each type of chord. If you add in the root, the number is now unique for every chord and type. This can be used to efficiently search for chord patterns and respond to them. The bottom addition to the patch will bang if you play C maj, F maj64, G maj6, C maj:

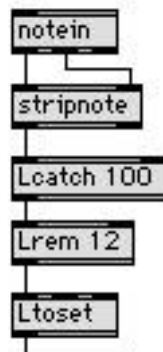


This method was suggested to the Max list by Richard Dudas, so I call it "Dudas Encoding".

Pattern Matching

The Dudas method is fast, but sensitive to inversions, which might be a disadvantage. It also would be hard to expand beyond seventh chords.

A more forgiving and open ended approach uses pattern matching. We intake the chord as before, and convert it into a set:



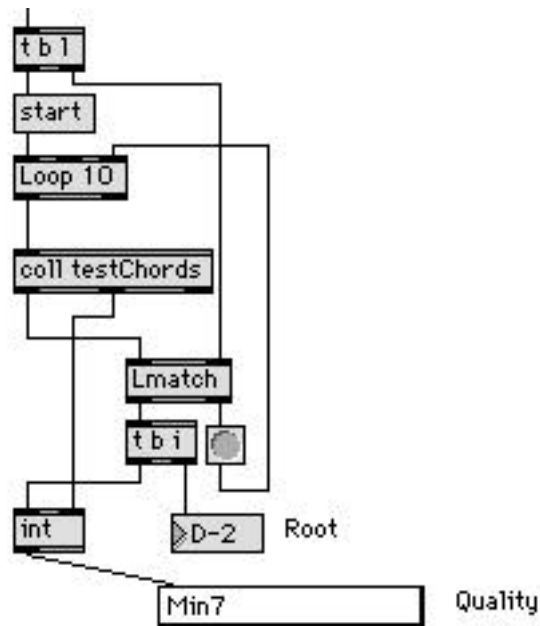
LtoSet transforms a list of numbers into a set of specified length (default is twelve). The values 0 4 7 would generate:

1 0 0 0 1 0 0 1 0 0 0 0

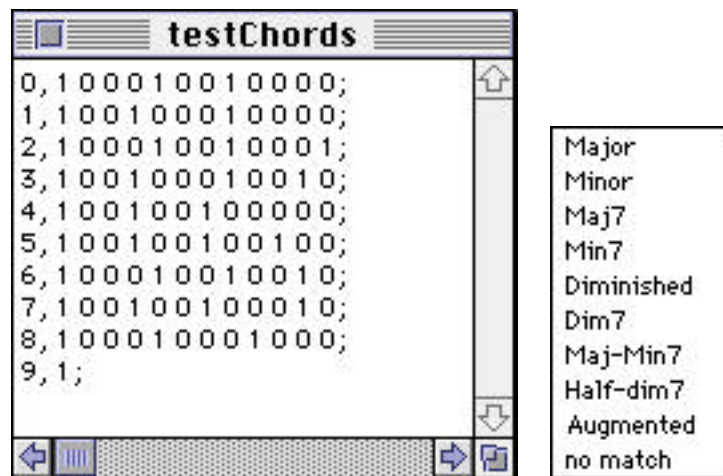
The ones appear in positions 0, 4, and 7, to give us an image of a C major chord. The order of the numbers in the original list does not matter. The set retains information about the quality and root, but not inversions. The rem operation wraps the chord around, so F major looks like this:

1 0 0 0 0 1 0 0 0 1 0 0

We are going to load this chord set into Lmatch, then feed in a series of test chords to see which one matches:



The test sets are in the coll, and the names are in a menu:



The Loop object will send these one at a time to Lmatch. Lmatch compares the test set with the unknown, starting at each possible position in order, like this:

```

unknown    001000100100
test       10001001
           10001001
match!    10001001
  
```

Lmatch would even find B major in this example, because it wraps the test set around when checking the last positions.

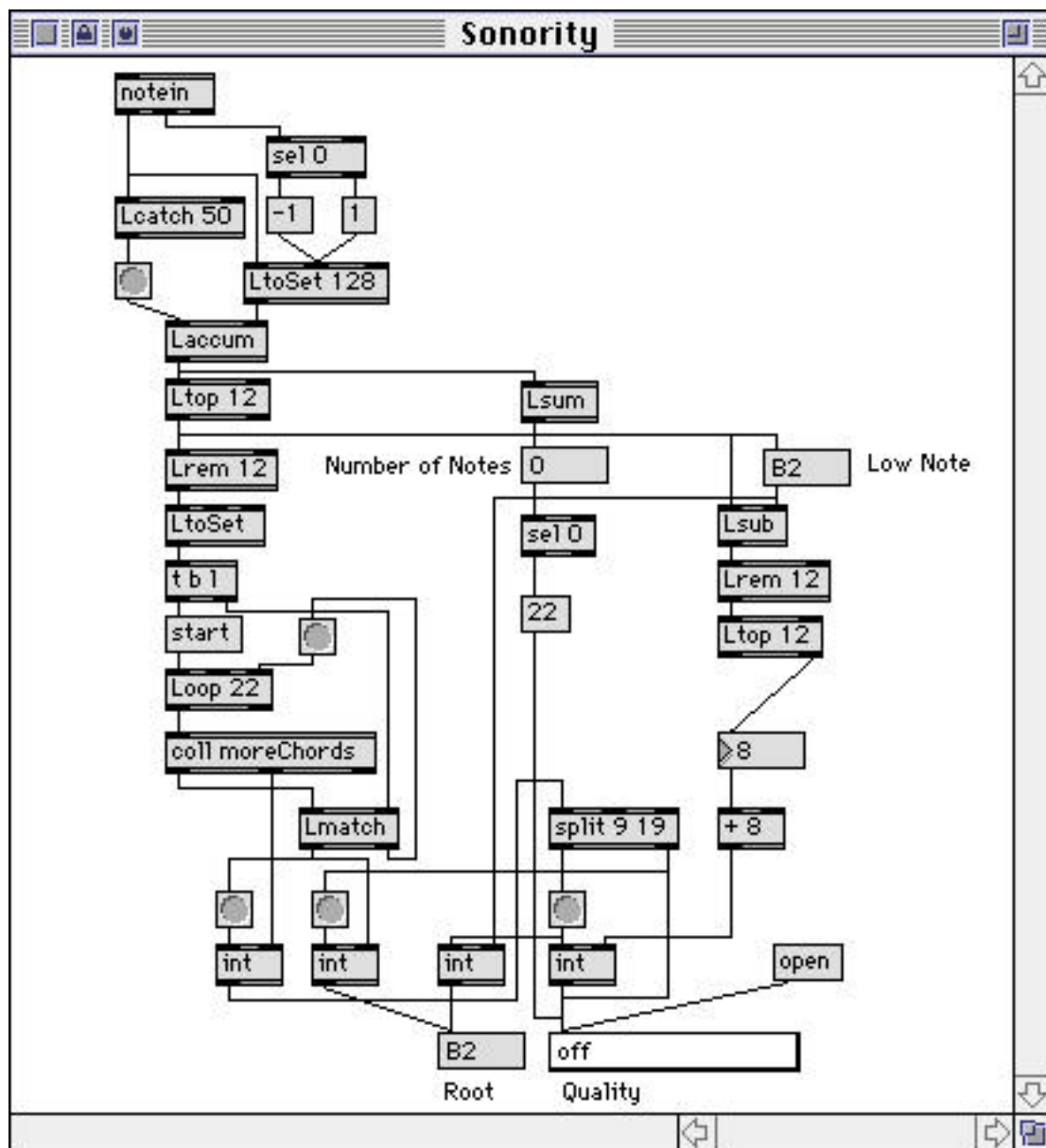
If a match is found on a particular test, the position of the match is reported at the left outlet. If it is not found, the test set is sent out the right outlet. In this patch, if the test fails, Loop is banged to get the next test out of the coll. When the match is found, the position is the pitch class of the root, and the address from the coll indicates the quality.

The order in which the tests are tried is unimportant. It can take an noticeable amount of time to do all of this checking, so overall performance is improved if the most common chords are listed first.

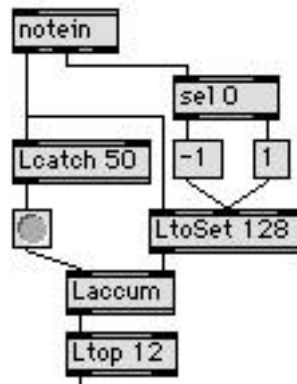
The root and quality are simply displayed here, but they could be encoded into a single value for matching chord progressions. You can easily expand the coll to include any chord imaginable, even covering partial chords.

Continuous Chord Analysis

The patchers presented so far respond to chords as they are played, defining a chord as a group of notes sounded together. Sometimes we need to watch harmonies that unfold slowly, and may include simple pitches or intervals. Here's a patcher that more or less combines everything we've shown so far, and keeps track of all currently sounding pitches:



Pitches are stored in the top section:



LtoSet responds to both note on and note off. Note on will generate a set of the type:

... 0 0 1 0 0 0 ...

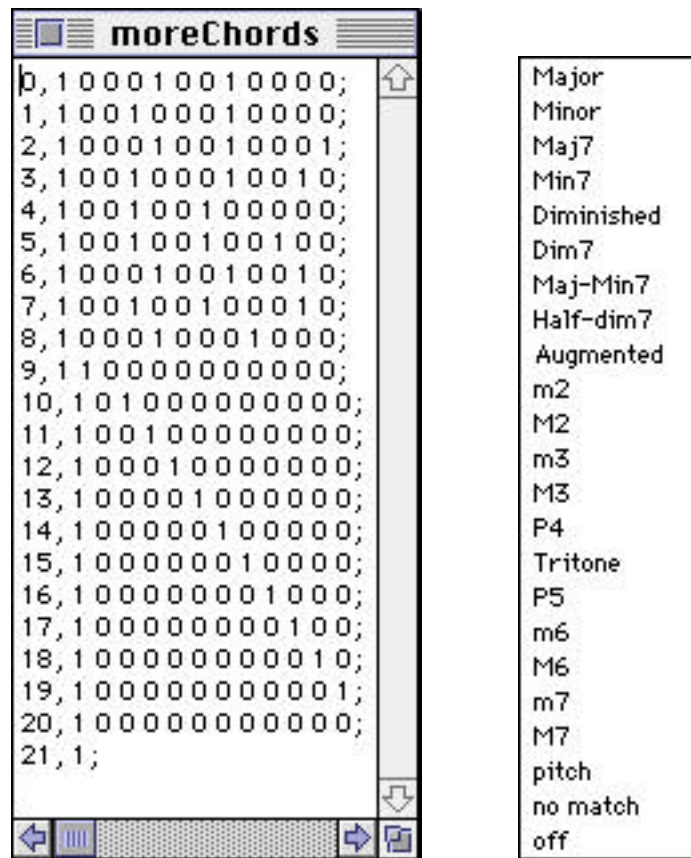
and note off will create its complement:

... 0 0 -1 0 0 0 ...

Of course, these sets have 128 members¹. The Laccum will have a set with a one representing each sounding note. This is sent out after each cluster of note messages. (The slight pause actually speeds up processing time. If Laccum reacted to every message, the patch would identify each interval of the chord separately before testing the chord.) Ltop converts the big set back into a list of pitches.

¹You have to have the new versions of the Lobjects to do this, the set distributed with Max 3.5. The older ones are limited to 64 members, and you would have to restrict the notes you were watching for.

The patch then splits into two parts. The left stack is the chordCatcher developed above, with more entries in the coll and menu:

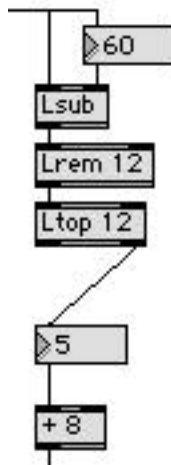


The screenshot shows a Pure Data patch window titled "moreChords". The patch contains a list of 21 lines of code, each representing a chord or interval. To the right of the patch is a menu box listing the corresponding chord names.

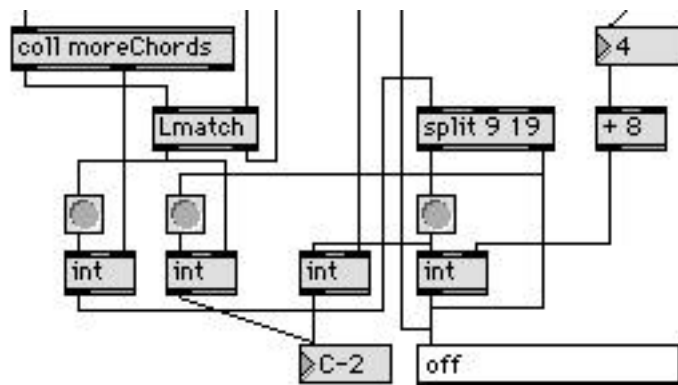
Line	Code	Chord Name
0	b, 1 0 0 0 1 0 0 1 0 0 0 0;	Major
1	1, 1 0 0 1 0 0 0 1 0 0 0 0;	Minor
2	2, 1 0 0 0 1 0 0 1 0 0 0 1;	Maj7
3	3, 1 0 0 1 0 0 0 1 0 0 1 0;	Min7
4	4, 1 0 0 1 0 0 1 0 0 0 0 0;	Diminished
5	5, 1 0 0 1 0 0 1 0 0 1 0 0;	Dim7
6	6, 1 0 0 0 1 0 0 1 0 0 1 0;	Maj-Min7
7	7, 1 0 0 1 0 0 1 0 0 0 1 0;	Half-dim7
8	8, 1 0 0 0 1 0 0 0 1 0 0 0;	Augmented
9	9, 1 1 0 0 0 0 0 0 0 0 0 0;	m2
10	10, 1 0 1 0 0 0 0 0 0 0 0 0;	M2
11	11, 1 0 0 1 0 0 0 0 0 0 0 0;	m3
12	12, 1 0 0 0 1 0 0 0 0 0 0 0;	M3
13	13, 1 0 0 0 0 1 0 0 0 0 0 0;	P4
14	14, 1 0 0 0 0 0 1 0 0 0 0 0;	Tritone
15	15, 1 0 0 0 0 0 0 1 0 0 0 0;	P5
16	16, 1 0 0 0 0 0 0 0 1 0 0 0;	m6
17	17, 1 0 0 0 0 0 0 0 0 1 0 0;	M6
18	18, 1 0 0 0 0 0 0 0 0 0 1 0;	m7
19	19, 1 0 0 0 0 0 0 0 0 0 0 1;	M7
20	20, 1 0 0 0 0 0 0 0 0 0 0 0;	pitch
21	21, 1;	no match
		off

The chords are the same as before. Line 20 will detect a single pitch (or octaves), and the single 1 at line 21 indicates something is sounding, but this patch couldn't figure it out.

Lines 9 through 19 represent simple intervals. Only half of them can actually be detected with this method, since it ignores inversions (seems everybody has trouble telling M6 from m3), so these are just place keepers. Intervals are calculated by the right side of the patch:

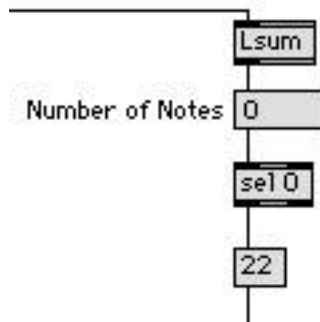


This calculates all of the intervals being played, but we are only interested in the first one that is not 0. (We want to ignore octaves). The right outlet of Ltop gives us a list without 0s, and the number box strips off the extras. Eight is added to the interval to make a pointer for the menu object at the bottom of it all:



The split and the various ints steer the proper values to the number box and the menu, taking them from the right side of the patch when intervals are detected.

There is one more state to detect, silence. The left side of the patch does not react to the last note off, because an empty set does not cause any output from Ltop. This section in the middle keeps track of the number of notes playing (always nice to know):



Lsum just adds the values in the Laccum, which has a one for each current note. When the last note shuts off, the number 22 is sent down to the menu to display the off message.