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:

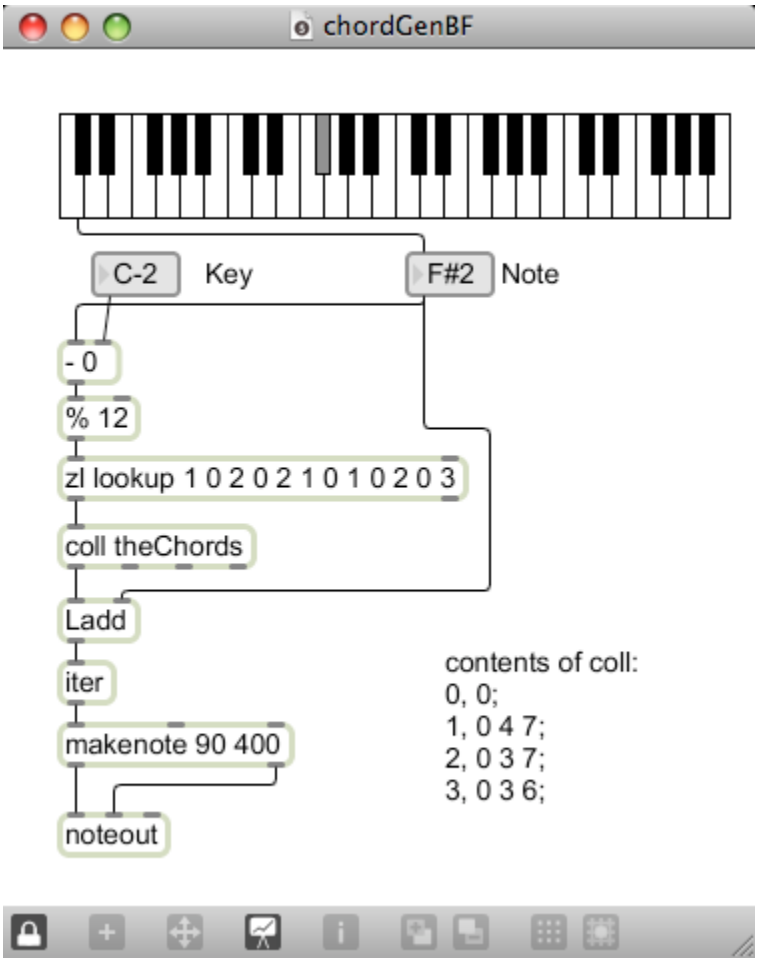

Figure 1.

This "brute force" approach simply picks the desired chord type out of a coll and builds it on a given root. The coll contains the intervals for major, minor, diminished, and augmented chords. They are added to the root by Ladd[1] and played. (Notice the iter required to play chords with makenote. Makenote interprets a list as a single note.)

[1] Ladd is one of my Lobjects, available from ftp://arts.ucsc.edu/pub/ems. It does more or less the same thing as vexpr $i1 + $i2 @scalarmode 1, but is easier to type.

Instructions as to which chord to play are contained in zl lookup. 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), no harmony on the sharped first minor on the second, and so forth. The pointer into zl lookup 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 zl lookup. This list:

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

Harmonizes minor pretty well.

To include sevenths, we could expand the coll:

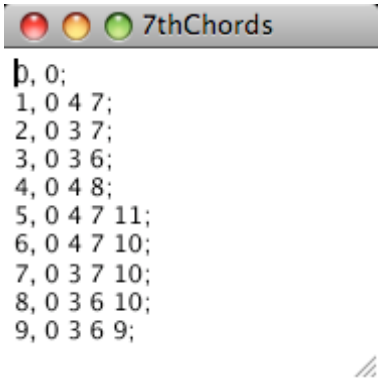

Figure 2.

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. The addition to the bottom of the patch shown in figure 3 will do the trick:

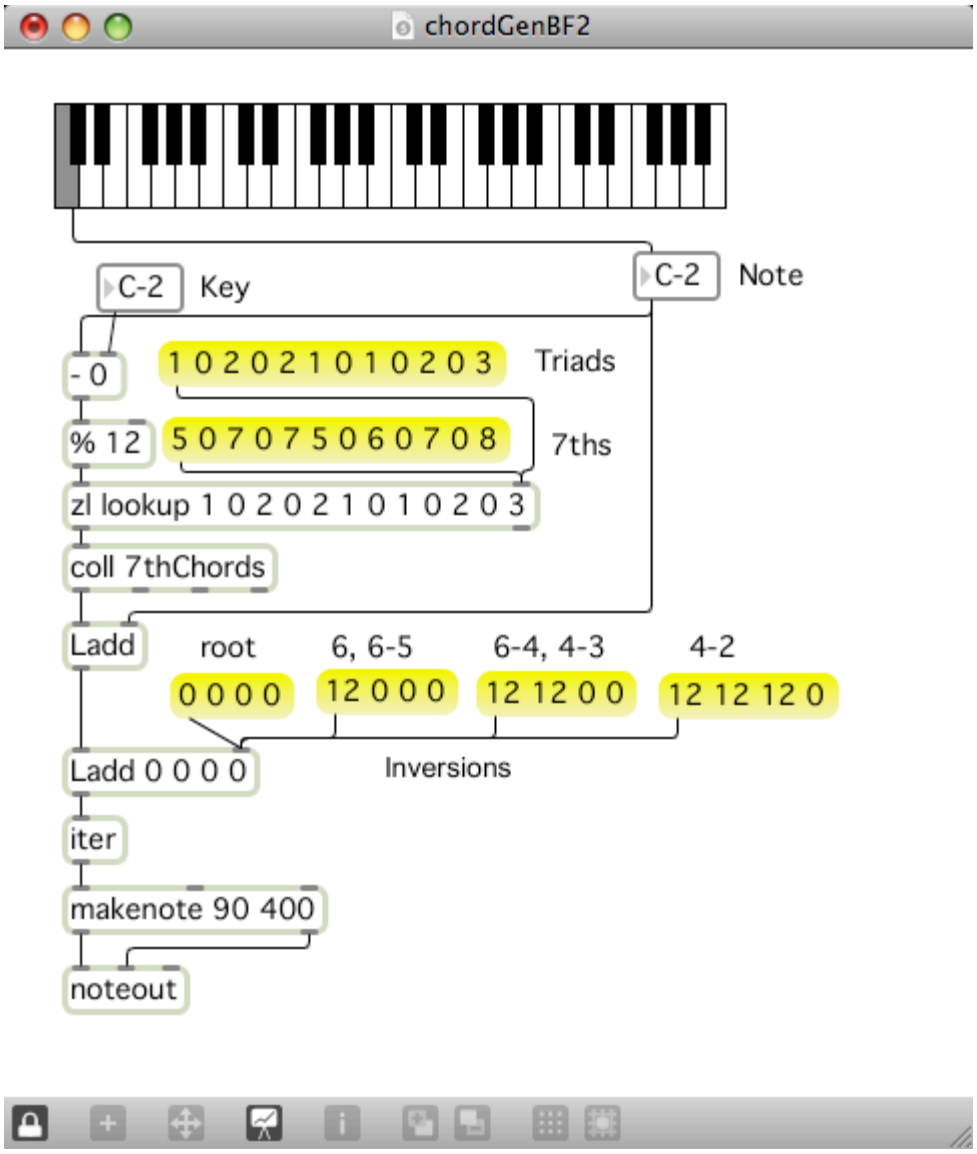

Figure 3.

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. The patch in figure 4 simply takes alternating notes out of scales.

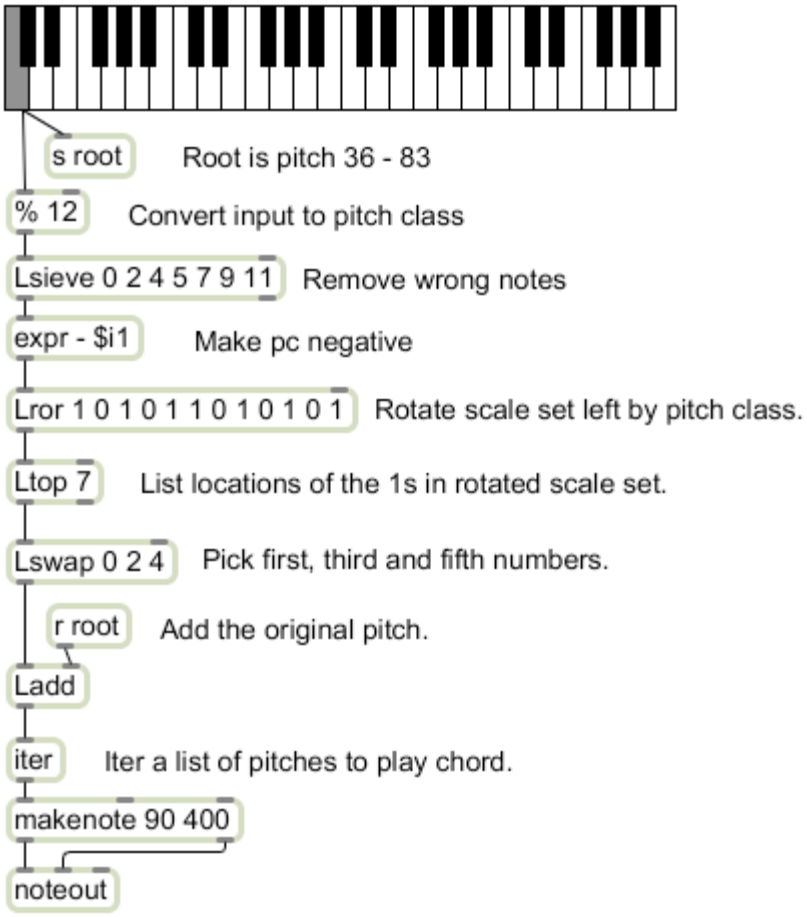

Figure 4.

The patch starts with a midi note number from 36 to 83. This is converted to pitch class, and notes not in the C major scale are filtered out[2].

The chord will be derived from a scale set. Scale sets are 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. You could think of this as a picture of a keyboard, with 1 marking the notes in the scale. The C scale is notated as a set like this: {1 0 1 0 1 1 0 1 0 1 0 1}[3].

To generate a chord, we rotate the scale set to the left till the root of the chord is in the first position. {1 0 1 0 1 1 0 1 0 1 0 1} rotated four steps is { 1 1 0 1 0 1 0 1 1 0 1 0}.

[2] Lsieve is covered in "Max & Pitch"

[3] This is not music set theory. These procedures are based on a branch of math called fuzzy logic.

Lror[4] produces a snapshot of the notes from E to D# in the C major scale. The E chord is the first, third and fifth notes in this set. Next we need the locations of these notes.

Ltop is an Lobject written with this trick in mind. It produces a list of the locations of the highest values in a list. (Also the values themselves, but we aren't interested in that here.) If I ask for the locations of the highest 7 values in a list of zeros and ones, I'll get the locations of the ones. The ones in the rotated set above are in positions[5] 0 1 3 5 7 8 and 10. Ltop 7 will generate that list.

Lswap is an Lobject that extracts particular members from a list. Lswap 0 2 4 extracts the first, third and fifth items, in this case 0 3 7. These three numbers are added to the original pitch to produce an E minor chord.

Generating Chords From The Third Or Fifth

Of course, any note may be harmonized three ways. It can serve as the root, third or fifth of the chord. The core technique of the above patcher can be used to find alternate roots. Just change the template in Lswap:

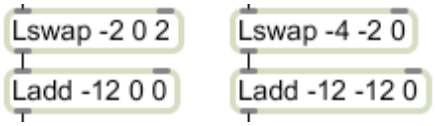

Figure 5.

A negative argument to Lswap means count backwards from the end of the list. -1 is the last item, -2 is the item before that. Lswap -2 0 2 applied to the list 0 1 3 5 7 8 will give 7 0 3. These are the correct values to make a C chord, but out of order. We need to drop the 7 an octave, which can be done by the Ladd shown. The second inversion is built in a similar manner. This technique can easily be expanded to produce 7th chords.

Choosing Inversions

With a variety of chords available for any pitch, the next challenge is making a sensible choice. This can lead to some very complex patches, as chord choice is determined by a variety of factors from harmonic motion to difficulty of fingering. Figure 6 illustrates a basic approach that can be expanded in many ways.

[4] Lror rotates lists right, but a negative input will produce a left rotation.

[5] All of the Lobjects refer to members of lists by index numbers, which start with 0. The index of the last item in a 12 member list is 11. Once you get used to it, the math really is easier than counting from 1.

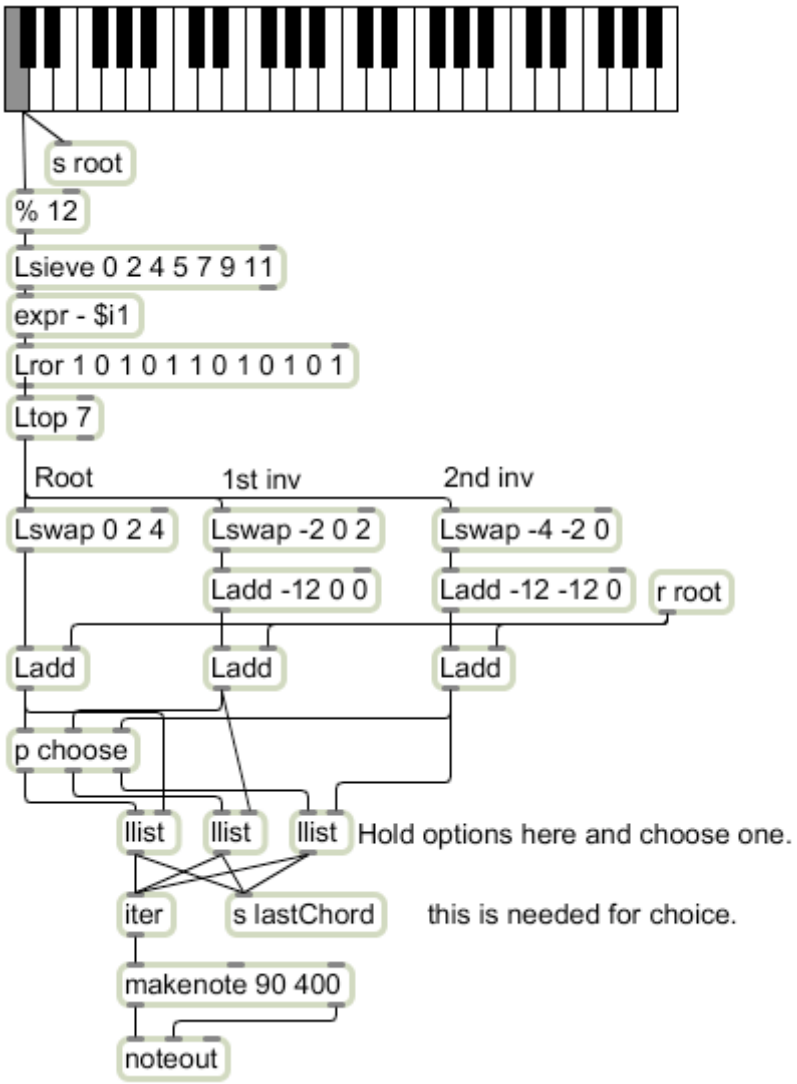

Figure 6.

The top half of this patch is identical to the previous example. Additions include the chord inversions, which are stored in llist[6] objects until a choice is made. Then one will be banged and played. Note that the chosen chord is also sent somewhere as "lastChord".

The decision is made in the choose subpatch shown in figure 7. This selects the inversion that has the most notes in common with the previous chord. The process is based on an Lobject named Lbag. Lbag is very similar to bag but accepts lists of ints and sends its output as a list. In this application, Lbag is set to "store" mode by the loadmess 1. When the lastChord comes in, all three lbags are cleared, then loaded with the chord pitches. On the next note, each Lbag gets one of the candidate chords and dumps its contents in a list. Since Lbag does not duplicate values, inversions with notes in common with lastChord will result in shorter lists. If the list has 6 members it will have no common notes, so

[6] Llist is an Lobject that provides the same service for lists that int and float provide for numbers: convenient temporary storage. You can now use zl reg.

subtracting the length of the list from 6 will show how many notes are in common. These results are packed into a list to be compared by Ltop, which will report the position of the highest value.

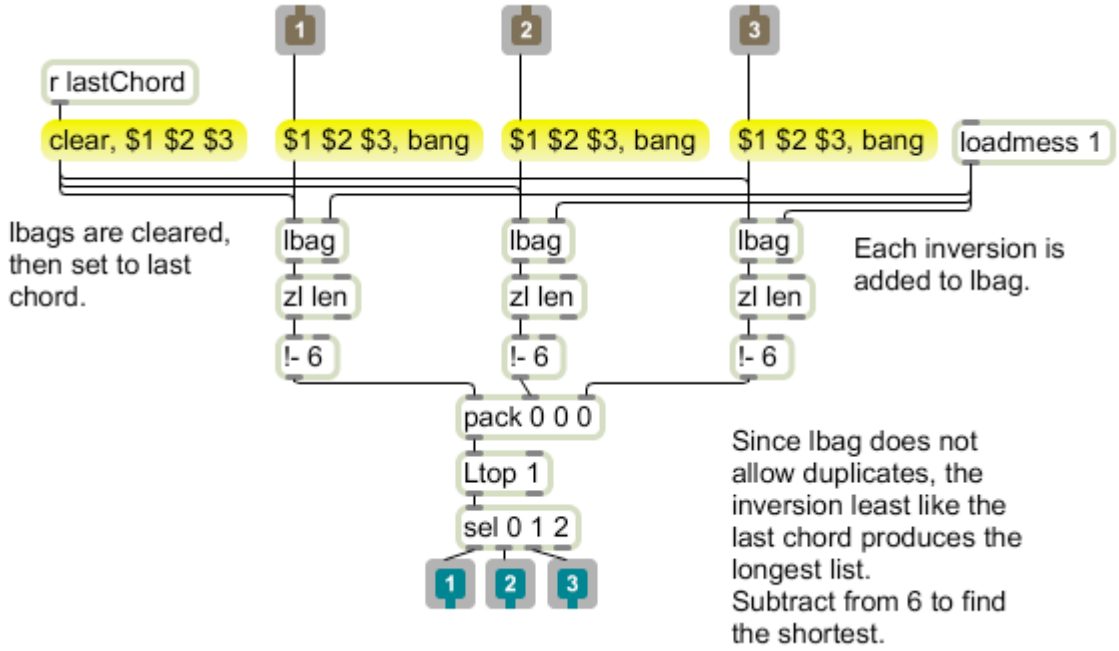

Figure 7.

These numbers show the patcher in action. The left column is the input note, the right the chord produced.

60	60 64 67
64	60 64 67
62	62 65 69
65	62 65 69
67	67 71 74
71	67 71 74
60	60 64 67

Figure 8.

This is not the ultimate in harmonizers, but the modular design of the patch makes it easy to make more sophisticated choices. Simply replace the choose subpatcher.

Changing Key

These patches are limited to C major, but the use of scale sets makes it easy to modify them to work in any key. The modifications are additions to the top portion shown in figure 9.

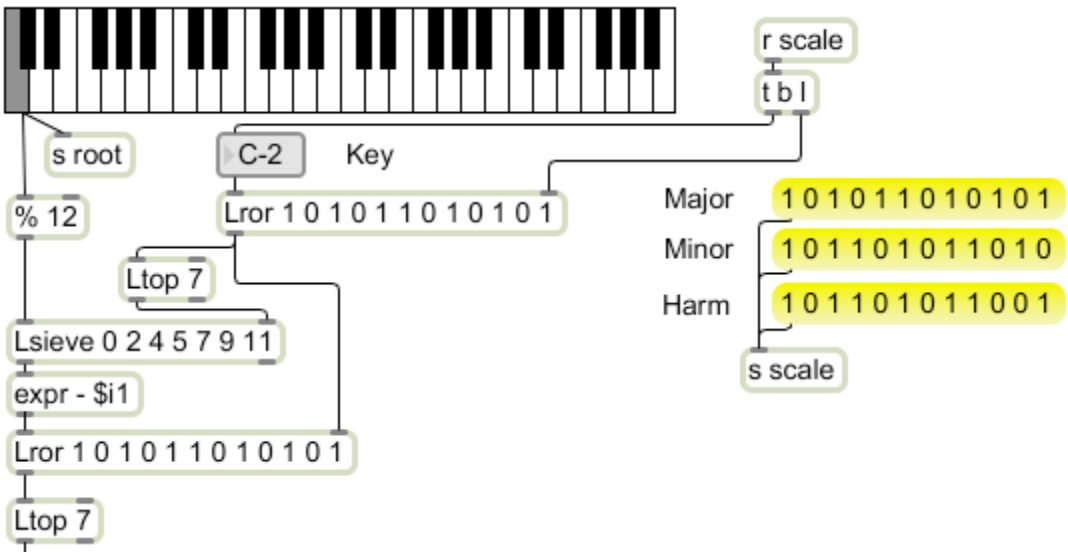

Figure 9.

Setting the number box labeled "Key" will create a new set for the Lsieve and lower Lror objects. Note that this is the same mechanism used to build a chord on any note. Here it changes the starting place.

Clicking on a scale list will change the pattern of notes to match major, minor or harmonic minor. The trigger object will provide a bang to the key value to update the set. Of course, this is a very simplistic approach to minor harmony. A more satisfactory minor harmonizer would probably generate chords from both minor scales and choose according to factors such as melodic motion.

This is only one of many possible approaches to generating harmony. For further development of this technique, see my essays on music and fuzzy logic.

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 figure 10 at the top of most chord detectors:

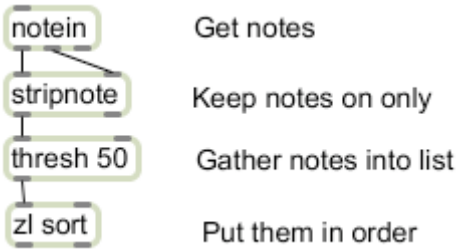

Figure 10.

From here, there are a number of approaches. For instance, it is easy to transform the list of pitches into a list of intervals as shown in figure 11.

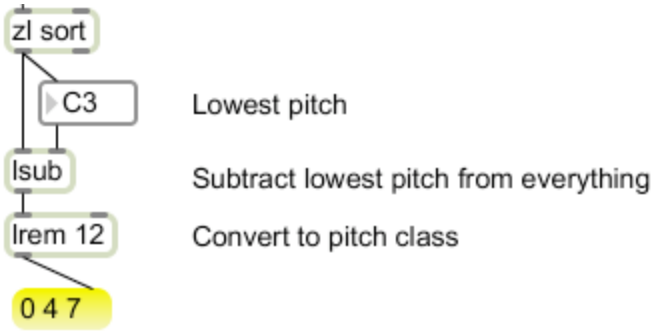

Figure 11.

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 figure 12.

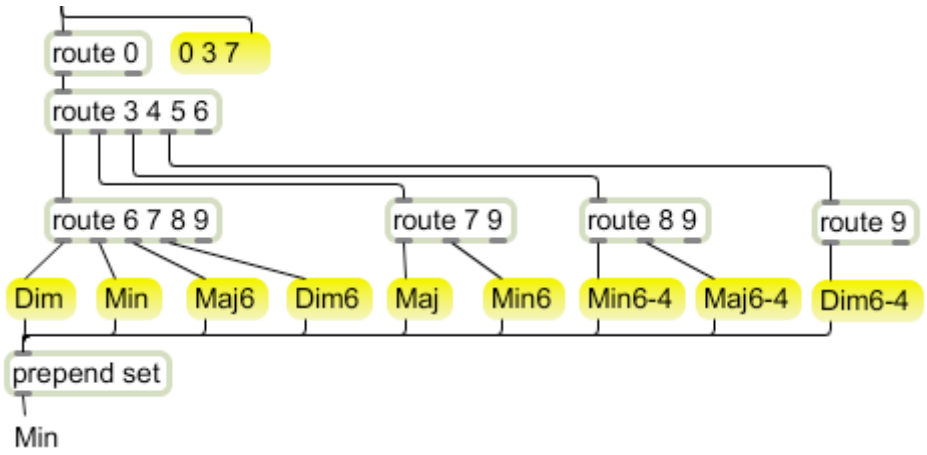

Figure 12.

Figure 12 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:

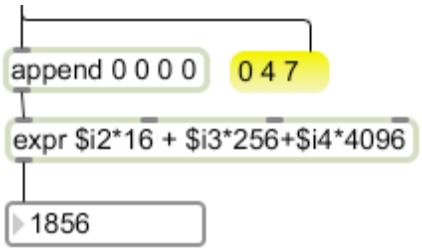

Figure 13.

Figure 13 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:

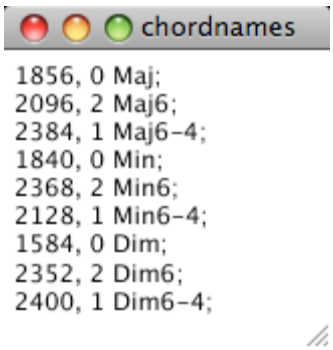

Figure 14.

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, as shown in figure 15.

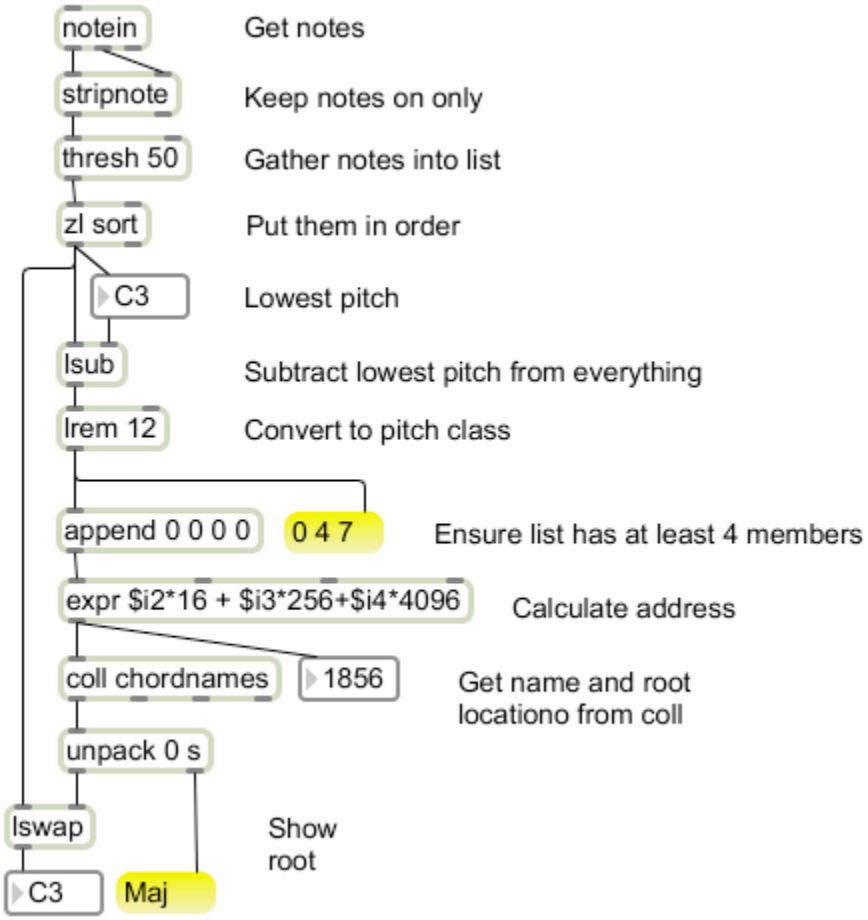

Figure 15.

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. 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. The section in figure 16 uses lreg to provide an up to date list of playing notes. Lreg works directly with note on and note off messages, and outputs a list of playing notes whenever the status changes. (There is a bang from the right outlet when all keys are released.) Lrem converts all notes to pitch class and Lunique removes duplicates.

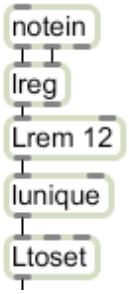

Figure 16.

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 (5 9 0) looks like this:

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

The hard work will be done in an Lmatch object. Lmatch finds where a list occurs in another list. It first compares the input list with the beginning of the stored list. If that is not a match, it tries again starting with the second element of the stored list, and so on. When the input list extends past the end of the stored list, the comparisons wrap around to the beginning. When a match is found, the position of the matching part of the stored list is output. (If no match, the input is sent out the right outlet, so these can be cascaded.)

Figure 17 shows how this is managed. The test sets are in a coll, and are dealt out by an uzi object.

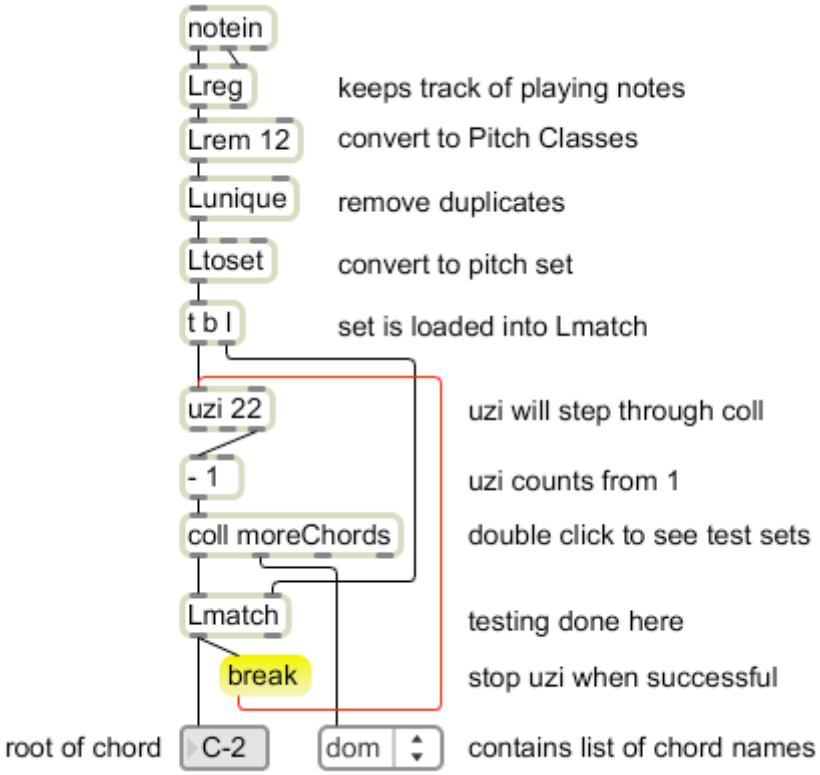

Figure 17.

Lmatch compares the test set with the unknown, starting at each possible position in order, like this:

```
unknown       0 0 1 0 0 0 1 0 0 1 0 0
test           1 0 0 0 1 0 0 1
test again       1 0 0 0 1 0 0 1
match!      1 0 0 0 1 0 0 1
```

When the match is found, the position is the pitch class of the root, and the address from the coll indicates the quality. Figure 18 shows the test sets in the coll. Note that the four note chords are tested first, followed by triads. 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. 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.

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.

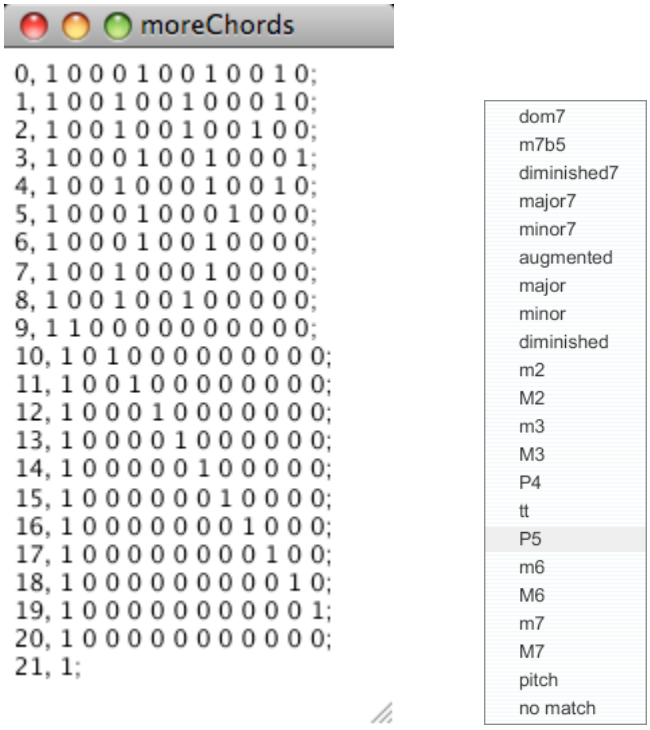
moreChords
0, 1 0 0 0 1 0 0 1 0 0 1 0;
1, 1 0 0 1 0 0 1 0 0 0 1 0;
2, 1 0 0 1 0 0 1 0 0 1 0 0;
3, 1 0 0 0 1 0 0 1 0 0 0 1;
4, 1 0 0 1 0 0 0 1 0 0 1 0;
5, 1 0 0 0 1 0 0 0 1 0 0 0;
6, 1 0 0 0 1 0 0 1 0 0 0 0;
7, 1 0 0 1 0 0 0 1 0 0 0 0;
8, 1 0 0 1 0 0 1 0 0 0 0 0;
9, 1 1 0 0 0 0 0 0 0 0 0 0;
10, 1 0 1 0 0 0 0 0 0 0 0 0;
11, 1 0 0 1 0 0 0 0 0 0 0 0;
12, 1 0 0 0 1 0 0 0 0 0 0 0;
13, 1 0 0 0 0 1 0 0 0 0 0 0;
14, 1 0 0 0 0 0 1 0 0 0 0 0;
15, 1 0 0 0 0 0 0 1 0 0 0 0;
16, 1 0 0 0 0 0 0 0 1 0 0 0;
17, 1 0 0 0 0 0 0 0 0 1 0 0;
18, 1 0 0 0 0 0 0 0 0 0 1 0;
19, 1 0 0 0 0 0 0 0 0 0 0 1;
20, 1 0 0 0 0 0 0 0 0 0 0 0;
21, 1;
dom7
m7b5
diminished7
major7
minor7
augmented
major
minor
diminished
m2
M2
m3
M3
P4
tt
P5
m6
M6
m7
M7
pitch
no match

Figure 18.