Max and Pitch

Representation of Pitches in Max

In the Max environment, pitches and durations are necessarily represented as numbers, typically by the MIDI code required to produce that pitch on a synthesizer. We must begin with and return to this representation, but for the actual manipulation of pitch data other methods are desirable, methods that are reflective of the phenomena of octave and key.

A common first step is to translate the midi pitch number (mpn) into two numbers, representing Pitch Class (pc) and octave (oct) this is done with the formulas:

$$\text{oct} = \text{mpn} / 12$$
$$\text{pc} = \text{mpn} \% 12$$

The eventual reconstruction of the mpn is done by

$$\text{mpn} = 12*\text{oct} + \text{pc}$$

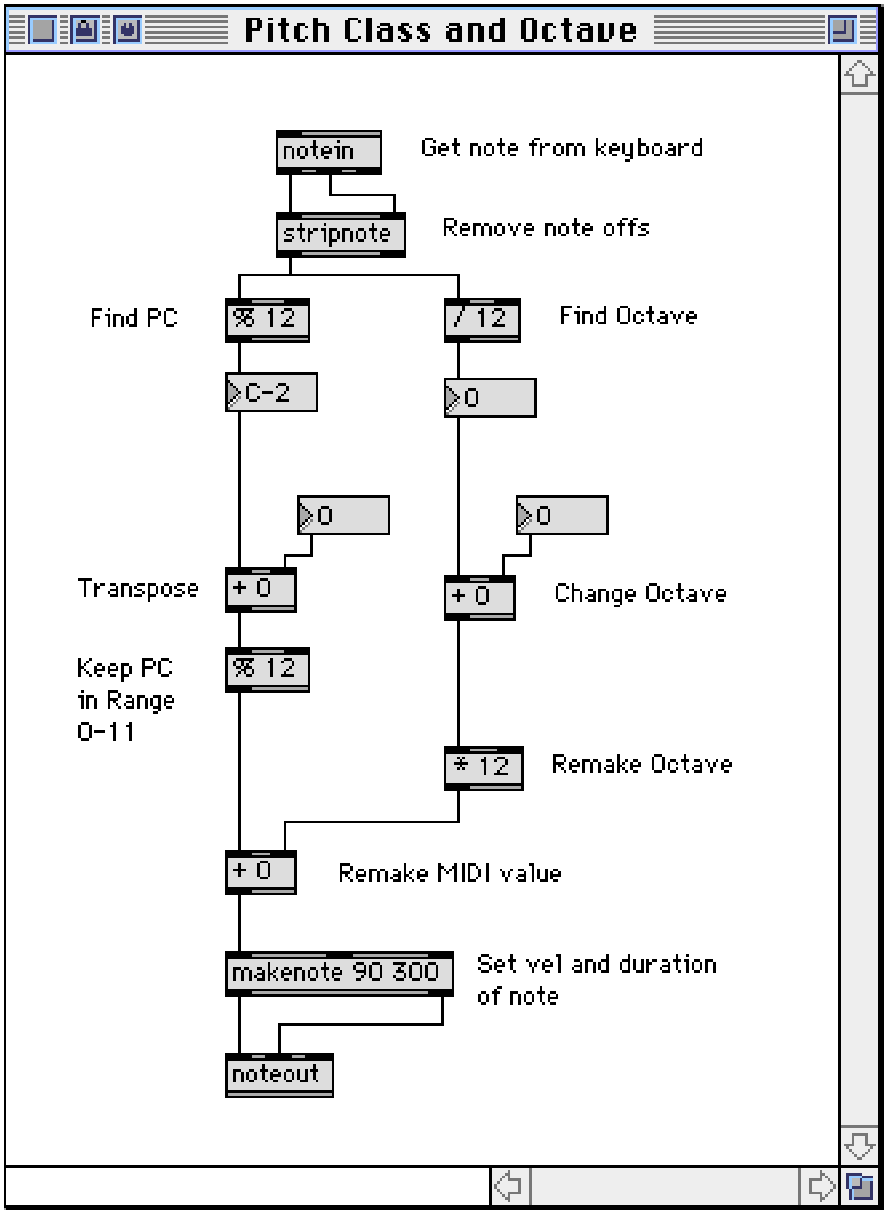
In this system pc can take the values 0 - 11, in which 0 represents a C. Oct typically ranges from 0 to 10. Middle C, which is called C3 in the MIDI literature and C4 by most musicians, is octave 5 under this convention.

Once the pc is split from its octave, a variety of manipulations can be undertaken. For instance, to transpose, you add the appropriate number of half steps. To go up a fifth (7 steps) from D(pc=2)

$$\text{new pitch} = (\text{old pitch} + \text{steps}) \% 12$$

gives 9 (A) as the answer. The modulus 12 is necessary to keep the answer within the range of 0 to 11.

To transpose down, you add the 12's complement (12-n) of the number. Down a fifth starts with the complement of 7 (5) but is otherwise the same as above. A fifth below D comes out (2+5)%12 or 7 (G).



This patcher illustrates the principles of extracting and manipulating pitch classes.

Generating Pitches

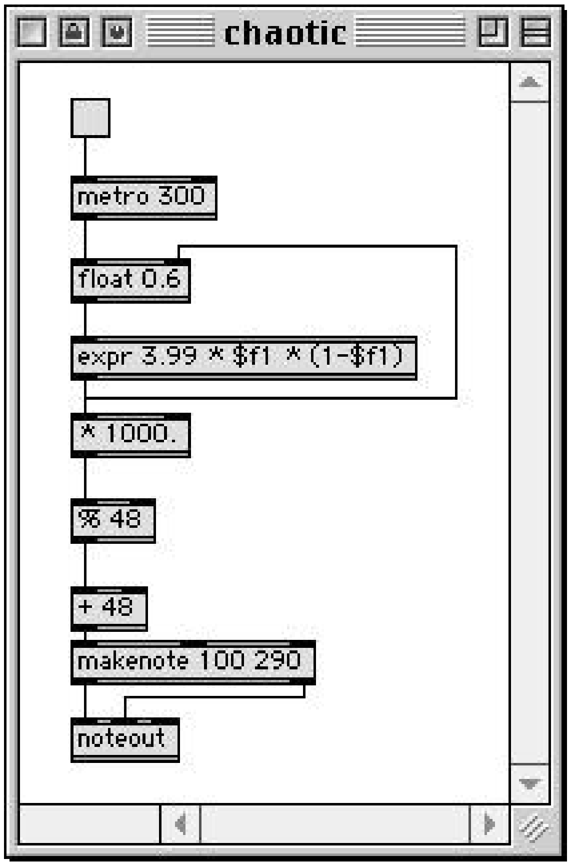
The notein object is not the only way to create notes in Max. Here are some strategies to play with:

The **random** object will create apparently random numbers. This is gaussian or "white noise" type of randomness. This is a good starting point for further processing, but random pitches taken straight are not very interesting. Actually, the random object will produce exactly the same series of "random" values every time you open the patcher. This is because it is impossible to calculate truly random numbers. (Think about it.) if you give random the message "seed X" , you will produce a different pattern for each value of X. The **date** object triggered by a **loadbang** can give you an X that will probably be different each time the patcher opens, so you can insure that the patterns are different.

The **urn** object selects from a series of numbers in a random order. When all have been output, the right outlet bangs to tell you.

The **drunk** object executes the "random walk" procedure. In this, the output is a random distance from the last output. The noise is "Brownian", and is sometimes interesting, but pitches repeat a lot.

There are a lot of ways to generate fractal note patterns. There's no object per se, but **expr** allows you to use any of the classic fractal formulas. Here's a patcher using one:

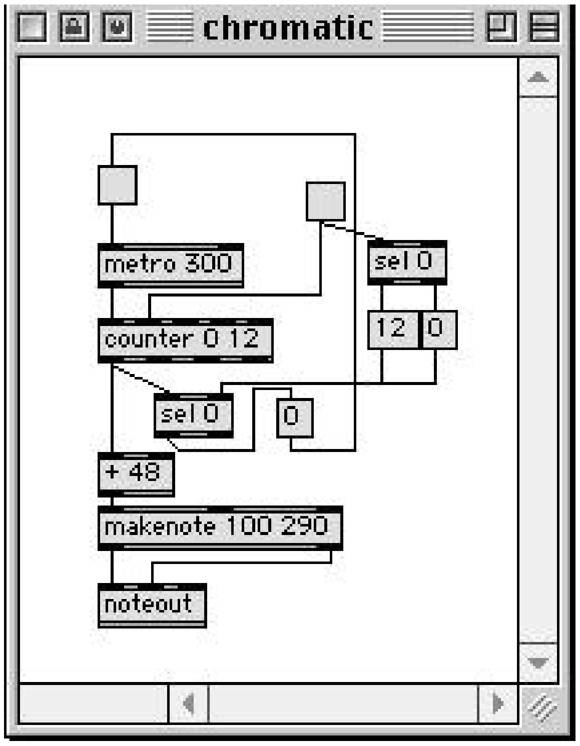


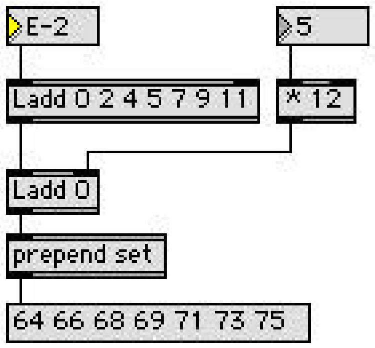
This will always play the same pattern of notes, but that pattern defies description. The pattern you get depends on the contents of the float object. If you "seed" it with a random number between 0 and 1 each time the patch is loaded the results will always come out different.

We multiply by 1000. (note the decimal) because the interesting patterns are in the right end of the fraction. The %48 reduces the range to 4 octaves.

Generating Scales

You can make chromatic scales with the **counter** object. It has inlets for direction as well as beginning and ending values so you can make very complex arabesques. The patcher shown just gives the basics, a scale up or down. The select objects are there because a long standing bug in counter makes it difficult to detect the end of the count. Instead of using the overflow and underflow outlets, you have to detect the last number of the count series and use that to turn the metro off.





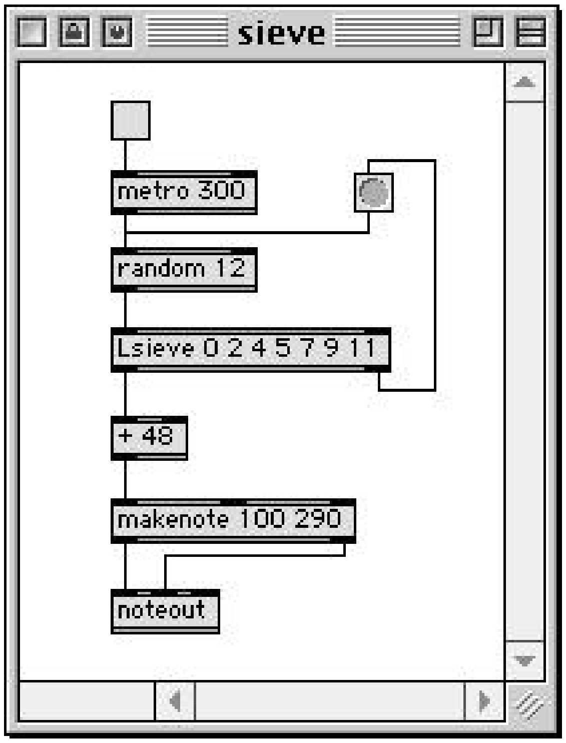
A major scale is represented by the numbers 0 2 4 5 7 9 11. If we call this a set in the mathematical sense, a number of useful manipulations suggest themselves. For instance, we can shift the scale to any octave just by adding a multiple of 12. We can transpose it to any key by adding the pitch class that represents the key desired:

Processing Random Pitches

Randomness is most effective when tamed by some musical rules. A very powerful rule type is the "sieve", which lets some notes through, but rejects others. The **Lsieve**[1] object does this handily;

Lsieve 0 2 4 5 7 9 11

will only allow the notes of C major through. In this patcher:

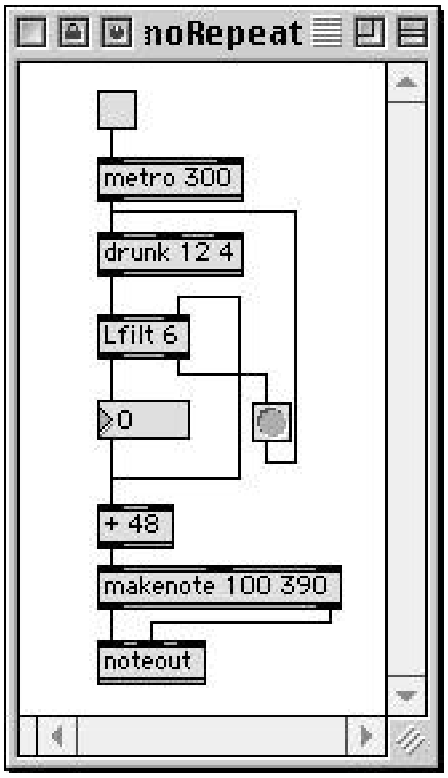


Any note that fails the test will cause the random object to try again, because failed values fall out the right outlet. You need to be very careful when using this type of feedback process. If Lsieve were to reject everything random puts out (for instance if it had all values higher than the range of the random object) the patcher would go into an endless loop and a stack overflow would occur. A safer approach would be to omit the feedback (leaving holes in the stream of notes) or to trigger some other process to create a note.

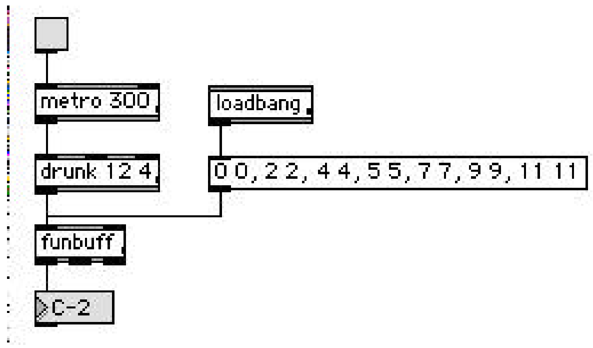
The secret to generating an interesting piece with sieves is to make the sieves change in some way. The values accepted by Lsieve can be changed by sending a list in the right inlet.

[1] You won't find Lsieve in the main Max documentation. That's because I wrote it myself. It is documented in the Lobjects folder along with all the other L(somethings), banger, and unlist.

The Lfilt object has a complimentary action. It will reject whatever is in its argument list. The values to reject can be changed by sending a list to the right inlet. What does this patch do?

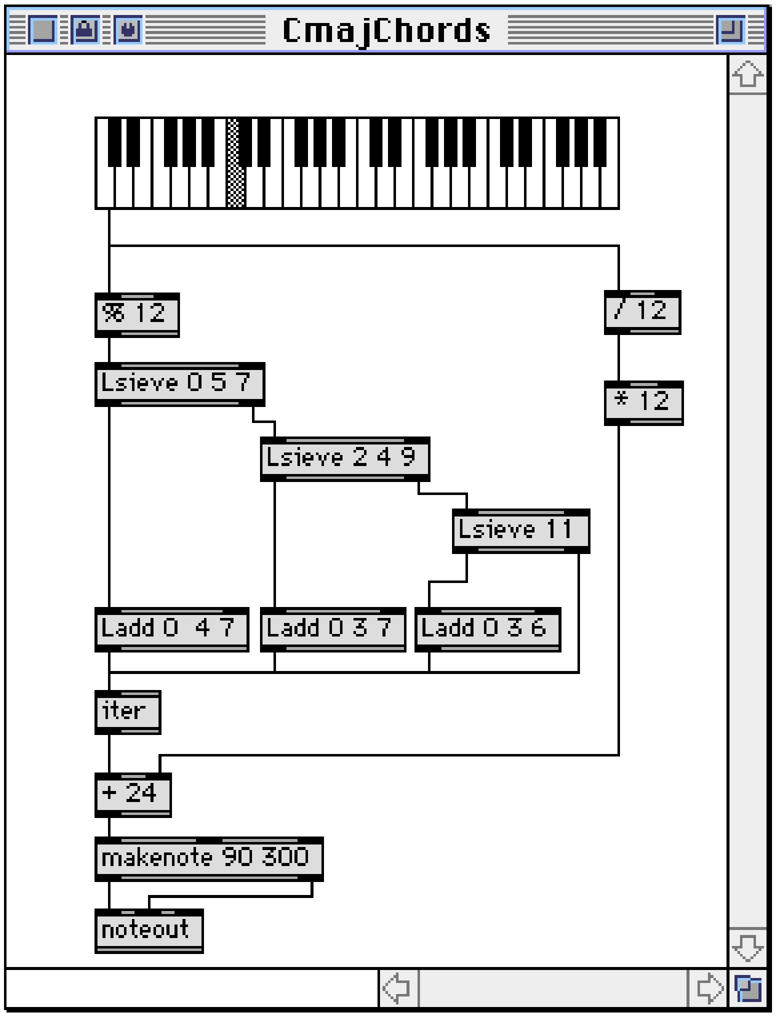


The Lsieve and Lfilt objects work by throwing data away. Another approach to the constraint problem is to change unwanted data somehow. A simple way to do this is with the funbuff object. The funbuff stores a series of pairs (that is two member lists). Once the pair has been input, the first value in the pair will be replaced by the second. If an input value is not in the funbuff, the next lower input value that is in there will be used. This patcher will keep everything in C major:



Chords

Playing block chords is easy in Max, all you have to do is send the individual notes to makenote at the same time. Of course, all actions in max are really carried out one at at time, but this can occur so fast that chords sound simultaneous to our ears. The main difficultly is figuring out whether to send a major or minor chord for some scale degree. Here is a basic approach:



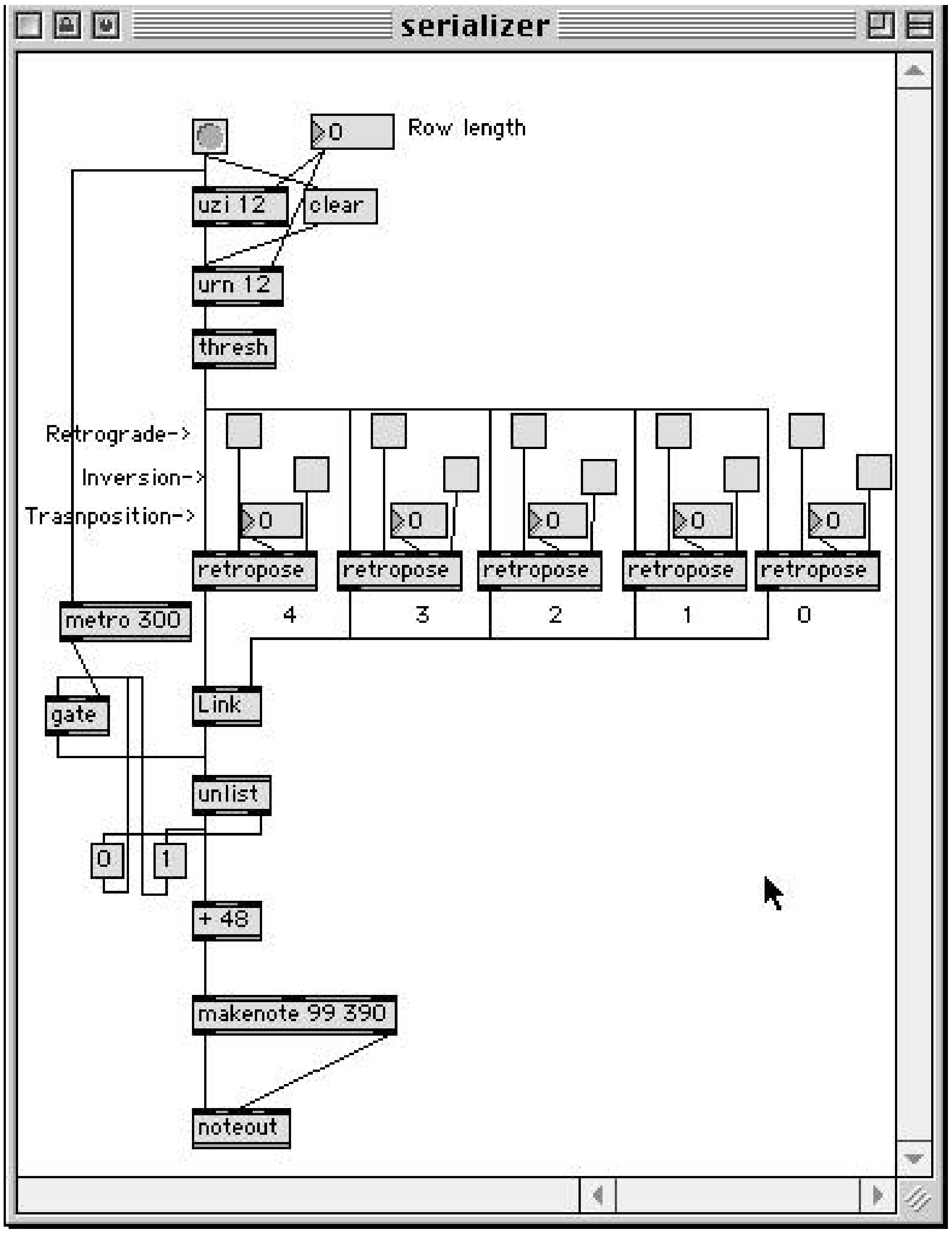
The Lsieve objects sort the scale notes according to the chords they should have in Cmajor. A "wrong" note is played, but doesn't get a chord.

The Ladd objects create the chord as a list. The pitch class input is added to each member of the initialized list. The iter object turns this list into three individual pitches.

All else is as described earlier.

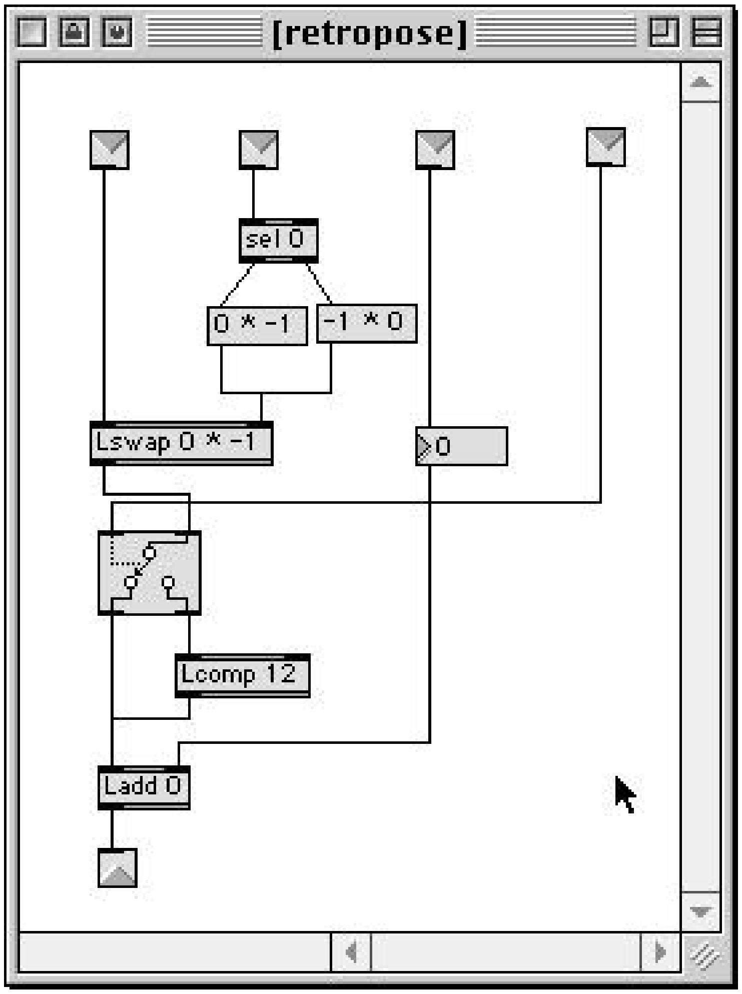
For more sophisticated ways to handle chords, see the essay on Max & Chords

Serial manipulations



Here is a patcher that demonstrates how to do traditional row manipulations. In this case, the row is generated randomly by the uzi and urn at the top. Thresh gathers the row into a list. The list is processed five times by the retropose subpatcher, and all five versions of the list are combined by Link (right to left, remember). The final list is fed to unlist and played at the rate determined by metro. (The gate prevents the first two notes from playing at once. That would happen because unlist sends the first member of a list out immediately, and metro bangs immediately when it is started.)

The real action is in the retropose subpatcher:



Retropose depends on Lswap for the reordering of pitches. Lswap rearranges input lists according to its template. A template is a list of the desired output order. The positions in a list are numbered 0, 1, 2 and so forth, and a template that started with 2 would put the third member of the list first in the output. Negative numbers mean count back from the end, so –1 is the last member of the list, -2 the next to last and so on. A symbol means "all the members between", so 0 * 4 would give the first 5 members of the list. 0 * -1 gives them all, and –1 * 0 gives them all backwards. Lswap templates can be very complex (you can repeat positions and get a member more than once), so you could produce every possible permutation of the list.

Lcomp 12 will provide the optional inversion of the row. There is an Linvert object, but the math definition of inversion is not the same as that in music. Musical inversions are produced by subtracting the pitch class from a constant (12), which is really the complement opeartion.

Ladd provides any desired transposition.