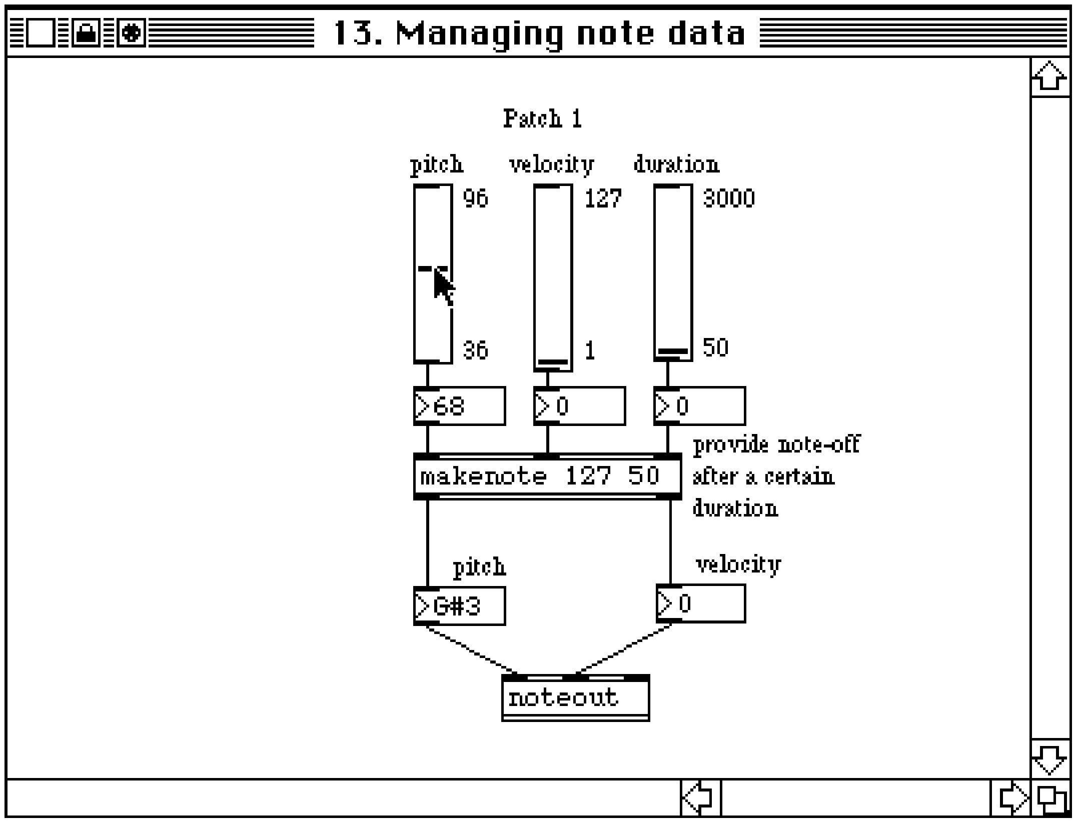
INTRODUCTION TO MAX

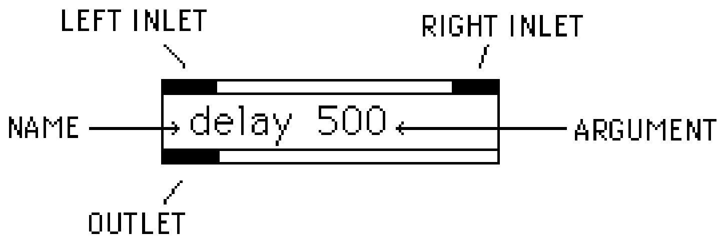
MAX is a programming language designed to look familiar to composers who have worked with patchable synthesizers. What you see on the screen is a lot of boxes with lines connecting them. The boxes are called objects and the lines are patch cords. What happens is that each object represents a process. The results of the process are passed along the patch cord to the next object. Ultimately, there is an object that generates output from the MIDI port.



Each windowfull of objects is called a patcher. Several patchers may be open at once and they can all be active, even if their window is hidden.

OBJECTS

The basic object looks like this:



The name of the the object determines what it does. There are a couple of hundred objects supplied on the disk, ranging in complexity from simple math to full featured sequencers. Arguments, if present, specify initial values for the object to work with. Data comes into the object via the inlets, and results are put out the outlets. Each inlet or outlet on an object has a specific meaning. This will be displayed in the lower left of the window as the mouse passes by (further details are in the manual). Usually, input to the left inlet triggers the operation of the object. For instance, the delay object (as shown) will send a bang (see below) out the outlet 500 milliseconds after a bang is received in the left inlet. Data applied to the right inlet will change the delay time.

MESSAGES

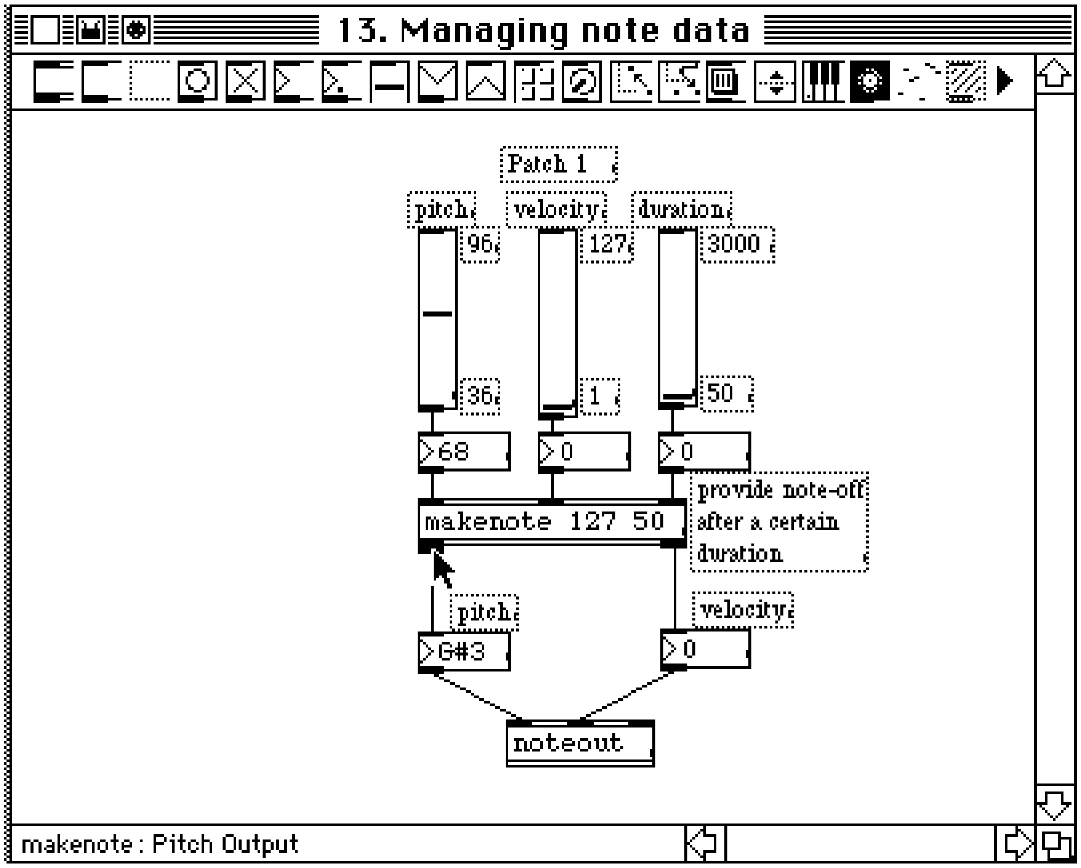
Data bytes sent down the patch cords are called messages, which fall into one of the following types:

int	A number without a decimal point.
float	A number with a decimal point.
symbol	A character string[1] such as “stop” that may be understood by certain objects.
list	Several of the above, separated by spaces. The first element of a list must be a number.
bang	A message that triggers the action of an object.

[1]String is computerese for a group of letters.

EDITING PATCHES

There is a padlock icon in the title bar of a patcher. If the padlock is opened (with a mouse click) the patcher may be edited.

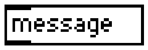


The palette across the top of the window contains object icons. Click on your choice (usually the left one) and move it into position and click again. You will get a dialog that allows you to select the name, or you may type the name directly (pay attention to capitol letters). If you option-click on an object, a help screen will come up that explains how it works. Enter patch cords by clicking on an outlet. You then drag a line to the inlet you want.[2] Many connections may be made to a single inlet or outlet.

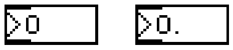
[2]Max won't let you make inappropriate connections.

INTERFACE AND DISPLAY OBJECTS

Most of the objects in the edit pallette are there to allow the user to control the patch. There are sliders and dials, even a cute little keyboard.
The most useful are the simple ones. For instance:



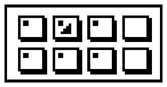
The message box will contain a symbol (up to 255 characters long) or a list. Usually these are entered when the box is created. When the user clicks on the box, the message is sent out. The message will also be sent if anything is received at the inlet. The symbol “set” at the inlet will cause the message to be changed to whatever follows.



Number boxes are used a lot. (There are two kinds, for ints and floats.) A number that comes in the inlet will be displayed. A bang at the inlet will send the number out. If the user clicks on the box and drags up or down, he can change the number, which is sent out in the process. If you select the box during editing and click on Get Info in the menu, you can customize the behavior of the box, such as setting limits on the values, or making it display pitch names instead of the number.

Buttons send a bang when the user clicks on them. If any message is received at the inlet a bang is sent.

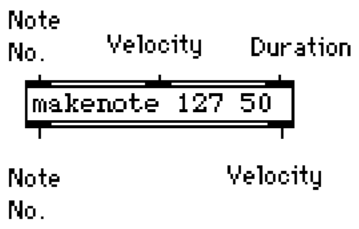
When the user clicks on the toggle the X disappears and a 0 is sent. If he does it again, the X comes back and a 1 is sent. A 0 at the inlet will clear the toggle, and any other number will set it. A bang will flip the toggle to the other state. (1 or 0 is sent only when the toggle changes.)


The preset object allows the user to store the state of all controls. To make this happen, you just put the preset in the patcher somewhere. If the user shift clicks on a button, all current values will be stored. Later, a click on the button will bring them back. The dot on a button indicates it contains data, the triangle indicates the most recently selected preset. A preset object with patchcords attached only affects the connected items.

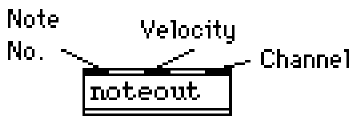
HOW TO GET MUSIC IN AND OUT OF MAX

Max is a number cruncher with timing features. Max knows nothing about music. As long as you keep this in mind you should have no trouble getting interesting things to happen.

In MIDI, as you may remember, there are six numbers associated with each note: the note number, velocity and channel number for the note on, and the same three for the note off. If the note off channel number and note number do not match the ones for the note on, the note will play forever. Also remember that a note on with a velocity of 0 counts as an off.



Down toward the bottom of almost every patch, you will find a makenote object. When a note number (an int between 0 and 127) hits the left inlet, a velocity (as set by the middle inlet or the first argument) goes out the right outlet, folowed by the note number out the left outlet. A few milliseconds later, as set by the right inlet or the second argument, the note number comes out again, but this time with a 0 velocity. These values are just what the next object needs:



Noteout uses the channel, velocity and note numbers provided to send a note message out the MIDI port. This works very reliably with makenote, with one warning: BE SURE MAKENOTE IS EMPTY IF YOU CHANGE MIDI CHANNELS. Else any pending note offs will be steered to the wrong channel and the note will hang.

Getting MIDI into MAX is duck soup. There’s a notein object that provides note numbers, velocities and channel numbers. Usually we are only interested in which note was played and not how long it was. The stripnote object will filter out noteoffs in that situation.

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

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

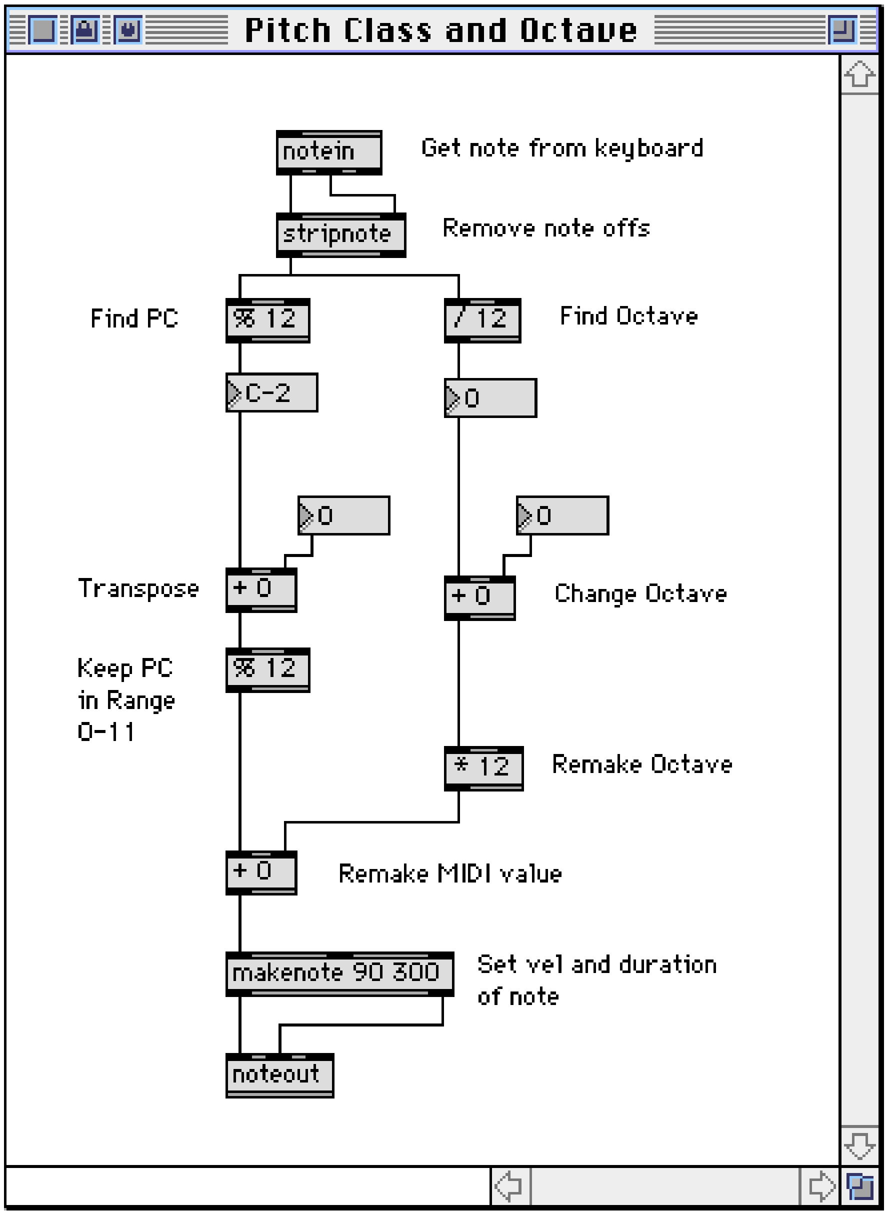
In this system pc can take the values 0 - 11, in which 0 represents a C. Oct typically ranges from 0 to 10. Middle C, which is called C3 in the MIDI literature, and C4 by most musicians, is octave 5 under this convention.

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.

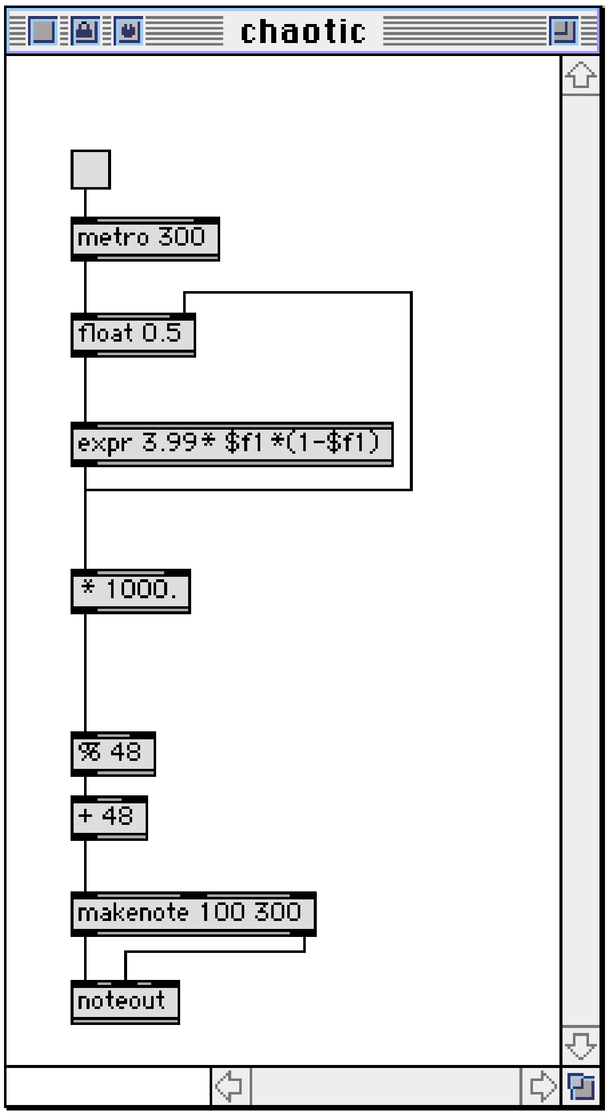
Generating Pitches

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.)

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 each time the patch is loaded the results will always come out different.

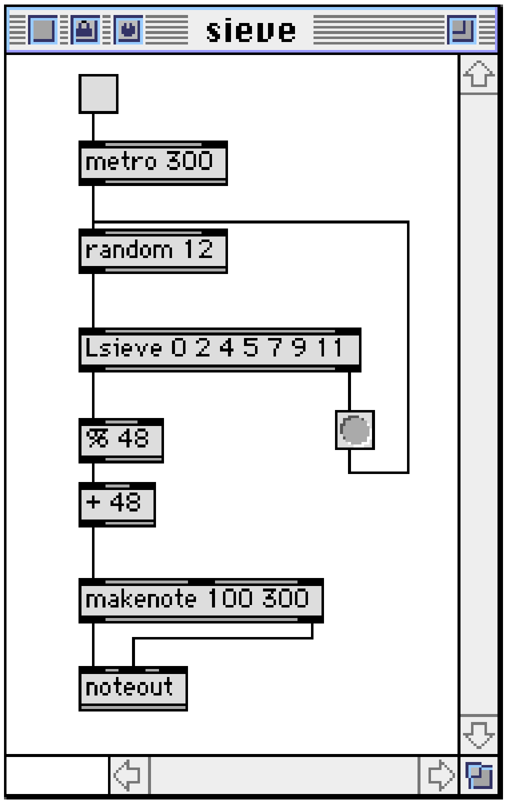
Processing Random Pitches

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**[3] object does this handily;

[3] 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.

Lsieve 0 2 4 5 7 9 11

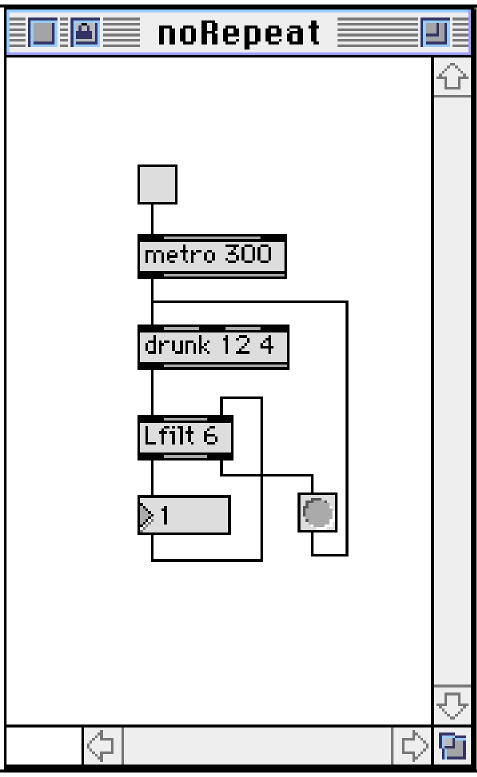
will only allow the notes of C major through. In this patcher:



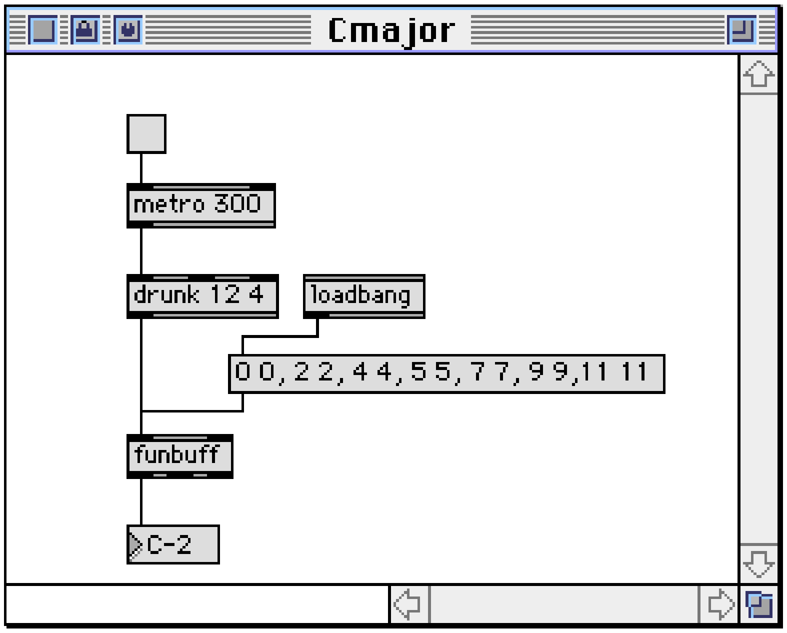
Any note that fails the test will cause the random object to try again, because failed values fall out the right outlet. You need to be very careful when using this type of feedback process. If Lsieve were to reject everything random puts out (for instance if it had all values higher than the range of 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.

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:



Generating Rhythms

The motive force in Max is the metro object. This is an object that simply emits bangs at a steady rate. Metro can easily be turned on or off by sending a 1 or 0 into the left inlet.

The way to control rhythms in Max is to control the argument to the metro object. To do this properly, we must first understand the relationships between Period, Tempo, Note Value, and Note Duration.

The argument in the metro object is a period[4], the number of milliseconds between bangs at the metro outlet. Tempo is the number of beats per minute.

Note Value is the kind of note: half, whole, thirtysecond triplet, whatever. In Max, these must be represented by a number of some sort. There are two approaches:

> In Denominator Notation, a whole is 1, a half note is 2, a half triplet is 3, and so forth (a 32nd triplet is 48, and a dotted half is 0.75.)
>
> In Tickcount notation, the smallest value you want to represent is assigned two ticks, and all the others are represented by the number of ticks they contain. In this system, a 32nd triplet would be a 2, a plain 32nd a 3, a quarter note 24, and a whole 96.

Each system has its advantages- a list of values in denominator notation is easy to read, but tickcount calculations are usually simpler.

Duration is the amount of time the note will actually last. We have to combine Note Value and tempo to find it. If you are using denominator notation, you first find the duration of a whole note

WholeDur = 60 * 1000 * number of beats in a whole / Tempo

This only has to be done when the tempo changes. For instance, a tempo of 90 would have a WholeDur of:

$$240000/90 = 2666.6 \text{ ms}$$

To find the duration of a note, you divide WholeDur by the denominator value. A quarter note at a tempo of 90 will have a duration of 666 ms.

When using the tickcount system, the first step is to find the duration of one tick in milliseconds. The formula is

[4]Of course you remember that period is the inverse of frequency. A 500 hz signal has a period of 2 milliseconds.

TickDur = 60* 1000/ (Tempo * tickcount of a beat)

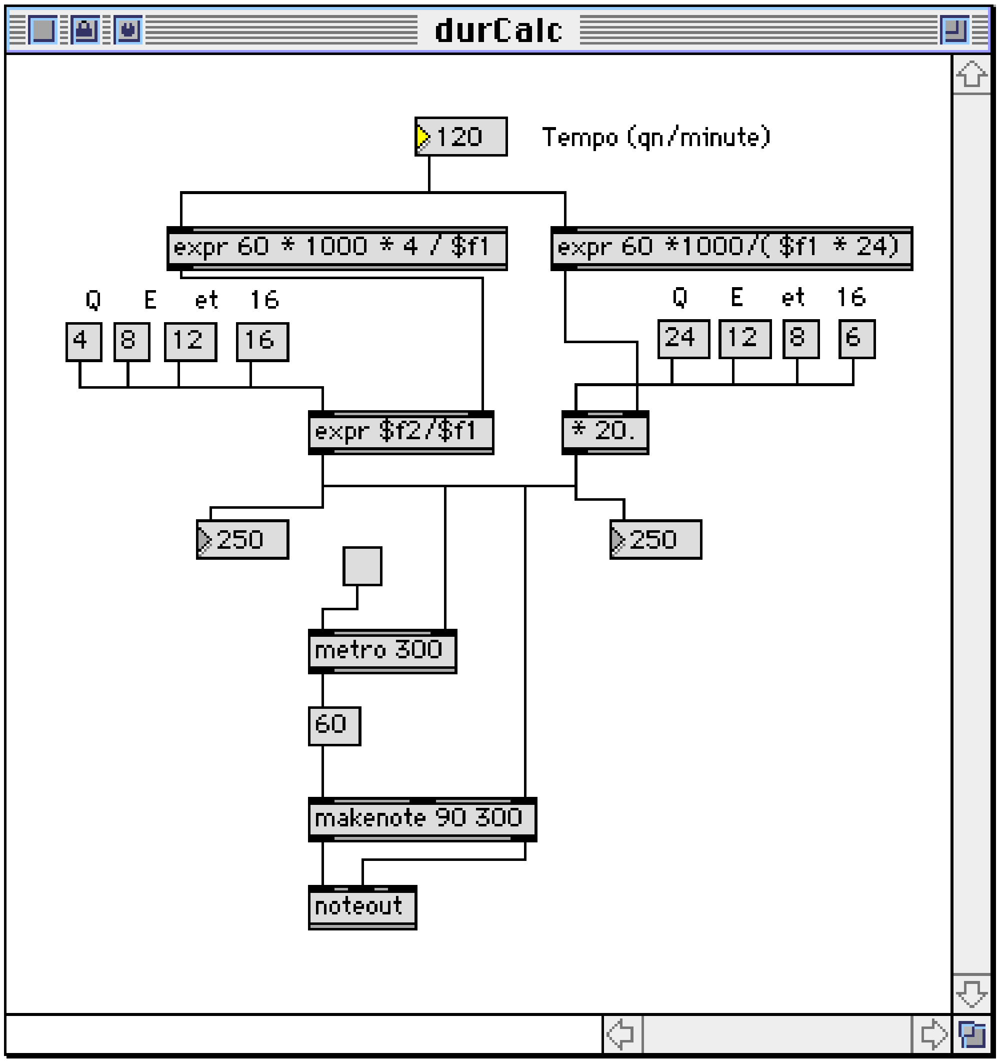
For example, again at tempo = 90, if the quarter note gets the beat

TickDur = 60* 1000/ (90 * 24) = 27.7

TickDur will be multiplied by the tickcount for each note. Again, a quarter note comes out as 666 ms.

These durations can be accurately calculated, but there may be errors in timing when the results are put to use. That's because the metro object is only accurate to the millsecond. If you use 666 as the argument to a metro object, hoping to get a tempo of 90, it will run a mite fast. (The period of a quarter note is really 666.6 ms.) This usually not important, but it can cause occasional problems.

Here is a patcher to illustrate these calculations:



This just plays Middle C at the tempo and value chosen. Turn on the metro object to hear, then click on the Q,E,et or 16 boxes to test various durations.

Notice that the operations are carried out as floats to maintain accuracy.

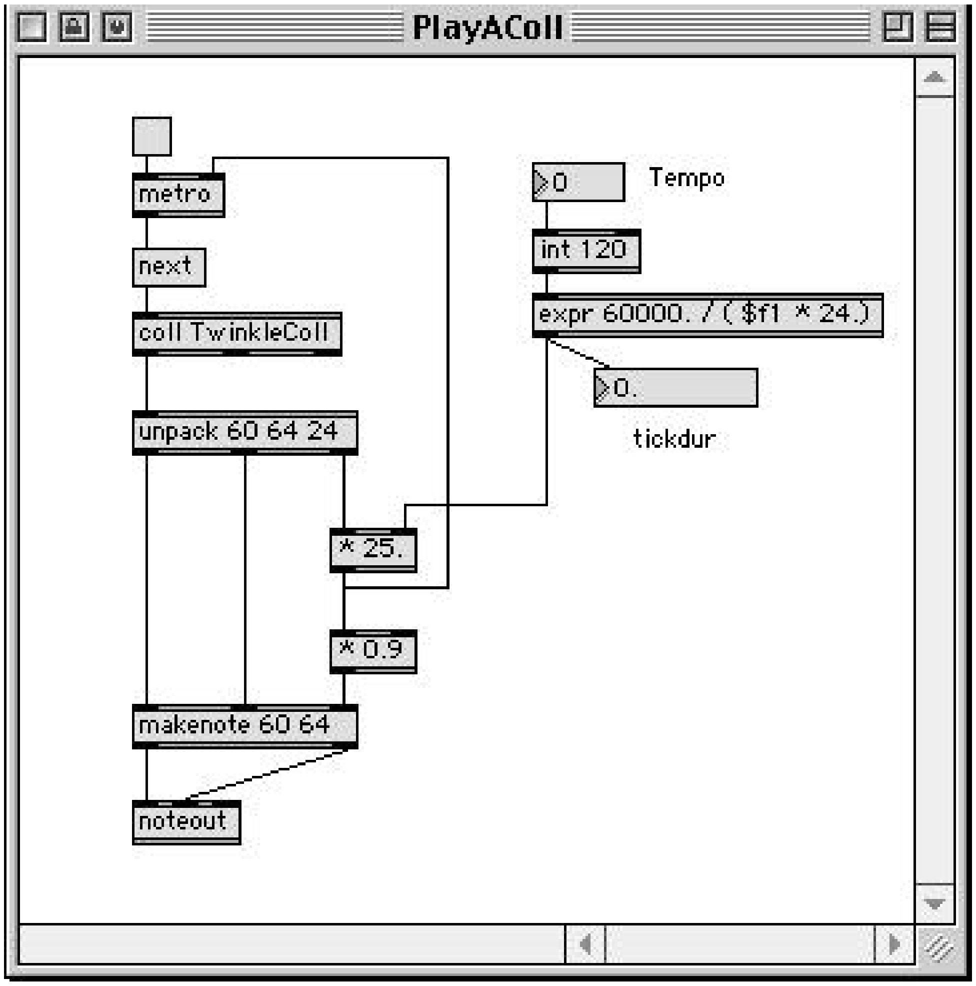
There are a couple of things I should point out about generating rhythms in general. First, the duration has to be sent to the metro and makenote before the note is played.

Second, the duration at makenote does not have to be the same as the duration at metro. However, when you play repeated notes like this, if makenote has a longer duration than metro, what is heard with many synthesizers will be the difference between the two durations, because of the way the note ons and offs will overlap. The messages will come out as on, on, off, on, off, on, off, on, off, off. So, it's a good idea to subtract a bit from the duration on the way into makenote. (Or use Lnote, which doesn't suffer from this problem.)

Third, in complex patchers, you may find yourself trying to generate a lot of notes at once for big fat chords. The system (both Max and MIDI) will tend to clog up if you send the noteoffs for the old chord at the same time as the new notes. Again, it's a good idea to shorten the durations sent to makenote.

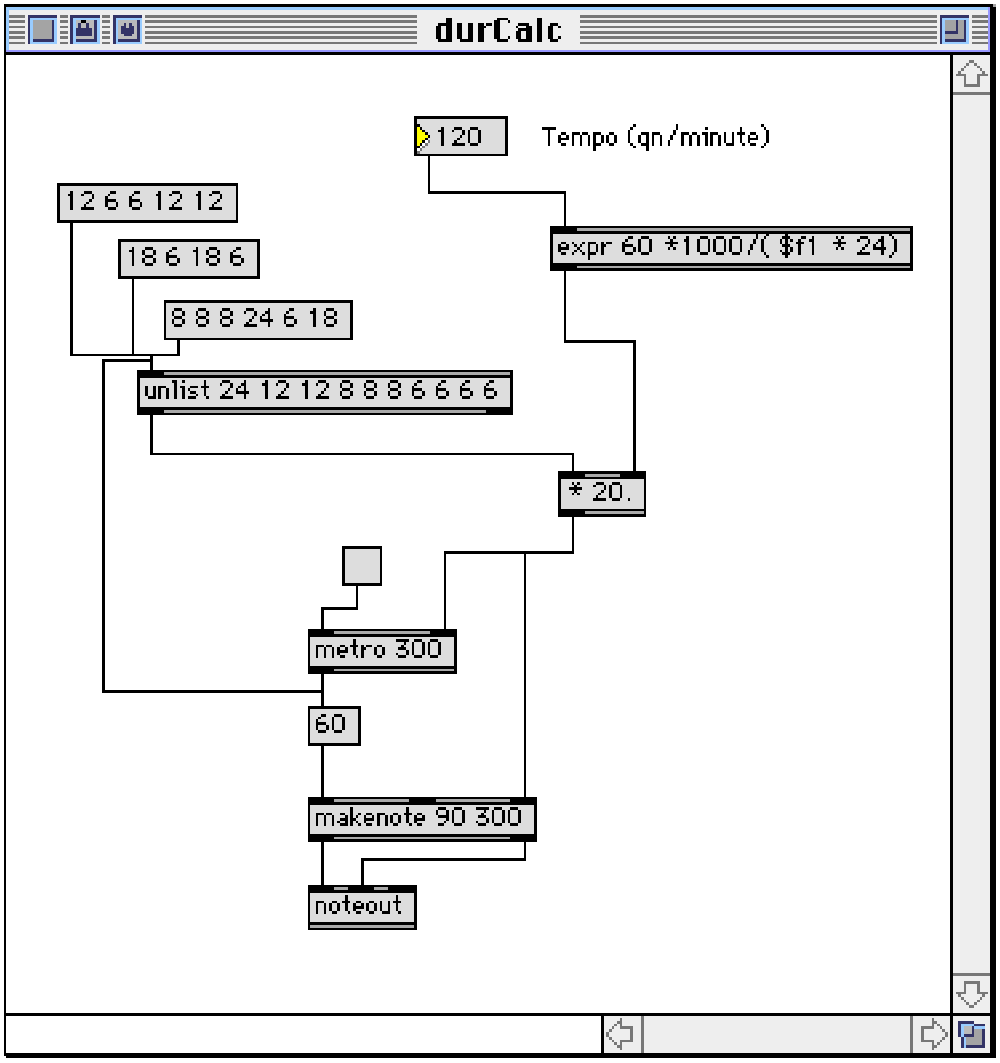
Patterns

Steady streams of notes eventually become boring. Most music includes a variety of durations, often in recognizable groups or patterns. Many composers generate a duration with each pitch, and then handle their notes as lists of pitch, velocity, duration. Here is how a series of note lists (stored in a coll) might be played. Notice that the duration is sent to metro before the note is triggered. That's because a change in the period of metro cannot take effect until after the next bang is output.

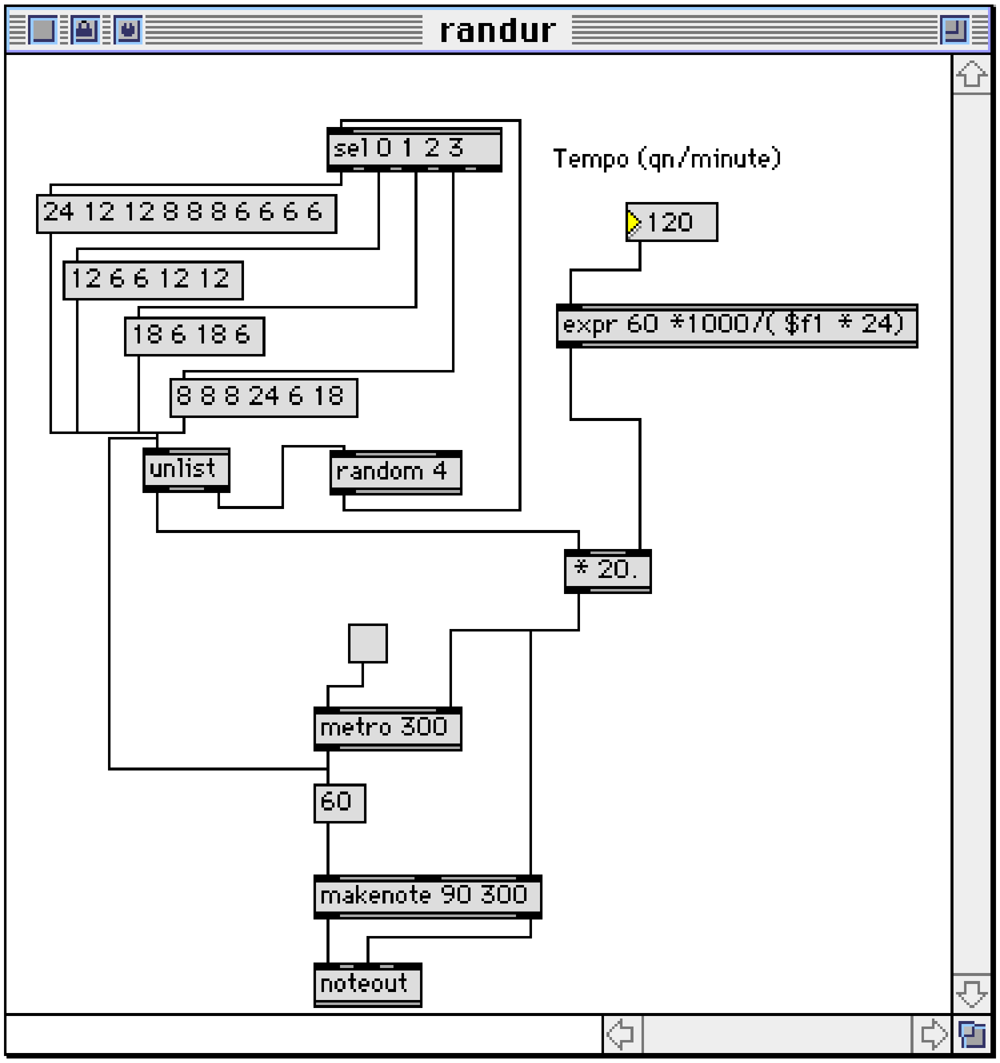


Another approach is to think of rhythm patterns as basic units that get filled up with pitches.
The problem presented to Max is to choose patterns at appropriate times. To facilitate this, I wrote an object called unlist.

The rhythm patterns, in either denominator or TickDur notation, are kept in lists: this patcher illustrates using unlist to manage them



When used with an argument list, unlist steps through the list over and over as the inlet is banged. If a new list is received, it replaces the stored list.With no arguments, unlist behaves differently. When you send a list into the inlet, the first member of the list is sent out immediately. Bangs then step you through the list. When the list is used up, a bang is sent out the right outlet. If this bang triggers some process that applies a new list to the inlet, the first member is sent out and the cycle begins again, with no real break in the output stream. This patcher illustrates the process:



Synchronizing Pitch and Rhythm

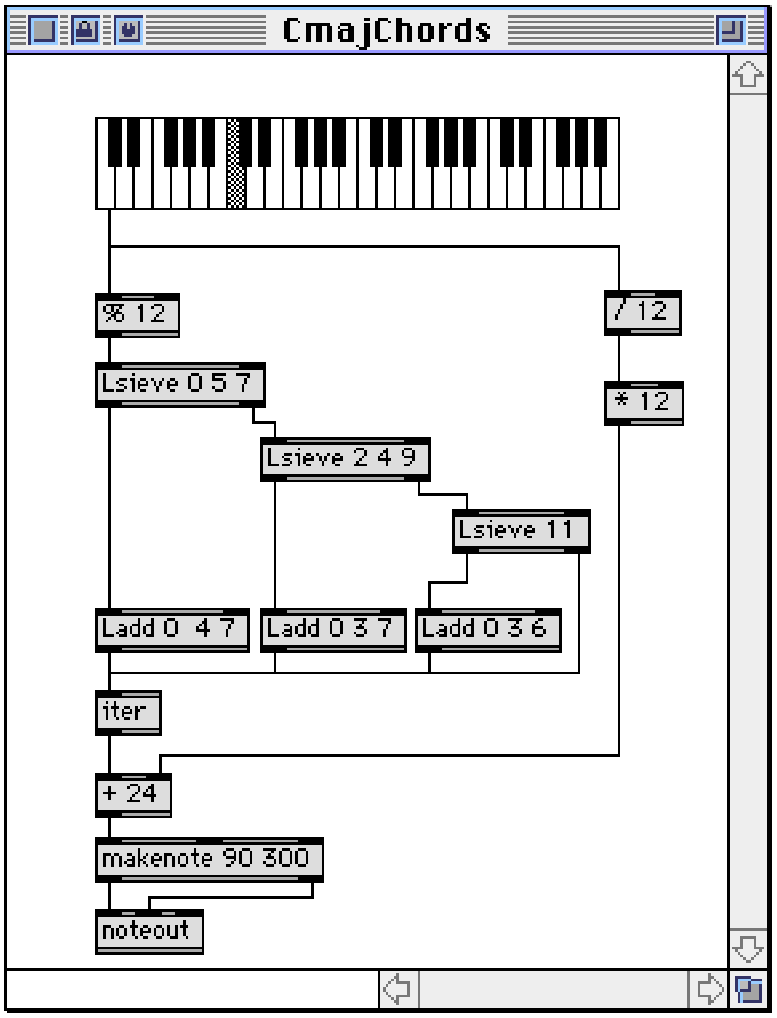
The previous two examples don't address the issue of synchronizing pitch and rhythm. There are many approaches to this. The core of the problem is get things loaded in the right order: period for metro and duration for makenote before the metro ticks, then the metro tick should send a pitch to makenote to actually play the note. So when you look at it, the output of metro really really has to:

- •Play this note
- •Set up period for next note.
- •Set up duration of next note.

It's a good idea to use a trigger object to parcel these tasks out. (The order of the last two is not important, but they both must be in place before the next note is played.)

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 the basic approach:



The Lseive 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.