21M.361 Composing with Computers I (Electronic Music Composition)
Spring 2008

21M.361: Composing with Computers I (Electronic Music Composition)

Peter Whincop

Spring 2008 OCW

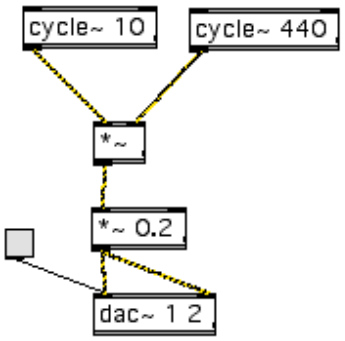
[All lab notes are being significantly revised for next term to reflect upgrades in software, different organization, and the incorporation of my own notes on DSP.]

Lab 4.2: More Max/MSP

1. **[trigger]** or **[t]** is one of the most important objects in Max. **It triggers events in a right-to-left order, so you don't have to rely on graphical right-to-left precedence**. [t b i] will have two outlets, since there are two arguments. When a number is received (float or integer), it outputs from the [trigger] right-to-left: first an integer (even an integer version of a float) is output through the right outlet, and once that chain of events has completed, the number causes a lbangl to be output through the left outlet. **You can many arguments to [t]; data types can be represented by i (integer), f (float), s (symbol), b (bang), and l (list)**. Actual values can be set, too; [t b 45.3 banana f] will pass a float, the messagelbananal, the float 45.3, and a bang, in that order. You have to understand **the concept of chain of events** to understand [t] fully. Once an event starts, e.g. the number 3 is passed down a patch cord, all that can be done down that branch of the tree (like a family tree) will be done, then the scheduler moves up to the last junction not initiated. (This will be revealed in lab, and I'll write it up next lab.) An example of [t] will show up when we look at [pack] in relation to [sfplay~].

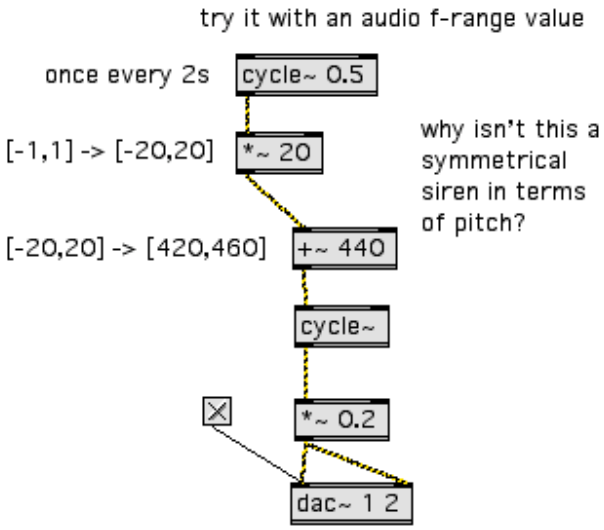
2. **[loadbang] sends a lbangl when the patch is loaded, or when it is double-clicked; [loadmess *args*] sends args right-to-left when the patch loads. [bangbang] or [b] sends two bangs, right-to-left; [b *int*] sends *int* bangs right-to-left.**

3. **AM synthesis**. The amplitude of a modulating oscillator modulates the amplitude (hence Amplitude Modulation) of a carrier oscillator. In the more general case, the modulator and carrier need not be oscillators, but any signal (i.e., sound). For example you can amplitude modulate two soundfiles. AM is commutative; the convention is that the carrier is the more constant of the signals (as in AM radio broadcast). Simply have the output of two oscillators (or other signals) multiplied together. (For those interested, AM is real-time—not whole sample—convolution in the frequency domain, i.e., multiplication in the time domain.)



4. **FM synthesis**. The amplitude of a modulating oscillator modulates the frequency (hence Frequency Modulation) of a carrier oscillator. Again, this can apply to any signal (for very interesting results). Using just oscillators,

interesting timbres can be developed. The simplest way to make an FM instrument is to have a number box feeding into the frequency input of a (modulating) [cycle~], [*~] multiply by the width above and below the center frequency of the carrier, [+~] add to the center frequency, and send to the output. A slow modulator will produce a siren; higher modulating frequencies will produce more complex timbres.



Courtesy of Cycling '74. Used with permission.

5. **More complicated FM synthesis**. Still just with two oscillators, more complex timbres can be created. Rather than using modulation frequency, width of modulation, and center frequency as your parameters, use carrier frequency, harmonicity ratio, and modulation index. A little math with these parameters will describe how many 'sidebands' and where they will be found in relation to the carrier frequency. Sidebands are other quieter frequencies that surround the center frequency. See MSP tutorial PDF 11 for a description of this; see the actual tutorial patch for how the parameters are connected. The actual FMing is carried out in the sub-patch called SimpleFM; double clicking on it will bring it up (not able to be edited) or you can find SimpleFM in the MSP tutorial folder, below the tutorials. If you use [SimpleFM] in your patch, don't forget to include the abstraction. I would prefer it if you wouldn't computer copy it, rather, reconstruct it yourself. In 21M.540 we look at the theory behind sidebands, using Bessel functions. Also, for now, don't try to understand the # argument in [SimpleFM].

6. **[adc~ 1 2]** means analog-to-digital converter, channels 1&2. The outputs are signals. The input is just like for [dac~] (aside from audio input to the latter): [x], or [startwindow] etc. You only need one [adc~] in your patch, just as you need only one [dac~]. And between them you need only one way of starting or stop audio.

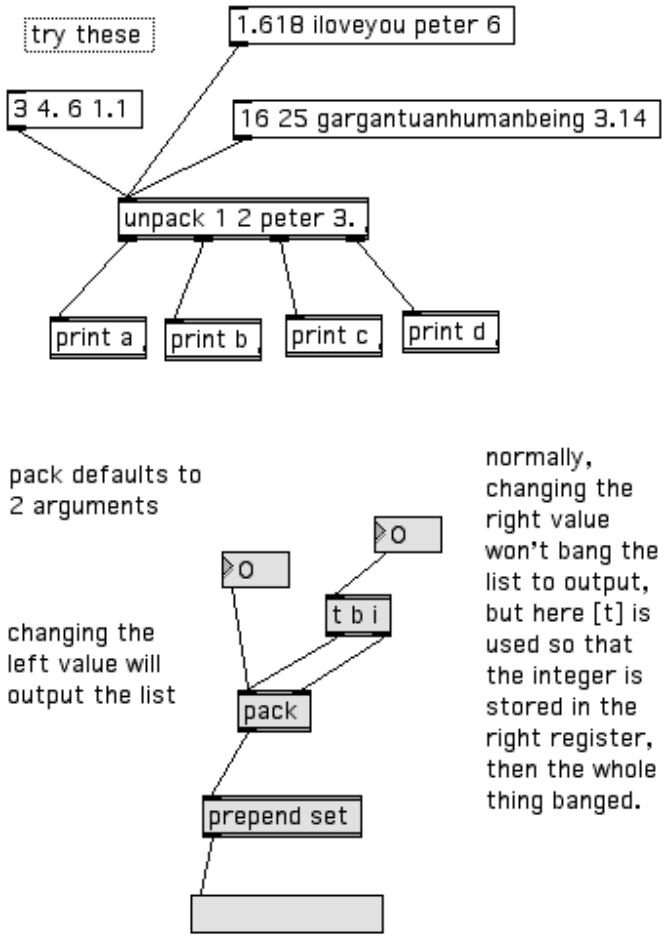
7. **[sfrecord~ 2]** records its inputs to a stereo soundfile. As with [sfplay~] it requires an lopenl message, and [x] to start and stop. Sometimes you will find yourself with a ton of patchchords going into a [dac~], and when you want to add an [sfrecord~] you then have to duplicate all those patchchords, this time going to the record object. This can be messy, and it is easy to forget to attach all chords. So it is helpful in these cases to have a [+~] above each input of [dac~], with everything being collected by the [+~]—one for each channel. From here you can attach the outputs to the [dac~] and to [sfrecord~]. Alternatively, you can use the **[adoutput~]** object; it intercepts all output to any [dac~] in

any window. I usually use [adoutput~] to feed [sfrecord~], just to keep things neat, especially if i have several active windows open at a time and I forgot to collect the signal all in one place. (Kind of sloppy, but it's a last resort.)

8. **[patcher *name*] or [p *name*]** will make a patch(er) within a patch. It is used to keep things neat and modular. As soon as you create the object, a new patch window will open, with the name in square brackets, meaning it is embedded. You use inlets and outlets in this sub-patch; creating them will create inlets and outlets in the parent-patch. They are read left-to-right for both inlets and outlets; if you change the order of them once you have created them, you will have to change the way you have patched to the [p] object in the parent-patch. [p] is used for one-off deals. If you want to make a sub-patch that can be used many times, either in one patch, or across many, then create a separate patch, using inlets and outlets. Save it in the same directory. In the main patch, creating an object with the name of the sub-patch, e.g. [SimpleFM] in the FM tutorial, will create an object with correct number of inlets and outlets, corresponding left-to-right in the sub-patch. Every time you use this new 'object' (it's called an abstraction in Max/MSP-talk, which is just a way of saying you have created a reusable patch that functions like an object), it is an independent instance of the sub-patch, functioning as if an object. Just as with [cycle~] being used many times in the same patch, or in many concurrent windows, the parameters are not confused between the instances. If you double-click on an abstraction in a window, the sub-patch will appear, with its name in square brackets; you will not be able to edit it, though it will show data pertinent to that particular instance. To edit it you must open the sub-patch itself, as a normal patch; any changes made must be saved.

9. **[send *name*] and [receive *name*], or [s *name*] and [r *name*]** for short, will send a message (number, symbol, list, etc.) to every instance of the correspondingly named [r] object. It is global in scope, meaning it passes across windows, and into sub-patches and abstractions. (There are ways of finessing this when various instances require different names, also using the pound key-symbol; I'll teach that in the advanced course with the [poly~] object.) Audio signals can be sent and received with **[send~ *name*] and [receive~ *name*]; there are no abbreviations**. Using send and receive keeps things tidy, and allow communication between patches without the need for inlets and outlets.

10. **[pack] packs individual data into a list**. The arguments of [pack] are the default values/data types; the number of arguments will determine the number of inputs to [pack] and the length of the list to be output. It cannot be changed dynamically. For example, [pack 0 1 peter 2.] means that the pack values are two integers, a symbol, and a float, with default values as given. Sending data into all but the left inlet will just store the value, rather as in arithmetic operations; sending data to the left inlet will store the value and bang the [pack] to output the list. (Recall that you can see the list by having a branch from the output that goes to the [prepend set] object leading to an empty message box.) Banging the pack object will send the list, which may or may not still have its default values. **[unpack] unpacks data from a list**. For example [unpack 0 1 peter 2.] expects a message (list if it has more than one component) of the form int int symbol float. There will be four outlets, since there are four arguments. If the data types of the list do not correspond to the data type of the arguments of [unpack], various things happen. A symbol going to a number will be output as 0, a number going to a symbol will not be output; numbers will be converted to the argument's type. One final thing: **[pak] (apparently pronounced 'pock') bangs with any input, not just through the left inlet**. (I should point out that I have been using 'symbol' a little freely as a data type; more on that in 21M.540.)



Courtesy of Cycling '74. Used with permission.

11. **[value *name*] ([v] for short) stores a value (of any data type); banging it will send it**. You can have other [value] objects with the same name; banging those (even without inputs) will send whatever value was sent to any other [value] of the same name. **[int] and [float] ([i] and [f]) accept integers and floats; into the left inlet they pass the value through; into the right inlet they store the value until banged**. **[sig~] converts control data into a constant audio signal**; this type of data-type conversion is necessary under certain circumstances. See, for instance, MSP tutorial 11, FM synthesis. In the sub-patch, [*~] is expecting a signal, but the value you choose in the parent-patch is a number box; [sig~] changes it to an audio rate version of the constant.

12. **[line] accepts a message in the form [*init-value*, *end-value time-in-ms*]** e.g. [440, 880 2000]. If you fed the last message into [line], then into [cycle~], then clicked on the message or banged it, [cycle~] would oscillate at 440Hz, immediately to rise to 880Hz over a period of 2 seconds. [line *init-value*] can be sent a message in the form [*end-value time-in-ms*]. **[line~] does the same, but at the audio rate, therefore more subtly and more usefully. Unlike [line] it can accept lists of values, like a breakpoint function**, e.g. [440, 880 2000 220 500 1320 3000]. **Read the values after the comma in pairs**.

13. **[function]** will give you a breakpoint function box. See the manual or the help patch for its subtleties. **It is usually**

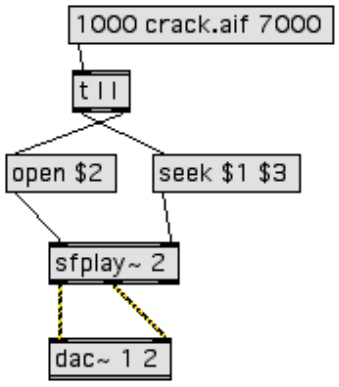
used to drive [line~], which is done so through the second output. See MSP tutorial 11 (FM synthesis) for an example. It is very useful for making envelopes (amplitude, anything else).

14. **[phasor~] can be used as a saw-tooth generator, though [saw~] is better**. **Use [phasor~] to drive objects that require a ramping input. [phasor~] outputs at the audio rate a ramp from 0 to 1 at the given frequency. [triangle~] produces a triangle wave**. To make a square wave, wait for next week's section on audio-date conditionals, or you can use **[rect~]**. All the common waveforms can be made by additive synthesis of sines (or cosines) in some harmonic relation, with amplitudes in such relationships as 1/*n* or (n-1)/*n*^2, etc.

15. **[scope~]** gives you an oscilloscope. It is therefore a time-domain representation of a signal. At low frequencies you won't have to feed it parameters; at higher frequencies you will have to set its parameters. **[spectroscope~]** gives you a frequency-domain representation (Fourier transform) of a signal.

16. [select] and [route] do what they say. **[route] compares the first item of a list with an argument, and sends the rest of the list down that argument's outlet**. For example, lpeter 100 10l sent to [route 1 2 peter] will result in l100 10l being sent from the third outlet, corresponding to lpeterl. If there is no match, the whole input list is output through the right outlet. [route] is a useful way to traffic data as if they are attributes around a patch. **[select] or [sel] compares its arguments with an input, and bangs the appropriate outlet**. For example, if l22l were sent to [select tEp 22 omg] then the second outlet would bang. If there is no match, the input (not a bang) is sent to the right outlet.

17. **Arguments and keywords in messages**. A messages can say (almost) anything, **but a message *box* can only receive messages that begin with a keyword**, such as lappend ...l or lset ...l. **lsetl in particular will set the new message box to the message (list etc.) but will stop the message from being sent any further**. This is useful for setting up a value for later use, say, to deal with timing/scheduling issues, and a bang later to the new message box will send it. (In other words, lbangl is another keyword that message boxes understand.) **Argument placeholders take the form $1, $2**. An example would best demonstrate this:



Use backslash for literals. The best way to format a message is with **[sprintf]** which uses the standard C library for printf (the manual of which you will have to look up; it isn't in the Max manual). If you are trying to manipulate data, and you can't quite get it right, email me, and I'll try to get it into the correct format.

18. **Aliasing is frequency reflection**. Frequencies reflect around 0Hz, i.e., -50Hz is the same as 50Hz (though phase is inverted); and frequencies reflect around the Nyquist frequency. **The Nyquist frequency is half the sampling rate**. The idea is that all frequencies can be represented up until half the sampling rate. Our sampling rate is 44.1kHz, so anything up to 22.05kHz (pretty much to top end of human hearing, hence it is the CD sampling rate) can be represented. Anything above it is reflected, so 22.2kHz comes out as 22.1kHz. You might notice this when transposing (which we haven't done yet), and especially with sidebands in FM synthesis.

19. If you are even more daring, you can use **[buffer~]** and all its related objects, and **[pfft~]** for frequency domain work. I won't be teaching it in this course; it's reserved for 21M.540. Please don't try to race ahead *too* far; leave me something to teach in the advanced course! But I'll help if you ask.