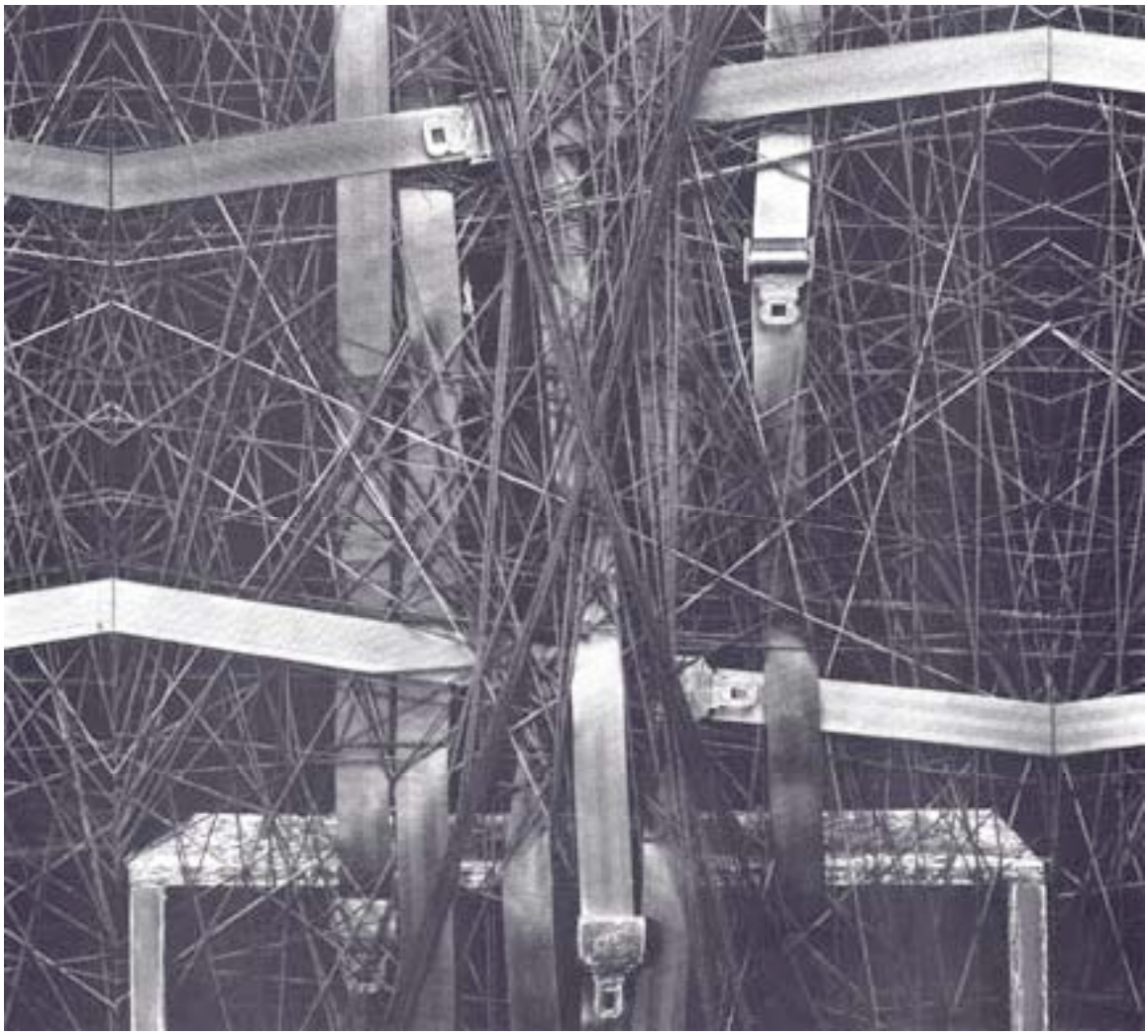
MSP



Tutorials and Topics

Version 4.6/20 June 2006

Copyright and Trademark Notices

Credits

Original MSP Documentation: Chris Dobrian

Audio I/O: David Zicarelli, Andrew Pask, Darwin Grosse

MSP Reference: David Zicarelli, Gregory Taylor, Joshua Kit Clayton, jhno, Richard Dudas, R. Luke DuBois, Andrew Pask

MSP Manual page example patches: R. Luke DuBois, Darwin Grosse, Ben Nevile, Joshua Kit Clayton, David Zicarelli

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Table of Contents

Table of Contents

Table of Contents

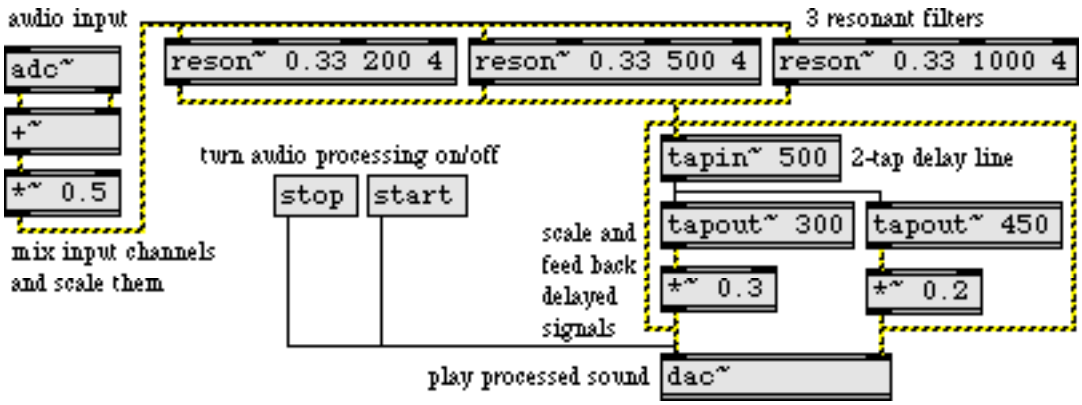
Table of Contents

Table of Contents

Table of Contents

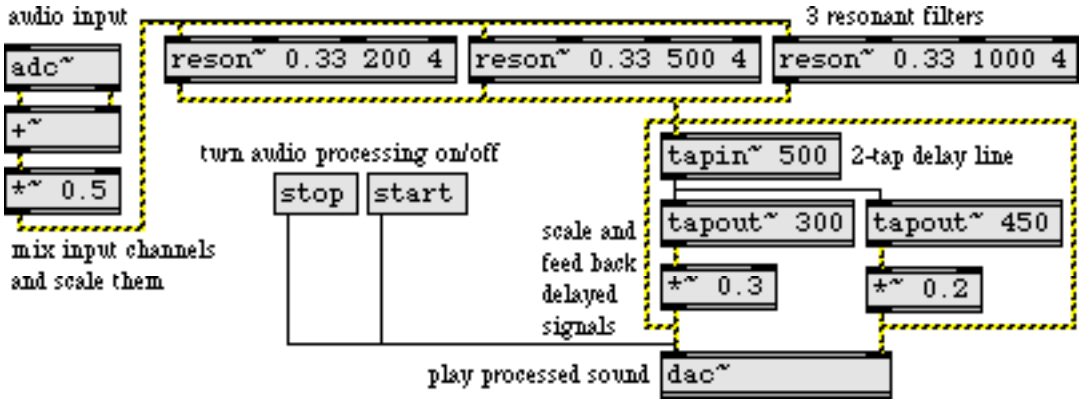
Signal processing in Max

MSP gives you over 170 Max objects with which to build your own synthesizers, samplers, and effects processors as software instruments that perform audio signal processing.



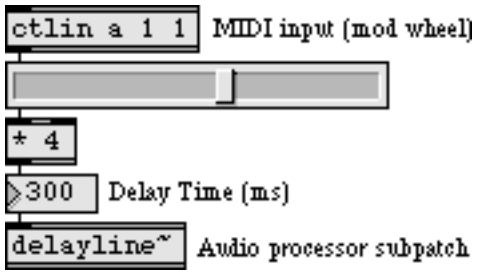
A filter and delay effect processor in MSP

As you know, Max enables you to design your own programs for controlling MIDI synthesizers, samplers, and effects processors.



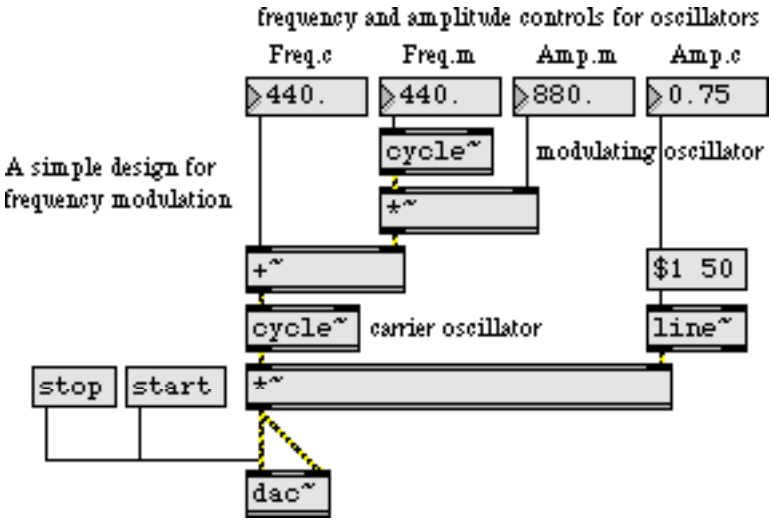
MIDI control with Max

With the addition of the MSP objects, you can also create your own digital audio device designs— your own computer music *instruments*—and incorporate them directly into your Max programs. You can specify exactly how you want your instruments to respond to MIDI control, and you can implement the entire system in a Max patch.



MIDI control of a parameter of an audio process

MSP objects are connected together by patch cords in the same way as Max objects. These connected MSP objects form a *signal network* which describes a scheme for the production and modification of digital audio signals. (This signal network is roughly comparable to the *instrument definition* familiar to users of *Music N* sound synthesis languages such as Csound.) The audio signals are played through the audio output jack of your computer, or through an installed sound card, using CoreAudio on the Macintosh, MME or DirectSound on Windows, or ASIO on either platform.



Signal network for an FM instrument

How To Use This Manual

The MSP Documentation contains the following sections:

Digital Audio explains how computers represent sound. Reading this chapter may be helpful if MSP is your first exposure to digital manipulation of audio. If you already have experience in this area, you can probably skip this chapter.

How MSP Works provides an overview of the ideas behind MSP and how the software is integrated into the Max environment. Almost everyone will want to read this brief chapter.

Audio Input and Output describes MSP support for Core Audio on Macintosh systems, support for DirectSound on Windows systems, and audio interface cards. It explains how to use the DSP Status window to monitor and tweak MSP performance.

The MSP Tutorials are over 30 step-by-step lessons in the basics of using MSP to create digital audio applications. Each chapter is accompanied by a patch found in the MSP Tutorial folder. If you're just getting set up with MSP, you should at least check out the first tutorial, which covers setting up MSP to make sound come out of your computer.

The *MSP Object Reference* section describes the workings of each of the MSP objects. It's organized in alphabetical order.

Reading the manual online

The table of contents of the MSP documentation is bookmarked, so you can view the bookmarks and jump to any topic listed by clicking on its names. To view the bookmarks, choose **Bookmarks** from the Windows menu. Click on the triangle next to each section to expand it.

Instead of using the Index at the end of the manual, it might be easier to use Acrobat Reader's Find command. Choose Find from the Tools menu, then type in a word you're looking for. **Find** will highlight the first instance of the word, and **Find Again** takes you to subsequent instances. We'd like to take this opportunity to discourage you from printing out the manual unless you find it absolutely necessary.

Other Resources for MSP Users

The help files found in the *max- help* folder provide interactive examples of the use of each MSP object.

The *Max/MSP Examples* folder contains a number of interesting and amusing demonstrations of what can be done with MSP.

The Cycling '74 web site provides the latest updates to our software as well as an extensive list of frequently asked questions and other support information.

Cycling '74 runs an on-line Max/MSP discussion where you can ask questions about programming, exchange ideas, and find out about new objects and examples other users are sharing. For information on joining the discussion, as well as a guide to third-party Max/MSP resources, visit *http://www.cycling74.com/community*

Finally, if you're having trouble with the operation of MSP, send e-mail to *support@cycling74.com*, and we'll try to help you. We'd like to encourage you to submit questions of a more conceptual nature ("how do I...?") to the Max/MSP mailing list, so that the entire community can provide input and benefit from the discussion.

How Digital Audio Works

A thorough explanation of how digital audio works is well beyond the scope of this manual. What follows is a very brief explanation that will give you the minimum understanding necessary to use MSP successfully.

For a more complete explanation of how digital audio works, we recommend *The Computer Music Tutorial* by Curtis Roads, published in 1996 by the MIT Press. It also includes an extensive bibliography on the subject.

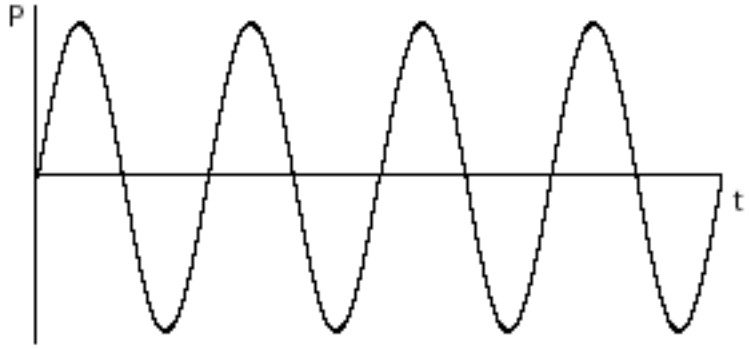
Sound

Simple harmonic motion

The sounds we hear are fluctuations in air pressure—tiny variations from normal atmospheric pressure—caused by vibrating objects. (Well, technically it could be water pressure if you're listening underwater, but please keep your computer out of the swimming pool.)

As an object moves, it displaces air molecules next to it, which in turn displace air molecules next to them, and so on, resulting in a momentary "high pressure front" that travels away from the moving object (toward your ears). So, if we cause an object to vibrate—we strike a tuning fork, for example—and then measure the air pressure at some nearby point with a microphone, the microphone will detect a slight rise in air pressure as the "high pressure front" moves by. Since the tine of the tuning fork is fairly rigid and is fixed at one end, there is a restoring force pulling it back to its normal position, and because this restoring force gives it momentum it overshoots its normal position, moves to the opposite extreme position, and continues vibrating back and forth in this manner until it eventually loses momentum and comes to rest in its normal position. As a result, our microphone detects a rise in pressure, followed by a drop in pressure, followed by a rise in pressure, and so on, corresponding to the back and forth vibrations of the tine of the tuning fork.

If we were to draw a graph of the change in air pressure detected by the microphone over time, we would see a sinusoidal shape (a *sine wave*) rising and falling, corresponding to the back and forth vibrations of the tuning fork.



Sinusoidal change in air pressure caused by a simple vibration back and forth

This continuous rise and fall in pressure creates a wave of sound. The amount of change in air pressure, with respect to normal atmospheric pressure, is called the wave's *amplitude* (literally, its "bigness"). We most commonly use the term "amplitude" to refer to the *peak amplitude*, the greatest change in pressure achieved by the wave.

This type of simple back and forth motion (seen also in the swing of a pendulum) is called *simple harmonic motion*. It's considered the simplest form of vibration because the object completes one full back-and-forth cycle at a constant rate. Even though its velocity changes when it slows down to change direction and then gains speed in the other direction—as shown by the curve of the sine wave—its average velocity from one cycle to the next is the same. Each complete vibratory cycle therefore occurs in an equal interval of time (in a given *period* of time), so the wave is said to be *periodic*. The number of cycles that occur in one second is referred to as the frequency of the vibration. For example, if the tine of the tuning fork goes back and forth 440 times per second, its *frequency* is 440 cycles per second, and its *period* is $^{1}/_{440}$ second per cycle.

In order for us to hear such fluctuations of pressure:

- The fluctuations must be substantial enough to affect our tympanic membrane (eardrum), yet not so substantial as to hurt us. In practice, the intensity of the changes in air pressure must be greater than about 10^{-9} times atmospheric pressure, but not greater than about 10^{-3} times atmospheric pressure. You'll never actually need that information, but there it is. It means that the softest sound we can hear has about one millionth the intensity of the loudest sound we can bear. That's quite a wide range of possibilities.

- The fluctuations must repeat at a regular rate fast enough for us to perceive them as a sound (rather than as individual events), yet not so fast that it exceeds our ability to hear it. Textbooks usually present this range of audible frequencies as 20 to 20,000 cycles per second (*cps*, also known as *hertz*, abbreviated *Hz*). Your own mileage may vary. If you are approaching middle age or have listened to too much loud music, you may top out at about 17,000 Hz or even lower.

Complex tones

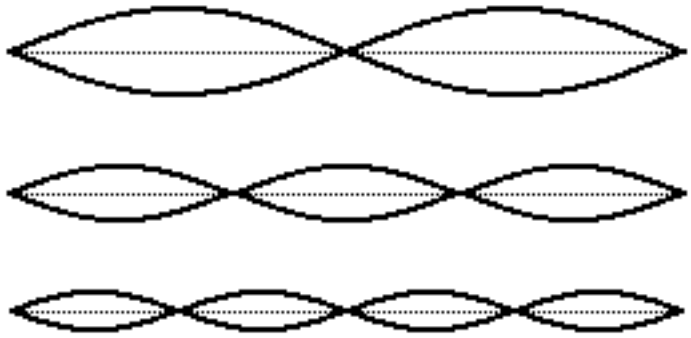
An object that vibrates in simple harmonic motion is said to have a resonant mode of vibration— a frequency at which it will naturally tend to vibrate when set in motion. However, most real- world objects have *several* resonant modes of vibration, and thus vibrate at many frequencies at once. Any sound that contains more than a single frequency (that is, any sound that is not a simple sine wave) is called a *complex tone*. Let's take a stretched guitar string as an example.

A guitar string has a uniform mass across its entire length, has a known length since it is fixed at both ends (at the "nut" and at the "bridge"), and has a given tension depending on how tightly it is tuned with the tuning peg. Because the string is fixed at both ends, it must always be stationary at those points, so it naturally vibrates most widely at its center.

A plucked string vibrating in its fundamental resonant mode

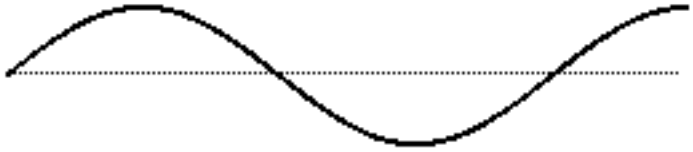
The frequency at which it vibrates depends on its mass, its tension, and its length. These traits stay fairly constant over the course of a note, so it has one fundamental frequency at which it vibrates.

However, other modes of vibration are still possible.



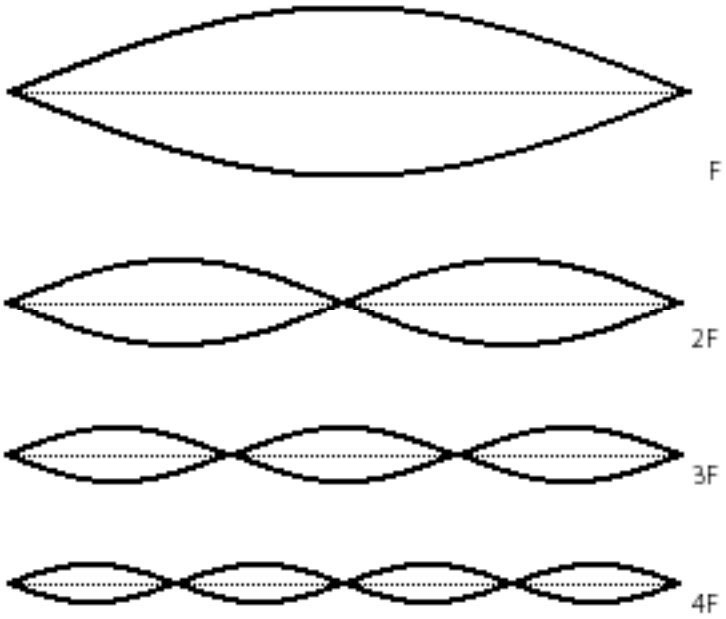
Some other resonant modes of a stretched string

The possible modes of vibration are constrained by the fact that the string must remain stationary at each end. This limits its modes of resonance to integer divisions of its length.



This mode of resonance would be impossible because the string is fixed at each end

Because the tension and mass are set, integer divisions of the string's length result in integer multiples of the fundamental frequency.



Each resonant mode results in a different frequency

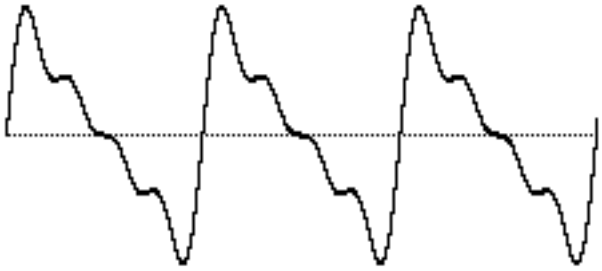
In fact, a plucked string will vibrate in all of these possible resonant modes simultaneously, creating energy at all of the corresponding frequencies. Of course, each mode of vibration (and thus each frequency) will have a different amplitude. (In the example of the guitar string, the longer segments of string have more freedom to vibrate.) The resulting tone will be the sum of all of these frequencies, each with its own amplitude.

As the string's vibrations die away due to the damping force of the fixture at each end, each frequency may die away at a different rate. In fact, in many sounds the amplitudes of the different component frequencies may vary quite separately and differently from each other. This variety seems to be one of the fundamental factors in our perception of sounds as having different *tone color* (i.e., *timbre*), and the timbre of even a single note may change drastically over the course of the note.

Harmonic tones

The combination of frequencies—and their amplitudes—that are present in a sound is called its *spectrum* (just as different frequencies and intensities of light constitute a color spectrum). Each individual frequency that goes into the makeup of a complex tone is called a *partial*. (It's one part of the whole tone.)

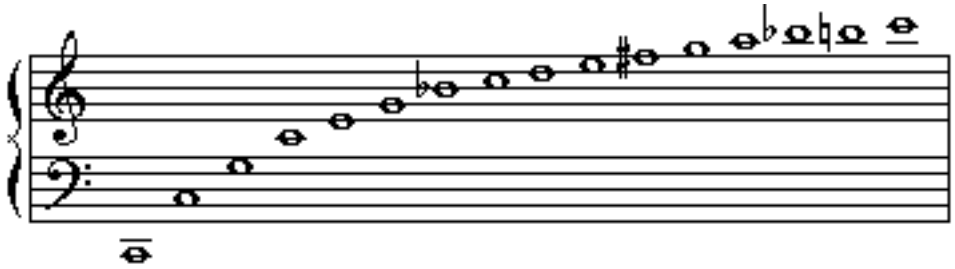
When the partials (component frequencies) in a complex tone are all integer multiples of the same fundamental frequency, as in our example of a guitar string, the sound is said to have a *harmonic spectrum*. Each component of a harmonic spectrum is called a *harmonic partial*, or simply a *harmonic*. The sum of all those harmonically related frequencies still results in a periodic wave having the fundamental frequency. The integer multiple frequencies thus fuse "harmoniously" into a single tone.



The sum of harmonically related frequencies still repeats at the fundamental frequency

This fusion is supported by the famous mathematical theorem of Jean-Baptiste Joseph Fourier, which states that any periodic wave, no matter how complex, can be demonstrated to be the sum of different harmonically related frequencies (sinusoidal waves), each having its own amplitude and phase. (*Phase* is an offset in time by some fraction of a cycle.)

Harmonically related frequencies outline a particular set of related pitches in our musical perception.



Harmonic partials of a fundamental frequency f, where f = 65.4 Hz = the pitch low C

Each time the fundamental frequency is multiplied by a power of 2—2, 4, 8, 16, etc.—the perceived musical pitch increases by one octave. All cultures seem to share the perception that there is a certain "sameness" of pitch class between such octave-related frequencies. The other integer multiples of the fundamental yield new musical pitches. Whenever you're hearing a harmonic complex tone, you're actually hearing a chord! As we've seen,

though, the combined result repeats at the fundamental frequency, so we tend to fuse these frequencies together such that we perceive a single pitch.

Inharmonic tones and noise

Some objects—such as a bell, for instance—vibrate in even more complex ways, with many different modes of vibrations which may not produce a harmonically related set of partials. If the frequencies present in a tone are not integer multiples of a single fundamental frequency, the wave does not repeat periodically. Therefore, an *inharmonic* set of partials does not fuse together so easily in our perception. We may be able to pick out the individual partials more readily, and—especially when the partials are many and are completely inharmonic—we may not perceive the tone as having a single discernible fundamental pitch.

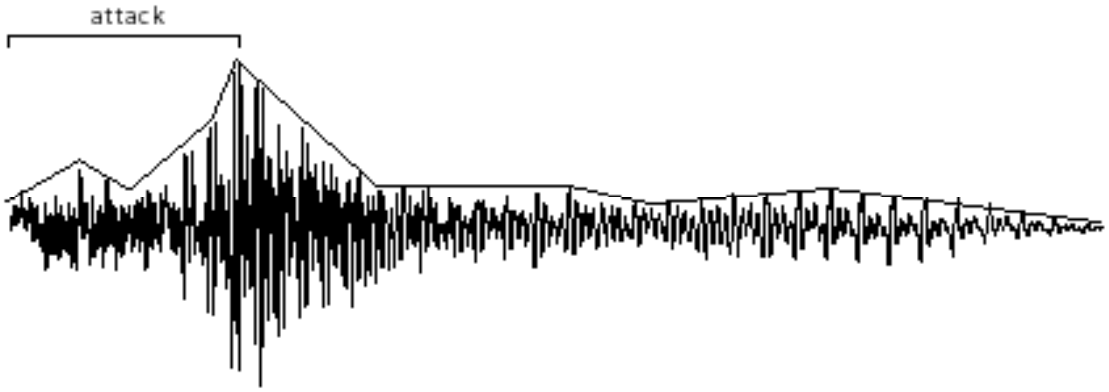
When a tone is so complex that it contains very many different frequencies with no apparent mathematical relationship, we perceive the sound as *noise*. A sound with many completely random frequencies and amplitudes—essentially all frequencies present in equal proportion—is the static-like sound known as *white noise* (analogous to white light which contains all frequencies of light).

So, it may be useful to think of sounds as existing on a continuum from total purity and predictability (a sine wave) to total randomness (white noise). Most sounds are between these two extremes. An harmonic tone—a trumpet or a guitar note, for example—is on the purer end of the continuum, while a cymbal crash is closer to the noisy end of the continuum. Timpani and bells may be just sufficiently suggestive of a harmonic spectrum that we can identify a fundamental pitch, yet they contain other inharmonic partials. Other drums produce more of a band-limited noise—randomly related frequencies, but restricted within a certain frequency range—giving a sense of pitch range, or non-specific pitch, rather than an identifiable fundamental. It is important to keep this continuum in mind when synthesizing sounds.

Amplitude envelope

Another important factor in the nearly infinite variety of sounds is the change in over-all amplitude of a sound over the course of its duration. The shape of this macroscopic over-all change in amplitude is termed the *amplitude envelope*. The initial portion of the sound, as the amplitude envelope increases from silence to audibility, rising to its peak amplitude, is known as the *attack* of the sound. The envelope, and especially the attack, of a sound are important factors in our ability to distinguish, recognize, and compare sounds. We have very little knowledge of how to read a graphic representation of a sound wave and hear the sound in our head the way a good sightreader can do with musical notation.

However, the amplitude envelope can at least tell us about the general evolution of the loudness of the sound over time.



The amplitude envelope is the evolution of a sound's amplitude over time

Amplitude and loudness

The relationship between the objectively measured amplitude of a sound and our subjective impression of its loudness is very complicated and depends on many factors. Without trying to explain all of those factors, we can at least point out that our sense of the relative loudness of two sounds is related to the ratio of their intensities, rather than the mathematical difference in their intensities. For example, on an arbitrary scale of measurement, the relationship between a sound of amplitude 1 and a sound of amplitude 0.5 is the same to us as the relationship between a sound of amplitude 0.25 and a sound of amplitude 0.125. The subtractive difference between amplitudes is 0.5 in the first case and 0.125 in the second case, but what concerns us perceptually is the ratio, which is 2:1 in both cases.

Does a sound with twice as great an amplitude sound twice as loud to us? In general, the answer is "no". First of all, our subjective sense of "loudness" is not directly proportional to amplitude. Experiments find that for most listeners, the (extremely subjective) sensation of a sound being "twice as loud" requires a much greater than twofold increase in amplitude. Furthermore, our sense of loudness varies considerably depending on the frequency of the sounds being considered. We're much more sensitive to frequencies in the range from about 300 Hz to 7,000 Hz than we are to frequencies outside that range. (This might possibly be due evolutionarily to the importance of hearing speech and many other important sounds which lie mostly in that frequency range.)

Nevertheless, there is a correlation—even if not perfectly linear—between amplitude and loudness, so it's certainly informative to know the relative amplitude of two sounds. As mentioned earlier, the softest sound we can hear has about one millionth the amplitude of the loudest sound we can bear. Rather than discuss amplitude using such a wide range of

numbers from 0 to 1,000,000, it is more common to compare amplitudes on a logarithmic scale.

The ratio between two amplitudes is commonly discussed in terms of *decibels* (abbreviated dB). A *level* expressed in terms of decibels is a statement of a ratio relationship between two values—not an absolute measurement. If we consider one amplitude as a reference which we call *A0*, then the relative amplitude of another sound in decibels can be calculated with the equation:

$$level\ in\ decibels = 20 \log10\ (A/A_0)$$

If we consider the maximum possible amplitude as a reference with a numerical value of 1, then a sound with amplitude 0.5 has $^1/_2$ the amplitude (equal to $10^{-0.3}$) so its level is

$$20 \log10\ (0.5/1) \_ 20\ (-0.3) = -6\ dB$$

Each halving of amplitude is a difference of about -6 dB; each doubling of amplitude is an increase of about 6 dB. So, if one amplitude is 48 dB greater than another, one can estimate that it's about 2^8 (256) times as great.

Summary

A theoretical understanding of sine waves, harmonic tones, inharmonic complex tones, and noise, as discussed here, is useful to understanding the nature of sound. However, most sounds are actually complicated combinations of these theoretical descriptions, changing from one instant to another. For example, a bowed string might include noise from the bow scraping against the string, variations in amplitude due to variations in bow pressure and speed, changes in the prominence of different frequencies due to bow position, changes in amplitude and in the fundamental frequency (and all its harmonics) due to vibrato movements in the left hand, etc. A drum note may be noisy but might evolve so as to have emphases in certain regions of its spectrum that imply a harmonic tone, thus giving an impression of fundamental pitch. Examination of existing sounds, and experimentation in synthesizing new sounds, can give insight into how sounds are composed. The computer provides that opportunity.

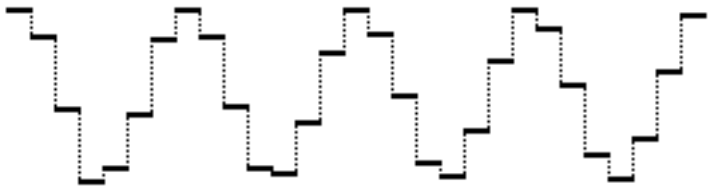
Digital representation of sound

Sampling and quantizing a sound wave

To understand how a computer represents sound, consider how a film represents motion. A movie is made by taking still photos in rapid sequence at a constant rate, usually twenty-four frames per second. When the photos are displayed in sequence at that same rate, it fools us into thinking we are seeing *continuous* motion, even though we are actually seeing twenty-four *discrete* images per second. Digital recording of sound works

on the same principle. We take many discrete samples of the sound wave's instantaneous amplitude, store that information, then later reproduce those amplitudes at the same rate to create the illusion of a continuous wave.

The job of a microphone is to transduce (convert one form of energy into another) the change in air pressure into an analogous change in electrical voltage. This continuously changing voltage can then be sampled periodically by a process known as *sample and hold*. At regularly spaced moments in time, the voltage at that instant is sampled and held constant until the next sample is taken. This reduces the total amount of information to a certain number of discrete voltages.



Time-varying voltage sampled periodically

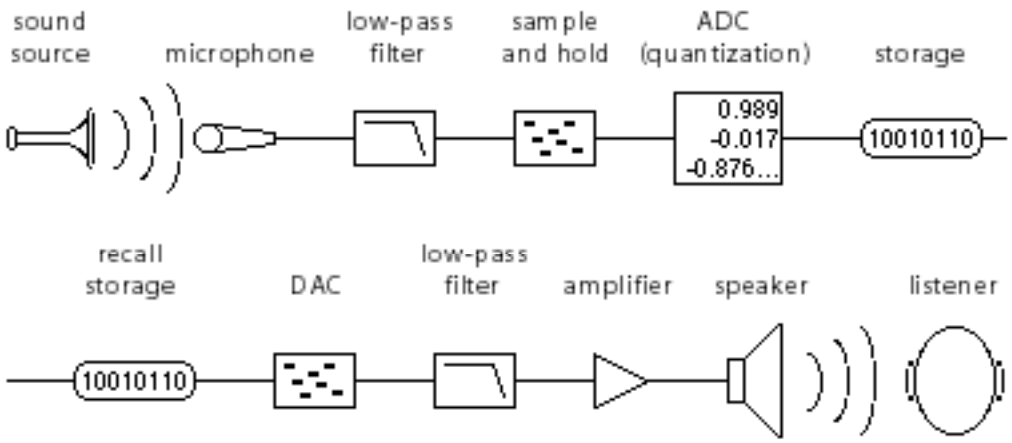
A device known as an *analog-to-digital converter* (ADC) receives the discrete voltages from the sample and hold device, and ascribes a numerical value to each amplitude. This process of converting voltages to numbers is known as *quantization*. Those numbers are expressed in the computer as a string of binary digits (1 or 0). The resulting binary numbers are stored in memory — usually on a digital audio tape, a hard disk, or a laser disc. To play the sound back, we read the numbers from memory, and deliver those numbers to a *digital-to-analog converter* (DAC) at the same rate at which they were recorded. The DAC converts each number to a voltage, and communicates those voltages to an amplifier to increase the amplitude of the voltage.

In order for a computer to represent sound accurately, many samples must be taken per second— many more than are necessary for filming a visual image. In fact, we need to take more than twice as many samples as the highest frequency we wish to record. (For an explanation of why this is so, see *Limitations of Digital Audio* on the next page.) If we want to record frequencies as high as 20,000 Hz, we need to sample the sound at least 40,000 times per second. The standard for compact disc recordings (and for "CD-quality" computer audio) is to take 44,100 samples per second for each channel of audio. The number of samples taken per second is known as the *sampling rate*.

This means the computer can only accurately represent frequencies up to half the sampling rate. Any frequencies in the sound that exceed half the sampling rate must be filtered out before the sampling process takes place. This is accomplished by sending the electrical signal through a *low-pass filter* which removes any frequencies above a certain

threshold. Also, when the digital signal (the stream of binary digits representing the quantized samples) is sent to the DAC to be re-converted into a continuous electrical signal, the sound coming out of the DAC will contain spurious high frequencies that were created by the sample and hold process itself. (These are due to the "sharp edges" created by the discrete samples, as seen in the above example.) Therefore, we need to send the output signal through a low-pass filter, as well.

The digital recording and playback process, then, is a chain of operations, as represented in the following diagram.



Digital recording and playback process

Limitations of digital audio

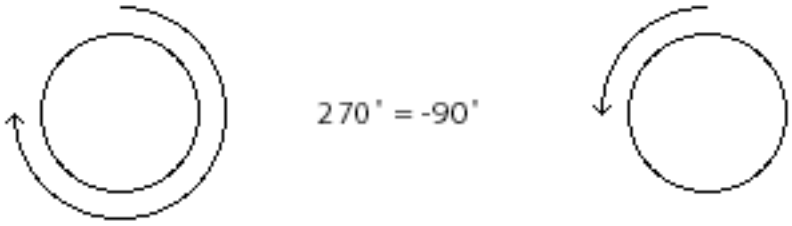
Sampling rate and Nyquist rate

We've noted that it's necessary to take at least twice as many samples as the highest frequency we wish to record. This was proven by Harold Nyquist, and is known as the *Nyquist theorem*. Stated another way, the computer can only accurately represent frequencies up to half the sampling rate. One half the sampling rate is often referred to as the *Nyquist frequency* or the *Nyquist rate*.

If we take, for example, 16,000 samples of an audio signal per second, we can only capture frequencies up to 8,000 Hz. Any frequencies higher than the Nyquist rate are perceptually "folded" back down into the range below the Nyquist frequency. So, if the sound we were trying to sample contained energy at 9,000 Hz, the sampling process would misrepresent that frequency as 7,000 Hz—a frequency that might not have been present at all in the original sound. This effect is known as *foldover* or *aliasing*. The main problem with

aliasing is that it can add frequencies to the digitized sound that were not present in the original sound, and unless we know the exact spectrum of the original sound there is no way to know which frequencies truly belong in the digitized sound and which are the result of aliasing. That's why it's essential to use the low-pass filter before the sample and hold process, to remove any frequencies above the Nyquist frequency.

To understand why this aliasing phenomenon occurs, think back to the example of a film camera, which shoots 24 frames per second. If we're shooting a movie of a car, and the car wheel spins at a rate greater than 12 revolutions per second, it's exceeding half the "sampling rate" of the camera. The wheel completes more than $^{1}/_{2}$ revolution per frame. If, for example it actually completes $^{18}/_{24}$ of a revolution per frame, it will appear to be going backward at a rate of 6 revolutions per second. In other words, if we don't witness what happens between samples, a 270˚ revolution of the wheel is indistinguishable from a -90˚ revolution. The samples we obtain in the two cases are precisely the same.

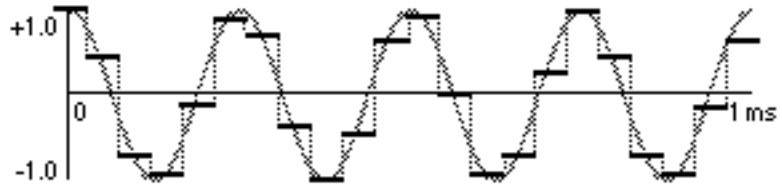


For the camera, a revolution of 18/24 is no different from a revolution of -6/24

For audio sampling, the phenomenon is practically identical. Any frequency that exceeds the Nyquist rate is indistinguishable from a *negative* frequency the same amount less than the Nyquist rate. (And we do not distinguish perceptually between positive and negative frequencies.) To the extent that a frequency exceeds the Nyquist rate, it is folded back down from the Nyquist frequency by the same amount.

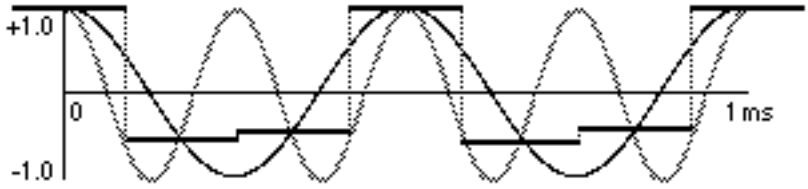
For a demonstration, consider the next two examples. The following example shows a graph of a 4,000 Hz cosine wave (energy only at 4,000 Hz) being sampled at a rate of 22,050 Hz. 22,050 Hz is half the CD sampling rate, and is an acceptable sampling rate for sounds that do not have much energy in the top octave of our hearing range.

In this case the sampling rate is quite adequate because the maximum frequency we are trying to record is well below the Nyquist frequency.



A 4,000 Hz cosine wave sampled at 22,050 Hz

Now consider the same 4,000 Hz cosine wave sampled at an inadequate rate, such as 6,000 Hz. The wave completes more than $^1/_2$ cycle per sample, and the resulting samples are indistinguishable from those that would be obtained from a 2,000 Hz cosine wave.



A 4,000 Hz cosine wave undersampled at 6,000 Hz

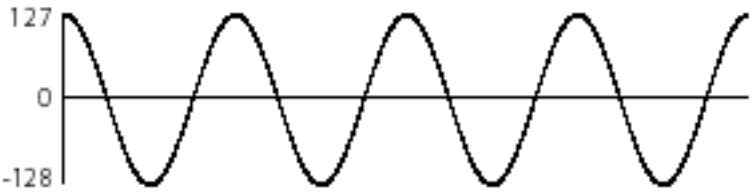
The simple lesson to be learned from the Nyquist theorem is that digital audio cannot accurately represent any frequency greater than half the sampling rate. Any such frequency will be misrepresented by being folded over into the range below half the sampling rate.

Precision of quantization

Each sample of an audio signal must be ascribed a numerical value to be stored in the computer. The numerical value expresses the *instantaneous* amplitude of the signal at the moment it was sampled. The range of the numbers must be sufficiently large to express adequately the entire amplitude range of the sound being sampled.

The range of possible numbers used by a computer depends on the number of binary digits (*bits*) used to store each number. A bit can have one of two possible values: either 1 or 0. Two bits together can have one of four possible values: 00, 01, 10, or 11. As the number of bits increases, the range of possible numbers they can express increases by a power of two. Thus, a single byte (8 bits) of computer data can express one of $2^8 = 256$ possible numbers. If we use two bytes to express each number, we get a much greater range of possible values because $2^{16} = 65{,}536$.

The number of bits used to represent the number in the computer is important because it determines the *resolution* with which we can measure the amplitude of the signal. If we use only one byte to represent each sample, then we must divide the entire range of possible amplitudes of the signal into 256 parts since we have only 256 ways of describing the amplitude.



Using one byte per sample, each sample can have one of only 256 different possible values

For example, if the amplitude of the electrical signal being sampled ranges from -10 volts to +10 volts and we use one byte for each sample, each number does not represent a precise voltage but rather a 0.078125 V portion of the total range. Any sample that falls within that portion will be ascribed the same number. This means each numerical description of a sample's value could be off from its actual value by as much as 0.078125V—$^{1}/_{256}$ of the total amplitude range. In practice each sample will be off by some random amount from 0 to $^{1}/_{256}$ of the total amplitude range. The mean error will be $^{1}/_{512}$ of the total range.

This is called *quantization error*. It is unavoidable, but it can be reduced to an acceptable level by using more bits to represent each number. If we use two bytes per sample, the quantization error will never be greater than $^{1}/_{65,536}$ of the total amplitude range, and the mean error will be $^{1}/_{131,072}$.

Since the quantization error for each sample is usually random (sometimes a little to high, sometimes a little too low), we generally hear the effect of quantization error as white noise. This noise is not present in the original signal. It is added into the digital signal by the imprecise nature of quantization. This is called *quantization noise.*

The ratio of the total amplitude range to the quantization error is called the *signal-to-quantization- noise-ratio* (SQNR). This is the ratio of the maximum possible signal amplitude to the average level quantization of the quantization noise, and is usually stated in decibels.

As a rule of thumb, each bit of precision used in quantization adds 6 dB to the SQNR. Therefore, sound quantized with 8-bit numerical precision will have a best case SQNR of about 48 dB. This is adequate for cases where fidelity is not important, but is certainly not desirable for music or other critical purposes. Sound sampled with 16-bit precision ("CD-

quality") has a SQNR of 96 dB, which is quite good—much better than traditional tape recording.

In short, the more bits used by the computer to store each sample, the better the potential ratio of signal to noise.

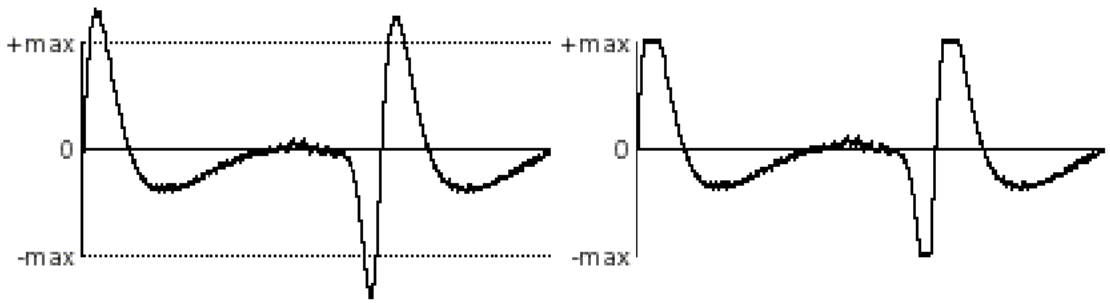
Memory and storage

We have seen that the standard sampling rate for high-fidelity audio is 44,100 samples per second. We've also seen that 16 bits (2 bytes) are needed per sample to achieve a good signal-to-noise ratio. With this information we can calculate the amount of data needed for digital audio: 41,000 samples per second, times 2 bytes per sample, times 2 channels for stereo, times 60 seconds per minute equals more than 10 megabytes of data per minute of CD-quality audio.

For this quality of audio, a high-density floppy disk holds less than 8 seconds of sound, and a 100 MB Zip cartridge holds less than 10 minutes. Clearly, the memory and storage requirements of digital audio are substantial. Fortunately, a compact disc holds over an hour of stereo sound, and a computer hard disk of at least 1 gigabyte is standard for audio recording and processing.

Clipping

If the amplitude of the incoming electrical signal exceeds the maximum amplitude that can be expressed numerically, the digital signal will be a clipped-off version of the actual sound.



A signal that exceeds maximum amplitude will be clipped when it is quantized

The clipped sample will often sound quite different from the original. Sometimes this type of clipping causes only a slight distortion of the sound that is heard as a change in timbre. More often though, it sounds like a very unpleasant noise added to the sound. For this reason, it's very important to take precautions to avoid clipping. The amplitude of the electrical signal should not exceed the maximum expected by the ADC.

It's also possible to produce numbers in the computer that exceed the maximum expected by the DAC. This will cause the sound that comes out of the DAC to be a clipped version of the digital signal. Clipping by the DAC is just as bad as clipping by the ADC, so care must be taken not to generate a digital signal that goes beyond the numerical range the DAC is capable of handling.

Advantages of digital audio

Synthesizing digital audio

Since a digital representation of sound is just a list of numbers, any list of numbers can theoretically be considered a digital representation of a sound. In order for a list of numbers to be audible as sound, the numerical values must fluctuate up and down at an audio rate. We can listen to any such list by sending the numbers to a DAC where they are converted to voltages. This is the basis of computer sound synthesis. Any numbers we can generate with a computer program, we can listen to as sound.

Many methods have been discovered for generating numbers that produce interesting sounds. One method of producing sound is to write a program that repeatedly solves a mathematical equation containing two variables. At each repetition, a steadily increasing value is entered for one of the variables, representing the passage of time. The value of the other variable when the equation is solved is used as the amplitude for each moment in time. The output of the program is an amplitude that varies up and down over time.

For example, a sine wave can be produced by repeatedly solving the following algebraic equation, using an increasing value for *n*:

$$y = A \sin(2\_fn/R+ø)$$

where *A* is the amplitude of the wave, *f* is the frequency of the wave, *n* is the sample number (0,1, 2,3, etc.), *R* is the sampling rate, and *ø* is the phase. If we enter values for *A*, *f*,and *ø*, and repeatedly solve for *y* while increasing the value of *n*, the value of *y* (the output sample) will vary sinusoidally.

A complex tone can be produced by adding sinusoids—a method known as *additive synthesis*:

$$y = A1 \sin(2\_f1n/R+ø1) + A2 \sin(2\_f2n/R+ø2) + ...$$

This is an example of how a single algebraic expression can produce a sound. Naturally, many other more complicated programs are possible. A few synthesis methods such as additive synthesis, wavetable synthesis, frequency modulation, and waveshaping are demonstrated in the *MSP Tutorial.*

Manipulating digital signals

Any sound in digital form—whether it was synthesized by the computer or was quantized from a "real world" sound—is just a series of numbers. Any arithmetic operation performed with those numbers becomes a form of audio processing.

For example, multiplication is equivalent to audio amplification. Multiplying each number in a digital signal by 2 doubles the amplitude of the signal (increases it 6 dB). Multiplying each number in a signal by some value between 0 and 1 reduces its amplitude.

Addition is equivalent to audio mixing. Given two or more digital signals, a new signal can be created by adding the first numbers from each signal, then the second numbers, then the third numbers, and so on.

An echo can be created by recalling samples that occurred earlier and adding them to the current samples. For example, whatever signal was sent out 1000 samples earlier could be sent out again, combined with the current sample.

$$y = xn + A\ yn\text{-}1000$$

As a matter of fact, the effects that such operations can have on the shape of a signal (audio or any other kind) are so many and varied that they comprise an entire branch of electrical engineering called digital signal processing (DSP). DSP is concerned with the effects of digital filters—formulae for modifying digital signals by combinations of delay, multiplication, addition, and other numerical operations.

Summary

This chapter has described how the continuous phenomenon of sound can be captured and faithfully reproduced as a series of numbers, and ultimately stored in computer memory as a stream of binary digits. There are many benefits obtainable only by virtue of this *digital* representation of sound: higher fidelity recording than was previously possible, synthesis of new sounds by mathematical procedures, application of digital signal processing techniques to audio signals, etc.

MSP provides a toolkit for exploring this range of possibilities. It integrates digital audio recording, synthesis, and processing with the MIDI control and object-based programming of Max.

How MSP Works: Max Patches and the MSP Signal Network

Introduction

Max objects communicate by sending each other messages through patch cords. These messages are sent at a specific moment, either in response to an action taken by the user (a mouse click, a MIDI note played, etc.) or because the event was scheduled to occur (by **metro**, **delay**, etc.).

MSP objects are connected by patch cords in a similar manner, but their inter-communication is conceptually different. Rather than establishing a path for messages to be sent, MSP connections establish a relationship between the connected objects, and that relationship is used to calculate the audio information necessary at any particular instant. This configuration of MSP objects is known as the *signal network*.

The following example illustrates the distinction between a Max patch in which messages are sent versus a signal network in which an ongoing relationship is established.

Max messages occur at a specific instant; MSP objects are in constant communication

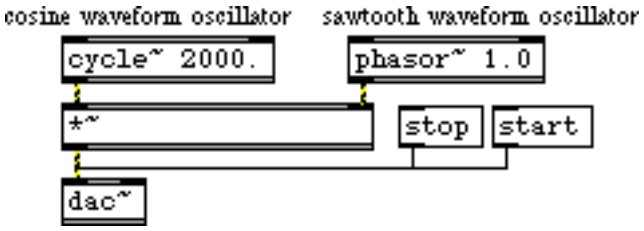
In the Max example on the left, the **number box** doesn't know about the number 0.75 stored in the **float** object. When the user clicks on the **button**, the **float** object sends out its stored value. Only then does the **number box** receive, display, and send out the number 0.75. In the MSP example on the right, however, each outlet that is connected as part of the signal network is constantly contributing its current value to the equation. So, even without any specific Max message being sent, the ***~** object is receiving the output from the two **sig~** objects, and any object connected to the outlet of ***~** would be receiving the product 0.75.

Another way to think of a MSP signal network is as a portion of a patch that runs at a faster (audio) rate than Max. Max, and you the user, can only directly affect that signal portion of the patch every millisecond. What happens in between those millisecond intervals is calculated and performed by MSP. If you think of a signal network in this way—as a very fast patch—then it still makes sense to think of MSP objects as "sending" and "receiving" messages (even though those messages are sent faster than Max can see them), so we will continue to use standard Max terminology such as *send, receive, input,* and *output* for MSP objects.

Audio rate and control rate

The basic unit of time for scheduling events in Max is the millisecond (0.001 seconds). This rate—1000 times per second—is generally fast enough for any sort of control one might want to exert over external devices such as synthesizers, or over visual effects such as QuickTime movies.

Digital audio, however, must be processed at a much faster rate—commonly 44,100 times per second per channel of audio. The way MSP handles this is to calculate, on an ongoing basis, all the numbers that will be needed to produce the next few milliseconds of audio. These calculations are made by each object, based on the configuration of the signal network.



An oscillator (cycle~), and an amplifier (~) controlled by another oscillator (phasor~)*

In this example, a cosine waveform oscillator with a frequency of 2000 Hz (the **cycle~** object) has its amplitude scaled (every sample is multiplied by some number in the ***~** object) then sent to the digital-to-analog converter (**dac~**). Over the course of each second, the (sub-audio) sawtooth wave output of the **phasor~** object sends a continuous ramp of increasing values from 0 to 1. Those increasing numbers will be used as the right operand in the ***~** for each sample of the audio waveform, and the result will be that the 2000 Hz tone will fade in linearly from silence to full amplitude each second. For each millisecond of audio, MSP must produce about 44 sample values (assuming an audio sample rate of 44,100 Hz), so for each sample it must look up the proper value in each oscillator and multiply those two values to produce the output sample.

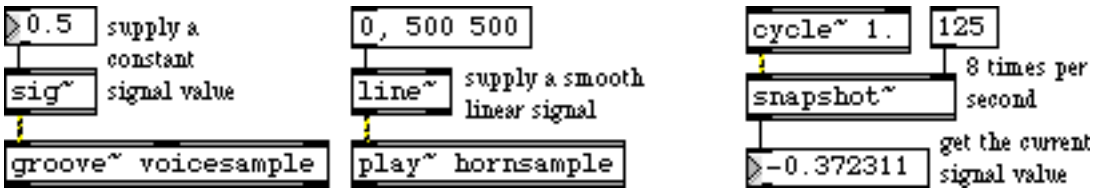
Even though many MSP objects accept input values expressed in milliseconds, they calculate samples at an audio sampling rate. Max messages travel much more slowly, at what is often referred to as a *control* rate. It is perhaps useful to think of there being effectively two different rates of activity: the slower *control* rate of Max's scheduler, and the faster audio *sample* rate.

Note: Since you can specify time in Max in floating-point milliseconds, the resolution of the scheduler varies depending on how often it runs. The exact control rate is set by a number of MSP settings we'll introduce shortly. However, it is far less efficient to

"process" audio using the "control" functions running in the scheduler than it is to use the specialized audio objects in MSP.

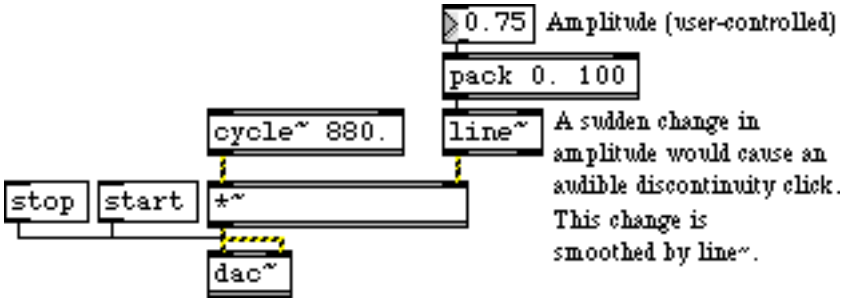
The link between Max and MSP

Some MSP objects exist specifically to provide a link between Max and MSP—and to translate between the control rate and the audio rate. These objects (such as **sig~** and **line~**) take Max messages in their inlets, but their outlets connect to the signal network; or conversely, some objects (such as **snapshot~**) connect to the signal network and can peek (but only as frequently as once per millisecond) at the value(s) present at a particular point in the signal network.



Supply a Max message to the signal network, or get a Max message from a signal

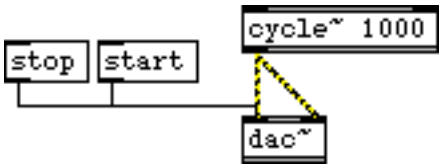
These objects are very important because they give Max, and you the user, direct control over what goes on in the signal network.



User interface control over the signal's amplitude

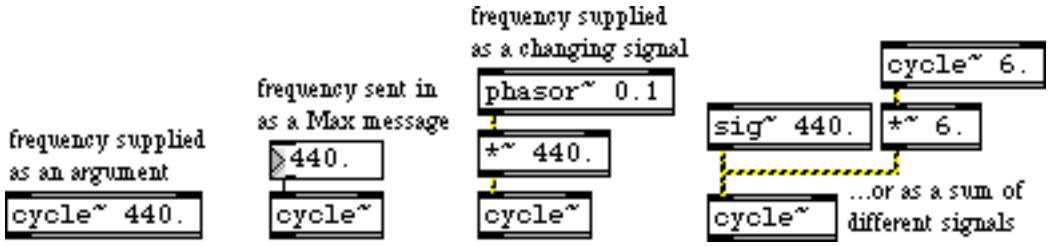
Some MSP object inlets accept both signal input and Max messages. They can be connected as part of a signal network, and they can also receive instructions or modifications via Max messages.

For example the **dac~** (digital-to-analog converter) object, for playing the audio signal, can be turned on and off with the Max messages start and stop.



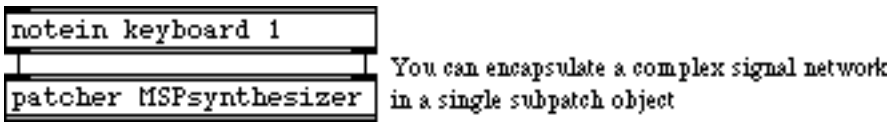
Some MSP objects can receive audio signals and Max messages in the same inlet

And the **cycle~** (oscillator) object can receive its frequency as a Max float or int message, or it can receive its frequency from another MSP object (although it can't do both at the same time, because the audio input can be thought of as constantly supplying values that would immediately override the effect of the float or int message).



Some MSP objects can receive either Max messages or signals for the same purpose

So you see that a Max patch (or subpatch) may contain both Max objects and MSP objects. For clear organization, it is frequently useful to encapsulate an entire process, such as a signal network, in a subpatch so that it can appear as a single object in another Max patch.



Encapsulation can clarify relationships in a Max patch

Limitations of MSP

From the preceding discussion, it's apparent that digital audio processing requires a lot of "number crunching". The computer must produce tens of thousands of sample values per second per channel of sound, and each sample may require many arithmetic calculations, depending on the complexity of the signal network. And in order to produce *realtime* audio, the samples must be calculated at least as fast as they are being played.

Realtime sound synthesis of this complexity on a general-purpose personal computer was pretty much out of the question until the introduction of sufficiently fast processors such as the PowerPC. Even with the PowerPC, though, this type of number crunching requires a great deal of the processor's attention. So it's important to be aware that there are limitations to how much your computer can do with MSP.

Unlike a MIDI synthesizer, in MSP you have the flexibility to design something that is too complicated for your computer to calculate in real time. The result can be audio distortion, a very unresponsive computer, or in extreme cases, crashes.

Because of the variation in processor performance between computers, and because of the great variety of possible signal network configurations, it's difficult to say precisely what complexity of audio processing MSP can or cannot handle. Here are a few general principles:

- The faster your computer's CPU, the better will be the performance of MSP. We strongly recommend computers that use the PowerPC 604 or newer processors. Older PowerBook models such as the 5300 series are particularly ill-suited to run MSP, and are not recommended.

- A fast hard drive and a fast SCSI connection will improve input/output of audio files, although MSP will handle up to about eight tracks at once on most computers with no trouble.

- Turning off background processes (such as file sharing) will improve performance.

- Reducing the audio sampling rate will reduce how many numbers MSP has to compute for a given amount of sound, thus improving its performance (although a lower sampling rate will mean degradation of high frequency response). Controlling the audio sampling rate is discussed in the *Audio Input and Output* chapter.

When designing your MSP instruments, you should bear in mind that some objects require more intensive computation than others. An object that performs only a few simple arithmetic operations (such as **sig~**, **line~**, **+~**, **-~**, ***~**, or **phasor~**) is computationally inexpensive. (However, **/~** is much more expensive.) An object that looks up a number in a function table and interpolates between values (such as **cycle~**) requires only a few calculations, so it's likewise not too expensive. The most expensive objects are those which must perform many calculations per sample: filters (**reson~**, **biquad~**), spectral analyzers (**fft~**, **ifft~**), and objects such as **play~**, **groove~**, **comb~**, and **tapout~** when one of their parameters is controlled by a continuous signal. Efficiency issues are discussed further in the *MSP Tutorial*.

Note: To see how much of the processor's time your patch is taking, look at the *CPU Utilization* value in the DSP Status window. Choose **DSP Status…** from the Options menu to open this window.

Advantages of MSP

The PowerPC is a general purpose computer, not a specially designed sound processing computer such as a commercial sampler or synthesizer, so as a rule you can't expect it to perform quite to that level. However, for relatively simple instrument designs that meet specific synthesis or processing needs you may have, or for experimenting with new audio processing ideas, it is a very convenient instrument-building environment.

1. *Design an instrument to fit your needs*. Even if you have a lot of audio equipment, it probably cannot do every imaginable thing you need to do. When you need to accomplish a specific task not readily available in your studio, you can design it yourself.

2. *Build an instrument and hear the results in real time*. With non-realtime sound synthesis programs you define an instrument that you think will sound the way you want, then compile it and test the results, make some adjustments, recompile it, etc. With MSP you can hear each change that you make to the instrument as you build it, making the process more interactive.

3. *Establish the relationship between gestural control and audio result*. With many commercial instruments you can't change parameters in real time, or you can do so only by programming in a complex set of MIDI controls. With Max you can easily connect MIDI data to the exact parameter you want to change in your MSP signal network, and you know precisely what aspect of the sound you are controlling with MIDI.

4. *Integrate audio processing into your composition or performance programs*. If your musical work consists of devising automated composition programs or computer-assisted performances in Max, now you can incorporate audio processing into those programs. Need to do a hands- free crossfade between your voice and a pre-recorded sample at a specific point in a performance? You can write a Max patch with MSP objects that does it for you, triggered by a single MIDI message.

Some of these ideas are demonstrated in the MSP tutorials.

See Also

Audio I/O Audio input and output with MSP

Audio I/O: Audio input and output with MSP

MSP interfaces with your computer's audio hardware via the **dac~** and **adc~** objects and their easy-to-use equivalents **ezdac~** and **ezadc~**. If you don't have any special audio hardware and have no need for inter-application audio routing, the default driver on your system will give you stereo full-duplex audio I/O with no special configuration on your part.

In addition to Core Audio or MME on Windows, there are a number of other ways to get audio into and out of Max/MSP. Each of these methods involves using what we call a *driver*, which is actually a special type of Max object. Some of these drivers facilitate the use of MSP with third- party audio hardware. Also, a non real-time driver allows you to use MSP as a disk-based audio processing and synthesis system, removing the limit of how much processing you can do with your CPU in real time.

MSP audio driver objects are located in the *ad* folder located in the /Library/Application Support/ Cycling '74 folder on Macintosh or in the C:\Program Files\Common Files\Cycling '74\ad folder on Windows. These object files must be in this folder called *ad* (which stands for *audio driver*), otherwise MSP will be unable to locate them.

We will begin with a discussion of audio input/output in MSP in general. Later in this chapter we will discuss aspects of specific audio drivers that are available to you in MSP. First we'll discuss the DSP Status window and how to use it to get information about your audio hardware and set parameters for how MSP handles audio input and output.

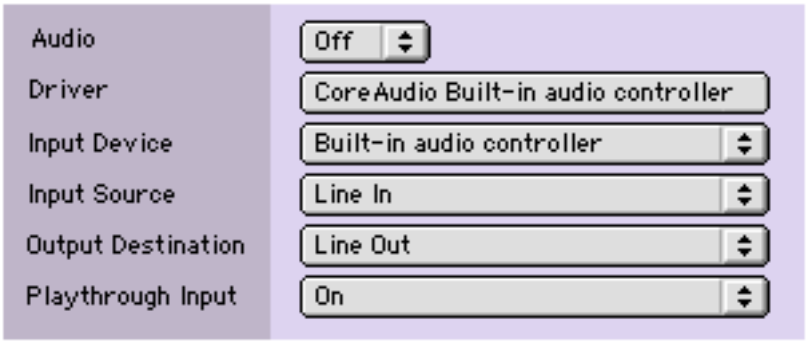
The DSP Status Window

All global audio parameters in MSP are displayed in the DSP Status window. To open the DSP Status window, just double-click on any **dac~** or **adc~** object in a locked Patcher window. Alternately, you can choose DSP Status… from the Options menu.

The DSP Status window is arranged as a group of menus and checkboxes that set all of the parameters of the audio input and output in MSP. Since all of these options can be changed from within your patch (see below), the DSP Status window serves as a monitor for your current audio settings as well.

Technical Detail: The DSP Status window is in fact a Max patch (called DSP Status, in the *patches* subfolder of Max). Every parameter shown in the DSP Status window is a menu or checkbox hooked up to an instance of the **adstatus** object. The **adstatus** object can be used inside of your MSP patches so that you can set and restore audio parameters specifically for certain patches. The **adstatus** object is also useful for obtaining information current CPU load, vector size, and sampling rate. See the **adstatus** object manual pages in the MSP Reference Manual for more details.

At the very top of the DSP Status window is a pop-up menu for turning the audio in MSP on and off. If you use another method to turn the audio on or off, the menu will update to reflect the current state.



The second pop-up menu allows you to view and select an audio driver for MSP. The specific audio drivers will be discussed later in this chapter. A brief summary will suffice for now:

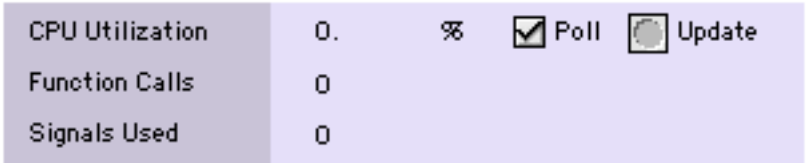
None	This setting shuts off MSP audio processing.
Core Audio	This is the default audio driver for MSP on Macintosh. It interfaces with the system's built-in Core Audio system and can be used with the built-in audio of the computer, or, with the proper software support, a third-party hardware interface, such as ASIO.
MME or DirectSound	(Windows only) On Windows, MSP loads the MME driver by default. If you have correctly installed external hardware and it also supports DirectSound, it should also appear as an option on the pop-up menu.

ad_rewire	This driver supports a standard developed by Propellerhead Software that allows sound generating applications (ReWire Devices) to send multiple channels of audio and midi to other applications (ReWire Mixers) that process and output it. Selecting the ad_rewire driver enables Max/MSP to function as a ReWire Device to route audio from MSP into applications that support ReWire (such as Live, Digital Performer or Cubase). Using MSP to host ReWire devices (such as Reason) can be accomplished with the **rewire~** object.
ASIO	(Windows only) If you have a third-party audio interface which supports ASIO (a cross-platform audio hardware standard developed by Steinberg), and it is installed correctly, it will be found by the MSP ASIO driver. You may have as many ASIO devices as you wish; they will all be found by the driver and will appear in the Driver pull-down menu in the DSP Status Window preceded by the word ASIO.
NonRealTime	This driver enables MSP to work in non real-time mode, allowing you to synthesize and process audio without any real-time processor performance limitations. Real-time audio input and output are disabled under this driver.

Only one audio driver can be selected at any given time. MSP saves the settings for each audio driver separately and will recall the last used audio driver when you restart Max.

The next two pop-up menus are active only when using the Core Audio driver on Macintosh or ASIO drivers. When the Core Audio driver or either the MME or DirectSound drivers on Windows are selected, the pop-up menus allow you to change the audio input source. On Macintosh only, an additional pop-up menu lets you choose whether or not audio playthrough is enabled. These settings can also be changed using the Audio MIDI Setup application on Macintosh or the Sounds and Audio Devices Properties window (Start – Settings – Control Panel – Sounds and Audio Devices) on Windows, but only with these menus while MSP is running.

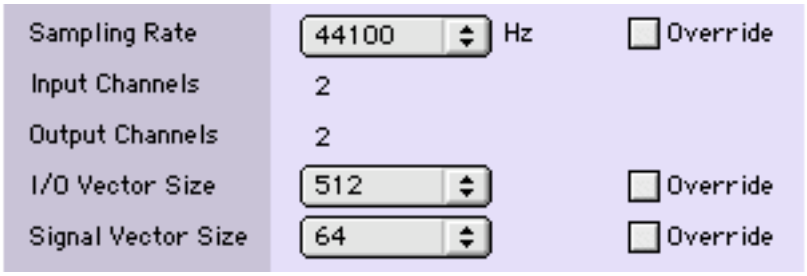
When ASIO is in use, the pop-up menus allow you to set the clock source for your audio hardware and whether or not to prioritize MIDI input and output over audio I/O.



The next three fields in the DSP Status window monitor the amount of signal processing MSP is currently doing. The *CPU Utilization* field displays a rough estimate of how much of your computer's CPU is being allocated for crunching audio in MSP. The Poll checkbox turns on and off the CPU Utilization auto-polling feature (it will update automatically four times a second when this is checked). If you turn off auto-polling, you can update the CPU readout manually by clicking on the Update button.

The number of *Function Calls* gives an approximate idea of how many calculations are being required for each sample of audio. The number next to *Signals Used* shows the number of internal buffers that were needed by MSP to connect the signal objects used in the current signal network.Both of these fields will update whenever you change the number of audio objects or how they are patched together.

The next two sections have *Override* checkboxes next to a number of the pop-up menus. When checked, Override means that the setting you pick will not be saved in the preferences file for the current audio driver. By default, all Overrides are disabled, meaning that the currently displayed settings will be saved and restored the next time you launch Max/MSP.



You can set the audio sampling rate with the *Sampling Rate* pop-up menu. For full-range audio, the recommended sampling rate is 44.1 kHz. Using a lower rate will reduce the number of samples that MSP has to calculate, thus lightening your computer's burden, but it will also reduce the frequency range. If your computer is struggling at 44.1 kHz, you should try a lower rate.

The *I/O Vector Size* may have an effect on latency and overall performance. A smaller vector size may reduce the inherent delay between audio input and audio output, because

MSP has to perform calculations for a smaller chunk of time. On the other hand, there is an additional computational burden each time MSP prepares to calculate another vector (the next chunk of audio), so it is easier over-all for the processor to compute a larger vector. However, there is another side to this story. When MSP calculates a vector of audio, it does so in what is known as an interrupt. If MSP is running on your computer, whatever you happen to be doing (word processing, for example) is interrupted and an I/O vector's worth of audio is calculated and played. Then the computer returns to its normally scheduled program. If the vector size is large enough, the computer may get a bit behind and the audio output may start to click because the processing took longer than the computer expected. Reducing the I/O Vector Size may solve this problem, or it may not. On the other hand, if you try to generate too many interrupts, the computer will slow down trying to process them (saving what you are doing and starting another task is hard work). Therefore, you'll typically find the smaller I/O Vector Sizes consume a greater percentage of the computer's resources. Optimizing the performance of any particular signal network when you are close to the limit of your CPU's capability is a trial-and-error process. That's why MSP provides you with a choice of vector sizes.

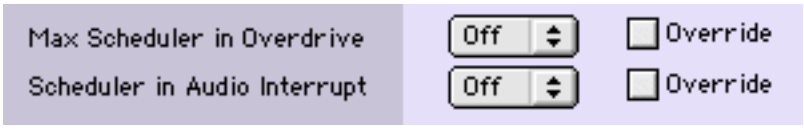
> **Technical Detail:** Some audio interface cards do not provide a choice of I/O Vector Sizes. There are also some ASIO drivers whose selection of I/O Vector Sizes may not conform to the multiple- of-a-power-of-2 limitation currently imposed by MSP's ASIO support. In some cases, this limitation can be remedied by using the ASIO driver at a different sampling rate.

Changing the vector sizes does not affect the actual quality of the audio itself, unlike changing the sampling rate, which affects the high frequency response. Changing the signal vector size won't have any effect on latency, and will have only a slight effect on overall performance (the larger the size, the more performance you can expect). However, certain types of algorithms benefit from a small signal vector size. For instance, the minimum delay you can get from MSP's delay line objects **tapin~** and **tapout~** is equal to the number of samples in one signal vector at the current sampling rate. With a signal vector size of 64 at 44.1 kHz sampling rate, this is 1.45 milliseconds, while at a signal vector size of 1024, it is 23.22 milliseconds. The Signal Vector size in MSP can be set as low as 2 samples, and in most cases can go as high as the largest available I/O Vector Size for your audio driver. However, if the I/O Vector Size is not a power of 2, the maximum signal vector size is the largest power of 2 that divides evenly into the I/O vector size.

> **Technical Detail:** Subpatches loaded into the **poly~** object can function at different sampling rates and vector sizes from the top-level patch. In addition, the **poly~** object allows up- and down-sampling as well as different vector sizes. The DSP Status window only displays and changes settings for the top-level patch.

The *Signal Vector Size* is how many audio samples MSP calculates at a time. There are two vector sizes you can control. The *I/O Vector Size* (I/O stands for input/output) controls the number of samples that are transferred to and from the audio interface at one time. The *Signal Vector Size* sets the number of samples that are calculated by MSP objects at one time. This can be less than or equal to the I/O Vector Size, but not more. If the Signal Vector Size is less than the I/O Vector Size, MSP calculates two or more signal vectors in succession for each I/O vector that needs to be calculated.

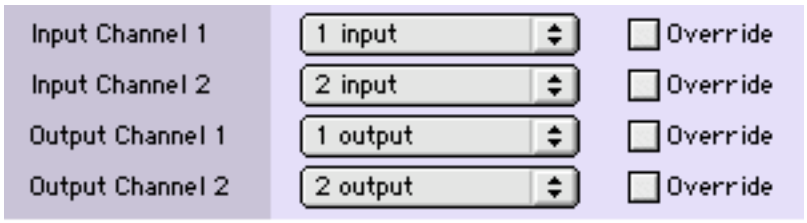
With an I/O vector size of 256, and a sampling rate of 44.1 kHz, MSP calculates about 5.8 milliseconds of audio data at a time.



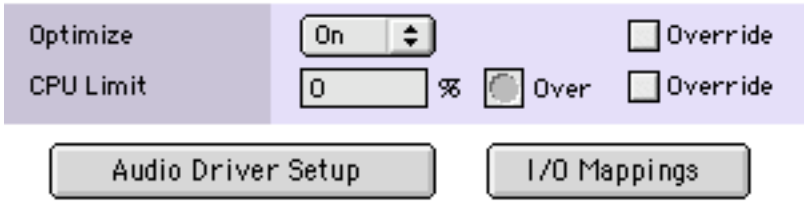
The *Max Scheduler in Overdrive* option enables you to turn Max's Overdrive setting on and off from within the DSP Status window. When Overdrive is enabled, the Max event scheduler runs at interrupt level. The event scheduler does things like trigger the bang from a repeating **metro** object, as well as send out any recently received MIDI data. When it is not enabled, overdrive runs the event scheduler inside a lower-priority event handling loop that can be interrupted by doing things like pulling down a menu. You can also enable and disable Overdrive using the Options menu. Overdrive generally improves timing accuracy, but there may be exceptions, and some third-party software may not work properly when Overdrive is enabled.

The *Scheduler in Audio Interrupt* feature is available when Overdrive is enabled. It runs the Max event scheduler immediately before processing a signal vector's worth of audio. Enabling Scheduler in Audio Interrupt can greatly improve the timing of audio events that are triggered from control processes or external MIDI input. However, the improvement in timing can be directly related to your choice of I/O Vector Size, since this determines the interval at which events outside the scheduler (such as MIDI input and output) affect Max. When the Signal Vector Size is 512, the scheduler will run every 512 samples. At 44.1 kHz, this is every 11.61 milliseconds, which is just at the outer limits of timing acceptability. With smaller Signal Vector Sizes (256, 128, 64), the timing will sound "tighter." Since you can change all of these parameters as the music is playing, you can experiment to find acceptable combination of precision and performance.

If you are not doing anything where precise synchronization between the control and audio is important, leave Scheduler in Audio Interrupt unchecked. You'll get a bit more overall CPU performance for signal processing



The pop-up menus labeled *Input Channel 1*, *Input Channel 2, Output Channel 1, and Output Channel 2* allow you to map the first two logical channels of I/O in MSP (i.e. the first two outlets of the **adc~** object and the first two inlets of the **dac~** object) to physical channels used by your audiodriver. Different audio drivers give you different options, for example, the MME driver on Windows only supports two channels, so you will normally use the default options. To map additional logical channels, use the I/O Mappings window, which you can view by clicking the I/O Mappings button at the bottom of the DSP Status window (see below for more information about the I/O Mappings window). In addition, you can use the **adstatus** object from within your patch to map any of the 512 logical audio I/O channels.



The *Optimize* pop-up menu is found only on the Macintosh version of MSP. It allows you to select whether G4 (Altivec) vector optimization will be used by MSP when computing audio. Vector optimization allows four samples to be processed within the space of a single instruction. However, not all audio signal processing algorithms can be optimized in this way (for example, recursive filter algorithms are substantially immune from vector optimization). Leaving this option on when using MSP on a G4 machine will enhance CPU utilization and performance, although the exact performance gain depends on the algorithm you are using and the number of MSP objects that implement it that have been vector-optimized. If you are using a pre-G4 Macintosh turning the option on will have no effect.

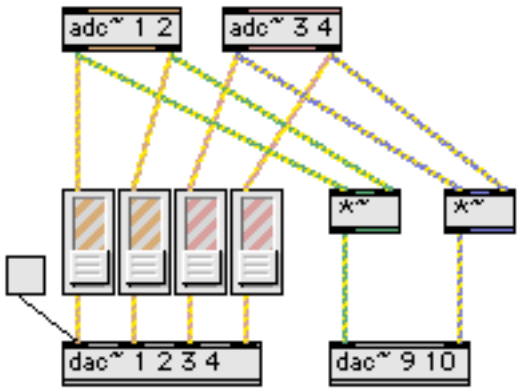
The *CPU Limit* option allows you to set a limit (expressed in terms of a percentage of your computer's CPU) to how much signal processing MSP is allowed to do. MSP will not go above the set CPU limit for a sustained period, allowing your computer to perform

other tasks without MSP locking them out. The trade-off, however, is that you'll hear clicks in the audio output when the CPU goes over the specified limit. Setting this value to either '0' or '100' will disable CPU limiting.

About Logical Input and Output Channels

In MSP 2, you can create a **dac~** or **adc~** object that uses a channel number between 1 and 512. These numbers refer to what we call logical channels and can be dynamically reassigned to physical device channels of a particular driver using either the DSP Status window, its I/O Mappings subwindow, or an adstatus object with an input or output keyword argument.

The **adc~** and **dac~** objects allow you to specify arguments which define which logical channels are mapped to their inlets and outlets, respectively. In the example below, multiple logical channels are in use in a simple patch:

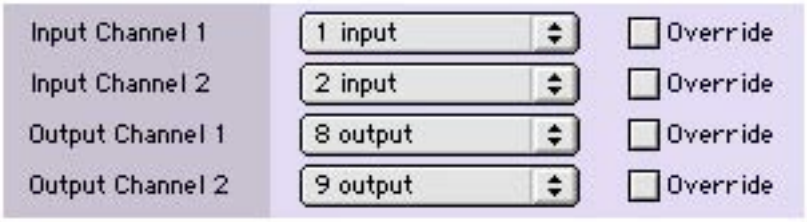


In this example, two separate **adc~** objects output audio signals from logical channel pairs 1/2 and 3/4, respectively. The output of all four channels is sent to **gain~** objects which attenuate the incoming signals and send them to the first four logical output channels, as specified by the first **dac~** object. The input signals are also multiplied (ring modulated) and sent out logical channels 9 and 10. Up to sixteen arguments can be typed into a single **adc~** or **dac~** object; if you want to use more than 16 logical channels, you can use multiple **adc~** and **dac~** objects. The **ezadc~** and **ezdac~** objects only access the first two logical input and output channels in MSP.

The purpose of having both logical channels and physical device channels is to allow you to create patches that use as many channels as you need without regard to the particular hardware configuration you're using. For instance, some audio interfaces use physical device channels 1 and 2 for S/ PDIF input and output. If you don't happen to have a S/PDIF-compatible audio interface, you may wish to use channels 8 and 9 instead. With

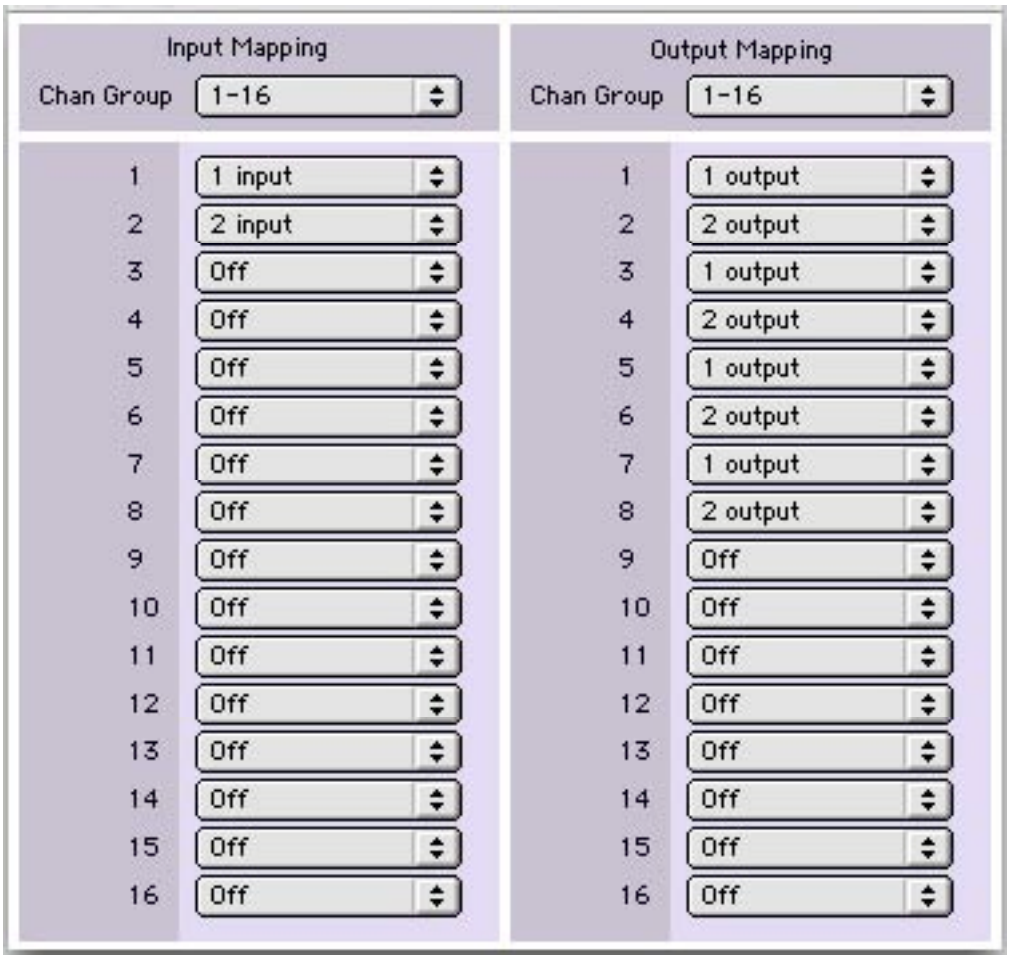
MSP 1.x, you would have been required to change all instances of **dac~** and/or **adc~** objects with arguments 1 and 2 to have arguments 8 and 9. With MSP 2, this is no longer necessary.

You can simply go to the DSP Status window and choose the eighth and ninth physical channels listed in the Input and Output pop-up menus.



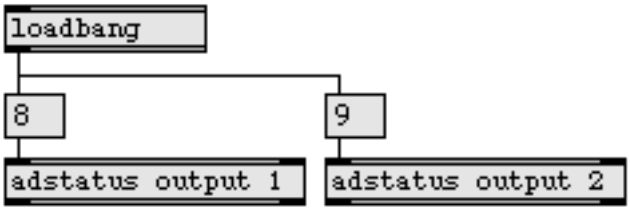
Logical channels in MSP are only created if there is a **dac~** or **adc~** object using them. In other words, if you're only using logical outputs 1 and 2, there aren't 510 unused audio streams sitting around hogging your CPU. However, since you can mix any number of logical channels to a single physical channel if necessary, you can create a complex multi-channel setup that will allow other people to hear all of your logical channels when they use it on a two-channel output device.

To assign multiple logical channels to one physical channel of an output device, use the I/O Mapping window. Click on the I/O Mappings button at the bottom of the DSP Status window.



The configuration shows that logical channels 1, 3, 5, and 7 have been mapped to the left output channel of the current audio device, and logical channels 2, 4, 6, and 8 have been mapped to the right output channel of the current audio device.

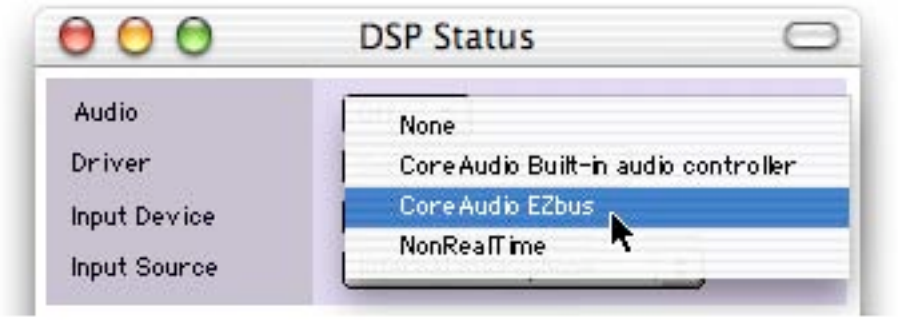
I/O Mappings are saved for each audio driver. You can also create I/O mappings within your patch using the **adstatus** object. The example patch below accomplishes the same remapping as that shown in the I/O Mapping window above, but does so automatically when the patch is loaded.



Using Core Audio on Macintosh

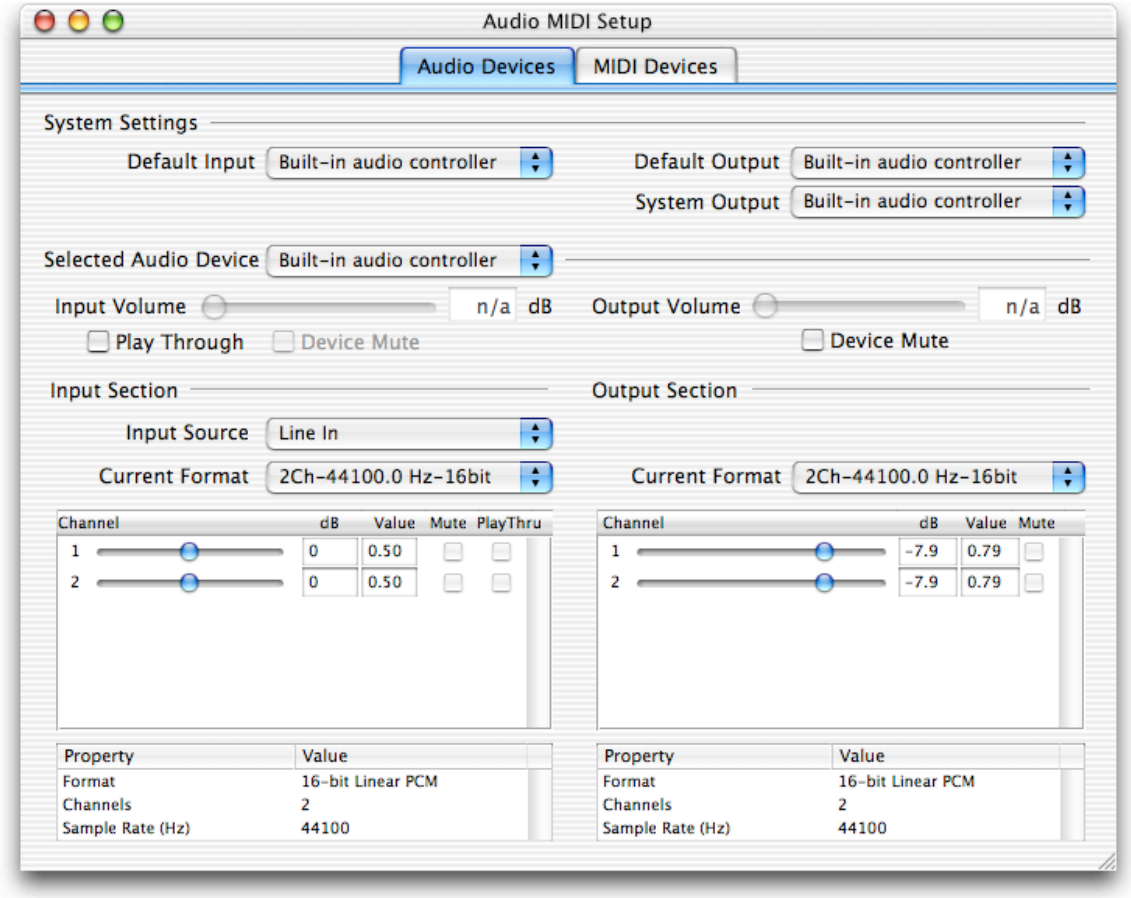
On Macintosh, MSP uses the Core Audio driver by default. As with all audio drivers, the Core Audio object file must be located in a folder called ad which is placed in /Library/Application Support/Cycling '74/. Core Audio is available on all Macintoshes running Mac OS X 10.2 or later, and provides Audio I/O to Macintosh applications from both the computer's built-in audio hardware as well as any external audio hardware you may have.

If you have external audio hardware, it should come the drivers to interface with Core Audio. When these drivers are installed and the hardware is present, Core Audio will include the external device as a Core Audio choice in the Driver menu in the DSP Status window.



The Sound part of the System Preferences application can be used to set basic sound settings for the system, such as the Output volume, left/right balance, and sound output device, as well as the Input volume and sound input device. You can also use the Audio MIDI Setup application (located in /Applications/Utilities) for more detailed control of the sound I/O settings. Note that modifications you make to the Sound section of the

System Preferences application, such as changing the output volume or balance, are reflected in the audio MIDI Setup (and vice versa). You can open the Audio MIDI Setup application by clicking on the Open Audio Control Panel button in the lower left corner of the DSP Status Window.



The Audio part of the Audio MIDI Setup application shows Input settings on the left side, and Output settings on the right.

The *System Settings* let you choose which audio device is used for system audio input and output, while the *Selected Audio Device* menu allows you to control the various settings for the built-in and any external hardware audio devices.

When using external audio devices, the *Input Volume* and *Output Volume* sliders can be used to set the overall input and output volumes of the selected device (they are not available when using the built-in audio controller). The *Device Mute* checkboxes allow you to mute the input and output devices, if applicable.

The *Play Through* checkbox just under the Input Volume slider lets you choose whether or not the input device is 'monitored' directly through to the output. When playthrough is enabled, the dry signal from the input source will play through to the output mixed in with any processed signal you may be sending to the output in MSP. Disabling playthrough will enable you to control how much (if any) dry signal from the audio input is routed to the output.

This option can be changed in MSP on Macintosh by sending a message to the **dsp** object to change it. Put the following in a message box and clicking on it will turn playthrough off:

```
; dsp driver playthrough 0
```

Using an argument of 1 will turn it on.

The *Input Section* allows you to select the Input Source (for example Line or Mic input for the selected device) as well as the sampling rate and bit depth in the *Current Format* pop-up menu. Similarly, the Output Section also allows you to select the sampling rate and bit-depth in its *Current Format* pop-up menu. The available selections will vary, depending on your audio hardware.

You can set the volume levels for the individual audio input and output channels, mute individual channels, and/or select them for playthrough using the controls located below the Current Format menus. The lower part of the window is used to display the current input and output settings.

Using MME Audio and DirectSound on Windows

Three types of sound card drivers are supported in Windows —MME, DirectSound and ASIO. Your choice of driver will have a significant impact on the performance and latency you will experience with MSP.

The MME driver (ad_mme) is the default used for output of Windows system sounds, and are provided for almost any sound card and built-in audio system. While compatibility with your hardware is almost guaranteed, the poor latency values you get from an MME driver make this the least desirable option for real-time media operation.

DirectSound drivers, built on Microsoft's DirectX technology, have become commonplace for most sound cards, and provide much better latency and performance than MME drivers. Whenever possible, a DirectSound driver (ad_directsound) should be used in preference to an MME driver. Occasionally, (and especially in the case of motherboard-based audio systems) you will find the DirectSound driver performs more poorly than the MME driver. This can happen when a hardware-specific DirectSound

driver is not available, and the system is emulating DirectSound while using the MME driver. In these cases, it is best to use MME directly, or find an ASIO driver for your system.

The best performance and lowest latency will typically be achieved using ASIO drivers. The ASIO standard, developed by Steinberg and supported by many media-oriented sound cards, is optimized for very low latency and high performance. As with the DirectSound driver, you need to verify that performance is actually better than other options; occasionally, an ASIO driver will be a simple "wrapper" around the MME or DirectSound driver, and will perform more poorly than expected.

Using MME and DirectSound Drivers on with MSP on Windows

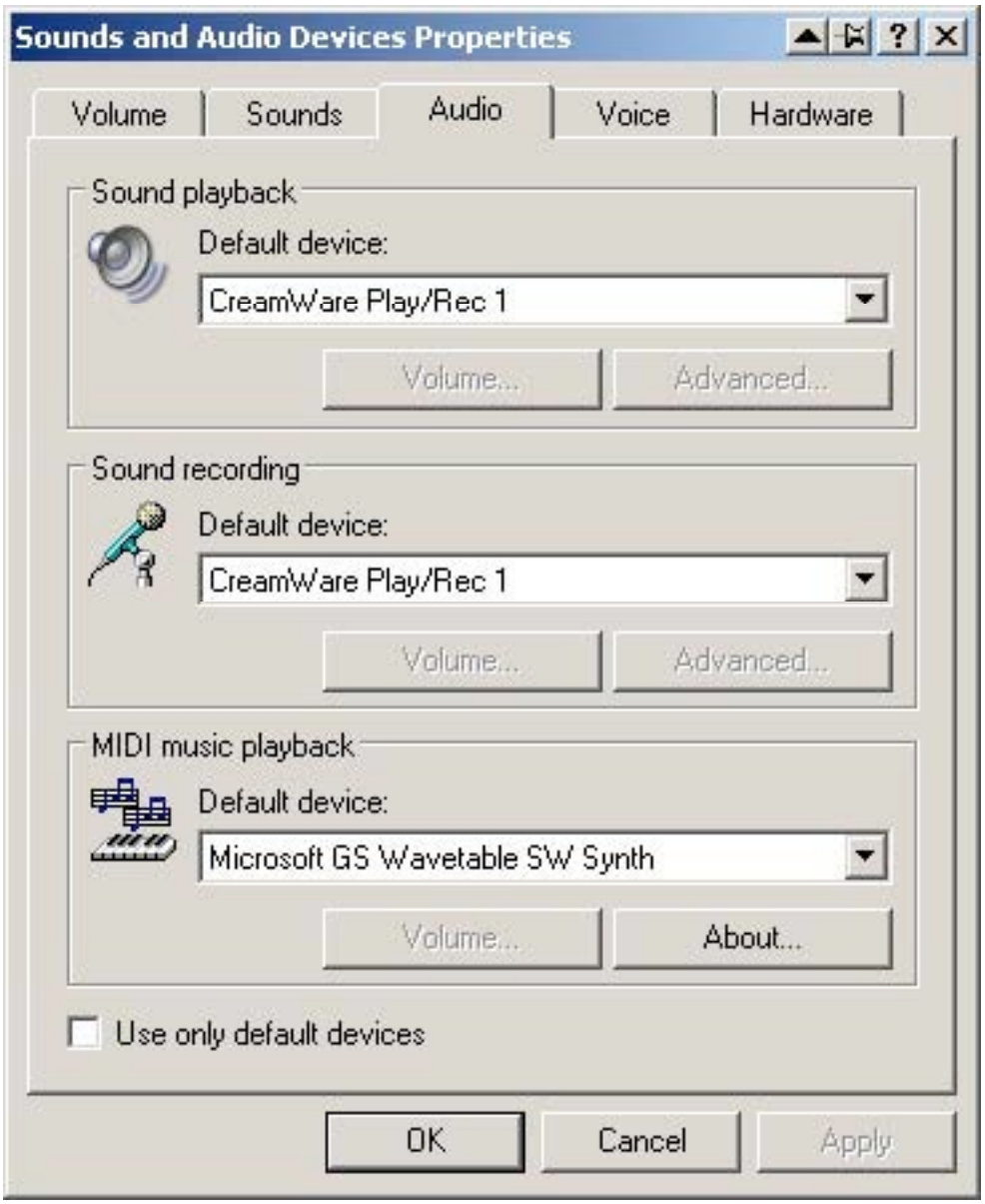
On Windows, MSP loads the MME driver by default. The MSP MME and DirectSound drivers are located in C:\Program Files\Common Files\Cycling '74\ad\.

If you have correctly installed external hardware, it should support playback and recording with the MME driver and the Direct Sound driver in the Driver Menu of the DSP Status Window.

If an audio device only supports MME or DirectSound, the Windows OS does an automatic mapping of one to the other. Since many audio devices initially did not support DirectSound, Microsoft emulated DirectSound with a layer that bridged from DirectSound to MME. Currently, there is greater support for native DirectSound drivers, and sometimes when you use MME drivers Windows is actually running a layer to convert from MME to DirectSound.

Note: Some devices such as the Digidesign mBox only support the ASIO driver standard. In such cases, you will need to select the proper ASIO driver in the DSP Status Window. See the section "Using ASIO Drivers on Windows" for more information.

You can make overall changes to the basic operation of your default audio driver by accessing the Sounds and Audio Devices Properties window (Start – Settings – Control Panel – Sounds and Audio Devices). Here you can select Audio devices, and create settings for balance and output volume.



MSP supports the use of different input and output devices with MME and DirectSound drivers. Use the DSP Status Window to choose input and output devices.

Input and Output Devices

When using MME or Directsound drivers, you may choose input and output devices from the pull-down menus in the DSP Status window, which will be automatically populated with the drivers for your audio hardware. When using the MME and Directsound drivers, it is possible to use different audio devices for input and output simultaneously. However, this is not recommended or supported and unless there is some external (from Max/MSP) provision for synchronizing the devices dropouts will likely occur over time.

Thread Priority and Latency Settings

Both MME and Directsound drivers include settings for Thread Priority and Latency. These are both set by default to settings which we hope will work on your computer in the majority of situations. However, you may find that when you are working with a patch that you have problems which you may be able to resolve by changing some of these settings. If your audio is crackling or there are glitches in it, you may want to try increasing the latency setting. This has the disadvantage of making your audio feel less responsive in real time, but it will allow the audio driver more time to work on the extra audio demands you have placed on it.

If your system is slow in other areas—such as screen redrawing or general timing accuracy—you may wish to decrease the thread priority of the audio driver. This will give your other tasks more room to get done, but may also result in you needing to increase latency in order to give your audio driver room to breathe at the new lower priority.

Timing between the max scheduler and MSP is best when the I/O vector size is on the order of 1ms. We recommend setting the IO vector size to 128 samples. Having a setting of the latency separate from the I/O vector size allows this to work without audio glitches on most hardware.

Using ReWire with MSP

The ad_rewire driver allows you to use MSP as a ReWire Device, where MSP audio will be routed into a ReWire Mixer application such as Cubase. Both Max/MSP and the mixer application must be running at the same time in order to take advantage of ReWire's services. The mixer application should be also compatible with ReWire 2 or later for best results.

When the ad_rewire driver is selected, audio from MSP can be routed to any of 16 inter-application ReWire channels which will appear as inputs in ReWire mixer host applications. The first time ad_rewire is selected it will register itself with the ReWire system. Subsequent launches of ReWire Mixer applications will then offer Max/MSP as a ReWire device.

For example, after the Max/MSP ReWire Device is registered, Cubase SX 1.0 will have a Max/ MSP menu item in the Devices menu. When you choose it you will see a list of the audio outputs from Max/MSP. They will default to the off state. Click on any of the buttons to activate that channel. Once the channel is activated it will show up in the Cubase Track Mixer.

MSP can also be used as a Mixer Application for ReWire Devices such as Reason. To do this, you use the **rewire~** object. Please see the MSP Reference Manual pages on the **rewire**~ object for more information.

If you try to use **rewire~**and the ad_rewire audio driver simultaneously, you won't get any audio output. This is because each is waiting for the other: the ad_rewire driver is waiting for the rewire~ object to ask it for an audio stream, but the **rewire~** object can't do anything unless given processing time by an audio driver.

However, you can use **rewire~** in conjunction with the Max Runtime or a standalone built using Max/MSP when the runtime or standalone is using the ad_rewire driver.

Inter-application Synchronization and MIDI in ReWire

ReWire supports sending synchronization, transport, and tempo information both to and from ReWire Devices. The **hostsync~**, **hostphasor~**, and **hostcontrol~** MSP objects can work with the ad_rewire driver to provide this information and to control the host's transport. See the MSP Reference Manual pages of these objects for more information.

Rewire 2 also supports MIDI communication to and from ReWire Devices. Currently both the **rewire~** object and the ad_rewire driver support MIDI, although they work in different ways. To send and receive midi using the **rewire~** object, you send message to and receive messages directly from the object. See the MSP Reference Manual pages for the **rewire~** object for more information.

The ad_rewire MIDI support is more integrated into the Max MIDI system—Max MIDI ports are created so you can use the standard Max MIDI objects to send and receive MIDI via the ad_rewire driver. After you choose the ad_rewire driver in the DSP Status Window, MIDI ports will appear in the MIDI Setup window the next time it is opened. The number of midi ports dedicated to ReWire can be changed using the MIDI Ports option in the DSP Status Window.

For example, you can choose one of the Max ReWire MIDI ports as a MIDI output device in Cubase and then use standard Max MIDI objects (such as **notein**) to control your Max/MSP created synthesizer. Likewise, you can send MIDI into Cubase using the max MIDI objects and the ReWire MIDI ports, and recorded the results to a track for further manipulation or playback.

Advanced ad_rewire Features

When you build a standalone application using Max/MSP you can use the ad_rewire driver in your standalone and it will create an ReWire Device that works independently of Max/MSP and other Max/MSP-created standalone applications. By default, the ReWire Device will take on the name of your application and will have 16 channels. You can customize this by editing the msprewire.config file in the *support/ad* folder for your standalone. Note: This file doesn't exist until the default one is created the first time you launch your standalone and choose ad_rewire in the DSP Status window.

The msprewire.config file is located in the ad folder found in the following locations:

Macintosh: *Library/Application Support/Cycling '74/ad/*

Windows: *c:\Program Files\Common Files\Cycling '74\ad*

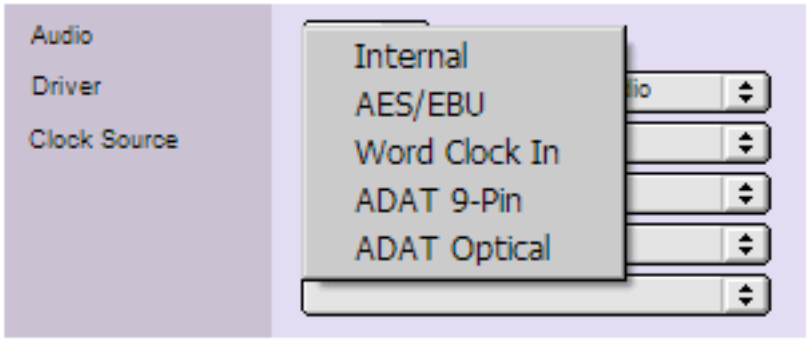
In a Max/MSP-built standalone this is in the standalone's support/ad/ folder. The msprewire.config contains two lines that specify the name that ReWire will use for the device and the number of audio channels. You can edit this to change the behavior of Max/MSP or your standalone.

Using ASIO on Windows

Selecting an ASIO driver from the DSP Status window allows MSP to talk directly to an audio interface. To use ASIO soundcards your device needs to be correctly installed and connected; The MSP ASIO driver will find it at startup.

All correctly installed ASIO devices should be available to you for selection in the DSP Status window. However, MSP does not check to see if the relevant audio interface hardware is installed correctly on your system until you explicitly switch to the ASIO driver for that interface card. If an ASIO driver fails to load when you try to use it, an error message will appear in the Max window (typically, an initialization error with a code of –1000) and the menus in the rest of the DSP Status window will blank out. Switching to the MME and/or DirectSound driver will re-enable MSP audio.

The *Clock Source* pop-up menu lets you to set the clock source for your audio hardware. Some ASIO drivers do not support an external clock; if this is the case there will only be one option in the menu, typically labeled Internal.



The *Prioritize MIDI* pop-up menu allows you to set the clock source for your audio hardware and whether or not to prioritize MIDI input and output over audio I/O.

Many ASIO drivers have other settings you can edit in a separate window. Click the Open ASIO Control Panel button at the bottom of the DSP Status window to access these settings. If your interface card has a control panel in its ASIO driver, the documentation for the interface should cover its operation.

Controlling ASIO Drivers with Messages to the dsp Object on Windows

Version 2 of the ASIO specification allows for direct monitoring of inputs to an audio interface. In other words, you can patch audio inputs to the interface directly to audio outputs without having the signals go through your computer. You also have control over channel patching, volume, and pan settings.

To control direct monitoring, you send the monitor message to the **dsp** object. The monitor message takes the following arguments

int	Obligatory. A number specifying an input channel number (starting at 1)
int	Optional. A number specifying an outlet channel number, or 0 to turn the routing for the specified input channel off. This is also what happens if the second argument is not present.
int or float	Optional. A number specifying the gain of the input -> output connection, between 0 and 4. 1 represents unity gain (and is the default).

int or float	Optional. A number specifying the panning of the output channel. -1 is left, 0 is center, and 1 is right. 0 is the default.

Here are some examples of the monitor message:

; dsp driver monitor 1 1 Patches input 1 to output 1 at unity gain with center pan.

; dsp driver monitor 1 0 Turns off input 1

; dsp driver monitor 1 4 2. -1. patches input 1 to output 4 with 6dB gain panned to the left

Note: When using these messages, the word "driver" is optional but recommended. Not all ASIO drivers support this feature. If you send the monitor message and get an ASIO result error -998 message in the Max window, then the driver does not support it.

Another feature of ASIO 2 is the ability to start, stop, and read timecode messages. To start timecode reading, send the following message:

```
; dsp driver timecode 1
```

To stop timecode reading, send the following message:

```
; dsp driver timecode 0
```

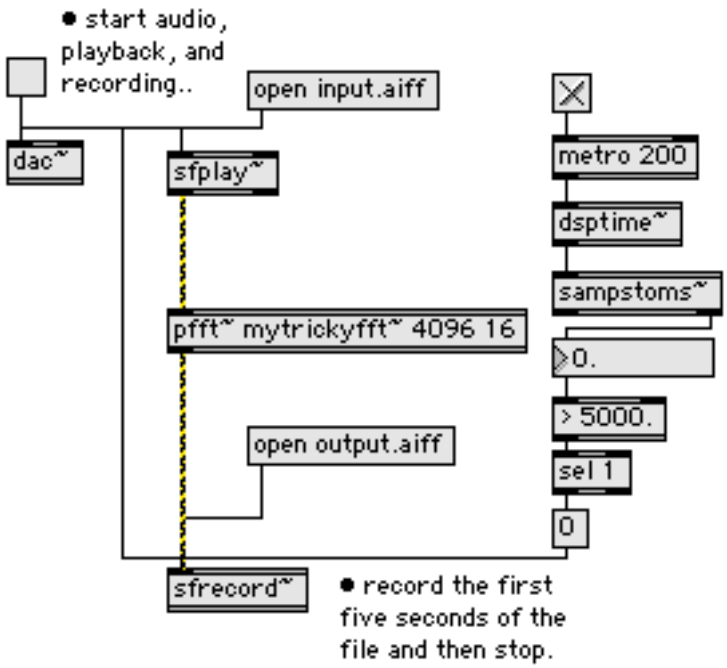
The **plugsync~** object reports the sample position reported by the audio interface when you enable timecode, but there isn't currently an object that reports the timecode of the interface.

Working in Non-Real Time with MSP

The MSP NonRealTime driver allows you to use MSP for synthesis and signal processing without worrying about the constraints imposed by the speed of your computer's CPU. Non-real-time mode simply calculates samples in MSP independently of any physical scheduling priority, allowing you to process a vector of audio using a signal path that might take your computer more than one vector's worth of real time to compute.

Typically, you will want to use the **dsptime~** object to see how long the audio has been turned on, and you will pipe the output of your routine to **sfrecord~** to capture the results. Hardware audio input and output under the non-real-time driver are disabled.

A typical non-real-time signal path in MSP would look something like this:



Starting the DSP (by toggling the **dac~** object) will start the **dsptime~** object at 0 samples, in sync with the playback of the audio out of **sfplay~** and the recording of audio into the **sfrecord~** at the bottom of the patch. When five seconds have elapsed, the **sfrecord~** object will stop recording the output audio file.

See Also

adc~	Audio input and on/off
adstatus	Access audio driver output channels
dac~	Audio output and on/off
ezadc~	Audio on/off; analog-to-digital converter
ezdac~	Audio output and on/off button

To open the example program for each chapter of the Tutorial, choose **Open...** from the File menu in Max and find the document in the MSP Tutorial folder with the same number as the chapter you are reading. It's best to have the current Tutorial example document be the only open Patcher window.

- Open the file called *Tutorial 01. Test tone*.

MSP objects are pretty much like Max objects

MSP objects are for processing digital audio (i.e., sound) to be played by your computer. MSP objects look just like Max objects, have inlets and outlets just like Max objects, and are connected together with patch cords just like Max objects. They are created the same way as Max objects— just by placing an object box in the Patcher window and typing in the desired name—and they co-exist quite happily with Max objects in the same Patcher window.

...but they're a little different

A patch of interconnected MSP objects works a little differently from the way a patch of standard Max objects works.

One way to think of the difference is just to think of MSP objects as working much faster than ordinary Max objects. Since MSP objects need to produce enough numbers to generate a high fidelity audio signal (commonly 44,100 numbers per second), they must work faster than the millisecond schedule used by standard Max objects.

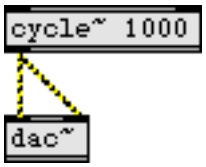
Here's another helpful way to think of the difference. Think of a patch of MSP objects not as a program in which events occur at specific instants (as in a standard Max patch), but rather as a description of an instrument design—a synthesizer, sampler, or effect processor. It's like a mathematical formula, with each object constantly providing numerical values to the object(s) connected to its outlet. At any given instant in time, this formula has a result, which is the instantaneous amplitude of the audio signal. This is why we frequently refer to an ensemble of inter-connected MSP objects as a *signal network*.

So, whereas a patch made up of standard Max objects sits idle and does nothing until something occurs (a mouse click, an incoming MIDI message, etc.) causing one object to send a message to another object, a signal network of MSP objects, by contrast, is always active (from the time it's turned on to the time it's turned off), with all its objects constantly communicating to calculate the appropriate amplitude for the sound at that instant.

...so they look a little different

The names of all MSP objects end with the tilde character (**~**). This character, which looks like a cycle of a sine wave, just serves as an indicator to help you distinguish MSP objects from other Max objects.

The patch cords between MSP objects have stripes. This helps you distinguish the MSP signal network from the rest of the Max patch.



MSP objects are connected by striped patch cords

Digital-to-analog converter: dac~

The digital-to-analog converter (DAC) is the part of your computer that translates the stream of discrete numbers in a digital audio signal into a continuous fluctuating voltage which will drive your loudspeaker.

Once you have calculated a digital signal to make a computer-generated sound, you must send the numbers to the DAC. So, MSP has an object called **dac~**, which generally is the terminal object in any signal network. It receives, as its input, the signal you wish to hear. It has as many inlets as there are available channels of audio playback. If you are using Core Audio (or WWWW on Windows) to play sounds directly from your computer's audio hardware, there are two output channels, so there will be two inlets to **dac~**. (If you are using more elaborate audio output hardware, you can type in an argument to specify other audio channels.)

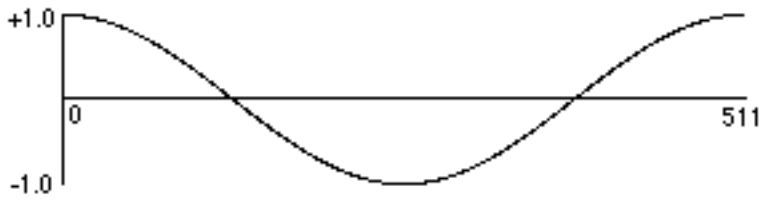
dac~ plays the signal

Important! dac~ must be receiving a signal of non-zero amplitude in order for you to hear anything. **dac~** expects to receive signal values in the range -1.0 to 1.0. Numbers that exceed that range will cause distortion when the sound is played.

Wavetable synthesis: cycle~

The best way to produce a periodic waveform is with **cycle~**. This object uses the technique known as "wavetable synthesis". It reads through a list of 512 values at a specified rate, looping back to the beginning of the list when it reaches the end. This simulates a periodically repeating waveform.

You can direct **cycle~** to read from a list of values that you supply (in the form of an audio file), or if you don't supply one, it will read through its own table which represents a cycle of a cosine wave with an amplitude of 1. We'll show you how to supply your own waveform in *Tutorial 3*. For now we'll use the cosine waveform.



Graph of 512 numbers describing one cycle of a cosine wave with amplitude 1

cycle~ receives a frequency value (in Hz) in its left inlet, and it determines on its own how fast it should read through the list in order to send out a signal with the desired frequency.

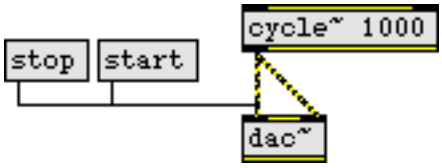
> **Technical detail:** To figure out how far to step through the list for each consecutive sample, **cycle~** uses the basic formula
>
> $$I = fL/R$$
>
> where I is the amount to increment through the list, f is the signal's frequency, L is the length of the list (512 in this case), and R is the audio sampling rate. **cycle~** is an "interpolating oscillator", which means that if I does not land exactly on an integer index in the list for a given sample, **cycle~** interpolates between the two closest numbers in the list to find the proper output value. Performing interpolation in a wavetable oscillator makes a substantial improvement in audio quality. The **cycle~** object uses linear interpolation, while other MSP objects use very high-quality (and more computationally expensive) polynomial interpolation.

By default **cycle~** has a frequency of 0 Hz. So in order to hear the signal, we need to supply an audible frequency value. This can be done with a number argument as in the example

patch, or by sending a number in the left inlet, or by connecting another MSP object to the left inlet.

Starting and stopping signal processing

The way to turn audio on and off is by sending the Max messages start and stop (or 1 and 0) to the left inlet of a **dac~** object (or an **adc~** object, discussed in a later chapter). Sending start or stop to any **dac~** or **adc~** object enables or disables processing for all signal networks.



The simplest possible signal network

Although **dac~** is part of a signal network, it also understands certain Max messages, such as start and stop. Many of the MSP objects function in this manner, accepting certain Max messages as well as audio signals.

Listening to the Test Tone

The first time you start up Max/MSP, it will try to use your computer's default sound card and driver (Core Audio on Macintosh or MME on Windows) for audio input and output. If you have the audio output of your computer connected to headphones or an amplifier, you should hear the output of MSP through it. If you don't have an audio cable connected to your computer, you'll hear the sound through the computer's internal speaker.

In order to get MSP up and running properly, we recommend that you start the tutorials using your computer's built-in sound hardware. If you want to use an external audio interface or sound care, please refer to the Audio Input and Output chapter for details on configuring MSP to work with audio hardware.

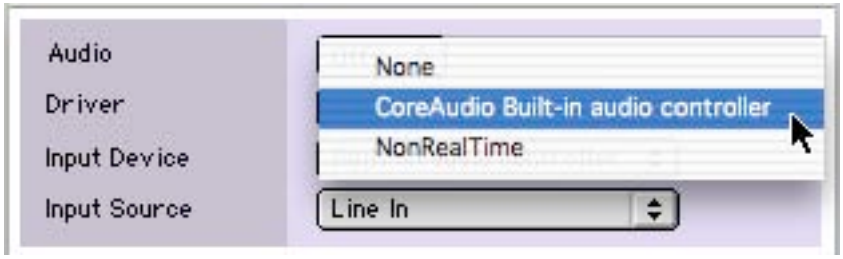
- Set your audio amplifier (or amplified speakers) to their minimum setting, then click on the start message box. Adjust your audio amplifier to the desired maximum setting, then click on the stop message box to turn off that annoying test tone.

Troubleshooting

If you don't hear any sound coming from your computer when you start the **dac~** in this example, check the level setting on your amplifier or mixer, and check all your audio connections. Check that the sound output isn't currently muted. On Macintosh, the

sound output level is set using the Sound preferences in the System Preferences application. On Windows, the sound output level is set using the Sounds and Audio Devices setup (Start - Control Panels - Sounds and Audio Devices).

If you are still are not hearing anything, choose DSP Status from the Options Menu and verify that Core Audio Built in Controller for Macintosh or MME driver for Windows is selected in the Driver pop-up menu. If it isn't, choose it.



Summary

A *signal network* of connected MSP objects describes an audio instrument. The digital-to-analog converter of the instrument is represented by the **dac~** object; **dac~** must be receiving a signal of non-zero amplitude (in the range -1.0 to 1.0) in order for you to hear anything. The **cycle~** object is a wavetable oscillator which reads cyclically through a list of 512 amplitude values, at a rate determined by the supplied frequency value. Signal processing is turned on and off by sending a start or stop message to any **dac~** or **adc~** object.

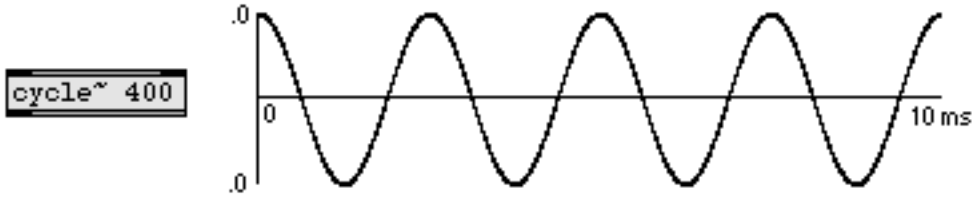
- Close the Patcher window before proceeding to the next chapter.

See Also

cycle~	Table lookup oscillator
dac~	Audio output and on/off
Audio I/O	Audio input and output with MSP

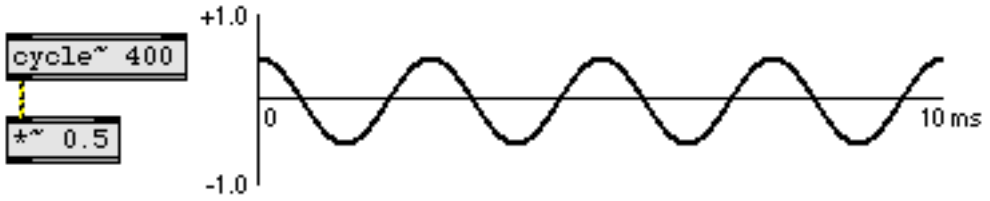
Amplifier: *~

A signal you want to listen to—a signal you send to **dac~—**must be in the amplitude range from -1.0 to +1.0. Any values exceeding those bounds will be clipped off by **dac~** (i.e. sharply limited to 1 or -1). This will cause (in most cases pretty objectionable) distortion of the sound. Some objects, such as **cycle~**, output values in that same range by default.



The default output of cycle~ has amplitude of 1

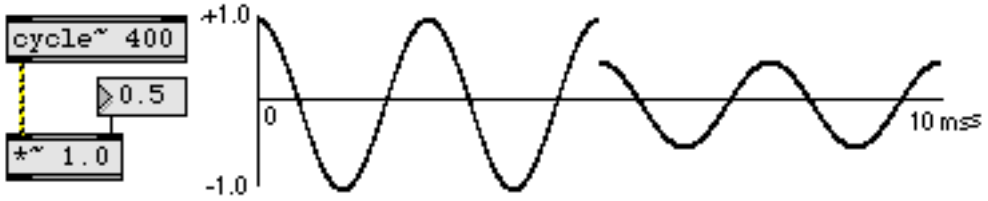
To control the level of a signal you simply multiply each sample by a scaling factor. For example, to halve the amplitude of a signal you simply multiply it by 0.5. (Although it would be mathematically equivalent to divide the amplitude of the signal by 2, multiplication is a more efficient computation procedure than division.



Amplitude adjusted by multiplication

If we wish to change the amplitude of a signal continuously over time, we can supply a changing signal in the right inlet of ***~**. By continuously changing the value in the right inlet of ***~**, we can fade the sound in or out, create a crescendo or diminuendo effect, etc.

However, a sudden drastic change in amplitude would cause a discontinuity in the signal, which would be heard as a noisy click.

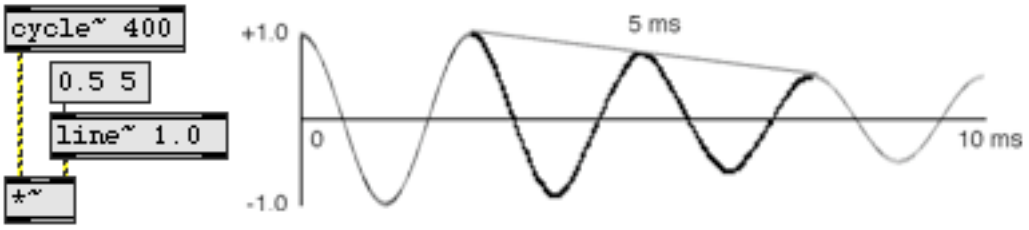


Instantaneous change of amplitude causes a noisy distortion of the signal

For that reason it's usually better to modify the amplitude using a signal that changes more gradually with each sample, say in a straight line over the course of several milliseconds.

Line segment generator: line~

If, instead of an instantaneous change of amplitude (which can cause an objectionable distortion of the signal), we supply a signal in the right inlet of ***~** that changes from 1.0 to 0.5 over the course of 5 milliseconds, we interpolate between the starting amplitude and the target amplitude with each sample, creating a smooth amplitude change.

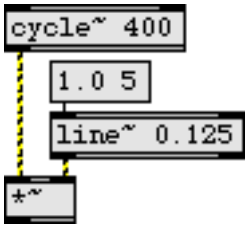


Linear amplitude change over 5 milliseconds using line~

The **line~** object functions similarly to the Max object **line**. In its left inlet it receives a target value and a time (in ms) to reach that target. **line~** calculates the proper intermediate value for each sample in order to change in a straight line from its current value to the target value.

Technical detail: Any change in the over-all amplitude of a signal introduces some amount of distortion during the time when the amplitude is changing. (The shape of the waveform is actually changed during that time, compared with the original signal.) Whether this distortion is objectionable depends on how sudden the change is, how great the change in amplitude is, and how complex the original signal is. A small amount of such distortion introduced into an already complex signal may go largely unnoticed by the listener. Conversely, even a slight distortion of a very pure original signal will add partials to the tone, thus changing its timbre.

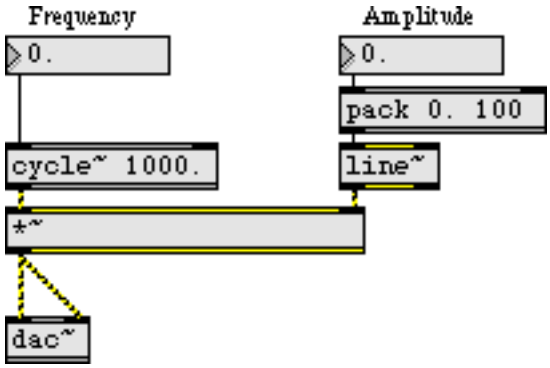
In the preceding example, the amplitude of a sinusoidal tone decreased by half (6 dB) in 5 milliseconds. Although one might detect a slight change of timbre as the amplitude drops, the shift is not drastic enough to be heard as a click. If, on the other hand, the amplitude of a sinusoid increases eightfold (18 dB) in 5 ms, the change is drastic enough to be heard as a percussive attack.



An eightfold (18 dB) increase in 5 ms creates a percussive effect

Adjustable oscillator

The example patch uses this combination of ***~** and **line~** to make an adjustable amplifier for scaling the amplitude of the oscillator. The **pack** object appends a transition time to the target amplitude value, so every change of amplitude will take 100 milliseconds. A **number box** for changing the frequency of the oscillator has also been included.



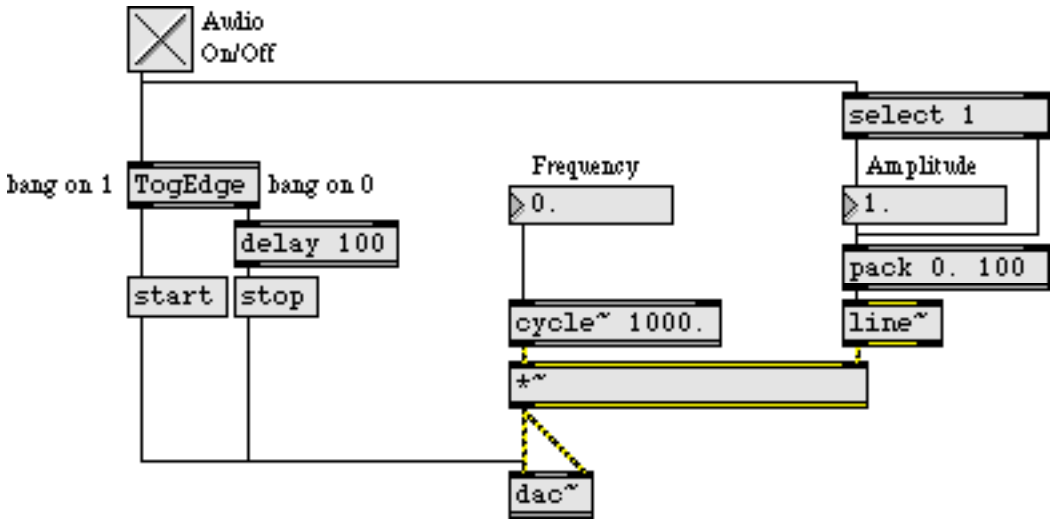
Oscillator with adjustable frequency and amplitude

Notice that the signal network already has default values before any Max message is sent to it. The **cycle~** object has a specified frequency of 1000 Hz, and the **line~** object has a default initial value of 0. Even if the ***~** had a typed-in argument for initializing its right inlet, its right operand would still be 0 because **line~** is constantly supplying it that value.

- Use the *Amplitude* **number box** to set the volume to the level you desire, then click on the **toggle** marked *Audio On/Off* to start the sound. Use the **number box** objects to change the frequency and amplitude of the tone. Click on the **toggle** again to turn the sound off.

Fade In and Fade Out

The combination of **line~** and ***~** also helps to avoid the clicks that can occur when the audio is turned on and off. The 1 and 0 "on" and "off" messages from the **toggle** are used to fade the volume up to the desired amplitude, or down to 0, just as the start or stop message is sent to **dac~**. In that way, the sound is faded in and out gently rather than being turned on instantaneously.



On and off messages fade audio in or out before starting or stopping the DAC

Just before turning audio off, the 0 from **toggle** is sent to the **pack** object to start a 100 ms fade-out of the oscillator's volume. A delay of 100 ms is also introduced before sending the stop message to **dac~**, in order to let the fade-out occur. Just before turning the audio on, the desired amplitude value is triggered, beginning a fade-in of the volume; the fade-in does not actually begin, however, until the **dac~** is started—immediately after, in this case. (In an actual program, the start and stop **message** boxes might be hidden from view or encapsulated in a subpatch in order to prevent the user from clicking on them directly.)

Summary

Multiplying each sample of an audio signal by some number other than 1 changes its amplitude; therefore the ***~** object is effectively an amplifier. A sudden drastic change of amplitude can cause a click, so a more gradual fade of amplitude—by controlling the amplitude with another signal—is usually advisable. The line segment signal generator **line~** is comparable to the Max object **line** and is appropriate for providing a linearly changing value to the signal network. The combination of **line~** and ***~** can be used to make an *envelope* for controlling the over-all amplitude of a signal.

See Also

cycle~	Table lookup oscillator
dac~	Audio output and on/off
line~	Linear ramp generator

Audio on/off switch: ezdac~

In this tutorial patch, the **dac~** object which was used in earlier examples has been replaced by a button with a speaker icon. This is the **ezdac~** object, a user interface object available in the object palette.

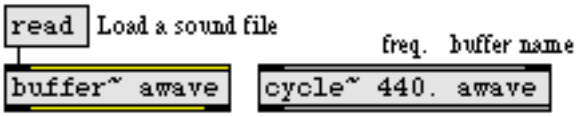
ezdac~ is an on/off button for audio, available in the object palette

The **ezdac~** works much like **dac~**, except that clicking on it turns the audio on or off. It can also respond to start and stop messages in its left inlet, like **dac~**. (Unlike **dac~**, however, it is appropriate only for output channels 1 and 2.) The **ezdac~** button is highlighted when audio is on.

A stored sound: buffer~

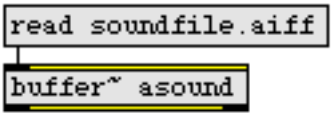
In the previous examples, the **cycle~** object was used to read repeatedly through 512 values describing a cycle of a cosine wave. In fact, though, **cycle~** can read through any 512 values, treating them as a single cycle of a waveform. These 512 numbers must be stored in an object called **buffer~**. (A *buffer* means a holding place for data.)

A **buffer~** object requires a unique name typed in as an argument. A **cycle~** object can then be made to read from that buffer by typing the same name in as its argument. (The initial frequency value for **cycle~**, just before the buffer name, is optional.)



cycle~ reads its waveform from a buffer~ of the same name

To get the sound into the **buffer~**, send it a read message. That opens an Open Document dialog box, allowing you to select an audio file to load. The word read can optionally be followed by a specific file name, to read a file in without selecting it from the dialog box, provided that the audio file is in Max's search path.



Read in a specific sound immediately

Regardless of the length of the sound in the **buffer~**, **cycle~** uses only 512 samples from it for its waveform. (You can specify a starting point in the **buffer~** for **cycle~** to begin its waveform, either with an additional argument to **cycle~** or with a set message to **cycle~**.) In the example patch, we use an audio file that contains exactly 512 samples.

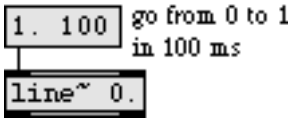
> **Technical detail:** In fact, **cycle~** uses 513 samples. The 513th sample is used only for interpolation from the 512th sample. When **cycle~** is being used to create a periodic waveform, as in this example patch, the 513th sample should be the same as the 1st sample. If the **buffer~** contains only 512 samples, as in this example, **cycle~** supplies a 513th sample that is the same as the 1st sample.

- Click on the **message** box that says read gtr512.aiff. This loads in the audio file. Then click on the **ezdac~** object to turn the audio on. (There will be no sound at first. Can you explain why?) Next, click on the **message** box marked B3 to listen to 1 second of the **cycle~** object.

There are several other objects that can use the data in a **buffer~**, as you will see in later chapters.

Create a breakpoint line segment function with line~

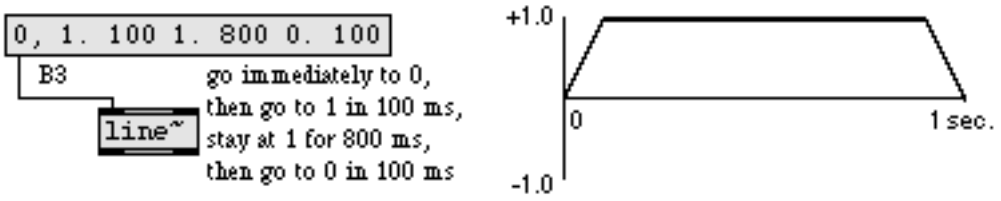
In the previous example patch, we used **line~** to make a linearly changing signal by sending it a list of two numbers. The first number in the list was a target value and the second was the amount of time, in milliseconds, for **line~** to arrive at the target value.



line~ is given a target value (1.) and an amount of time to get there (100 ms)

If we want to, we can send **line~** a longer list containing many value-time pairs of numbers (up to 64 pairs of numbers). In this way, we can make **line~** perform a more elaborate function composed of many adjoining line segments. After completing the first

line segment, **line~** proceeds immediately toward the next target value in the list, taking the specified amount of time to get there.

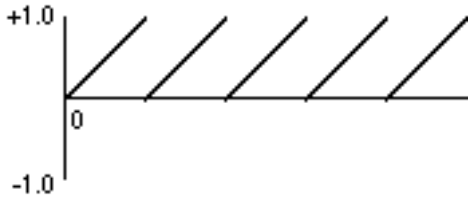


A function made up of line segments

Synthesizer users are familiar with using this type of function to generate an "ADSR" amplitude envelope. That is what we're doing in this example patch, although we can choose how many line segments we wish to use for the envelope.

Other signal generators: phasor~ and noise~

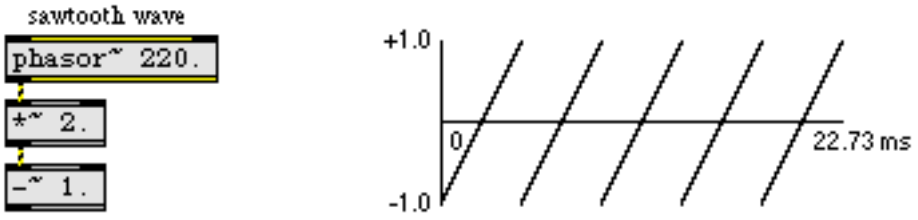
The **phasor~** object produces a signal that ramps repeatedly from 0 to 1.



Signal produced by phasor~

The frequency with which it repeats this ramp can be specified as an argument or can be provided in the left inlet, in Hertz, just as with **cycle~**. This type of function is useful at sub-audio frequencies to generate periodically recurring events (a crescendo, a filter sweep, etc.). At a sufficiently high frequency, of course, it is audible as a sawtooth waveform.

In the example patch, the **phasor~** is pitched an octave above **cycle~**, and its output is scaled and offset so that it ramps from -1 to +1.



220 Hz sawtooth wave

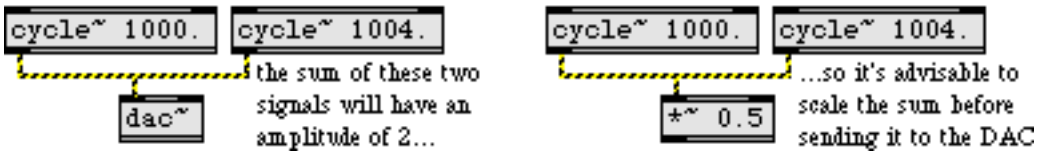
Technical detail: A sawtooth waveform produces a harmonic spectrum, with the amplitude of each harmonic inversely proportional to the harmonic number. Thus, if the waveform has amplitude A, the fundamental (first harmonic) has amplitude A, the second harmonic has amplitude A/ 2, the third harmonic has amplitude A/3, etc.

The **noise~** object produces white noise: a signal that consists of a completely random stream of samples. In this example patch, it is used to add a short burst of noise to the attack of a composite sound.

- Click on the **message** box marked B1 to hear white noise. Click on the **message** box marked B2 to hear a sawtooth wave.

Add signals to produce a composite sound

Any time two or more signals are connected to the same signal inlet, those signals are added together and their sum is used by the receiving object.

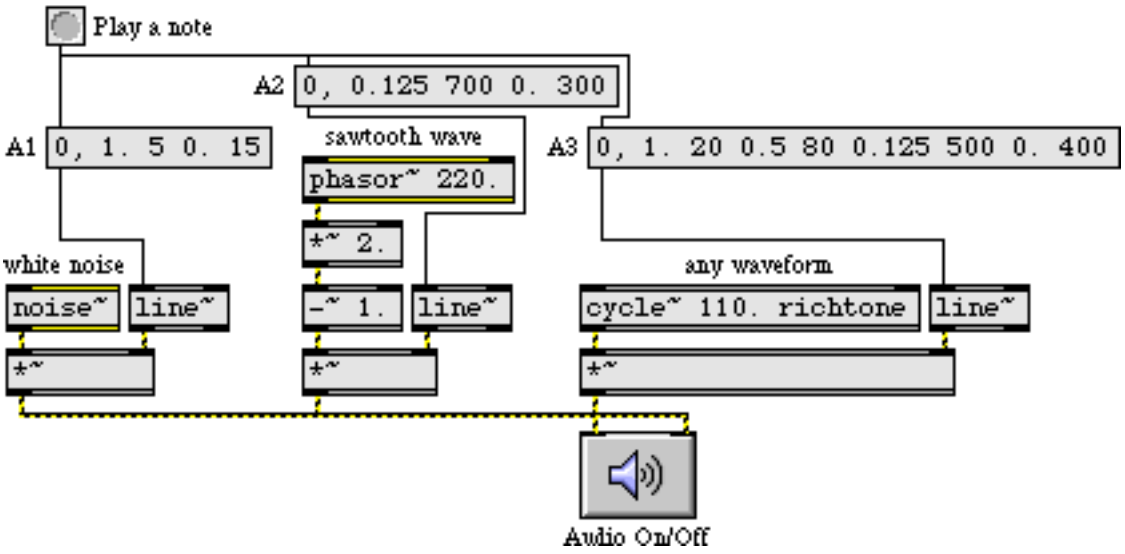


Multiple signals are added (mixed) in a signal inlet

Addition of digital signals is equivalent to unity gain mixing in analog audio. It is important to note that even if all your signals have amplitude less than or equal to 1, the sum of such signals can easily exceed 1. In MSP it's fine to have a signal with an

amplitude that exceeds 1, but before sending the signal to **dac~** you must scale it (usually with a ***~** object) to keep its amplitude less than or equal to 1. A signal with amplitude greater than 1 will be distorted by **dac~**.

In the example patch, white noise, a 220 Hz sawtooth wave, and a 110 Hz tone using the waveform in **buffer~** are all mixed together to produce a composite instrument sound.



Three signals mixed to make a composite instrument sound

Each of the three tones has a different amplitude envelope, causing the timbre of the note to evolve over the course of its 1-second duration. The three tones combine to form a note that begins with noise, quickly becomes electric-guitar-like, and gets a boost in its overtones from the sawtooth wave toward the end. Even though the three signals crossfade, their amplitudes are such that there is no possibility of clipping (except, possibly, in the very earliest milliseconds of the note, which are very noisy anyway).

- Click on the **button** to play all three signals simultaneously. To hear each of the individual parts that comprise the note, click on the **message** boxes marked *A1*, *A2*, and *A3*. If you want to hear how each of the three signals sound sustained at full volume, click on the **message** boxes marked *B1*, *B2*, and *B3*. When you have finished, click on **ezdac~** to turn the audio off.

Summary

The **ezdac~** object is a button for switching the audio on and off. The **buffer~** object stores a sound. You can load an audio file into **buffer~** with a read message, which opens an Open Document dialog box for choosing the file to load in. If a **cycle~** object has a typed-in

argument which gives it the same name as a **buffer~** object has, the **cycle~** will use 512 samples from that buffered sound as its waveform, instead of the default cosine wave.

The **phasor~** object generates a signal that increases linearly from 0 to 1. This ramp from 0 to 1 can be generated repeatedly at a specific frequency to produce a sawtooth wave. For generating white noise, the **noise~** object sends out a signal consisting of random samples.

Whenever you connect more than one signal to a given signal inlet, the receiving object adds those signals together and uses the sum as its input in that inlet. Exercise care when mixing (adding) audio signals, to avoid distortion caused by sending a signal with amplitude greater than 1 to the DAC; signals must be kept in the range -1 to +1 when sent to **dac~** or **ezdac~**.

The **line~** object can receive a list in its left inlet that consists of up to 64 pairs of numbers representing target values and transition times. It will produce a signal that changes linearly from one target value to another in the specified amounts of time. This can be used to make a function of line segments describing any shape desired, which is particularly useful as a control signal for amplitude envelopes. You can achieve crossfades between signals by using different amplitude envelopes from different **line~** objects.

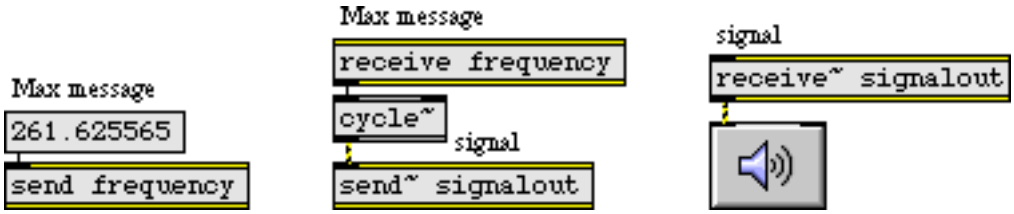
See Also

buffer~	Store audio samples
ezdac~	Audio output and on/off button
phasor~	Sawtooth wave generator
noise~	White noise generator

Remote signal connections: send~ and receive~

The patch cords that connect MSP objects look different from normal patch cords because they actually do something different. They describe the order of calculations in a signal network. The connected objects will be used to calculate a whole block of samples for the next portion of sound.

Max objects can communicate remotely, without patch cords, with the objects **send** and **receive** (and some similar objects such as **value** and **pv**). You can transmit MSP signals remotely with **send** and **receive**, too, but the patch cord(s) coming out of **receive** will not have the yellow-and-black striped appearance of the signal network (because a **receive** object doesn't know in advance what kind of message it will receive). Two MSP objects exist specifically for remote transmission of signals: **send~** and **receive~**.



send and receive for Max messages; send~ and receive~ for signals

The two objects **send~** and **receive~** work very similarly to **send** and **receive**, but are only for use with MSP objects. Max will allow you to connect normal patch cords to **send~** and **receive~**, but only signals will get passed through **send~** to the corresponding **receive~**. The MSP objects **send~** and **receive~** don't transmit any Max messages besides signals.

There are a few other important differences between the Max objects **send** and **receive** and the MSP objects **send~** and **receive~**.

1. The names of **send** and **receive** can be shortened to **s** and **r**; the names of **send~** and **receive~** cannot be shortened in the same way.

2. A Max message can be sent to a **receive** object from several other objects besides **send**, such as **float**, **forward**, **grab**, **if**, **int**, and **message**; **receive~** can receive a signal only from a **send~** object that shares the same name.

3. If **receive** has no typed-in argument, it has an inlet for receiving **set** messages to set or change its name; **receive~** also has an inlet for that purpose, but is nevertheless required to have a typed-in argument.

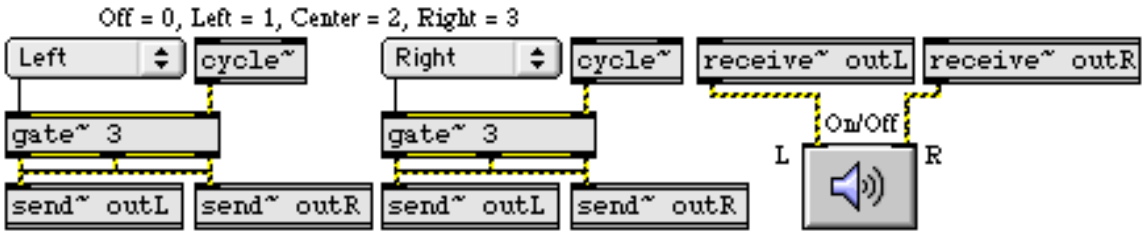
4. Changing the name of a **receive** object with a set message is a useful way of dynamically redirecting audio signals. Changing the name of **receive~**, however, does not redirect the signal until you turn audio off and back on again.

Examples of each of these usages can be seen in the tutorial patch.

Routing a signal: gate~

The MSP object **gate~** works very similarly to the Max object **gate**. Just as **gate** is used to direct messages to one of several destinations, or to shut the flow of messages off entirely, **gate~** directs a signal to different places, or shuts it off from the rest of the signal network.

In the example patch, the **gate~** objects are used to route signals to the left audio output, the right audio output, both, or neither, according to what number is received from the **umenu** object.



gate~ sends a signal to a chosen location

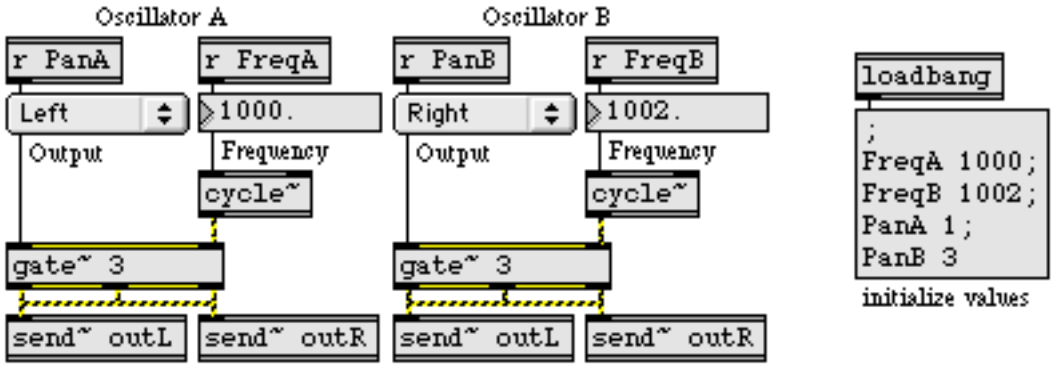
It is worth noting that changing the chosen outlet of a **gate~** while an audio signal is playing through it can cause an audible click because the signal shifts abruptly from one outlet to another. To avoid this, you should generally design your patch in such a way that the **gate~** object's outlet will only be changed when the audio signal going through it is at zero or when audio is off. (No such precaution was taken in the tutorial patch.)

Wave interference

It's a fundamental physical fact that when we add together two sinusoidal waves with different frequencies we create *interference* between the two waves. Since they have different frequencies, they will usually not be exactly in phase with each other; so, at some times they will be sufficiently in phase that they add together constructively, but at other times they add together destructively, canceling each other out to some extent. They only arrive precisely in phase with each other at a rate equal to the difference in their frequencies. For example, a sinusoid at 1000 Hz and another at 1002 Hz come into phase exactly 2 times per second. In this case, they are sufficiently close in frequency that we don't hear them as two separate tones. Instead, we hear their recurring pattern of constructive and destructive interference as *beats* occurring at a sub-audio rate of 2 Hz, a

rate known as the *difference frequency* or *beat frequency*. (Interestingly, we hear the two waves as a single tone with a sub-audio beat frequency of 2 Hz and an audio frequency of 1001 Hz.)

When the example patch is opened, a **loadbang** object sends initial frequency values to the **cycle~** objects—1000 Hz and 1002 Hz—so we expect that these two tones sounded together will cause a beat frequency of 2 Hz. It also sends initial values to the **gate~** objects (passing through the **umenu**s on the way) which will direct one tone to the left audio output and one to the right audio output.



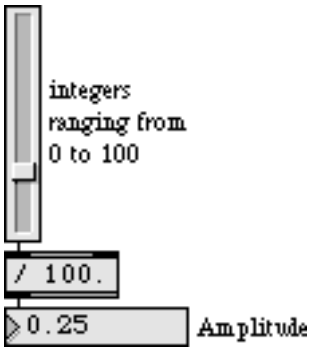
The two waves interfere at a rate of 2 Hz

- Click on **ezdac~** to turn audio on, then use the **uslider** marked "Volume" to adjust the loudness of the sound to a comfortable level. Note that the beats occur exactly twice per second. Try changing the frequency of Oscillator B to various other numbers close to 1000, and note the effect. As the difference frequency approaches an audio rate (say, in the range of 20-30 Hz) you can no longer distinguish individual beats, and the effect becomes more of a timbral change. Increase the difference still further, and you begin to hear two distinct frequencies.

  Philosophical tangent: It can be shown mathematically and empirically that when two sinusoidal tones are added, their interference pattern recurs at a rate equal to the difference in their frequencies. This apparently explains why we hear beats; the amplitude demonstrably varies at the difference rate. However, if you listen to this patch through headphones—so that the two tones never have an opportunity to interfere mathematically, electrically, or in the air—you still hear the beats! This phenomenon, known as *binaural beats* is caused by "interference" occurring in the nervous system. Although such interference is of a very different physical nature than the interference of sound waves in the air, we experience it as similar. An experiment like this demonstrates that our auditory system actively shapes the world we hear.

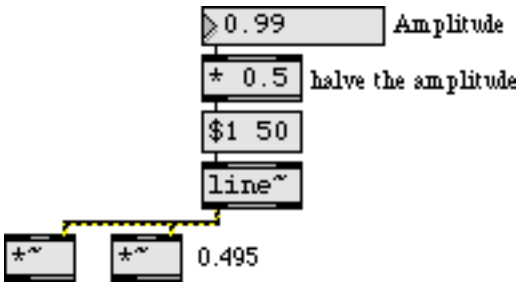
Amplitude and relative amplitude

The **uslider** marked "Volume" has been set to have a range of 101 values, from 0 to 100, which makes it easy to convert its output to a float ranging from 0 to 1 just by dividing by 100. (The decimal point in argument typed into the **/** object ensures a float division.)



A volume fader is made by converting the int output of uslider to a float from 0. to 1.

The ***~** objects use the specified amplitude value to scale the audio signal before it goes to the **ezdac~**. If both oscillators get sent to the same inlet of **ezdac~**, their combined amplitude will be 2. Therefore, it is prudent to keep the amplitude scaling factor at or below 0.5. For that reason, the amplitude value—which the user thinks of as being between 0 and 1—is actually kept between 0 and 0.5 by the * 0.5 object.



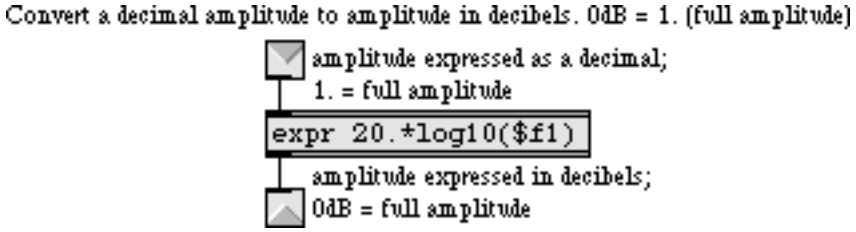
The amplitude is halved in case both oscillators are going to the same output channel

Because of the wide range of possible audible amplitudes, it may be more meaningful in some cases to display volume numerically in terms of the logarithmic scale of decibels (*dB*), rather than in terms of absolute amplitude. The decibel scale refers to *relative* amplitude—the amplitude of a signal relative to some reference amplitude. The formula for calculating amplitude in decibels is

$$dB = 20(\log10(A/Aref))$$

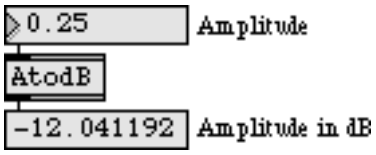
where *A* is the amplitude being measured and *Aref* is a fixed reference amplitude.

The subpatch **AtodB** uses a reference amplitude of 1 in the formula shown above, and converts the amplitude to dB.



The contents of the subpatch AtodB

Since the amplitude received from the **uslider** will always be less than or equal to 1, the output of **AtodB** will always be less than or equal to 0 dB. Each halving of the amplitude is approximately equal to a 6 dB reduction.



AtodB reports amplitude in dB, relative to a reference amplitude of 1

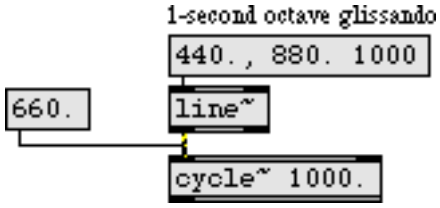
- Change the position of the **uslider** and compare the linear amplitude reading to the logarithmic decibel scale reading.

Constant signal value: sig~

Most signal networks require some changing values (such as an amplitude envelope to vary the amplitude over time) and some constant values (such as a frequency value to keep an oscillator at a steady pitch). In general, one provides a constant value to an MSP object in the form of a float message, as we have done in these examples when sending a frequency in the left inlet of a **cycle~** object.

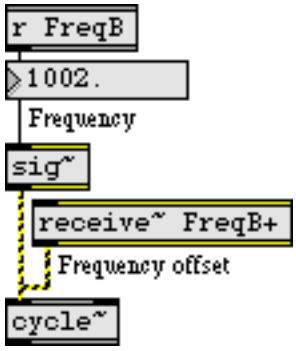
However, there are some cases when one wants to combine both constant and changing values in the same inlet of an MSP object. Most inlets that accept either a float or a signal (such as the left inlet of **cycle~**) do not successfully combine the two.

For example, **cycle~** ignores a float in its left inlet if it is receiving a signal in the same inlet.



cycle~ ignores its argument or a float input when a signal is connected to the left inlet

One way to combine a numerical Max message (an int or a float) with a signal is to convert the number into a steady signal with the **sig~** object. The output of **sig~** is a signal with a constant value, determined by the number received in its inlet.

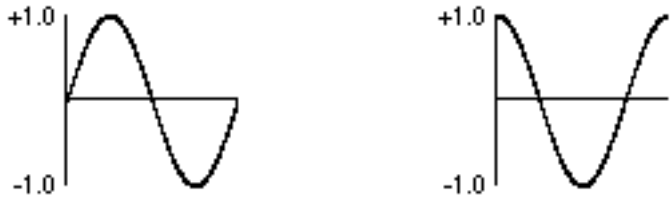


sig~ converts a float to a signal so it can be combined with another signal

In the example patch, Oscillator B combines a constant frequency (supplied as a float to **sig~**) with a varying frequency offset (an additional signal value). The sum of these two signals will be the frequency of the oscillator at any given instant.

Changing the phase of a waveform

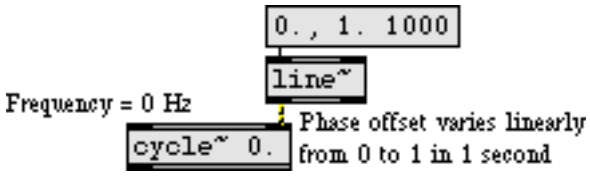
For the most part, the phase offset of an isolated audio wave doesn't have a substantial effect perceptually. For example, a sine wave in the audio range sounds exactly like a cosine wave, even though there is a theoretical phase difference of a quarter cycle. For that reason, we have not been concerned with the rightmost phase inlet of **cycle~** until now.



A sine wave offset by a quarter cycle is a cosine wave

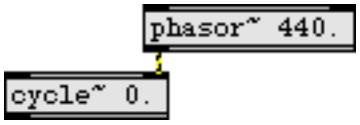
However, there are some very useful reasons to control the phase offset of a wave. For example, by leaving the frequency of **cycle~** at 0, and continuously increasing its phase offset, you can change its instantaneous value (just as if it had a positive frequency). The phase offset of a sinusoid is usually referred to in degrees (a full cycle is 360˚) or *radians* (a full cycle is 2π radians). In the **cycle~** object, phase is referred to in wave cycles; so an offset of π radians is $^1/_2$ cycle, or 0.5. In other words, as the phase varies from 0 to 2π radians, it varies from 0 to 1 wave cycles. This way of describing the phase is handy since it allows us to use the common signal range from 0 to 1.

So, if we vary the phase offset of a stationary (0 Hz) **cycle~** continuously from 0 to 1 over the course of one second, the resulting output is a cosine wave with a frequency of 1 Hz.



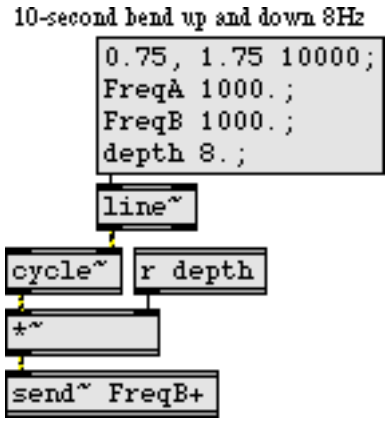
The resulting output is a cosine wave with a frequency of 1 Hz

Incidentally, this shows us how the **phasor~** object got its name. It is ideally suited for continuously changing the phase of a **cycle~** object, because it progresses repeatedly from 0 to 1. If a **phasor~** is connected to the phase inlet of a 0 Hz **cycle~**, the frequency of the **phasor~** will determine the rate at which the **cycle~** object's waveform is traversed, thus determining the effective frequency of the **cycle~**.



The effective frequency of the 0 Hz cycle~ is equal to the rate of the phasor~

The important point demonstrated by the tutorial patch, however, is that the phase inlet can be used to read through the 512 samples of **cycle~** object's waveform at any desired rate. (In fact, the contents of **cycle~** can be scanned at will with any value in the range 0 to 1.) In this case, **line~** is used to change the phase of **cycle~** from .75 to 1.75 over the course of 10 seconds.

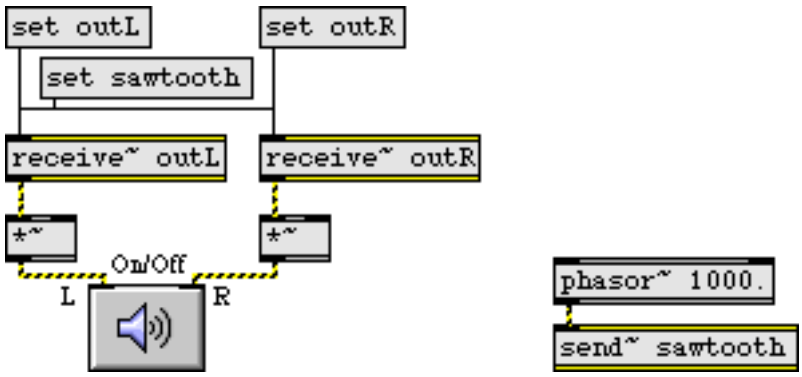


The result is one cycle of a sine wave. The sine wave is multiplied by a "depth" factor to scale its amplitude up to 8. This sub-audio sine wave, varying slowly from 0 up to 8, down to -8 and back to 0, is added to the frequency of Oscillator B. This causes the frequency of Oscillator B to fluctuate very slowly between 1008 Hz and 992 Hz.

- Click on the **message** box in the lower-left part of the window, and notice how the beat frequency varies sinusoidally over the course of 10 seconds, from 0 Hz up to 8 Hz (as the frequency of Oscillator B approaches 1008 Hz), back to 0 Hz, back up to 8 Hz (as the frequency of Oscillator B approaches 992 Hz), and back to 0 Hz.

Receiving a different signal

The remaining portion of the tutorial patch exists simply to demonstrate the use of the set message to the **receive~** object. This is another way to alter the signal flow in a network. With set, you can change the name of the **receive~** object, which causes **receive~** to get its input from a different **send~** object (or objects).



Giving receive~ a new name changes its input

- Click on the **message** box containing set sawtooth. Both of the connected **receive~** objects now get their signal from the **phasor~** in the lower-right corner of the window. Click on the **message** boxes containing set outL and set outR to receive the sinusoidal tones once again. Click on **ezdac~** to turn audio off.

Summary

It is possible to make signal connections without patch cords, using the MSP objects **send~** and **receive~**, which are similar to the Max objects **send** and **receive**. The set message can be used to change the name of a **receive~** object, thus switching it to receive its input from a different **send~** object (or objects). Signal flow can be routed to different destinations, or shut off entirely, using the **gate~** object, which is the MSP equivalent of the Max object **gate**.

The **cycle~** object can be used not only for periodic audio waves, but also for sub-audio control functions: you can read through the waveform of a **cycle~** object at any rate you wish, by keeping its frequency at 0 Hz and changing its phase continuously from 0 to 1. The **line~** object is appropriate for changing the phase of a **cycle~** waveform in this way, and **phasor~** is also appropriate because it goes repeatedly from 0 to 1.

The **sig~** object converts a number to a constant signal; it receives a number in its inlet and sends out a signal of that value. This is useful for combining constant values with

varying signals. Mixing together tones with slightly different frequencies creates interference between waves, which can create beats and other timbral effects.

See Also

gate~	Route a signal to one of several outlets
receive~	Receive signals without patch cords
send~	Transmit signals without patch cords
sig~	Constant signal of a number

Tutorial 5: Fundamentals—Turning signals on and off

Turning audio on and off selectively

So far we have seen two ways that audio processing can be turned on and off:

1. Send a start or stop message to a **dac~**, **adc~**, **ezdac~**, or **ezadc~** object.

2, Click on a **ezdac~** or **ezadc~** object.

There are a couple of other ways we have not yet mentioned:

3. Send an int to a **dac~**, **adc~**, **ezdac~**, or **ezadc~** object. 0 is the same as stop, and a non-zero number is the same as start.

4. Double-click on a **dac~** or **adc~** object to open the DSP Status window, then use its Audio on/ off pop-up menu. You can also choose **DSP Status…** from the Options menu to open the DSP Status window.

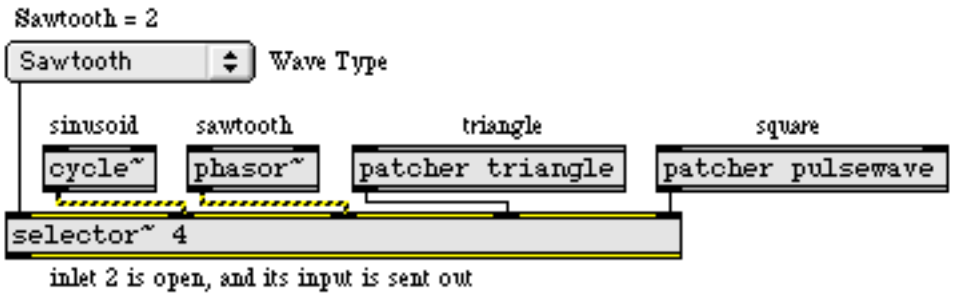
Any of those methods of starting MSP will turn the audio on in all open Patcher windows and their subpatches. There is also a way to turn audio processing on and off in a single Patcher.

Sending the message startwindow—instead of start—to a **dac~**, **adc~**, **ezdac~**, or **ezadc~** object turns the audio on *only* in the Patcher window that contains that object, and in its subpatches. It turns audio off in all other Patchers. The startwindow message is very useful because it allows you to have many different signal networks loaded in different Patchers, yet turn audio on only in the Patcher that you want to hear. If you encapsulate different signal networks in separate patches, you can have many of them loaded and available but only turn on one at a time, which helps avoid overtaxing your computer's processing power. (Note that startwindow is used in all MSP help files so that you can try the help file's demonstration without hearing your other work at the same time.)

In some cases startwindow is more appropriate than start

Selecting one of several signals: selector~

In the previous chapter, we saw the **gate~** object used to route a signal to one of several possible destinations. Another useful object for routing signals is **selector**~, which is comparable to the Max object **switch**. Several different signals can be sent into **selector~**, but it will pass only one of them—or none at all—out its outlet. The left inlet of **selector~** receives an int specifying which of the other inlets should be opened. Only the signal coming in the opened inlet gets passed on out the outlet.



The number in the left inlet determines which other inlet is open

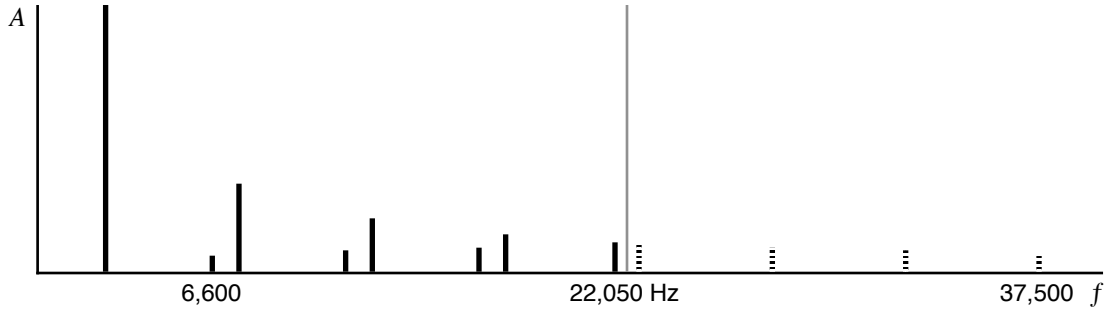
As with **gate~**, switching signals with **selector~** can cause a very abrupt change in the signal being sent out, resulting in unwanted clicks. So if you want to avoid such clicks it's best to change the open inlet of **selector~** only when audio is off or when all of its input signal levels are 0.

In the tutorial patch, **selector~** is used to choose one of four different classic synthesizer wave types: sine, sawtooth, triangle, or square. The **umenu** contains the names of the wave types, and sends the correct number to the control inlet of **selector~** to open the desired inlet.

- Choose a wave type from the pop-up menu, then click on the startwindow **message**. Use the pop-up menu to listen to the four different waves. Click on the stop **message** to turn audio off.

Technical detail: A sawtooth wave contains all harmonic partials, with the amplitude of each partial proportional to the inverse of the harmonic number. If the fundamental (first harmonic) has amplitude *A*, the second harmonic has amplitude *A/2*, the third harmonic has amplitude *A/3*, etc. A square wave contains only odd numbered harmonics of a sawtooth spectrum. A triangle wave contains only odd harmonics of the fundamental, with the amplitude of each partial proportional to the square of the inverse of the harmonic number. If the fundamental has amplitude A, the third harmonic has amplitude *A/9*, the fifth harmonic has amplitude *A/25*, etc.

Note that the waveforms in this patch are ideal shapes, not band-limited versions. That is to say, there is nothing limiting the high frequency content of the tones. For the richer tones such as the sawtooth and pulse waves, the upper partials can easily exceed the Nyquist rate and be folded back into the audible range. The partials that are folded over will not belong to the intended spectrum, and the result will be an inharmonic spectrum. As a case in point, if we play an ideal square wave at 2,500 Hz, only the first four partials can be accurately represented with a sampling rate of 44.1 kHz. The frequencies of the other partials exceed the Nyquist rate of 22,050 Hz, and they will be folded over back into the audible range at frequencies that are not harmonically related to the fundamental. For example, the eighth partial (the 15th harmonic) has a frequency of 37,500 Hz, and will be folded over and heard as 6,600 Hz, a frequency that is not a harmonic of 2,500. (And its amplitude is only about 24 dB less than that of the fundamental.) Other partials of the square wave will be similarly folded over.



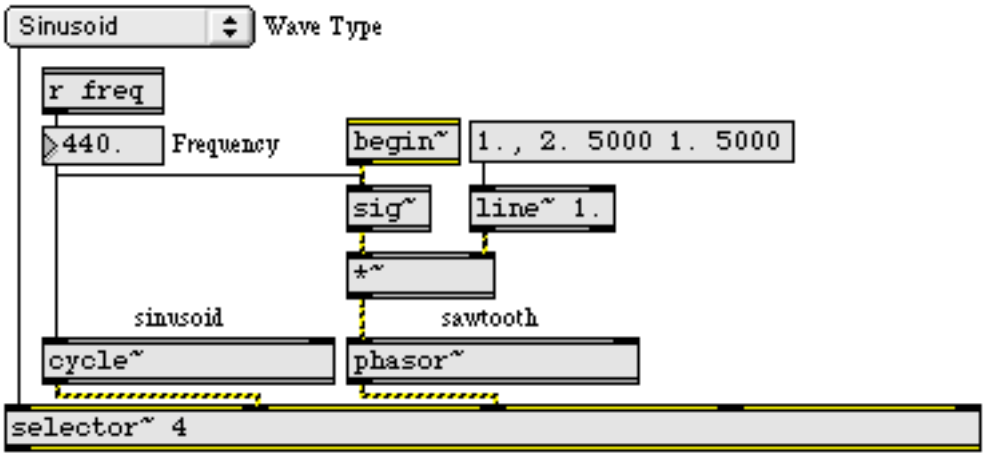
Partials that exceed the Nyquist frequency are folded over

- Choose the square wave from the pop-up menu, and set the frequency to 2500 Hz. Turn audio on. Notice that some of the partials you hear are not harmonically related to the fundamental. If you move the frequency up further, the folded-over partials will go down by the same amount. Turn audio off.

Turning off part of a signal network: begin~

You have seen that the **selector~** and **gate~** objects can be used to listen selectively to a particular part of the signal network. The parts of the signal network that are being ignored—for example, any parts of the network that are going into a closed inlet of **selector~**—continue to run even though they have been effectively disconnected. That means MSP continues to calculate all the numbers necessary for that part of the signal network, even though it has no effect on what you hear. This is rather wasteful, computationally, and it would be preferable if one could actually shut down the processing for the parts of the signal network that are not needed at a given time.

If the **begin~** object is placed at the beginning of a portion of a signal network, and that portion goes to the inlet of a **selector~** or **gate~** object, all calculations for that portion of the network will be completely shut down when the **selector~** or **gate~** is ignoring that signal. This is illustrated by comparing the sinusoid and sawtooth signals in the tutorial patch.



When the sinusoid is chosen, processing for the sawtooth is turned off entirely

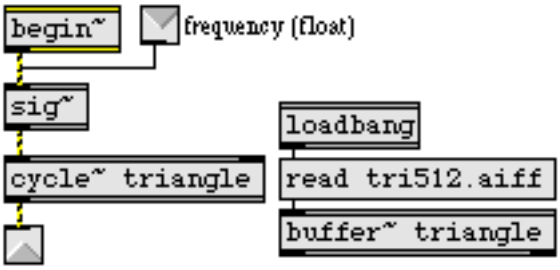
When the first signal inlet of **selector~** is chosen, as in the example shown above, the other signal inlets are ignored. Calculations cease for all the objects between **begin~** and **selector~**—in this case, the **sig~**, ***~**, and **phasor~** objects. The **line~** object, because it is not in the chain of objects that starts with **begin~**, continues to run even while those other objects are stopped.

- Choose "Sawtooth" from the pop-up menu, set the frequency back to 440 Hz, and turn audio on. Click on the **message** box above the **line~** object. The **line~** makes a glissando up an octave and back down over the course of ten seconds. Now click on it again, let the glissando get underway for two seconds, then use the pop-up menu to switch the **selector~** off. Wait five seconds, then switch back to the sawtooth. The

glissando is on its way back down, indicating that the **line~** object continued to work even though the **sig~**, ***~**, and **phasor~** objects were shut down. When the glissando has finished, turn audio off.

The combination of **begin~** and **selector~** (or **gate~**) can work perfectly well from one subpatch to another, as well.

- Double-click on the **patcher triangle** object to view its contents.



Contents of the patcher triangle object

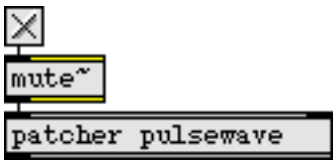
Here the **begin~** object is inside a subpatch, and the **selector~** is in the main patch, but the combination still works to stop audio processing in the objects that are between them. There is no MSP object for making a triangle wave, so **cycle~** reads a single cycle of a triangle wave from an AIFF file loaded into a **buffer~**.

begin~ is really just an indicator of a portion of the signal network that will be disabled when **selector~** turns it off. What actually comes out of **begin~** is a constant signal of 0, so **begin~** can be used at any point in the signal network where a 0 signal is appropriate. It can either be added with some other signal in a signal inlet (in which case it adds nothing to that signal), or it can be connected to an object that accepts but ignores signal input, such as **sig~** or **noise~**.

Disabling audio in a Patcher: mute~ and pcontrol

You have seen that the startwindow message to **dac~** turns audio on in a single Patcher and its subpatches, and turns audio off in all other patches. There are also a couple of ways to turn audio off in a specific subpatch, while leaving audio on elsewhere.

One way is to connect a **mute~** object to the inlet of the subpatch you want to control.



Stopping audio processing in a specific subpatch

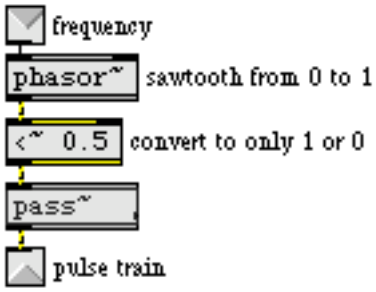
To mute a subpatch, connect a **mute~** object to the inlet of the subpatch, as shown. When **mute~** receives a non-zero int in its inlet, it stops audio processing for all MSP objects in the subpatch. Sending 0 to **mute~** object's inlet unmutes the subpatch.

- Choose "Square" from the pop-up menu, and turn audio on to hear the square wave. Click on the **toggle** above the **mute~** object to disable the **patcher pulsewave** subpatch. Click on the same **toggle** again to unmute the subpatch.

This is similar to using the **begin~** and **selector~** objects, but the **mute~** object disables the entire subpatch. (Also, the syntax is a little different. Because of the verb "mute", a non-zero int to **mute~** has the effect of turning audio off, and 0 turns audio on.)

In the tutorial example, it really is overkill to have the output of **patcher pulsewave** go to **selector~** *and* to have a **mute~** object to mute the subpatch. However, it's done here to show a distinction. The **selector~** can cut off the flow of signal from the **patcher pulsewave** subpatch, but the MSP objects in the subpatch continue to run (because there is no **begin~** object at its beginning). The **mute~** object allows one to actually stop the processing in the subpatch, without using **begin~** and **selector~**.

- Double-click on the **patcher pulsewave** object to see its contents.

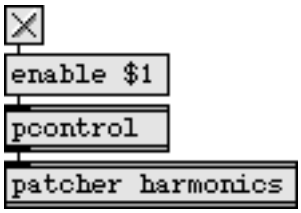


Output is 1 for half the cycle, and 0 for half the cycle

To make a square wave oscillator, we simply send the output of **phasor~**—which goes from 0 to 1—into the inlet of **<~** 0.5 (**<~** is the MSP equivalent of the Max object **<**). For the first half of each wave cycle, the output of **phasor~** is less than 0.5, so the **<~** object sends out 1. For the second half of the cycle, the output of **phasor~** is greater than 0.5, so the **<~** object sends out 0.

The **pass~** object between the **<~** object and the outlet is necessary to avoid unwelcome noise when the subpatcher is muted. It merely passes its input to its output unless the subpatcher is muted, when it outputs a zero signal. **pass~** objects are needed above any outlet of a patcher that might be muted.

Another way to disable the MSP objects in a subpatch is with the **pcontrol** object. Sending the message enable 0 to a **pcontrol** object connected to a subpatch disables all MSP objects—and all MIDI objects!—in that subpatch. The message enable 1 re-enables MIDI and audio objects in the subpatch.

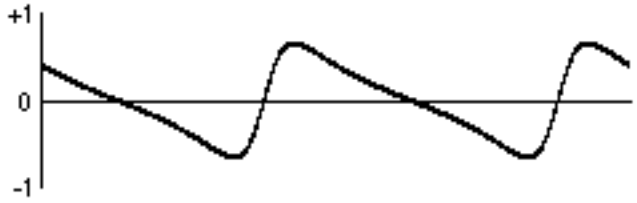


pcontrol can disable and re-enable all MIDI and audio objects in a subpatch

The **patcher harmonics** subpatch contains a complete signal network that's essentially independent of the main patch. We used **pcontrol** to disable that subpatch initially, so that it won't conflict with the sound coming from the signal network in the main patch. (Notice that **loadbang** causes an enable 0 message to be sent to **pcontrol** when the main patch is loaded, disabling the MSP objects in the subpatch.)

- Turn audio off, click on the **toggle** above the **patcher harmonics** object to enable it, then double- click on the **patcher harmonics** object to see its contents.

This subpatch combines 8 harmonically related sinusoids to create a complex tone in which the amplitude of each harmonic (harmonic number n) is proportional to $1/2^n$. Because the tones are harmonically related, their sum is a periodic wave at the fundamental frequency.



Wave produced by the patcher harmonics subpatch

The eight frequencies fuse together psychoacoustically and are heard as a single complex tone at the fundamental frequency. It is interesting to note that even when the fundamental tone is removed, the sum of the other seven harmonics still implies that fundamental, and we perceive only a loudness change and a timbral change but no change in pitch.

The same tone, minus its first harmonic, still has the same period

- Click on the startwindow message to start audio in the subpatch. Try removing and replacing the fundamental frequency by sending 0 and 1 to the **selector~**. Click on stop to turn audio off.

Summary

The startwindow message to **dac~** (or **adc~**) starts audio processing in the Patcher window that contains the **dac~**, and in any of that window's subpatches, but turns audio off in all other patches. The **mute~** object, connected to an inlet of a subpatch, can be used to disable all MSP objects in that subpatch. An enable 0 message to a **pcontrol** object connected to an inlet of a subpatch can also be used to disable all MSP objects in that subpatch. (This

disables all MIDI objects in the subpatch, too.) The **pass~** object silences the output of a subpatcher when it is muted.

You can use a **selector~** object to choose one of several signals to be passed on out the outlet, or to close off the flow of all the signals it receives. All MSP objects that are connected in a signal flow between the outlet of a **begin~** object and an inlet of a **selector~** object (or a **gate~** object) get completely disconnected from the signal network when that inlet is closed.

Any of these methods is an effective way to play selectively a subset of all the MSP objects in a given signal network (or to select one of several different networks). You can have many signal networks loaded, but only enable one at a time; in this way, you can switch quickly from one sound to another, but the computer only does processing that affects the sound you hear.

See Also

begin~	Define a switchable part of a signal network
mute~	Disable signal processing in a subpatch
pass~	Eliminate noise in a muted subpatcher
pcontrol	Open and close subwindows within a patcher
selector~	Assign one of several inputs to an outlet

Exercises in the fundamentals of MSP

In this chapter, we suggest some tasks for you to program that will test your understanding of the fundamentals of MSP presented in the *Tutorial* so far. A few hints are included to get you started. Try these three progressive exercises on your own first, in new file of your own. Then check the example patch to see a possible solution, and read on in this chapter for an explanation of the solution patch.

Exercise 1

- Write a patch that plays the note E above middle C for one second, ten times in a row, with an electric guitar-like timbre. Make it so that all you have to do is click once to turn audio on, and once to play the ten notes.

Here are a few hints:

1. The frequency of E above middle C is 329.627557 Hz.

2. For an "electric guitar-like timbre" you can use the AIFF file *gtr512.aiff* that was used in *Tutorial 3*. You'll need to read that file into a **buffer~** object, and access the **buffer~** with a **cycle~** object. In order to read the file in directly, without a dialog box to find the file, your patch and the audio file should be saved in the same folder. You can either save your patch in the *MSP Tutorial* folder or, in the Finder, option-drag a copy of the *gtr512.aiff* file into the folder where you have saved your patch.

3. Your sound will also need an amplitude envelope that is characteristic of a guitar: very fast attack, fast decay, and fairly steady (only slightly diminishing) sustain. Try using a list of line segments (target values and transition times) to a **line~** object, and using the output of **line~** to scale the amplitude of the **cycle~**.

4. To play the note ten times in a row, you'll need to trigger the amplitude envelope repeatedly at a steady rate. The Max object **metro** is well suited for that task. To stop after ten notes, your patch should either count the notes or wait a specific amount of time, then turn the **metro** off.

Exercise 2

- Modify your first patch so that, over the course of the ten repeated notes, the electric guitar sound crossfades with a sinusoidal tone a perfect 12th higher. Use a linear crossfade, with the amplitude of one sound going from 1 to 0, while the other sound goes from 0 to 1. (We discuss other ways of crossfading in a future chapter.) Send the guitar tone to the left audio output channel, and the sine tone to the right channel.

Hints:

1. You will need a second **cycle~** object to produce the tone a 12th higher.

2. To obtain the frequency that's a (just tuned) perfect 12th above E, simply multiply 329.627557 times 3. The frequency that's an equal tempered perfect 12th above E is 987.7666 Hz. Use whichever tuning you prefer.

3. In addition to the amplitude envelope for each note, you will need to change the over-all amplitude of each tone over the course of the ten seconds. This can be achieved using an additional ***~** object to scale the amplitude of each tone, slowly changing the scaling factor from 1 to 0 for one tone, and from 0 to 1 for the other.

Exercise 3

- Modify your second patch so that, over the course of the ten repeated notes, the two crossfading tones also perform an over-all *diminuendo*, diminishing to $^{1}/_{32}$ their original amplitude (i.e., by 30 dB).

Hints:

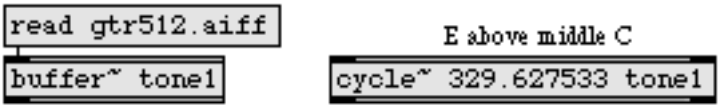
1. This will require yet another amplitude scaling factor (presumably another ***~** object) to reduce the amplitude gradually by a factor of .03125.

2. Note that if you scale the amplitude linearly from 1 to .03125 in ten seconds, the diminuendo will seem to start slowly and accelerate toward the end. That's because the linear distance between 1 and .5 (a reduction in half) is much greater than the linear distance between .0625 and .03125 (also a reduction in half). The first 6 dB of diminuendo will therefore occur in the first 5.16 seconds, but the last 6 dB reduction will occur in the last .32 seconds. So, if you want the diminuendo to be perceived as linear, you will have to adjust accordingly.

Solution to Exercise 1

- Scroll the example Patcher window all the way to the right to see one possible solution to these exercises.

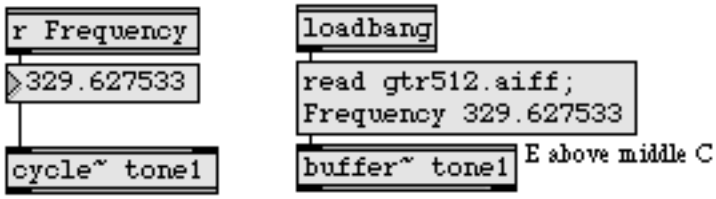
To make an oscillator with a guitar-like waveform, you need to read the audio file *gtr512.aiff* (or some similar waveform) into a **buffer~**, and then refer to that **buffer~** with a **cycle~**. (See *Tutorial 3*.)



cycle~ traverses the buffer~ 329.627533 times per second

Note that there is a limit to the precision with which Max can represent decimal numbers. When you save your patch, Max may change float values slightly. In this case, you won't hear the difference.

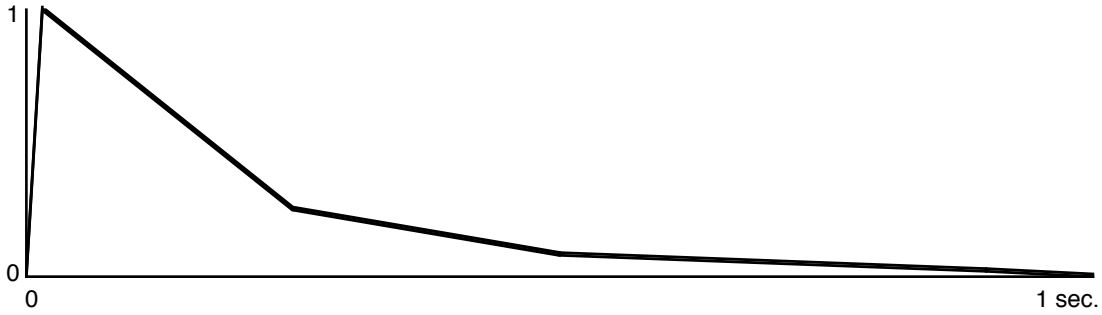
If you want the audio file to be read into the **buffer~** immediately when the patch is loaded, you can type the filename in as a second argument in the **buffer~** object, or you can use **loadbang** to trigger a read message to **buffer~**. In our solution we also chose to provide the frequency from a **number box**—which allows you to play other pitches—rather than as an argument to **cycle~**, so we also send **cycle~** an initial frequency value with **loadbang**.



loadbang is used to initialize the contents of buffer~ and the frequency of cycle~

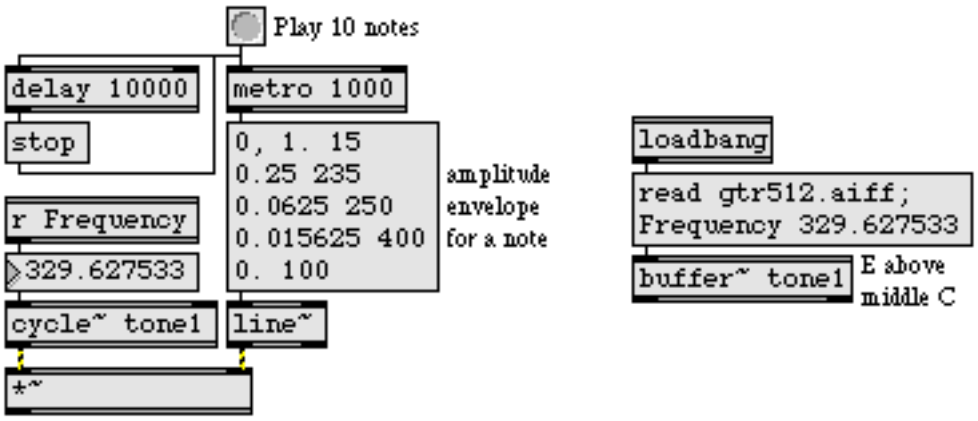
Now that we have an oscillator producing the desired tone, we need to provide an amplitude envelope to shape a note.

We chose the envelope shown below, composed of straight line segments. (See *Tutorial 3*.)



"Guitar-like" amplitude envelope

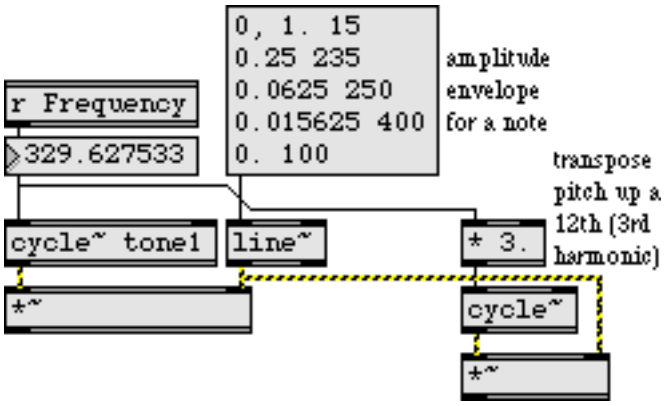
This amplitude envelope is imposed on the output of **cycle~** with a combination of **line~** and ***~**. A **metro** is used to trigger the envelope once per second, and the **metro** gets turned off after a 10-second delay.



Ten guitar-like notes are played when the button is clicked

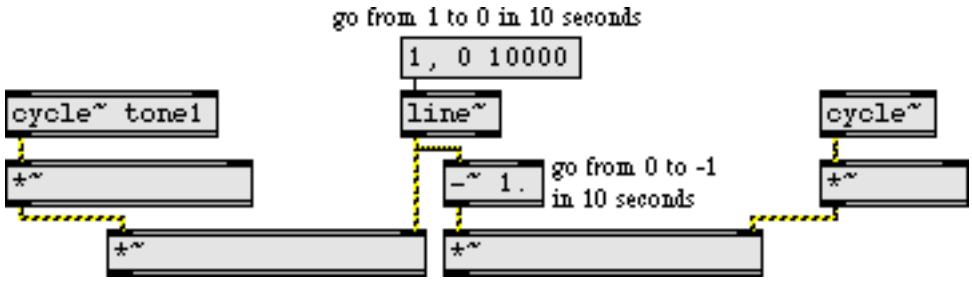
Solution to Exercise 2

For the right output channel we want a sinusoidal tone at three times the frequency (the third harmonic of the fundamental tone), with the same amplitude envelope.



Two oscillators with the same amplitude envelope and related frequencies

To crossfade between the two tones, the amplitude of the first tone must go from 1 to 0 while the amplitude of the second tone goes from 0 to 1. This can again be achieved with the combination of **line~** and ***~** for each tone.



Linear crossfade of two tones

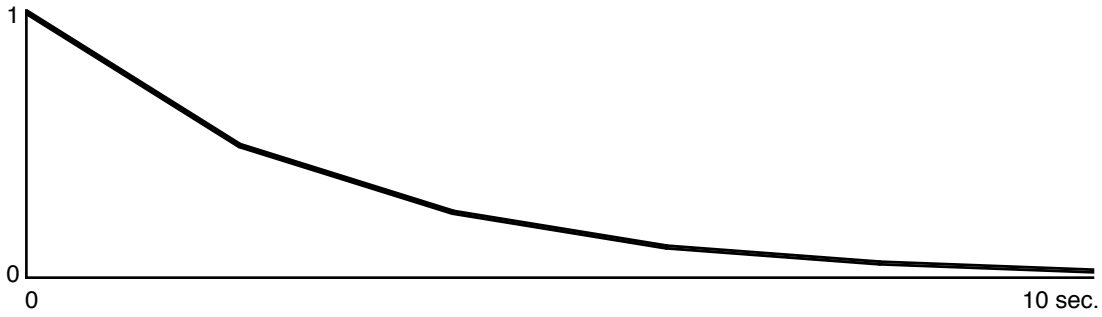
We used a little trick to economize. Rather than use a separate **line~** object to fade the second tone from 0 to 1, we just subtract 1 from the output of the existing **line~**, which gives us a ramp from 0 to -1. Perceptually this will have the same effect.

This crossfade is triggered (via **s** and **r** objects) by the same **button** that triggers the **metro**, so the crossfade starts at the same time as the ten individual notes do.

Solution to Exercise 3

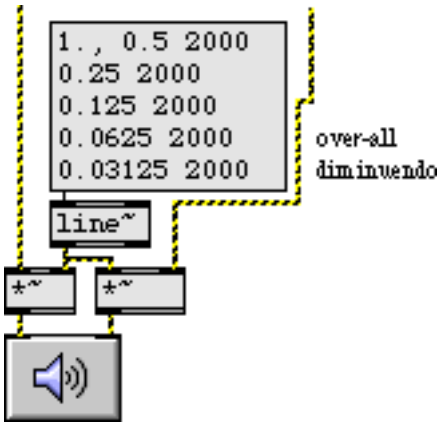
Finally, we need to use one more amplitude envelope to create a global *diminuendo*. The two tones go to yet another ***~** object, controlled by another **line~**. As noted earlier, a straight line decrease in amplitude will not give the perception of constant diminuendo in loudness.

Therefore, we used five line segments to simulate a curve that decreases by half every two seconds.



Global amplitude envelope decreasing by half every two seconds

This global amplitude envelope is inserted in the signal network to scale both tones down smoothly by a factor of .03125 over 10 seconds.

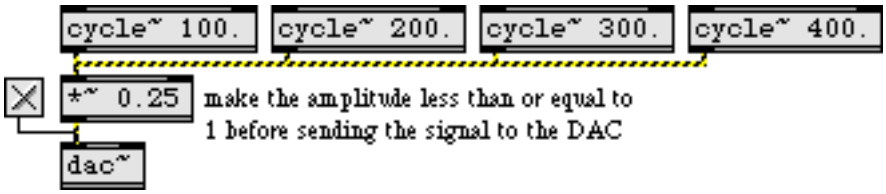


Both tones are scaled by the same global envelope

In the tutorial examples up to this point we have synthesized sound using basic waveforms. In the next few chapters we'll explore a few other well known synthesis techniques using sinusoidal waves. Most of these techniques are derived from pre-computer analog synthesis methods, but they are nevertheless instructive and useful.

Combining tones

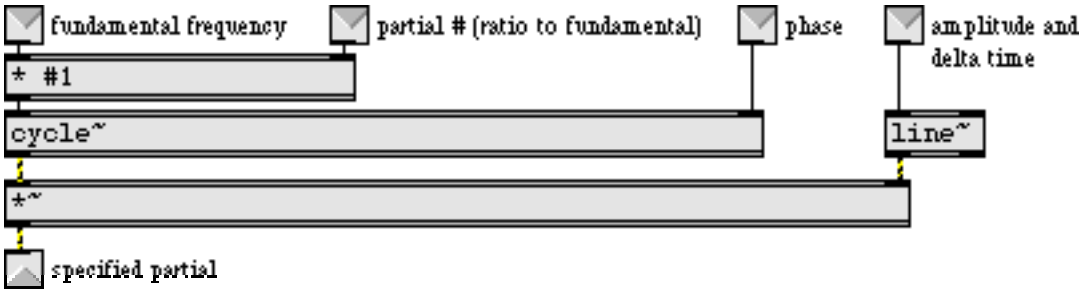
A sine wave contains energy at a single frequency. Since complex tones, by definition, are composed of energy at several (or many) different frequencies, one obvious way to synthesize complex tones is to use multiple sine wave oscillators and add them together.



Four sinusoids added together to make a complex tone

Of course, you can add any waveforms together to produce a composite tone, but we'll stick with sine waves in this tutorial example. Synthesizing complex tones by adding sine waves is a somewhat tedious method, but it does give complete control over the amplitude and frequency of each component (*partial*) of the complex tone.

In the tutorial patch, we add together six cosine oscillators (**cycle~** objects), with independent control over the frequency, amplitude, and phase of each one. In order to simplify the patch, we designed a subpatch called **partial~** which allows us to specify the frequency of each partial as a ratio relative to a fundamental frequency.



The contents of the subpatch partial~

For example, if we want a partial to have a frequency twice that of the fundamental we just type in 2.0 as an argument (or send it in the second inlet). This way, if several **partial~** objects are receiving their fundamental frequency value (in the left inlet) from the same

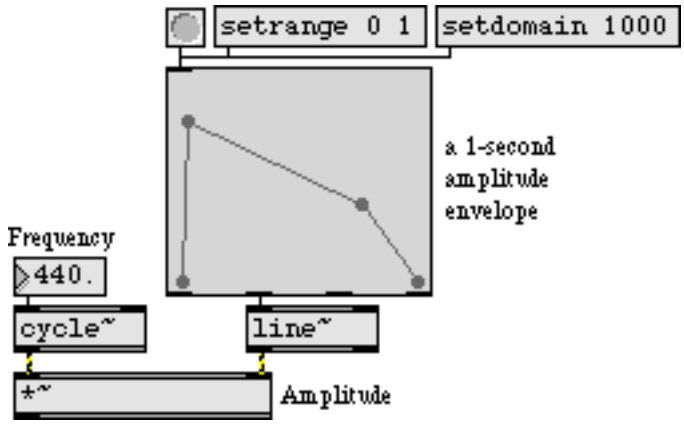
source, their relative frequencies will stay the same even when the value of the fundamental frequency changes.

Of course, for the sound to be very interesting, the amplitudes of the partials must evolve with relative independence. Therefore, in the main patch, we control the amplitude of each partial with its own *envelope generator*.

Envelope generator: function

In *Tutorial 3* you saw how to create an amplitude envelope by sending a list of pairs of numbers to a **line~** object, thus giving it a succession of target values and transition times. This idea of creating a control function from a series of line segments is useful in many contexts—generating amplitude envelopes happens to be one particularly common usage—and it is demonstrated in *Tutorial 6*, as well.

The **function** object is a great help in generating such line segment functions, because it allows you to draw in the shape that you want, as well as define the function's domain and range (the numerical value of its dimensions on the *x* and *y* axes). You can draw a function simply by clicking with the mouse where you want each breakpoint to appear. When **function** receives a bang, it sends a list of value-time pairs out its 2nd outlet. That list, when used as input to the **line~** object, produces a changing signal that corresponds to the shape drawn.



function is a graphic function generator for a control signal when used with line~

By the way, **function** is also useful for non-signal purposes in Max. It can be used as an interpolating lookup table. When it receives a number in its inlet, it considers that number to be an *x* value and it looks up the corresponding *y* value in the drawn function (interpolating between breakpoints as necessary) and sends it out the left outlet.

A variety of complex tones

Even with only six partials, one can make a variety of timbres ranging from "realistic" instrument- like tones to clearly artificial combinations of frequencies. The settings for a few different tones have been stored in a **preset** object, for you to try them out. A brief explanation of each tone is provided below.

- Click on the **ezdac~** speaker icon to turn audio on. Click on the button to play a tone. Click on one of the stored presets in the **preset** object to change the settings, then click the button again to hear the new tone.

Preset 1. This tone is not really meant to emulate a real instrument. It's just a set of harmonically related partials, each one of which has a different amplitude envelope. Notice how the timbre of the tone changes slightly over the course of its duration as different partials come to the foreground. (If you can't really notice that change of timbre, try changing the note's duration to something longer, such as 8000 milliseconds, to hear the note evolve more slowly.)

Preset 2. This tone sounds rather like a church organ. The partials are all octaves of the fundamental, the attack is moderately fast but not percussive, and the amplitude of the tone does not diminish much over the course of the note. You can see and hear that the upper partials die away more quickly than the lower ones.

Preset 3. This tone consists of slightly mistuned harmonic partials. The attack is immediate and the amplitude decays rather rapidly after the initial attack, giving the note a percussive or plucked effect.

Preset 4. The amplitude envelopes for the partials in this tone are derived from an analysis of a trumpet note in the lower register. Of course, these are only six of the many partials present in a real trumpet sound.

Preset 5. The amplitude envelopes for the partials of this tone are derived from the same trumpet analysis. However, in this case, only the odd-numbered harmonics are used. This creates a tone more like a clarinet, because the cylindrical bore of a clarinet resonates the odd harmonics. Also, the longer duration of this note slows down the entire envelope, giving it a more characteristically clarinet-like attack.

Preset 6. This is a completely artificial tone. The lowest partial enters first, followed by the sixth partial a semitone higher. Eventually the remaining partials enter, with frequencies that lie between the first and sixth partial, creating a microtonal cluster. The beating effect is due to the interference between these waves of slightly different frequency.

Preset 7. In this case the partials are spaced a major second apart, and the amplitude of each partial rises and falls in such a way as to create a composite effect of an arpeggiated whole-tone cluster. Although this is clearly a whole-tone chord rather than a single tone, the gradual and overlapping attacks and decays cause the tones to fuse together fairly successfully.

Preset 8. In this tone the partials suggest a harmonic spectrum strongly enough that we still get a sense of a fundamental pitch, but they are sufficiently mistuned that they resemble the inharmonic spectrum of a bell. The percussive attack, rapid decay, and independently varying partials during the sustain portion of the note are also all characteristic of a struck metal bell.

Notice that when you are adding several signals together like this, their sum will often exceed the amplitude limits of the **dac~** object, so the over-all amplitude must be scaled appropriately with a ***~** object.

Experiment with complex tones

- Using these tones as starting points, you may want to try designing your own tones with this additive synthesis patch. Vary the tones by changing the fundamental frequency, partials, and duration of the preset tones. You can also change the envelopes by dragging on the breakpoints.

To draw a function in the **function** object:

- Click in the **function** object to create a new breakpoint. If you click and drag, the x and y coordinates of the point are shown in the upper portion of the object, and you can immediately move the breakpoint to the position you want.
- Similarly, you can click and drag on any existing breakpoint to move it.
- Shift-click on an existing point to delete it.

Although not demonstrated in this tutorial, it is also possible to create, move, and delete breakpoints in a **function** just by using Max messages. See the description of **function** in the *Objects* section of the manual for details.

The message setdomain, followed by a number, changes the scale of the x axis in the **function** without changing the shape of the envelope. When you change the number in the "Duration" **number box**, it sends a setdomain message to the **function**.

Tutorial 7

Summary

Additive synthesis is the process of synthesizing new complex tones by adding tones together. Since pure sine tones have energy at only one frequency, they are the fundamental building blocks of additive synthesis, but of course any signals can be added together. The sum signal may need to by scaled by some constant signal value less than 1 in order to keep it from being clipped by the DAC.

In order for the timbre of a complex tone to remain the same when its pitch changes, each partial must maintain its relationship to the fundamental frequency. Stating the frequency of each partial in terms of a ratio to (i.e., a multiplier of) the fundamental frequency maintains the tone's spectrum even when the fundamental frequency changes.

In order for a complex tone to have an interesting timbre, the amplitude of the partials must change with a certain degree of independence. The **function** object allows you to draw control shapes such as amplitude envelopes, and when **function** receives a `bang` it describes that shape to a **line~** object to generate a corresponding control signal.

See Also

function	Graphical function breakpoint editor

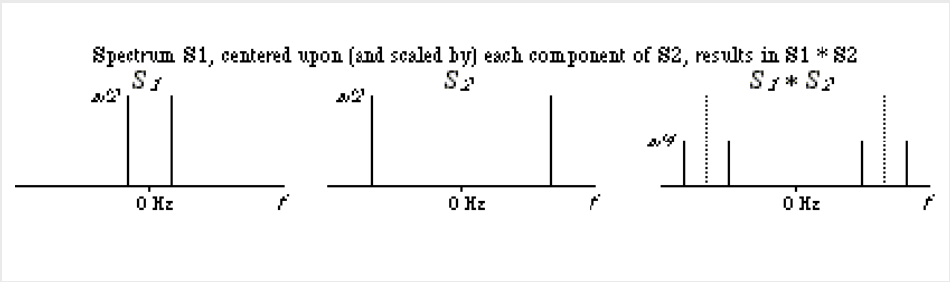
Tutorial 8: Synthesis—Tremolo and ring modulation

Multiplying signals

In the previous chapter we added sine tones together to make a complex tone. In this chapter we will see how a very different effect can be achieved by *multiplying* signals. Multiplying one wave by another—i.e., multiplying their instantaneous amplitudes, sample by sample—creates an effect known as *ring modulation* (or, more generally, *amplitude modulation*). "Modulation" in this case simply means change; the amplitude of one waveform is changed continuously by the amplitude of another.

Technical detail: Multiplication of waveforms in the time domain is equivalent to convolution of waveforms in the frequency domain. One way to understand convolution is as the superimposition of one spectrum on every frequency of another spectrum. Given two spectra *S1* and *S2*, each of which contains many different frequencies all at different amplitudes, make a copy of *S1* at the location of every frequency in *S2*, with each copy scaled by the amplitude of that particular frequency of *S2*.

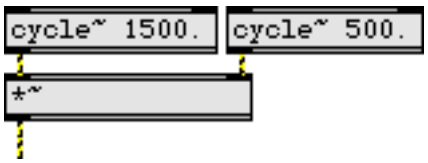
Since a cosine wave has equal amplitude at both positive and negative frequencies, its spectrum contains energy (equally divided) at both f and $-f$. When convolved with another cosine wave, then, a scaled copy of (both the positive and negative frequency components of) the one wave is centered around both the positive and negative frequency components of the other.



Multiplication in the time domain is equivalent to convolution in the frequency domain

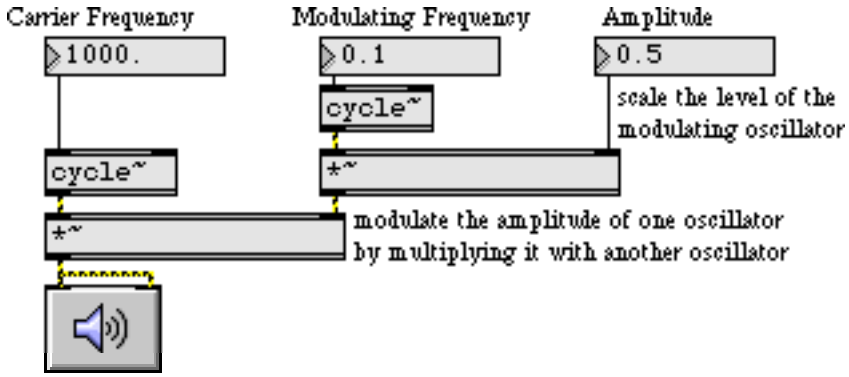
In our example patch, we multiply two sinusoidal tones. Ring modulation (multiplication) can be performed with any signals, and in fact the most sonically interesting uses of ring modulation involve complex tones.

However, we'll stick to sine tones in this example for the sake of simplicity, to allow you to hear clearly the effects of signal multiplication.



Simple multiplication of two cosine waves

The tutorial patch contains two **cycle~** objects, and the outlet of each one is connected to one of the inlets of a ***~** object. However, the output of one of the **cycle~** objects is first scaled by an additional ***~** object, which provides control of the over-all amplitude of the result. (Without this, the over-all amplitude of the product of the two **cycle~** objects would always be 1.)



Product of two cosine waves (one with amplitude scaled beforehand)

Tremolo

When you first open the patch, a **loadbang** object initializes the frequency and amplitude of the oscillators. One oscillator is at an audio frequency of 1000 Hz. The other is at a sub-audio frequency of 0.1 Hz (one cycle every ten seconds). The 1000 Hz tone is the one we hear (this is termed the *carrier* oscillator), and it is modulated by the other wave (called the *modulator*) such that we hear the amplitude of the 1000 Hz tone dip to 0 whenever the 0.1 Hz cosine goes to 0. (Twice per cycle, meaning once every five seconds.)

- Click on the **ezdac~** to turn audio on. You will hear the amplitude of the 1000 Hz tone rise and fall according to the cosine curve of the modulator, which completes one full cycle every ten seconds. (When the modulator is negative, it inverts the carrier, but we don't hear the difference, so the effect is of two equivalent dips in amplitude per modulation period.)

The amplitude is equal to the product of the two waves. Since the peak amplitude of the carrier is 1, the over-all amplitude is equal to the amplitude of the modulator.

- Drag on the "Amplitude" **number box** to adjust the sound to a comfortable level. Click on the **message** box containing the number 1 to change the modulator rate.

With the modulator rate set at 1, you hear the amplitude dip to 0 two times per second. Such a periodic fluctuation in amplitude is known as *tremolo*. (Note that this is distinct from *vibrato*, a term usually used to describe a periodic fluctuation in pitch or frequency.) The perceived rate of tremolo is equal to two times the modulator rate, since the amplitude goes to 0 twice per cycle. As described on the previous page, ring modulation produces the sum and difference frequencies, so you're actually hearing the frequencies 1001 Hz and 999 Hz, and the 2 Hz beating due to the interference between those two frequencies.

- One at a time, click on the **message box** objects containing 2 and 4. What tremolo rates do you hear? The sound is still like a single tone of fluctuating amplitude because the sum and difference tones are too close in frequency for you to separate them successfully, but can you calculate what frequencies you're actually hearing?

- Now try setting the rate of the modulator to 8 Hz, then 16 Hz.

In these cases the rate of tremolo borders on the audio range. We can no longer hear the tremolo as distinct fluctuations, and the tremolo just adds a unique sort of "roughness" to the sound. The sum and difference frequencies are now far enough apart that they no longer fuse together in our perception as a single tone, but they still lie within what psychoacousticians call the critical band. Within this *critical band* we have trouble hearing the two separate tones as a pitch interval, presumably because they both affect the same region of our basilar membrane.

Sidebands

- Try setting the rate of the modulator to 32 Hz, then 50 Hz.

At a modulation rate of 32 Hz, you can hear the two tones as a pitch interval (approximately a minor second), but the sensation of roughness persists. With a modulation rate of 50 Hz, the sum and difference frequencies are 1050 Hz and 950 Hz—a pitch interval almost as great as a major second—and the roughness is mostly gone. You might also hear the tremolo rate itself, as a tone at 100 Hz.

You can see that this type of modulation produces new frequencies not present in the carrier and modulator tones. These additional frequencies, on either side of the carrier frequency, are often called sidebands.

- Listen to the remaining modulation rates.

At certain modulation rates, all the sidebands are aligned in a harmonic relationship. With a modulation rate of 200 Hz, for example, the tremolo rate is 400 Hz and the sum and difference frequencies are 800 Hz and 1200 Hz. Similarly, with a modulation rate of 500 Hz, the tremolo rate is 1000 Hz and the sum and difference frequencies are 500 Hz and 1500 Hz. In these cases, the sidebands fuse together more tightly as a single complex tone.

- Experiment with other carrier and modulator frequencies by typing other values into the **number box** objects. When you have finished, click on **ezdac~** again to turn audio off.

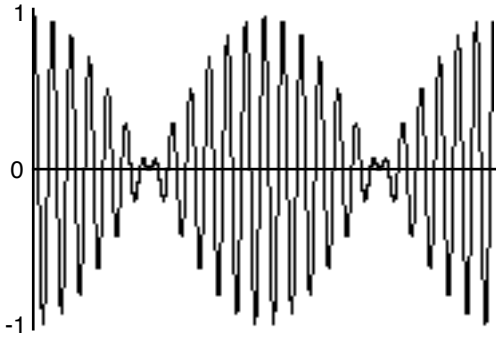
Summary

Multiplication of two digital signals is comparable to the analog audio technique known as *ring modulation*. Ring modulation is a type of *amplitude modulation*—changing the amplitude of one tone (termed the *carrier*) with the amplitude of another tone (called the *modulator*). Multiplication of signals in the time domain is equivalent to convolution of spectra in the frequency domain.

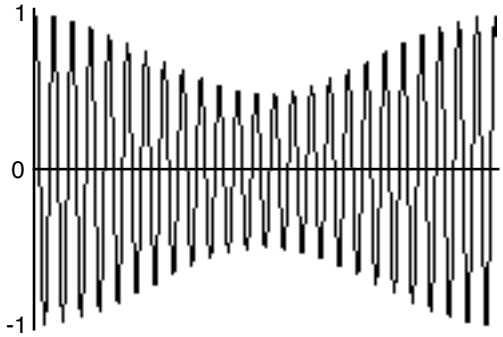
Multiplying an audio signal by a sub-audio signal results in regular fluctuations of amplitude known as *tremolo*. Multiplication of signals creates *sidebands*—additional frequencies not present in the original tones. Multiplying two sinusoidal tones produces energy at the sum and difference of the two frequencies. This can create beating due to interference of waves with similar frequencies, or can create a fused complex tone when the frequencies are harmonically related. When two signals are multiplied, the output amplitude is determined by the product of the carrier and modulator amplitudes.

Ring modulation and amplitude modulation

Amplitude modulation (AM) involves changing the amplitude of a "carrier" signal using the output of another "modulator" signal. In the specific AM case of ring modulation (discussed in Tutorial 8) the two signals are simply multiplied. In the more general case, the modulator is used to alter the carrier's amplitude, but is not the sole determinant of it. To put it another way, the modulator can cause fluctuation of amplitude around some value other than 0. The example below illustrates the difference between ring modulation and more common amplitude modulation.



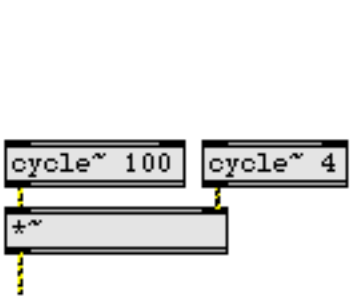
Ring modulation



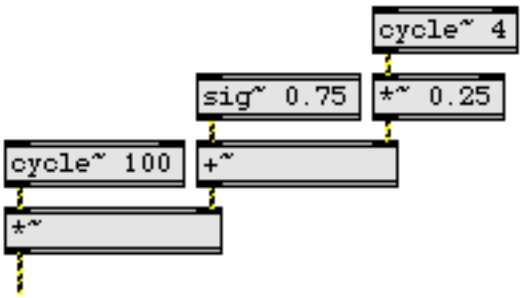
Amplitude modulation

The example on the left is $^1/_4$ second of a 100 Hz cosine multiplied by a 4 Hz cosine; the amplitude of both cosines is 1. In the example on the right, the 4 Hz cosine has an amplitude of 0.25, which is used to vary the amplitude of the 100 Hz tone ±0.25 around 0.75 (going as low as 0.5 and as high as 1.0). The two main differences are a) the AM example never goes all the way to 0, whereas the ring modulation example does, and b) the ring modulation is perceived as two amplitude dips per modulation period (thus creating a tremolo effect at twice the rate of the modulation) whereas the AM is perceived as a single cosine fluctuation per modulation period.

The two MSP patches that made these examples are shown below.



Ring modulation



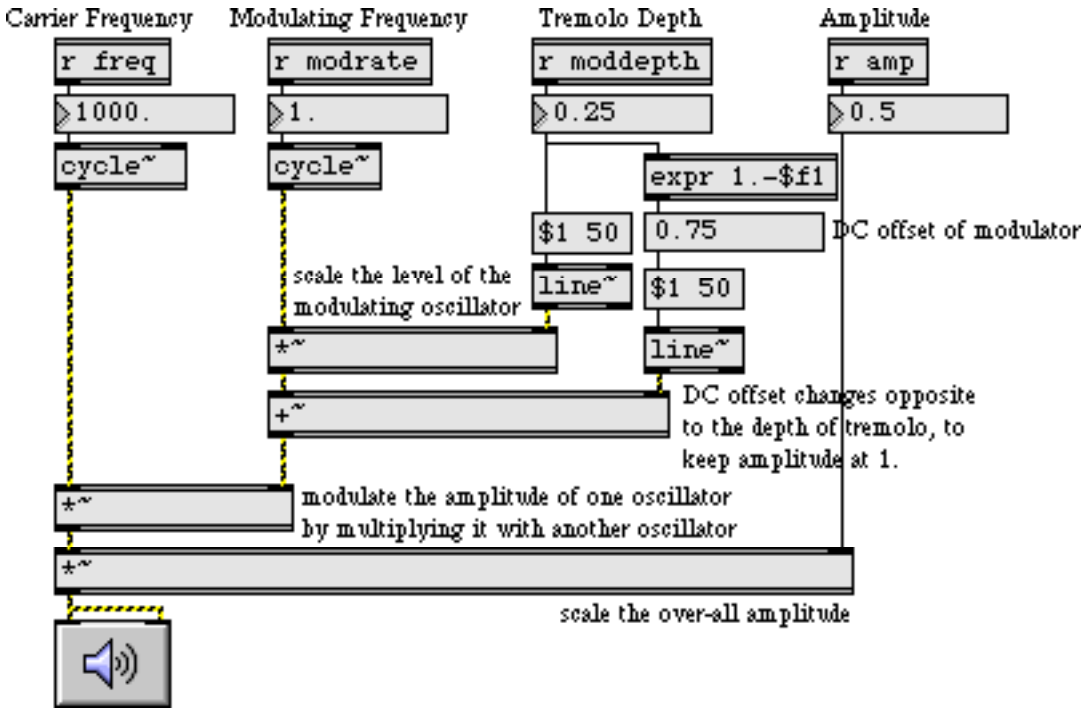
Amplitude modulation

The difference in effect is due to the constant value of 0.75 in the AM patch, which is varied by a modulator of lesser amplitude. This constant value can be thought of as the carrier's amplitude, which is varied by the instantaneous amplitude of the modulator. The amplitude still varies according to the shape of the modulator, but the modulator is not centered on 0.

> **Technical detail:** The amount that a wave is offset from 0 is called the DC offset. A constant amplitude value such as this represents spectral energy at the frequency 0 Hz. The modulator in AM has a DC offset, which distinguishes it from ring modulation.

Implementing AM in MSP

The tutorial patch is designed in such a way that the DC offset of the modulator is always 1 minus the amplitude of its sinusoidal variation. That way, the peak amplitude of the modulator is always 1, so the product of carrier and modulator is always 1. A separate ***~** object is used to control the over-all amplitude of the sound.



The modulator is a sinusoid with a DC offset, which is multiplied by the carrier

- Click on the **ezdac~** to turn audio on. Notice how the tremolo rate is the same as the frequency of the modulator. Click on the **message** boxes 2, 4, and 8 in turn to hear different tremolo rates.

Achieving different AM effects

The primary merit of AM lies in the fact that the intensity of its effect can be varied by changing the amplitude of the modulator.

- To hear a very slight tremolo effect, type the value 0.03 into the **number box** marked "Tremolo Depth". The modulator now varies around 0.97, from 1 to 0.94, producing an amplitude variation of only about half a decibel. To hear an extreme tremolo effect, change the tremolo depth to 0.5; the modulator now varies from 1 to 0—the maximum modulation possible.

Amplitude modulation produces sidebands—additional frequencies not present in the carrier or the modulator—equal to the sum and the difference of the frequencies present in the carrier and modulator. The presence of a DC offset (technically energy at 0 Hz) in the modulator means that the carrier tone remains present in the output, too (which is not the case with ring modulation).

- Click on the **message** boxes containing the numbers 32, 50, 100, and 150, in turn. You will hear the carrier frequency, the modulator frequency (which is now in the low end of the audio range), and the sum and difference frequencies.

When there is a harmonic relationship between the carrier and the modulator, the frequencies produced belong to the harmonic series of a common fundamental, and tend to fuse more as a single complex tone. For example, with a carrier frequency of 1000 Hz and a modulator at 250 Hz, you will hear the frequencies 250 Hz, 750 Hz, 1000 Hz, and 1250 Hz— the 1st, 3rd, 4th, and 5th harmonics of the fundamental at 250 Hz.

- Click on the **message** boxes containing the numbers 200, 250, and 500 in turn to hear harmonic complex tones. Drag on the "Tremolo Depth" **number box** to change the depth value between 0. and 0.5, and listen to the effect on the relative strength of the sidebands.

- Explore different possibilities by changing the values in the **number box** objects. When you have finished, click on the **ezdac~** to turn audio off.

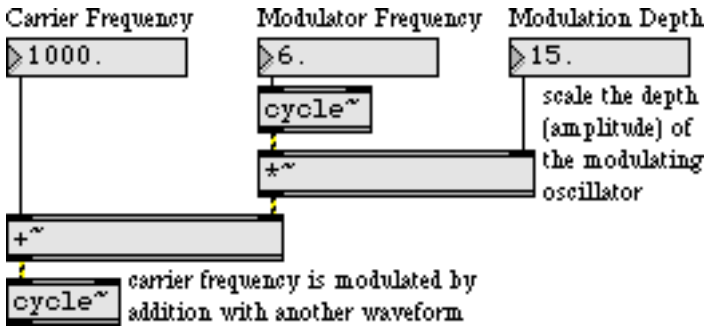
It is worth noting that any audio signals can be used as the carrier and modulator tones, and in fact many interesting results can be obtained by amplitude modulation with complex tones. (Tutorial 23 allows you to perform amplitude modulation on the sound coming into the computer.)

Summary

The amplitude of an audio (carrier) signal can be modulated by another (modulator) signal, either by simple multiplication (ring modulation) or by adding a time-varying modulating signal to a constant signal (DC offset) before multiplying it with the carrier signal (amplitude modulation). The intensity of the amplitude modulation can be controlled by increasing or reducing the amplitude of the time-varying modulator relative to its DC offset. When the modulator has a DC offset, the carrier frequency will remain present in the output sound, along with sidebands at frequencies determined by the sum and the difference of the carrier and the modulator. At sub-audio modulating frequencies, amplitude modulation is heard as tremolo; at audio frequencies the carrier, modulator, and sidebands are all heard as a chord or as a complex tone.

Basic FM in MSP

Frequency modulation (FM) is a change in the frequency of one signal caused by modulating it with another signal. In the most common implementation, the frequency of a sinusoidal carrier wave is varied continuously with the output of a sinusoidal modulating oscillator. The modulator is *added* to the constant base frequency of the carrier.



Simple frequency modulation

The example above shows the basic configuration for FM. The frequency of the modulating oscillator determines the rate of modulation, and the amplitude of the modulator determines the "depth" (intensity) of the effect.

- Click on the **ezdac~** to turn audio on.

The sinusoidal movement of the modulator causes the frequency of the carrier to go as high as 1015 Hz and as low as 885 Hz. This frequency variation completes six cycles per second, so we hear a 6 Hz vibrato centered around 1000 Hz. (Note that this is distinct from tremolo, which is a fluctuation in amplitude, not frequency.)

- Drag upward on the **number box** marked "Modulation Depth" to change the amplitude of the modulator. The vibrato becomes wider and wider as the modulator amplitude increases. Set the modulation depth to 500.

With such a drastic frequency modulation, one no longer really hears the carrier frequency. The tone passes through 1000 Hz so fast that we don't hear that as its frequency. Instead we hear the extremes—500 Hz and 1500 Hz—because the output frequency actually spends more time in those areas.

Note that 500 Hz is an octave below 1000 Hz, while 1500 Hz is only a perfect fifth above 1000 Hz. The interval between 500 Hz and 1500 Hz is thus a perfect 12th (as one would expect, given their 1:3 ratio). So you can see that a vibrato of equal frequency variation

around a central frequency does not produce equal pitch variation above and below the central pitch. (In *Tutorial 17* we demonstrate how to make a vibrato that is equal in pitch up and down.)

- Set the modulation depth to 1000. Now begin dragging the "Modulator Frequency" **number box** upward slowly to hear a variety of effects.

As the modulator frequency approaches the audio range, you no longer hear individual oscillations of the modulator. The modulation rate itself is heard as a low tone. As the modulation frequency gets well into the audio range (at about 50 Hz), you begin to hear a complex combination of sidebands produced by the FM process. The precise frequencies of these sidebands depend on the relationship between the carrier and modulator frequencies.

- Drag the "Modulator Frequency" **number box** all the way up to 1000. Notice that the result is a rich harmonic tone with fundamental frequency of 1000 Hz. Try typing in modulator frequencies of 500, 250, and 125 and note the change in perceived fundamental.

In each of these cases, the perceived fundamental is the same as the modulator frequency. In fact, though, it is not determined just by the modulator frequency, but rather by the relationship between carrier frequency and modulator frequency. This will be examined more in the next chapter.

- Type in 125 as the modulator frequency. Now drag up and down on the "Modulation Depth" **number box**, making drastic changes. Notice that the pitch stays the same but the timbre changes.

The timbre of an FM tone depends on the ratio of modulator amplitude to modulator frequency. This, too, will be discussed more in the next chapter.

Summary

Frequency modulation (FM) is achieved by adding a time-varying signal to the constant frequency of an oscillator. It is good for vibrato effects at sub-audio modulating frequencies, and can produce a wide variety of timbres at audio modulating frequencies. The rich complex tones created with FM contain many partials, even though only two oscillators are needed to make the sound. This is a great improvement over additive synthesis, in terms of computational efficiency.

Tutorial 11: Synthesis—Frequency modulation

Elements of FM synthesis

Frequency modulation (FM) has proved to be a very versatile and effective means of synthesizing a wide variety of musical tones. FM is very good for emulating acoustic instruments, and for producing complex and unusual tones in a computationally efficient manner.

Modulating the frequency of one wave with another wave generates many sidebands, resulting in many more frequencies in the output sound than were present in the carrier and modulator waves themselves. As was mentioned briefly in the previous chapter, the frequencies of the sidebands are determined by the relationship between the carrier frequency (*Fc*) and the modulator frequency (*Fm*); the relative strength of the different sidebands (which affects the timbre) is determined by the relationship between the modulator amplitude (*Am*) and the modulator frequency (*Fm*).

Because of these relationships, it's possible to boil the control of FM synthesis down to two crucial values, which are defined as ratios of the pertinent parameters. One important value is the *harmonicity ratio*, defined as *Fm/Fc*; this will determine what frequencies are present in the output tone, and whether the frequencies have an harmonic or inharmonic relationship. The second important value is the *modulation index*, defined as *Am/Fm;* this value affects the "brightness" of the timbre by affecting the relative strength of the partials.

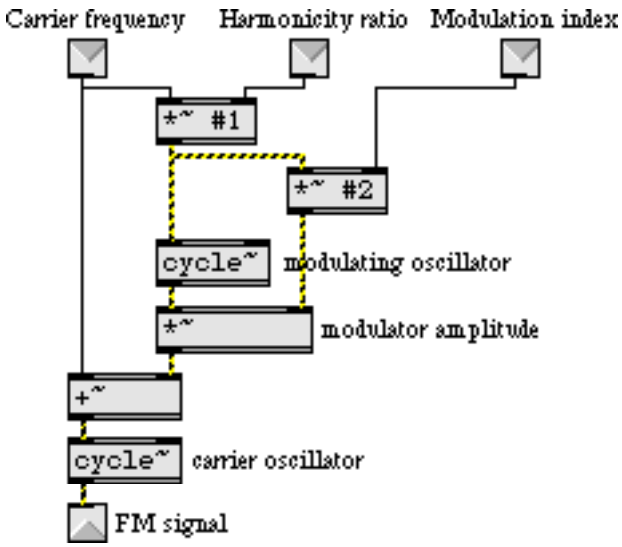
> **Technical detail:** In John Chowning's article "Synthesis of Complex Audio Spectra by Means of Frequency Modulation" and in Curtis Roads' Computer Music Tutorial, they write about the ratio *Fc/Fm*. However, in F.R. Moore's Elements of Computer Music he defines the term harmonicity ratio as *Fm/Fc.* The idea in all cases is the same, to express the relationship between the carrier and modulator frequencies as a ratio. In this tutorial we use Moore's definition because that way whenever the harmonicity ratio is an integer the result will be a harmonic tone with *Fc a*s the fundamental.

The frequencies of the sidebands are determined by the sum and difference of the carrier frequency plus and minus integer multiples of the modulator frequency. Thus, the frequencies present in an FM tone will be *Fc*, *Fc+Fm, Fc-Fm*, *Fc+2Fm*, *Fc-2Fm*, *Fc+3Fm*, *Fc-3Fm*, etc. This holds true even if the difference frequency turns out to be a negative number; the negative frequencies are heard as if they were positive. The number and strength of sidebands present is determined by the modulation index; the greater the index, the greater the number of sidebands of significant energy.

An FM subpatch: simpleFM~

The **simpleFM~** object in this tutorial patch is not an MSP object; it's a subpatch that implements the ideas of harmonicity ratio and modulation index.

- Double-click on the **simpleFM~** subpatch object to see its contents.



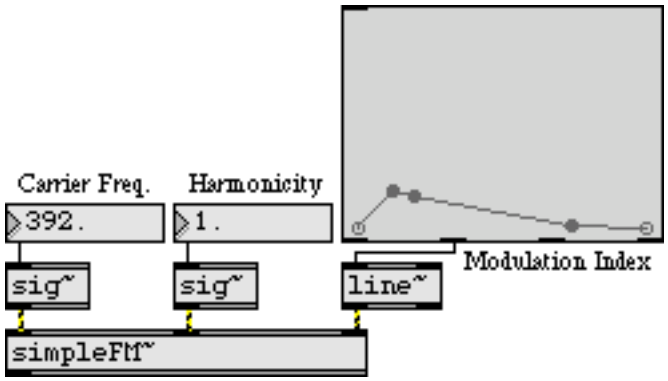
The simpleFM~ subpatch

The main asset of this subpatch is that it enables one to specify the carrier frequency, harmonicity ratio, and modulation index, and it then calculates the necessary modulator frequency and modulator amplitude (in the ***~** objects) to generate the correct FM signal. The subpatch is flexible in that it accepts either signals or numbers in its inlets, and the harmonicity ratio and modulation index can be typed in as arguments in the main patch.

- Close the [**simpleFM~**] window.

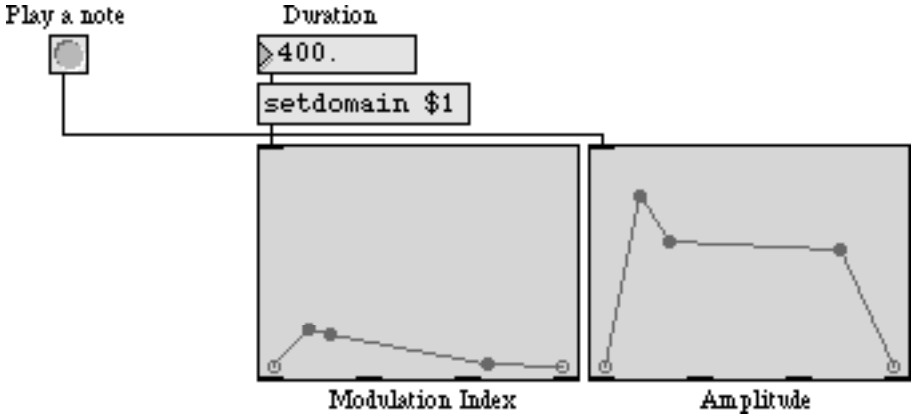
Producing different FM tones

In the main patch, the carrier frequency and harmonicity ratio are provided to **simpleFM~** as constant values, and the modulation index is provided as a time-varying signal generated by the envelope in the **function** object.



Providing values for the FM instrument

Because modulation index is the main determinant of timbre (brightness), and because the timbre of most real sounds varies over time, the modulation index is a prime candidate to be controlled by an envelope. This timbre envelope may or may not correspond exactly with the amplitude of the sound, so in the main patch one envelope is used to control amplitude, and another to control brightness.



Over the course of the note, the timbre and the amplitude evolve independently

Each of the presets contains settings to produce a different kind of FM tone, as described below.

- Turn audio on and click on the first preset in the **preset** object to recall some settings for the instrument. Click on the **button** to play a note. To hear each of the different preset tones, click on a different preset in the **preset** object to recall the settings for the instrument, then click on the **button** to play a note.

Preset 1. The carrier frequency is for the pitch C an octave below middle C. The non-integer value for the harmonicity ratio will cause an inharmonic set of partials. This inharmonic spectrum, the steady drop in modulation index from bright to pure, and the long exponential amplitude decay all combine to make a metallic bell-like tone.

Preset 2. This tone is similar to the first one, but with a (slightly mistuned) harmonic value for the harmonicity ratio, so the tone is more like an electric piano.

Preset 3. An "irrational" (1 over the square root of 2) value for the harmonicity ratio, a low modulation index, a short duration, and a characteristic envelope combine to give this tone a quasi- pitched drum-like quality.

Preset 4. In brass instruments the brightness is closely correlated with the loudness. So, to achieve a trumpet-like sound in this example the modulation index envelope essentially tracks the amplitude envelope. The amplitude envelope is also characteristic of brass instruments, with a slow attack and little decay. The pitch is G above middle C, and the harmonicity ratio is 1 for a fully harmonic spectrum.

Preset 5. On the trumpet, a higher note generally requires a more forceful attack; so the same envelope applied to a shorter duration, and a carrier frequency for the pitch high C, emulate a staccato high trumpet note.

Preset 6. The same pitch and harmonicity, but with a percussive attack and a low modulation index, give a xylophone sound.

Preset 7. A harmonicity ratio of 4 gives a spectrum that emphasizes odd harmonics. This, combined with a low modulation index and a slow attack, produces a clarinet-like tone.

Preset 8. Of course, the real fun of FM synthesis is the surreal timbres you can make by choosing unorthodox values for the different parameters. Here, an extreme and wildly fluctuating modulation index produces a sound unlike that produced by any acoustic object.

- You can experiment with your own envelopes and settings to discover new FM sounds. When you have finished, click on the **ezdac~** to turn audio off.

As with amplitude modulation, frequency modulation can also be performed using complex tones. Sinusoids have traditionally been used most because they give the most

predictable results, but many other interesting sounds can be obtained by using complex tones for the carrier and modulator signals.

Summary

FM synthesis is an effective technique for emulating acoustic instrumental sounds as well as for generating unusual new sounds.

The frequencies present in an FM tone are equal to the carrier frequency plus and minus integer multiples of the modulator frequency. Therefore, the harmonicity of the tone can be described by a single number—the ratio of the modulator and carrier frequencies—sometimes called the *harmonicity ratio.* The relative amplitude of the partials is dependent on the ratio of the modulator's amplitude to its frequency, known as the *modulation index*.

In most acoustic instruments, the timbre changes over the course of a note, so envelope control of the modulation index is appropriate for producing interesting sounds. A non-integer harmonicity ratio yields an inharmonic spectrum, and when combined with a percussive amplitude envelope can produce drum-like and bell-like sounds. An integer harmonicity ratio combined with the proper modulation index envelope and amplitude envelope can produce a variety of pitched instrument sounds.

Using a stored wavetable

In *Tutorial 3* we used 512 samples stored in a **buffer~** as a wavetable to be read by the **cycle~** object. The name of the **buffer~** object is typed in as an argument to the **cycle~** object, causing **cycle~** to use samples from the **buffer~** as its waveform, instead of its default cosine wave. The frequency value received in the left inlet of the **cycle~** determines how many times per second it will read through those 512 samples, and thus determines the fundamental frequency of the tone it plays.

Just to serve as a reminder, an example of that type of wavetable synthesis is included in the lower right corner of this tutorial patch.

The cycle~ object reads repeatedly through the 512 samples stored in the buffer~

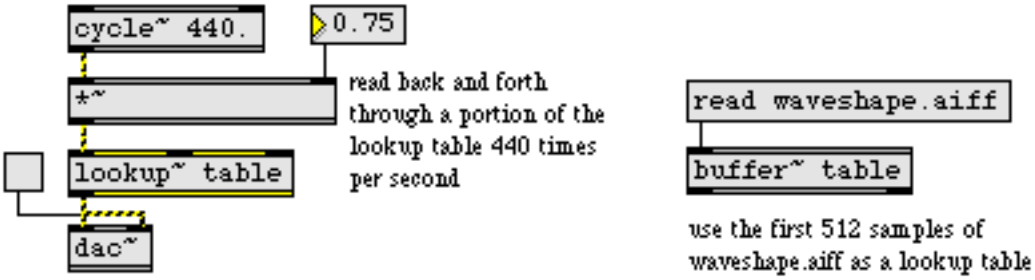
- Double-click on the **buffer~** object to see its contents. The file *gtr512.aiff* contains one cycle of a recorded electric guitar note. Click on the **ezdac~** speaker icon to turn audio on. Click on the **toggle** to open the **gate~**, allowing the output of **cycle~** to reach the **dac~**. Click on the **toggle** again to close the **gate~**.

This type of synthesis allows you to use any waveform for **cycle~**, but the timbre is static and somewhat lifeless because the waveform is unchanging. This tutorial presents a new way to obtain dynamically changing timbres, using a technique known as *waveshaping*.

Table lookup: lookup~

In *waveshaping synthesis* an audio signal—most commonly a sine wave—is used to access a *lookup table* containing some shaping function (also commonly called a *transfer function*). Each sample value of the input signal is used as an index to look up a value stored in a table (an array of numbers). Because a lookup table may contain any values in any order, it is useful for mapping a linear range of values (such as the signal range -1 to 1) to a nonlinear function (whatever is stored in the lookup table). The Max object **table** is an example of a lookup table; the number received as input (commonly in the range 0 to 127) is used to access whatever values are stored in the **table**.

The MSP object **lookup~** allows you to use samples stored in a **buffer~** as a lookup table which can be accessed by a signal in the range -1 to 1. By default, **lookup~** uses the first 512 samples in a **buffer~**, but you can type in arguments to specify any excerpt of the **buffer~** object's contents for use as a lookup table. If 512 samples are used, input values ranging from -1 to 0 are mapped to the first 256 samples, and input values from 0 to 1 are mapped to the next 256 samples; **lookup~** interpolates between two stored values as necessary.



Sine wave used to read back and forth through an excerpt of the buffer~

The most commonly used input signal for indexing the lookup table is a sine wave—it's a reasonable choice because it reads smoothly back and forth through the table—but any audio signal can be used as input to **lookup~**.

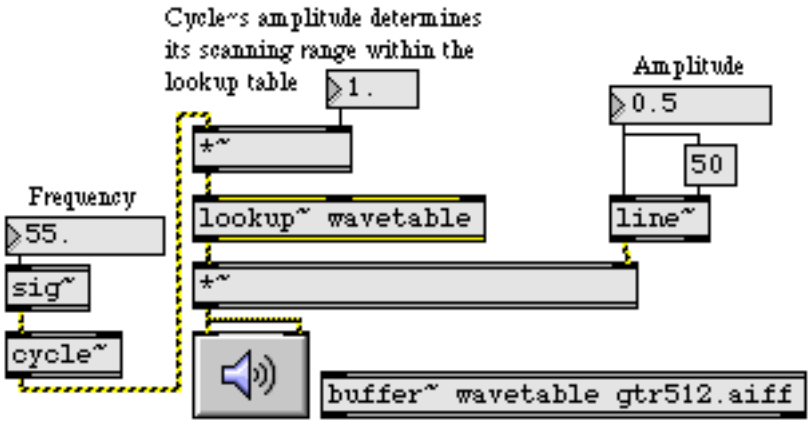
The important thing to observe about waveshaping synthesis is this: changing the amplitude of the input signal changes the amount of the lookup table that gets used. If the range of the input signal is from -1 to 1, the entire lookup table is used. However, if the range of the input signal is from -0.33 to 0.33, only the middle third of the table is used. As a general rule, the timbre of the output signal becomes brighter (contains more high frequencies) as the amplitude of the input signal increases.

It's also worth noting that the amplitude of the input signal has no direct effect on the amplitude of the output signal; the output amplitude depends entirely on the values being indexed in the lookup table.

Varying timbre with waveshaping

The waveshaping part of the tutorial patch is in the lower left portion of the Patcher window. It's very similar to the example shown above.

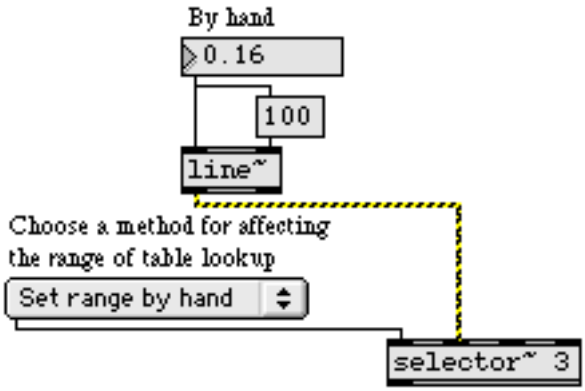
The lookup table consists of the 512 samples in the **buffer~**, and it is read by a cosine wave from a **cycle~** object.



Lookup table used for waveshaping

The upper portion of the Patcher window contains three different ways to vary the amplitude of the cosine wave, which will vary the timbre.

- With the audio still on, choose "Set range by hand" from the pop-up **umenu**. This opens the first signal inlet of the **selector~**, so you can alter the amplitude of the **cycle~** by dragging in the **number box** marked "By hand". Change the value in the **number box** to hear different timbres.



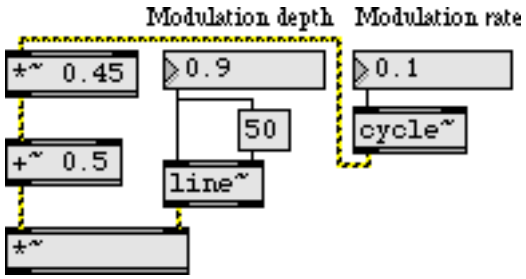
Set the amplitude of the input signal to change the timbre of the output

To make the timbre change over the course of the note, you can use a control function envelope to vary the amplitude of the **cycle~** automatically over time.

- Choose "Control range by envelope" from the **umenu**. Set a note duration by typing a value into the **number box** marked "Duration" (such as 1000 ms), then click on the **button** to play a note. Experiment with different durations and envelopes.

You can also modulate the amplitude of the input wave with another signal. An extremely slow modulating frequency (such as 0.1 Hz) will change the timbre very gradually. A faster sub-audio modulating frequency (such as 8 Hz) will create a unique sort of "timbre tremolo". Modulating the input wave at an audio rate creates sum and difference frequencies (as you have seen in *Tutorial 9*) which may interfere in various ways depending on the modulation rate.

- Choose "Modulate range by wave" from the **umenu**. Set the modulation rate to 0.1 Hz and set the modulation depth to 0.9.



Very slow modulation of the input wave's amplitude creates a gradual timbre change

Notice that the amplitude of the **cycle~** is multiplied by 0.45 and offset by 0.5. That makes it range from 0.05 to 0.95. (If it went completely to 0 the amplitude of the wave it's modulating would be 0 and the sound would stop.) The "Modulation depth" number box goes from 0 to 1, but it's actually scaling the **cycle~** within that range from 0.05 to 0.95.

- Experiment with other values for the depth and rate of modulation.

If you're designing an instrument for musical purposes, you might use some combination of these three ways to vary the timbre, and you almost certainly would have an independent amplitude envelope to scale the amplitude of the output sound. (Remember that the amplitude of the signal coming out of **lookup**~ depends on the sample values being read, and is not directly affected by the amplitude of the signal coming into it.)

Summary

Waveshaping is the nonlinear distortion of a signal to create a new timbre. The sample values of the original signal are used to address a lookup table, and the corresponding value from the lookup table is sent out. The **lookup**~ object treats samples from a **buffer~** as such a lookup table, and uses the input range -1 to 1 to address those samples. A sine wave is commonly used as the input signal for waveshaping synthesis. The amplitude of

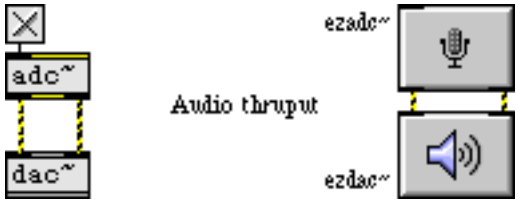
the input signal determines how much of the lookup table gets used. As the amplitude of the input signal increases, more of the table gets used, and consequently more frequencies are generally introduced into the output. Thus, you can change the timbre of a waveshaped signal dynamically by continuously altering the amplitude of the input signal, using a control function or a modulating signal.

See Also

buffer~	Store audio samples
cycle~	Table lookup oscillator
lookup~	Transfer function lookup table

Sound input: adc~

For getting sound from the "real world" into MSP, there is an analog-to-digital conversion object called **adc~**. It recognizes all the same messages as the **dac~** object, but instead of sending signal to the audio output jacks of the computer, **adc~** receives signal from the audio input jacks, and sends the incoming signal out its outlets. Just as **dac~** has a user interface version called **ezdac~**, there is an iconic version of **adc~** called **ezadc~**.

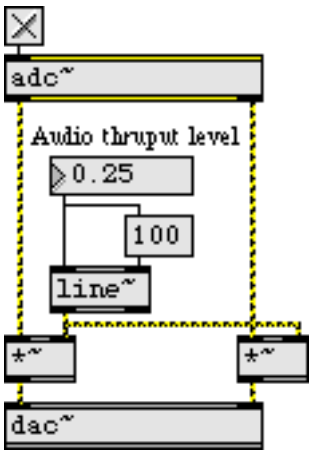


adc~ and ezadc~ get sound from the audio input jacks and send it out as a signal

To use the **adc~** object, you need to send sound from some source into the computer. The sound may come from the CD player of your computer, from any line level source such as a tape player, or from a microphone—your computer might have a built-in microphone, or you can use a standard microphone via a preamplifer..

- Double click on the **adc~** object to open the DSP Status window. Make sure that the *Input Source* popup menu displays the input device you want. Depending on your computer system, audio card and driver, you may not have a choice of input device-this is nothing to be concerned about.

- Click on the toggle above the **adc~** object to turn audio on. If you want to hear the input sound played directly out the output jacks, adjust the **number box** marked *Audio thruput level*.

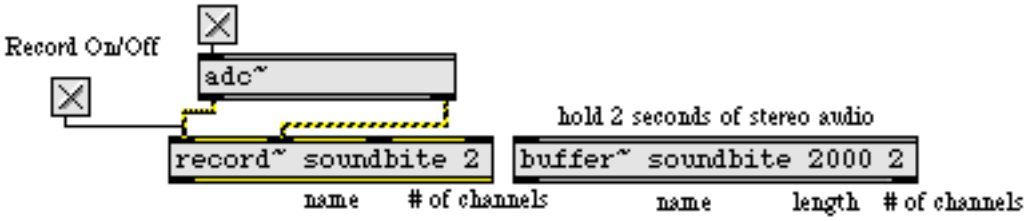


Adjust the audio throughput to a comfortable listening level

If your input source is a microphone, you'll need to be careful not to let the output sound from your computer feed back into the microphone.

Recording a sound: record~

To record a sample of the incoming sound (or any signal), you first need to designate a buffer in which the sound will be stored. Your patch should therefore include at least one **buffer~** object. You also need a **record~** object with the same name as the **buffer~**. The sound that you want to record must go in the inlet of the **record~** object.

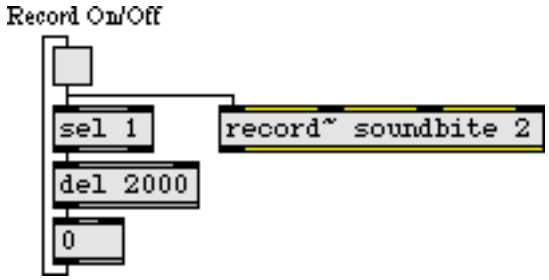


Record two seconds of stereo sound into the buffer~ named soundbite

When **record~** receives a non-zero int in its left inlet, it begins recording the signals connected to its record inlets; 0 stops the recording. You can specify recording start and end points within the **buffer~** by sending numbers in the two right inlets of **record~**. If you don't specify start and end points, recording will fill the entire **buffer~**. Notice that the

length of the recording is limited by the length of the **buffer~**. If this were not the case, there would be the risk that **record~** might be left on accidentally and fill the entire application memory.

In the tutorial patch, **record~** will stop recording after 2 seconds (2000 ms). We have included a delayed bang to turn off the **toggle** after two seconds, but this is just to make the **toggle** accurately display the state of **record~**. It is not necessary to stop **record~** explicitly, because it will stop automatically when it reaches its end point or the end of the **buffer~**.



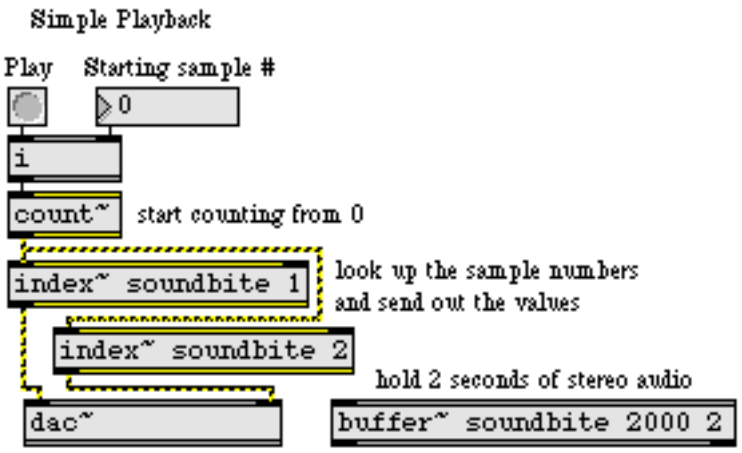
A delayed bang turns off the toggle after two seconds so it will display correctly

- Make sure that you have sound coming into the computer, then click on the **toggle** to record two seconds of the incoming sound. If you want to, you can double-click on the **buffer~** afterward to see the recorded signal.

Reading through a buffer~: index~

So far you have seen two ways to get sound into a **buffer~**. You can read in an existing audio file with the read message, and you can record sound into it with the **record~** object. Once you get the sound into a **buffer~**, there are several things you can do with it. You can save it to an audio file by sending the write message to the **buffer~**. You can use 513 samples of it as a wavetable for **cycle~**, as demonstrated in *Tutorial 3*. You can use any section of it as a transfer function for **lookup~**, as demonstrated in *Tutorial 12*. You can also just read straight through it to play it out the **dac~**. This tutorial patch demonstrates the two most basic ways to play the sound in a **buffer~**. A third way is demonstrated in *Tutorial 14*.

The **index~** object receives a signal as its input, which represents a sample number. It looks up that sample in its associated **buffer~**, and sends the value of that sample out its outlet as a signal. The **count~** object just sends out a signal value that increases by one with each sample. So, if you send the output of **count~**—a steady stream of increasing numbers—to the input of **index~**—which will treat them as sample numbers—**index~** will read straight through the **buffer~**, playing it back at the current sampling rate.



Play the sound in a buffer~ by looking up each sample and sending it to the dac~

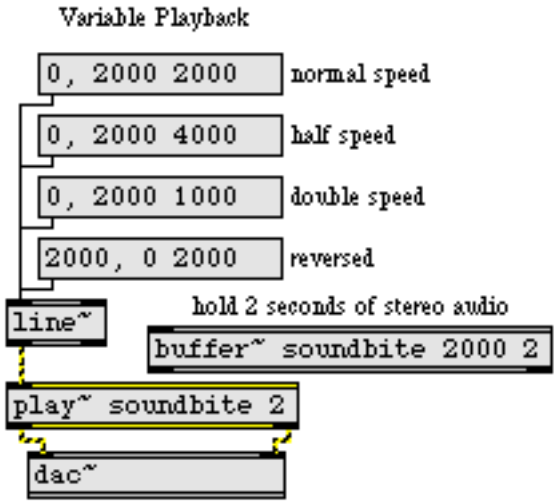
- Click on the **button** marked "Play" to play the sound in the **buffer~**. You can change the starting sample number by sending a different starting number into **count~**.

This combination of **count~** and **index~** lets you specify a precise sample number in the **buffer~** where you want to start playback. However, if you want to specify starting and ending points in the **buffer~** in terms of milliseconds, and/or you want to play the sound back at a different speed—or even backward—then the **play~** object is more appropriate.

Variable speed playback: play~

The **play~** object receives a signal in its inlet which indicates a position, in milliseconds, in its associated **buffer~**; **play~** sends out the signal value it finds at that point in the **buffer~**. Unlike **index~**, though, when **play~** receives a position that falls between two samples in the **buffer~** it interpolates between those two values. For this reason, you can read through a **buffer~** at any speed by sending an increasing or decreasing signal to **play~**, and it will interpolate between samples as necessary. (Theoretically, you could use **index~** in a similar manner, but it does not interpolate between samples so the sound fidelity would be considerably worse.)

The most obvious way to use the **play~** object is to send it a linearly increasing (or decreasing) signal from a **line~** object, as shown in the tutorial patch.



Read through a buffer~, from one position to another, in a given amount of time

Reading from 0 to 2000 (millisecond position in the **buffer~**) in a time of 2000 ms produces normal playback. Reading from 0 to 2000 in 4000 ms produces half-speed playback, and so on.

- Click on the different **message** box objects to hear the sound played in various speed/direction combinations. Turn audio off when you have finished.

Although not demonstrated in this tutorial patch, it's worth noting that you could use other signals as input to **play~** in order to achieve accelerations and decelerations, such as an exponential curve from a **curve~** object or even an appropriately scaled sinusoid from a **cycle~** object.

Summary

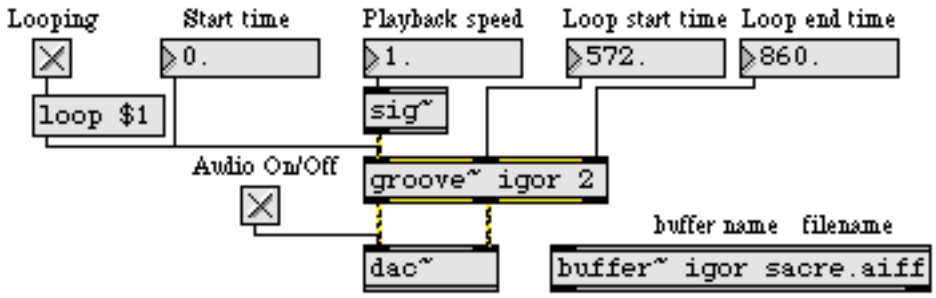
Sound coming into the computer enters MSP via the **adc~** object. The **record~** object stores the incoming sound—or any other signal—in a **buffer~**. You can record into the entire **buffer~**, or you can record into any portion of it by specifying start and end buffer positions in the two rightmost inlets of **record~**. For simple normal-speed playback of the sound in a **buffer~**, you can use the **count~** and **index~** objects to read through it at the current sampling rate. Use the **line~** and **play~** objects for variable-speed playback and/or for reading through the **buffer~** in both directions.

See Also

adc~	Audio input and on/off
ezadc~	Audio on/off; analog-to-digital converter
index~	Sample playback without interpolation
play~	Position-based sample playback
record~	Record sound into a buffer

Playing samples with groove~

The **groove~** object is the most versatile object for playing sound from a **buffer~**. You can specify the **buffer~** to read, the starting point, the playback speed (either forward or backward), and starting and ending points for a repeating loop within the sample. As with other objects that read from a **buffer~**, **groove~** accesses the **buffer~** remotely, without patch cords, by sharing its name.

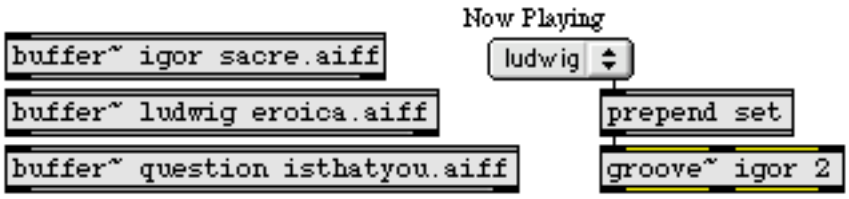


A standard configuration for the use of groove~

In the example above, the message loop 1 turns looping on, the start time of 0 ms indicates the beginning of the **buffer~**, the playback speed of 1 means to play forward at normal speed, and the loop start and end times mean that (because looping is turned on) when **groove~** reaches a point 860 milliseconds into the **buffer~** it will return to a point 572 ms into the **buffer~** and continue playing from there. Notice that the start time must be received as a float (or int), and the playback speed must be received as a signal. This means the speed can be varied continuously by sending a time- varying signal in the left inlet.

Whenever a new start time is received, **groove~** goes immediately to that time in the **buffer~** and continues playing from there at the current speed. When **groove~** receives the message loop 1 or startloop it goes to the beginning of the loop and begins playing at the current speed. (Note that loop points are ignored when **groove~** is playing in reverse, so this does not work when the playback speed is negative.) **groove~** stops when it reaches the end of the **buffer~** (or the beginning if it's playing backward), or when it receives a speed of 0.

In the tutorial patch, three different **buffer~** objects are loaded with AIFF files so that a single **groove~** object can switch between various samples instantly. The message set, followed by the name of a **buffer~**, refers **groove~** to that new **buffer~** immediately. (If **groove~** always referred to the same **buffer~**, and we used read messages to change the contents of the **buffer~**, some time would be needed to open and load each new file.)

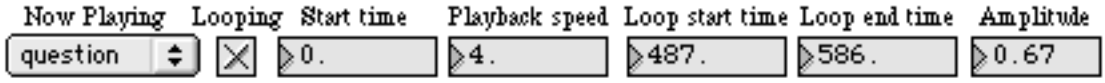


Refer groove~ to a different buffer~ with a set message

- Click on the **preset** object to play the samples in different ways.

The first preset just functions as an "Off" button. The next three presets play the three **buffer~** objects at normal speed without looping. The rest of the presets demonstrate a variety of sound possibilities using different playback speeds on different excerpts of the buffered files, with or without looping.

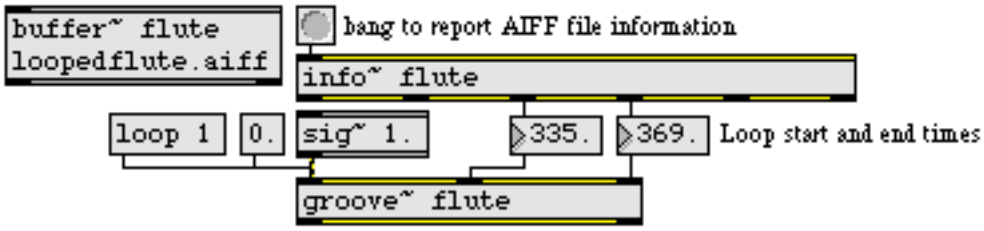
- You may want to experiment with your own settings by changing the user interface objects directly.



You can control all aspects of the playback by changing the user interface object settings

If you want to create smooth undetectable loops with **groove~**, you can use the loopinterp message to enable crossfades between the end of a loop and the beginning of the next pass through the loop to smooth out the transition back to the start point (see the **groove~** reference page for more information on this message). If the **buffer~** contains an AIFF file that has its own loop points—points established in a separate audio editing program—there is a way to use those loop points to set the loop points of **groove~**.

The **info~** object can report the loop points of an AIFF file contained in a **buffer~**, and you can send those loop start and end times directly into **groove~**.



Using info~ to get loop point information from an AIFF file

Summary

The **groove~** object is the most versatile way to play sound from a **buffer~**. You can specify the **buffer~** to read, the starting point, the playback speed (either forward or backward), and starting and ending points for a repeating loop within the sample. If the **buffer~** contains an AIFF file that has its own pre-established loop points, you can use the **info~** object to get those loop times and send them to **groove~**. The playback speed of **groove~** is determined by the value of the signal coming in its left inlet. You can set the current buffer position of **groove~** by sending a float time value in the left inlet.

See Also

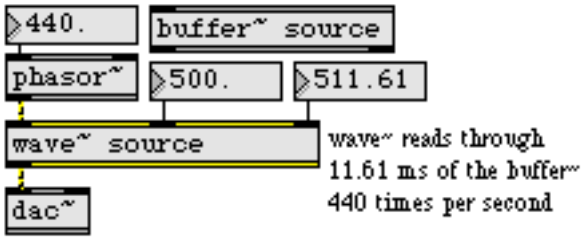
buffer~	Store audio samples
groove~	Variable-rate looping sample playback
sig~	Constant signal of a number

Tutorial 15: Sampling—Variable-length wavetable

Use any part of a buffer~ as a wavetable: wave~

As was shown in *Tutorial 3*, the **cycle~** object can use 512 samples of a **buffer~** as a wavetable through which it reads repeatedly to play a periodically repeating tone. The **wave~** object is an extension of that idea; it allows you to use *any* section of a **buffer~** as a wavetable.

The starting and ending points within the **buffer~** are determined by the number or signal received in the middle and right inlets of **wave~**. As a signal in the **wave~** object's left inlet goes from 0 to 1, **wave~** sends out the contents of the **buffer~** from the specified start point to the end point. The **phasor~** object, ramping repeatedly from 0 to 1, is the obvious choice as an input signal for the left inlet of **wave~**.



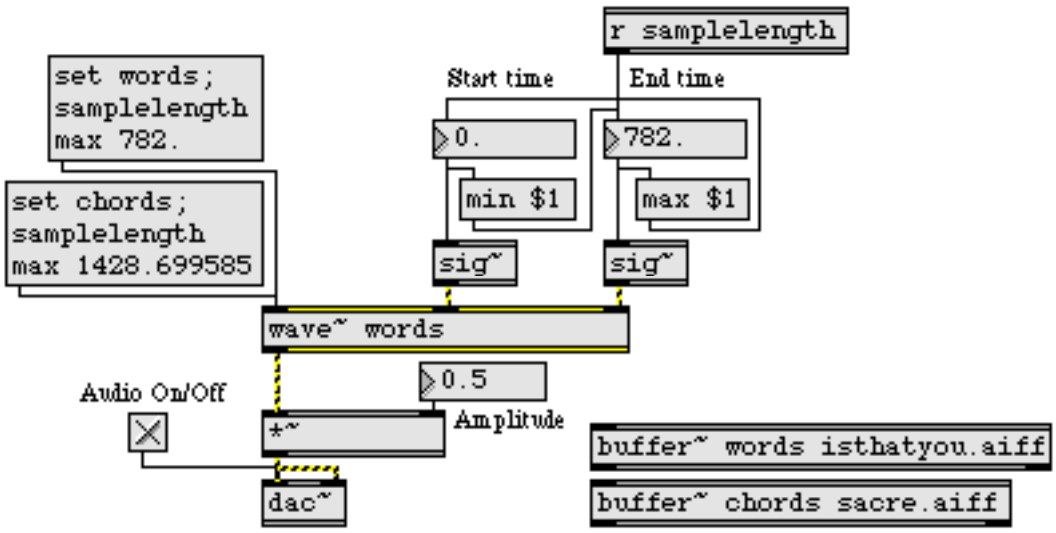
phasor~ drives wave~ through the section of the buffer~ specified as the wavetable

In a standard implementation of wavetable synthesis, the wavetable (512 samples in the case of **cycle~**, or a section of any length in the case of **wave~**) would be one single cycle of a waveform, and the frequency of the **cycle~** object (or the **phasor~** driving the **wave~**) would determine the fundamental frequency of the tone. In the case of **wave~**, however, the wavetable could contain virtually anything (an entire spoken sentence, for example).

wave~ yields rather unpredictable results compared to some of the more traditional sound generation ideas presented so far, but with some experimentation you can find a great variety of timbres using **wave~**. In this tutorial patch, you will see some ways of reading the contents of a **buffer~** with **wave~**.

Synthesis with a segment of sampled sound

The tutorial patch is designed to let you try three different ways of driving **wave~**: with a repeating ramp signal (**phasor~**), a sinusoid (**cycle~**), or a single ramp (**line~**). The bottom part of the Patcher window is devoted to the basic implementation of **wave~**, and the upper part of the window contains the three methods of reading through the wavetable. First, let's look at the bottom half of the window.

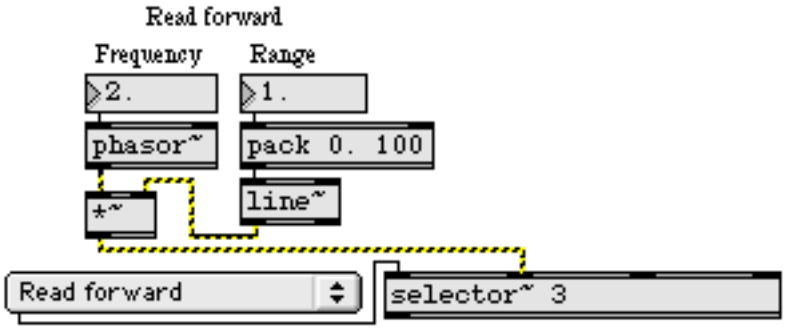


wave~ can use an excerpt of any length from either buffer~ as its wavetable

- Click on the **toggle** to turn audio on. Set the amplitude to some level greater than 0. Set the end time of the wavetable to 782 (the length in milliseconds of the file *isthatyou.aiff*).

With these settings, **wave~** will use the entire contents of **buffer~** words isthatyou.aiff as its wavetable. Now we are ready to read through the wavetable.

- Choose "Read forward" from the pop-up **umenu** in the middle of the window. This will open the first signal inlet of the **selector~**, allowing **wave~** to be controlled by the **phasor~** object.



Read through wave~ by going repeatedly from 0 to 1 with a phasor~ object

- Set the **number box** marked "Range" to 1. This sets the amplitude of the **phasor~**, so it effectively determines what fraction of the wavetable will be used. Set the **number box**

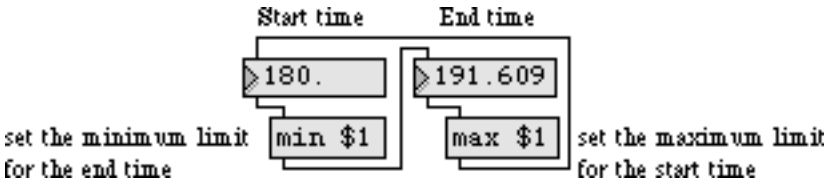
marked "Frequency" to 2. The **phasor~** now goes from 0 to 1 two times per second, so you should hear **wave~** reading through the **buffer~** every half second.

- Try a few different sub-audio frequency values for the **phasor~**, to read through the **buffer~** at different speeds. You can change the portion of the **buffer~** being read, either by changing the "Range" value, or by changing the start and end times of the **wave~**. Try audio frequencies for the **phasor~** as well.

Notice that the rate of the **phasor~** often has no obvious relationship to the perceived pitch, because the contents of the wavetable do not represent a single cycle of a waveform. Furthermore, such rapid repetition of an arbitrarily selected segment of a complex sample has a very high likelihood of producing frequencies well in excess of the Nyquist rate, which will be folded back into the audible range in unpredictable ways.

- Click on the **message** box to refer **wave~** to the **buffer~** chords object.

This changes the contents of the wavetable (because **wave~** now accesses a different **buffer~**), and sets the maximum value of the "End time" **number box** equal to the length of the file *sacre.aiff*. Notice an additional little programming trick—shown in the example below—employed to prevent the user from entering inappropriate start and end times for **wave~**.



Each time the start or end time is changed, it revises the limits of the other number box

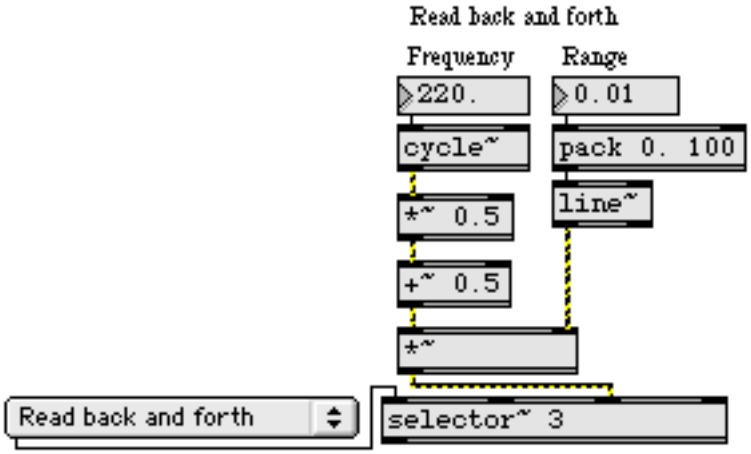
- With this new **buffer~**, experiment further by reading different length segments of the **buffer~** at various rates.

Using wave~ as a transfer function

The **buffer~** object is essentially a lookup table that can be accessed in different ways by other objects. In Tutorial 12 the **lookup~** object was used to treat a segment of a **buffer~** as a transfer function, with a cosine wave as its input. The **wave~** object can be used similarly. The only difference is that its input must range from 0 to 1, whereas **lookup~** expects input in the range from -1 to 1. To use **wave~** in this way, then, we must scale and offset the incoming cosine wave so that it ranges from 0 to 1.

- Set the start and end times of **wave~** close together, so that only a few milliseconds of sound are being used for the wavetable. Choose "Read back and forth" from the pop-

up **umenu** in the middle of the window. This opens the second signal inlet of the **selector~**, allowing **wave~** to be controlled by the **cycle~** object.



cycle~, scaled and offset to range from 0 to 1, reads back and forth in the wavetable

- Set the "Range" **number box** to a very small value such as 0.01 at first, to limit the **cycle~** object's amplitude. This way, **cycle~** will use a very small segment of the wavetable as the transfer function. Set the frequency of **cycle~** to 220 Hz. You will probably hear a rich tone with a fundamental frequency of 220 Hz. Drag on the "Range" **number box** to change the amplitude of the cosine wave; the timbre will change accordingly. You can also experiment with different wavetable lengths by changing the start and end times of **wave~**. Sub-audio frequencies for the **cycle~** object will produce unusual vibrato-like effects as it scans back and forth through the wavetable.

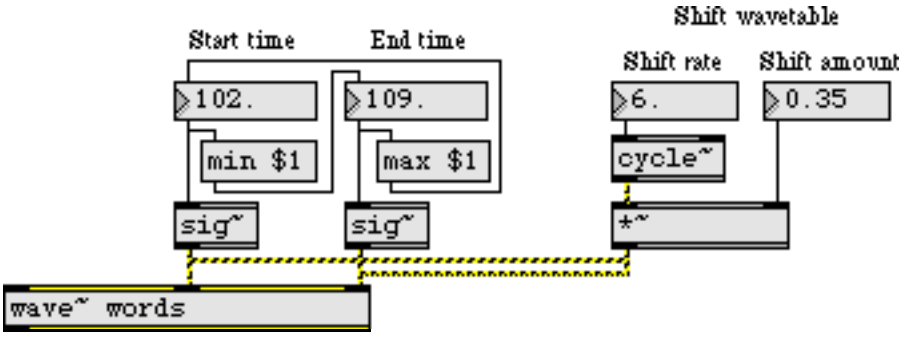
Play the segment as a note

Because **wave~** accepts any signal input in the rage 0 to 1, you can read through the wavetable just once by sending **wave~** a ramp signal from 0 to 1 (or backward, from 1 to 0). Other objects such as **play~** and **groove~** are better suited for this purpose, but it is nevertheless possible with **wave~**.

- Choose "Read once" from the pop-up **umenu** in the middle of the window. This opens the third signal inlet of the **selector~**, allowing **wave~** to be controlled by the **line~** object. Set start and end times for your wavetable, set the "Duration" **number box** to 1000, and click on the **button** to traverse the wavetable in one second. Experiment with both **buffer~** objects, using various wavetable lengths and durations.

Changing the wavetable dynamically

The **cycle~** object in the right part of the Patcher window is used to add a sinusoidal position change to the wavetable. As the cosine wave rises and falls, the start and end times of the wavetable increase and decrease. As a result, the wavetable is constantly shifting its position in the **buffer~**, in a sinusoidally varying manner. Sonically this produces a unique sort of vibrato, not of fundamental frequency but of timbre. The wavetable length and the rate at which it is being read stay the same, but the wavetable's contents are continually changing.



Shifting the wavetable by adding a sinusoidal offset to the start and end times

- Set the "Shift amount" to 0.35, and set the "Shift rate" to 6. Set the start time of the wavetable to 102 and the end time to 109. Click on the **message** box to refer **wave~** to the **buffer~** chords object. Choose "Read forward" from the pop-up **umenu**. Set the frequency of the **phasor~** to an audio rate such as 110, and set its range to 1. You should hear a vibrato-like timbre change at the rate of 6 Hz. Experiment with varying the shift rate and the shift amount. When you are done, click on the **toggle** to turn audio off.

Summary

Any segment of the contents of a **buffer~** can be used as a wavetable for the **wave~** object. You can read through the wavetable by sending a signal to **wave~** that goes from 0 to 1. So, by connecting the output of a **phasor~** object to the input of **wave~**, you can read through the wavetable repeatedly at a sub-audio or audio rate. You can also scale and offset the output of a **cycle~** object so that it is in the range 0 to 1, and use that as input to **wave~**. This treats the wavetable as a transfer function, and results in waveshaping synthesis. The position of the wavetable in the **buffer~** can be varied dynamically—by adding a sinusoidal offset to the start and end times of **wave~**, for example—resulting in unique sorts of timbre modulation.

See Also

buffer~	Store audio samples
phasor~	Sawtooth wave generator
wave~	Variable-size wavetable

Tutorial 16: Sampling—Record and play audio files

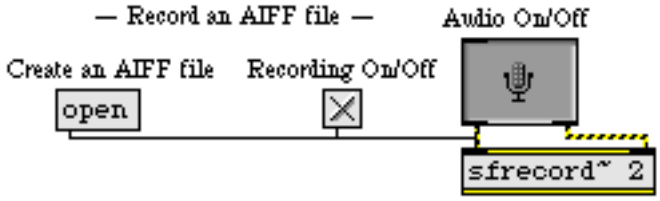
Playing from memory vs. playing from disk

You have already seen how to store sound in memory—in a **buffer~**—by recording into it directly or by reading in a pre-recorded audio file. Once the sound is in memory, it can be accessed in a variety of ways with **cycle~**, **lookup~**, **index~**, **play~**, **groove~**, **wave~**, etc.

The main limitation of **buffer~** for storing samples, of course, is the amount of unused RAM available to the Max application. You can only store as much sound in memory as you have memory to hold it. For playing and recording very large amounts of audio data, it is more reasonable to use the hard disk for storage. But it takes more time to access the hard disk than to access RAM; therefore, even when playing from the hard disk, MSP still needs to create a small buffer to preload some of the sound into memory. That way, MSP can play the preloaded sound *while* it is getting more sound from the hard disk, without undue delay or discontinuities due to the time needed to access the disk.

Record audio files: sfrecord~

MSP has objects for recording directly into, and playing directly from, an AIFF file: **sfrecord~** and **sfplay~**. Recording an audio file is particularly easy, you just open a file, begin recording, and stop recording. (You don't even need to close the file; **sfrecord~** takes care of that for you.) In the upper right corner of the Patcher window there is a patch for recording files.



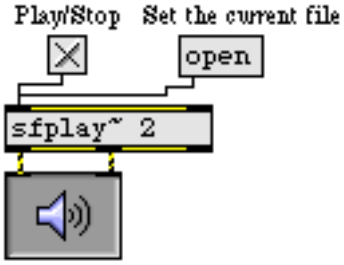
Recording audio into an audio file on disk

sfrecord~ records to disk whatever signal data it receives in its inlets. The signal data can come directly from an **adc~** or **ezadc~** object, or from any other MSP object.

- Click on the **message** box marked "Create an AIFF file". You will be shown a Save As dialog box for naming your file. (Make sure you save the file on a volume with sufficient free space.) Navigate to the folder where you want to store the sound, name the file, and click Save. Turn audio on. Click on the **toggle** to begin recording; click on it again when you have finished.

Play audio files: sfplay~

In the left part of the Patcher window there is a patch for playing audio files. The basic usage of **sfplay~** requires only a few objects, as shown in the following example. To play a file, you just have to open it and start **sfplay~**. The audio output of **sfplay~** can be sent directly to **dac~** or **ezdac~**, and/or anywhere else in MSP.



Simple implementation of audio file playback

- Click on the open **message** box marked "Set the current file", and open the audio file you have just recorded. Then (with audio on) click on the **toggle** marked "Play/Stop" to hear your file.

Play excerpts on cue

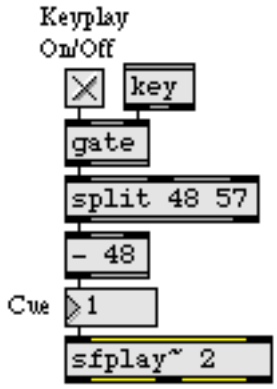
Because **sfplay~** does not need to load an entire audio file into memory, you can actually have many files open in the same **sfplay~** object, and play any of them (or any portion of them) on cue. The most recently opened file is considered by **sfplay~** to be the "current" file, and that is the file it will play when it receives the message 1.

- Click on the remaining open **message** boxes to open some other audio files, and then click on the **message** box marked "Define cues, 2 to 9".

The preload message to **sfplay~** specifies an entire file or a portion of a file, and assigns it a *cue number*. From then on, every time **sfplay~** receives that number, it will play that cue. In the example patch, cues 2, 3, and 4 play entire files, cue 5 plays the first 270 milliseconds of *sacre.aiff,* and so on. Cue 1 is always reserved for playing the current (most recently opened) file, and cue 0 is reserved for stopping **sfplay~**.

Whenever **sfplay~** receives a cue, it stops whatever it is playing and immediately plays the new cue. (You can also send **sfplay~** a *queue of cues*, by sending it a list of numbers, and it will play each cue in succession.) Each preload message actually creates a small buffer containing the audio data for the beginning of the cue, so playback can start immediately upon receipt of the cue number.

Now that cues 0 through 9 are defined, you can play different audio excerpts by sending **sfplay~** those numbers. The upper-left portion of the patch permits you to type those numbers directly from the computer keyboard.



ASCII codes from the number keys used to send cues to sfplay~

- Click on the toggle marked "Keyplay On/Off". Type number keys to play the different pre- defined cues. Turn "Keyplay" off when you are done.

Try different file excerpts

Before you define a cue, you will probably need to listen to segments of the file to determine the precise start and end times you want. You can use the seek message to hear any segment of the current file.

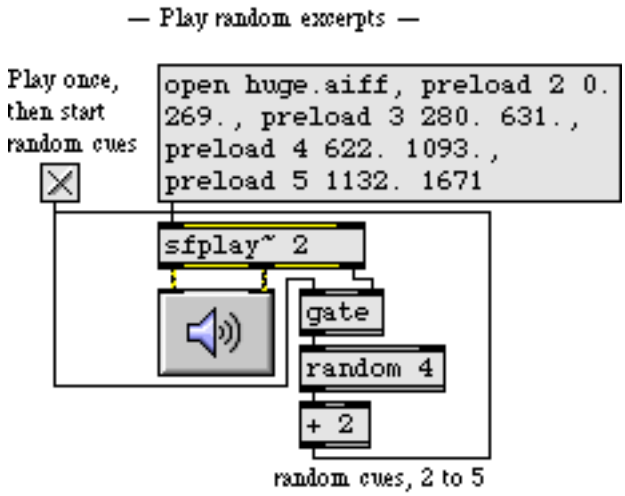
- Open your own audio file again (or any other audio file) to make it the current file. In the right portion of this patch, enter an end time for the seek message. The excerpt you have specified will begin playing. Try different start and end times.

Once you find start and end times you like, you could use them in a preload message to establish a cue. Because **sfplay~** can't know in advance what excerpt it will be required to play in response to a seek message, it can't preload the excerpt. There will be a slight delay while it accesses the hard disk before it begins playing. For that reason, seek is best used as an auditioning tool; preloaded cues are better for performance situations where immediate playback is more critical.

Trigger an event at the end of a file

The patch in the lower right portion of the Patcher window demonstrates the use of the right outlet of **sfplay~**. When a cue is done playing (or when it is stopped with a 0 message), **sfplay~** sends a bang out the right outlet. In this example patch, the bang is used

to trigger the next (randomly chosen) cue, so **sfplay~** effectively restarts itself when each cue is done.



When a cue is completed, sfplay~ triggers the next cue

Note the importance of the **gate** object in this patch. If it were not present, there would be no way to stop **sfplay~** because each 0 cue would trigger another non-zero cue. The **gate** must be closed before the 0 cue is sent to **sfplay~**.

- In the patch marked "Play random excerpts", click on the **message** box to preload the cues, then click on the **toggle** to start the process. To stop it, click on the **toggle** again. Turn audio off.

Summary

For large and/or numerous audio samples, it is often better to read the samples from the hard disk than to try to load them all into RAM. The objects **sfrecord~** and **sfplay~** provide a simple way to record and play audio files to and from the hard disk. The **sfplay~** object can have many audio files open at once. Using the preload message, you can pre-define ready cues for playing specific files or sections of files. The seek message to **sfplay~** lets you try different start and end points for a cue. When a cue is done playing (or is stopped) **sfplay~** sends a bang out its right outlet. This bang can be used to trigger other processes, including sending **sfplay~** its next cue.

See Also

sfplay~	Play audio file from disk
sfrecord~	Record to audio file on disk

A sampling exercise

In this chapter we suggest an exercise to help you check your understanding of how to sample and play audio. Try completing this exercise in a new file of your own before you check the solution given in the example patch. (But don't have the example Patcher open while you design your own patch, or you will hear both patches when you turn audio on.) The exercise is to design a patch in which:

- Typing the *a* key on the computer keyboard turns audio on. Typing *a* again toggles audio off.
- Typing *r* on the computer keyboard makes a one-second recording of whatever audio is coming into the computer (from the input jacks or from the internal CD player).
- Typing *p* plays the recording. Playback is to be at half speed, so that the sound lasts two seconds.
- An amplitude envelope is applied to the sample when it is played, tapering the amplitude slightly at the beginning and end so that there are no sudden clicks heard at either end of the sample.
- The sample is played back with a 3 Hz vibrato added to it. The depth of the vibrato is one semitone (a factor of $2^{\pm 1/12}$) up and down.

Hints

You will need to store the sound in a **buffer~** and play it back from memory.

You can record directly into the **buffer~** with **record~**. (See *Tutorial 13*.) The input to **record~** will come from **adc~** (or **ezadc~**).

The two obvious choices for playing a sample from a **buffer~** at half speed are **play~** and **groove~**. However, because we want to add vibrato to the sound—by continuously varying the playback speed—the better choice is **groove~**, which uses a (possibly time-varying) signal to control its playback speed directly. (See *Tutorial 14*.)

The amplitude envelope is best generated by a **line~** object which is sending its output to a ***~** object to scale the amplitude of the output signal (coming from **groove~**). You might want to use a **function** object to draw the envelope, and send its output to **line~** to describe the envelope. (See *Tutorial 7*.)

Tutorial 17

Sampling review

The computer keyboard will need to trigger messages to the objects **adc~**, **record~**, **groove~**, and **line~** (or **function**) in order to perform the required tasks. Use the **key** object to get the keystrokes, and use **select** to detect the keys you want to use.

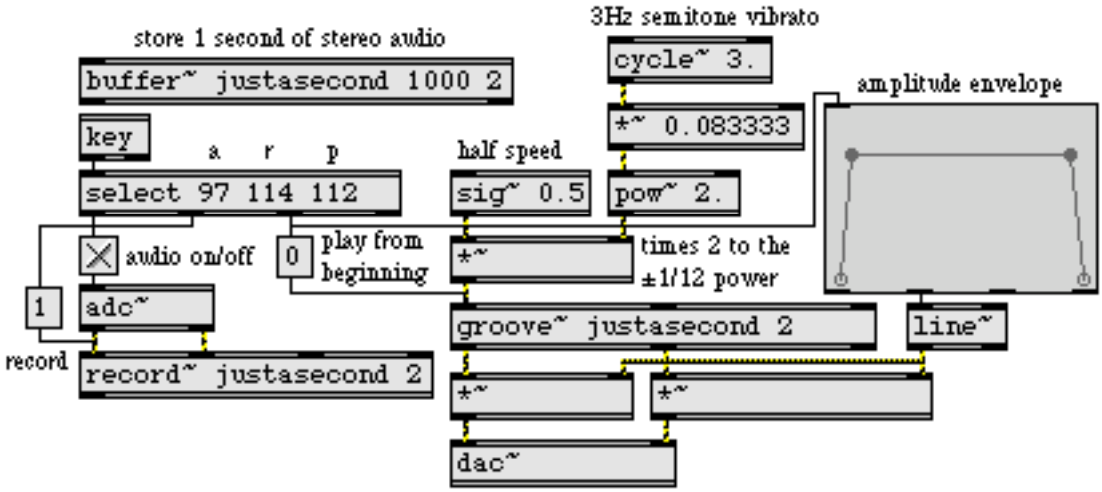
Use a sinusoidal wave from a **cycle~** object to apply vibrato to the sample. The frequency of the **cycle~** will determine the *rate* of the vibrato, and the amplitude of the sinusoid will determine the *depth* of vibrato. Therefore, you will need to scale the **cycle~** object's amplitude with a ***~** object to achieve the proper vibrato depth.

In the discussion of vibrato in *Tutorial 10*, we created vibrato by adding the output of the modulating oscillator to the frequency input of the carrier oscillator. However, two things are different in this exercise. First of all, the modulating oscillator needs to modulate the playback speed of **groove~** rather than the frequency of another **cycle~** object. Second, adding the output of the modulator to the input of the carrier—as in *Tutorial 10*—creates a vibrato of equal *frequency* above and below the carrier frequency, but does not create a vibrato of equal *pitch* up and down (as required in this exercise). A change in pitch is achieved by *multiplying* the carrier frequency by a certain amount, rather than by adding an amount to it.

To raise the pitch of a tone by one semitone, you must multiply its frequency by the twelfth root of 2, which is a factor of 2 to the $^{1}/_{12}$ power (approximately 1.06). To lower the pitch of a tone by one semitone, you must multiply its frequency by 2 to the $^{-1}/_{12}$ power (approximately 0.944). To calculate a signal value that changes continuously within this range, you may need to use an MSP object not yet discussed, **pow~**. Consult its description in the Objects section of this manual for details.

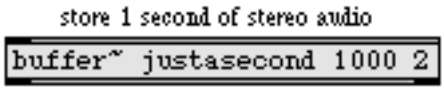
Solution

- Scroll the example Patcher window all the way to the right to see a solution to this exercise.



Solution to the exercise for recording and playing an audio sample

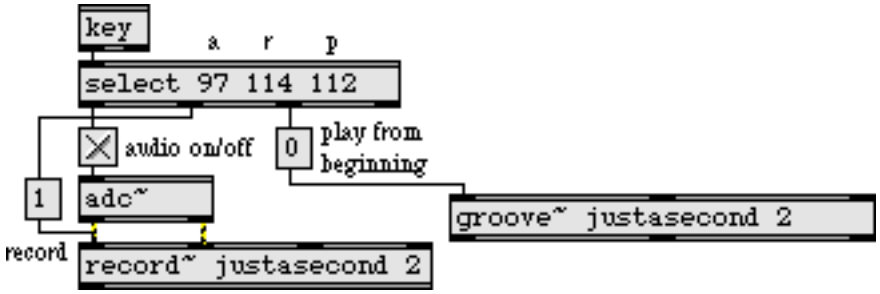
The arguments to the **buffer~** object specify a length in milliseconds (1000) and a number of channels (2). This determines how much memory will initially be allocated to the **buffer~**.



Set name, length, and channels of the buffer~

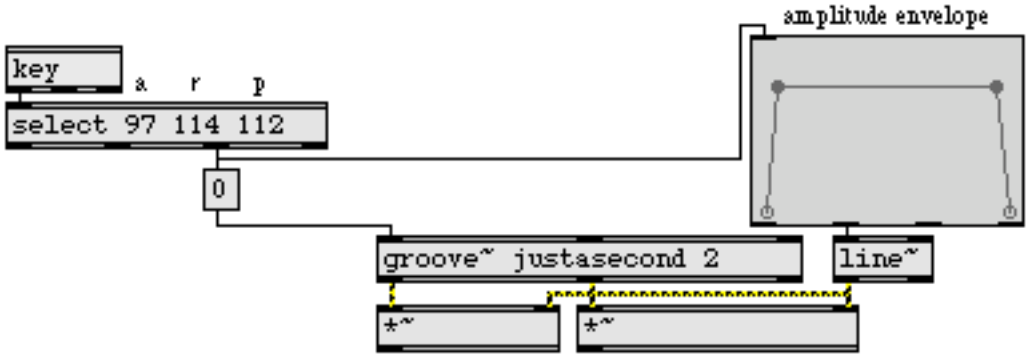
Since the memory allocated in the **buffer~** is limited to one second, there is no need to tell the **record~** object to stop when you record into the **buffer~**. It stops when it reaches the end of the **buffer~**.

The keystrokes from the computer keyboard are reported by **key**, and the **select** object is used to detect the *a*, *r*, and *p* keys. The bangs from **select** trigger the necessary messages to **adc~**, **record~**, and **groove~**.



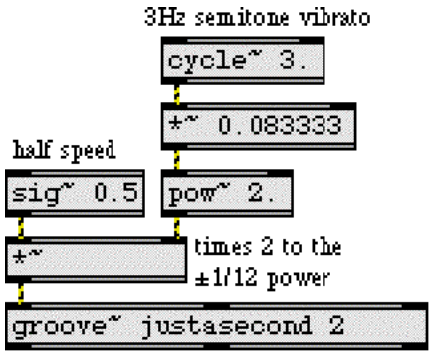
Keystrokes are detected and used to send messages to MSP objects

The keystroke *p* is also used to trigger the amplitude envelope at the same time as the sample is played. This envelope is used to scale the output of **groove~**.



A two-second envelope tapers the amplitude at the beginning and end of the sample

A **sig~ 0.5** object sets the basic playback speed of **groove~** at half speed. The amplitude of a 3 Hz cosine wave is scaled by a factor of 0.083333 (equal to $^{1}/_{12}$, but more computationally efficient than dividing by 12) so that it varies from $^{-1}/_{12}$ to $^{1}/_{12}$. This sinusoidal signal is used as the exponent in a power function in **pow~** (2 to the power of the input), and the result is used as the factor by which to multiply the playback speed.



Play at half speed, ± one semitone

Tutorial 18: MIDI control—Mapping MIDI to MSP

MIDI range vs. MSP range

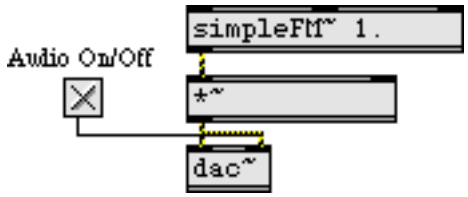
One of the greatest assets of MSP is the ease with which one can combine MIDI and digital signal processing. The great variety of available MIDI controllers means that you have many choices for the instrument you want to use to control sounds in MSP. Because Max is already a well developed environment for MIDI programming, and because MSP is so fully integrated into that environment, it is not difficult to use MIDI to control parameters in MSP.

The main challenge in designing programs that use MIDI to control MSP is to reconcile the numerical ranges needed for the two types of data. MIDI data bytes are exclusively integers in the range 0 to 127. For that reason, most numerical processing in Max is done with integers and most Max objects (especially user interface objects) deal primarily with integers. In MSP, on the other hand, audio signal values are most commonly decimal numbers between -1.0 and 1.0, and many other values (such as frequencies, for example) require a wide range and precision to several decimal places. Therefore, almost all numerical processing in MSP is done with floating-point (decimal) numbers.

Often this "incompatibility" can be easily reconciled by linear mapping of one range of values (such as MIDI data values 0 to 127) into another range (such as 0 to 1 expected in the inlets of many MSP objects). Linear mapping is explained in *Tutorial 29* of the Tutorials and Topics manual from the Max documentation, and is reviewed in this chapter. In many other cases, however, you may need to map the linear numerical range of a MIDI data byte to some nonlinear aspect of human perception—such as our perception of a 12-semitone increase in pitch as a power of 2 increase in frequency, etc. This requires other types of mapping; some examples are explored in this tutorial chapter.

Controlling synthesis parameters with MIDI

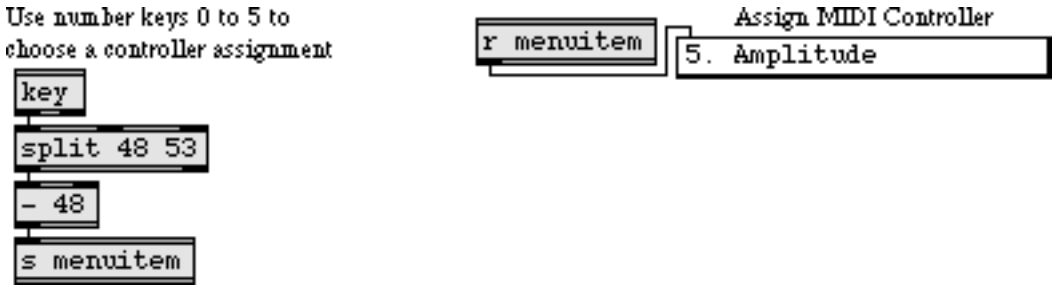
In this tutorial patch, we use MIDI continuous controller messages to control several different parameters in an FM synthesis patch. The synthesis is performed in MSP by the subpatch **simpleFM~** which was introduced in *Tutorial 11*, and we map MIDI controller 1 (the mod wheel) to affect, in turn, its amplitude, modulation index, vibrato depth, vibrato rate, and pitch bend.



An FM synthesis subpatch is the sound generator to be modified by MIDI

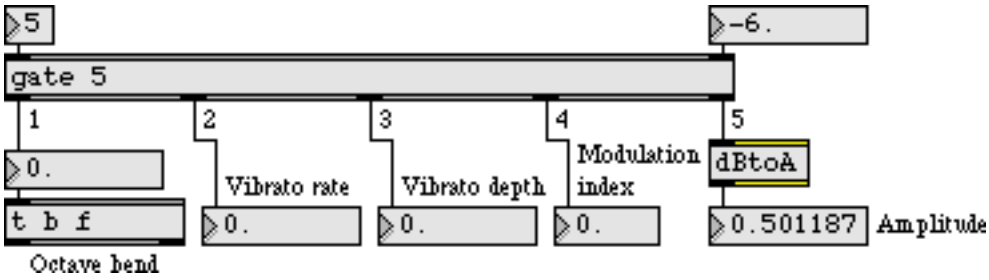
If we were designing a real performance instrument, we would probably control each of these parameters with a separate type of MIDI message—controller 7 for amplitude, controller 1 for vibrato depth, pitchbend for pitch bend, and so on. In this patch, however, we use the mod wheel controller for everything, to ensure that the patch will work for almost any MIDI keyboard. While this patch is not a model of good synthesizer design, it does let you isolate each parameter and control it with the mod wheel.

In the lower right corner of the Patcher window, you can see that keys 0 to 5 of the computer keyboard can be used to choose an item in the pop-up **umenu** at the top of the window.



Use ASCII from the computer keyboard to assign the function of the MIDI controller

The **umenu** sends the chosen item number to **gate** to open one of its outlets, thus directing the controller values from the mod wheel to a specific place in the signal network.



gate directs the control messages to a specific place in the signal network

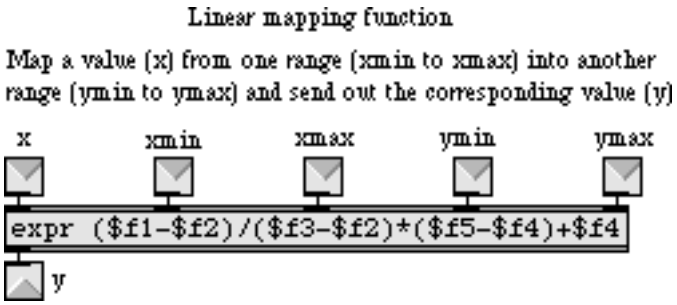
We will look at the special mapping requirements of each parameter individually. But first, let's review the formula for linear mapping.

Linear mapping

The problem of linear mapping is this: Given a value *x* which lies in a range from *xmin* to *xmax*, find the value *y* that occupies a comparable location in the range *ymin* to *ymax*. For example, 3 occupies a comparable location within the range 0 to 4 as 0.45 occupies within the range 0 to 0.6. This problem can be solved with the formula:

*y = ((x - xmin) * (ymax - ymin) ÷ (xmax - xmin)) + ymin*

For this tutorial, we designed a subpatch called **map** to solve the equation. **map** receives an *x* value in its left inlet, and—based on the values for *xmin*, *xmax*, *ymin*, and *ymax* received in its other inlets—it sends out the correct value for *y*. This equation will allow us to map the range of controller values—0 to 127—onto various other ranges needed for the signal network. The **map** subpatch appears in the upper right area of the Patcher window.



The contents of the map subpatch: the linear mapping formula expressed in an expr object

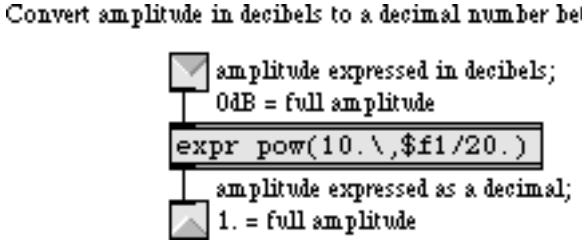
Once we have scaled the range of control values with **map**, some additional mapping may be necessary to suit various signal processing purposes, as you will see.

Mapping MIDI to amplitude

As noted in *Tutorial 4*, we perceive relative amplitude on a multiplicative rather than an additive scale. For example we hear the same relationship between amplitudes of 0.5 and 0.25 (a factor of $^1/_2$, but a difference of 0.25) as we do between amplitudes of 0.12 and 0.06 (again a factor of $^1/_2$, but a difference of only 0.06). For this reason, if we want to express relative amplitude on a linear scale (using the MIDI values 0 to 127), it is more appropriate to use decibels.

- Click on the **toggle** to turn audio on. Type the number 5 (or choose "Amplitude" from the **umenu**) to direct the controller values to affect the output amplitude.

The item number chosen in the **umenu** also recalls a preset in the **preset** object, which provides range values to **map**. In this case, *ymin* is -80 and *ymax* is 0, so as the mod wheel goes from 0 to 127 the amplitude goes from -80 dB to 0 dB (full amplitude). The decibel values are converted to amplitude in the subpatch called **dBtoA**. This converts a straight line into the exponential curve necessary for a smooth increase in perceived loudness.



The contents of the dBtoA subpatch

- Move the mod wheel on your MIDI keyboard to change the amplitude of the tone. Set the amplitude to a comfortable listening level.

With this mapping, the amplitude changes by approximately a factor of 2 every time the controller value changes by 10. This permits the same amount of control at low amplitudes as at high amplitudes (which would not be the case with a straight linear mapping).

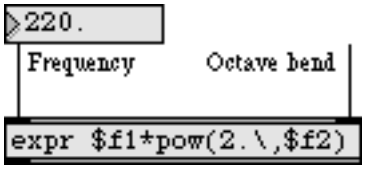
Mapping MIDI to frequency

Our perception of relative pitch is likewise multiplicative rather than additive with respect to frequency. In order for us to hear equal spacings of pitch, the frequency must change in

equal powers of 2. (See the discussions of pitch-to-frequency conversion in *Tutorial 17* and *Tutorial 19*.)

- Type the number 1 (or choose "Octave Pitch Bend" from the **umenu**) to direct the controller values to affect the carrier frequency. Move the mod wheel to bend the pitch upward as much as one octave, and back down to the original frequency.

In order for the mod wheel to perform a pitch bend of one octave, we map its range onto the range 0 to 1. This number is then used as the exponent in a power of 2 function and multiplied times the fundamental frequency in **expr**.



Octave bend factor ranges from 20 to 21

$2^0 = 1$, and $2^1 = 2$, so as the control value goes from 0 to 1 the carrier frequency increases from 220 to 440, which is to say up an octave. The increase in frequency from 220 to 440 follows an *exponential* curve, which produces a *linear* increase in perceived pitch from A to A.

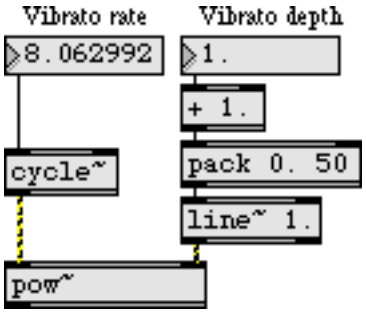
Mapping MIDI to modulation index

Mapping the MIDI controller to the modulation index of the FM instrument is much simpler, because a linear control is what's called for. Once the controller values are converted by the **map** subpatch, no further modification is needed. The mod wheel varies the modulation index from 0 (no modulation) to 24 (extreme modulation).

- Type the number 4 (or choose "Modulation Index" from the **umenu**) to direct the controller values to affect the modulation index. Move the mod wheel to change the timbre of the tone.

Mapping MIDI to vibrato

This instrument has an additional low-frequency oscillator (LFO) for adding vibrato to the tone by modulating the carrier frequency at a sub-audio rate. In order for the depth of the vibrato to be equal above and below the fundamental frequency, we use the output of the LFO as the exponent of a power function in **pow~**.



Calculate the vibrato factor

The base of the power function (controlled by the mod wheel) varies from 1 to 2. When the base is 1 there is no vibrato; when the base is 2 the vibrato is ± one octave.

- You'll need to set both the vibrato rate and the vibrato depth before hearing the vibrato effect. Type 2 and move the mod wheel to set a non-zero vibrato rate. Then type 3 and move the mod wheel to vary the depth of the vibrato.

The clumsiness of this process (re-assigning the mod wheel to each parameter in turn) emphasizes the need for separate MIDI controllers for different parameters (or perhaps linked simultaneous control of more than one parameter with the same MIDI message). In a truly responsive instrument, you would want to be able to control all of these parameters at once. The next chapter shows a more realistic assignment of MIDI to MSP.

Summary

MIDI messages can easily be used to control parameters in MSP instruments, provided that the MIDI data is mapped into the proper range. The **map** subpatch implements the linear mapping equation. When using MIDI to control parameters that affect frequency and amplitude in MSP, the linear range of MIDI data from 0 to 127 must be mapped to an exponential curve if you want to produce linear variation of perceived pitch and loudness. The **dBtoA** subpatch maps a linear range of decibels onto an exponential amplitude curve. The **pow~** object allows exponential calculations with signals.

Implementing standard MIDI messages

In this chapter we'll demonstrate how to implement MIDI control of a synthesis instrument built in MSP. The example instrument is a MIDI FM synthesizer with velocity sensitivity, pitch bend, and mod wheel control of timbre. To keep the example relatively simple, we use only a single type of FM sound (a single "patch", in synthesizer parlance), and only 2-voice polyphony.

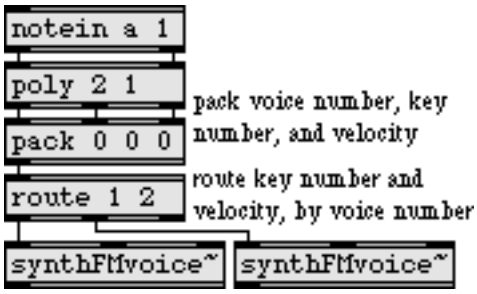
The main issues involved in MIDI control of an MSP synthesizer are

- converting a MIDI key number into the proper equivalent frequency
- converting a MIDI pitch bend value into an appropriate frequency-scaling factor
- converting a MIDI controller value into a modulator parameter (such as vibrator rate, vibrato depth, etc.).

Additionally, since a given MSP object can only play one note at a time, we will need to handle simultaneous MIDI note messages gracefully.

Polyphony

Each sound-generating object in MSP—an oscillator such as **cycle~** or **phasor~**, or a sample player such as **groove~** or **play~—**can only play one note at a time. Therefore, to play more than one note at a time in MSP you need to have more than one sound-generating object. In this tutorial patch, we make two identical copies of the basic synthesis signal network, and route MIDI note messages to one or the other of them. This 2-voice polyphony allows some overlap of consecutive notes, which normally occurs in legato keyboard performance of a melody.



Assign a voice number with poly to play polyphonic music

The **poly** object assigns a voice number—1 or 2 in this case—to each incoming note message, and if more than two keys are held down at a time **poly** provides note-off

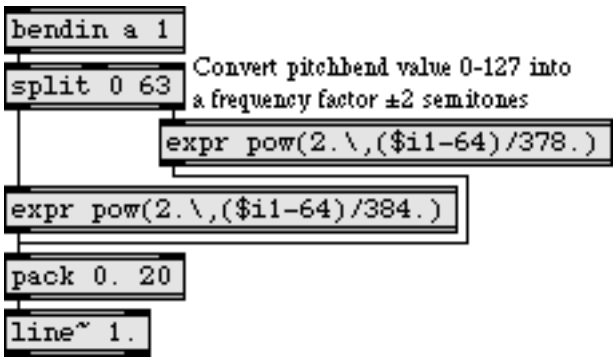
messages for the earlier notes so that the later notes can be played. The voice number, key number, and velocity are packed together in a three-item list, and the **route** object uses the voice number to send the key number and velocity to one synthesizer "voice" or the other. If your computer is fast enough, of course, you can design synthesizers with many more voices. You can test the capability of your computer by adding more and more voices and observing the CPU Utilization in the DSP Status window.

There is another way to manage polyphonic voice allocation in MSP—the **poly~** object. We'll look at the elegant and efficient **poly~** object (and its helper objects **in**, **in~**, **out**, **out~**, and **thispoly~**) in Tutorial 21; in the meantime, we'll use the **poly** object to make polyphonic voice assignments for the simple case required for this tutorial.

Pitch bend

In this instrument we use MIDI pitch bend values from 0 to 127 to bend the pitch of the instrument up or down by two semitones. Bending the pitch of a note requires multiplying its (carrier) frequency by some amount. For a bend of ±2 semitones, we will need to calculate a bend factor ranging from $2^{-2/12}$ (approximately 0.891) to $2^{2/12}$ (approximately 1.1225).

MIDI pitch bend presents a unique mapping problem because, according to the MIDI protocol, a value of 64 is used to mean "no bend" but 64 is not precisely in the center between 0 and 127. (The precise central value would be 63.5.) There are 64 values below 64 (0 to 63), but only 63 values above it (65 to 127). We will therefore need to treat upward bends slightly differently from downward bends.



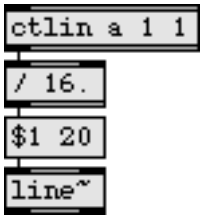
Downward bend is calculated slightly differently from upward bend

The downward bend values (0 to 63) are offset by -64 and divided by 384 so that the maximum downward bend (pitch bend value 0) produces an exponent of $^{-64}/_{384}$, which is equal to $^{-2}/_{12}$. The upward bend values (64 to 127) are offset by -64 and divided by 378 so that an upward bend produces an exponent ranging from 0 to $^{63}/_{378}$, which is equal to $^{2}/_{12}$.

The **pack** and **line~** objects are used to make the frequency factor change gradually over 20 milliseconds, to avoid creating the effect of discrete stepwise changes in frequency.

Mod wheel

The mod wheel is used here to change the modulation index of our FM synthesis patch. The mapping is linear; we simply divide the MIDI controller values by 16 to map them into a range from 0 to (nearly) 8. The precise way this range is used will be seen when we look at the synthesis instrument itself.



Controller values mapped into the range 0 to 7.9375

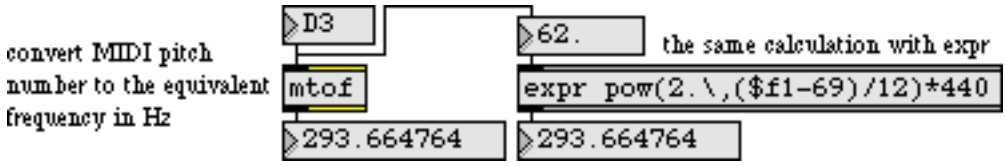
The FM synthesizer

- Double-click on one of the **synthFMvoice~** subpatch objects to open its Patcher window.

The basis for this FM synthesis subpatch is the **simpleFM~** subpatch introduced (and explained) in *Tutorial 11*. A typed-in argument is used to set the harmonicity ratio at 1, yielding a harmonic spectrum. The MIDI messages will affect the frequency and the modulation index of this FM sound. Let's look first at the way MIDI note and pitch bend information is used to determine the frequency.

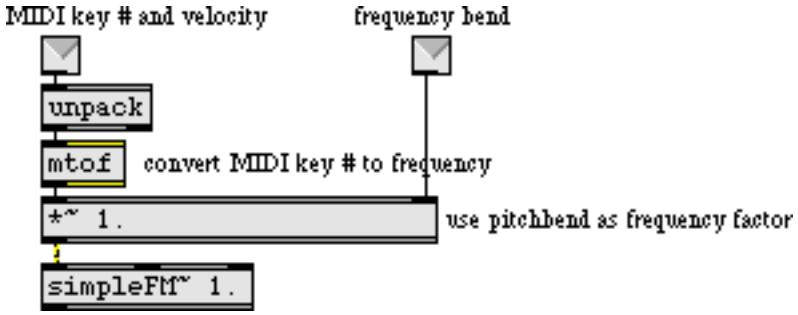
MIDI-to-frequency conversion

The object **mtof** is not a signal object, but it is very handy for use in MSP. It converts a MIDI key number into its equivalent frequency.



Calculate the frequency of a given pitch

This frequency value is multiplied by the bend factor which was calculated in the main patch, and the result is used as the carrier frequency in the **simpleFM~** subpatch.



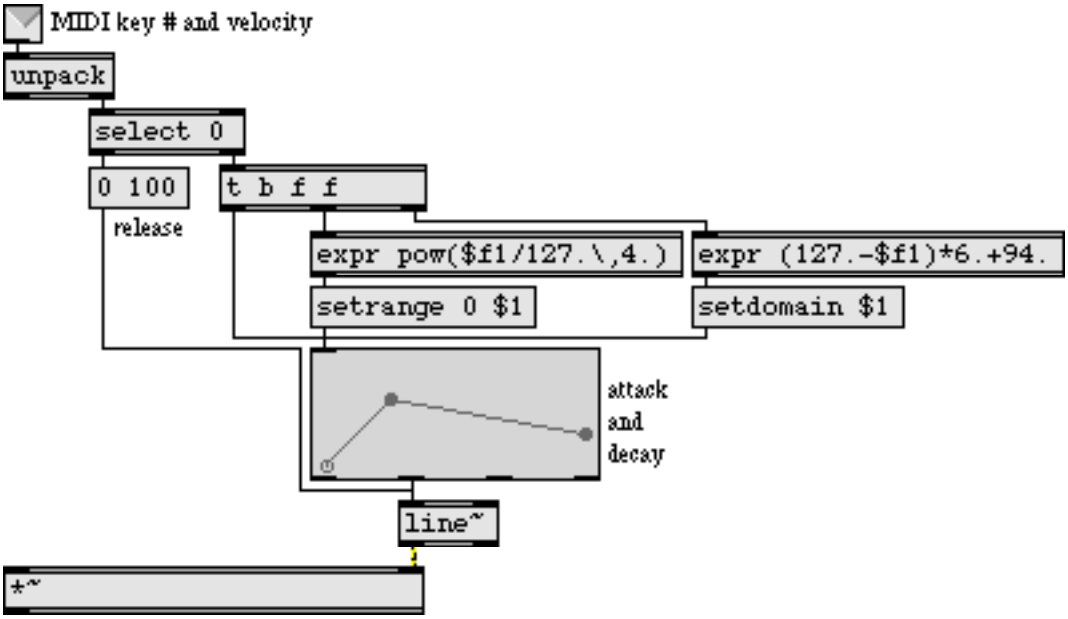
The frequency of the note calculated from key number and pitch bend data

Velocity control of amplitude envelope

MIDI note-on velocity is used in this patch, as in most synthesizers, to control the amplitude envelope. The tasks needed to accomplish this are

- Separate note-on velocities from note-off velocities.
- Map the range of note-on velocities—1 to 127—into an amplitude range from 0 to 1 (a non- linear mapping is usually best).
- Map note-on velocity to rate of attack and decay of the envelope (in this case).

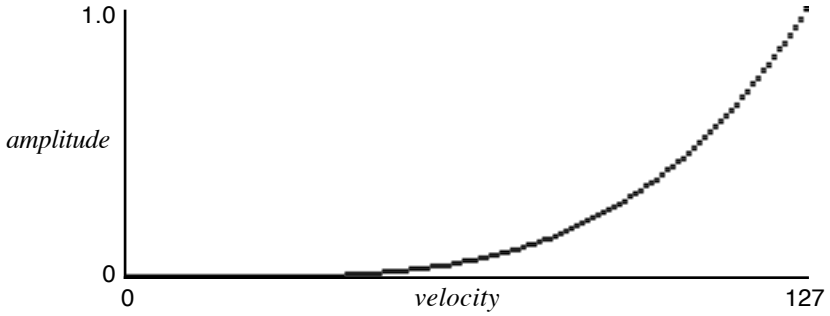
The first task is achieved easily with a **select** 0 object, so that note-on velocity triggers a **function** object to send the attack and decay shape, and note-off velocity returns the amplitude to 0, as shown in the following example.



MIDI note-on velocity sets domain and range of the amplitude envelope

Before the **function** is triggered, however, we use the note-on velocity to set the *domain* and *range*, which determine the duration and amplitude of the envelope. The **expr** object on the right calculates the amount of time in which the attack and decay portions of the envelope will occur. Maximum velocity of 127 will cause them to occur in 100 ms, while a much lesser velocity of 60 will cause them to occur in 496 ms. Thus notes that are played more softly will have a slower attack, as is the case with many wind and brass instruments.

The **expr** object on the left maps velocity to an exponential curve to determine the amplitude.



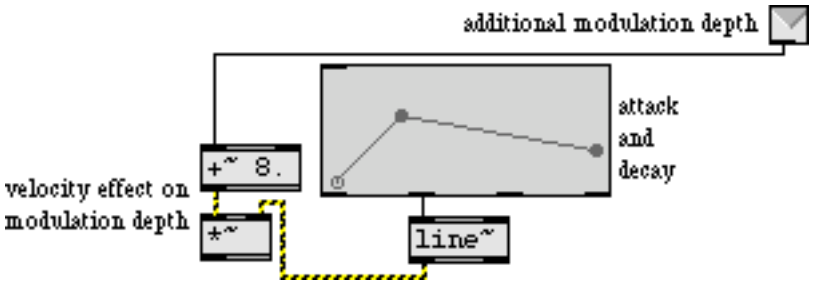
Velocity mapped to amplitude with an exponent of 4

If we used a straight linear mapping, MIDI velocities from 127 to 64 (the range in which most notes are played) would cover only about a 6 dB amplitude range. The exponential mapping increases this to about 24 dB, so that change in the upper range of velocities produces a greater change in amplitude.

MIDI control of timbre

It's often the case that acoustic instruments sound brighter (contain more high frequencies) when they're played more loudly. It therefore makes sense to have note-on velocity affect the timbre of the sound as well as its loudness. In the case of brass instruments, the timbre changes very much in correlation with amplitude, so in this patch we use the same envelope to control both the amplitude *and* the modulation index of the FM instrument. The envelope is sent to a ***~** object to scale it into the proper range. The **+~ 8** object ensures that the modulation index affected by velocity ranges from 0 to 8 (when the note is played with maximum velocity). As we saw earlier, in the main patch the modulation wheel can be used to increase the modulation index still further (adding up to 8 more to the modulation index range).

Thus, the combination of velocity and mod wheel position can affect the modulation index substantially.



Envelope and mod wheel control modulation index

- Listening only to MSP (with the volume turned down on your keyboard synth), play a single- line melody on the MIDI keyboard. As you play, notice the effect that velocity has on the amplitude, timbre, and rate of attack. Move the mod wheel upward to increase the over-all brightness of the timbre. You can also use the mod wheel to modulate the timbre during the sustain portion of the note. Try out the pitch bend wheel to confirm that it has the intended effect on the frequency.

Summary

MIDI data can be used to control an MSP synthesis patch much like any other synthesizer. In normal instrument design, MIDI key number and pitch bend wheel position are both used to determine the pitch of a played note. The key number must be converted into frequency information with the **mtof** object. The pitch bend value must be converted into the proper frequency bend factor, based on the twelfth-root-of-two per semitone used in equal temperament. Since the designated "no-bend" value of 64 is not in the precise center of the 0 to 127 range, upward bend must be calculated slightly differently from downward bend.

Note-on velocity is generally used to determine the amplitude of the note, and triggers the attack portion of the amplitude envelope. The note-off message triggers the release portion of the envelope. The velocity value can be used to alter the range of the envelope (or to provide a factor for scaling the amplitude). It is usually best to map velocity to amplitude exponentially rather than linearly. Velocity can also be used to alter the rate of the envelope, and/or other parameters such as modulation index.

An MSP object can only make one sound at a time, so if you want to play more than one simultaneous note via MIDI you will need to assign each note a voice number with **poly**, and route each voice to a different MSP object. In the next tutorial, we'll use the **poly** object to make polyphonic voice assignments for the simple case required for this tutorial.

Tutorial 21 will introduce another way to manage polyphonic voice allocation in MSP—the **poly~** object.

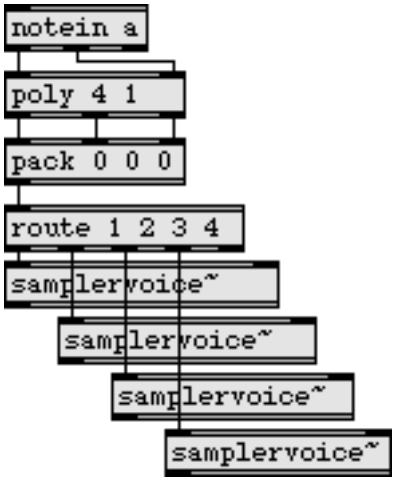
See Also

mtof	Convert a MIDI note number to frequency
poly	Allocate notes to different voices

Basic sampler features

In this chapter we demonstrate a design for playing pre-recorded samples from a MIDI keyboard. This design implements some of the main features of a basic sampler keyboard: assigning samples to regions of the keyboard, specifying a base (untransposed) key location for each sample, playing samples back with the proper transposition depending on which key is played, and making polyphonic voice assignments. For the sake of simplicity, this patch does not implement control from the pitchbend wheel or mod wheel, but the method for doing so would not be much different from that demonstrated in the previous two chapters.

In this patch we use the **groove~** object to play samples back at various speeds, in some cases using looped samples. As was noted in *Tutorial 19*, if we want a polyphonic instrument we need as many sound-generating objects as we want separate simultaneous notes. In this tutorial patch, we use four copies of a subpatch called **samplervoice~** to supply four-voice polyphony. As in *Tutorial 19*— we use a **poly** object to assign a voice number to each MIDI note, and we use **route** to send the note information to the correct **samplervoice~** subpatch.



poly assigns a voice number to each MIDI note, to send information to the correct subpatch

Before we examine the workings of the **samplervoice~** subpatch, it will help to review what information is needed to play a sample correctly.

1. The sound samples must be read into memory (in **buffer~** objects), and a list of the memory locations (**buffer~** names) must be kept.

2. Each sample must be assigned to a region of the keyboard, and a list of the key assignments must be kept.

3. A list of the base key for each region—the key at which the sample should play back untransposed—must be kept.

4. A list of the loop points for each sample (and whether looping should be on or off) must be kept.

5. When a MIDI note message is received, and is routed to a **samplervoice~** subpatch, the **groove~** object in that subpatch must first be told which **buffer~** to read (based on the key region being played), how fast to play the sample (based on the ratio between the frequency being played and the base key frequency for that region), what loop points to use for that sample, whether looping is on or off, and what amplitude scaling factor to use based on the note- on velocity.

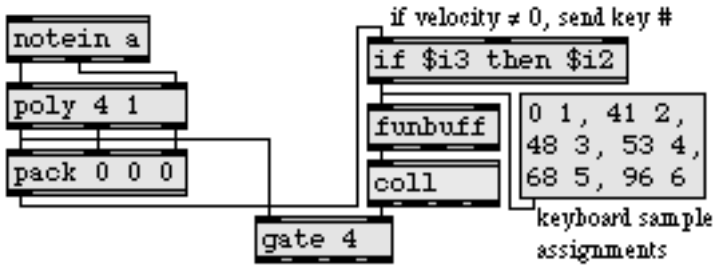
In this patch, the samples are all read into memory when the patch is first loaded.

- Double-click on the **p** samplebuffers subpatch to open its Patcher window.

You can see that six samples have been loaded into **buffer~** objects named sample1, sample2, etc. If, in a performance situation, you need to have access to more samples than you can store at once in RAM, you can use read messages with filename arguments to load new samples into **buffer~** objects as needed.

- Close the subpatch window. Click on the **message** box marked "keyboard sample assignments".

This stores a set of numbered key regions in the **funbuff** object. (This information could have been embedded in the **funbuff** and saved with the patch, but we left it in the **message** box here so that you can see the contents of the **funbuff**.) MIDI key numbers 0 to 40 are key region 1, keys 41 to 47 are key region 2, etc. When a note-on message is received, the key number goes into **funbuff**, and **funbuff** reports the key region number for that key. The key region number is used to look up other vital information in the **coll**.



Note-on key number finds region number in funbuff, which looks up sample info in coll

- Double-click on the **coll** object to see its contents.

```
1, 24 sample1 0 0 0;
2, 33 sample2 0 0 0;
3, 50 sample3 0.136054 373.106537 1;
4, 67 sample4 60.204079 70.476189 1;
5, 84 sample5 0 0 0;
6, 108 sample6 0 0 0;
```

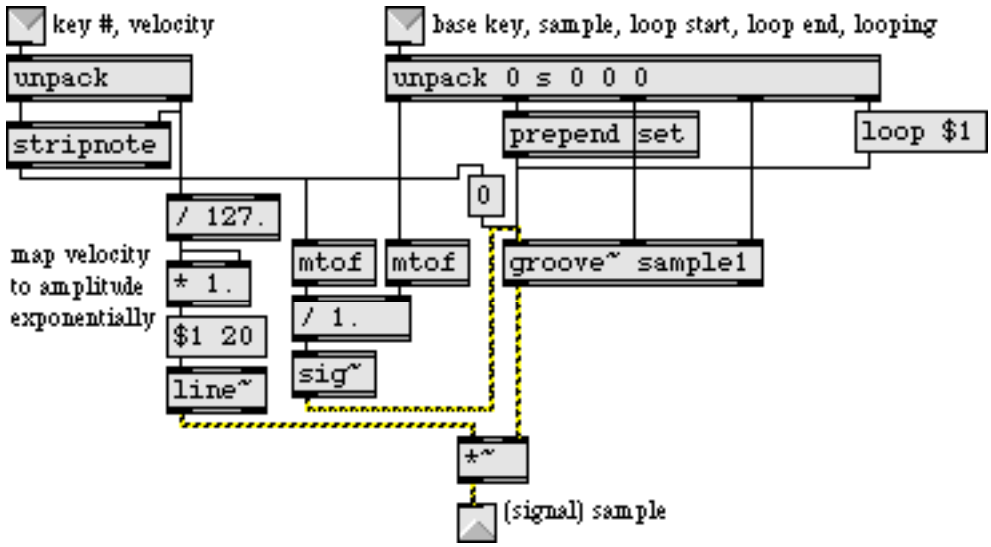
coll contains sample information for each key region

The key region number is used to index the information in **coll**. For example, whenever a key from 48 to 52 is pressed, **funbuff** sends out the number 3, and the information for key region 3 is recalled and sent to the appropriate **samplervoice~** subpatch. The data for each key region is: base key, **buffer~** name, loop start time, loop end time, and loop on/off flag.

The voice number from **poly** opens the correct outlet of **gate** so that the information from **coll** goes to the right subpatch.

Playing a sample: the samplervoice~ subpatch

- Close the **coll** window, and double-click on one of the **samplervoice~** subpatch objects to open its Patcher window.



The samplervoice~ subpatch

You can see that the information from **coll** is unpacked in the subpatch and is sent to the proper places to prepare the **groove~** object for the note that is about to be played. This tells **groove~** what **buffer~** to read, what loop times to use, and whether looping should be on or off. Then, when the note information comes in the left inlet, the velocity is used to send an amplitude value to the ***~** object, and the note-on key number is used (along with the base key number received from the right inlet) to calculate the proper playback speed for **groove~** and to trigger **groove~** to begin playback from time 0.

MSP sample rate vs. audio file sample rate

- Close the subpatch window.

You're almost ready to begin playing samples, but there is one more detail to attend to first. To save storage space, the samples used in this patch are mono AIFF files with a sample rate of 22,050 Hz. To hear them play properly you should set the sample rate of MSP to that rate.

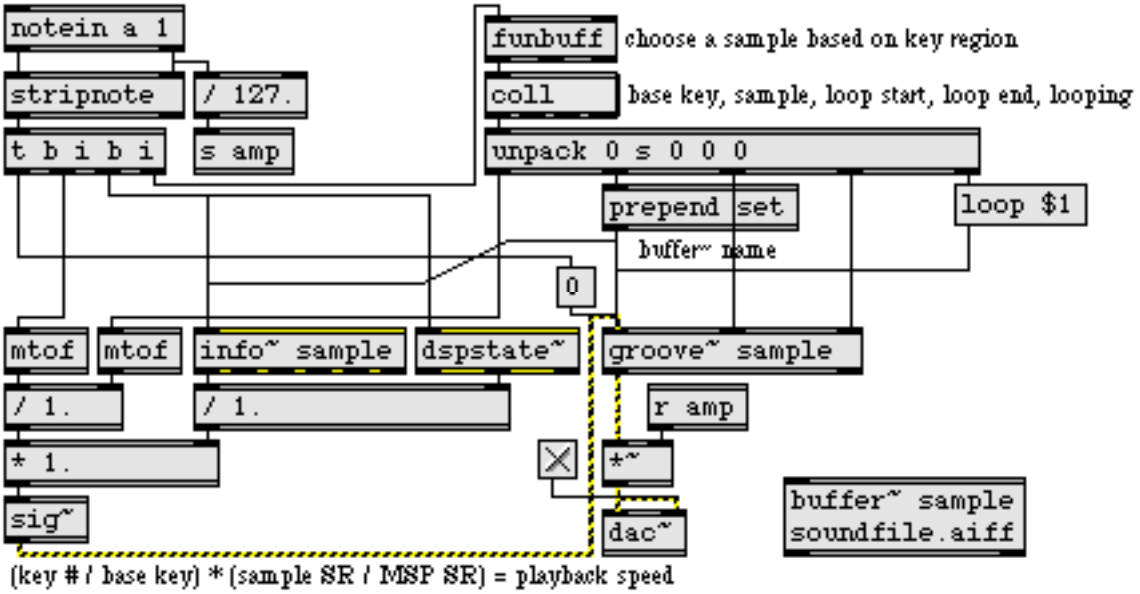
- Double-click on the **dac~** object to open the DSP Status window. Set the Sampling Rate to 22.050 kHz, then close the DSP Status window.

Note: Resetting the sampling rate may not be possible, depending on your hardware.

The difference between the sample rate of an audio file and the sample rate being used in MSP is a potential problem when playing samples. This method of resolving the difference suffices in this situation because the audio files are all at the same sample rate and because these samples are the only sounds we will be playing in MSP. In other situations, however, you're likely to want to play samples (perhaps with different sampling rates) combined with other sounds in MSP, and you'll want to use the optimum sampling rate.

For such situations, you would be best advised to use the ratio between the audio file sample rate and the MSP sample rate as an additional factor in determining the correct playback speed for **groove~**. For example, if the sample rate of the audio file is half the sample rate being used by MSP, then **groove~** should play the sample half as fast.

You can use the objects **info~** and **dspstate~** to find out the sampling rate of the sample and of MSP respectively, as demonstrated in the following example.



Calculate playback speed based on the sampling rates of the audio file and of MSP

The note-on key number is used first to recall the information for the sample to be played. The name of a **buffer~** is sent to **groove~** and **info~**. Next, a bang is sent to **dspstate~** and **info~**. Upon receiving a bang, **dspstate~** reports the sampling rate of MSP and **info~** reports the sampling rate of the AIFF file stored in the **buffer~**. In the lower left part of the example, you can see how this sampling rate information is used as a factor in determining the correct playback speed for **groove~**.

Playing samples with MIDI

- Turn audio on and set the "Output Level" **number box** to a comfortable listening level. Play a slow chromatic scale on the MIDI keyboard to hear the different samples and their arrangement on the keyboard.

To arrange a unified single instrument sound across the whole keyboard, each key region should contain a sample of a note from the same source. In this case, though, the samples are arranged on the keyboard in such a way as to make available a full "band" consisting of drums, bass, and keyboard. This sort of multi-timbral keyboard layout is useful for simple keyboard splits (such as bass in the left hand and piano in the right hand) or, as in this case, for accessing several different sounds on a single MIDI channel with a sequencer.

- For an example of how a multi-timbral sample layout can be used by a sequencer, click on the **toggle** marked "Play Sequence". Click on it again when you want to stop the sequence. Turn audio off. Double-click on the **p** sequence object to open the Patcher window of the subpatch.

The p sequence subpatch

The **seq** sampleseq.midi object contains a pre-recorded MIDI file. The **midiparse** object sends the MIDI key number and velocity to **poly** in the main patch. Each time the sequence finishes playing, a bang is sent out the right outlet of **seq**; the bang is used to restart the **seq** immediately, to play the sequence as a continuous loop. When the sequence is stopped by the user, a bang is sent to **midiflush** to turn off any notes currently being played.

- When you have finished with this patch, don't forget to open the DSP Status window and restore the Sampling Rate to its original setting.

Summary

To play samples from the MIDI keyboard, load each sample into a **buffer~** and play the samples with **groove~**. For polyphonic sample playback, you will need one **groove~** object per voice of polyphony. You can route MIDI notes to different **groove~** objects using voice assignments from the **poly** object.

To assign each sample to a region of the MIDI keyboard, you will need to keep a list of key regions, and for each key region you will need to keep information about which **buffer~** to use, what transposition to use, what loop points to use, etc. A **funbuff** object is good for storing keyboard region assignments. The various items of information about each sample can be best stored together as lists in a **coll**, indexed by the key region number. When a note is played, the key region is looked up in the **funbuff**, and that number is used to look up the sample information in **coll**.

The proper transposition for each note can be calculated by dividing the played frequency (obtained with the **mtof** object) by the base frequency of the sample. The result is used as the playback speed for **groove~**. If the sampling rate of the recorded samples differs from the sampling rate being used in MSP, that fact must be accounted for when playing the samples with **groove~**. Dividing the audio file sampling rate by the MSP sampling rate provides the correct factor by which to multiply the playback speed of **groove~**. The sampling rate of MSP can be obtained with the **dspstate~** object. The sampling rate of the AIFF file in a **buffer~** can be obtained with **info~** (Remember—resetting the sampling rate may not be possible on your hardware).

Note-on velocity can be used to control the amplitude of the samples. An exponential mapping of velocity to amplitude is usually best. Multi-timbral sample layouts on the keyboard can be useful for playing many different sounds, especially from a sequencer. The end-of-file bang from the right outlet of **seq** can be used to restart the **seq** to play it in a continuous loop. If the MIDI data goes through a **midiflush** object, any notes that are on when the **seq** is stopped can be turned off by sending a bang to **midiflush**.

See Also

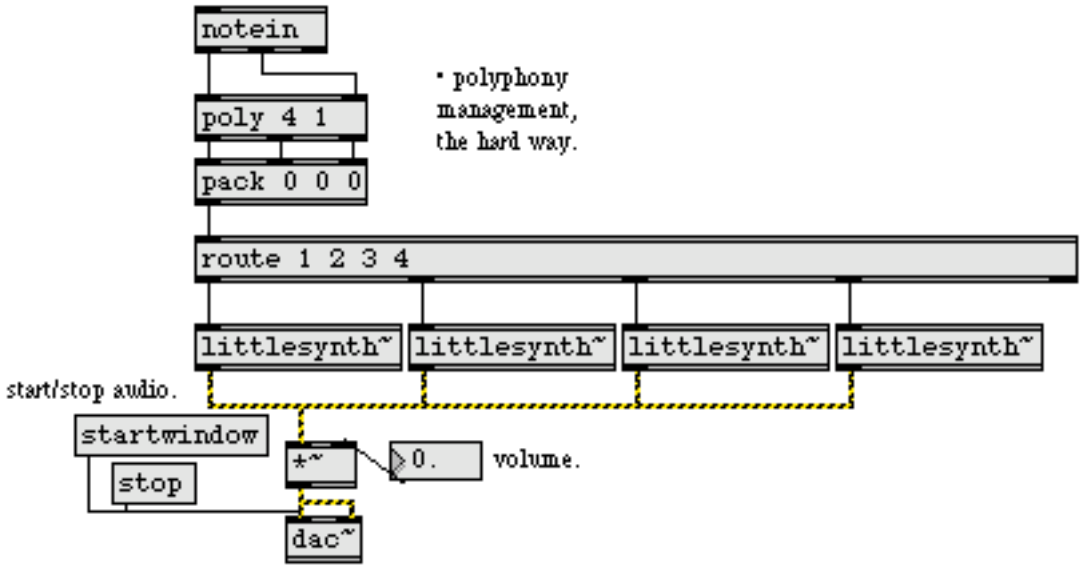
buffer~	Store audio samples
dspstate~	Report current DSP setting
groove~	Variable-rate looping sample playback
poly~	Polyphony/DSP manager for patchers

Tutorial 21: MIDI control—Using the poly~ object

A different approach to polyphony

In the last chapter, we demonstrated how to use the **poly** object to make polyphonic voice assignments in a simple case. This chapter will describe a more elegant and efficient way to handle polyphonic voice allocation—the **poly~** object.

In the example in the previous chapter, we created multiple copies of our sampler subpatch and used the poly object's voice numbering to route messages to different copies of the subpatch. Our example could just as easily have used any kind of sound-producing subpatch. The following example uses the subpatch **littlesynth~** to implement a simple four-voice polyphonic synthesizer:

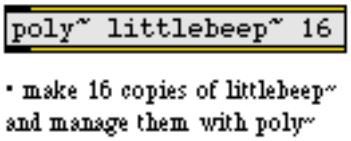


While this method works, it has two disadvantages. First, there's a lot of housekeeping necessary to duplicate and patch the multiple copies of **littlesynth~** together. But there is also a problem in terms of CPU usage. All four copies of the **littlesynth~** subpatcher are always on, processing their audio even when there is no sound being produced.

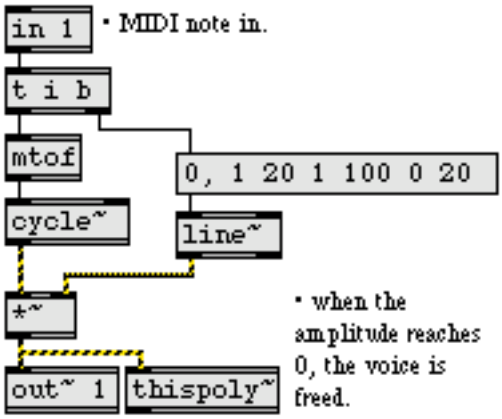
MSP 2.0, introduces a different way to solve the problem—the **poly~** object allows you to create and manage multiple copies of the same MSP subpatch all within one object. You can also control the signal processing activity within each copy of the subpatch to conserve CPU resources.

The poly~ object

The **poly~** object takes as its argument the name of a patcher file, followed by a number that specifies the number of copies (or *instances*) of the patch to be created. You'll want to specify the same number of copies as you would have had to duplicate manually when implementing polyphony the old-fashioned way. Here's an example of the **poly~** object.



Double-clicking on the **poly~** object opens up the subpatcher to show you the inside of the **littlebeep~** object:



Let's look at the **littlebeep~** patch for a minute. While you haven't seen the **in**, **out~**, or **thispoly~** objects before, the rest of the patcher is pretty straightforward; it takes an incoming MIDI note number, converts it to a frequency value using the **mtof** object, and outputs a sine wave at that frequency with a duration of 140 milliseconds and an amplitude envelope supplied by the **line~** object for 140 ms with an envelope on it.

But what about the **in** and **out~** objects? Subpatches created for use in the **poly~** object use special objects for inlets and outlets. The objects **in** and **out** create control inlets and outlets, and the **in~** and **out~** objects create signal inlets and outlets. You specify which inlet is assigned to which object by adding a number argument to the object—the **in 1** object corresponds to the leftmost inlet on the **poly~** object, and so on. The **poly~** object

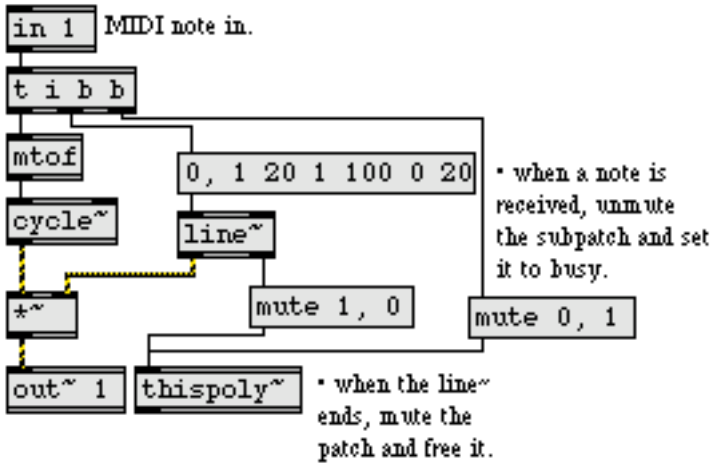
keeps track of the number of inlets and outlets it needs to create when you tell it which subpatch to load.

Messages sent to a **poly~** object are directed to different instances of the subpatch dynamically using the note and midinote messages, and manually using the target message.

When **poly~** receives a note message in its left inlet, it scans through the copies of the subpatch it has in memory until it finds one that is currently not busy, and then passes the message to it. A subpatch instance can tell its parent **poly~** object that it is busy using the **thispoly~** object. The **thispoly~** object accepts either a signal or number in its inlet to set its busy state. A zero signal or a value of 0 sent to its inlet tells the parent **poly~** that this instance is available for note or midinote messages. A non-zero signal or value sent to its inlet tells the parent **poly~** that the instance is busy; no note or midinote messages will be sent to the object until it is no longer busy. The busy state was intended to correspond to the duration of a note being played by the subpatcher instance, but it could be used to mean anything. In the example above, when the audio level out of the ***~** is nonzero—that iteration of the subpatch is currently busy. Once the amplitude envelope out of **line~** reaches zero and the sound stops, that subpatch's copy of **thispoly~** tells **poly~** that it is ready for more input.

The **thispoly~** object can also control the activity of signal processing within each copy of the subpatch. When the mute message is sent to **thispoly~** followed by a 1, all signal processing in that subpatch stops. When a mute 0 message is received, signal processing starts again.

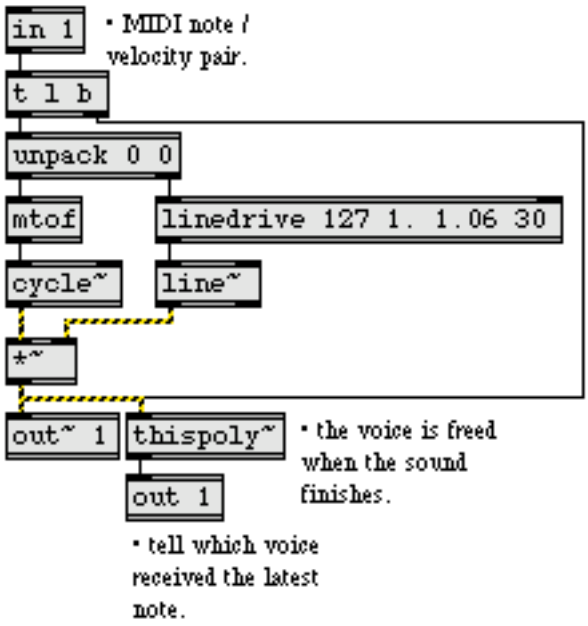
We can rewrite the **littlebeep~** subpatcher to take advantage of this by turning off signal processing when a note is finished and turning it on again when a new event is received:



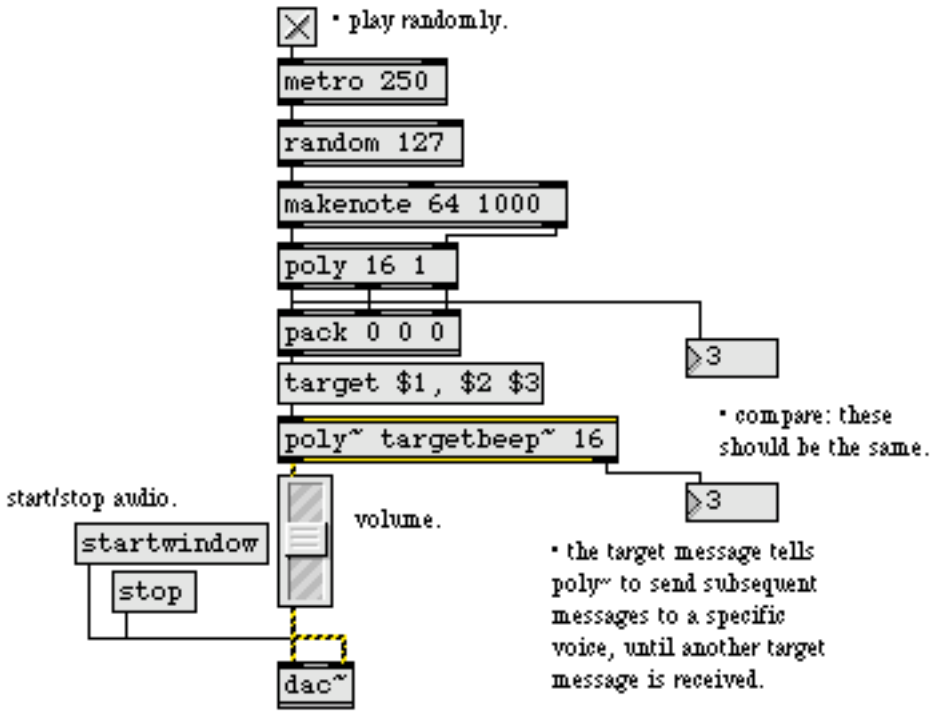
While this doesn't change the function of the patch, it would be more efficient, since the amount of CPU allocated is always based on the number of notes currently sounding.

Another way to allocate events using **poly~** is through the target message. Sending a target message followed by an integer in the left inlet of a **poly~** subpatch tells **poly~** to send all subsequent messages to that instance of the subpatch. You can then use **poly~** in conjunction with the **poly** object from the last chapter to create a MIDI synthesizer.

A **poly~** subpatch that uses the target message looks like this:

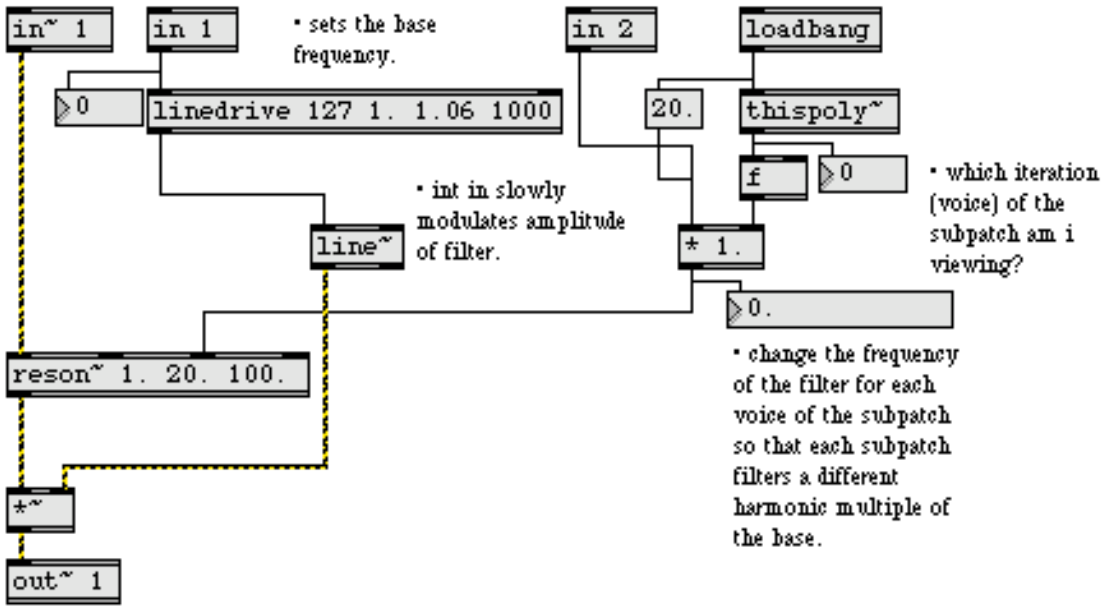


In this example subpatcher, pairs of incoming MIDI pitches and velocities are used to synthesize a sine tone. When a list is received, the subpatcher sends a bang to **thispoly~**, causing it to output the instance or voice number. In the example below, the voice number is sent out an outlet so you can watch it from the parent patch.



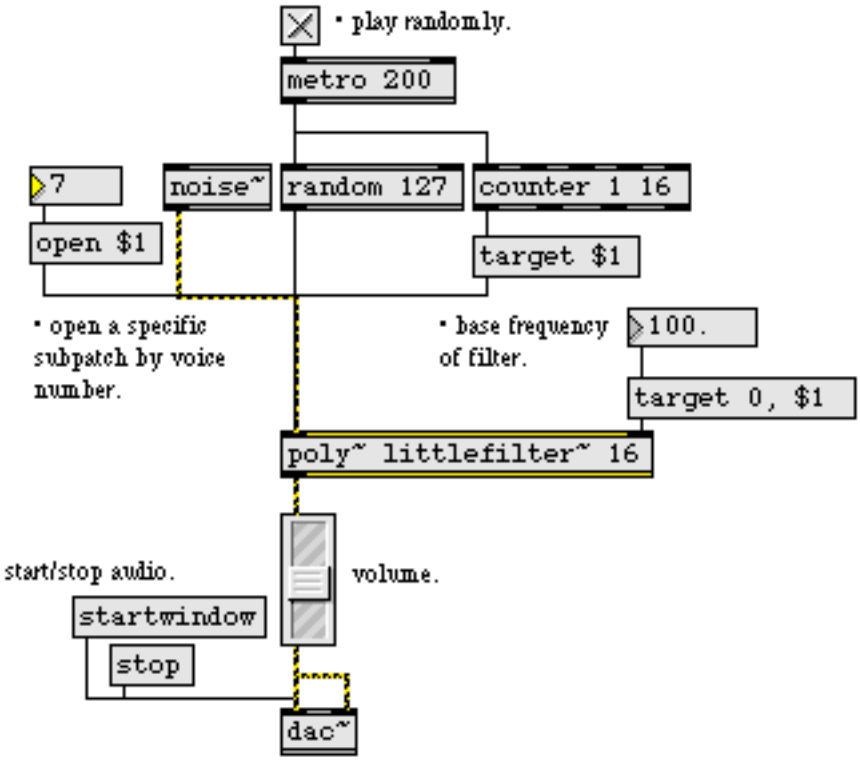
In the parent patch the **poly** object assigns voice numbers to MIDI pitch/velocity pairs output by **makenote**. The voice number from the **poly** object is sent to **poly~** with the target message prepended to it, telling **poly~** to send subsequent data to the instance of the **targetbeep~** subpatcher specified by **poly~**. When a new note is generated, the target will change. Since **poly** keeps track of note-offs, it should recycle voices properly. The second outlet of **poly~** reports the voice that last received a message—it should be the same as the voice number output by **poly**, since we're using **poly** to specify a specific target.

The **thispoly~** object can be used to specify parameters to specific instances of a **poly~** subpatcher. By connecting a **loadbang** object to **thispoly~**, we can use the voice number to control the center frequency of a filter:



The **littlefilter~** subpatcher, shown here uses the voice number from **thispoly~** and multiplies it by the base frequency received in the second inlet. The incoming signal is filtered by all sixteen instances simultaneously, with the output amplitude of each instance being controlled by an integer coming into the first inlet.

Here's an example of a patch which uses **littlefilter~**

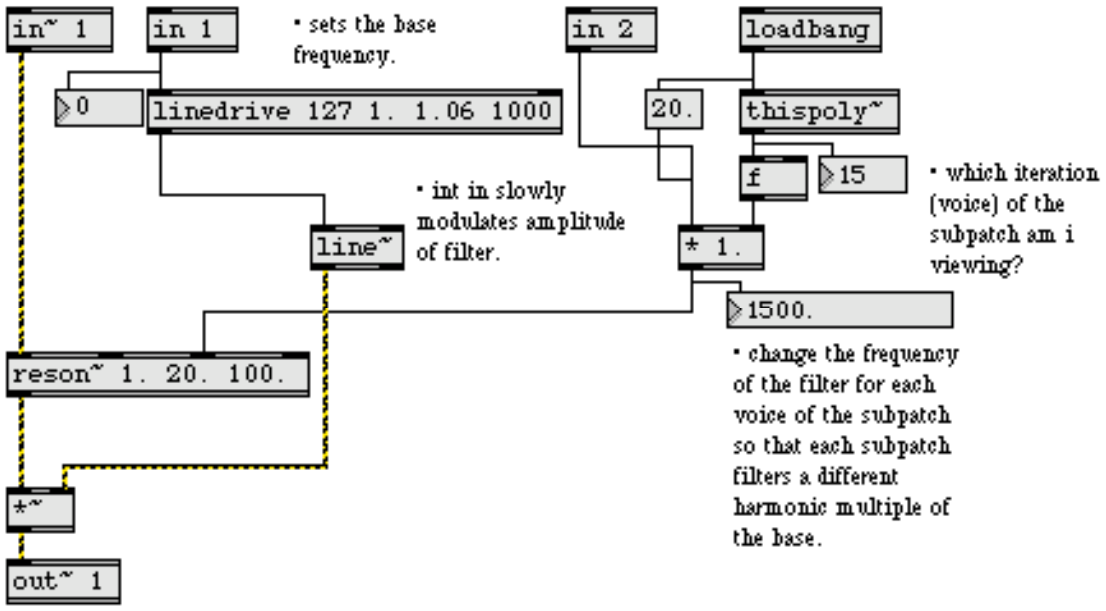


:

The **metro** object is hooked up to both a **counter** and a **random**. The **counter**, which feeds the target message, cycles through the 16 voices of **littlefilter~** loaded into the **poly~** object, supplying each with a random number which is used to control the amplitude of that voice.

A signal connected to an inlet of **poly~** will be sent to the corresponding **in~** objects of all subpatcher instances, so the **noise~** object in the example above feeds noise to all the subpatchers inside the **poly~**. The second inlet (which corresponds to the **in** 2 box in the subpatcher) controls the base frequency of the filters. Note that for the frequency to get sent to all **poly~** iterations it is preceded by a target 0 message. You can open a specific instance of a **poly~** subpatch by giving the object the open message, followed by the voice you want to look at.

The subpatch assigned to voice number 15 looks like this:



As you can see, the base frequency of this particular iteration of **littlefilter~** is 1500. Hz, which is the multiple of the voice number (15) with the most recently entered base frequency into the second inlet (100. Hz).

Summary

poly~ is a powerful way to manage multiple copies of the same subpatch for polyphonic voice allocation. The **thispoly~** object works inside a subpatch to control its busy state and turn signal processing on and off. The objects **in**, **in~**, **out**, and **out~** create special control and signal inputs and outputs that work with the inlets and outlets of the **poly~** object.

See Also

in	Message input for a patcher loaded by **poly~**
in~	Signal input for a patcher loaded by **poly~**
out	Message output for a patcher loaded by **poly~**
out~	Signal output for a patcher loaded by **poly~**
poly~	Polyphony/DSP manager for patchers
thispoly~	Control **poly~** voice allocation and muting

Panning for localization and distance effects

Loudness is one of the cues we use to tell us how far away a sound source is located. The relative loudness of a sound in each of our ears is a cue we use to tell us in what direction the sound is located. (Other cues for distance and location include inter-aural delay, ratio of direct to reflected sound, etc. For now we"ll only be considering loudness.)

When a sound is coming from a single speaker, we localize the source in the direction of that speaker. When the sound is equally balanced between two speakers, we localize the sound in a direction precisely between the speakers. As the balance between the two speakers varies from one to the other, we localize the sound in various directions between the two speakers.

The term panning refers to adjusting the relative loudness of a single sound coming from two (or more) speakers. On analog mixing consoles, the panning of an input channel to the two channels of the output is usually controlled by a single knob. In MIDI, panning is generally controlled by a single value from 0 to 127. In both cases, a single continuum is used to describe the balance between the two stereo channels, even though the precise amplitude of each channel at various intermediate points can be calculated in many different ways.

All other factors being equal, we assume that a softer sound is more distant than a louder sound, so the overall loudness effect created by the combined channels will give us an important distance cue. Thus, panning must be concerned not only with the proper balance to suggest direction of the sound source; it must also control the perceived loudness of the combined speakers to suggest distance.

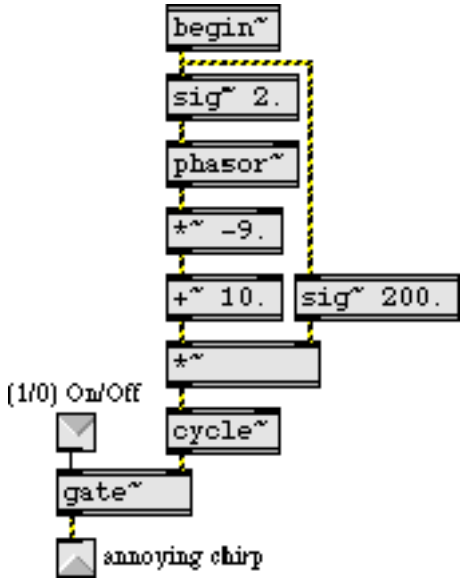
This tutorial demonstrates three ways of calculating panning, controllable by MIDI values 0 to 127. You can try out the three methods and decide which is most appropriate for a given situation in which you might want to control panning.

Patch for testing panning methods

In this tutorial patch, we use a repeated “chirp” (a fast downward glissando spanning more than three octaves) as a distinctive and predictable sound to pan from side to side.

- To see how the sound is generated, double-click on the **p** "sound source" subpatch to open its Patcher window.

Because of the **gate~** and **begin~** objects, audio processing is off in this subpatch until a 1 is received in the inlet to open the **gate~**. At that time, the **phasor~** generates a linear frequency glissando going from 2000 to 200 two times per second.

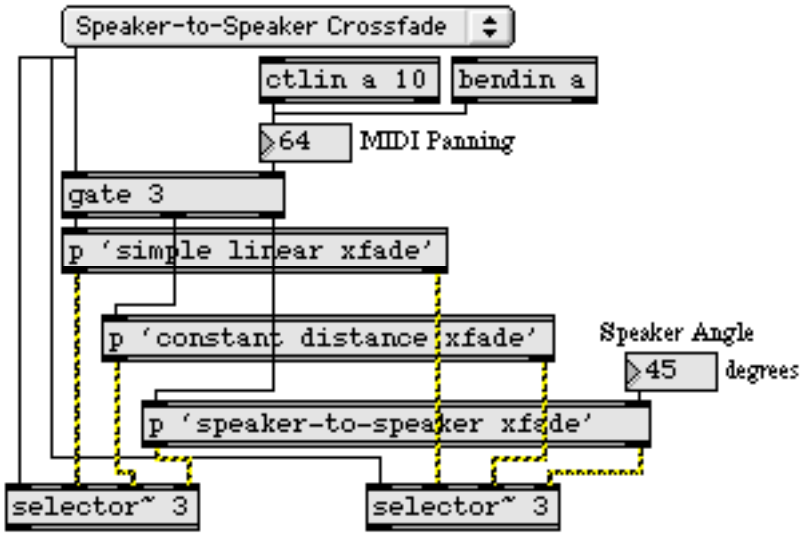


The p "sound source"subpatch

- Close the subpatch window.

The output of this subpatch is sent to two ***~** objects—one for each output channel—where its amplitude at each output channel will be scaled by one of the panning algorithms. You can choose the panning algorithm you want to try from the pop-up **umenu** at the top of the patch. This opens the inlet of the two **selector~** objects to receive the control signals from the correct panning subpatch. It also opens an outlet of the **gate** object to allow control values into the desired subpatch. The panning is controlled by MIDI input from continuous controller No. 10 (designated for panning in MIDI).

In case your MIDI keyboard doesn"t send controller 10 easily, you can also use the pitch bend wheel to test the panning. (For that matter, you don"t need MIDI at all. You can just drag on the **number box** marked “MIDI panning”.)

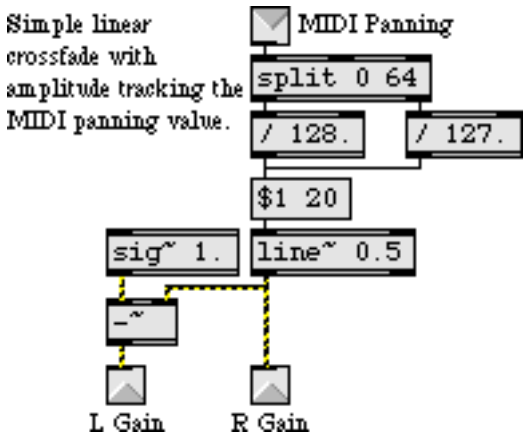


Selection from the umenu opens input and output for one of the three panning subpatches

Linear crossfade

The most direct way to implement panning is to fade one channel linearly from 0 to 1 as the other channel fades linearly from 1 to 0. This is the easiest type of panning to calculate. We map the range of MIDI values 0 to 127 onto the amplitude range 0 to 1, and use that value as the amplitude for the right channel; the left channel is always set to 1 minus the amplitude of the left channel. The only hitch is that a MIDI pan value of 64 is supposed to mean equal balance between channels, but it is not precisely in the center of the range ($^{64}/_{127}$ - 0.5). So we have to treat MIDI values 0 to 64 differently from values 65 to 127.

- Double-click on the **p** "simple linear xfade" object to open its Patcher window.



Linear crossfade using MIDI values 0 to 127 for control

This method seems perfectly logical since the sum of the two amplitudes is always 1. The problem is that the *intensity* of the sound is proportional to the sum of the *squares* of the amplitudes from each speaker. That is, two speakers playing an amplitude of 0.5 do not provide the same intensity (thus not the same perceived loudness) as one speaker playing an amplitude of 1. With the linear crossfade, then, the sound actually seems softer when panned to the middle than it does when panned to one side or the other.

- Close the subpatch window. Choose "Simple Linear Crossfade" from the **umenu**. Click on the **ezdac~** to turn audio on, click on the **toggle** to start the "chirping" sound, and use the "Amplitude" **number box** to set the desired listening level. Move the pitch bend wheel of your MIDI keyboard to pan the sound slowly from one channel to the other. Listen to determine if the loudness of the sound seems to stay constant as you pan.

While this linear crossfade might be adequate in some situations, we may also want to try to find a way to maintain a constant intensity as we pan.

Equal distance crossfade

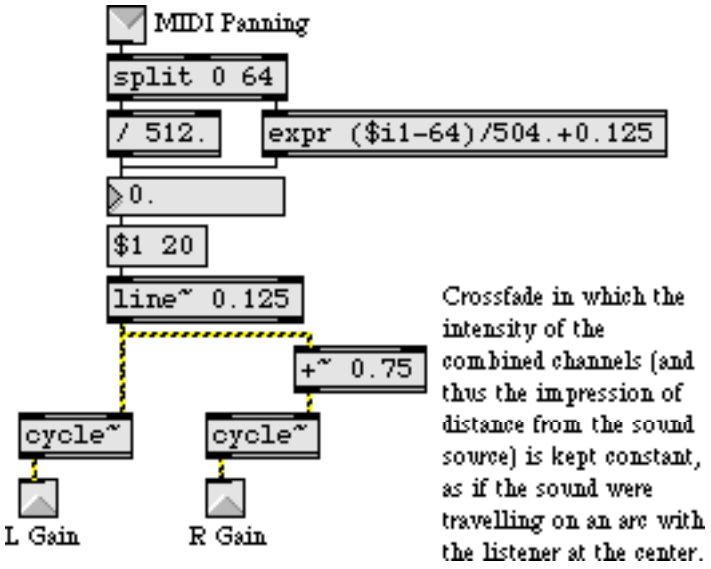
If we can maintain a constant intensity as we pan from one side to the other, this will give the impression that the sound source is maintaining a constant distance from the listener. Geometrically, this could only be true if the sound source were moving in an arc, with the listener at the center, so that the distance between the sound source and the listener was always equal to the radius of the arc.

It happens that we can simulate this condition by mapping one channel onto a quarter cycle of a cosine wave and the other channel onto a quarter cycle of a sine wave.

Therefore, we"ll map the range of MIDI values 0 to 127 onto the range 0 to 0.25, and use the result as an angle for looking up the cosine and sine values.

> **Technical detail:** As the sound source travels on a hypothetical arc from 0˚ to 90˚ (1/4 cycle around a circle with the listener at the center), the cosine of its angle goes from 1 to 0 and the sine of its angle goes from 0 to 1. At all points along the way, the square of the cosine plus the square of the sine equals 1. This trigonometric identity is analogous to what we are trying to achieve—the sum of the squares of the amplitudes always equaling the same intensity—so these values are a good way to obtain the relative amplitudes necessary to simulate a constant distance between sound source and listener.

- Double-click on the **p** "constant distance xfade" object to open its Patcher window.



MIDI values 0 to 127 are mapped onto 1/4 cycle of cosine and sine functions

Once again we need to treat MIDI values greater than 64 differently from those less than or equal to 64, in order to retain 64 as the "center" of the range. Once the MIDI value is mapped into the range 0 to 0.25, the result is used as a phase angle two **cycle~** objects, one a cosine and the other (because of the additional phase offset of 0.75) a sine.

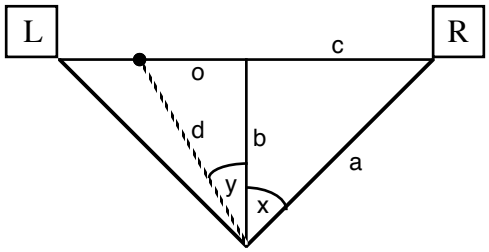
- Close the subpatch window. Choose "Equal Distance Crossfade" from the **umenu**. Listen to the sound while panning it slowly from one channel to the other.

Is the difference from the linear crossfade appreciable? Perhaps you don"t care whether the listener has the impression of movement in an arc when listening to the sound being panned. But the important point is that the equal distance method is preferable if only

because it does not cause a noticeable dip in intensity when panning from one side to the other.

Speaker-to-speaker crossfade

Given a standard stereo speaker placement—with the two speakers in front of the listener at equal distances and angles—if an actual sound source (say, a person playing a trumpet) moved in a straight line from one speaker to the other, the sound source would actually be *closer* to the listener when it"s in the middle than it would be when it"s at either speaker. So, to emulate a sound source moving in a straight line from speaker to speaker, we will need to calculate the amplitudes such that the intensity is proportional to the distance from the listener.



Distance b is shorter than distance a

Technical detail: If we know the angle of the speakers (*x* and *-x*), we can use the cosine function to calculate distance *a* relative to distance *b*. Similarly we can use the tangent function to calculate distance c relative to b. The distance between the speakers is thus *2c*, and as the MIDI pan value varies away from its center value of 64 it can be mapped as an offset (*o*) from the center ranging from *-c* to *+c*. Knowing *b* and *o*, we can use the Pythagorean theorem to obtain the distance (*d*) of the source from the listener, and we can use the arctangent function to find its angle (*y*). Armed with all of this information, we can finally calculate the gain for the two channels as $a \cdot cos(y \pm x)/d$.

- Choose "Speaker-to-Speaker Crossfade" from the **umenu**. Listen to the sound while panning it slowly from one channel to the other. You can try different speaker angles by changing the value in the "Speaker Angle" **number box**. Choose a speaker angle best suited to your actual speaker positions.

This effect becomes more pronounced as the speaker angle increases. It is most effective with "normal" speaker angles ranging from about 30˚ up to 45˚, or even up to 60˚. Below 30˚ the effect is too slight to be very useful, and above about 60˚ it"s too extreme to be realistic.

- Double-click on the **p** "speaker-to-speaker xfade" object to open its Patcher window.

The trigonometric calculations described above are implemented in this subpatch. The straight ahead distance (*b*) is set at 1, and the other distances are calculated relative to it. The speaker angle—specified in degrees by the user in the main patch—is converted to a fraction of a cycle, and is eventually converted to radians (multiplied by 2π, or 6.2832) for the trigonometric operations. When the actual gain value is finally calculated, it is multiplied by a normalizing factor of *2/ (d+b)* to avoid clipping. When the source reaches an angle greater than 90˚ from one speaker or the other, that speaker"s gain is set to 0.

- To help get a better understanding of these calculations, move the pitch bend wheel and watch the values change in the subpatch. Then close the subpatch and watch the gain values change in the main Patcher window.

The signal gain values are displayed by an MSP user interface object called **number~**, which is explained in the next chapter.

Summary

MIDI controller No. 10 (or any other MIDI data) can be used to pan a signal between output channels. The relative amplitude of the two channels gives a localization cue for direction of the sound source. The overall intensity of the sound (which is proportional to the sum of the squares of the amplitudes) is a cue for perceived distance of the sound source.

Mapping the MIDI data to perform a linear crossfade of the amplitudes of the two channels is one method of panning, but it causes a drop in intensity when the sound is panned to the middle. Using the panning value to determine the *angle* of the sound source on an arc around the listener (mapped in a range from 0˚ to 90˚), and setting the channel amplitudes proportional to the cosine and sine of that angle, keeps the intensity constant as the sound is panned.

When a sound moves past the listener in a straight line, it is loudest when it passes directly in front of the listener. To emulate straight line movement, one can calculate the relative distance of the sound source as it travels, and modify the amplitude of each channel (and the overall intensity) accordingly.

See Also

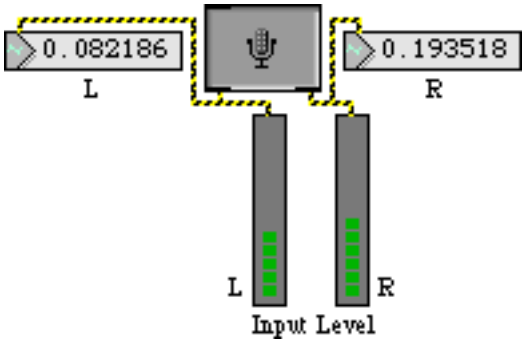
expr	Evaluate a mathematical expression
gate~	Route a signal to one of several outlets

Display the value of a signal: number~

This chapter demonstrates several MSP objects for observing the numerical value of signals, and/ or for translating those values into Max messages.

- Turn audio on and send some sound into the input jacks of the computer.

Every 250 milliseconds the **number~** objects at the top of the Patcher display the current value of the signal coming in each channel, and the **meter~** objects show a graphic representation of the peak amplitude value in the past 250 milliseconds, like an analog LED display.

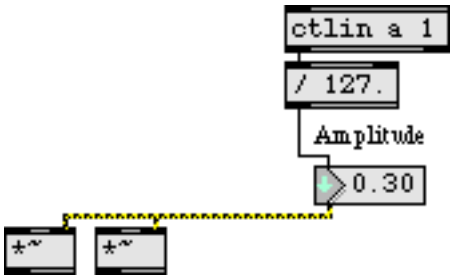


Current signal value is shown by number~; peak amplitude is shown by meter~

The signal coming into **number~** is sent out its right outlet as a float once every time it's displayed. This means it is possible to sample the signal value and send it as a message to other Max objects.

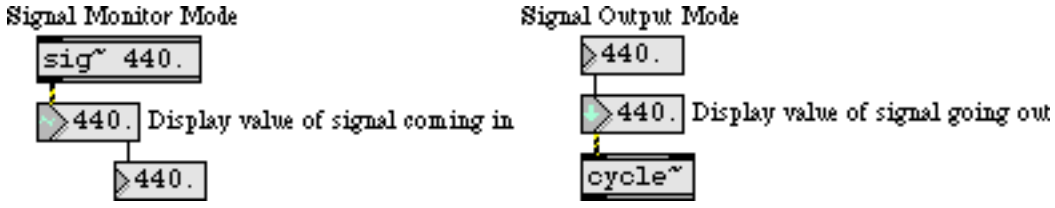
The **number~** object is actually like two objects in one. In addition to receiving signal values and sending them out the right outlet as a float, **number~** also functions as a floating-point **number box** that sends a signal (instead of a float) out its left outlet.

- Move the mod wheel of your MIDI keyboard or drag on the right side of the **number~** marked "Amplitude". This sets the value of the signal being sent out the **number~** object's left outlet. The signal is connected to the right inlet of two ***~** objects, to control the amplitude of the signal sent to the **ezdac~**.



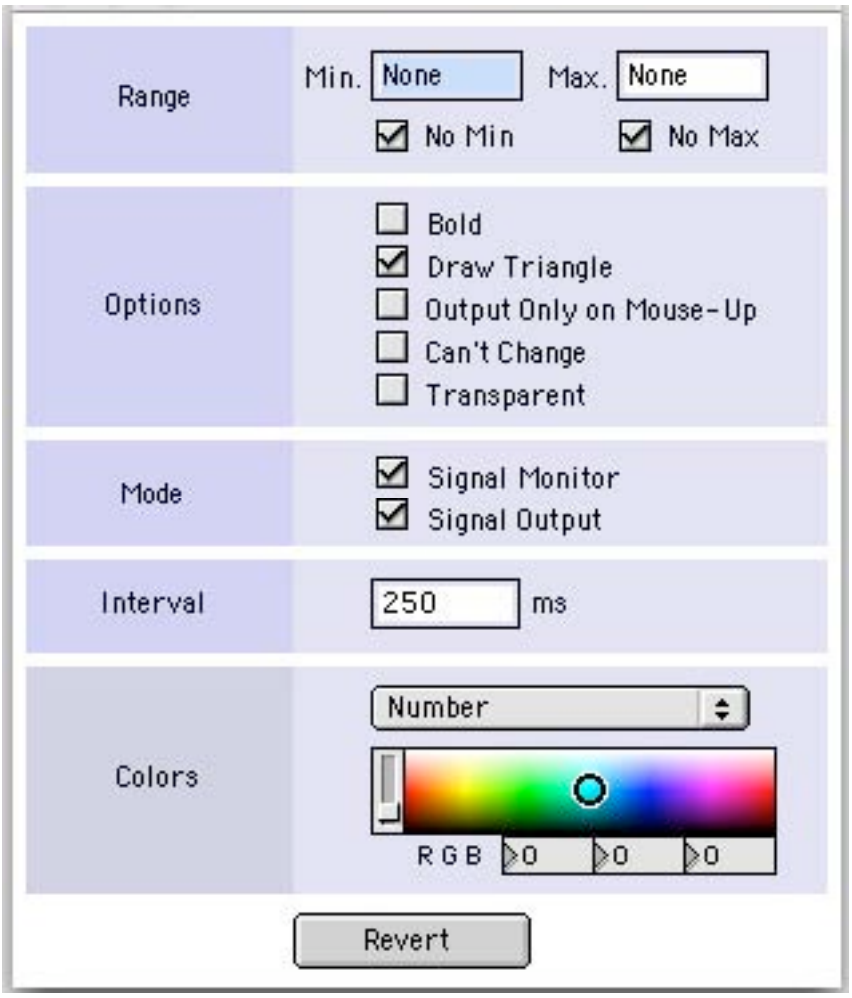
float input to number~ sets the value of the signal sent out the left outlet

A **number~** object simultaneously converts any signal it receives into floats sent out the right outlet, and converts any float it receives into a signal sent out the left outlet. Although it can perform both tasks at the same time, it can only display one value at a time. The value displayed by **number~** depends on which *display mode* it is in. When a small waveform appears in the left part of the **number~**, it is in *Signal Monitor Mode*, and shows the value of the signal coming in the left inlet. When a small arrow appears in the left part of **number~**, it is in *Signal Output Mode*, and shows the value of the signal going out the left outlet.



The two display modes of number~

You can restrict **number~** to one display mode or the other by selecting the object in an unlocked Patcher and choosing **Get Info…** from the Object menu.



Allowed display modes can be chosen in the number~ Inspector

At least one display mode must be checked. By default, both display modes are allowed, as shown in the above example. If both display modes are allowed, you can switch from one display mode to the other in a locked Patcher by clicking on the left side of the **number~**. The output of **number~** continues regardless of what display mode it's in.

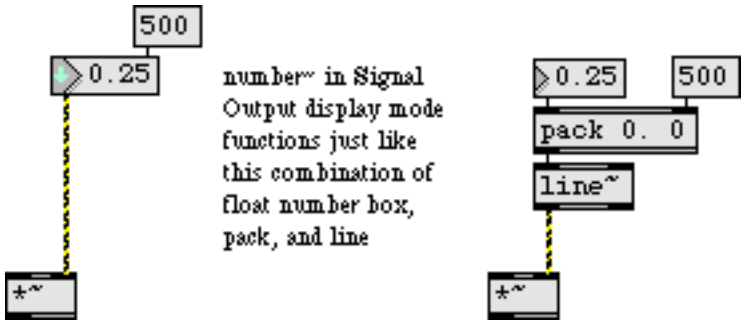
In the tutorial patch you can see the two display modes of **number~**. The **number~** objects at the top of the Patcher window are in *Signal Monitor Mode* because we are using them to show the value of the incoming signal. The "Amplitude" **number~** is in *Signal Output Mode* because we are using it to send a signal and we want to see the value of that signal. (New values can be entered into a **number~** by typing or by dragging with the mouse only

when it is in *Signal Output* display mode.) Since each of these **number~** objects is serving only one function, each has been restricted to only one display mode in the Inspector window.

- Click on the left side of the **number~** objects. They don't change display mode because they have been restricted to one mode or the other in the Inspector window.

Interpolation with number~

The **number~** object has an additional useful feature. It can be made to interpolate between input values to generate a ramp signal much like the **line~** object. If **number~** receives a non-zero number in its right inlet, it uses that number as an amount of time, in milliseconds, to interpolate linearly to the new value whenever it receives a number in the left inlet. This is equivalent to sending a list to **line~**.

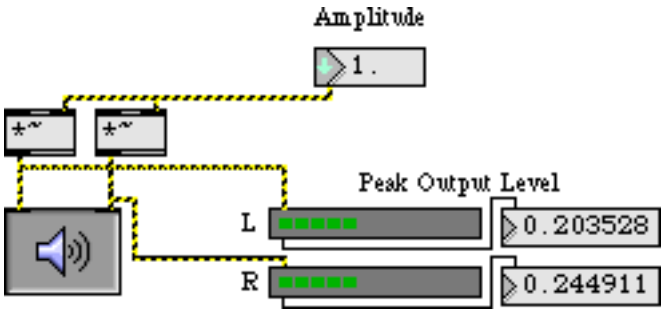


number~ can send a linear ramp signal from its old value to a new value

Unlike **line~**, however, **number~** does not need to receive the interpolation time value more than once; it remembers the interpolation time and uses it for each new number received in the left inlet. This feature is used for the "Amplitude" **number~** so that it won't cause discontinuous changes of amplitude in the output signal.

Peak amplitude: meter~

The **meter~** object periodically displays the peak amplitude it has received since the last display. At the same time it also sends the peak signal value out its outlet as a float. The output value is always a positive number, even if the peak value was negative.



meter~ displays the peak signal amplitude and sends it out as a float

meter~ is useful for observing the peak amplitude of a signal (unlike **number~**, which displays and sends out the *instantaneous* amplitude of the signal). Since **meter~** is intended for audio signals, it expects to receive a signal in the range -1 to 1. If that range is exceeded, **meter~** displays a red "clipping" LED as its maximum.

- If you want to see the clipping display, increase the amplitude of the output signal until it exceeds 1. (Then return it to a desirable level.)

The default interval of time between the display updates of **meter~** is 250 milliseconds, but the display interval can be altered with the interval message. A shorter display interval makes the LED display more accurate, while a longer interval gives you more time to read its visual and numerical output.

- You can try out different display intervals by changing the number in the **number box** marked "Display Interval" in the lower left corner of the Patcher window.

By the way, the display interval of a **number~** object can be set in the same manner (as well as via its Inspector window).

Use a signal to generate Max messages: snapshot~

The **snapshot~** object sends out the current value of a signal, as does the right inlet of **number~**. With **snapshot~**, though, you can turn the output on and off, or request output of a single value by sending it a bang. When you send a non-zero number in the right inlet, **snapshot~** uses that number as a millisecond time interval, and begins periodically

reporting the value of the signal in its left inlet. Sending in a time interval of 0 stops **snapshot~**.

This right half of the tutorial patch shows a simple example of how a signal waveform might be used to generate MIDI data. We'll sample a sub-audio cosine wave to obtain pitch values for MIDI note messages.

- Use the **number~** to set the output amplitude to 0. In the **number box** objects at the top of the patch, set the "Rate" number box to 0.14 and set the "Depth" **number box** to 0.5. Click on the message box 200 to start **snapshot~** reporting signal values every fifth of a second.

Because **snapshot~** is reporting the signal value every fifth of a second, and the period of the **cycle~** object is about 7 seconds, the melody will describe one cycle of a sinusoidal wave every 35 notes. Since the amplitude of the wave is 0.5, the melody will range from 36 to 84 (60±24).

- Experiment with different "Rate" and "Depth" values for the **cycle~**. Since **snapshot~** is sampling at a rate of 5 Hz (once every 200 ms), its Nyquist rate is 2.5 Hz, so that limits the effective frequency of the **cycle~** (any greater frequency will be "folded over"). Click on the 0 **message** box to stop **snapshot~**.

Amplitude modulation

- Set the tremolo depth to 0.5 and the tremolo rate to 4. Increase the output amplitude to a desirable listening level.

The **cycle~** object is modulating the amplitude of the incoming sound with a 4 Hz tremolo.

- Experiment with faster (audio range) rates of modulation to hear the timbral effect of amplitude modulation. To hear ring modulation, set the modulation depth to 1. To remove the modulation effect, simply set the depth to 0.

View a signal excerpt: capture~

The **capture~** object is comparable to the Max object **capture**. It stores many signal values (the most recently received 4096 samples, by default), so that you can view an entire excerpt of a signal as text.

- Set the tremolo depth to 1, and set the tremolo rate to 172. Double-click on the **capture~** object to open a text window containing the last 4096 samples.

This object is useful for seeing precisely what has occurred in a signal over time. (4096 samples is about 93 milliseconds at a sampling rate of 44.1 kHz.) You can type in an argument to specify how many samples you want to view, and **capture~** will store that many samples (assuming there is enough RAM available in Max. There are various arguments and messages for controlling exactly what will be stored by **capture~**. See its description in the MSP Reference Manual for details.

Summary

The **capture~** object stores a short excerpt of a signal to be viewed as text. The **meter~** object periodically displays the peak level of a signal and sends the peak level out its outlet as a float. The **snapshot~** object sends out a float to report the current value of a signal. **snapshot~** reports the signal value once when it receives a bang, and it can also report the signal value periodically if it receives a non-zero interval time in its right inlet.

The **number~** object is like a combination of a float **number box**, **sig~**, and **snapshot~**, all at once. A signal received in its left inlet is sent out the right outlet as a float, as with **snapshot~**. A float or int received in its left inlet sets the value of the signal going out its left outlet, as with **sig~**. Both of these activities can go on at once in the same **number~** object, although **number~** can only *display* one value at a time. When **number~** is in *Signal Monitor Mode*, it displays the value of the incoming signal. When **number~** is in *Signal Output Mode*, it displays the value of the outgoing signal.

number~ can also function as a signal ramp generator, like the **line~** object. If a non-zero number has been received in the right inlet (representing interpolation time in milliseconds), whenever **number~** receives a float, its output signal interpolates linearly between the old and new values.

These objects (along with a few others such as **sig~**, **peek~** and **avg~**) comprise the primary links between MSP and Max. They convert signals to numerical Max messages, or vice versa.

See Also

capture~	Store a signal to view as text
meter~	Visual peak level indicator
number~	Signal monitor and constant generator
snapshot~	Convert signal values to numbers

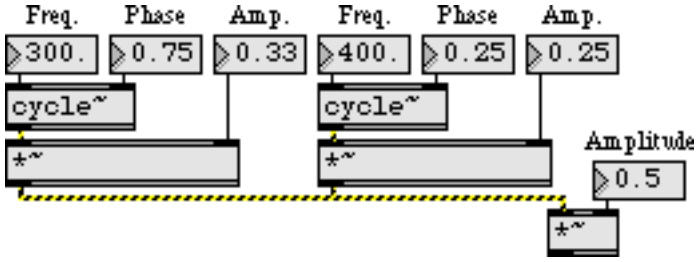
Graph of a signal over time

There are times when seeing a picture of a signal is instructive. The **scope~** object depicts the signal it receives, in the manner of an analog oscilloscope, as a graph of amplitude over time.

There are two problems **scope~** must consider when plotting a graph of a signal in real time. First of all, in order for your eye to follow a time-varying signal, an excerpt of the signal must be captured and displayed for a certain period of time (long enough for you really to see it). Therefore, the graph must be displayed periodically, and will always lag a bit behind what you hear. Second, there aren't enough pixels on the screen for you to see a plot of every single sample (at least, not without the display being updated at blinding speed), so **scope~** has to use a single pixel to summarize many samples.

A patch to view different waveforms

This tutorial shows how to get a useful display of a signal. The patch adds four cosine oscillators to create a variety of waveforms, and displays them in **scope~**. The frequency, phase, and amplitude of each sinusoid is set independently, and the over-all amplitude of the sum of the oscillators is scaled with an additional ***~** object. The settings for each waveform are stored in a **preset** object.



Additive synthesis can be used to create a variety of complex waveforms

- Click on the first preset in the **preset** object.

When audio is turned on, the **dspstate~** object sends the current sampling rate out its middle outlet. This is divided by the number of pixels per display buffer (the display buffer is where the display points are held before they're shown on the screen), and the result is the number of signal samples per display point (samples per pixel). This number is sent in the left inlet of **scope~** to tell it how many samples to assign to each display pixel. The default number of pixels per display buffer is 128, so by this method each display will consist of exactly one second of signal. In other words, once per second **scope~** displays

the second that has just passed. We have stretched the **scope~** (using its grow handle) to be 256 pixels wide—twice its default width—in order to provide a better view.

On the next page we will describe the different waveforms created by the oscillators.

- One by one, click on the different presets to see different waveforms displayed in the **scope~**. The first eight waves are at the sub-audio frequency of 1 Hz to allow you to see a single cycle of the waveform, so the signal is not sent to the **dac~** until the ninth preset is recalled.

Preset 1. A 1 Hz cosine wave.

Preset 2. A 1 Hz sine wave. (A cosine wave with a phase offset of $^3/_4$ cycle.)

Preset 3. A 1 Hz cosine wave plus a 2 Hz cosine wave (i.e. octaves).

Preset 4. Four octaves: cosine waves of equal amplitude at 1, 2, 4, and 8 Hz.

Preset 5. A band-limited square wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a square wave. (Although the amplitudes of the oscillators are only shown to two decimal places, they are actually stored in the preset with six decimal place precision.)

Preset 6. A band-limited sawtooth wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a sawtooth wave.

Preset 7. A band-limited triangle wave. The four oscillators produce four sine waves with the correct frequencies and amplitudes to represent the first four partials of a triangle wave (which, it appears, is actually not very triangular without its upper partials).

Preset 8. This wave has the same frequencies and amplitudes as the band-limited square wave, but has arbitrarily chosen phase offsets for the four components. This shows what a profound effect the phase of components can have on the appearance of a waveform, even though its effect on the sound of a waveform is usually very slight.

Preset 9. A 32 Hz sinusoid plus a 36 Hz sinusoid (one-half cycle out of phase for the sake of the appearance in the **scope~**). The result is interference causing beating at the difference frequency of 4 Hz.

Preset 10. Combined sinusoids at 200, 201, and 204 Hz, producing beats at 1, 3, and 4 Hz.

Preset 11. Although the frequencies are all displayed as 200 Hz, they are actually 200, 200.25, 200.667, and 200.8. This produces a complicated interference pattern of six

different sub-audio beat frequencies, a pattern which only repeats precisely every minute. We have set the number of samples per pixel much lower, so each display represents about 50 ms. This allows you to see about 10 wave cycles per display.

Preset 12. Octaves at 100, 200, and 400 Hz (with different phase offsets), plus one oscillator at 401 Hz creating beats at 1 Hz.

Preset 13. A cluster of equal-tempered semitones. The dissonance of these intervals is perhaps all the more pronounced when pure tones are used. Each display shows about 100 ms of sound.

Preset 14. A just-tuned dominant seventh chord; these are the 4th, 5th, 6th, and 7th harmonics of a common fundamental, so their sum has a periodicity of 100 Hz, two octaves below the chord itself.

Preset 15. Total phase cancellation. A sinusoid is added to a copy of itself 180° out of phase.

Preset 16. All oscillators off.

Summary

The **scope~** object gives an oscilloscope view of a signal, graphing amplitude over time. Because **scope~** needs to collect the samples before displaying them, and because the user needs a certain period of time to view the signal, the display always lags behind the signal by one display period. A display period (in seconds) is determined by the number of pixels per display buffer, times the number of samples per pixel, divided by the signal sampling rate. You can control those first two values by sending integer values in the inlets of **scope~**. The sampling rate of MSP can be obtained with the **dspstate~** object.

See Also

dspstate~	Report current DSP setting
scope~	Signal oscilloscope

Fourier's theorem

The French mathematician Joseph Fourier demonstrated that any periodic wave can be expressed as the sum of harmonically related sinusoids, each with its own amplitude and phase. Given a digital representation of a periodic wave, one can employ a formula known as the discrete Fourier transform (DFT) to calculate the frequency, phase, and amplitude of its sinusoidal components. Essentially, the DFT *transforms* a time-domain representation of a sound wave into a frequency- domain spectrum.

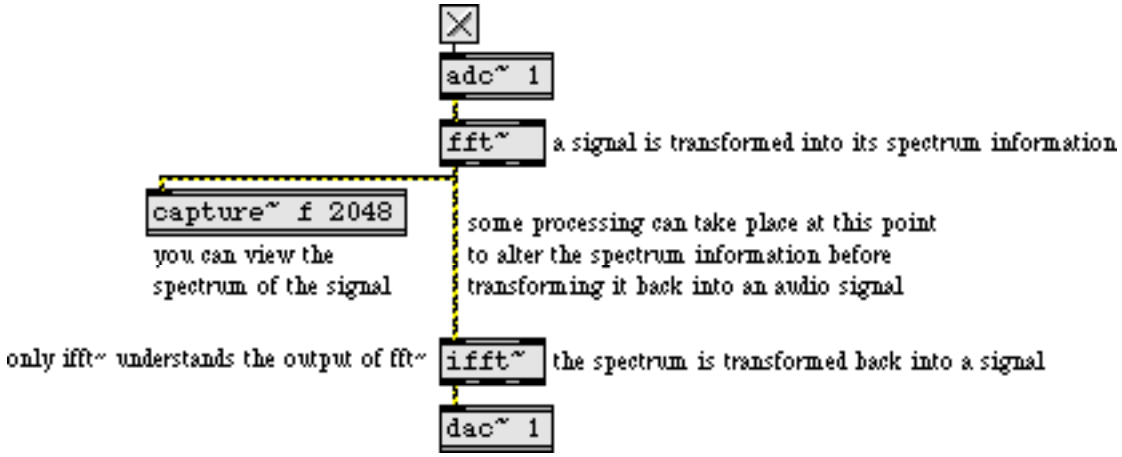
Typically the Fourier transform is used on a small "slice" of time, which ideally is equal to exactly one cycle of the wave being analyzed. To perform this operation on "real world" sounds—which are almost invariably *not* strictly periodic, and which may be of unknown frequency—one can perform the DFT on consecutive time slices to get a sense of how the spectrum changes over time.

If the number of digital samples in each time slice is a power of 2, one can use a faster version of the DFT known as the fast Fourier transform (FFT). The formula for the FFT is encapsulated in the **fft~** object. The mathematics of the Fourier transform are well beyond the scope of this manual, but this tutorial chapter will demonstrate how to use the **fft~** object for signal analysis.

Spectrum of a signal: fft~

fft~ receives a signal in its inlet. For each slice of time it receives (512 samples long by default) it sends out a signal of the same length listing the amount of energy in each frequency region. The signal that comes out of **fft~** is not anything you're likely to want to listen to. Rather, it's a list of relative amplitudes of 512 different frequency bands in the received signal. This "list" happens to be exactly the same length as the samples received in each time slice, so it comes out at the same rate as the signal comes in. The signal coming out of **fft~** is a frequency-domain analysis of the samples it received in the previous time slice.

Although the transform comes out of **fft~** in the form of a signal, it is not a time-domain signal. The only object that "understands" this special signal is the **ifft~** object, which performs an *inverse* FFT on the spectrum and transforms it back into a time-domain waveform.

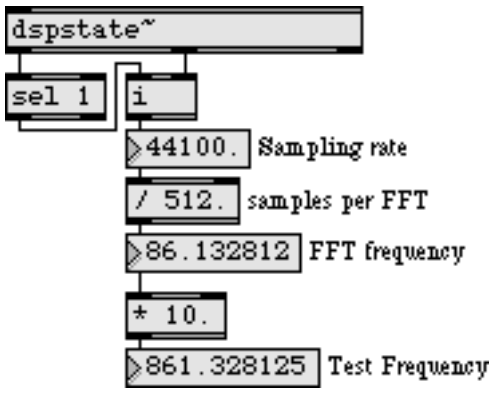


The signal coming out of fft~ is spectral information, not a time-domain signal

With the **capture~** object you can grab some of the output of **fft~** and examine the frequency analysis of a signal.

- Click on one of the **ezdac~** objects to turn audio on.

When audio is turned on, **dspstate~** sends the MSP sampling rate out its middle outlet. We use this number to calculate a frequency that has a period of exactly 512 samples. This is the fundamental frequency of the FFT itself. If we send a wave of that frequency into **fft~**, each time slice would contain exactly one cycle of the waveform. We will actually use a cosine wave at ten times that frequency as the test tone for our analysis, as shown below.



The test tone is at 10 times the base frequency of the FFT time slice

The upper left corner of the Patcher window shows a very simple use of **fft~**. The analysis is stored in a **capture~** object, and an **ifft~** object transforms the analysis back into an audio signal. (Ordinarily you would not transform and inverse-transform an audio signal for no reason like this. The **ifft~** is used in this patch simply to demonstrate that the analysis-resynthesis process works.)

- Click on the **toggle** in the upper left part of the patch to hear the resynthesized sound. Click on the **toggle** again to close the **gate~**. Now double-click on the **capture~** object in that part of the patch to see the analysis performed by **fft~**.

In the **capture~** text window, the first 512 numbers are all 0.0000. That is the output of **fft~** during the first time slice of its analysis. Remember, the analysis it sends out is always of the previous time slice. When audio was first turned on, there was no previous audio, so the **fft~** object's analysis shows no signal.

- Scroll past the first 512 numbers. (The numbers in the **capture~** object's text window are grouped in blocks, so if your signal vector size is 256 you will have two groups of numbers that are all 0.0000.) Look at the second time slice of 512 numbers.

Each of the 512 numbers represents a harmonic of the FFT frequency itself, starting at the 0th harmonic (0 Hz). The analysis shows energy in the eleventh number, which represents the 10th harmonic of the FFT, $^{10}/_{512}$ the sampling rate—precisely our test frequency. (The analysis also shows energy at the 10th number from the end, which represents $^{502}/_{512}$ the sampling rate. This frequency exceeds the Nyquist rate and is actually equivalent to $-^{10}/_{512}$ of the sampling rate.

> **Technical detail:** An FFT divides the entire available frequency range into as many bands (regions) as there are samples in each time slice. Therefore, each set of 512 numbers coming out of **fft~** represents 512 divisions of the frequency range from 0 to the sampling rate. The first number represents the energy at 0 Hz, the second number represents the energy at $^{1}/_{512}$ the sampling rate, the third number represents the energy at $^{2}/_{512}$ the sampling rate, and so on.
>
> Note that once we reach the Nyquist rate on the 257th number ($^{256}/_{512}$ of the sampling rate), all numbers after that are *folded back* down from the Nyquist rate. Another way to think of this is that these numbers represent negative frequencies that are now ascending from the (negative) Nyquist rate. Thus, the 258th number is the energy at the Nyquist rate *minus* $^{1}/_{512}$ of the sampling rate (which could also be thought of as $-^{255}/_{512}$ the sampling rate). In our example, we see energy in the 11th frequency region ($^{10}/_{512}$ the sampling rate) and the 503rd frequency region ($-^{256}/_{512} - -^{246}/_{512} = -^{10}/_{512}$ the sampling rate).

It appears that **fft~** has correctly analyzed the signal. There's just one problem...

Practical problems of the FFT

The FFT assumes that the samples being analyzed comprise one cycle of a periodic wave. In our example, the cosine wave was the 10th harmonic of the FFT's fundamental frequency, so it worked fine. In most cases, though, the 512 samples of the FFT will not be precisely one cycle of the wave. When that happens, the FFT still analyzes the 512 samples as if they were one cycle of a waveform, and reports the spectrum of that wave. Such an analysis will contain many spurious frequencies not actually present in the signal.

- Close the text window of **capture~**. With the audio still on, set the "Test Frequency" **number box** to 1000. This also triggers the clear message in the upper left corner of the patch to empty the **capture~** object of its prior contents. Double-click once again on **capture~**, and scroll ahead in the text window to see its new contents.

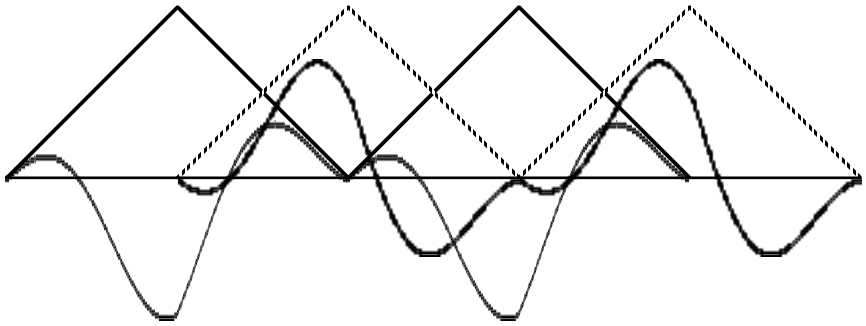
The analysis of the 1000 Hz tone does indeed show greater energy at 1000 Hz—in the 12th and 13th frequency regions if your MSP sampling rate is 44,100 Hz—but it also shows energy in virtually every other region. That's because the waveform it analyzed is no longer a sinusoid. (An exact number of cycles does not fit precisely into the 512 samples.) All the other energy shown in this FFT is an artifact of the "incorrect" interpretation of those 512 samples as one period of the correct waveform.

To resolve this problem, we can try to "taper" the ends of each time slice by applying an amplitude envelope to it, and use overlapping time slices to compensate for the use of the envelope.

Overlapping FFTs

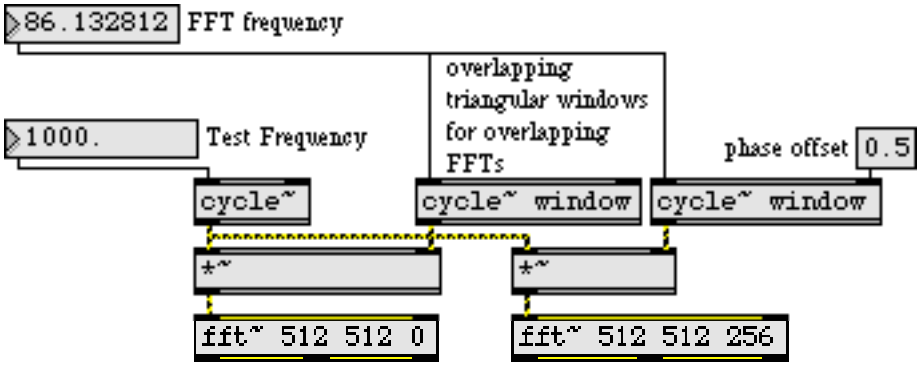
The lower right portion of the tutorial patch takes this approach of using overlapping time slices, and applies a triangular amplitude envelope to each slice before analyzing it. (Other shapes of amplitude envelope are often used for this process.

The triangular window is simple and quite effective.) In this way, the **fft~** object is viewing each time slice through a triangular window which tapers its ends down, thus filtering out many of the false frequencies that would be introduced by discontinuities.



Overlapping triangular windows (envelopes) applied to a 100 Hz cosine wave

To accomplish this windowing and overlapping of time slices, we must perform two FFTs, one of which is offset 256 samples later than the other. (Note that this part of the patch will only work if your current MSP Signal Vector size is 256 or less, since **fft~** can only be offset by a multiple of the vector size.) The offset of an FFT can be given as a (third) typed-in argument to **fft~**, as is done for the **fft~** object on the right. This results in overlapping time slices.



One FFT is taken 256 samples later than the other

The windowing is achieved by multiplying the signal by a triangular waveform (stored in the **buffer~** object) which recurs at the same frequency as the FFT—once every 512 samples. The window is offset by $^1/_2$ cycle (256 samples) for the second **fft~**.

- Double-click on the **buffer~** object to view its contents. Then close the **buffer~** window and double-click on the **capture~** object that contains the FFT of the windowed signal. Scroll past the first block or two of numbers until you see the FFT analysis of the windowed 1000 Hz tone.

As with the unwindowed FFT, the energy is greatest around 1000 Hz, but here the (spurious) energy in all the other frequency regions is greatly reduced by comparison with the unwindowed version.

Signal processing using the FFT

In this patch we have used the **fft~** object to view and analyze a signal, and to demonstrate the effectiveness of windowing the signal and using overlapping FFTs. However, one could also write a patch that alters the values in the signal coming out of **fft~**, then sends the altered analysis to **ifft~** for resynthesis. An implementation of this frequency-domain filtering scheme will be seen in a future tutorial.

Summary

The fast Fourier transform (FFT) is an algorithm for transforming a time-domain digital signal into a frequency-domain representation of the relative amplitude of different frequency regions in the signal. An FFT is computed using a relatively small excerpt of a signal, usually a slice of time 512 or 1024 samples long. To analyze a longer signal, one performs multiple FFTs using consecutive (or overlapping) time slices.

The **fft~** object performs an FFT on the signal it receives, and sends out (also in the form of a signal) a frequency-domain analysis of the received signal. The only object that understands the output of **fft~** is **ifft~** which performs an inverse FFT to synthesize a time-domain signal based on the frequency-domain information. One could alter the signal as it goes from **fft~** to **ifft~**, in order to change the spectrum.

The FFT only works perfectly when analyzing exactly one cycle (or exactly an integer number of cycles) of a tone. To reduce the artifacts produced when this is not the case, one can window the signal being analyzed by applying an amplitude envelope to taper the ends of each time slice. The amplitude envelope can be applied by multiplying the signal by using a **cycle~** object to read a windowing function from a **buffer~** repeatedly at the same rate as the FFT itself (i.e., once per time slice).

See Also

buffer~	Store audio samples
capture~	Store a signal to view as text
fft~	Fast Fourier transform
ifft~	Inverse Fast Fourier transform

Tutorial 26: Frequency Domain Signal Processing with pfft~

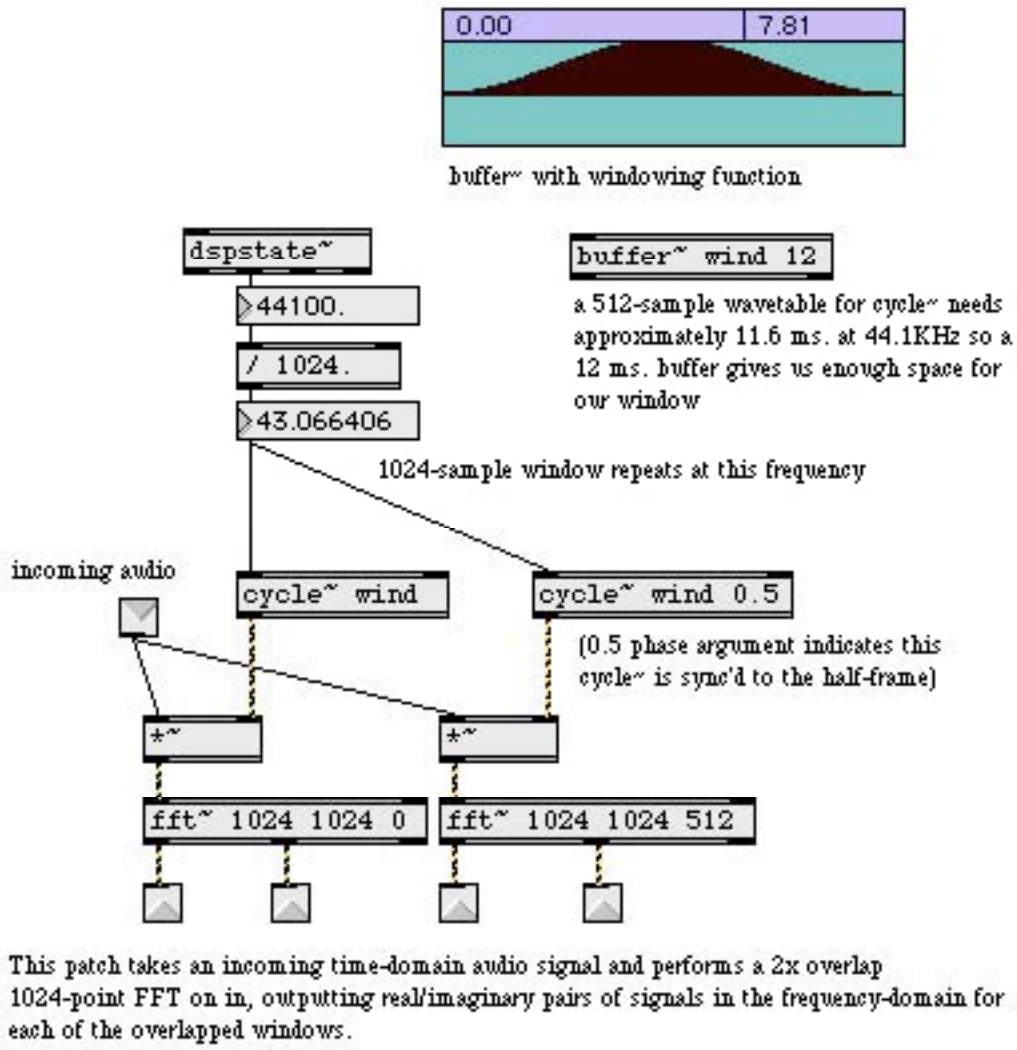
Working in the Frequency Domain

Most digital signal processing of audio occurs in what is known as the time domain. As the other MSP tutorials show you, many of the most common processes for manipulating audio consist of varying samples (or groups of samples) in amplitude (ring modulation, waveshaping, distortion) or time (filters and delays). The Fast Fourier Transform (FFT) allows you to translate audio data from the time domain into the frequency domain, where you can directly manipulate the spectrum of a sound (the component frequencies of a slice of audio).

As we have seen in Tutorial 25, the MSP objects **fft~** and **ifft~** allow you to transform signals into and out of the frequency domain. The **fft~** object takes a a group of samples (commonly called a frame) and transforms them into pairs of real and imaginary numbers representing the amplitude and phase of as many frequencies as there are samples in the frame. These are usually referred to as *bins* or *frequency bins.* (We will see later that the real and imaginary numbers are not themselves the amplitude and phase, but that the amplitude and phase can be derived from them.) The **ifft~** object performs the inverse operation, taking frames of frequency-domain samples and converting them back into a time domain audio signal that you can listen to or process further. The number of samples in the frame is called the *FFT size* (or sometimes *FFT point size*). It must be a power of 2 such as 512, 1024 or 2048 (to give a few commonly used values).

One of the shortcomings of the **fft~** and **ifft~** objects is that they work on successive frames of samples without doing any overlapping or cross-fading between them. For most practical musical uses of these objects, we usually need to construct such an overlap and crossfade system around them. There are several reasons for needing to create such a system when using the Fourier transform to process sound. In FFT analysis there is always a trade-off between frequency resolution and timing resolution. For example, if your FFT size is 2048 samples long, the FFT analysis gives you 2048 equally-spaced frequency bins from 0 Hz. up to the sampling frequency (only 1024 of these bins are of any use; see Tutorial 25 for details). However, any timing resolution that occurs within those 2048 samples will be lost in the analysis, since all temporal changes are lumped together in a single FFT frame. In addition, if you modify the spectral data after the FFT analysis and before the IFFT resynthesis you can no longer guarantee that the time domain signal output by the IFFT will match up in successive frames. If the output time domain vectors don't fit together you will get clicks in your output signal. By designing a *windowing function* in MSP (see below), you can compensate for these artifacts by having successive frames cross-fade into each other as they overlap. While this will not compensate for the loss of time resolution, the overlapping of analysis data will help to eliminate the clicks and pops that occurs at the edges of an IFFT frame after resynthesis.

This analysis/resynthesis scheme (using overlapping, windowed slices of time with the FFT and IFFT) is usually referred to as a *Short Term* (or *Short Time*) *Fourier Transform* (STFT). An STFT can be designed in MSP by creating a patch that uses one or more pairs of **fft~/ifft~** objects with the input signal "windowed" into and out of the frequency domain. While this approach works fairly well, it is somewhat cumbersome to program since every operation performed in the frequency domain needs to be duplicated correctly for each **fft~/ifft~** pair. The following subpatch illustrates how one would window incoming FFT data in this manner:



How to properly window audio for use with the fft~ object

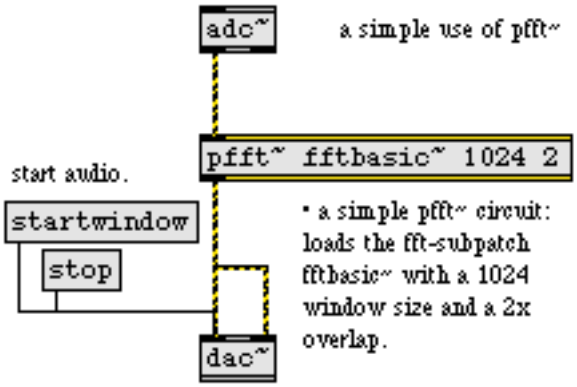
In addition to the fact that this approach can often be a challenge to program, there is also the difficulty of generalizing the patch for multiple combinations of FFT size and overlap. Since the arguments to **fft~/ifft~** for FFT frame size and overlap can't be changed,

multiple hand-tweaked versions of each subpatch must be created for different situations. For example, a percussive sound would necessitate an analysis with at least four overlaps, while a reasonably static, harmonically rich sound would call for a very large FFT size.

> **Technical detail:** Time vs. Frequency Resolution
>
> The FFT size we use provides us with a tradeoff; because the Fourier transform mathematically converts a small slice of time into a frozen "snapshot" representing its spectrum, you might first think that it would be beneficial to use small FFT sizes in order to avoid grouping temporal changes together in one analysis spectrum. While this is true, an FFT size with a smaller number of points also means that our spectrum will have a smaller number of frequency bins, which means that the frequency resolution will be lower. Smaller FFT sizes result in better temporal resolution, but at the cost of lower frequency resolution when the sound is modified in the frequency domain and resynthesized. Conversely, larger FFT sizes give us finer frequency detail, but tend to "smear" temporal changes in the sound. In practice, we therefore need to choose an appropriate FFT size based on the kind of sound we want to process.

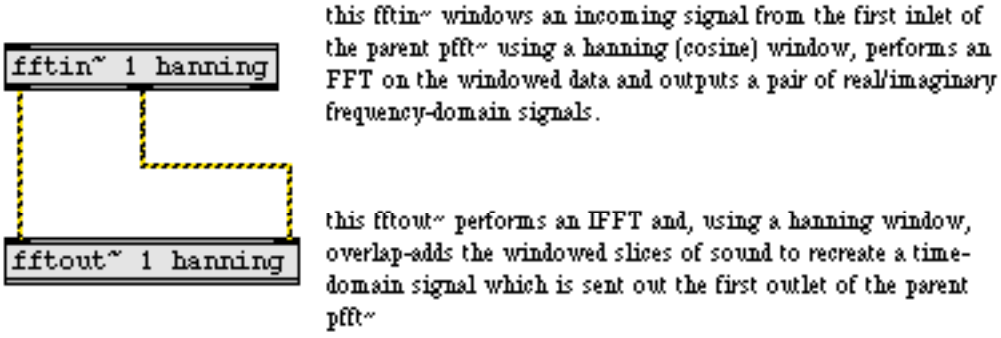
The **pfft~** object addresses many of the shortcomings of the "old" **fft~** and **ifft~** objects, allowing you to create and load special "spectral subpatches" that manipulate frequency-domain signal data independently of windowing, overlap and FFT size. A single subpatch can therefore be suitable for multiple applications. Furthermore, the **pfft~** object manages the overlapping of FFT frames, handles the windowing functions for you, and eliminates the redundant mirrored data in the spectrum, making it both more convenient to use and more efficient than the traditional **fft~** and **ifft~** objects.

The **pfft~** object takes as its argument the name of a specially designed subpatch containing the **fftin~** and **fftout~** objects (which will be discussed below), a number for the FFT size in samples, and a number for the overlap factor (these must both be integers which are a power of 2):



A simple use of pfft~.

The **pfft~** subpatch fftbasic~ referenced above might look something like this:



The fftbasic~ subpatch used in the previous example

The **fftbasic~** subpatch shown above takes a signal input, performs an FFT on that signal with a Hanning window (see below), and performs an IFFT on the FFT'd signal, also with a Hanning window. The **pfft~** object communicates with its sub-patch using special objects for inlets and outlets. The **fftin~** object receives a time-domain signal from its parent patch and transforms it via an FFT into the frequency domain. This time-domain

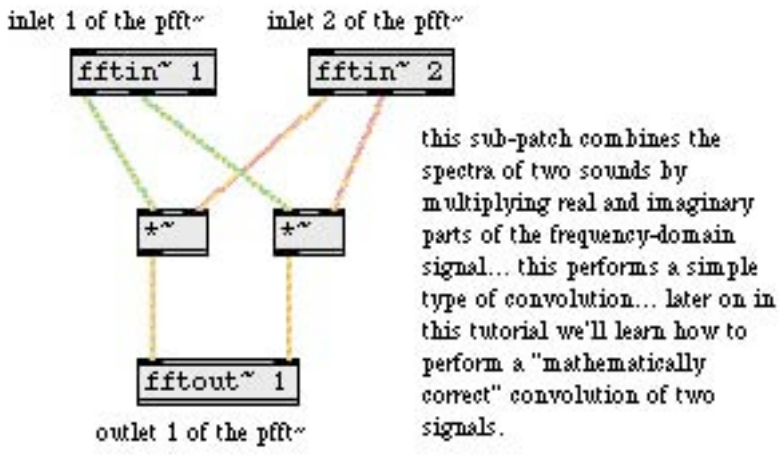
signal has already been converted, by the **pfft~** object into a sequence of frames which overlap in time, and the signal that **fftin~** outputs into the spectral subpatch represents the spectrum of each of these incoming frames.

> **Technical detail:** The signal vector size inside the spectral subpatch is equal to half the FFT size specified as an argument to the **pfft~**. Here's the reason why: for efficiency's sake, **fftin~** and **fftout~** perform what is known as a *real FFT*, which is faster than the traditional *complex FFT* used by **fft~** and **ifft~**. This is possible because the time-domain signals we transform have no imaginary part (or at least they have an imaginary part which is equal to zero). A real FFT is a clever mathematical trick which re-arranges the real-only time-domain input to the FFT as real and imaginary parts of a complex FFT that is half the size of our real FFT. The result of this FFT is then re-arranged into a complex spectrum representing half (from 0Hz to half the sampling rate) of our original real-only signal. The smaller FFT size means it is more efficient for our computer's processor, and, because a complex FFT produces a mirrored spectrum of which only half is really useful to us, the real FFT contains all the data we need to define and subsequently manipulate the signal's spectrum.

The **fftout~** object does the reverse, accepting frequency domain signals, converting them back into a time domain signal, and passing it via an outlet to the parent patch. Both objects take a numbered argument (to specify the inlet or outlet number), and a symbol specifying the window function to use. The available window functions are Hanning (the default if none is specified), Hamming, Blackman, Triangle, and Square. The nofft argument to **fftin~** and **fftout~** creates a generic signal inlet or outlet for control data where no FFT/IFFT or windowing is performed. In addition, the symbol can be the name of a **buffer~** object which holds a custom windowing function. Different window functions have different bandwidths and stopband depths for each channel (or bin, as it is sometimes called) of the FFT. A good reference on FFT analysis will help you select a window based on the sound you are trying to analyze and what you want to do with it. We recommend *The Computer Music Tutorial* by Curtis Roads or *Computer Music* by Charles Dodge and Thomas Jerse.

For testing and debugging purposes, there is a handy nofft argument to **fftin~** and **fftout~** which allows the overlapping time-domain frames to and from the **pfft~** to be passed directly to and from the subpatch without applying a window function nor performing a Fourier transform. In this case (because the signal vector size of the spectral subpatch is half the FFT size), the time- domain signal is split between the real and imaginary outlets of the **fftin~** and **fftout~** objects, which may be rather inconvenient when using an overlap of 4 or more. Although the nofft option can be used to send signal data from the parent patch into the spectral subpatch and may be useful for debugging subpatches, it is not recommended for most practical uses of **pfft~**.

A more complicated **pfft~** subpatch might look something like this:



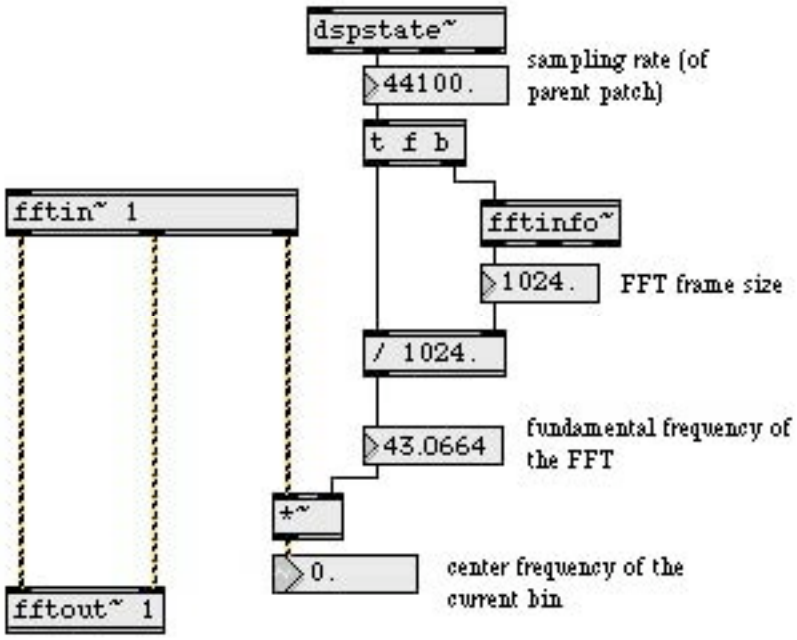
A simple type of spectral convolution

This subpatch takes two signal inputs (which would appear as inlets in the parent **pfft~** object), converts them into the frequency domain, multiplies the real signals with one another and multiplies the imaginary signals with one another and outputs the result to an **fftout~** object that converts the frequency domain data into a time domain signal. Multiplication in the frequency domain is called *convolution*, and is the basic signal processing procedure used in cross synthesis (morphing one sound into another). The result of this algorithm is that frequencies from the two analyses with larger values will reinforce one another, whereas weaker frequency values from one analysis will diminish or cancel the value from the other, whether strong or weak. Frequency content that the two incoming signals share will be retained, therefore, and disparate frequency content (i.e. a pitch that exists in one signal and not the other) will be attenuated or eliminated. This example is not a "true" convolution, however, as the multiplication of *complex numbers* (see below) is not as straightforward as the multiplication performed in this example. We'll learn a couple ways of making a "correct" convolution patch later in this tutorial.

You have probably already noticed that there are always two signals to connect when connecting **fftin~** and **fftout~**, as well as when processing the spectra in-between them. This is because the FFT algorithm produces *complex numbers* — numbers that contain a real and an imaginary part. The real part is sent out the leftmost outlet of **fftin~**, and the imaginary part is sent out its second outlet. The two inlets of **fftout~** also correspond to real and imaginary, respectively. The easiest way to understand complex numbers is to think of them as representing a point on a 2-dimensional plane, where the real part represents the X-axis (horizontal distance from zero), and the imaginary part represents

the Y-axis (vertical distance from zero). We'll learn more about what we can do with the real and imaginary parts of the complex numbers later on in this tutorial.

The **fftin~** object has a third outlet that puts out a stream of samples corresponding to the current frequency bin index whose data is being sent out the first two outlets (this is analogous to the third outlet of the **fft~** and **ifft~** objects discussed in Tutorial 25). For **fftin~**, this outlet outputs a number from 0 to half the FFT size minus 1. You can convert these values into frequency values (representing the "center" frequency of each bin) by multiplying the signal (called the sync signal) by the base frequency, or fundamental, of the FFT. The fundamental of the FFT is the lowest frequency that the FFT can analyze, and is inversely proportional to the size of the FFT (i.e. larger FFT sizes yield lower base frequencies). The exact fundamental of the FFT can be obtained by dividing the FFT frame size into the sampling rate. The **fftinfo~** object, when placed into a **pfft~** subpatch, will give you the FFT frame size, the FFT half-frame size (i.e. the number of bins actually used inside the **pfft~** subpatch), and the FFT hop size (the number of samples of overlap between the windowed frames). You can use this in conjunction with the **dspstate~** object or the **adstatus** object with the sr (sampling rate) argument to obtain the base frequency of the FFT:

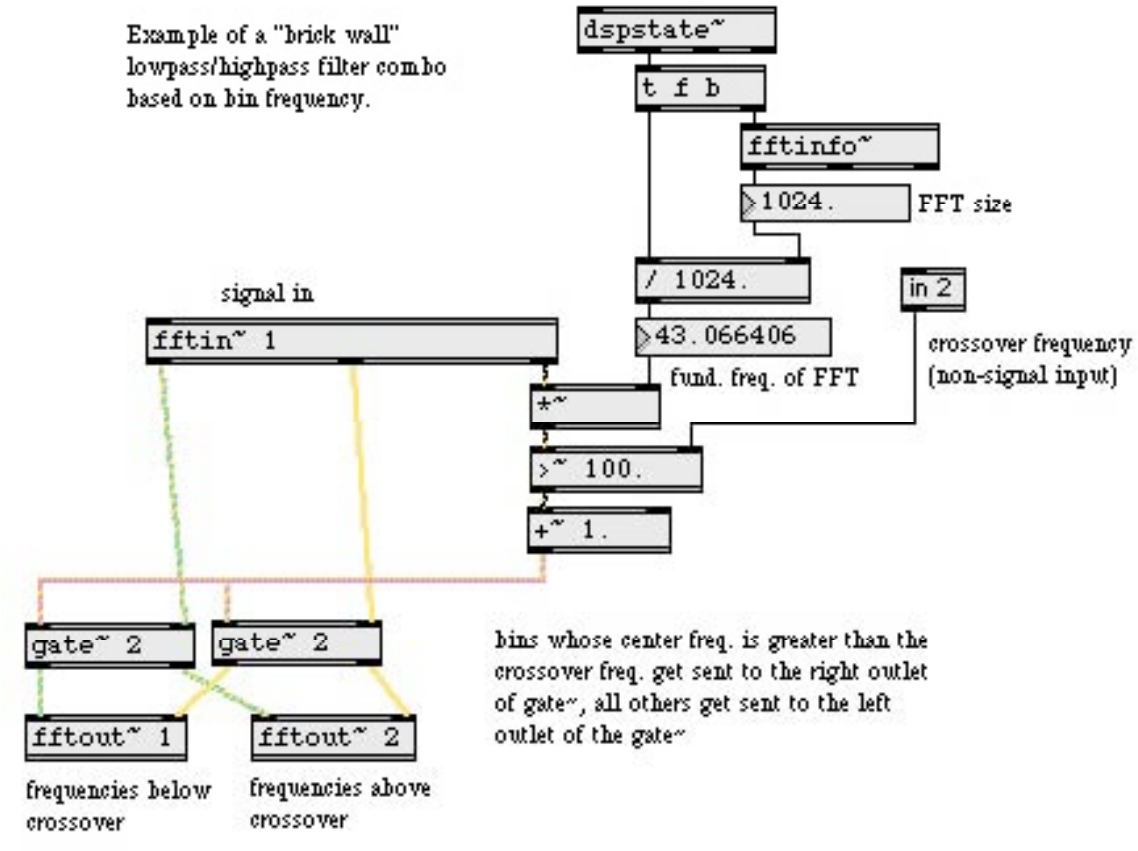


Finding the center frequency of the current analysis bin.

Note that in the above example the **number~** object is used for the purposes of demonstration only in this tutorial. When DSP is turned on, the number displayed in the signal number box will not appear to change because the signal number box by default

displays the first sample in the signal vector, which in this case will always be 0. To see the center frequency values, you will need to use the **capture~** object or record this signal into a **buffer~**.

Once you know the frequency of the bins being streamed out of **fftin~**, you can perform operations on the FFT data based on frequency. For example:

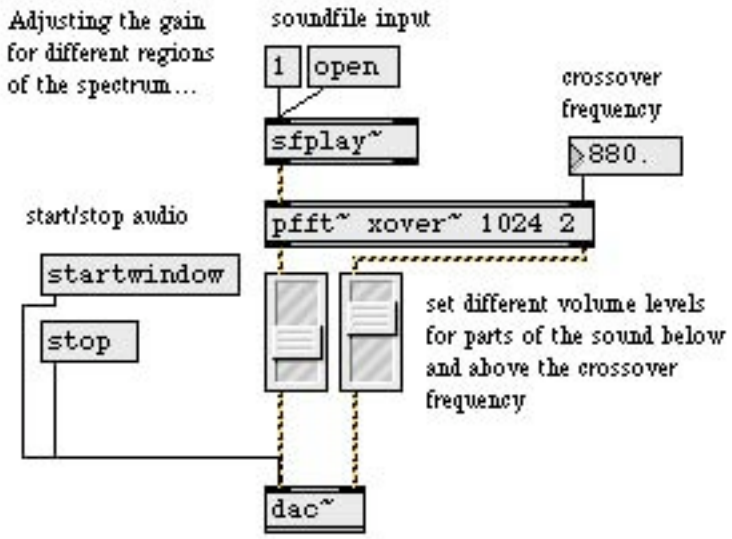


A simple spectral crossover.

The above **pfft~** subpatch, called **xover~**, takes an input signal and sends the analysis data to one of two **fftout~** objects based on a crossover frequency. The crossover frequency is sent to the **pfft~** subpatch by using the in object, which passes max messages through from the parent patch via the **pfft~** object's right inlet. The center frequency of the current bin — determined by the sync outlet in conjunction with **fftinfo~** and **dspstate~** as we mentioned above — is compared with the crossover frequency.

The result of this comparison flips a gate that sends the FFT data to one of the two **fftout~** objects: the part of the spectrum that is lower in pitch than the crossover frequency is sent

out the left outlet of the **pfft~** and the part that is higher than the crossover frequency is sent out the right. Here is how this subpatcher might be used with **pfft~** in a patch



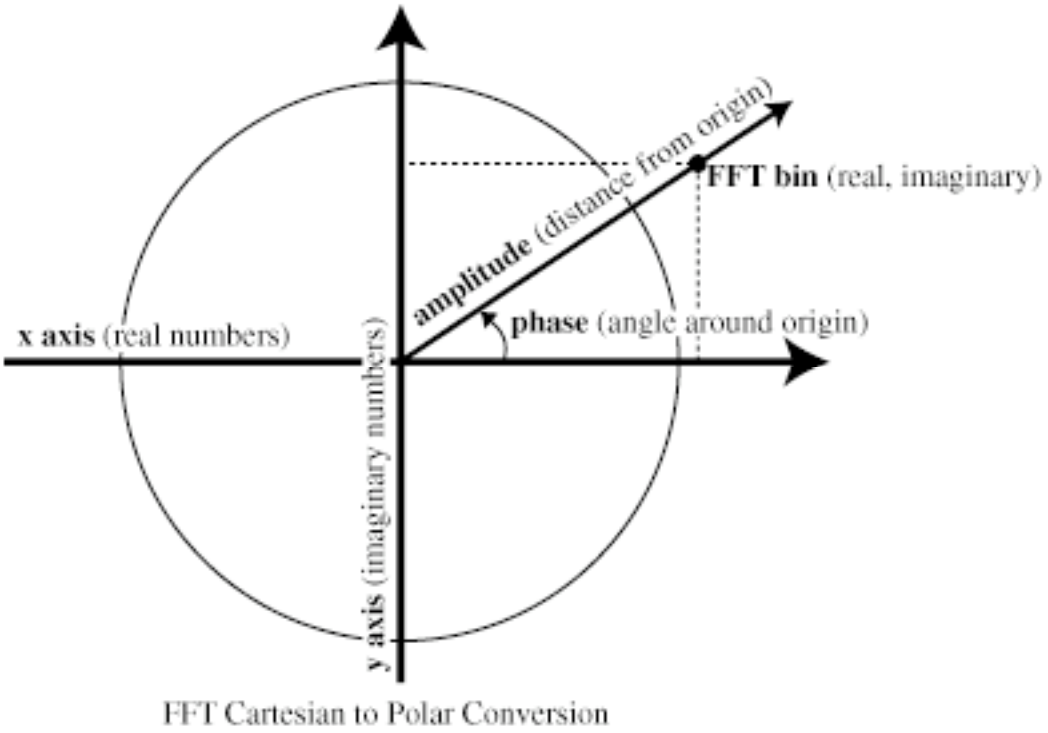
:

One way of using the xover~ subpatch

Note that we can send integers, floats, and any other Max message to and from a subpatch loaded by **pfft~** by using the **in** and **out** objects. (See *Tutorial 21, Using the* **poly~** *object* for details. Keep in mind, however, that the signal objects **in~** and **out~** currently do not function inside a **pfft~**.)

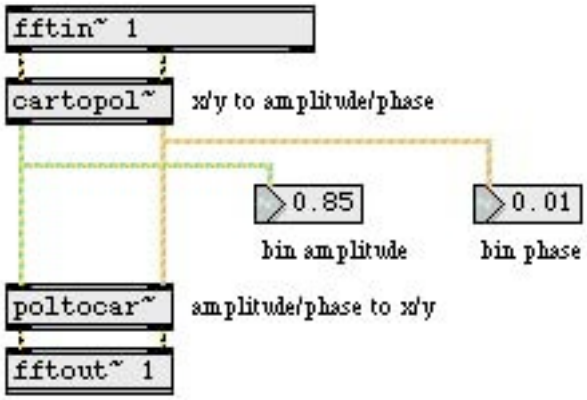
As we have already learned, the first two outlets of **fftin~** put out a stream of real and imaginary numbers for the bin response for each sample of the FFT analysis (similarly, **fftout~** expects these numbers). These are not the amplitude and phase of each bin, but should be thought of instead as pairs of Cartesian coordinates, where *x* is the real part and *y* is the imaginary, representing points on a 2-dimensional plane.

The amplitude and phase of each frequency bin are the polar coordinates of these points, where the distance from the origin is the bin amplitude and the angle around the origin is the bin phase:



Unit-circle diagram showing the relationship of FFT real and imaginary values to amplitude and phase

You can easily convert between real/imaginary pairs and amplitude/phase pairs using the objects **cartopol~** and **poltocar~**:



Cartesian to polar conversion

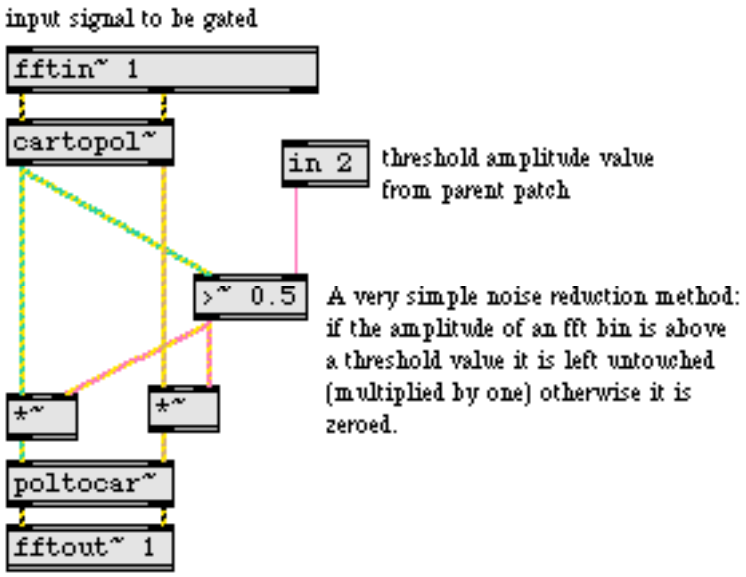
Technical detail: The amplitude values output by the left outlet of **cartopol~** depend on the amplitude of the signal you send to the **pfft~** object. Due to the way **fftin~** and **fftout~** automatically scale their window functions (in order to maintain the same output amplitude after overlap-adding), the maximum amplitude value for a constant signal of 1.0 will be

(FFT size / (sqrt(sum of points in the window/hop size))

So, when using a 512-point FFT with a square window with an overlap of 2, the maximum possible amplitude value will be roughly 362, with 4-overlap it will be 256. When using a hanning or hamming window and 2 overlap, it will be approximately 325 or 341, and with 4-overlap, it will be 230 or 241, respectively. Generally, however, the peak amplitudes in a spectral frame will most likely be only one-fourth to half this high for non-periodic or semi-periodic "real-world" sounds normalized between -1.0 and 1.0.

The phase values output by the right outlet of **cartopol~** will always be between -π and π.

You can use this information to create signal processing routines based on amplitude/phase data. A spectral noise gate would look something like this:

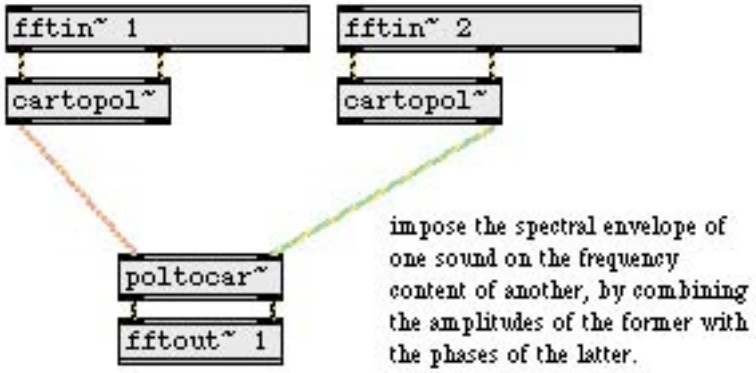


A spectral noise gate

By comparing the amplitude output of **cartopol~** with the threshold signal sent into inlet 2 of the **pfft~**, each bin is either passed or zeroed by the ***~** objects. This way only frequency bins that exceed a certain amplitude are retained in the resynthesis (For information on amplitude values inside a spectral subpatch, see the Technical note above.).

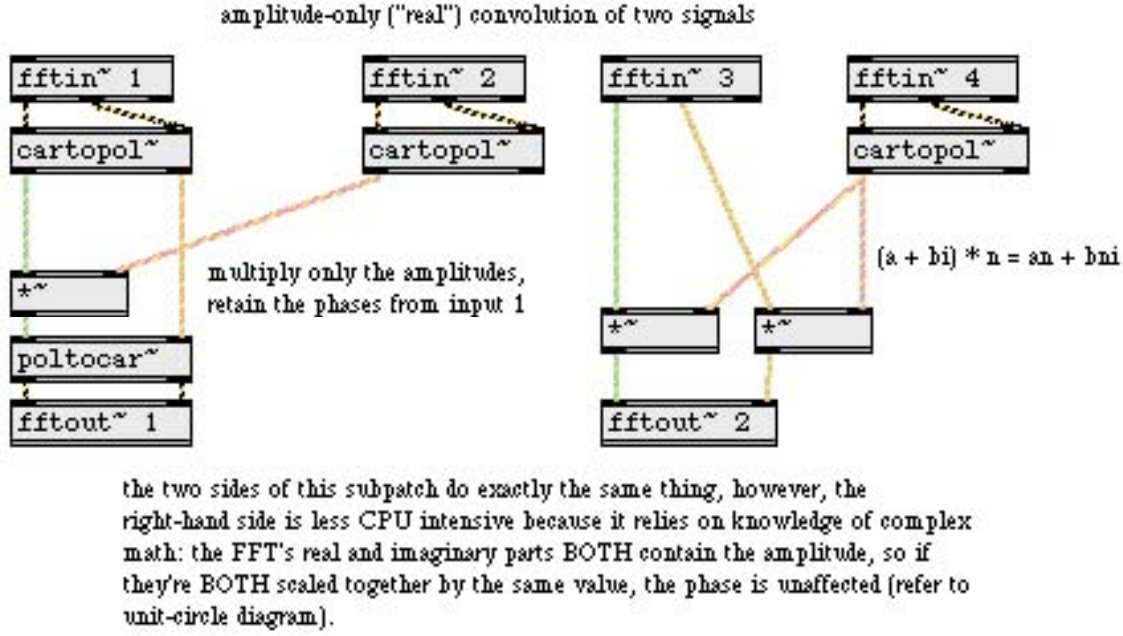
Convolution and cross-synthesis effects commonly use amplitude and phase data for their processing. One of the most basic cross-synthesis effects we could make would use the amplitude spectrum of one sound with the phase spectrum of another. Since the phase spectrum is related to information about the sound's frequency content, this kind of cross synthesis can give us the harmonic content of one sound being "played" by the spectral envelope of another sound. Naturally, the success of this type of effect depends heavily on the choice of the two sounds used.

Here is an example of a spectral subpatch which makes use of **cartopol~** and **poltocar~** to perform this type of cross-synthesis:



Simple cross-synthesis

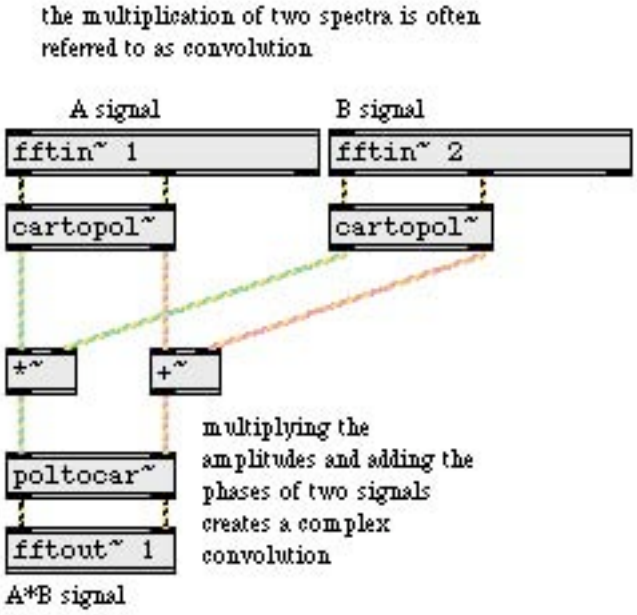
The following subpatch example shows two ways of convolving the amplitude of one input with the amplitude of another:



Amplitude-only convolution

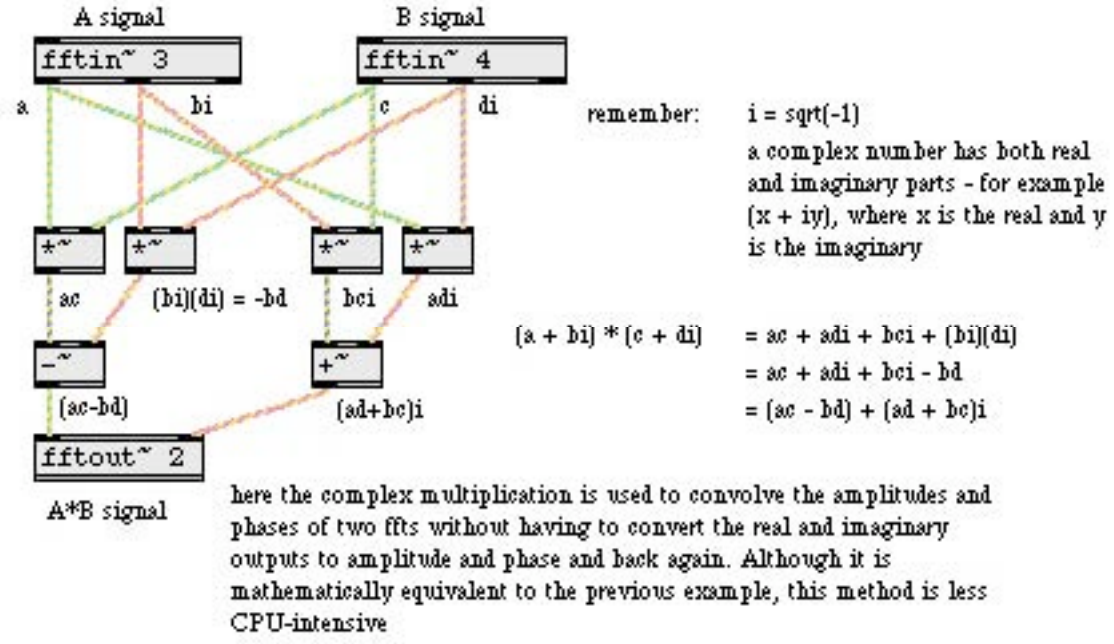
You can readily see on the left-hand side of this subpatch that the amplitude values of the input signals are multiplied together. This reinforces amplitudes which are prominent in both sounds while attenuating those which are not. The phase response of the first signal is unaffected by complex- real multiplication; the phase response of the second signal input is ignored. You will also notice that the right-hand side of the subpatch is mathematically equivalent to the left, even though it uses only one **cartopol~** object.

Toward the beginning of this tutorial, we saw an example of the multiplication of two real/imaginary signals to perform a convolution. That example was kept simple for the purposes of explanation but was, in fact, incorrect. If you wondered what a "correct" multiplication of two complex numbers would entail, heres one way to do it:



The correct method for doing complex convolution

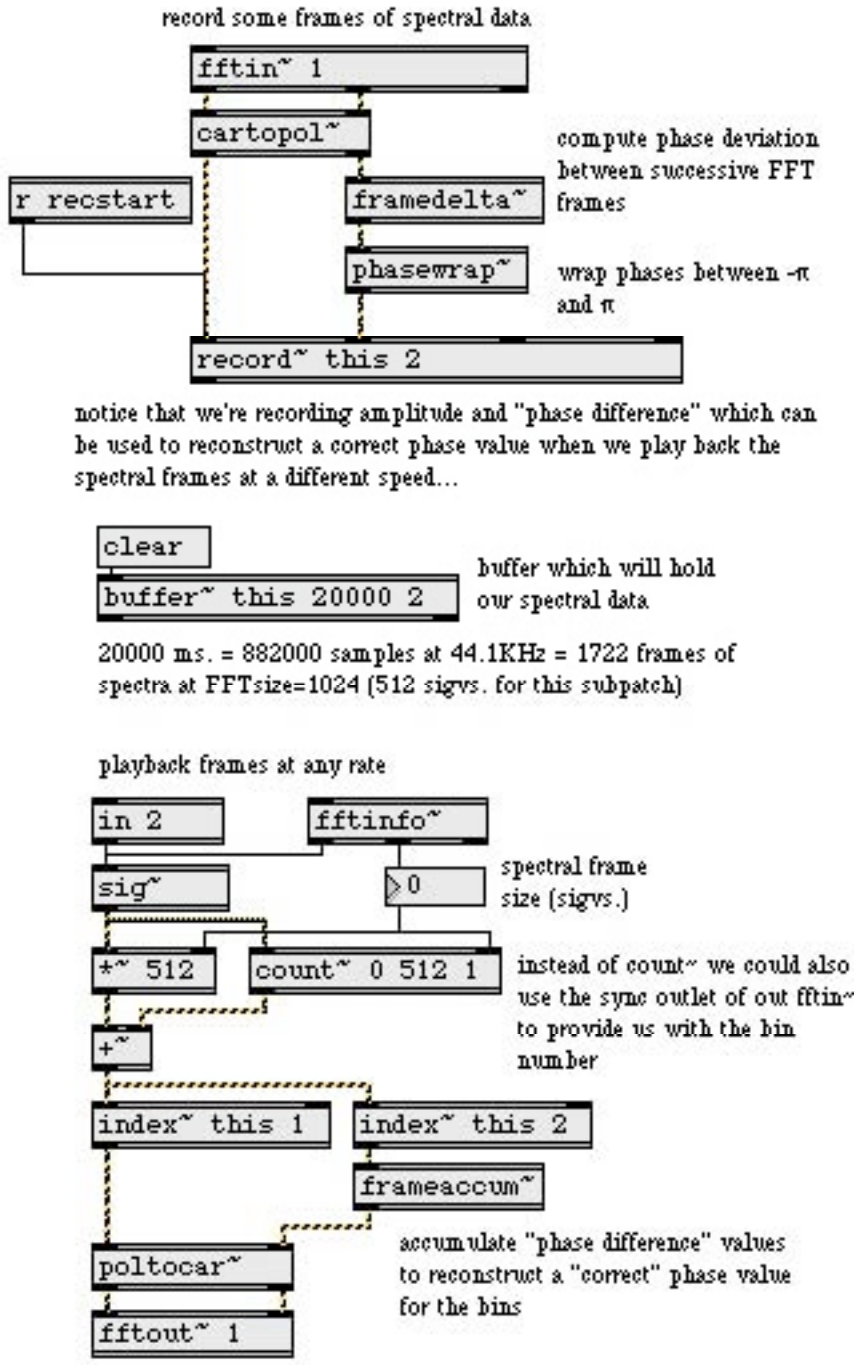
Here's a second and somewhat more clever approach to the same goal:



A correct and clever way of doing complex convolution

Subpatchers created for use with **pfft~** can use the full range of MSP objects, including objects that access data stored in a **buffer~** object. (Although some objects which were designed to deal with timing issues may not always behave as initially expected when used inside a **pfft~**.)

The following example records spectral analysis data into two channels of a stereo **buffer~** and then allows you to resynthesize the recording at a different speed.



Recording and playback in a pfft~ subpatch

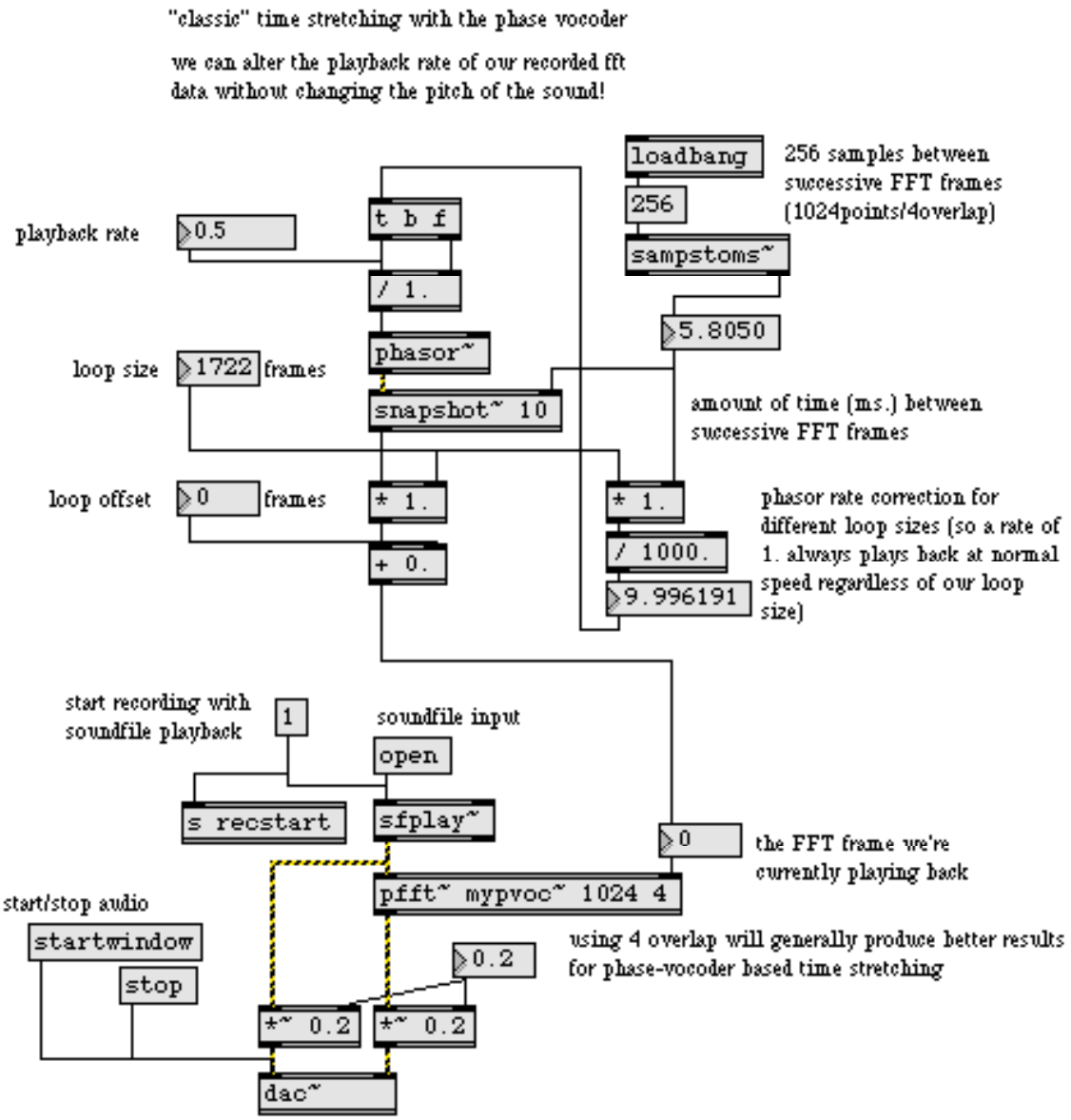
The example subpatcher records spectral data into a **buffer~**, and the second reads data from that **buffer~**. In the recording portion of the subpatch you will notice that we don't just record the amplitude and phase as output from **cartopol~**, but instead use the **framedelta~** object to compute the phase difference (sometimes referred to as the phase deviation, or phase derivative). The phase difference is quite simply the difference in phase between equivalent bin locations in successive FFT frames. The output of **framedelta~** is then fed into a **phasewrap~** object to ensure that the data is properly constrained between -and Messages can be sent to the **record~** object from the parent patch via the **send** object in order to start and stop recording and turn on looping.

In the playback part of the subpatch we use a non-signal inlet to specify the frame number for the resynthesis. This number is multiplied by the spectral frame size and added to the output of a **count~** object which counts from 0 to the spectral frame size minus 1 in order to be able to recall each frequency bin in the given frame successively using **index~** to read both channels of our **buffer~**. (We could also have used the sync outlet of the **fftin~** object in place of **count~**, but are using the current method for the sake of visually separating the recording and playback parts of our subpatch, as well as to give an example of how to make use of **count~** in the context of a spectral subpatch.) You'll notice that we reconstruct the phase using the **frameaccum~** object, which accumulates a "running phase" value by performing the inverse of **framedelta~**. We need to do this because we might not be reading the analysis frames successively at the original rate in which they were recorded. The signals are then converted back into real and imaginary values for **fftout~** by the **poltocar~** object.

This is a simple example of what is known as a *phase vocoder*. Phase vocoders allow you to time- stretch and compress signals independently of their pitch by manipulating FFT data rather than time-domain segments. If you think of each frame of an FFT analysis as a single frame in a film, you can easily see how moving through the individual frames at different rates can change the apparent speed at which things happen. This is more or less what a phase vocoder does.

Note that because **pfft~** does window overlapping, the amount of data that can be stored in the **buffer~** is dependent on the settings of the **pfft~** object. This can make setting the buffer size correctly a rather tricky matter, especially since the spectral frame size (i.e. the signal vector size) inside a **pfft~** is half the FFT size indicated as its second argument, and because the spectral subpatch is processing samples at a different rate to its parent patch! If we create a stereo **buffer~** with 1000 milliseconds of sample memory, we will have 44100 samples available for our analysis data. If our FFT size is 1024 then each spectral frame will take up 512 samples of our buffer's memory, which amounts to 86 frames of analysis data(44100 / 512 = 86.13). Those 86 frames do not represent one second of sound, however! If we are using 4-times overlap, we are processing one spectral frame every 256 samples, so 86 frames means roughly 22050 samples, or a half second's worth of time with respect to the parent patch. As you can see this all can get rather complicated...

Let's take a look at the parent patch for the above phase vocoder subpatch (called **mypvoc~**):



Wrapper for mypvoc

Notice that we're using a **phasor~** object with a **snapshot~** object in order to generate a ramp specifying the read location inside our subpatch. We could also use a **line** object, or even a slider, if we wanted to "scrub" our analysis frames by hand. Our main patch allows us to change the playback rate for a loop of our analysis data. We can also specify the loop size and an offset into our collection of analysis frames in order to loop a given section of analysis data at a given playback rate. You'll notice that changing the playback rate does

not affect the pitch of the sound, only the speed. You may also notice that at very slow playback rates, certain parts of your sound (usually note attacks, consonants in speech or other percussive sounds) become rather "smeared" and gain an artificial sound quality.

Summary

Using **pfft~** to perform spectral-domain signal processing is generally easier and visually clearer than using the traditional **fft~** and **ifft~** objects, and lets you design patches that can be used at varying FFT sizes and overlaps. There are myriad applications of **pfft~** for musical signal processing, including filtering, cross synthesis and time stretching.

See Also

adstatus	Access audio driver output channels
cartopol~	Signal Cartesian to Polar coordinate conversion
dspstate~	Report current DSP setting
fftin~	Input for a patcher loaded by **pfft~**
fftout~	Output for a patcher loaded by **pfft~**
framedelta~	Compute phase deviation between successive FFT frames
pfft~	Spectral processing manager for patchers
phasewrap~	Wrap a signal between -π and π
poltocar~	Signal Polar to Cartesian coordinate conversion

Effects achieved with delayed signals

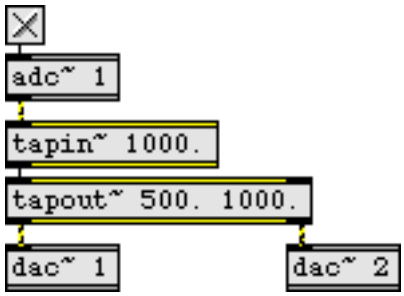
One of the most basic yet versatile techniques of audio processing is to delay a signal and mix the delayed version with the original signal. The delay time can range from a few milliseconds to several seconds, limited only by the amount of RAM you have available to store the delayed signal.

When the delay time is just a few milliseconds, the original and delayed signals interfere and create a subtle filtering effect but not a discrete echo. When the delay time is about 100 ms we hear a "slapback" echo effect in which the delayed copy follows closely behind the original. With longer delay times, we hear the two signals as discrete events, as if the delayed version were reflecting off a distant mountain.

This tutorial patch delays each channel of a stereo signal independently, and allows you to adjust the delay times and the balance between direct signal and delayed signal.

Creating a delay line: tapin~ and tapout~

The MSP object **tapin~** is a buffer that is continuously updated so that it always stores the most recently received signal. The amount of signal it stores is determined by a typed-in argument. For example, a **tapin~** object with a typed-in argument of 1000 stores the most recent one second of signal received in its inlet.



A 1-second delay buffer tapped 500 and 1000 ms in the past

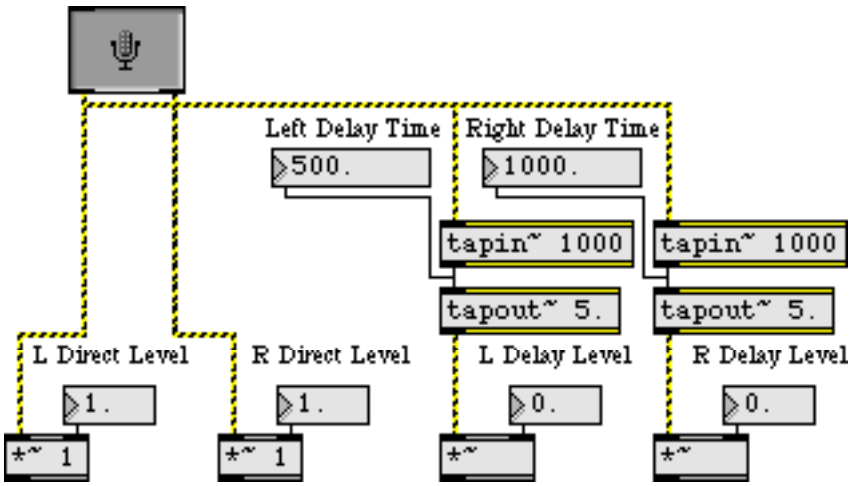
The only object to which the outlet of **tapin~** should be connected is a **tapout~** object. This connection links the **tapout~** object to the buffer stored by **tapin~**. The **tapout~** object "taps into" the delayed signal at certain points in the past. In the above example, **tapout~** gets the signal from **tapin~** that occurred 500 ms ago and sends it out the left outlet; it also gets the signal delayed by 1000 ms and sends that out its right outlet. It should be obvious that **tapout~** can't get signal delayed beyond the length of time stored in **tapin~**.

A patch for mixing original and delayed signals

The tutorial patch sends the sound coming into the computer to two places: directly to the output of the computer and to a **tapin~**-**tapout~** delay pair. You can control how much signal you hear from each place for each of the stereo channels, mixing original and delayed signal in whatever proportion you want.

- Turn audio on and send some sound in the input jacks of your computer. Set the **number box** marked "Output Level" to a comfortable listening level. Set the "Left Delay Time" **number box** to 500 and the "Right Delay Time" to 1000.

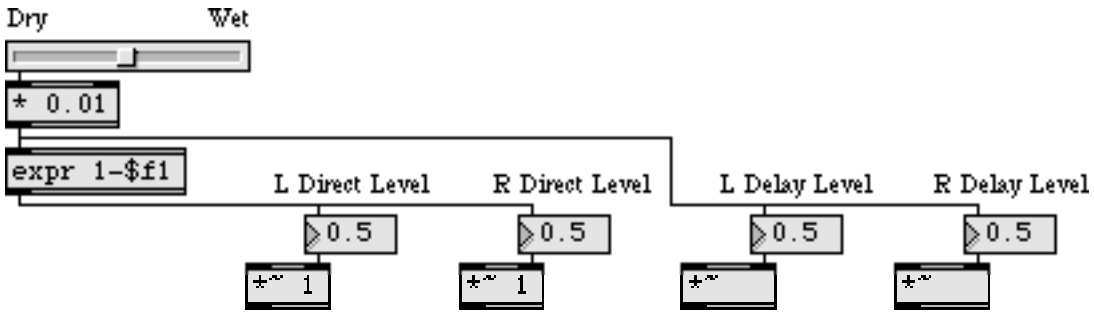
At this point you don't hear any delayed signal because the "Direct Level" for each channel is set at 1 and the "Delay Level" for each channel is set at 0. The signal is being delayed, but you simply don't hear it because its amplitude is scaled to 0.



Direct signal is on full; delayed signal is turned down to 0

The **hslider** in the left part of the Patcher window serves as a balance fader between a "Dry" (all direct) output signal and a "Wet" (fully processed) output signal.

- Drag the **hslider** to the halfway point so that both the direct and delayed signal amplitudes are at 0.5. You hear the original signal in both channels, mixed with a half-second delay in the left channel and a one-second delay in the right channel.



Equal balance between direct signal and delayed signal

- You can try a variety of different delay time combinations and wet-dry levels. Try very short delay times for subtle comb filtering effects. Try creating rhythms with the two delay times (with, for example, delay times of 375 and 500 ms).

Changing the parameters while the sound is playing can result in clicks in the sound because this patch does not protect against discontinuities. You could create a version of this patch that avoids this problem by interpolating between parameter values with **line~** or **number~** objects.

In future tutorial chapters, you will see how to create delay feedback, how to use continuously variable delay times for flanging and pitch effects, and other ways of altering sound using delays, filters, and other processing techniques.

Summary

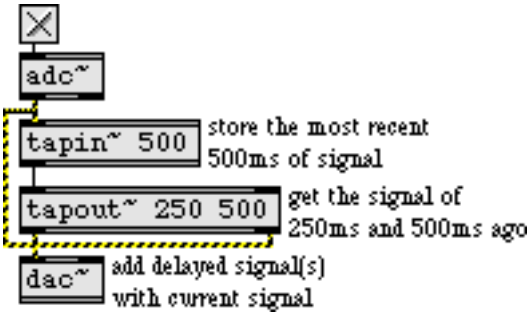
The **tapin~** object is a continuously updated buffer which always stores the most recently received signal. Any connected **tapout~** object can use the signal stored in **tapin~**, and access the signal from any time in the past (up to the limits of the **tapin~** object's storage). A signal delayed with **tapin~** and **tapout~** can be mixed with the undelayed signal to create discrete echoes, early reflections, or comb filtering effects.

See Also

tapin~	Input to a delay line
tapout~	Output from a delay line

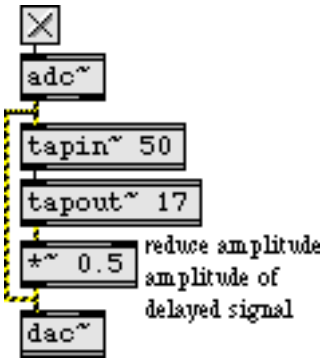
Delay emulates reflection

You can delay a signal for a specific amount of time using the **tapin~** and **tapout~** objects. The **tapin~** object is a continually updated buffer that stores the most recent signal it has received, and **tapout~** accesses that buffer at one or more specific points in the past.



Delaying a signal with tapin~ and tapout~

Combining a sound with a delayed version of itself is a simple way of emulating a sound wave reflecting off of a wall before reaching our ears; we hear the direct sound followed closely by the reflected sound. In the real world some of the sound energy is actually absorbed by the reflecting wall, and we can emulate that fact by reducing the amplitude of the delayed sound, as shown in the following example.

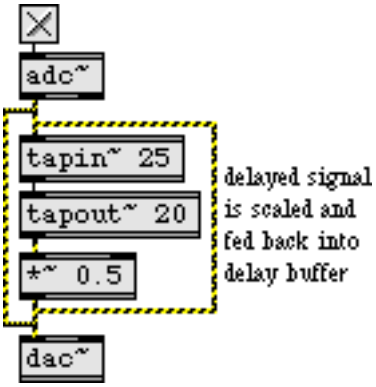


Scaling the amplitude of a delayed signal, to emulate absorption

Technical detail: Different materials absorb sound to varying degrees, and most materials absorb sound in a way that is frequency-dependent. In general, high frequencies get absorbed more than low frequencies. That fact is being ignored here.

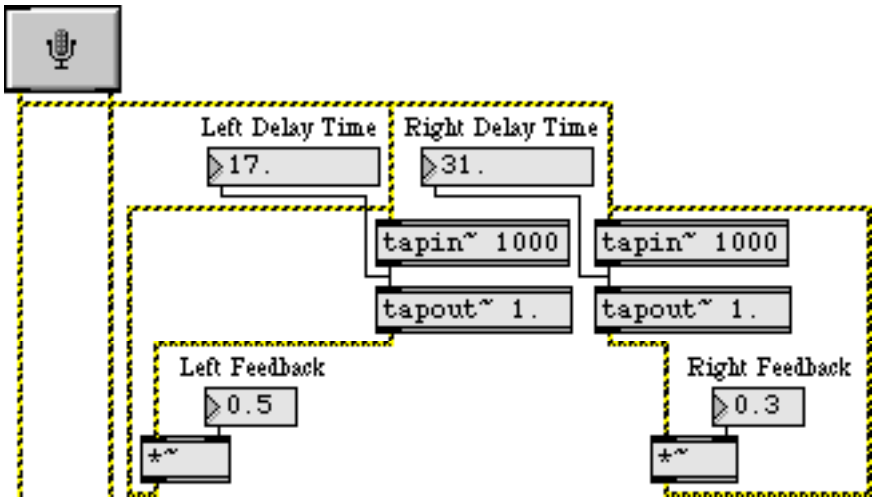
Delaying the delayed signal

Also, in the real world there's usually more than one surface that reflects sound. In a room, for example, sound reflects off of the walls, ceiling, floor, and objects in the room in myriad ways, and the reflections are in turn reflected off of other surfaces. One simple way to model this "reflection of reflections" is to feed the delayed signal back into the delay line (after first "absorbing" some of it).



Delay with feedback

A single feedback delay line like the one above is too simplistic to sound much like any real world acoustical situation, but it can generate a number of interesting effects. Stereo delay with feedback is implemented in the example patch for this tutorial. Each channel of audio input is delayed, scaled, and fed back into the delay line.



Stereo delay with individual delay times and feedback amounts

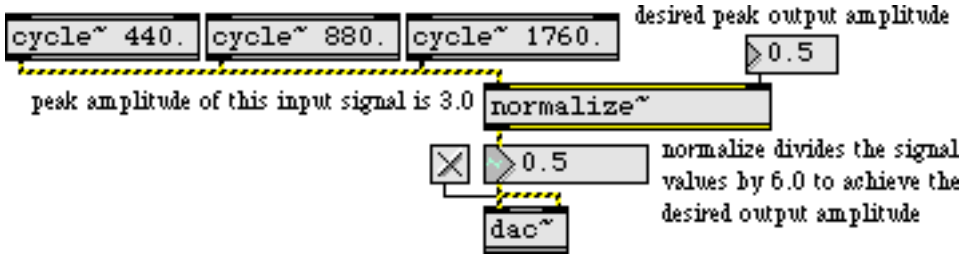
- Set the **number box** marked "Output Level" to 1., and move the **hslider** to its middle position so that the "Direct Level" and "Delay Level" **number box** objects read 0.5. Turn audio on, and send some sound into the audio input of the computer. Experiment with different delay times and feedback amounts. For example, you can use the settings shown above to achieve a blurring effect. Increase the feedback amounts for a greater resonant ringing at the rate of feedback (1000 divided by the delay time). Increase the delay times to achieve discrete echoes. You can vary the Dry/Wet mix with the **hslider**.

Note that any time you feed audio signal back into a system, you have a potential for overloading the system. That's why it's important to scale the signal by some factor less than 1.0 (with the ***~** objects and the "Feedback" **number box** objects) before feeding it back into the delay line. Otherwise the delayed sound will continue indefinitely and even increase as it is added to the new incoming audio.

Controlling amplitude: normalize~

Since this patch contains user-variable level settings (notably the feedback levels) and since we don't know what sound will be coming into the patch, we can't really predict how we will need to scale the final output level. If we had used a ***~** object just before the **ezdac~** to scale the output amplitude, we could set the output level, but if we later increase the feedback levels, the output amplitude could become excessive. The **normalize~** object is good for handling such unpredictable situations.

The **normalize~** object allows you to specify a peak (maximum) amplitude that you want sent out its outlet. It looks at the peak amplitude of its input, and calculates the factor by which it must scale the signal in order to keep the peak amplitude at the specified maximum. So, with **normalize~** the peak amplitude of the output will never exceed the specified maximum.



*normalize~ sends out the current input * peak output / peak input*

One potential drawback of **normalize~** is that a single loud peak in the input signal can cause **normalize~** to scale the entire signal way down, even if the rest of the input signal is

very soft. You can give **normalize~** a new peak input value to use, by sending a number or a reset message in the left inlet.

- Turn audio off and close the Patcher window before proceeding to the next chapter.

Summary

One way to make multiple delayed versions of a signal is to feed the output of **tapout~** back into the input of **tapin~**, in addition to sending it to the DAC. Because the fed back delayed signal will be added to the current incoming signal at the inlet of **tapin~**, it's a good idea to reduce the output of **tapout~** before feeding it back to **tapin~**.

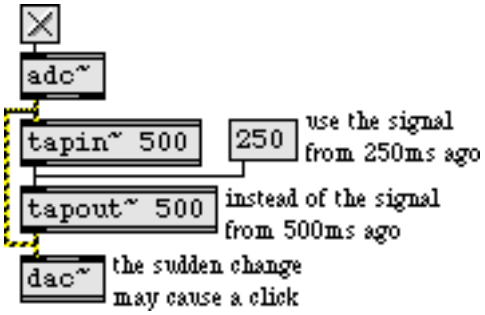
In a patch involving addition of signals with varying amplitudes, it's often difficult to predict the amplitude of the summed signal that will go to the DAC. One way to control the amplitude of a signal is with **normalize~**, which uses the peak amplitude of an incoming signal to calculate how much it should reduce the amplitude before sending the signal out.

See Also

normalize~	Scale on the basis of maximum amplitude
tapin~	Input to a delay line
tapout~	Output from a delay line

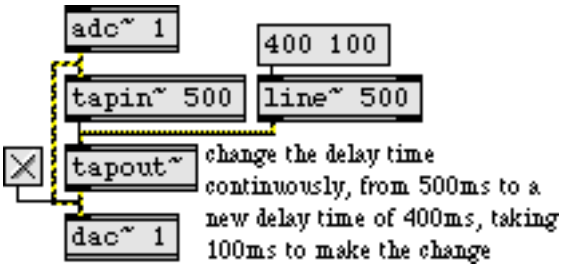
Variable delay time

So far, we have been delaying signals for a fixed amount of time using **tapin~** and **tapout~**. You can change the delay time of any tap in the **tapout~** object by sending a new number in the proper inlet; however, this will cause a discontinuity in the output signal at the instant when then new delay time is received, because **tapout~** suddenly begins tapping a new location in the **tapin~** buffer.



Changing the delay time creates a discontinuity in the output signal

On the other hand, it's possible to provide a new delay time to **tapout~** using a continuous signal instead of a discrete Max message. We can use the **line~** object to make a continuous transition between two delay times (just as we did to make continuous changes in amplitude in *Tutorial 2*).



Providing delay time in the form of a signal

Technical detail: Note that when the delay time is being changed by a continuous signal, **tapout~** has to interpolate between the old delay time and the new delay time for every sample of output. Therefore, a **tapout~** object has to do much more computation whenever a signal is connected to one of its inlets.

While this avoids the click that could be caused by a sudden discontinuity, it does mean that the pitch of the output signal will change while the delay time is being changed, emulating the *Doppler effect*

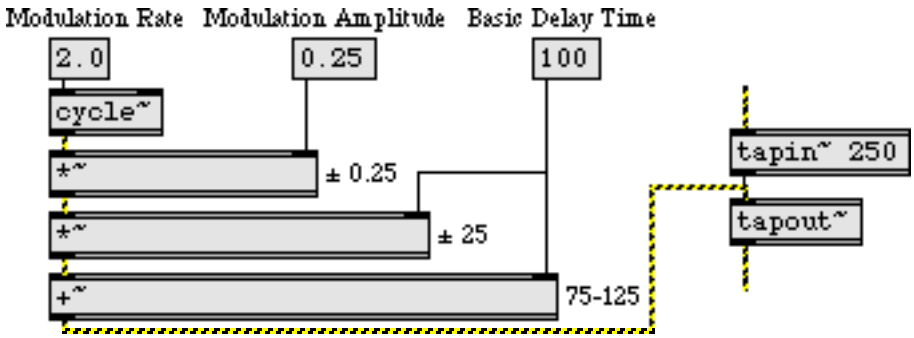
> **Technical detail:** The Doppler effect occurs when a sound source is moving toward or away from the listener. The moving sound source is, to some extent, outrunning the wavefronts of the sound it is producing. That changes the frequency at which the listener receives the wavefronts, thus changing the perceived pitch. If the sound source is moving toward the listener, wavefronts arrive at the listener with a slightly greater frequency than they are actually being produced by the source. Conversely, if the sound source is moving away from the listener, the wavefronts arrive at the listener slightly less frequently than they are actually being produced. The classic case of Doppler effect is the sound of an ambulance siren. As the ambulance passes you, it changes from moving toward you (producing an increase in received frequency) to moving away from you (producing a decrease in received frequency). You perceive this as a swift drop in the perceived pitch of the siren.
>
> A delayed signal emulates a reflection of the sound wave. As the delay time decreases, it is as if the (virtual) reflecting wall were moving toward you. The source of the delayed sound (the reflecting wall) is "moving toward you", causing an increase in the received frequency of the sound. As the delay time increases, the reverse is true; the source of the delayed sound is effectively moving away from you. That is why, during the time when the delay time is actually changing, the perceived pitch of the output sound changes.

A delayed signal emulates a reflection of the sound wave. As the delay time decreases, it is as if the (virtual) reflecting wall were moving toward you. The source of the delayed sound (the reflecting wall) is "moving toward you", causing an increase in the received frequency of the sound. As the delay time increases, the reverse is true; the source of the delayed sound is effectively moving away from you. That is why, during the time when the delay time is actually changing, the perceived pitch of the output sound changes.

A pitch shift due to Doppler effect is usually less disruptive than a click that's caused by discontinuity of amplitude. More importantly, the pitch variance that results from continuously varying the delay time can be used to create some interesting effects.

Flanging: Modulating the delay time

Since the delay time can be provided by any signal, one possibility is to use a time-varying signal like a low-frequency cosine wave to modulate the delay time. In the example below, a **cycle~** object is used to vary the delay time.



Modulating the delay time with a low-frequency oscillator

The output of **cycle~** is multiplied by 0.25 to scale its amplitude. That signal is multiplied by the basic delay time of 100 ms, to create a signal with an amplitude ±25. When that signal is added to the basic delay time, the result is a signal that varies sinusoidally around the basic delay time of 100, going as low as 75 and as high as 125. This is used to express the delay time in milliseconds to the **tapout~** object.

When a signal with a time-varying delay (especially a very short delay) is added together with the original undelayed signal, the result is a continually varying comb filter effect known as *flanging*. Flanging can create both subtle and extreme effects, depending on the rate and depth of the modulation.

Stereo flange with feedback

This tutorial patch is very similar to that of the preceding chapter. The primary difference here is that the delay times of the two channels are being modulated by a cosine wave, as was described on the previous page. This patch gives you the opportunity to try a wide variety of flanging effects, just by modifying the different parameters: the wet/dry mix between delayed and undelayed signal, the left and right channel delay times, the rate and depth of the delay time modulation, and the amount of delayed signal that is fed back into the delay line of each channel.

- Send some sound into the audio input of the computer, and click on the buttons of the **preset** object to hear different effects. Using the example settings as starting points, experiment with different values for the various parameters. Notice that the modulation depth can also be controlled by the mod wheel of your synth,

demonstrating how MIDI can be used for realtime control of audio processing parameters.

The different examples stored in the **preset** object are characterized below.

1. Simple thru of the audio input to the audio output. This is just to allow you to test the input and output.

2. The input signal is combined equally with delayed versions of itself, using short (mutually prime) delay times for each channel. The rate of modulation is set for 0.2 Hz (one sinusoid every 5 seconds), but the depth of modulation is initially 0. Use the mod wheel of your synth (or drag on the "Mod Wheel" **number box**) to introduce some slow flanging.

3. The same as before, but now the modulation rate is 6 Hz. The modulation depth is set very low for a subtle vibrato effect, but you can increase it to obtain a decidedly un-subtle wide vibrato.

4. A faster vibrato, with greater depth, and with the delayed signal fed back into the delay line, creates a complex warbling flange effect.

5. The right channel is delayed a short time for a flange effect and the left channel is delayed a longer time for an echo effect. Both delay times change sinusoidally over a two second period, and each delayed signal is fed back into its own delay line (causing a ringing resonance in the right channel and repeated echoes in the left channel).

6. Both delay times are set long with considerable feedback to create repeated echoes. The rate (and pitch) of the echoes is changed up and down by a very slow modulating frequency—one cycle every 10 seconds.

7. A similar effect, but modulated sinusoidally every 2 seconds.

8. Similar to example 5, but with flanging occurring at an audio rate of 55 Hz, and no original sound in the mix. The source sound is completely distorted, but the modulation rate gives the distortion its fundamental frequency.

Summary

You can provide a continuously varying delay time to **tapout~** by sending a signal in its inlet. As the delay time varies, the pitch of the delayed sound shifts oppositely. You can use a repeating low frequency wave to modulate the delay time, achieving either subtle or extreme pitch-variation effects. When a sound with a varying delay time is mixed with the original undelayed sound, the result is a variable comb filtering effect known as *flanging*.

The depth (strength) of the flanging effect depends primarily on the amplitude of the signal that is modulating the delay time.

See Also

noise~	White noise generator
rand~	Band-limited random signal
tapin~	Input to a delay line
tapout~	Output from a delay line

The chorus effect

Why does a chorus of singers sound different from a single singer? No matter how well trained a group of singers may be, they don't sing identically. They're not all singing precisely the same pitch in impeccable unison, so the random, unpredictable phase cancellations that occur as a result of these slight pitch differences are thought to be the source of the *chorus effect*.

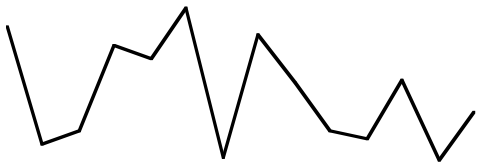
We've already seen in the preceding chapter how slight pitch shifts can be introduced by varying the delay time of a signal. When we mix this signal with its original undelayed version, we create interference between the two signals, resulting in a constantly varying filtering effect known as flanging. A less predictable effect called chorusing can be achieved by substituting a random fluctuation of the delay time in place of the sinusoidal fluctuation we used for flanging.

Low-frequency noise: rand~

The **noise~** object (introduced in Tutorial 3) produces a signal in which every sample has a randomly chosen value between -1 and 1; the result is *white noise*, with roughly equal energy at every frequency. This white noise is not an appropriate signal to use for modulating the delay time, though, because it would randomly change the delay time so fast (every sample, in fact) that it would just sound like added noise. What we really want is a modulating signal that changes more gradually, but still unpredictably.

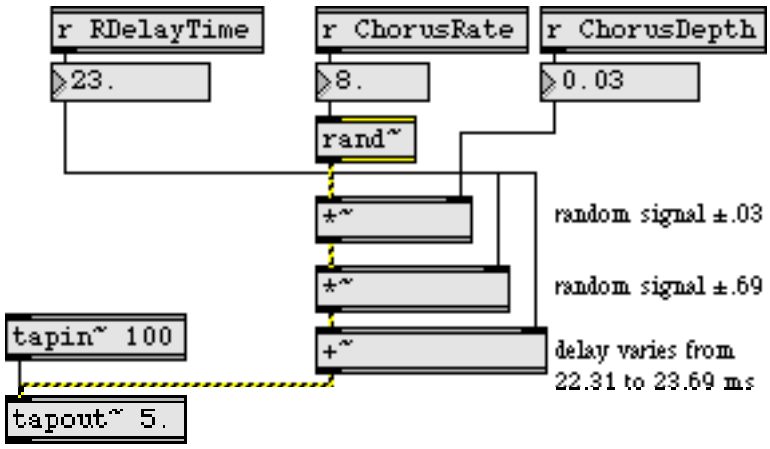
The **rand~** object chooses random numbers between -1 and 1, but does so less frequently than every sample. You can specify the frequency at which it chooses a new random value. In between those randomly chosen samples, **rand~** interpolates linearly from one value to the next to produce an unpredictable but more contiguous signal.

Random values chosen every sample



Choosing values less frequently

The output of **rand~** is therefore still noise, but its spectral energy is concentrated most strongly in the frequency region below the frequency at which it chooses its random numbers. This "low-frequency noise" is a suitable signal to use to modulate the delay time for a chorusing effect.

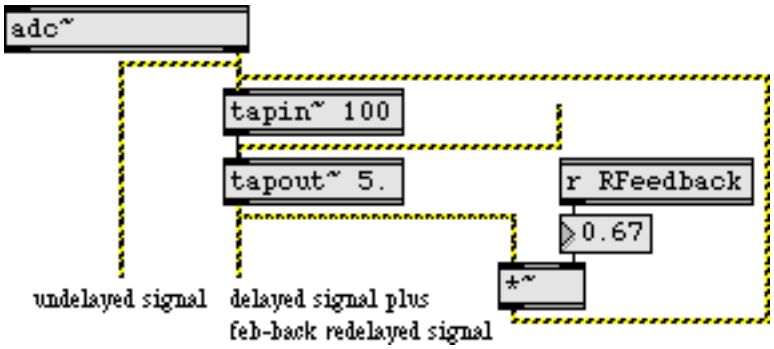


Unpredictable variations using rand~

The tutorial patch for this chapter is substantially similar to the flanging patch in the previous chapter. The main difference between the two signal networks is that the **cycle~** object for flanging has been replaced by a **rand~** object for chorusing. The **scope~** object in this patch is just for visualizing the modulating effect of the **rand~** object.

Multiple delays for improved chorus effect

We can improve this chorus effect by increasing the number of slightly different signals we combine. One way to do this —as we have done in this patch— is to feed the randomly delayed signal back into the delay line, where it's combined with new incoming signal. The output of **tapout~** will thus be a combination of the new variably delayed (and variably pitch shifted) signal and the previously (but differently) delayed/shifted signal.



Increasing the number of "voices" using feedback to the delay line

The balance between these signals is determined by the settings for "LFeedback" and "RFeedback", and the combination of these signals and the undelayed signal is balanced by the "DryWetMix" value. To obtain the fullest "choir" with this patch, we chose delay times (17 ms and 23 ms) and a modulation rate (8 Hz, a period of 125 ms) that are all mutually prime numbers, so that they are never in sync with each other.

> **Technical detail:** One can obtain an even richer chorus effect by increasing the number of different delay taps in **tapout~**, and applying a different random modulation to each delay time.

- Click on the **toggle** to turn audio on. Send some sound into the audio input of the computer to hear the chorusing effect. Experiment by changing the values for the different parameters. For a radically different effect, try some extreme values (longer delay times, more feedback, much greater chorus depth, very slow and very fast modulation rates, etc.).

Summary

The *chorus effect* is achieved by combining multiple copies of a sound—each one delayed and pitch shifted slightly differently—with the original undelayed sound. This can be done by continual slight random modulation of the delay time of two or more different delay taps. The **rand~** object sends out a signal of linear interpolation between random

values (in the range -1 to 1) chosen at a specified rate; this signal is appropriate for the type of modulation required for chorusing. Feeding the delayed signal back into the delay line increases the complexity and richness of the chorus effect. As with most processing effects, interesting results can also be obtained by choosing "outrageous" extreme values for the different parameters of the signal network.

See Also

rand~	Band-limited random signal
tapout~	Output from a delay line

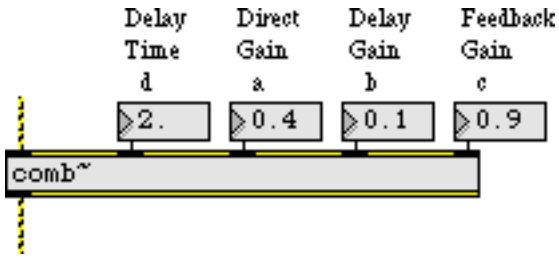
Comb filter: comb~

The minimum delay time that can be used for feedback into a delay line using **tapin~** and **tapout~** is determined by the signal vector size. However, many interesting filtering formulae require feedback using delay times of only a sample or two. Such filtering processes have to be programmed within a single MSP object.

An example of such an object is **comb~**, which implements a formula for *comb filtering*. Generally speaking, an audio filter is a frequency-dependent amplifier; it boosts the amplitude of some frequency components of a signal while reducing other frequencies. A comb filter accentuates and attenuates the input signal at regularly spaced frequency intervals—that is, at integer multiples of some fundamental frequency.

> **Technical detail:** The fundamental frequency of a comb filter is the inverse of the delay time. For example, if the delay time is 2 milliseconds ($^{1}/_{500}$ of a second), the accentuation occurs at intervals of 500 Hz (500, 1000, 1500, etc.), and the attenuation occurs between those frequencies. The extremity of the filtering effect depends on the factor (between 0 and 1) by which the feedback is scaled. As the scaling factor approaches 1, the accentuation and attenuation become more extreme. This causes the sonic effect of resonance (a "ringing" sound) at the harmonics of the fundamental frequency.

The **comb~** object sends out a signal that is a combination of a) the input signal, b) the input signal it received a certain time ago, and c) the output signal it sent that same amount of time ago (which would have included prior delays). In the inlets of **comb~** we can specify the desired amount of each of these three (*a*, *b*, and *c*), as well as the delay time (we'll call it *d*).



You can adjust all the parameters of the comb filter

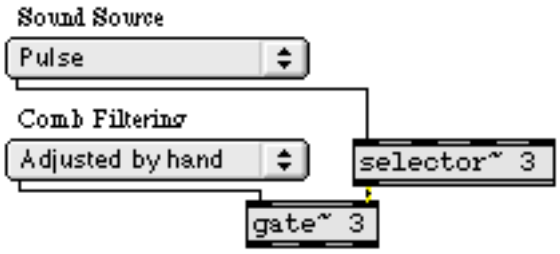
Technical detail: At any given moment in time (we'll call that moment t), **comb~** uses the value of the input signal (x_t), to calculate the output y_t in the following manner.

$$y_t = ax_t + bx_{(t-d)} + cy_{(t-d)}$$

The fundamental frequency of the comb filter depends on the delay time, and the intensity of the filtering depends on the other three parameters. Note that the scaling factor for the feedback (the right inlet) should usually not exceed 1, since that would cause the output of the filter to increase steadily as a greater and greater signal is fed back.

Trying out the comb filter

The tutorial patch enables you to try out the comb filter by applying it to different sounds. The patch provides you with three possible sound sources for filtering—the audio input of your computer, a band-limited pulse wave, or white noise—and three filtering options—unfiltered, comb filter with parameters adjusted manually, or comb filter with parameters continuously modulated by other signals.



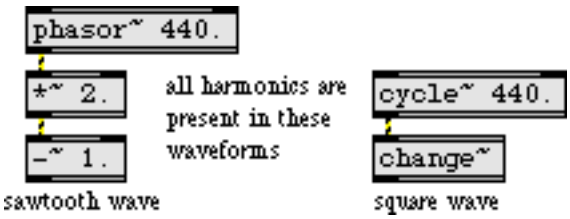
Choose a sound source and route it to the desired filtering using the pop-up menus

- Click on the buttons of the **preset** to try out some different combinations, with example parameter settings. Listen to the effect of the filter, then experiment by changing parameters yourself. You can use MIDI note messages from your synth to provide pitch and velocity (frequency and amplitude) information for the pulse wave, and you can use the mod wheel to change the delay time of the filter.

A comb filter has a characteristic harmonic resonance because of the equally spaced frequencies of its peaks and valleys of amplification. This trait is particularly effective when the comb is swept up and down in frequency, thus emphasizing different parts of the source sound. We can cause this frequency sweep simply by varying the delay time.

Band-limited pulse

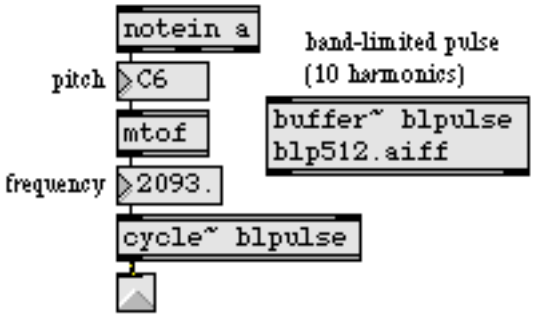
The effects of a filter are most noticeable when there are many different frequencies in the source sound, which can be altered by the filter. If we want to apply a comb filter to a pitched sound with a harmonic spectrum, it makes most sense to use a sound that has many partials such as a sawtooth wave or a square wave.



These mathematically ideal waves may be too "perfect" for use as computer sound waves

The problem with such mathematically derived waveforms, though, is that they may actually be too rich in high partials. They may have partials above the Nyquist rate that are sufficiently strong to cause inharmonic aliasing. (This issue is discussed in more detail in *Tutorial 5*.)

For this tutorial we're using a waveform called a *band-limited pulse*. A band-limited pulse has a harmonic spectrum with equal energy at all harmonics, but has a limited number of harmonics in order to prevent aliasing. The waveform used in this tutorial patch has ten harmonics of equal energy, so its highest frequency component has ten times the frequency of the fundamental. That means that we can use it to play fundamental frequencies up to 2,205 Hz if our sampling rate is 44,100 Hz. (Its highest harmonic would have a frequency of 22, 050 Hz, which is equal to the Nyquist rate.) Since the highest key of a 61-key MIDI keyboard plays a frequency of 2,093 Hz, this waveform will not cause aliasing if we use that as an upper limit.

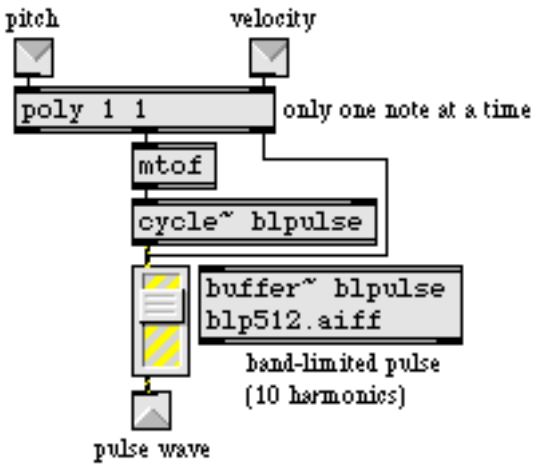


Playing a band-limited pulse wave with MIDI

Technical detail: In an idealized (optimally narrow) pulse wave, each cycle of the waveform would consist of a single sample with a value of 1, followed by all samples at 0. This would create a harmonic spectrum with all harmonics at equal amplitude, continuing upward infinitely. It's possible to make an MSP signal network that calculates—based on the fundamental frequency and the sampling rate—a band-limited pulse signal containing the maximum number of possible harmonics without foldover. In this case, though, we have chosen just to use a stored waveform containing ten partials.

Velocity-to-amplitude conversion: gain~

The subpatch **p** Pulse_Wave contains a simple but effective way to play a sound in MSP via MIDI. It uses a **poly** object to implement voice stealing, limiting the incoming MIDI notes to one note at a time. (It turns off the previous note by sending it out with a velocity of 0 before it plays the incoming note.) It then uses **mtof** to convert the MIDI note number to the correct frequency value for MSP, and it uses the MSP object **gain~** to scale the amplitude of the signal according to the MIDI velocity.



Converting MIDI pitch and velocity data to frequency and amplitude information for MSP

The **gain~** object takes both a signal and a number in its left inlet. The number is used as an amplitude factor by which to scale the signal before sending it out. One special feature of **gain~** (aside from its utility as a user interface object for scaling a signal) is that it can convert the incoming numbers from a linear progression to a logarithmic or exponential curve. This is very appropriate in this instance, since we want to convert the linear velocity range (0 to 127) into an exponential amplitude curve (0 to 1) that corresponds roughly to the way that we hear loudness. Each change of velocity by 10 corresponds to a change of amplitude by 6 dB. The other useful feature of **gain~** is that, rather than

changing amplitude abruptly when it receives a new number in its left inlet, it takes a few milliseconds to progress gradually to the new amplitude factor. The time it takes to make this progression can be specified by sending a time, in milliseconds, in the right inlet. In this patch, we simply use the default time of 20 ms.

- Choose one of the preset example settings, and choose "Pulse Wave" from the "Sound Source" pop-up menu. Play long notes with the MIDI keyboard. You can also obtain a continuous sound at any amplitude and frequency by sending numbers from the pitch and velocity **number box** objects (first velocity, then pitch) into the inlets of the **p** Pulse_Wave subpatch.

Varying parameters to the filter

As illustrated in this patch, it's usually best to change the parameters of a filter by using a gradually changing signal instead of making an abrupt change with single number. So parameter changes made to the "Adjusted By Hand" **comb~** object are sent first to a **line~** object for interpolation over a time of 25 ms.

The "Modulated" **comb~** object has its delay time varied at low frequency according to the shape of the band-limited pulse wave (just because it's a more interesting shape than a simple sinusoid). The modulation could actually be done by a varying signal of any shape. You can vary the rate of this modulation using the mod wheel of your synth (or just by dragging on the **number box**). The gain of the *x* and *y* delays (the two rightmost inlets) is modulated by a sine wave ranging between 0.01 and 0.99 (for the feedback gain) and a cosine wave ranging from 0.01 to 0.49 (for the feedforward gain). As the amplitude of one increases, the other decreases.

- Experimenting with different combinations of parameter values may give you ideas for other types of modulation you might want to design in your own patches.

Summary

The **comb~** object allows you to use very short feedback delay times to *comb filter* a signal. A comb filter creates frequency-dependent increases and decreases of amplitude in the signal that passes through it, at regularly spaced (i.e., harmonically related) frequency intervals. The frequency interval is determined by the inverse of the delay time. The comb filter is particularly effective when the delay time (and thus the frequency interval) changes over time, emphasizing different frequency regions in the filtered signal.

The user interface object **gain~** is useful for scaling the amplitude of a signal according to a specific logarithmic or exponential curve. Changes in amplitude caused by **gain~** take place gradually over a certain time (20 ms by default), so that there are no unwanted sudden discontinuities in the output signal.

See Also

comb~	Comb filter
gain~	Exponential scaling volume slider

The dsp Object—Controlling and Automating MSP

In order to provide low-level control over the MSP environment from within Max, a special object named dsp has been defined. This object is similar to the object **max** that can accept messages to change various options in the Max application environment. Sending a message to the dsp object is done by placing a semicolon in a message box, followed by dsp and then the message and arguments (if any). An example is shown below.

Turn the audio on or off without a dac~ or adc~ object

You need not connect the message box to anything, although you may want to connect something to the inlet of the message box to supply a message argument or trigger it from a **loadbang** object to configure MSP signal processing parameters when your patcher file is opened.

Here is a list of messages the dsp object understands:

Message	*Parameters*
; dsp start	Start Audio
; dsp stop	Stop Audio
; dsp set N	N = 1, Start Audio; N = 0, Stop Audio
; dsp status	Open DSP Status Window
; dsp open	Open DSP Status Window
; dsp sr N	N = New Sampling Rate in Hz
; dsp iovs N	N = New I/O Vector Size
; dsp sigvs N	N = New Signal Vector Size
; dsp debug N	N = 1, Internal debugging on; N = 0, Internal debugging off
; dsp takeover N	N = 1, Scheduler in Audio Interrupt On; N = 0, Scheduler in Audio Interrupt Off
; dsp wclose	Close DSP Status window
; dsp inremap X Y	Maps physical device input channel *Y* to logical input *X*
; dsp outremap X Y	Maps logical output *X* to physical device output channel *Y*

; dsp setdriver D S	If *D* is a number starting at 0, a new audio driver is chosen based on its index into the currently generated menu of driverrs created by the adstatus driver object. If *D* is a symbol, a new driver is selected by name (if *D* names a valid driver). The second argument *S* is optional and names the "subdriver." For instance, with ASIO drivers, ASIO is the name of the driver and PCI-324 is an example of a subdriver name.
; dsp timecode N	N = 1 or 0 to start/stop timecode reading by the audio driver (only supported currently by ASIO 2 drivers).
; dsp optimize N	N = 1 or 0 to turn Altivec optimization on/off
; dsp cpulimit N	Sets a utilization limit for the CPU, above this limit, MSP will not process audio vectors until the utilization comes back down, causing a click. N is a number between 0 and 100. If N is 0 or 100, there is no limit checking.

Certain audio drivers can be controlled with the ; dsp driver message. Refer to the Audio Input and Output section for more information on drivers that support this capability.

Index